UNIVERSITY OF CALIFORNIA
RIVERSIDE

Evasion Attacks on Network Intrusion Detection: Investigation, Automation, and Mitigation

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

Doctor of Philosophy

in

Computer Science

by

Zhongjie Wang

June 2021

The Dissertation of Zhongjie Wang is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

The pursuit of a PhD is a solitary journey towards the wilderness of the unknown. Oftentimes I feel lost and stressed in this prolonged process. However, I was lucky to have a bunch of amazing people and good friends around me.

First, I want to express my sincere gratitude to my outstanding PhD advisors, Prof. Zhiyun Qian and Prof. Srikanth Krishnamurthy, for their guidance and help. Prof. Krishnamurthy is always patient and supportive, generously offering his help when I'm in need, which gives me enormous strength and courage to carry on. Prof. Qian lights my way, guides my direction in research with his extraordinary talents, and sets an excellent example for me. His training helps me to acquire the skills needed as a PhD student. I have learned a lot from both of you, and will always be grateful.

I also want to thank my committee members, Prof. Chengyu Song and Prof. Heng Yin. They are both very knowledgeable in the area of program anlaysis, and provided lots of helpful discussions and comments on my dissertation. Prof. Song also offered tremendous help in my first and second projects.

Besides, I would like to thank all my co-authors for their contributions to my publications, which are included in this dissertation. Specifically, chapter 2 was published in ACM IMC 2017, and chapter 3 was published in NDSS 2020. The content of chapter 4 is in submission to CCS 2021. I want to give a special shout-out to Yue Cao, Shitong Zhu, Keyu Man, Pengxiong Zhu, and Yu Hao, for your great contributions.

What supports me to finish my PhD include not only the help in my research, but also the help in my life. I want to thank the good friends and labmates who make my PhD life amazing and colorful. They are Yue Cao, Pengxiong Zhu, Shitong Zhu, Keyu Man, Xiaochen Zou, Zheng Zhang, Yongqing Zhu, Shasha Li, Yizhuo Zhai, Hang Zhang, Weiteng Chen, Yu Hao, Xukan Ran, Xingyu Li, Guoren Li, Daimeng Wang, Chun-Yu Chuang, Ahmad Darki, Joobin Gharibshah, Kittipat Apicharttrisorn, Dang Tu Nguyen, and many others.

Finally, I want to show my deepest gratitude to my family and my parents, who always have my back and support me unreservedly. Without your love, I would not have been here.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Evasion Attacks on Network Intrusion Detection: Investigation, Automation, and Mitigation

by

Zhongjie Wang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2021
Professor Zhiyun Qian, Co-Chairperson
Professor Srikanth V. Krishnamurthy, Co-Chairperson

Stateful network protocols, such as the Transmission Control Protocol (TCP), play a significant role in the modern Internet, taking part in almost every network application running on billions of user devices, including computers, smartphones, IoT devices, vehicles, etc. However, due to inevitable ambiguities in network protocol specifications, discrepancies are prevalent among different network protocol implementations and even different versions of the same implementation. As a result, discrepancies could lead to severe security vulnerabilities. One kind of such vulnerabilities is caused by discrepancies between the network stack of a network intrusion detection system (NIDS) and those of the endhosts. A deliberate attacker could leverage the discrepancies to craft network traffic that will be interpreted differently by the NIDS and the endhosts, and then mount an attack that can bypass the NIDS. Furthermore, due to the statefulness of the network protocol, the attacker can manipulate the state on the NIDS to permanently disable the NIDS on any connection.

Our research focuses on the study of discrepancies among TCP implementations of

NIDSes and endhosts, towards understanding the exploitation of and defense against vulnerabilities caused by discrepancies. We start first by manually investigating the discrepancies and then move on to automated techniques. More specifically, 1) we first investigate the most powerful censorship firewall on the Internet and discover the discrepancies between its implementation and that of a Linux server, which allows an adversary to evade the firewall; 2) in order to automatically discover such implementation-level discrepancies, we develop a general approach which employs automated testing and symbolic execution techniques to automatically explore the program space of the Linux TCP stack and thereby discover network packets that can evade deep packet inspection used by modern stateful firewalls and intrusion detection systems; 3) we develop a systematic approach to extract comprehensive and high-fidelity models from various versions of the Linux TCP implementation and exhaustively discover all discrepancies between them; we then build a NIDS that incorporates the discovered discrepancies and is immune to all evasion attacks. Ultimately, we seek to develop widely used tools that employ automated testing techniques to significantly improve the effectiveness and efficiency in discovering discrepancies among stateful network protocol implementations and prevent attacks that exploit such discrepancies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has become an essential part of people's everyday life. Along with the digital transformation, more and more devices are connected to the ubiquitous Internet, including computers, smartphones, IoT devices, vehicles, etc. Prosperity in the information age also brings new threats that could subvert network systems in a flash. One source of the threats lies in discrepancies between network protocol implementations. The reasons are multifold. First, protocol specifications are defined in natural languages, ambiguities are inevitable. This gives software developers the freedom to implement their own implementations differently in certain details, and still conform with the specification. Second, protocol specifications may evolve over time, new features added and old features deprecated. This will lead to discrepancies between older and newer versions.

These discrepancies are usually tolerated and ignored. However, they could lead to severe security vulnerabilities. For example, an attacker can leverage the discrepancies between the network stacks of a network intrusion detection system (NIDS) and an endhost to send crafted network traffic that will be interpreted differently by the NIDS and the endhost, and then mount an attack that can bypass the NIDS. And it is extremely challenging for the NIDS to be fully immune to such kind of attacks due to its lack of ground truth of the implementation running on the endhosts.

Discrepancies are commonplace in network protocol implementations, and are especially worth noticing in those of stateful network protocols, such as the Transmission Control Protocol (TCP). Stateful network protocols are more complex by design, and it's likely there are more ambiguities lie in the state machines of the protocols. A subtle discrepancy can be leveraged by a deliberate attacker to launch an attack against the NIDS to de-synchronize it from the connections it protects. Such discrepancies are usually tricky to find and exploit.

In order to understand the threats caused by discrepancies in stateful network protocol implementations and build defenses against them, my Ph.D. research focuses on analyzing the TCP stacks of Linux and network intrusion detection systems, using automated testing techniques for blackbox and whitebox systems, and formal verification techniques such as symbolic execution to discover discrepancies between various TCP implmenetations. In Chapter 2, we model the most powerful censorship system, which operates using the same technology as NIDS, deployed on today's Internet, to discover the discrepancies between its implementation and those of servers, and then study its evadability based on those discrepancies; In Chapter 3, we aim at automatically finding discrepancies between network protocol implementations of NIDSs and endhosts; we use the Linux TCP stack as a reference model, employ automated testing and symbolic execution techniques to explore its program space, and then use differential testing to discover discrepancies between it and the TCP stacks of state-of-the-arts NIDSs; In Chapter 4, we seek to address the root causes of the evasion attacks, which stem from the diversity of endhost implementations; we use exhaustive symbolic execution to extract high-fidelity models from various TCP implementations, systematically discover discrepancies among them, and then integrate the discrepancies into the NIDS to build a robust and ambiguity-aware NIDS that is immune to discrepancy-based evasion attacks.

1.1 A Closer Look at Evading Stateful Internet Censorship

There has been recent interest in understanding the behaviors of, and evading state-level, stateful Internet scale censorship systems such as the Great Firewall (GFW) of China. We undertake, arguably, the most extensive measurement study to date on TCP-level censorship evasion on GFW, with several vantage points within and outside China, and with clients subscribed to multiple ISPs. Interestingly, we find that the state-of-the art evasion techniques are no longer very effective on the GFW. We perform a series of measurements and reverse engineering to reveal the failure reasons of previously successful censorship evasion techniques — primarily because the GFW has evolved over time. In addition, other factors such as the presence of middleboxes on the route from the client to the server, result in previously unexpected behaviors. Our measurement study leads us to propose new techniques of evasion based on a new understanding of the GFW. We also build an easy to use, flexible measurement-driven tool, INTANG, that iteratively determines the best evasion strategy for a given client server pair, and allows us to perform extensive evaluations of our new methods. Our results show that our new techniques provide extremely high success rates (unlike the prior schemes) of close to 100 %, thus, validating our new understanding of the GFW's Internet scale censorship. We show that these TCP-level evasion techniques can help evading not only HTTP censorship but also DNS (over TCP) and Tor/VPN.

1.2 SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery

A key characteristic of commonly deployed deep packet inspection (DPI) systems is that they implement a simplified state machine of the network stack that often differs from that of endhosts. The discrepancies between the two state machines have been exploited to bypass such DPI based middleboxes. However, most prior approaches to do so rely on manually crafted adversarial packets, which not only are labor-intensive but may not work well across a plurality of DPI-based middleboxes. Our goal in this work is to develop an automated way to craft candidate adversarial packets, targeting TCP implementations in particular. Our approach to achieving this goal hinges on the key insight that while the TCP state machines of DPI implementations are obscure, those of the endhosts are well established. Thus, in our system SymTCP, using symbolic execution, we systematically explore the TCP implementation of an endhost, identifying candidate packets that can reach critical points in the code (e.g., which causes the packets to be accepted or dropped/ignored); such automatically identified packets are then fed through the DPI middlebox to determine if a discrepancy is induced and the middlebox can be eluded. We find that our approach is extremely effective. It can generate tens of thousands of candidate adversarial packets in less than an hour. When evaluating against multiple state-of-the-art DPI systems such as Zeek and Snort, as well as a state-level censorship system, viz. the Great Firewall of China, we identify not only previously known evasion strategies, but also novel ones that were never previously reported (e.g., involving the urgent pointer). The system can be extended easily towards other combinations of operating systems and DPI middleboxes, and serves as a valuable tool for testing future DPIs' robustness against evasion attempts.

1.3 Themis: Ambiguity-Aware Network Intrusion Detection based on Symbolic Model Comparison

It is known that network intrusion detection systems (NIDS) can be evaded by carefully crafted packets that exploit implementation-level discrepancies between how they are processed on the NIDS and at the endhosts. Recently, new evasion strategies have emerged due to improvements in methods that can uncover them. These discrepancies arise because of the plethora of endhost implementations and evolutions thereof. It is prohibitive to proactively employ a large set of implementations at the NIDS and check incoming packets against all of those. As a result, NIDS typically choose simplified implementations that attempt to approximate and generalize across the different endhost implementations. Unfortunately, this solution is fundamentally flawed since such approximations are bound to have discrepancies with some endhost implementations. In this paper, we develop a lightweight system Themis, which empowers the NIDS in identifying these discrepancies and reactively forking its connection states when any packets with "ambiguities" are encountered. Specifically, Themis incorporates an offline phase in which it extracts models from various popular implementations using symbolic execution. During runtime, it maintains a nondeterministic finite automaton to keep track of the states for each possible implementation. Our extensive evaluations show that Themis is extremely effective and can detect all evasion attacks known to date, while consuming extremely low overhead. En route, we also discovered multiple previously unknown discrepancies that can be exploited to bypass current NIDS.

Chapter 2

A Closer Look at Evading Stateful Internet Censorship

2.1 Introduction

Internet censorship and surveillance are prevalent nowadays. State-level censorship systems such as NSA's PRISM and the Great Firewall (GFW) of China, have the capability of analyzing terabyte-level traffic across the country in realtime. Protocols with plaintext (*e.g.,* HTTP, DNS, IMAP), are directly subject to surveillance and manipulation by the governors [3, 4, 86, 7, 56, 121], while protocols with encryption (*e.g.,* SSH, TLS/SSL, PPTP/MPPE) and Tor, can be identified via traffic fingerprinting, leading to subsequent blocking at the IP-level [49, 130].

The key technology behind these censorship systems is Deep Packet Inspection (DPI) [117], which also powers Network Intrusion Detection Systems (NIDS). As previously reported, most censorship NIDS are deployed “on-path” in the backbone and at border routers [117, 121, 134].

In order to examine application-level payloads, DPI techniques have to correctly implement the underlying protocols like TCP, which is the cornerstone of today’s Internet. Ptacek *et al.* [92] have shown that any NIDS is inherently incapable of always reconstructing a TCP stream the same way as its endpoints. The root cause for this is the presence of discrepancies between the implementations of the TCP (and possibly other) protocol at the end-host and at the NIDS. Even if the NIDS perfectly mirrors the implementation of one specific TCP implementation, it may still have problems processing a stream of packets generated by another TCP implementation.

Because of this ambiguity in packet processing, it is possible for a sender to send carefully crafted packets to desynchronize the TCP Control Block (TCB) maintained by the NIDS from the TCB on the receiver side. In some cases, the NIDS can even be tricked to completely deactivate the TCB (*e.g.,* after receiving a spurious RST packet), effectively allowing an adversary to “manipulate” the TCB on the NIDS. Censorship monitors suffer from the same fundamental flaw—a client can evade censorship if the TCB on the censorship monitor can be successfully desynchronized with the one on the server. Different from other censorship evasion technologies such as VPN, Tor, and Telex [132], that rely on additional network infrastructure (*e.g.,* proxy node) [117], TCB-manipulation-based evasion techniques only require crafting/manipulating packets on the client-side and can potentially help all TCP-based application-layer protocols “stay under the radar.” Based on this idea, Khattak *et al.* [68] explored several practical evasion techniques against the GFW, by studying its behaviors at the TCP and HTTP layers. The West Chamber Project [101] provides a practical tool that implemented a few of the evasion strategies but has ceased development since 2011; unfortunately none of the strategies were found to be effective during our measurement study. Besides these attempts, there is no recent data point, showing how these evasion techniques work in the wild.

In this work, we extensively evaluate TCP-layer censorship evasion techniques against the GFW. By testing from 11 vantage points inside China spread across 9 cities (and 3 ISPs), we are able to cover a variety of network paths that potentially include different types of GFW devices and middleboxes (see subsection 2.3.3 for details). We measure how TCB manipulation can help HTTP, DNS, and Tor evade the GFW.

First, we measure how existing censorship evasion strategies work in practice. Interestingly, we find that most of them no longer work well due to unexpected network conditions, interference from the network middleboxes, or more importantly, new updates to the GFW (different from the model considered previously). These initial measurement results motivate us to construct probing tests to infer the "new" updated GFW model. Finally, based on the new GFW model and lessons learned with regards to other practical challenges in deploying TCP-layer censorship evasion, we develop a set of new evasion strategies. Our measurement results show that the new strategies have a 90% or higher, evasion success rate. We also evaluate how these new strategies can help HTTP, DNS, Tor, and VPN evade the GFW.

In addition, during the course of our measurement study, we design and implement a censorship evasion tool, INTANG, integrating all of the censorship evasion strategies considered in this work; INTANG is easily extensible to incorporate additional strategies. It requires zero configuration and runs in the background to help normal traffic evade censorship. We plan to open source the tool to support future research in this direction.

We summarize our contributions as the follows:

- We perform the largest measurement study to date, of the GFW's behaviors with TCP-layer censorship evasion techniques.
- We demonstrate that existing strategies are either not working or are limited in practice.
- We develop an updated and more comprehensive model of the GFW based on the measurement results.
- We propose new, measurement-driven strategies that can bypass the new model.
- We measure the success rates of our improved strategies with regards to censorship evasion for HTTP, DNS, VPN, and Tor. The results show very high success rates ($>$ 90 %).
- We develop an open-source tool to automatically measure the GFW's responsiveness, and for censorship circumvention. The tool is extensible as a framework for the integration of additional evasion strategies that may emerge from future research.

2.2 Background

In this section, we provide the background on DPI-based censorship techniques employed by the GFW and discuss previously proposed evasion strategies.

2.2.1 On-path Censorship Systems

An "on-path" censorship system wiretaps routers of the ISPs controlled by the censor, makes copies of the packets on the fly and performs analysis in parallel with ongoing traffic. In contrast, an "in-path" censorship system places devices as part of a route, analyzes the traffic and then passes the same to the next hop. The capabilities of an "on-path" system include reading packets and injecting new packets, while an "in-path" system can also discard and/or modify packets. For an "on-path" system, processing time is not critical and thus, it can do more sophisticated analysis; for an "in-path" system, it is critical not to perform heavy analysis that will introduce packet delays. Large-scale censorship systems like the GFW usually deploy the "on-path" design in order to ensure extremely high throughput.

To examine the application-layer content with DPI, a censorship system like the GFW needs to first reassemble TCP streams from the packets. As reported [68], the GFW has a simplified TCP implementation to reconstruct the TCP data flow and pass it to the upper layer for further analysis. The GFW is able to analyze a wide range of application protocols (*e.g.,* HTTP, DNS, IMAP), and can apply its rule-based detection engine to detect sensitive application content.

TCP connection reset is a versatile censorship technique. Due to the "on-path" nature of the GFW, it cannot discard the undesired packets between a pair of end-hosts. Instead it can inject packets to force the connection to shut down, or disrupt connection establishment. Once any sensitive content is detected, the GFW injects RST (type-1) and RST/ACK (type-2) packets to both the corresponding client and the server to disrupt the ongoing connection and sustains the disruption for a certain period (90 seconds as per our measurements). During this period, any SYN packet between the two end-hosts will trigger a *forged* SYN/ACK packet with a wrong sequence number from the GFW, which will obstruct the legitimate handshake; any other packets will trigger forged RST and RST/ACK packets, which will tear down the connection.

According to previous work [5, 101] and our measurements, RST (type-1) and RST/ACK (type-2) are likely from two types of GFW instances that usually exist together. We have encountered some occurrences where a type-1 or a type-2 reset occurs individually; thus, we are able to measure their features separately. Type-1 reset has only the RST flag set, and random TTL value and window sizes, while type-2 reset has the RST and ACK flags set, and cyclically increasing TTL value and window sizes.

Once a sensitive keyword detected, the GFW sends one type-1 RST and three type-2 RST/ACK with sequence numbers X, X+1460 and X+4380 (X is the current server-side sequence number). [1] Note that only type-2 resets entail forged SYN/ACK packets during the 90-second subsequent blocking period; furthermore, only type-2 resets are seen when we split a HTTP request into two TCP packets. From all of the above, we speculate that the type-2 resets are from more advanced GFW instances or devices.

Numerous studies have focused on the TCP connection reset of the GFW. Xu *et al.* [134] perform measurements to determine the locations of the censor devices injecting RST packets. Crandall *et al.* [41] employ latent semantic analysis to automatically generate an up-to-date list of censored keywords. Park *et al.* [86] measure the effectiveness of RST packet injection for keyword filtering on HTTP requests and responses, and provide

[1]The common size of a full TCP packet is 1460 bytes. Sometimes injected packets can fall behind a server's response and thus, become obsolete and discarded. Sending packets with future sequence numbers can offset this effect to a large extent.

insights on why filtering based on HTTP responses has been discontinued. Performing TCP connection reset does come with shortcomings. For instance, it is costly to track the TCP state of each and every connection and match keywords against a massive number of TCP packets. It is also not completely resistant to evasion.

DNS poisoning is another common technique used by the GFW [6, 7, 76]. The GFW censors the DNS requests over both UDP and TCP. For a UDP DNS request with a blacklisted domain, it simply injects a fake DNS response; for a TCP DNS request, it turns to the connection reset mechanism. Our measurements also cover *DNS over TCP*.

2.2.2 Evasion of NIDS and Censorship Systems

Ptacek *et al.* [92] have systematically studied the vulnerabilities of NIDS in the way that NIDS construct and maintain TCP state. In particular, NIDS maintain a TCP Control Block (TCB) for each live connection to track its state information (*e.g.,* TCP state, sequence number, acknowledgment number, etc.). The goal is to replicate the same exact connection information that exists at both endpoints. However, in practice this is very challenging due to the following factors:

- *Diversity in host information.* Due to ambiguity and updates in TCP specifications, different OS implementations may have very different behaviors in handling TCP packets. For instance, when unexpected TCP flag combinations are encountered, different OSes can behave differently (as how to handle these remains unspecified in the standard). Another example is that RST packet handling has drastically changed over different TCP standards (RFC 793 to RFC 5961).

- *Diversity in network information.* A NIDS usually cannot learn the network topology with respect to the endpoints it is protecting, since the topology itself may change over time. For a LAN, a NIDS can probe and maintain the topology. However, for a censorship system, monitoring the massive scale of the entire Internet is extremely challenging if at all possible. Further, such a system will be unaware of network failures or packet losses. Thus, it cannot judge accurately whether or not a packet has arrived at its destination.

- *Presence of middleboxes.* NIDS usually are not aware of other middleboxes that may be encountered between any pair of communicating endpoints. These middleboxes may drop or even alter packets after the NIDS process them, which makes it even more difficult to reason about how a receiver will behave.

This observation has motivated work on TCP reset attack evasion. For example, Khattak *et al.* [68] manually crafted a fairly comprehensive set of the evasion strategies at the TCP and HTTP layers against the GFW and verified them successfully in a limited setting with a fixed client and server. Unfortunately, there are a large number of factors that were not taken into account (*e.g.,* different types of GFW devices may be encountered on different network paths, various middleboxes may interfere with the evasion strategies by dropping crafted packets).

2.3 Measurement of Existing Evasion Strategies

Based on the fundamental limitations of NIDS outlined by Ptacek *et al.* [92], the GFW's modeling by the Khattak *et al.* [68], and the implementation of the West Cham-

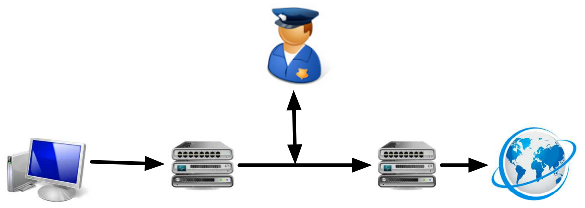


Figure 2.1: Threat Model of INTANG

ber Project [101], we divide censorship evasion strategies based on TCB-manipulations into three high-level categories, viz., (1) *TCB creation*, (2) *data reassembly*, and (3) *TCB teardown*. In this section, we perform in-depth measurements to evaluate the effectiveness of existing evasion strategies, developed based on the currently known model of the GFW in these categories.

2.3.1 Threat Model

The threat model is depicted in Figure 2.1. The client initiates a TCP connection with the server. The GFW establishes a *shadow* connection by creating a TCB and can read from and inject packets to the original connection. Meanwhile, there could be network middleboxes on the path. We refer to the middleboxes between the client and the GFW as *client-side middleboxes* and the middleboxes between the GFW and the server as *server-side middleboxes*.

2.3.2 Existing Evasion Strategies

The goal of current evasion strategies (listed below) is to cause the GFW and the server to enter different states (*i.e.,* become desynchronized) by sending specially crafted packets, especially "insertion" packets. These insertion packets are crafted such that they are ignored by the intended server (or never reach the server) but are accepted and processed by the GFW.

TCB Creation. As per previous work [68], the GFW creates a TCB upon seeing a SYN packet. Thus the client can send a SYN insertion packet with a fake/wrong sequence number to create a false TCB on the GFW, and *then* build the real connection. The GFW will ignore the real connection because of its "unexpected" sequence number. The TTL (time to live) or checksum in the insertion packet, is manipulated to prevent the acceptance of the first injected SYN by the server—a packet with a lower TTL value would never reach the intended server and a packet with wrong checksum would be discarded by the server.

Data Reassembly. The data reassembly strategy has two cases:

1. Out-of-order data overlapping. Different TCP implementations treat overlapping out-of-order data fragments in different ways. Previous work [68] has shown that if the GFW encounters two out-of-order IP fragments with the same offset and length, it prefers (records) the former and discards the latter. However, with regards to out-of-order TCP segments with the same sequence number and length, it prefers the latter (details in [68]). This characteristic with regards to IP fragmentation can be exploited as follows. First, a gap is intentionally left in the payload and a fragment with offset X and length Y, containing random garbage data is sent. Subsequently, the real data with offset X and length Y, containing the sensitive keyword, is sent to evade the GFW (since the GFW is expected to choose the former packet). Finally the gap is filled by sending the real data with offset 0 and length X. To exploit the GFW's handling of TCP segments, we simply switch the order of the garbage data and the real data.

2. In-order data overlapping. When two in-order data packets carrying IP or TCP fragments arrive, both the GFW and the server will accept the first in-order packet that carries a specific fragment (specified by offset/sequence number). One can then craft insertion packets that contain junk data to fill the GFW's receive buffer, while making them to be ignored by the server. For example, one can craft an insertion data packet with a small TTL or a wrong checksum; such packets either never reach or are dropped by the server but are accepted and processed by the GFW.

TCB Teardown. As per the known model, the GFW is expected to tear down the TCB that it maintains when it sees a RST, RST/ACK, or a FIN packet. One can craft such packets to cause the TCB teardown, while manipulating fields such as the TTL or the checksum to ensure that the connection on the server is alive.

2.3.3 Experimental Setup

We employ 11 vantage points in China, in 9 different cities (Beijing, Shanghai, Guangzhou, Shenzhen, Hangzhou, Tianjin, Qingdao, Zhangjiakou, Shijiazhuang) and spanning 3 ISPs. 9 of these use the cloud service providers (Ailyun and Qcloud) and the other two use home networks (China Unicom). The servers are chosen from Alexa's top websites worldwide. We first filter out the websites that are affected by IP blocking, DNS poisoning, or are located inside China. We exclude the websites that use HTTPS by default, for two reasons. First, HTTPS traffic is not currently censored by the GFW; thus, we can already access them freely without using any anti-censorship technique. Second, if we access these HTTPS websites using HTTP, they send HTTP 301 responses to redirect us to HTTPS, and the sensitive keyword is copied to the *Location* header field of the response. We find

Table 2.1: Probing the GFW from 11 vantage points with 77 websites

Strategy	Discrepancy	w/ sensitive keyword			w/o sensitive keyword	
		Success	Failure 1	Failure 2	Success	Failure 1
No Strategy	N/A	2.8%	0.4%	96.8%	98.9%	1.1%
TCB creation (SYN)	TTL	6.9%	4.2%	88.9%	95.3%	4.7%
	Bad shecksum	6.2%	5.1%	88.7%	93.5%	6.5%
Out-of-order reassembly	IP fragments	1.6%	54.8%	43.6%	45.1%	54.9%
	TCP segments	30.8%	6.5%	62.6%	92.8%	7.2%
In-order reassembly	TTL	90.6%	5.7%	3.7%	95.1%	4.9%
	Bad ACK num	83.1%	7.5%	9.5%	93.5%	6.5%
	Bad checksum	87.2%	1.9%	10.8%	98.4%	1.6%
	No TCP flag	48.3%	3.3%	48.4%	97.1%	2.9%
TCB teardown (RST)	TTL	73.2%	3.2%	23.6%	94.7%	5.3%
	Bad checksum	63.1%	7.6%	29.3%	89.5%	10.5%
TCB teardown (RST/ACK)	TTL	73.1%	3.2%	23.7%	97.1%	2.9%
	Bad checksum	68.9%	1.9%	29.2%	98.2%	1.8%
TCB teardown (FIN)	TTL	11.1%	1.0%	87.9%	99.4%	0.6%
	Bad checksum	8.4%	0.8%	90.7%	99.0%	1.0%

that the GFW devices on some paths can in fact detect this in the response packets. This is similar to the HTML response censorship measured in [86]. Furthermore, assuming that GFW devices deployed in a particular autonomous systems (AS) usually are of the same type and version, and configured with the same policy, we choose only one IP from each AS, in order to diversify our experiments by spanning a large set of ASes. By applying filters based on the above rules, and removing a few slow or unresponsive websites, we finally obtain a dataset of 77 websites (from the considered 77 ASes) with Alexa ranks between 41 and 2091. We manually verify that these websites are accessible (outside of China) and are affected by GFW's TCP connection reset upon containing a sensitive keyword, *i.e.*, *ultrasurf*, in the HTTP request. For each strategy and website, we repeat the test 50 times and find the average. Since the GFW will blacklist a pair of hosts for 90 seconds upon the detection of any sensitive keyword, we add intervals between tests when necessary.

2.3.4 Results

We measure the effectiveness of existing strategies in evading the GFW during April and May in 2017. The results are summarized in Table 2.1.

Notation: We use the following notation in Table 2.1: *Success* means that we receive the HTTP response from the server and receive no reset packets from the GFW. *Failure 1* means that we receive no HTTP response from the server nor do we receive any resets from the GFW. *Failure 2* means that we receive reset packets from the GFW, i.e., either RST (type-1) or RST/ACK (type-2).

Results. Our findings are summarized below.

- We find that, possibly because of overloading of the GFW, even if we do not use any evasion strategy, there is a still a 2.8% success rate with regards to retrieving sensitive content. Interestingly this behavior was first documented in 2007 [41] and persists until now.

- We see that TCB creation with SYN does not generally work and has a high "Failure 2" rate (around 89%).

- With regards to data reassembly, we find that (a) out of order data reassembly strategies have both high "Failure 1" and high "Failure 2" rates but (b) sending in-order data to prefill the GFW's buffer results in a much higher success rate (typically $>$ 80%).

- TCB teardown with FIN experiences high "Failure 2" rates while TCB teardown with RST or RST/ACK experience around a 70% success rate, but with a 25% chance trigger reset packets from the GFW.

Evolution of the GFW. We believe that the primary reason for the high failure rates with many existing strategies is because the model of GFW assumed in previous work [68] is no longer valid. While we defer a detailed discussion of how the model has evolved to the next section, we point out here that the "checksum" field is still not validated by the GFW, *i.e.,* a packet with a wrong checksum is still a good insertion packet (the GFW considers it to update its TCB but the server discards it) if there's no interference from network middleboxes. We break down the results with regards to the other reasons why these strategies fail, and analyze them below.

Interference from client-side middleboxes. Client-side middleboxes may drop our insertion packets. Since we manipulate packet fields (*e.g.,* wrong checksum, no TCP flag, *etc.*) to cause the server or server-side middleboxes to discard insertion packets, client-side middleboxes could also discard them. Thus the strategies are voided, and will result in "Failures 2."

On the other hand, some NAT or state/sequence checking firewalls deployed on the client-side of the network might intercept and accept the insertion packets and change their maintained connection state. In such cases, later packets will not go through these middleboxes, resulting in "Failures 1." For example, if a RST packet tears down the connection on a client-side middlebox which it traverses, the middlebox blocks later packets on that connection.

Some client-side middleboxes may discard IP fragments (wrt data reassembly strategies) and cause "Failures 1." Others buffer and reassemble them into a whole IP

Table 2.2: Client-side middlebox behaviors

Packet Type	Aliyun	QCloud	China Unicom SJZ	China Unicom TJ
IP fragments	Discarded	Reassembled	Reassembled	Reassembled
Bad TCP checksum	Pass	Pass	Pass	Dropped
No TCP flag	Pass	Pass	Pass	Dropped
RST packets	Pass	Sometimes dropped	Pass	Pass
FIN packets	Sometimes dropped	Pass	Dropped	Dropped

packet and this might cause "Failures 2" depending on the implementation of the middlebox.

We probed for client-side middleboxes from all our 11 vantage points trying to connect with our own servers. As shown in Table 2.2, we found that our 6 clients using Aliyun were unable to send out IP fragments. One can conclude within reason that Aliyun has configured its middleboxes to discard certain kinds of IP fragments. We found that connections from the other 5 nodes encounter client-side middleboxes, which reassemble the IP fragments into a full IP packet containing the original HTTP request; thus these packets were deterministically captured by the GFW. Since we found that most of the routers and/or middleboxes interfere with IP-layer manipulations, we argue that this is not as *generally* applicable as TCP-layer manipulations for evasion.

The vantage point in Tianjin China Unicom has client-side middleboxes that drop packets with wrong TCP checksums or containing no TCP flag; thus these two strategies did not work at that point. Finally, we found Aliyun sometimes drops FIN insertion packets and QCloud sometimes drops RST insertion packets. Both the clients in Shijiazhuang and Tianjin (China Unicom) have client-side middleboxes that drop FIN insertion packets.

Interference from server-side middleboxes. Server-side middleboxes only affect the server but not the GFW. Our insertion packet may terminate the connection or change the connection state on the server-side middleboxes causing later packets to be blocked by the middleboxes. This will cause "Failures 1." To verify interference from server-side middleboxes, we need to either control the server or set up our own server on the same path behind those middleboxes, which are infeasible in practice for all our targets, *i.e.,* the Alexa's top websites.

Other reasons for failures. There could be a few other reasons for observing failures of the two types. Network or server failures although rare could occur. We performed microscopic studies of our failure cases and list the cases that we observed below.

Variations in server implementations. We find that with some server implementations (*e.g.,* Linux versions prior to 3.8), a data packet under "in-order data overlapping strategy" carrying no TCP flag can sometimes be accepted by the server and thus causes "Failures 1." With the "out-of-order data overlapping strategy," a server might accept the junk data (just like the GFW) and discard the correct packet.

Network dynamics. Since routes are dynamic and could change unexpectedly, the TTL values used in the insertion packets to prevent them reaching the server could be incorrect. As a result, they may reach the server and disrupt the connection (Failures 1). In other cases, the insertion packets might not reach the GFW and lead to "Failures 2." We also found that packet losses on the network could affect the insertion packets and cause "Failures 2." We cope with such dynamics by repeating the sending of the insertion packets thrice with 20ms intervals.

Summary. Our measurement uses real web servers instead of controlled servers in order to represent cases of daily web browsing. The results demonstrate the complexity induced by many factors (*e.g.,* middlebox interference, server diversity, network diversity, *etc.*). We showcase the overall success rates with existing evasion strategies and enumerate possible reasons for the failure cases. To fully untangle the factors causing failures and to quantify the impact of each, more in-depth analysis and controlled experiements are required (*e.g.,* using controlled replay server as in [74]), which we leave for future work.

2.4 Evolved GFW Behaviors

As alluded to in section 2.3, high failure rates were experienced even if we eliminated the effects from middleboxes, server implementations, and network dynamics. To understand the root cause, we take a closer look and argue that this is due to evolved GFW behaviors that break many prior assumptions. Based on our measurements, we hypothesize these new behaviors as follows. To verify these hypotheses, in section 4.5 we design and extensively evaluate new evasion strategies.

Prior Assumption 1: *The GFW creates a TCB* only *upon seeing a SYN packet.*

To test this assumption, we used pairs of clients and servers under our control, and executed partial TCP 3-way handshakes (*e.g.,* intentionally omitting the SYN, SYN/ACK and/or ACK) followed by a HTTP request with a sensitive keyword. If a correct TCB was created on the GFW, the HTTP request would trigger TCP reset packets from it. First, our results confirmed that the GFW still creates a TCB upon seeing a SYN packet as described in [68]. Second and more interestingly, we found that the GFW also creates a TCB upon seeing a SYN/ACK packet *without* the SYN packet. We speculate that the GFW has evolved to incorporate this feature to counter SYN packet losses. Given these, we hypothesize that the GFW exhibits the following new behavior.

Hypothesized New Behavior 1: *The GFW creates a TCB not only upon receiving SYN packets, but also SYN/ACK packets.*

Prior Assumption 2: *The GFW uses the sequence number in the first SYN packet to create a TCB, and ignores later SYN packets during the lifetime of the TCB.*

This assumption is based on the rationale that the GFW mimics a normal TCP implementation. Our closer look revealed that it does not. From the results in section 2.3, we see that the TCB creation with a SYN insertion packet failed in most cases. This leads us to re-examine this case. We send multiple SYN packets among which only one has the "true" sequence number, and then send a sensitive HTTP request. However, no matter where we put the "true" SYN packet, the GFW can always detect the later sensitive keyword. We hypothesize that this could be because of any of three possible reasons:

- (1) the GFW establishes multiple TCBs, one for each SYN packet;
- (2) the GFW enters a "stateless mode", in which it checks every individual packet instead of re-assembling the data first (and check for a sensitive keyword);
- (3) the GFW uses the sequence number in the HTTP request to re-synchronize its TCB.

To check (1), we set the sequence number in the HTTP request to be a "out-of-window" value with respect to the sequence numbers in the SYN packets; however, we find that the GFW can still detect the keyword. To examine (2), we split the sensitive keyword into halves, each of which by itself is *not* a sensitive keyword; however, we find that the GFW can still detect it. For (3), before sending the HTTP request, we send some random data with a “false” sequence number, and then we send the HTTP request with “true” sequence number; the GFW cannot detect it in this case. This suggests that the GFW re-synchonrizes its TCB with the sequence number in the random data, and thus, ignores the later HTTP request because of its out-of-window sequence number. This validates hypothesis (3) that the GFW enters a “re-synchronization state” upon seeing multiple SYN packets. We further validate this extensively in section 4.5.

Besides multiple SYN packets, we found that multiple SYN/ACK packets or a SYN/ACK packet with an incorrect acknowledgment number can also cause the GFW to enter the re-synchronization state.

Next, we try to find out “which packet the GFW uses to re-synchronize its TCB once in *re-synchronization state*.” From the previous experiement, we learn that the GFW re-synchronizes using data packets from the client to the server. Thus, instead, we try to use data packets from the server to the client; in addition, we try pure ACK packets without data in both directions. We find none of these packets affect the GFW. However, we do find that a SYN/ACK packet from the server to the client can cause re-synchronization. We admit that the cases we found may not be complete but it is hard to enumerate an exhaustive set of these cases. However, our measurements lead us to a better understanding of the GFW behavior than what exists today and leads us to the following new hypothesis.

Hypothesized New Behavior 2: *The GFW enters what we call the “re-synchronization state”, where it re-synchronizes its TCB using the information in the next SYN/ACK packet from server to client or data packet from client to server upon experiencing any of the following three cases: (a) it sees multiple SYN packets from client-side, (b) it sees* *multiple SYN/ACK packets from server-side, or (c) it sees a SYN/ACK packet with an acknowledgment number different from the sequence number in the SYN packet.*

Prior Assumption 3: *The GFW tears down a TCB when it sees a RST, RST/ACK or FIN packet.*

The results in section 2.3 suggest that the evolved GFW generally does not tear down a TCB merely upon seeing FIN packets. At the same time, we also observed high failure rates of above 20% with our RST and RST/ACK insertion packets. A closer look suggests that this probably is due to “Hypothesized New Behavior 2.” More specifically, we found that when the GFW is in the newly discovered “re-synchronization state”, its TCB sometimes cannot be torn down with RST or RST/ACK packets. To verify this, we force the GFW to enter the re-synchronization state using one of the techniques above, and then immediately send a RST packet and a HTTP request with sensitive keyword. However, the GFW sometimes can still detect it. We repeated the experiement at different times with multiple pairs of clients and servers, and found inconsistency between different measurements across pairs at different times. The overall success rate is roughly 80%, and for a specific client-server pair, the GFW’s behavior is usually consistent during a certain period (although not always across periods). We are unable to unearth the explicit reason behind at this time; we conjecture that it is due to dynamics with regards to the heterogeneity in the types of GFW encountered and the complexity of interactions among different GFW instances and middleboxes. We discuss this further in section 2.8.

In addition, we performed extensive measurements wherein we sent a RST packet between the SYN/ACK and the ACK packet of the 3-way handshake, and also after the

3-way handshake. We found that in both cases the TCB sometimes is not torn down but the RST packet caused the GFW to enter the re-synchronization state; further, we find that this happens way more frequently for the former case (the exact reason for the discrepancy remains unknown). These observations lead to the following new hypothesis.

Hypothesized New Behavior 3: *Upon receiving a RST or RST/ACK packet, the GFW may enter the re-synchronization state instead of tearing down the TCB.*

2.5 New Ways to Evade the GFW

In this section, we discuss new opportunities for evasion from two perspectives. First, based on the new hypothesized behaviors of the GFW, we propose new evasion strategies. Second, we attempt to systematically discover new insertion packets (besides wrong checksum or small TTL).

2.5.1 Desynchronize the GFW

First of all, we describe a building block to counter the re-synchronization state in the GFW. It is helpful in supporting our new evasion strategies, which are discussed next. Specifically, when we expect that the GFW is in the re-synchronization state (this can be forced), we send a insertion data packet with a sequence number that is out of window. Once the GFW synchronizes with the sequence number in this insertion packet, subsequent legitimate packets of the connection will be perceived to have sequence numbers that are out of window, and thus be ignored by the GFW. We say that now the GFW is *desynchronized* from the connection. Note that the insertion data packet is ignored by the server since it contains an out-of-window sequence number.

Desynchronizing the GFW drastically helps improve the "TCB Teardown" and the "In-order Data Overlapping" strategy that still work relatively well but occasionally experience undesired high "Failure 1" and "Failure 2" rates.

2.5.2 New Evasion Strategies

Our evasion strategies are primarily based on exploiting the newly discovered state of the GFW. We propose two new evasion strategies along with improvements to two existing strategies.[2] We evaluate these extensively in section 4.5. The two new strategies are as follows:

Resync + Desync. To coerce the GFW into entering the re-synchronization state, the client sends a SYN insertion packet after the 3-way handshake. Subsequently, the client sends a 1-byte data packet containing an out-of-window sequence number to desynchronize the GFW. This is then followed by the real request. Note that the SYN insertion packet cannot be sent prior to receiving the SYN/ACK packet, as the GFW will eventually resynchronize the expected client-side sequence number based on the ACK number of the SYN/ACK. In addition, the SYN insertion packet should take a sequence number outside of the expected receive window of the server (as in older Linux this can cause the connection to reset). Newer versions of Linux will never accept such a SYN packet regardless of its sequence number and will simply respond with a challenge ACK [28]. In addition, we can craft the insertion SYN packets with small TTL in case the server or

[2]For brevity we only describe the new strategies in this section and leave the detailed discussion of improved strategies to section 4.5.

middleboxes interfere.

TCB Reversal. As discussed, the GFW currently only censors traffic from the client to the server (*e.g.,* HTTP/DNS requests), and the censorship of HTTP response has been discontinued except in a few rare cases [86]. When the GFW first sees a SYN/ACK, it assumes that the source is the server and the destination is the client. It creates a TCB to reflect that this is the case. It will now monitor data packets from the server to the client (mistakenly thinking that it is monitoring data packets from the client to the server). To exploit this property, the client will first send a SYN/ACK insertion packet. It later performs the TCP three way handshake in a normal way. The GFW will ignore these handshake packets since there already exists a TCB for this connection. Note that the SYN/ACK insertion packet has to be crafted with care. In normal cases, the server responds with a RST which causes a teardown of the original TCB at GFW. To address this, one of the discrepancies (*e.g.,* lower TTL) will need to be used in the insertion packet. In addition, we point out that here the SYN/ACK and subsequent SYN packet from the client do not trigger the GFW to enter the resynchronization state.

2.5.3 New Insertion Packets

All GFW evasion strategies require injecting additional packets or modifying existing packets to disrupt the TCP state maintained on GFW [92, 68]. Insertion packets are especially handy as they are the most suitable for supporting evasion strategies against the GFW.

As alluded to in section 2.3, insertion packets can be tricky to craft. They may fail because of many reasons such as network dynamics, routing asymmetry, obscure network

Table 2.3: Discrepancies between GFW and server on ignoring packets (candidate insertion packets)

TCP State	GFW State	TCP Flags	Condition
Any	Any	Any	IP total length > actual length
Any	Any	Any	TCP Header Length < 20
Any	Any	Any	TCP checksum incorrect
SR	EST/RESYNC	RST+ACK	Wrong acknowledgement number
SR/EST	EST/RESYNC	ACK	Wrong acknowledgement number
SR/EST	EST/RESYNC	Any	Has unsolicited MD5 Optional Header
SR/EST	EST/RESYNC	No flag	TCP packet with no flag
SR/EST	EST/RESYNC	FIN	TCP packet with only FIN flag
SR/EST	EST/RESYNC	ACK	Timestamps too old

* SR - SYN_RECV; EST - ESTABLISHED; RESYNC - Re-synchronization.

middleboxes, and variations in server TCP stacks. Our observation is that none of the insertion packets are universally good. This motivates us to discover additional insertion packets that may be viable and complementary to existing insertion packets.

The ideal solution to discovering insertion packets is to obtain a precise TCP model for the GFW, the server, and network middleboxes that can be fed into an automated reasoning engine (to see what kinds of packets can qualify as insertion packets). However, since the GFW is a blackbox with only one observable feedback attribute (viz., the RST injection), it is quite hard to infer its internal state accurately and completely. The evolved GFW model that we infer in section 2.4 is also unlikely to be complete. Therefore, even if one were to leave network middleboxes aside, the problem is very challenging.

Our solution is as follows: instead of attempting to model the GFW accurately, we first model the servers (*e.g.,* popular Linux and FreeBSD TCP stacks) using "ignore" paths analysis. By this we mean that we want to identify and reason about points in a server's TCP implementation which cause it to ignore received packets. Specifically, for an incoming packet, we analyze all possible program paths that lead to the packet being either discarded completely, or “ignored” possibly with an ACK sent in response. An example of the first case is a packet with an incorrect checksum; the second case can be a data packet with an out-of-window sequence number, which triggers a duplicate ACK [90]. In both cases, the TCP state (*e.g.,* the next expected sequence number) of the host (server) remains unchanged. After we derive this server model, we use it to develop probing tests against the GFW.

For open source operating systems such as Linux, this can be achieved through static analysis similar to what is done in PacketGuardian [35]. The challenge is to manually identify all program points where “ignore” events occur. Once the ignore paths are identified, the constraints that lead to each path need to be computed, and used to guide test packets against the GFW. Once we identify cases where the packets are “accepted” by the GFW, *i.e.,* the GFW updates its TCB according to the information in the packet, we can conclude that such packets are effective insertion packets (note that we have not yet considered interference from network middleboxes).

During the analysis, we only need to consider the TCP states that still have the potential to receive data, *i.e.,* TCP_LISTEN, TCP_SYN_RECV, TCP_ESTABLISHED. For instance, we omit the TIME_WAIT state because the server can no longer receive data in this state and it is fruitless to understand its ignore paths. After we generate the ignore paths of the server for each TCP state, we first generate a sequence of packets that lead to the specific TCP state; then for the set of constraints generated for each ignore path, we generate one or more test packets (as candidate insertion packets). Note that each ignore path will lead to a unique reason for why the packet will be ignored by the server (*e.g.,* either wrong checksum or invalid ACK, but never both). Ptacek *et al.* [92] used a similar approach to study the FreeBSD TCP stack, which is unfortunately too old to be applicable. In contrast, we study the latest Linux TCP stack, which has many new behaviors. Further, we improve the methodology by pruning a number of "ignore" paths in irrelevant TCP states such as TIME_WAIT, as well as correlating the "ignore" cases with middlebox behaviors.

As a demonstration, we conduct such an analysis of Linux kernel version 4.4. In Table 2.3, we list the confirmed cases in which Linux ignores a packet but the GFW does not. We also try to compare the server state with the GFW state to make the discrepancies more clear. Note that this is a more complete list than what was previously reported [68, 92], demonstrating the advantage of our systematic analysis. For instance, the finding includes two new insertion packets:

1) RST/ACK packets with incorrect ACK number are ignored by the server in TCP_RECV state but GFW will accept such a packet and change its state to either TCP_LISTEN (previous state terminated) or TCP_RESYNC, depending on the GFW model.

2) Packets with unsolicited MD5 headers are ignored by the server (if no prior negotiation of optional MD5 authentication has been done) while GFW will process the packet as normal.

The MD5 header [63] discrepancy can be exploited in an insertion packet with any TCP flag. For example, this can be used in a RST packet to tear down the TCB on the GFW, or in a data packet to fool the GFW into changing its maintained client sequence number.

Note that we intentionally omit the analysis of data overlapping (for processing out-of-order and overlapping data packets) discrepancies as it has been understood that different OSes may employ different strategies [92] and thus it may not lead to a safe insertion packet.

Cross-validation with network middleboxes. Even though the insertion packets generated from the analysis work well according to our experiments, they may not play well with middleboxes. Note that IP layer discrepancies such as wrong IP checksum, IP optional header, and IP header length can be used under all TCP states for all TCP flags, but packets with such properties are often dropped by routers or middleboxes. The only feature that we find useful is the one where the "IP total length" is larger than the "actual packet length" (listed in Table 2.3); however, packets with this feature may still be checked and dropped by some middleboxes. Even insertion packets that leverage TCP layer discrepancies (such as those relating to improper TCP header lengths or the wrong TCP checksum) may still be dropped by middleboxes, especially in cases where the perturbation applies to all TCP states and flags. The only exceptions are insertion packets leveraging the unsolicited MD5 header; these are never dropped by the middleboxes we encounter during our experiments (presumably because it requires a stateful firewall middlebox to understand when such packets should be dropped).

The remainder of the insertion packets can be useful only for data packets. Effective control packets cannot be crafted with these; for instance, when the server is in the ESTABLISHED state, even if the RST/ACK has a wrong ACK number or old timestamp, it will still be able to reset the connection successfully. According to our experiments, we have not encountered middleboxes that drop packets with unexpected MD5 options, old timestamps, or incorrect ACK numbers.

Cross-validation with other TCP stacks. It is difficult, if not impossible, to exhaustively test the ignore paths of all deployed TCP stacks. We cross-validate the ignore paths of Linux kernel 4.4 with several other popular Linux versions, including 4.0, 3.14, 2.6.34, and 2.4.37. We summarize the results here:

- In Linux 3.14, when a connection is in the ESTABLISHED state, an incoming packet with a SYN flag will be ignored, while the new GFW model will accept it.
- In Linux 2.6.34 and 2.4.37, when a connection is in ESTABLISHED state, an incoming packet without a set ACK flag will not be ignored. Instead, a data packet without the ACK flag will in fact be accepted. This indicates that such an insertion packet will not work against older Linux versions.
- In Linux 2.4.37, an incoming packet with an unsolicited MD5 header will not be ignored. This is due to the fact that older Linux versions have not implemented the feature proposed in RFC 2385 [63]. Upon closer inspection, the MD5 option check on the server can be turned off via kernel compilation options and therefore the corresponding insertion packet in fact may not always work.

This shows most insertion packets are applicable to a wide range of Linux operating systems, with some minor exceptions (if the encountered Linux version is too old). As Linux is dominant in the server market [102], we envision that evasion strategies built on top of these insertion packets will work well. Indeed, as we show in section 4.5, our GFW evasion success rate is extremely high if we are to leverage these insertion packets properly. To discover additional discrepancies and perform automatic "ignore path" analysis, we plan to use selective symbolic execution in the future (*e.g.,* S2E [37]). We leave a more rigorous analysis of TCP stacks of other Linux versions and operating systems, including closed-source OS like Windows Server, to our future work.

2.6 INTANG

All the strategies described in section 2.3 and section 2.4, are together integrated in a unified *measurement driven censorship evasion tool* we call INTANG. [3] The implementation contains roughly 3.3K lines of C code and some analysis scripts written in Python. INTANG is designed as an extensible framework that supports add-on strategies. The components of INTANG are depicted in Figure 2.2.

Overview. INTANG's functionalities are divided into three threads, viz., the main thread, the caching thread, and the DNS thread. The main thread is time-sensitive, and all time-consuming operations are pushed to the other two threads. The main thread runs a packet processing loop which intercepts certain packets using the netfilter queue [12] and injects insertion packets using raw sockets. While the packets are being processed, they are held in the queue *i.e.,* are not sent out until the processing is complete.

When a new connection is initiated, INTANG chooses the most promising strategy based on historical measurement results (with the help of caching), to a particular server IP address. Upon the completion of a successful trial, it caches the strategy ID along with the

[3]INTANG source code is publicly available at `https://github.com/seclab-ucr/INTANG`.

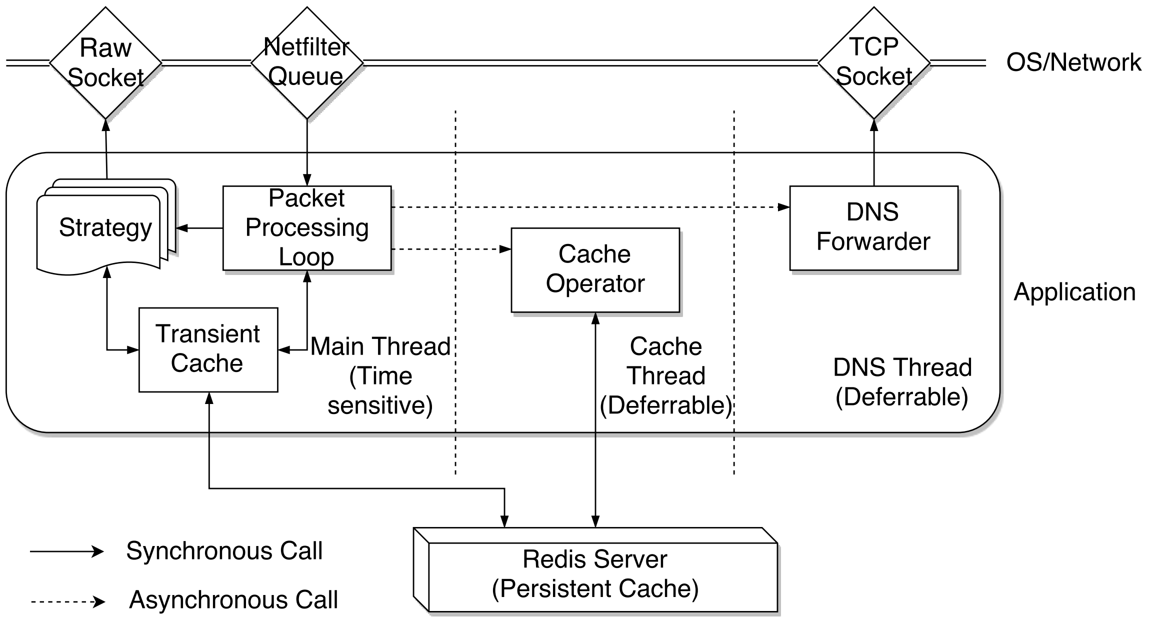


Figure 2.2: INTANG and its components

four-tuple of the connection in memory. When it later receives further packets associated with the four-tuple, it will invoke the callback functions of the strategy to process the incoming and outgoing packets. Usually, only a small set of specific packets (e.g. SYN/ACK packet, HTTP request) are relevant to each strategy and need monitoring (as discussed earlier).

DNS forwarder. The DNS thread is a specialized thread that aims at converting DNS requests over UDP to DNS requests over TCP. As mentioned in subsection 2.2.1, TCP-layer evasion not only helps with evading censorship on HTTP connections, but can also support the evasion of DNS poisoning by GFW. For this purpose, a simple DNS forwarder is integrated within INTANG. It converts each DNS over UDP request to a DNS TCP request and sends it to an unpolluted, public DNS resolver (likely outside of China). We

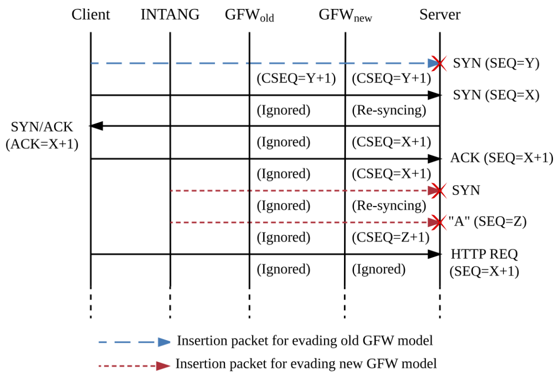


Figure 2.3: Combined Strategy: TCB Creation + Resync/Desync

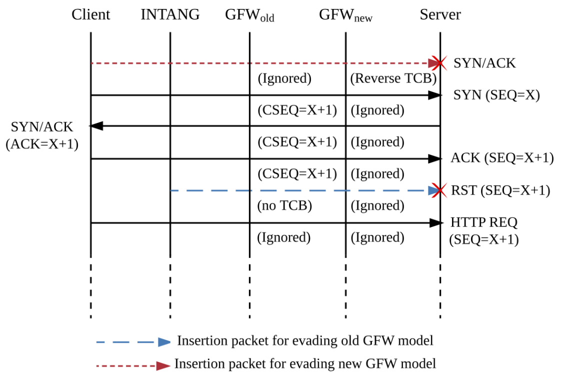


Figure 2.4: Combined Strategy: TCB Teardown + TCB Reversal

apply the same set of strategies for the TCP connection that carries DNS requests and responses, to prevent the GFW from resetting the connection upon detecting a censored domain in the request. The main thread intercepts outgoing DNS UDP requests, which may contain sensitive domain names and redirects such requests to the DNS thread that does the forwarding. When a DNS TCP response is received, it will be converted back to a DNS UDP response and processed normally by the application. So it is completely transparent to applications. We have probed GFW with Alexa's top 1 million domain names to generate a list of poisoned domain names using the same method as in [48].

Strategies. Each evasion strategy dictates specific interception points (*i.e.*, the types of packets to intercept) and the corresponding actions to take at each point (*e.g.*, inject an insertion packet). A new strategy can be derived from our suite of basic strategies by implementing new logic in the callback functions registered as interception points. A strategy can decide on whether to accept or to drop an intercepted packet, and can also modify the packet. It can craft and inject new packets as well.

Caches. INTANG employs Redis [95] as an in-memory key-value store. Redis provides desirable features like data persistency, event-driven programming, key expiration, etc. We also maintain in the main thread, a transient Least Recently Used (LRU) cache implemented using linked lists and hash tables (to reduce Redis store access latency that typically involves inter-thread or inter-process communications). Caching allows us to understand the effectiveness of the strategies against different websites and converge on the best one quickly. Of course, to counter changes in the network or the server, the cached record is retained only for a certain period of time before expiration. We omit the details of cache management in the interest of space.

2.7 Evaluation

We now extensively evaluate the hypothesized new behaviors of the GFW discussed in section 2.4 using the new strategies described in section 2.5 and our tool INTANG. We use the same 11 vantage points and 77 web servers as discussed in section 2.3; unless otherwise specified, all other measurement settings remain the same to ensure the consistency of the results. The experiments were conducted during April and May, 2017. In addition, since the GFW not only censors outbound traffic but also inbound traffic (both are client-to-server traffic),[4] we conduct measurements from 4 vantage points outside China, viz, in US, UK, Germany, and Japan, using instances on Amazon EC2, to targets inside China. This dataset includes top 33 Chinese websites chosen from the same Alexa's top 10,000 websites using the same method as in subsection 2.3.3 except they are inside China. By doing the

[4]A possible reason for doing this could be achieving bi-directional information barriers such as censoring what outsiders can see or restricting certain services, *e.g.,* VPN.

Table 2.4: Success rate of new strategies

Strategy (Outbound)	Success			Failure 1			Failure 2		
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Improved TCB Teardown	89.2%	98.2%	**95.8%**	1.7%	6.7%	**3.1%**	0.0%	5.4%	**1.1%**
Improved In-order Data Overlapping	86.7%	97.1%	**94.5%**	2.9%	8.9%	**4.4%**	0.0%	5.2%	**1.1%**
TCB Creation + Resync/Desync	88.5%	98.1%	**95.6%**	1.9%	7.0%	**3.3%**	0.0%	5.5%	**1.1%**
TCB Teardown + TCB Reversal	90.2%	98.2%	**96.2%**	1.7%	5.6%	**2.6%**	0.0%	5.7%	**1.1%**
INTANG Performance	93.7%	100.0%	**98.3%**	0.0%	3.0%	**0.9%**	0.0%	3.5%	**0.6%**

Strategy (Inbound)	Success			Failure 1			Failure 2		
	Min	Max	Avg.	Min	Max	Avg.	Min	Max	Avg.
Improved TCB Teardown	85.6%	92.9%	**89.8%**	4.6%	7.6%	**6.8%**	0.3%	6.8%	**3.5%**
Improved In-order Data Overlapping	89.4%	96.0%	**92.7%**	1.3%	6.2%	**3.6%**	0.6%	7.0%	**3.7%**
TCB Creation + Resync/Desync	78.1%	95.6%	**84.6%**	2.4%	18.6%	**12.9%**	0.9%	4.0%	**2.6%**
TCB Teardown + TCB Reversal	84.6%	93.1%	**89.5%**	5.5%	8.7%	**7.1%**	0.1%	7.9%	**3.3%**

bi-directional evaluation, we are in hope to examine if our new hypotheses/strategies work well for both directions and the implementations/policies of the GFW in both directions are the same.

2.7.1 Evading HTTP Censorship

There are 4 basic strategies that we evaluate in this subsection. These include two improved strategies based on previous strategies. These still worked but had high "Failure 1" and "Failure 2" rates. Specifically, they are *TCB Teardown with RST* and *In-order Data Overlapping.* The other two are new strategies viz., *Resync-Desync* and *TCB Reversal.* Note that these latter strategies explicitly leverage the new features that only exist in the evolved GFW model. We combine them with the aforementioned existing strategies that work for the old GFW model, in order to defeat both GFW models (*i.e.,* the objective is to defeat the GFW regardless of whether an old GFW model or an evolved model is encountered, or both).

Making old strategies robust. We make the *TCB Teardown with RST* strategy more robust by integrating within it, the sending of a "desynchronization packet" mentioned in Table 2.4. We send this desynchronization packet right after the RST packet(s) and before the legitimate HTTP request, to address the case wherein the GFW enters the "resynchronization state" due to the RST packets. We improve the reliability of the *In-order Data Overlapping* strategy by using more carefully chosen insertion packets to reduce potential interference from middleboxes, or because of hitting the server.

Accounting for both old and new GFW models. We combine the *Resync-Desync* strategy with the *TCB Creation with SYN* strategy. The latter can evade the old GFW model by causing the creation of a false TCB, while the former can desynchronize the evolved GFW model by forcing them into the resynchronization state first. Specifically, as illustrated in Figure 2.3, we will send two SYN insertion packets (both with wrong sequence numbers), one before the legitimate 3-way handshake and one after, and followed by a desynchronization packet and then the HTTP request. Note that the first SYN insertion packet followed by the legitimate SYN does also cause an evolved GFW to enter the resynchronization state; however, it is later resynchronized with SYN/ACK packet. We therefore need another SYN insertion packet after the handshake to cause the evolved GFW devices to "re-transition" into the resynchronization state.

We combine the *TCB Reversal* strategy with the *TCB Teardown with RST* strategy. Specifically, as shown in Figure 2.4, we first send a fake SYN/ACK packet from the client to the server to create a false TCB on the evolved GFW device. Next, we establish the legitimate 3-way handshake, which invalid with respect to the evolved GFW due to the existing TCB. Then we send a RST insertion packet to teardown the TCB on the old GFW model, followed by the HTTP request.

Avoiding interference from middleboxes or server. When crafting "insertion" packet, we choose the insertion packets wisely so as to not experience interference from the middleboxes, and not result in side-effects on the server. We primarily use TTL-based insertion packets since it is generally applicable. The key challenge here is to choose an accurate TTL value to hit the GFW, while not hitting server-side middleboxes or servers. We do that by first measuring the hop count from the client to the server using a way similar as *tcptraceroute*. Then, we subtract a small δ from the measured hop count, to try and prevent the insertion packet from reaching (hitting) the server-side middleboxes or the server. In our evaluation, we heuristically choose $\delta = 2$, but INTANG can iteratively change this to converge to a good value.

In addition, we exploit the new MD5 and old timestamp insertion packets, which allow the bypassing of the GFW without interfering with middelboxes or the server. Table 2.5 summarizes how we choose insertion packets for each type of TCP packet.

Table 2.5: Preferred construction of insertion packets

Packet Type	TTL	MD5	Bad ACK	Timestamp
SYN	✓			
RST	✓	✓		
Data	✓	✓	✓	✓

Results. We first analyze the results for individual evasion strategies. As seen from Table 2.4, the overall "Failure 2" rate is as low as 1% for all the strategies, which (a) show that our new strategies have a high success rate on the GFW which suggests that (b) our hypotheses with regards to the GFW evolution seem accurate.

We find that both the Failures 1 and Failures 2 always happen with regards to a few specific websites/IPs. One can presume that this is caused by some unknown GFW behavior or middlebox interference. However, since these cases are not sustained (are very rare), we argue that this is more likely to be due to middlebox interference.

Overall, we find that high Failure 1 rates is the major reason for overall low success rates. An introspective look suggests that because some servers/middleboxes accept packets regardless of the (wrong) ACK number or the presence of the MD5 option header, Failures 1 happen. Further, the TTL chosen is sometimes inaccurate due to (a) network dynamics or (b) hitting server-side middleboxes; this results in undesired side-effects that increase "Failures 1".

In addition, we find that for vantage points outside China, the TTL discrepancy unfortunately has a significant drawback. When accessing the servers in China, the GFW devices and the desired servers are usually within a few hops of each other (sometimes co-located). As a result it is extremely hard to converge to a TTL value for the insertion packet, that satisfies the requirement of hitting the GFW but not the server. As a consequence, in these scenarios, use of this discrepancy can cause either type of failures. We see from Table 2.4 that both the Failure 1 and Failure 2 rates are on average a bit higher than for the vantage points inside China.

Finally, because INTANG can choose the best strategy and insertion packets for each server IP based on historic results, we also evaluated INTANG performance in an additional row in Table 2.4 for vantage points inside China. It shows an average success rate of 98.3% which represents the performance with the optimal strategy specific to each website and network path. This is without further optimizing our implementation (*e.g.,* measuring packet losses and adjusting the level of redundancy for insertion packets).

Take away: While we do magnify the causes for failures, the biggest take away from this section is that our new hypothesized behaviors of the GFW seem to be fairly accurate, and that the new strategies are seemingly very effective in realizing the goal of evading the GFW, especially when the best strategies are chosen according to websites and network paths.

2.7.2 Evading TCP DNS Censorship

Table 2.6: Success rate of TCP DNS censorship evasion

DNS resolver	IP	except Tianjin	All
Dyn 1	216.146.35.35	98.6%	92.7%
Dyn 2	216.146.36.36	99.6%	93.1%

The GFW censors UDP DNS requests with DNS poisoning. It censors TCP DNS requests by injecting RST packets just like how it censors HTTP connections. Thus, our evasion strategies can also be used to help evade TCP DNS censorship. As discussed in section 2.6, INTANG converts UDP DNS requests into TCP DNS requests. To evaluate the effectiveness of our strategies on evading TCP DNS censorship, we use 2 public DNS resolvers from Dyn, and the same 11 vantage points in China. Google's DNS resolvers 8.8.8.8 and 8.8.4.4 have been IP hijacked by the GFW and thus cannot be used. By repeatedly requesting a censored domain, (*e.g.,* www.dropbox.com) 100 times, using the "improved TCB Teardown with RST strategy," we get the results shown in Table 2.6. The vantage points in Tianjin have low success rates of 38% and 24%. However, the others jointly yield success rates of over 99.5 %. Interestingly, we accidentally discover that if we use the TCP DNS through the two OpenDNS's DNS resolvers 208.67.222.222 and 208.67.220.220, even without applying INTANG we do not experience any censorship from any of our vantage points.

2.7.3 Tor and VPN

Tor is famous for supporting anonymous communications [91], and poses a serious threat to censorship. It is not surprising that it is reported that the GFW has been blocking Tor Bridge nodes through passive traffic analysis and active probing for more than 7 years [118]. Next, we examine if INTANG can help cover up Tor connections.

In our experiments, we first verify whether and how Tor nodes are blocked by the GFW. Subsequently, we test if INTANG can help clients from China evade Tor censorship.

We try to access hidden Tor bridge nodes setup on Amazon EC2 in the US from the same 11 vantage points (over 9 cities) (See section 2.3) inside China acting as Tor clients. Surprisingly, we find that there are four vantage points (in three cities Beijing, Zhangjiakou, and Qingdao) from which Tor connections to the hidden Tor bridge can operate without issues (as is) for over 2 days with periodic, manually generated traffic. Meanwhile, any hidden bridge nodes requested by the remaining 7 vantage points triggers active probing [49, 130] and are immediately blocked by the GFW, *i.e.,* any node in China can no longer connect to this IP via any port. This is very different from what was previously reported *i.e.,* the GFW only blocks the Tor port on that hidden bridge [130], and could cause collateral damage as the Amazon EC2 IPs are recycled. We test 5 different hidden bridge IPs and find no exceptions so far. The common characteristic of the first four locations is that they are all in Northern China. Thus, we speculate that Tor-filtering GFW nodes are most probably not encountered on the paths from this region.

Now, for the remaining vantage points where the Tor connections do trigger censorship blocking, we apply INTANG with the "improved TCB teardown strategy," five times each, and the success rate for the Tor connections is 100 %. We periodically repeat these experiments over a 9-hour period, and are able to keep using the Tor bridge node. This shows that: (a) our hypothesis that some of the GFW devices have evolved to a new model holds; and (b) INTANG is extremely effective in crafting the right measurement-driven strategy towards evading the GFW. We envision that Tor clients can even integrate INTANG in the future to improve its censorship evasion chances.

Similar to Tor, virtual private networks (VPN), which help users evade censorship, are also popular targets of the GFW. It is shown that there are multiple approaches used by the GFW to disconnect VPNs [133, 123]. They include DPI, IP address blocking, bandwidth throttling, *etc.* In November 2016, we set up an openvpn server in China, and used one node in America as a client. As per our experimental results, *a preliminary version of* INTANG helped openvpn over TCP evade censorship, while openvpn without INTANG was disconnected due to the client receiving a reset packet from the GFW during the handshake phase (the GFW seemingly used DPI). Unfortunately, we could not replay such experiments recently via either the PPTP protocol or with the openvpn protocol. Both protocols did not experience disconnections because of the GFW, nor did they suffer from rate limits imposed by the GFW. Unfortunately, we do not yet know what caused this change in behavior and we plan to continue monitoring the potential opportunity of applying INTANG to improve VPN stability.

2.8 Discussion

GFW Countermeasures. Our work is based on the latest developments of the GFW. It is certainly possible that GFW may undergo additional improvements to defeat our evasion strategies, and we acknowledge that it is an arms race. For instance, we demonstrate that GFW is more liberal in accepting RST packets than normal servers. It is possible that the censor may perform additional checks on the RST packets (*e.g.,* checksum and MD5 option fields) as a defense. But that may open up a new evasion attack on the GFW (*e.g.,* when the server does not check MD5 option fields). One can also leverage GFW's agnostic nature to network topology. For example, we can measure the exact TTL value to bypass the GFW while not to reach the server (although it is also a challenge to achieve accuracy and efficiency simultaneously).

Another potential improvement the GFW can make is to trust the data packet sent by the client only after seeing the server's ACK packet acknowledging the appropriate sequence number. However, this will greatly complicates the GFW's design and implementation.

In summary, we believe this is an arms race. As GFW evolves, so can the evasion strategies. We believe that the cost of rolling out new GFW models is quite high and such evolution will happen at the timescale of months (if not years), which leaves enough time for evasion strategy development (especially when tools like INTANG are leveraged). For instance, as soon as the GFW evolves, a new GFW model will be derived and subjected to the "ignore path" analysis, which can lead to the generation of new evasion strategies.

Complexity and (sometimes) inconsistency of the GFW. During our long-term study of the GFW since 2015, we have observed that type-1 and type-2 resets sometimes occur individually. For example, on certain days, from a vantage point in CERNET Beijing we could observe only type-1 resets, while on other days, both types were seen. Our observations indicate that the two types of GFW devices are usually deployed together, and sometimes one is down. Also, we found there were some rather intricate effects when the two types were working together. During a measurement in May 2016, we found the type-1 GFW devices also have a subsequent 90-second blocking period (which it normally does not) as the type-2 devices does, after we using our new strategy to evade type-2 devices. And when we used no strategies, only the type-2 reset can be observed (*i.e.,* type-1 devices are not enforcing the 90-second blocking period). It looked like the type-2 reset suppressed type-1 reset. This rare behavior is not observed during other measurements. Furthermore, in May 2016 and May 2017, we have observed that RST packets sometimes were unable to tear down the TCB on the GFW, with different pairs of controlled clients and servers. This inconsistent behavior could be due to load balancing among different versions of the GFW, or some intricate effects caused by several GFW devices deployed together. However, we have no way to obtain the ground truth. We acknowledge our measurements are largely limited by being agnostic to the interference among different versions of GFW devices (or even middleboxes) and to the way how they are deployed, in addition to the blackbox nature of the GFW device itself. We are interested in further exploring this complexity and inconsistency in our future work.

Combination of Strategies. The GFW is heterogeneous with different co-existing versions. As a result, as we did in this work, it is necessary to combine strategies that are effective against different versions of the GFW. This is normally not an issue as long as the strategies are not in conflict with each other. However, it is likely that the "Failure 1" rate will increase when a plurality of strategies are employed. This is because of the increase in insertion packets, which increases the likelihood of middlebox interference or side-effects on the server.

Ethical Considerations. All our experiments are carefully designed so as to not cause disruption to normal network operations. All connections are established from machines that we rent or control directly. The additional insertion packets are simply regular TCP packets (sometimes with incorrect field values) and may simply be discarded by the server. We control the traffic to each website to be low to avoid any unintended denial-of-service.

Note that INTANG doesn't guarentee unobservability for all its strategies. It is the user's discretion as to whether to use INTANG within the censor's jurisdiction. However, in China, due to heavy censorship [66], "crossing the wall" and accessing websites such as Google, Facebook, *etc.* has become a prevalent need. The censor usually punishes those who provide censorship circumvention services to the masses (*e.g.,* proxy/VPN providers) instead of punishing the users of the service. A client-side only tool like INTANG will be harder for the censor to trace and thwart.

2.9 Related Work

We have already alluded to various related efforts throughout the chapter (especially in section 2.2). They all focus on evaluating the censorship techniques *or* anti-censorship techniques aided by additional facilities, like VPN.

Clayton et al., propose to ignore the RST packets sent by the GFW [40]. This requires cooperation from the server-side, and is thus impractical (all servers will need to install a patch to do that). It does not prevent the censor from monitoring user traffic. Thus, we do not explicitly consider this in our work. As discussed earlier, Ptacek et al. [92], develop a deep understanding of the vulnerabilities of current NIDS, which has largely influenced later efforts (including ours) on TCP reset attack evasion. The West Chamber Project [101] is a censorship-circumvention tool that implemented the Ptacek et al.'s theory. However, it just uses two kinds of crafted packets to teardown the TCB on the GFW from both directions, and has now become ineffective.

Khattak et al.'s research [68] is the most relevant work to ours. Their strategies, and the problems thereof were already discussed in section 2.3. In addition, our measurement utilizes multiple vantage points instead of one vantage point as in [68]. Our measurement study leads to the discovery of the differences in deployment and features of the GFW from what was presented in that work. Li et al. [74] tested known TCP/IP insertion packets against censorship firewalls and DPI boxes in three countries and evaluated their effectiveness. In contrast, our work focuses on understanding and uncovering the latest development (new state machine) of the largest and most complex censorship system, which allows us to devise new evasion strategies.

2.10 Conclusion

In this chapter we undertake, arguably, the most in depth measurement study of stateful (TCP-level) Internet censorship evasion on the GFW of China. Our work is divided into multiple stages. First, we perform extensive measurements of prior approaches and find that they are no longer effective. We attribute the reasons for this to two primary causes: (a) the GFW has evolved to imbibe new behaviors and, (b) the presence of middleboxes on the path between the client and the server that can interfere with the evasion strategies. Second, based on the knowledge gained, we hypothesize about new GFW behaviors and design new strategies that can possibly evade GFW today. We also build a novel, measurement driven tool INTANG that can converge on the right evasion strategy for a given client server pair. In the final stage, we perform extensive measurements of our new strategies and INTANG, and demonstrate that they provide near-to-perfect evasion rates when combined, thereby validating our new understanding of the GFW's stateful censorship model of today.

Chapter 3

SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery

3.1 Introduction

Deep packet inspection (DPI) has become a technology commonly deployed in modern network security infrastructures. By assembling and checking application layer content, DPI enables powerful functionalities that are not present in traditional firewalls. These include malware detection [22], remote exploit prevention [106], phishing attack detection [36], data leakage prevention [116], government network surveillance [16, 15], targeted advertising [72, 11], and traffic differentiation for tiered services [137, 81, 51].

Unfortunately, to assemble application layer content from stateful protocols like

TCP, DPI needs to engineer the corresponding state machine of the protocol. This introduces a fundamental limitation of DPI, which is a susceptibility to *protocol ambiguities*. In brief, most network protocol specifications (e.g., RFCs for TCP [90]) are written in a natural language (English), which makes them inherently ambiguous. To make things worse, some parts of the specifications are deliberately left unspecified, which in turn leads to *vendor-specific* implementations. Consequently, different network stack implementations (e.g., Windows and Linux) typically have inherent discrepancies in their state machines [107, 29, 93]. In fact, even different versions of the same network stack implementation, can have discrepancies. To ensure low overheads and compatibility with most implementations, DPI middleboxes usually implement their own simplified state machines, which are bound to differ from the ones on endhosts.

As pointed out by previous works [92, 125, 74], such discrepancies lead to certain network packets being accepted/dropped by either a "DPI middlebox" or the endhost. Exploiting this property, one can use *insertion* packets (i.e., a packet which is accepted and acted upon by the DPI middlebox to change its state, whereas the remote host drops/ignores it) and *evasion* packets (i.e., a packet which is ignored by the DPI middlebox but the remote host accepts and acts on it) [92] to mislead the DPI's protocol state machine. Specifically, such packets cause the DPI to enter a different state than the one on the endhost. Consequently, the DPI can no longer faithfully assemble the same application layer content as the endhost, failing to catch any malicious or sensitive payload.

To date, research on *insertion* and *evasion* packets are based on manually crafting such packets targeting specific DPI middleboxes [92, 125, 74]. Unfortunately, it is a labor-intensive task to analyze each and every middlebox implementation and come up with the corresponding strategies for such adversarial packet generation. One can potentially automate the process by searching through all possible sequences of packets to identify *insertion* and *evasion* packets. Unfortunately, the search space is exponentially large, i.e., there are 2^{160} possibilities to cover a 20-byte TCP header of even a single packet, let alone testing a sequence of packets.

"Can we develop automated ways to construct packets that can successfully de-synchronize the state of a DPI middlebox from that of a (end) server?" This question is at the crux of the work we target in this chapter, answering which not only can help test future generations of DPIs but also help stay on top of the arms race against future censorship technologies. Our focus here is on TCP, since it is the cornerstone upon which most popular application-layer protocols are built. We develop an approach that is driven by the insight that even though the TCP state machines of DPI middleboxes are obscure, the implementations of TCP on the endhosts are well established (e.g., a very large fraction of the servers run Linux operating systems). Given this, we explore the TCP state machine of endhosts (using symbolic execution) and generate groups of candidate packets based on what critical points and states they can reach, i.e., states where packets are either accepted or dropped/ignored due to various reasons. Next, we perform differential testing by feeding such packets through the DPI middlebox and observe whether they induce any discrepancies, i.e., whether the DPI middlebox can still perform its intended function of identifying connections that contain malicious/sensitive payloads.

The major contributions of the work are the following:

- We formulate the problem of automatically identifying *insertion* and *evasion* packets by focusing on exploring the TCP state machine on endhosts, and conducting differential testing against blackbox DPIs.
- We develop SymTCP, a complete end-to-end approach to automatically discover discrepancies between any TCP implementation (currently Linux) and a blackbox DPI. We have released the source code of SymTCP and datasets at `https://github.com/seclab-ucr/sym-tcp`.
- We evaluate our approach against three DPI middleboxes, Zeek, Snort, and Great Firewall of China (GFW), and automatically find numerous evasion opportunities (several are never reported in the literature). The system can extend to other DPIs easily and serves as a useful testing tool against future implementations of DPIs.

3.2 Background

In this section, we first provide a brief background on why eluding attacks are possible against DPI. Subsequently, we provide some background on symbolic execution and associated techniques since these are integral to building SymTCP.

3.2.1 Eluding Attacks against Deep Packet Inspection

DPI is specially designed to examine content related to higher-layers, such as the application layer (e.g., HTTP, IMAP). To examine application-layer payloads, DPI first reconstructs data streams from network packets (TCP packets) captured from an interface. Then it automatically assigns an appropriate protocol parser to parse the raw data stream [47]. Finally, it performs "pattern matching" on the parsed output. To illustrate as an example, consider the common case of keyword-based filtering of HTTP requests (e.g., deployed on censorship firewalls). When the DPI module (referred to as simply DPI for ease of exposition) detects a specific keyword in the HTTP URI, it may take follow-up actions (e.g., blocking the connection or silently recording the behavior). Sometimes the pattern matching signatures can be more complex, wherein the DPI examines a combination of fields from multiple layers and data from both directions (to and from a server) in a sequence [110]. For example, one endhost first sends a "HELLO" message to port 443, and then the other party responds with an "OLLEH" message.

However, DPI suffers from the inherent vulnerability of evasion because of discrepancies between its TCP implementation and that of the endhost (e.g., a server) arising because of protocol ambiguities [92, 61]. An example is that Snort [109] accepts a TCP RST packet as long as its sequence number is within the receive window (which is too lenient), while the latest Linux implementation will make sure that the sequence number of the RST packet matches the next expected number (`rcv_next`) exactly. This allows an attacker to send an *insertion* RST packet with an intentionally marked "bad" in-window sequence number, which terminates the connection on Snort, whereas the remote host will actually drop/ignore such a packet. Such discrepancies open up a gap for attackers to elude the DPI by sending carefully crafted packets.

Besides discrepancies due to protocol implementations, lack of knowledge of the network topology could also introduce additional ambiguities. For example, it is hard for a DPI to infer whether a packet will reach the destination. Thus, the attacker can send a packet with a smaller TTL to cause it not to reach the remote host, however, such a packet has an influence on the DPI.

Previous research works [125, 74] have exploited the network ambiguities and protocol implementation discrepancies to design evasion strategies against real-world DPI systems, such as the national censorship systems in China and Iran, and ISPs' traffic differentiation systems for tiered services. Those evasion strategies are shown to have high success rates in rendering the DPI ineffective. However, most of the common discrepancies can be patched by the DPI devices, leading to an arms race. In contrast, our system presents a major step towards automating the evasion strategies, which not only can serve as a valuable testing tool against future generations of DPIs but also keep pace in the escalating arms race in the context of DPI evasion.

3.2.2 Symbolic Execution vs. Concolic Execution vs. Selective Symbolic Execution

Symbolic execution [70] is a powerful and precise software analysis/testing technique that is widely employed for its ability to break through complex and tight branch conditions and reach deeper along execution paths, which is a distinct advantage compared to other less precise techniques such as fuzzing. In symbolic execution, instead of using concrete values, variables are assigned symbolic values to explore the execution space of a target program. The symbolic execution engine simulates the program execution by interpreting each instruction (either at an intermediate representation level like LLVM-IR [26] or VEX [105], or at the binary level [135]), and maintain symbolic expressions for each program variable. En route, the engine collects path constraints in the form of symbolic expressions. Whenever a branch with a symbolic predicate is encountered, the engine checks whether the corresponding true/false path is satisfiable (with the help of an SMT solver); if so, it forks the execution path into two and adds a new path constraint according to the branch condition (`true` or `false`). The disadvantage of symbolic execution, however, is in its efficiency or scalability. Both simulated execution and constraint solving can be extremely slow even with optimizations such as caching and incremental solving [26]. Moreover, the total number of feasible execution paths in a common size modern software can be huge, leading to the notorious path explosion problem.

Concolic execution [27] is a practical testing technique that enhances symbolic execution with concrete execution. The basic idea is to bind a concrete value to each symbolic expression, and so, it can switch modes between symbolic execution and concrete execution at any time. When a branch with symbolic predicate is encountered, the concolic execution engine first uses the concrete value to decide which path to go; subsequently, it also tries to generate a new concrete value for the opposite branch. When a particular part of the code or a function may cause path explosion or if the constraint solver is unable to or inefficient in solving, it can switch to concrete execution which prevents forking and constraint solving, and switch back at a later time. However, this may cause a loss in terms of both completeness and soundness as a trade-off [13]. Most of the state-of-the-art symbolic execution engines like Angr [105] and S2E [37] support concolic execution.

Selective symbolic execution [37] further extends the idea of concolic execution and makes it more flexible and practical for testing large and complex software (like an operating system kernel). In particular, a selective symbolic execution engine allows the testing of only a sub-system of a program (e.g., the TCP implementation). This is achieved by transitioning between the concrete mode (where most symbolic variables already have concrete values) and the symbolic mode as follows:

- Transition from concrete to symbolic: the engine symbolizes the inputs of the scope (data coming into the scope), such as function parameters, to offer the possibility of exploring all execution paths within the scope at the cost of *under-constraining*, i.e., losing additional constraints imposed over the inputs from external components.
- Transition from symbolic to concrete: the engine concretizes symbolic variables, which can cause *over-constraining* as we are arbitrarily choosing one of the possible values to assign to any symbolic variable and this can harm completeness.

S2E [37] is a representative system that combines selective symbolic execution with whole-system emulation to test the Linux kernel. Its performance of symbolic execution is controlled by selectively running part of the code of interest (e.g., specific functions) in symbolic mode while keeping most other parts and the external system running in the concrete mode. S2E provides different levels of execution consistencies that allow trade-offs between performance, completeness, and soundness of analyses.

In our solution, to address the complexity of real-world TCP implementations, we employ the selective symbolic execution feature in S2E to effectively explore the TCP implementation in the Linux kernel.

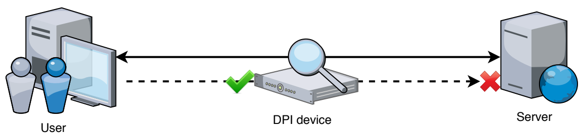


Figure 3.1: Threat Model of SymTCP

3.3 Threat Model and Problem Definition

In this section, we first describe our threat model. Subsequently, we formalize the problem that we set out to solve when we design SymTCP.

3.3.1 Threat Model

The threat model that we consider is depicted as in Figure 3.1. We assume that a DPI engine is located in between the client and the server, and is capable of reading all the packets exchanged between the client and the server. We only focus on the TCP protocol in this work since it is arguably the most popular transport layer protocol. By eluding DPI from the TCP-layer, we can disrupt TCP packet reassembly of the DPI, and therefore can allow upper-layer protocols to elude DPI (e.g., HTTP, HTTPS).

We assume that the DPI engine has its own TCP implementation that can reassemble and cast the captured IP packets into TCP data streams. It then performs checks on the reassembled data streams for whatever is needed based on the function of the middlebox (e.g., censorship, network intrusion detection, etc.), and its behavior is deterministic. We also assume that the inspections will lead to observable effects, e.g., blocking or resetting of a connection, if an alarm is triggered; otherwise we cannot tell whether an eluding attack is successful or not.

The goal of a host (e.g., client) is to elude inspection of the DPI engine, by sending carefully crafted packets that exploit discrepancies between the TCP implementation of a DPI and that of the host on the other end (e.g., server), prior to sending the sensitive content. For ease of discussion, throughout the rest of the chapter, we consider the client to be the one attempting to elude the inspection unless otherwise explicitly stated. We consider the DPI's TCP implementation to be a blackbox, and thus, the client can send only probe packets. The responses (or lack thereof) to the probe packets allows the client to infer the state of DPI's TCP state machine. We assume that the server uses a publicly available TCP stack implementation (e.g., Linux), and thus, the client can perform analysis as a whitebox. These assumptions also imply that the server is not colluding with the client by using a specialized or custom TCP stack as otherwise arbitrary covert channels can be established [82].

3.3.2 Problem Definition

Conceptually, an *evasion* packet is a TCP packet that is accepted by the server but dropped/ignored by the DPI engine. Similarly, an *insertion* packet is a TCP packet that is dropped by the server but accepted by the DPI engine. However, such a definition is imprecise. In this section, we aim to provide a more precise definition of the concepts we use in this work, as well as the problem that SymTCP solves. First, we define what are *accept* and *drop* attributes associated with a packet.

TCP State Machine. Conceptually, each TCP implementation can be modeled as a deterministic Mealy machine, $M = (Q, q_0, \Sigma, \Lambda, T, G)$ where

- Q is the set of states,

- $q_0 \in Q$ is the initial state,
- Σ is the input alphabet, i.e., a TCP packet,
- Λ is the output alphabet, i.e., the TCP data payload,
- $T : Q \times \Sigma \to Q$ is the state transition function, and
- $G : Q \times \Sigma \to \Lambda$ is the output function.

Compared with a traditional deterministic finite state machine, the output of the Mealy machine is determined by both its current state and the current inputs. Note that in this work, we define the output of a TCP state machine M as the *output to the buffer that stores data which will be used by the application layer* (i.e., payload), instead of the response packet. The reasons are that (1) DPI's detection of sensitive keywords is strictly on the application layer payload, and (2) the TCP layer of the DPI engine will not generate any TCP level output like ACK packets. This model allows us to unify the definition of state machines for both the DPI and an endhost. We also simplify the output behavior as follows: as long as the data payload will be output to the application layer, even in a delayed manner, we consider that the packet generates a non-empty output.

Definition 1: Drop. Given a TCP state machine M, a packet $P \in \Sigma$ is *dropped* if it neither causes a state change nor generates any output. Here the state can be either the high-level TCP states (e.g., `LISTEN`, `ESTABLISHED`), or low-level/implementation-level states (e.g., the number of challenge ACKs that have been sent [29]).

$$T(q, P) = q \wedge G(q, P) = \varepsilon \tag{3.1}$$

Correspondingly, we define *drop paths* as the program paths of a TCP implementation that free an incoming TCP packet without changing the current state of a TCP session or producing any output. To identify *drop paths* in practice, we also define *drop points* as the program points or statements where any path that traverses it would become a *drop path*. In the Linux kernel we analyzed, we manually labeled 38 unique *drop points* in total (more details in section 3.8). Note that a single *drop point* may correspond to many different packet instances. For example, a packet with "bad checksum" can have arbitrary SEQ or ACK numbers, as well as arbitrary TCP headers.

Definition 2: Accept. Given a TCP state machine M, a packet $P \in \Sigma$ is *accepted* if it causes a state change (including both a high-level, TCP state change and a low-level, implementation-specific state change) or the output is not empty:

$$T(q, P) \neq q \vee G(q, P) \neq \varepsilon \tag{3.2}$$

Correspondingly, we define *accept paths* as the program paths of a TCP implementation that change the current state of a TCP session or append the payload of a TCP packet to the receive buffer. Technically, all paths that are not *drop paths* are considered *accept paths*; equivalently, any path that does not traverse any *drop point* is considered an *accept path*, and can be therefore be identified automatically.

Next, we note that any evasion or insertion packet needs to be sent along with other packets in a sequence (e.g., the TCP handshake, a data packet that contains sensitive keyword), in order to discover discrepancies. For ease of exposition, we first define two shortcut functions for handling a sequence of packets.

Let M_s be the TCP state machine of the server and M_d be the TCP state machine of the DPI engine. For simplicity, we assume M_s and M_d have the same input and output alphabet. Although the set of states of M_s and M_d are different, we assume that their initial states (q_0) are the same, i.e., the `LISTEN` state. Given a state q of a TCP state machine M and a sequence of packets $P_{1...n} \in \Sigma^*$, we denote $\mathcal{T}_M(q, P_{1...n})$ as the state transition from q after handling $P_{1...n}$, and $\mathcal{G}_M(q, P_{1...n})$ as the generated TCP data stream to the application layer.

Because the goal of the DPI's TCP layer is to extract the *data stream* from the monitored TCP session between the client and the server, we define the concept of "synchronized" for the ease of discussion.

Definition 3: Synchronized. Given a sequence of packets $P_{1...n} \in \Sigma^*$, we say that the DPI engine's TCP state machine M_d is *synchronized* with the server's state machine M_s if and only if the generated (application) data streams from the initial `LISTEN` state are the same for both i.e.,

$$\mathcal{G}_{M_s}(q_0, P_{1...n}) = \mathcal{G}_{M_d}(q_0, P_{1...n}) \tag{3.3}$$

At a high-level, what insertion and evasion packets aim to achieve is to "de-synchronize" the TCP state machine of the server (M_s) from that of the DPI engine (M_d),[1] so that the payload with sensitive information will not be output to the application layer filters for inspection. However, because the DPI engine is a black box in our threat model, whether the two state machines have been de-synchronized can only be inferred from the behavior of application layer filters (e.g., the decision to block or reset a connection after

[1]It is also possible that a packet can be accepted differently, exerting different effects on the server and DPI; we do find such cases in practice.

sending a probe packet). To model such behaviors, we define an abstracted filter function.

Definition 4: Bad Keywords and Alarm. For simplicity, we use *bad keywords* to represent any content that can trigger an alarm, and we assume that the entire content fits into a single TCP packet for the ease of discussion (but we can also support keywords which are split into multiple packets). Given a packet P containing a bad keyword, a filter function $F : \Lambda \rightarrow \{0, 1\}$ performs arbitrary checks over its data payload.

$$F(G(q,P)) = \begin{cases} 1 & \text{if G(q,P) contains any bad keyword} \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

The function applies to both DPIs and servers.

Definition 5: Evasion Packet. Given a sequence of packets $P_{1...n} \in \Sigma^*$, we say that the last packet P_n is an *evasion packet* if the following three requirements are satisfied. ① The server will accept every packet $P_{1...n}$ (Equation 3.2). ② When handling $P_{1...n-1}$, the state machine of the server and the DPI engine are synchronized (Equation 3.3). ③ Once P_n is sent, the two state machines would be "de-synchronized" as the DPI engine will drop P_n (Equation 3.1) and thus fail to output the payload of P_n or its follow-up packets (as P_n itself may not be a data packet). Let P_{n+r} be the data packet that contains the bad keywords ($r = 0, 1, ...$), we have:

$$G_{M_s}(\mathcal{T}_{M_s}(q_0, P_{1...n+r-1}), P_{n+r}) \neq \varepsilon \wedge$$
$$G_{M_d}(\mathcal{T}_{M_d}(q_0, P_{1...n+r-1}), P_{n+r}) = \varepsilon$$

Unfortunately, as mentioned above, we can only indirectly infer whether the G_{M_d} output is empty by means of the filtering function F. Given this, we use P_{n+r} as the probe packet with bad keywords in the payload, and change the requirement ③ to:

$$F(G_{M_s}(\mathcal{T}_{M_s}(q_0, P_{1\ldots n+r-1}), P_{n+r})) = 1 \wedge$$
$$F(G_{M_d}(\mathcal{T}_{M_d}(q_0, P_{1\ldots n+r-1}), P_{n+r})) = 0 \tag{3.5}$$

Note that our definition of evasion is purely based on the outputs to the application layer and thus, is more strict. Specifically, $P_{1\ldots n-1}$ may already have triggered discrepancies between M_s and M_d (they are accepted and processed differently on the DPI and server); however, without triggering observable behavioral changes at the application layer, we cannot ascertain that such packet(s) are evasion packet(s). Note that the requirement ② and ③ together explicitly exclude the cases that $P_{1\ldots n-1}$ already ends with an evasion or insertion packet.

Definition 6: Insertion Packet. Given a sequence of packets $P_{1\ldots n} \in \Sigma^*$, we say that the last packet P_n is an *insertion packet* if the following three requirements are satisfied. ① The server will accept every packet $P_{1\ldots n-1}$ but will drop P_n (Equation 3.1). ② When handling $P_{1\ldots n-1}$, the state machine of the server and the DPI engine are synchronized (Equation 3.3). ③ P_n will "de-synchronize" the two state machines as the DPI will accept P_n (Equation 3.2), which has to be inferred through some follow-up probe packets $P_{n+1\ldots n+r}$ where the last packet P_{n+r} contains bad keywords ($r = 1, 2, \ldots$) (same as Equation 5). $P_{n+1\ldots n+r-1}$ are needed for the purpose of reaching the `ESTABLISHED` state.

Goal. Given the above definitions, the goal of SymTCP is to *automatically* find packet sequences $P_{1\ldots n}$ where the last packet P_n is an evasion/insertion packet.

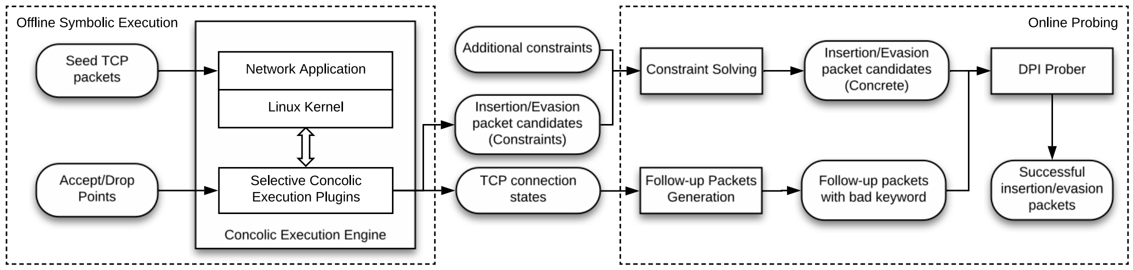

Offline Symbolic Execution
Seed TCP packets
Accept/Drop Points
Network Application
Linux Kernel
Selective Concolic Execution Plugins
Concolic Execution Engine
Additional constraints
Insertion/Evasion packet candidates (Constraints)
TCP connection states
Online Probing
Constraint Solving
Insertion/Evasion packet candidates (Concrete)
DPI Prober
Follow-up Packets Generation
Follow-up packets with bad keyword
Successful insertion/evasion packets

Figure 3.2: Overview of SymTCP's Workflow

3.4 Workflow of SymTCP

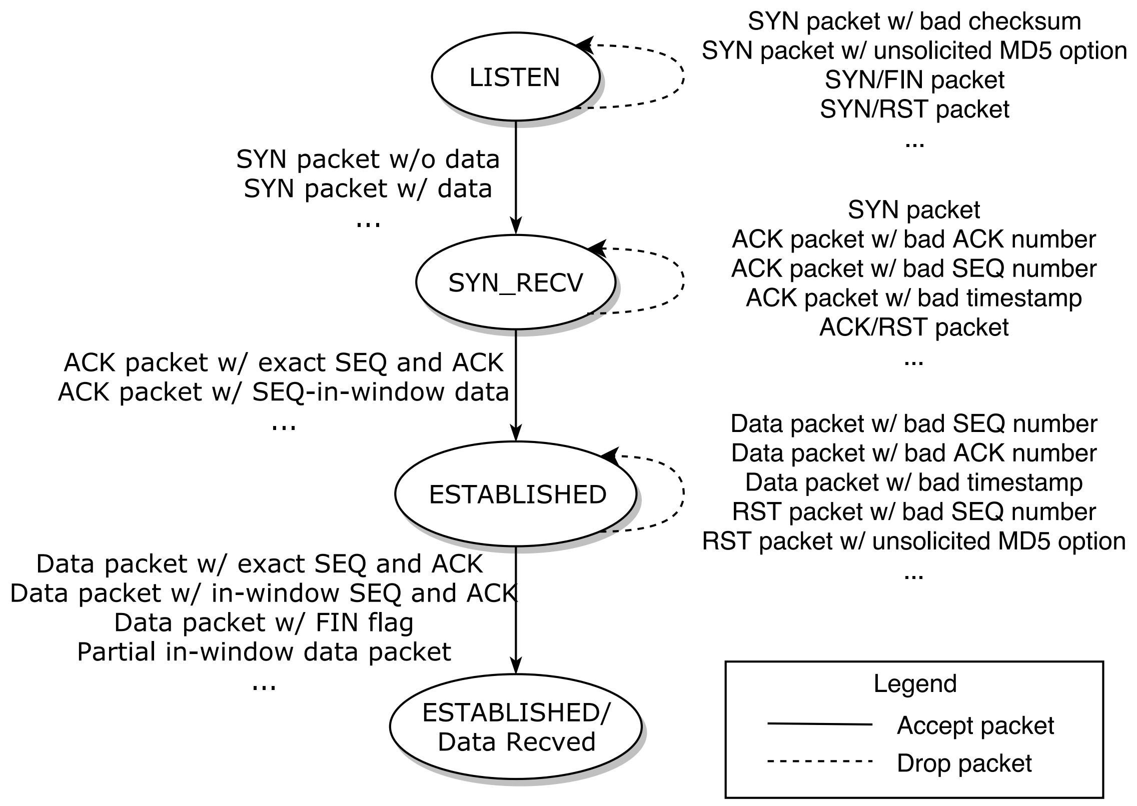
An overview of SymTCP's workflow is depicted in Figure 3.2. The workflow is divided into an offline selective concolic execution phase and an online testing phase. The inputs of the offline phase include a set of initial seed TCP packets (e.g., initial SYN) that can drive the concolic execution engine, and a manually curated list of accept and drop points of a Linux TCP implementation (as defined earlier).

During the offline phase, by running concolic execution on the server's TCP implementation, we attempt to gather all execution paths (if possible) that reach an accept or a drop point (as defined in subsection 3.3.2) at different TCP states and collect the corresponding path constraints. Each path corresponds to a packet sequence $P_{1...n}$ and the collected path constraints are later used to generate concrete test packets for differential testing, i.e., serving as candidate insertion/evasion packets.

Figure 3.3 illustrates some example packets that reach drop points (Equation 3.1: the packets do not have any effect and are simply discarded and optionally ACKed) *and* some example packets that reach accept points (Equation 3.2: advancing the TCP state machine or causing data to be accepted). Note that our analysis will always start from the

TCP `LISTEN` state and end with the TCP `ESTABLISHED` state as it represents the complete window of opportunity to inject insertion/evasion packets. For instance, it has been reported in [125] that if a client sends a SYN-ACK to a server in the `LISTEN` state, the server will drop the packet (and send a RST) whereas the Great Firewall of China (GFW) will be confused into thinking that the client is the server. Such a SYN-ACK packet is effectively an insertion packet that allows the client to then move on with the normal three-way handshake and start sending data unchecked (Definition 6). Another example is a SYN packet containing a data payload, which is allowed by the TCP standard (the payload will be buffered until the completion of the three-way handshake), but a DPI may incorrectly ignore it [92], making this packet an evasion packet (Equation 3.5). We do not wish to advance the server's state beyond `ESTABLISHED` (e.g., `TIME_WAIT`) because we can then no longer deliver data.

Offline phase: In brief, the offline concolic execution engine first boots a running Linux kernel with a TCP socket in the `LISTEN` state. Then we feed it with multiple symbolized packets to explore the server's TCP state machine as exhaustively as possible. The primary output of this phase is the sequence of candidate insertion/evasion packets in the form of symbolic formulas and symbolic constraints that describe what possible values the TCP header fields should take (including the constraints that describe the inter-relationships between packets). Note that each packet sequence will contain at most one packet that reaches a drop point. This is because each such a "drop packet" by itself does not impact the TCP state machine whatsoever; thus, a sequence with two (successive) "drop packets" is equivalent to two sequences each with a single "drop packet" (i.e., splitting the original sequence). The shorter sequences are discovered first with the symbolic execution



LISTEN
SYN packet w/ bad checksum
SYN packet w/ unsolicited MD5 option
SYN/FIN packet
SYN/RST packet
...
SYN packet w/o data
SYN packet w/ data
...
SYN_RECV
SYN packet
ACK packet w/ bad ACK number
ACK packet w/ bad SEQ number
ACK packet w/ bad timestamp
ACK/RST packet
...
ACK packet w/ exact SEQ and ACK
ACK packet w/ SEQ-in-window data
...
ESTABLISHED
Data packet w/ bad SEQ number
Data packet w/ bad ACK number
Data packet w/ bad timestamp
RST packet w/ bad SEQ number
RST packet w/ unsolicited MD5 option
...
Data packet w/ exact SEQ and ACK
Data packet w/ in-window SEQ and ACK
Data packet w/ FIN flag
Partial in-window data packet
...
ESTABLISHED/ Data Recved
Legend
Accept packet
Drop packet

Examples: { P_{ic}(SYN/bad checksum) }
{ P_{ec}(SYN/data), P_{ic}(ACK/bad ACK number) }
{ P_{ec}(SYN/no data), P_{ec}(ACK/SEQ-in-window data), P_{ic}(Data/unsolicited MD5 option) }

P_{ic} denotes candidate insertion packet, P_{ec} denotes candidate evasion packet

Figure 3.3: Candidate packet generation with symbolic execution

engine—we use a strategy similar to breadth-first search to discover sequences of packets and limit the total number of symbolic packets to be practical (more details in section 3.5); thus, the longer sequence containing multiple drop packets is unnecessary and redundant. In contrast, different paths reaching the same accept/drop point are not redundant and can represent distinct events. For instance, as shown in Figure 3.3, if the current TCP state is `SYN_RECV`, one can send two types of ACK packets to advance the TCP state to `ESTABLISHED` (both lead to the same accept point): (1) an ACK packet with a 0-byte of payload (where the SEQ and ACK number match exactly what are expected), or (2) an ACK packet with an in-window payload (as long as the END_SEQ is greater than the expected SEQ number). They correspond to two different accept paths that represent two distinct ways of moving the TCP state forward. Discovering these different paths is critical as not all paths are handled equivalently by the DPI (thus leading to possible evasion opportunities).

An additional output of the offline symbolic execution engine (as shown in Figure 3.2) is that for each sequence of candidate packets, there is a corresponding TCP connection state that the server will end up in after the sequence of packets is consumed. Recording this information facilitates the generation of follow-up probe packets. For example, if the sequence of candidate packets is a single TCP SYN with a bad checksum, then we know that the server will stay in the `LISTEN` state; therefore a proper three-way handshake is needed before we can send a data packet to check if the DPI was confused by the initial candidate insertion packet.

Online phase: During the online phase, we attempt to concretize these candidate insertion/evasion packets by adding additional constraints (more details to follow in section 3.6). One such constraint is the *server's* initial sequence number (which is randomly generated every time we probe the server). Once the constraint solver generates the sequence of concrete candidate insertion/evasion packets, they are fed to the DPI prober (together with the follow-up packets).

We illustrate the process in Figure 3.4. For each sequence of packets, we start from the first packet and perform probes according to the current packet. If the current packet reaches a drop point, we will treat it as a candidate insertion packet and probe the

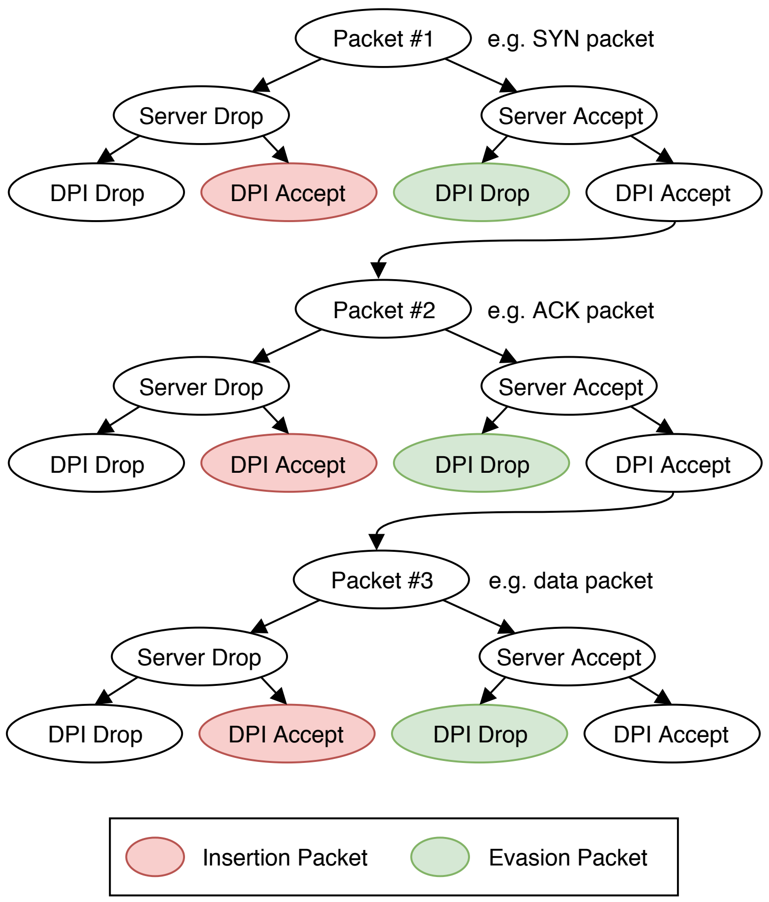


Figure 3.4: Evaluation of insertion/evasion packet candidates

DPI to see whether it causes the DPI to later ignore the data packet with a known bad payload (Definition 6). For example, a SYN packet with bad checksum will be considered a candidate insertion packet (while the server is in the `LISTEN` state). If the current packet is one that reaches an accept point such as a SYN packet with data (as in the example mentioned earlier), we will feed it to the DPI and observe whether it qualifies as an evasion packet (Equation 3.5). If the DPI accepts the packet just as the server (which is the common case as a DPI typically is lenient in accepting packets [92]), we will move on to the next packet and repeat the process. Note that for different sequences of packets that share the common prefix packets, we only need to evaluate the common packets once (as candidate insertion or evasion packets).

3.5 The Offline Phase: Practical Concolic Execution on the TCP implementation

Our solution is built on top of the popular concolic execution engine S2E [37] that is capable of analyzing OS kernels. The challenge is that a full-size TCP implementation has a rather complicated finite state machine (especially with the low-level states). Thus, applying concolic execution on the same is extremely challenging. We describe how we tackle the more detailed challenges in this section. Specifically, in subsection 3.5.1, we describe how we employ selective concolic execution to bound the symbolic execution space. In subsection 3.5.2, we describe how we symbolize the input, i.e., the fields in the TCP header and options. In subsection 3.5.3, we discuss how we abstract checksum functions in TCP. Finally, in subsection 3.5.4, we discuss how to deal with server-side inputs (specifically, the sequence number used by the server) that are not known a priori.

3.5.1 Selective Concolic Execution Favoring Completeness

Because it is heavyweight, we want to run symbolic execution only on the TCP code base; for the rest of the system, we seek to use concrete execution to reduce complexity. To realize this vision, we need to define the boundary between where symbolic execution and concrete execution are applied. One way to achieve this is to perform a fine-grained, function-level analysis to identify those functions that are related to the TCP logic, but this will require a prohibitively expensive manual effort. To solve this problem, we use a more conservative, coarse-grained boundary, which is the entire net/ipv4 compilation unit (object file) in Linux. When we are inside the address space of the net/ipv4 compilation unit, we run the code with symbolic execution and enable forking. When we are outside this address space, we run the code concretely with forking disabled, but still keep the original constraints (as is supported by S2E). The benefit of this is that we do not lose the symbolic expressions when switching back from the concrete mode to the symbolic mode. S2E also maintains a concrete value for each symbolic variable and these will be used during concrete execution. The concrete values are generated by constraint solving at the first time they are accessed in the concrete execution. We emphasize that this is different from applying pure concrete execution from the beginning; switching from the symbolic to the concrete mode still retains the symbolic variables and propagates them during concrete execution.

By default, even when running in the concrete mode, S2E collects path constraints as the concrete branches are taken (standard in concolic execution [13]). The reason for doing so is that during concrete execution, only one branch is taken, and the result is bound to that branch. However, this will result in the previously discussed "over-constraining" problem (in §4.2), i.e., forcing certain branches to be taken (because of the concretization when switching to concrete execution). More importantly, our focus is on the TCP code base only, and the executions outside of our scope are irrelevant (regardless of which paths were taken). We therefore discard any constraints collected during the concrete execution mode. For example, the netfilter module outside the TCP code base will read the symbolic TCP header fields and introduce constraints. However, the execution results of netfilter do not affect the main TCP logic at all, and therefore we can safely ignore those constraints. Specifically, the netfilter ConnTrack module tracks the TCP connections passively and maintains connection states separately from the main TCP logic. Therefore, its execution is insignificant — even if we ignore its constraints and force a different execution path, it would have no consequence on the main TCP states we are interested in exploring.

3.5.2 Symbolizing the TCP Header and Options

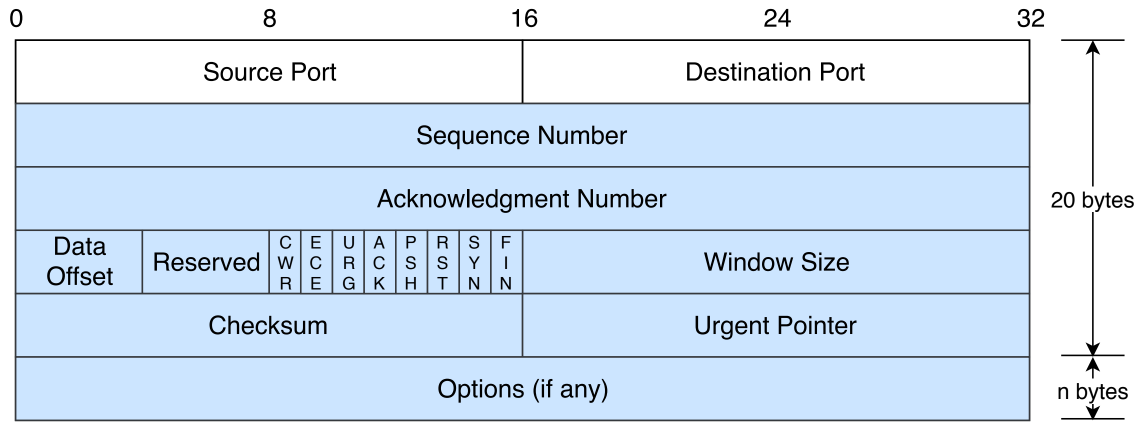


Figure 3.5: Symbolized TCP header and options.

Since we limit our scope to TCP-level insertion and evasion packets, we only symbolize the TCP header of a packet (not the IP header or the application payload) (see Figure 3.5). We symbolize all TCP header fields except the source and destination port numbers. The symbolized fields include the sequence number, acknowledgment number, data offset, flags, window size, checksum, and urgent pointer. In addition, we want to symbolize *TCP options*, which refers to the last part of the TCP header and has an associated variable length.

Symbolizing the TCP options field is intrinsically hard because it consists of a list of nested TLV (Type-Value-Length) structures. Currently there are 35 existing TCP option related numbers assigned by IANA [65], including those that are standard and others that are obsolete, and the number is growing. Some options have associated fixed lengths, and some are of variable length (e.g., SACK). Some options have associated subtypes (e.g., MPTCP). Although the maximum length of the TCP option field is 40 bytes, the number of combinations of all top-level option types is still huge. The problem worsens if we also include illegal cases (e.g., an option appears multiple times) or also want to consider the ordering of the options.

Linux only implements 10 TCP options using a parsing loop, which can still easily cause the path explosion problem. Theoretically there are at least $2^{10} = 1024$ execution paths even if we just execute the loop once. In practice, when it is compiled into a binary form, additional branches are introduced; hence, the number of possible paths is much larger. The problem is exacerbated exponentially given the already large number of paths that exist in the TCP logic. Because of these reasons, we need to bound the search space by limiting the number of possible combinations of TCP options.

While we attempted to bound the loop execution times and the number of occurrences of each TCP option, we found that the number of paths was still prohibitively large even if we executed the loop just once and allowed each option to occur at most once. Hence, as a practical means to mitigate this problem, in addition to bounding the execution times, we also feed a specific combination of TCP options as a seed (from traffic observed on the Internet) to our concolic execution engine; the execution explores our seed value first and then other values.

3.5.3 Abstracting the Checksum Function

The TCP checksum is calculated based on a pseudo-header that includes the IP addresses, the entire TCP header and the payload. As mentioned earlier, we do not want to symbolize the IP header or the payload since this is likely to harm the symbolic execution performance. Thus, instead, we *abstract* the checksum validation function as follows:

$$f(pkt) = \begin{cases} true & \text{if } header.checksum == 1 \\ false & \text{if } header.checksum == 0 \end{cases}$$

where f denotes the checksum validation function and pkt is the network packet under consideration. If the checksum field in the TCP header is equal to 1, then it is considered to be a valid checksum; if it is equal to 0, then it is an invalid checksum. The constraint solver thus generates a checksum of 1 for a valid checksum case, and 0 for an invalid checksum case. When we probe the DPI (discussed later), we fill the checksum field with either the proper valid or an invalid checksum, correspondingly. By abstracting the checksum function, we avoid solving complex constraints on the TCP header fields and thus improve performance.

3.5.4 Symbolizing the Server's Initial Sequence Number

During TCP's 3-way handshake, the server's initial sequence number (ISN) is a random number generated and sent in the SYN/ACK packet to the client. When the client receives the SYN/ACK packet, it needs to echo the server's ISN by sending an ACK packet with an acknowledgment number that is equal to the server's ISN plus 1. Because the server's ISN is randomly generated for each TCP connection, we need to symbolize the server's ISN in the offline symbolic execution phase and collect the path constraint that expresses the relationship between the server's ISN and the client's acknowledgment number. Then in the online probing phase (section 3.6), we constrain the server's ISN using the concrete value obtained from the SYN/ACK packet, and generate concrete values for the client's packets on the fly.

3.5.5 Multi-round Symbolic Execution

As mentioned earlier in section 3.4, we start our symbolic execution from the `LISTEN` state. We symbolize multiple packets in order to explore the state machine in more depth (up to the `ESTABLISHED` state). Specifically, we choose to symbolize 3 packets in total for several reasons. First, 3 packets should offer a reasonable coverage of the TCP state machine because only 2 packets are needed to advance the TCP state from `LISTEN` to `ESTABLISHED` (the SYN and ACK in a three-way handshake); the third packet can further explore other minor states in `ESTABLISHED`. Second, we prefer shorter sequences of insertion and evasion packets as longer sequences can be unreliable in practice (e.g., due to packet losses).

To explore different sequences of packets, we develop a custom path searcher/scheduler to guide S2E to explore packet sequences of 1 and 2 first (up to certain threshold), and then allow the third packet to arrive.

As discussed later in section 3.8, even though there are not many accept and drop points in TCP, the number of possible accept and drop paths is exponential and impossible to exhaust in our experiments, which motivated our search strategy to balance the exploration of sequences of different lengths.

3.6 Generating Online Evasion Attacks

By means of the offline concolic execution phase described in section 3.5, SymTCP obtains path constraints that can be used to generate insertion/evasion packet candidates. In this section, we describe SymTCP's differential testing phase to probe the DPI to identify behavioral discrepancies between the DPI's TCP implementation and that of the server.

3.6.1 Constructing insertion/evasion packet candidates

Armed with the constraints relating to each execution path collected during the symbolic execution, as described in section 3.5, together with some additional constraints, we can then feed these to a constraint solver to generate concrete values of TCP header fields. Using those values, SymTCP constructs a sequence of packets, $P_{1 \dots n} (n \leq 3)$, to probe the DPI.

There are two additional constraints. The first is the server's initial sequence number (ISN) as mentioned in subsection 3.5.4. The second includes additional constraints on TCP flags, SEQ and ACK numbers. These are especially necessary for candidate insertion packets when a packet hits a drop point early (and practically most fields are unconstrained). For example, if a packet is dropped because of an unsolicited MD5 TCP option, then it has no constraint on TCP flags, SEQ or ACK number. Since the hope is that the error is ignored by the DPI (not checking the MD5 TCP option), these other fields will have a direct effect on how the DPI processes the packet. Our solution in such cases is to generate these constraints to make the packet as legitimate as possible (i.e., with the correct SEQ and ACK number). For TCP flags, we just enumerate the most common ones, which are more likely to be accepted by the DPI (SYN, SYN/ACK, ACK, RST, RST/ACK, FIN, FIN/ACK). For example, we may generate a RST packet with an unsolicited MD5 option (with the additional constraint of the SEQ number to match the next expected one). The server of course will reject the packet but the DPI will accept it and terminate the connection incorrectly, allowing subsequent data to pass through unchecked. For candidate evasion packets, we do the opposite by generating random values of various fields and hope that it will be ignored by the DPI. Note that since an evasion packet is to be accepted by the server, most of the fields are already constrained and so we do not have much room to select the values of different fields.

3.6.2 Constructing follow-up probe packets

As mentioned in subsection 3.3.2, after sending a candidate evasion or insertion packet, we may still need to craft additional follow-up packets that contain bad keywords targeted by the DPI, in order to infer if there is any state discrepancy between the DPI and server (otherwise there is no observable feedback).

To construct follow-up packets, we need to know the current state of the TCP connection. If the current TCP state is not in the `ESTABLISHED` state, we need to send packets that cause it to transition into it. If the current TCP state is already the `ESTABLISHED` state, then we can directly send the data packet with the correct sequence and acknowledgment number. Due to this reason, we log the current TCP state after processing each packet during symbolic execution. Based on this, we use a simplified version of the TCP state machine to generate the follow-up packets for transitioning the connection from the specific TCP state to the `ESTABLISHED` state if need be. Subsequently, we send a data packet with the sensitive payload, and observe if it triggers any alarm on the DPI.

3.7 Implementation

Our system is built upon S2E 2.0 [37], which uses KLEE as its symbolic execution engine. We implement SymTCP as a set of S2E plugins written with around 2.5K lines of C++, and the probing and peripheral scripts were written with around 6.5K lines of Python.

3.7.1 Selective Concolic Execution

We start the selective concolic execution whenever `tcp_v4_rcv()` is entered, where the TCP header fields are symbolized. When the current program counter is outside the TCP scope, i.e., it leaves the `tcp_v4_rcv` function or it wades into some other territory (e.g., netfilter), we disable forking to let S2E run in a way similar to concrete execution, except that it still maintains and propagates symbolic variables. In this way, we can switch from symbolic execution to concrete execution and later switch back to symbolic execution again. In addition, we modify KLEE to prevent it from adding branch conditions to the path constraints when forking is disabled; thus, it does not over-constrain the symbolic variables.

S2E only instruments basic blocks and instructions but not the edges connecting basic blocks. However, in Linux TCP implementations, often it is the edge that determines the reason for acceptance or rejection; for example, an `if` and `goto` statement can enter the same exact basic block, but representing different reasons (acceptance or rejection). Thus, we also instrument the edges and implement an event. Finally, we bound the number of loops that can be traversed and the number of occurrences of TCP options, by limiting the number of executions of related edges of interest — we allow at most 5 TCP options in a packet, and each TCP option only occurs once, except the NOP option. We do not encounter any other loops where the number of iterations is symbolic.

3.7.2 Online Constraint Solving

We use the state-of-the-art Z3 [136] theorem prover as our online constraint solver to generate concrete values of TCP header fields. As mentioned previously, if we receive a SYN/ACK packet from the server, we then add its initial sequence number to the constraint and consult Z3 again to generate new concrete values for the following probing packets. This is because the following packets will need to acknowledge that number. Note that when we consult the constraint solver multiple times (to generate subsequent packets), we need to carry over the concrete values generated for the previous packets in order to maintain consistency. For example, the first packet has a payload of 4 bytes, and the second packet's sequence number needs to advance by 4.

3.8 Evaluation

Our evaluations of SymTCP are run on an server with 72 cores Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz, and 256GB memory. The host OS is Ubuntu 16.04 64-bit, and the guest OS is Debian 9.2.1 64-bit. We evaluate our system with Linux kernel version 4.9.3. We run S2E in parallel mode with 48 cores, which is the maximum number of processes S2E currently supports.

3.8.1 Experiment Setup

Table 3.1: A summary of labeled drop points

Reason	Count
TCP checksum error	5
TCP header length too small	1
TCP header length too large	4
MD5 option error	2
TCP flags invalid	7
SEQ number invalid	10
ACK number invalid	3
Challenge ACK	6
Receive window closed	2
Empty data packet	1
Data overlap in OFO queue	1
PAWS check failed	2
Embryonic reset	1
TCP_DEFER_ACCEPT drop bare ACK	1
TCP Fastopen check request failed	1
Total number	47

Before evaluating the system, we first manually label all the drop points reachable from *tcp_v4_rcv()* which is the TCP-level entry function for processing incoming packets. Specifically, in the Linux kernel, since an incoming packet will eventually be freed after being processed via *__kfree_skb()*, we inspect all invocations of it in the TCP implementation (both direct and indirect through wrapper functions *kfree_skb* and *tcp_drop*), and identify the program points or the branch statements (transitions between basic blocks such as `if`) that satisfy the definition of a *drop point* (see subsection 3.3.2).

We only consider drop points in the TCP `LISTEN`, `SYN_RECV`, and `ESTABLISHED` states. Because we assume the server doesn't initiate a connection, we know that it will not go into the `SYN_SENT` state. In other TCP states such as `TCP_CLOSE`, the server will not accept any further data packets. We also excluded some cases that are not practical for insertion packets: 1) a packet dropped due to memory allocation failures because it is rare to encounter memory pressure on the server; 2) a packet dropped due to listen queue overflow, which is not a common case; 3) a packet dropped due to SELinux check failed; 4) a packet dropped due to Xfrm check failed; 5) a packet dropped due to socket filter; 6) a packet dropped due to route error or no route; 7) a few other minor cases, e.g., unusual server configurations.

As a result, we eventually labeled 38 places in the source code where a packet gets dropped without changing states. Because S2E works on the binary level, we map the source code lines to binary addresses, and they are mapped to 47 binary level drop points (as one source-level conditional statement can be translated into multiple basic blocks in binary) as summarized in Table 3.1. A detailed list of all drop points can be found in Table 3.2.

Currently, we use two seed packets as inputs to the symbolic execution: 1. a SYN packet with all 0s in its TCP option fields; 2. a SYN packet with a TCP Timestamp option turned on. In practice, with 1, we can cover most drop points and accept points but can rarely cover the 2 drop points related to the TCP Timestamp option. With 2 as another seed packet, we are able to cover all drop points and accept points easily. We believe that the complete coverage of all accept and drop points is a good indication of our results.

We employ an HTTP request with bad keyword "ultrasurf" in our experiment:

Table 3.2: All drop points labeled in Linux kernel v4.9.3

Source file	Line number	TCP State	Major Reason	Total	Covered
tcp_ipv4.c	1404	Non-ESTABLISHED	TCP checksum error	1	1
	1607	Any	TCP header length <20	1	1
	1609	Any	TCP header length >TCP packet size	2	1
	1617	Any	TCP checksum error	2	1
	1655	SYN_RECV	TCP MD5 option check failed	1	1
	1672	SYN_RECV	ACK number != server ISN + 1	1	1
	1690	Non-SYN_RECV	TCP MD5 option check failed	1	1
tcp_input.c	3616	Non-LISTEN	Challenge ACK (the ACK case)	1	1
	3736	Non-LISTEN	ACK number >server send next	1	1
	3750	Non-LISTEN	ACK number older than previous acks but still in window	1	1
	4503	ESTABLISHED	OFO packet overlap	1	1
	4641	ESTABLISHED	Empty data packet	1	1
	4657	ESTABLISHED	Receive window is zero	1	0
	4716	ESTABLISHED	End SEQ number <= rcv_nxt (Retrans)	1	1
	4729	ESTABLISHED	SEQ >= rcv_nxt + window (out of window)	1	1
	4745	ESTABLISHED	Receive window is zero	1	0
	5195	ESTABLISHED	SEQ number <copied_seq (SEQ num too old)	1	1
	5270	Non-LISTEN	PAWS check failed (Timestamp)	1	1
	5284	Non-LISTEN	Challenge ACK (SYN) (out-of-window)	1	1
	5291	Non-LISTEN	SEQ out of window	4	3
	5325	Non-LISTEN	Challenge ACK (RST)	3	3
	5333	Non-LISTEN	Challenge ACK (SYN)	1	1
	5453	ESTABLISHED	Packet length <TCP header length	1	0
	5487	ESTABLISHED	TCP checksum error	1	1
	5531	ESTABLISHED	Packet size <TCP header length —— TCP checksum error	2	1
	5534	ESTABLISHED	No RST and no SYN and no ACK flag	1	1
	5911	LISTEN	ACK flag set	1	1
	5914	LISTEN	RST flag set	1	1
	5918	LISTEN	SYN and FIN flags set	1	1
	5925	LISTEN	No RST and no SYN and no ACK flag	1	1
	5947	Non-LISTEN	Fastopen tcp_check_req failed	1	0
	5951	Non-LISTEN	No RST and no SYN and no ACK flag	1	1
	6141	Non-LISTEN	SEQ $\geq$ rcv_nxt	1	1
tcp_minisocks.c	634	SYN_RECV	Retransmitted SYN	1	1
	716	SYN_RECV	PAWS check failed —— SEQ out of window	2	2
	735	SYN_RECV	SYN or RST flag set	1	1
	745	SYN_RECV	No ACK flag	1	1
	758	SYN_RECV	TCP_DEFER_ACCEPT drop bare ACK	1	1
Overall				47	39

`GET /AA...A#ultrasurf#<test_case_id># HTTP/1.1\r\nHost: local_test_host\r\n\r\n`

"A" is used to pad the HTTP request so that the first n packets before the follow-up packet will not contain the bad keyword (by definitions in §3.3.2 the first n packets may be accepted by the DPI). It is the follow-up packet that will carry the bad keyword "ultrasurf" and the remaining part of the request.

Table 3.3: Performance of offline symbolic execution

# of pkts	20-byte TCP pkts		40-byte TCP pkts		60-byte TCP pkts	
	Time to cover	Covered drop points	Time to cover	Covered drop points	Time to cover	Covered drop points
1	5s	8	5s	9	10s	8
2	20s	16	20m	19	18m	18
3	50s	31	1h2m	39	40m	38

Time cost could vary due to randomness in path selection of symbolic execution.

3.8.2 Symbolic Execution Results

In our experiments, we send symbolic packets with 20, 40, and 60 bytes in total, including the TCP header and the payload. As discussed in subsection 3.5.2, since we symbolize the TCP data offset header field, the length of the TCP header is variable. For example, if we send a TCP packet of 60 bytes, it always has a 20-byte TCP basic header, and the length of the TCP option can vary between 0 and 40-byte. As a result, the rest will be the TCP payload (from 0 to 40 bytes as well). We choose not to symbolize the length of the entire packet or the payload because more or fewer bytes in the payload does not really affect how TCP accepts or drops a packet.

As shown in Table 3.3, when we send 1 symbolic packet, we can cover only 8/9/8 drop points with a TCP packet of 20/40/60 bytes. By comparing the drop points covered, we found that the 40-byte case can cover one more drop point than the 20-byte case, which checks the TCP MD5 option. The 60-byte case covers one less drop point than the 40-byte case because it misses a drop point when the TCP data offset is larger than the actual TCP packet size. Because the TCP data offset is by design no more than 60, if we pick the actual size of a TCP packet to be 60, the condition can never be satisfied. Finally, by sending 1 symbolic packet, we can only cover drop points in TCP `LISTEN` state.

When we send 2 symbolic packets, we can cover 16/19/18 drop points with 20/40/60 bytes of TCP options and payload. The increased coverage of drop points is because we can now cover drop points in TCP `SYN_RECV` and part of them in `ESTABLISHED`. In addition, the 40-byte case covers 3 more drop points related to TCP options, i.e., MD5 and Timestamp. The 60-byte case still covers one less drop point related to TCP data offset.

When we send 3 symbolic packets, we can cover 31/39/38 drop points with 20/40/60 bytes of TCP options and payload. The increased coverage of drop points are because of more drop points in `ESTABLISHED` state covered, and also cases like data overlapping. The 20-byte case covers much less since it doesn't send packets with a payload.

We take a further look at the 8 drop points not covered by any of our experiments. 2 of them requires the TCP receive window size becomes 0. That means the server's receive buffer has to be full. This is very hard to achieve in reality and we don't want to flood the server. 1 drop point requires TCP Fast Open to be enabled on the server. The other 5 drop points are also infeasible due to various reasons. Overall, all 8 uncovered drop points are either not of interest or cannot be reached in reality. Furthermore, we found that 2 of the covered drop points are reached when the TCP state is in `CLOSE_WAIT`, which we ignore.

Because the 40-byte experiment can already cover all of the drop points covered by the 20-byte and 60-byte experiments, we use the dataset generated from the 40-byte experiment to probe the DPI. This dataset includes 56,787 test cases generated in around one hour which covers 37 drop points in binary (after filtering infeasible drop points).

Since the original dataset is too large, we cull out 10,000 test cases by sampling the dataset, and then use the sampled dataset to probe the DPI. The original dataset is highly

Table 3.4: Important accept points in Linux kernel v4.9.3

Source file	Line #	TCP State	Major Reason
tcp_input.c	4461	Non-LISTEN	OFO: Initial out of order segment
	4477	Non-LISTEN	OFO: Coalesce
	4533	Non-LISTEN	OFO: Insert segment into RB tree
	4684	Non-LISTEN	In sequence. In window.
	6408	LISTEN	Enter SYN_RECV
tcp_minisocks.c	773	SYN_RECV	Enter ESTABLISHED

imbalanced, ranging from 2 to 9,790 test cases for different drop points. To make it more balanced, we undersample the majorities while keeping the minorities intact. We order the drop points by the number of their corresponding test cases, and use the 50th percentile as a threshold. For the drop points whose corresponding numbers of test cases are below the threshold, we keep them intact; for the ones above the threshold, we proportionally sample the test cases corresponding to the overly represented drop points.

Finally, since we consider every path not reaching a drop point as an accept path, the accept paths can be diverse and overwhelming in number. To sample them, we explicitly label some important accept points, as listed in Table 3.4, which indicates TCP state changes and data entering receive buffer. During sampling, we group the test cases by the sets of labeled accept/drop points they reached.

3.8.3 Evaluation against DPI

We evaluated our sampled test set of 10,000 candidate insertion/evasion packets against 3 DPI systems, 2 open-source NIDSes, Zeek (formerly known as Bro), Snort, and a nation-wide censorship system, the Great Firewall of China (GFW).

We downloaded the latest version of Zeek (2.6) and Snort (2.9.13) at the time of writing, and conducted the experiment against the GFW on August 18, 2019.

Out of 10,000 test cases, we found 6,082 test cases can evade Zeek, including 5,771 cases caused by insertion packets and 311 cases caused by evasion packets; 652 test cases can evade Snort, including 432 cases caused by insertion packets, and 220 cases caused by evasion packets; 4,587 test cases can evade the GFW, including 1,435 cases caused by insertion packets and 3,152 cases caused by evasion packets. For GFW, most of the successful test cases caused by evasion packets are due to the "$SEQ \leq ISN$" strategy listed in Table 3.7, as a common condition shared by many test cases. For Zeek, though it has a similar "$SEQ < ISN$" strategy, most of such test cases are successful for different reasons, i.e., due to some preceding packets turning into insertion packets (as Zeek has a very loose check on incoming packets). For example, the third packet has a SEQ number less than ISN, which is an evasion packet, but the second packet is an insertion packet so the test case works because of the insertion packet.

To reason about the successful test cases and abstract them into high-level evasion strategies, we conducted postmortem analysis and evasion strategies summarization. For Zeek and Snort, even though we treat them as blackboxes when generating candidate insertion/evasion packets, they are actually both open-sourced, which allows us to pinpoint the underlying cause of evasion. In order to expedite this process, we replay the successful cases and record the binary execution trace of the DPI for each case. Then, we group the cases by the execution trace of the data packet containing the sensitive keyword which evaded the detection of the DPI (the trace therefore explains why this occurred), For Snort, we additionally record the trace caused by processing the server's ACK packet as some checks performed on Snort are delayed until the ACK packet is seen. In the end, we still manually verify the cases within the same group in case they actually belong to different reasons for evasion.

For GFW, since it is really a blackbox, we have to make hypotheses about the success reasons from prior knowledge [125] and then validate them. Specifically, we first replay the captured packet traces and verify if the result is stable; this eliminates the noisy results caused by random events such as packet loss or GFW overload. Then we slightly tweak the TCP header fields of the insertion/evasion packet and then replay the modified packet trace. If it cannot work, then it's likely the discrepancy is caused by that field.

We summarize a few featured evasion strategy (not a complete list) for each DPI in the next few sections. Overall, we not only rediscovered already known strategies but also found 14 novel strategies comparing with previous works using manually crafted insertion/evasion packets.

3.8.4 Zeek

Table 3.5: Successful strategies on Zeek v2.6

Strategy	SYN with data
TCP state	LISTEN/SYN_RECV/ESTABLISHED
Description	(Insertion) SYN packet with data
Linux	Ignore data
Zeek	Accept data

Strategy	Multiple SYN
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) SYN packet with out-of-window SEQ num
Linux	Discard and send ACK
Zeek	Reset TCB

Strategy	Pure FIN
TCP state	ESTABLISHED
Description	(Insertion) Pure FIN packet without ACK flag
Linux	Discard (may send ACK)
Zeek	Flush and reset receive buffer

Strategy	Bad RST/FIN
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) RST or FIN packet with out-of-window SEQ num
Linux	Discard (may send ACK)
Zeek	Flush and reset receive buffer

Strategy	Data overlapping
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) Out-of-order data packet, then overlapping in-order data packet
Linux	Accept in-order data
Zeek	Accept first data

Strategy	Data without ACK
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) Data packet without ACK flag
Linux	Discard
Zeek	Accept

Strategy	Data bad ACK
TCP state	ESTABLISHED
Description	(Insertion) Data packet with ACK > snd_nxt or < snd_una - window_size
Linux	Discard
Zeek	Accept

Strategy	Big gap (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) Data packet with SEQ > rcv_nxt + max_gap_size (16384)
Linux	Accept
Zeek	Ignore later data

Strategy	SEQ < ISN (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Evasion) Data packet with SEQ num < client ISN and in-window data
Linux	Accept in-window data
Zeek	Ignore

Zeek (formerly known as Bro) [87] is very liberal in accepting incoming packets.[2] It is therefore relatively easy to bypass using insertion packets. We list some strategies in Table 3.5. In most cases, it only looks at the TCP flags of a packet but does not check SEQ or ACK number for TCP control packets, e.g., SYN, RST, FIN. This makes many strategies that were previously reported feasible [92, 68, 125]. For example, whenever Zeek receives a SYN packet in an existing connection, it simply tears down the TCB and creates a new one. But Linux doesn't accept out-of-window SYN packets in `SYN_RECV` state or any SYN packets in `ESTABLISHED` state. As a result, an attacker can easily inject a SYN packet (as insertion packet) to tear down the TCB and recreate a TCB with a different ISN that Zeek will keep track of, thus allowing later packets to evade detection.

Another interesting strategy which we have not seen applied (only hypothesized in [92]) in any prior work: TCP RFC 793 allows data in SYN packet to be buffered and delivered to the user only when the connection is fully established, but Linux doesn't buffer data in SYN packet unless in the TCP Fastopen cases. In this case, Zeek correctly implements the RFC and accepts data in SYN packets. However, this allows an attacker to attach junk payload in a SYN packet as "cover" for the actual data sent in later packets.

In addition, we also found a novel evasion strategy that was not mentioned in any prior work: if we send a data packet with SEQ number less than the client ISN but has partial data in server's receive window, the data will be ignored by Zeek, but Linux will accept the data in window (an evasion packet).

[2]Zeek does log weird packets to a weird.log for offline analysis.

3.8.5 Snort

Table 3.6: Successful strategies on Snort v2.9.13

Strategy	Multiple SYN
TCP state	ESTABLISHED
Description	(Insertion) SYN packet with in-window SEQ num
Linux	Discard and send ACK
Snort	Teardown TCB

Strategy	In-window FIN
TCP state	ESTABLISHED
Description	(Insertion) FIN packet with SEQ num in window but $\neq$ rcv_nxt
Linux	Ignore FIN (may accept data)
Snort	Cut off later data

Strategy	FIN/ACK bad ACK
TCP state	ESTABLISHED
Description	(Insertion) FIN/ACK packet with ACK num $>$ snd_nxt or $<$ snd_una - window_size
Linux	Discard (may send ACK)
Snort	Cut off later data

Strategy	FIN/ACK MD5
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) FIN/ACK packet with TCP MD5 option
Linux	Discard
Snort	Cut off later data

Strategy	In-window RST
TCP state	ESTABLISHED
Description	(Insertion) RST packet with SEQ num $\neq$ rcv_nxt but still in window
Linux	Discard and send ACK
Snort	Teardown TCB

Strategy	RST bad timestamp
TCP state	SYN_RECV
Description	(Insertion) RST packet with bad timestamp
Linux	Discard
Snort	Teardown TCB

Strategy	RST MD5
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) RST packet with TCP MD5 option
Linux	Discard
Snort	Teardown TCB

Strategy	RST/ACK bad ACK num
TCP state	SYN_RECV
Description	(Insertion) RST/ACK packet with ACK num $\neq$ server ISN + 1
Linux	Discard
Snort	Teardown TCB

Strategy	Partial in-window RST (New)
TCP state	ESTABLISHED
Description	(Insertion) RST packet with SEQ num < rcv_nxt but partial data in window
Linux	Discard
Snort	Teardown TCB

Strategy	Urgent data (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Evasion) Data packet with URG flag and urgent pointer set
Linux	Consume 1 byte urgent data
Snort	Ignore all data

Strategy	Time gap (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Evasion) Data packet timestamp = last timestamp + 0x7fffffff/0x80000000
Linux	Accept
Snort	Ignore

Snort implements OS-specific TCP state machines, including Windows, Linux, and Mac OS; its TCP implementation is the most rigorous among the three DPIs. However, from our results, even its Linux version still has discrepancies from the Linux kernel we analyzed. The strategies we found are listed in Table 3.6. In general, Snort checks the SEQ number for control packets but doesn't check ACK number. Also, it doesn't check TCP

MD5 option and accepts in-window SYN, FIN, and RST packets too liberally. Whenever it receives an in-window SYN or RST packet, it will tear down the TCB (matching the behavior of older versions of Linux); and whenever it receives an in-window FIN packet, it will mark the connection as CLOSED and discard data which SEQ number larger than the end SEQ number of the FIN packet. On the contrary, the latest Linux doesn't accept any SYN packet in ESTABLISHED state, and requires SEQ number of FIN or RST packet to be equal to `rcv_nxt`. In addition, Snort also accepts FIN or RST packet with out-of-window ACK number or TCP MD5 option, which will be discarded by Linux. Most of these strategies have also been mentioned in [92] (though not all of them are tested in practice), and the usage of TCP MD5 option was done in [125].

Now we discover two novel strategies unique to the Snort implementation. The first strategy is related to how Snort implements TCP Timestamp option validation (it is the only DPI we are aware of that attempts to perform timestamp checks). Interestingly, we found its implementation to be slightly different from Linux in 2 ways: 1) Snort doesn't check timestamp for RST packets in `SYN_RECV` state (as mandated by RFC 7323) while Linux does. 2) In PAWS checking, if the TSval in the current packet is older than that in the last packet, it will reject the current packet. However, due to slightly different implementations of the check of Snort and Linux, the acceptable TSval ranges are "off by two". As a result, say if the first packet has a TSval of `0x80000000` and the second packet has a TSval of `0` or `0xffffffff`, then Linux will accept the second packet, but Snort will reject it. The pseudo-code of their implementations can be found in Listing 3.1 and Listing 3.2.

```
if ((signed int)(last_packet->tsval - current_packet->tsval) <= 1) {
    // PAWS check succeeded
}
```

Listing 3.1: Pseudo-code of Linux PAWS (timestamp) check

```
if ((signed int)((current_packet->tsval - last_packet->ts_val) + 1) < 0) {
    // PAWS check failed
}
```

Listing 3.2: Pseudo-code of Snort PAWS (timestamp) check

The second novel strategy is related to the urgent pointer processing logic, which is notoriously ambiguous [90] and often implemented incorrectly, even in major OSes such as Linux [60]. Simply put, an urgent pointer is supposed to allow TCP to specify some range of data in the payload to be marked as urgent, which will be treated differently when a receiver sees it (e.g., immediately pushed to the application layer using a separate interface [60]). In Snort, it interprets the urgent pointer as the offset to the last byte of the urgent data and simply discards all of the bytes before this offset. In Linux though, it consumes 1 byte of urgent data (right before the urgent pointer offset) which is stored in a separate place, and leaves the remaining payload intact. Our system initially discovered an evasion packet with urgent flag and urgent pointer set to a random location in a packet (which happens to point to an insignificant padding byte), and therefore preserving the semantic and the keyword in the HTTP request. However, Snort discards all the data before the urgent pointer offset and fails to reconstruct the HTTP request.

3.8.6 Great Firewall of China

Table 3.7: Successful strategies on the GFW

Strategy	Bad RST
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) RST packet with bad checksum or TCP MD5 option
Linux	Discard
GFW	Teardown TCB
Strategy	Bad data
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) Data packet with bad checksum or TCP MD5 option or bad timestamp
Linux	Discard
GFW	Accept
Strategy	Data without ACK
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) Data packet without ACK flag
Linux	Discard
GFW	Accept
Strategy	SEQ $\leq$ ISN (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Evasion) Data packet with SEQ num $\leq$ client ISN and in-window data
Linux	Accept in-window data
GFW	Ignore
Strategy	Small segments (New)
TCP state	SYN_RECV
Description	(Evasion) Data packet with payload size $\leq$ 8 bytes
Linux	Accept
GFW	Ignore
Strategy	FIN with data (New)
TCP state	SYN_RECV/ESTABLISHED
Description	(Insertion) FIN packet with data and without ACK flag
Linux	Discard
GFW	Teardown TCB

Strategy	Bad FIN/ACK data (New)
TCP state	ESTABLISHED
Description	(Insertion) FIN/ACK packet with data and bad checksum or TCP MD5 option or bad timestamp
Linux	Discard
GFW	Teardown TCB

Strategy	FIN/ACK data bad ACK (New)
TCP state	ESTABLISHED
Description	(Insertion) FIN/ACK packet with data and ACK num $>$ snd_nxt or ACK num $<$ snd_una - window_size
Linux	Discard
GFW	Teardown TCB

Strategy	Out-of-window SYN data (New)
TCP state	SYN_RECV
Description	(Insertion) SYN packet with SEQ num out of window and data
Linux	Discard and send ACK
GFW	Desynchronized

Strategy	Retransmitted SYN data (New)
TCP state	SYN_RECV
Description	(Insertion) SYN packet with SEQ num = client ISN and data
Linux	Discard
GFW	Desynchronized

Strategy	RST bad timestamp (New)
TCP state	SYN_RECV
Description	(Insertion) RST packet with bad timestamp
Linux	Discard
GFW	Teardown TCB

Strategy	RST/ACK bad ACK num (New)
TCP state	SYN_RECV
Description	(Insertion) RST/ACK packet with SEQ num $\neq$ server ISN + 1
Linux	Discard
GFW	Teardown TCB

The GFW conducts a wide range of censorship on different network protocols, such as HTTP/HTTPS, DNS, Tor, etc. Although it has a relatively lenient checking on individual packets, it's known to have some sophisticated and robust mechanism to thwart desynchronization attacks according to recent research [125]. In addition to the strategies that were previously known, we also identify several novel strategies which we will describe below.

Interestingly, we found that the GFW ignores data packet with a start SEQ number less than or equal to the initial sequence number (ISN) but has in-window data, whereas Linux accepts the in-window data. Therefore the strategy discovered by our system is to send such an evasion packet with a sensitive keyword as in-window data (and with padding automatically prepended to cover the bytes that are out-of-window).

Another interesting and surprising finding is that the GFW ignores data segments whose sizes are less than or equal to 8 bytes. This is discovered through a small first data packet (remember each of our packets has a maximum payload length of 20), which is simply ignored by the GFW. Missing the first data packet will cause the GFW to miss the fact that it is an HTTP request and subsequently ignore the sensitive keyword. However, we found this strategy works perfectly in `SYN_RECV` state only but not `ESTABLISHED`. To understand the reason, we conducted further investigation. It turns out that in `ESTABLISHED` state, this strategy can only evade one type of GFW devices that inject RST/ACK packets, but not the ones injecting RST packets [125]. The GFW devices injecting RST packets will establish a TCB and start monitoring payload (including packets of 8 bytes or fewer) only after the 3-way handshake. This explains why this strategy works perfectly in `SYN_RECV` state only.

The last set of novel strategies are related to tearing down the state on GFW.

First, we found FIN packets or malformed FIN/ACK packets with data can cause the GFW to tear down its TCB, but without data it does not work. More interestingly, if we first send some in-order or out-of-order data packets and then send the FIN or malformed FIN/ACK packet, the FIN or FIN/ACK packet does not have to have data. This seems to indicate that GFW will agree to accept FIN packets only after some data have been transmitted (otherwise the FIN is suspicious and will not be accepted). Similarly, we found an out-of-window SYN packet with data or a retransmitted SYN packet with data can also desynchronize the GFW (causing it to synchronize its expected sequence number to the one in the SYN packet) but they don't work without data. For RST packet with a bad timestamp, it only works in `SYN_RECV` state since Linux only validates timestamp on RST packet in `SYN_RECV` state but not in `ESTABLISHED` state. None of the strategies were reported in the latest study of GFW [125].

Comparing our strategies with previous works on GFW with manually crafted packets [125, 74], we have rediscovered all the TCP-layer strategies used in [74] (except the IP and HTTP layer strategies which are beyond our scope). In addition, we have rediscovered all the primitive strategies used in [125], except that they also discovered compound strategies that can evade multiple types of GFW devices, based on their manually inferred GFW model. Specifically, Bad RST, Bad Data, and Data without ACK are old strategies, and the other strategies are all new. Strategies SEQ $\leq$ ISN and Small segments are completely new, while the other new strategies are subtle variations of known strategies that no longer work. This demonstrates the power of an automated tool that is capable of discovering such subtle variations.

3.9 Discussion and limitations

Path Explosion. In our evaluation, we show that processing only three symbolic packets can already lead to path explosion — tens of thousands of paths (the result of handling three packets) generated in an hour. This is because there can be multiple different paths reaching the same drop/accept point. Each of these paths corresponds to a unique sequence of packets (determined by the path constraints), which may potentially lead to various evasion and insertion strategies.

In order to tackle with path explosion, besides restricting symbolic execution within the scope of TCP code, we have also made some pruning decisions based on our domain knowledge. We summarize them in one place as follows (details discussed in section 3.4 and section 3.5): 1) bound occurrences of TCP option fields by allowing each TCP option to occur only once, since redundant options are not useful in triggering any new code; also we only allow at most 5 TCP options in a packet, since most of the options are independent of each other thus complex combinations of options are unlikely useful; 2) terminate an execution path once reaching a drop point, because packets reaching drop points don't cause any state changes; 3) terminate an execution path once the connection is in a state that cannot further deliver data, e.g., `CLOSE_WAIT`; 4) carefully label accept and drop points, we are aiming at covering all accept and drop points but not all execution paths, therefore reduce the search space.

At the moment, we randomly sample from these paths with equal probability and do not differentiate or prioritize them. However, a better solution is to understand the relationships among these paths and avoid visiting paths that are unlikely to lead to any fruitful results. One example is that for different paths reaching the same accept point, we know that they correspond to packets accepted by the server, but we hope that they are ignored by the DPI. In such cases we should theoretically prefer longer paths, because they go through more corner cases (e.g., more checks or different conditions of acceptance) and the DPI is less likely to handle them perfectly. Another example is that, in our evaluation, we find that there are many packet sequences sharing the same prefix of two accept packets, and the second packet happens to be a valid evasion packet; this means that regardless of what the third packet is in a sequence, it will always succeed in eluding the DPI for the same reason (Figure 3.4). Unfortunately, during the offline path exploration phase, we are unable to tell if the second packet will be a successful evasion packet and terminate any further exploration. We plan to use the result we obtain from online testing to prune the offline analysis in the future.

Handling Overlapping Data as Evasion Strategies. Our model currently does not handle overlapping data well and cannot generate all data overlapping strategies as done manually in prior work. This is because it is necessary to model how the TCP implementation evicts data in the buffer. For example, in certain operating systems, if data overlapping is detected, they prefer to discard the old copy and accept the new one. More generally, we need to model how a packet may retroactively change the effect of a previous packet, and at the moment our model assumes the effect of each packet is independently exerted and cannot be revoked. We plan to handle this corner case by extending our model as future work.

Extending SymTCP to Other TCP Implementations / DPIs / Network Proto-

cols / Server-side. Although we pick a specific version of Linux kernel to evaluate our system, our system is not restricted to any specific version and can be easily applied to other versions as well. The minimal requirement is to label all drop points, and optionally, some critical accept points (to group accept paths), as shown in section 3.8. Since the TCP implementations doesn't change much across kernel version, it should take less efforts for someone with experience to label another version. It took us less than an hour to do the labeling on the most up-to-date Linux kernel version (v5.4.5). In order to apply our method to another OS or TCP implementation much different from the current one, we may need to do more path pruning depending on the coverage, i.e., if symbolic execution cannot cover all desired accept and drop points, manual analysis is required to improve coverage. Extending SymTCP to other DPIs is easy. With results from symbolic execution, we can immediately probe the new DPI with the generated candidate packets; however, needed is the manual analysis of the results. Extending SymTCP to another protocol is in principle possible (we believe the insertion and evasion definitions are general). However, it can be tricky due to protocol-specific adaptations. For example, our pruning decisions and abstractions are specific to TCP. Furthermore, if the protocol uses crypto functions, they must be explicitly handled, since SMT solver is unable to solve complex constraints accumulated in crypto functions [14, 79]. Besides, we need to label drop and accept points. These aspects will require additional research.

In our demonstration, we use SymTCP to help the client side to elude DPI. Our approach can be applied to the server side as well. In that case, we will need to model the client-side TCP implementation, i.e., run symbolic execution on the client's TCP implementation. For example, if the client is using Linux, the process should be similar to what we do to model the server-side TCP implementation. Note that, since the client is the initiator of a TCP connection, we will need to consider TCP states corresponding to the initiator, e.g., exploring execution paths related to the `SYN_SENT` state.

Defenses: Traffic Normalization and Per-Host Packets Reassembly. To mitigate DPI elusion attacks, solutions have been proposed to normalize the traffic [61, 45, 124], where packets are actively manipulated and sometimes additional packets are injected to confirm the result of the previous packet. These normalization strategies are deemed to prevent many evasion strategies. However, they are based on a large number of hand-crafted rules (38 rules for TCP in [61] without formal guarantees. We believe our automated system can in fact be a great test against these defenses. Unfortunately we are not aware of any real-world implementations. Another strategy is proposed in [103], where the authors argue that the DPI's behaviors should be tailored to each host that it is responsible for protecting (e.g., those in intranet). In theory, this strategy is sound, but in practice it comes with high cost, as the behavior of the DPI needs to be customized for different operating systems (and even across many versions). Snort is the closest to this line of thinking; unfortunately its Linux version of TCP state machine is shown to be clearly vulnerable. Furthermore, in certain contexts, e.g., state-level censorship, it is simply infeasible to build per-host profiles of the majority of machines on the Internet.

3.10 Related Works

Evading Deep Packet Inspection. A major line of research on evading deep packet inspection is unilateral traffic manipulation, by injecting crafted network packets to desynchronize the DPI system from one endhost. This attack is practical since it needs to be deployed on only a local host, and doesn't require any cooperation from the remote host. The underlying idea dates back to 1998 in a report by Ptacek et al. [92]. They proposed the idea of insertion and evasion attacks on NIDS and enumerated a variety of implementation-level discrepancies in TCP and IP protocols. The discovered strategies are based on analyzing out-of-date DPIs and operating systems (FreeBSD 2.2), and many of the strategies no longer apply. Khattak et al. [68] and Wang et al. [125] followed the same principle to study evasions against the Great Firewall of China and demonstrated their effectiveness in practice. Li et al. [74] conduct a comprehensive measurement that leverages similar TCP and IP level discrepancies to evade a wide range of middleboxes such as traffic classification systems in multiple ISPs and the censorship systems in China and Iran. All of the above research rely on manual analysis of the TCP implementations in operating systems and reverse engineering of DPIs. In this work, we propose to make an important step towards automating the evasion tests of DPI systems. A concurrent work by Bock et al. [20] automates censorship evasion strategy discovery by mutating existing packet traces. In contrast, we propose a more principled approach to search for the evasion strategies, by targeting the corner cases in packet processing logic on Linux, which may be handled differently on DPIs.

Symbolic execution of network protocol implementations. In the past decade, symbolic execution has emerged as a powerful formal verification technique and been widely applied in the analysis and verification of network protocol and network function implementations. For example, in [32, 33], the authors employ symbolic execution to extract the accept and reject paths in essential components of the TLS protocol, i.e., X.509 certificate validation and PKCS#1 signature verification, to find semantic bugs by cross-validating different implementations. Kothari et al. [71] use symbolic execution to find protocol manipulation attacks where a malicious endhost can induce a remote peer to send more packets more aggressive than it should. Song et al. [111] explore the possibility of sending multiple packets in symbolic execution, and they aim at finding low-level and semantic bugs given rule-based specifications extracted from protocol specifications.

DPI model inference. Ideally, if we can infer the DPI model (i.e., state machine) automatically and completely, then it is much easier to identify the discrepancies with the endhost's state machine. Argyros et al. [9, 8] proposed the first algorithm that learns symbolic finite automata with enough queries and observations of a target system. The algorithm is applied to regular expression filters, TCP implementations and Web Application Firewalls (WAFs), to do fingerprinting and discover evasion attacks. Similarly, Moon et al. [80] synthesize high-fidelity symbolic models of stateful network functions (including TCP state machines of DPI middleboxes), by generating queries and probes offline (albeit it requires the availability of the network function's binary). Unfortunately, the completeness and accuracy of the inferred model is inherently dependent on the queries. Therefore, we choose to consider the DPI a complete blackbox and do not attempt to learn its state machine explicitly. To some extent, though, we indeed attempt to "learn its model" by generating proper queries to it (with the guidance of a Linux TCP state machine).

Grammar-based fuzzing and exhaustive testing. Generating meaningful inputs guided by a grammar that describes their formats can be beneficial to fuzzing [17, 58, 89]. However, fuzzing tends to generate overly many inputs and in our case will be inefficient in testing all the candidate packets. Furthermore, defining a grammar or model at the implementation-level requires a thorough analysis of all the subtleties of TCP. Therefore, models extracted from the specification are not sufficiently detailed to capture the intricacies of the protocol. In contrast, our work can be viewed as attempts to "extract" the implementation-level model.

3.11 Conclusion

In this chapter, we explore the use of symbolic execution to guide the generation of insertion and evasion packets at the TCP level for automated testing against DPI middleboxes. We developed a system from end to end following this idea and demonstrated its effectiveness with both known and novel strategies against three popular DPIs: Zeek (Bro), Snort, and GFW. The system can be easily extended to other DPIs. We believe our work is an important step towards automating the testing of DPI middleboxes in terms of their robustness against evasion.

Chapter 4

Themis: Ambiguity-Aware Network Intrusion Detection based on Symbolic Model Comparison

4.1 Introduction

Network Intrusion Detecion Systems (NIDS) are inherently vulnerable to evasion attacks that exploit implementation-level discrepancies stemming from ambiguities in network protocol specifications. NIDS will need to interpret network traffic in the same way as endhosts, to derive accurate information. However, different endhosts may run slightly different implementations of the same protocol, while traditional NIDS only incorporate one specific implementation. To be compatible with various endhost implementations, NIDS typically opt for a simplified implementation that over-approximates the behaviors on the endhosts. This in turn, is the underlying cause of the aforementioned discrepancies.

Numerous studies [92, 61, 68, 125, 74, 20, 126] have shown that an attacker can unilaterally manipulate his packets to trick the NIDS into losing track of its connections, and thereby successfully achieve evasion. In particular, when stateful protocols such as TCP are used, attackers can inject as few as a single packet to desynchronize the NIDS with respect to the current connection state permanently. Recently, crafting techniques for evasion have matured from manual to automatic strategy generation [20, 126, 19, 18, 96], which enables the easy generation of a large number of successful evasion strategies in a short time. Therefore, the threats faced by NIDSs are increasingly severe.

As defenders, we seek to regain the advantage and proactively prevent such potential attacks from happening, instead of reactively patching NIDSs. In this work, we propose a novel framework, Themis, to defend against evasion attacks. As running all possible network protocol implementations on a NIDS can be prohibitively expensive, prior work has proposed to choose a specific implementation for each endhost that it aims to protect [104]. However, due to the diversity of endhost implementations and the challenge in tracking the software versions on all the protected endhosts, such an approach has not been adopted in practice. In this work, we show that it is not necessary to choose between the various implementations. Rather, one can learn the discrepancies among different implementations ahead of time, and fork the connection states on the NIDS when ambiguous packets are received. The NIDS will then analyze the plurality of forked states in parallel, ensuring that one of the connection states will be synchronized with that of the endhost.

In this work, we focus on TCP, because it is the underlying protocol of most application-layer protocols, and widely targeted in evasion attacks due to its statefulness [20, 126, 19]. To learn the discrepancies between various implementations (especially for TCP with a long history), Themis leverages symbolic execution to automatically extract high-fidelity models from common endhost network protocol implementations. Compared to manually reconstructed models, our models are faithful representations of the actual software running on the endhosts, and are guaranteed to have exactly the same behaviors. Moreover, the extracted models are in the form of high-level SMT (satisfiability modulo theories) formulas; thus, it is easy to use SMT solvers to automatically compare two models to find discrepancies. Upon finding a comprehensive set of discrepancies, we can then build an ambiguity-aware NIDS based on nondeterministic finite automata (NFA) that can effectively and simultaneously support multiple different implementations. This approach has several benefits. First, since we go straight to the endhost implementations, we can abandon the existing over-simplified and over-approximated NIDS implementations that have potentially many more discrepancies. Second, with the distilled discrepancies, we no longer need to blindly run many different implementations (some of them may not exhibit any discrepancy) at the same time. In fact, our Themis-enabled NIDS forks its connection states only when ambiguities are encountered and thus, is cost-effective.

The biggest challenge in applying symbolic execution in practice is its scalability, especially when the goal is to achieve a complete analysis on complex modern software [27], as in our case with the TCP implementation. This is largely due to the nature of the heavy-weight analysis of symbolic execution and well-known problems such as path explosion. In our work, we employ several techniques to improve the performance of symbolic execution, without degrading the fidelity of the results. Specifically, we leverage state merging [73] to drastically reduce the cost of symbolic execution and constraint solving (from days to minutes) with domain knowledge of TCP.

In summary, our main contributions in designing and implementing Themis are the following:

- We use symbolic execution to extract high-fidelity models from TCP implementations. We solve the scalability challenge in symbolic execution leveraging state merging without degrading the fidelity of the results. To the best of our knowledge, we are the first to successfully conduct an exhaustive symbolic execution on full-fledged modern TCP implementations.
- We use constraint solving to automatically compare the symbolic models extracted from different versions of Linux kernels, and then summarize the discrepancies between them. We are able to reproduce all discrepancies that were previously reported between modern Linux kernel versions in the past decade, from 3.0 to 5.10. We even discover a few previously unknown subtle discrepancies.
- We design a novel NFA-based NIDS model that accounts for ambiguities and identify them during runtime. This model enables the NIDS to fork its connection states upon encountering a potential ambiguity associated with an incoming packet, to explore all possible ways that an endhost might handle the packet. We demonstrate that with Themis, a NIDS can successfully capture all existing evasion strategies and the new ones presented in this work, with negligible additional overhead.

4.2 Background

4.2.1 NIDS Evasion Based on Traffic Manipulation

NIDSs are known to be inherently vulnerable to evasion attacks [92], which typically exploit discrepancies between network protocol implementations of the NIDS (e.g., at the IP, TCP and HTTP levels) and those of the endhosts. Stateful protocols like TCP with complex implementations, are likely to manifest a larger set of discrepancies. An attacker can send a sequence of specially crafted network packets with a malicious payload, to make the NIDS and the remote host reassemble them into different data streams. The NIDS will see the reassembled data stream without the malicious payload while the remote host will see the reassembled data stream with the malicious payload, and thus, will be subject to attack. Such discrepancies arise largely due to ambiguities in network protocol specifications, as well as evolution of such specifications (e.g., new features added over the years). For example, in TCP implementations based on RFC 5961 [94], RST packets with sequence numbers in the receive window but not equal to the next expected sequence number are no longer accepted; however, older implementations still accept such RST packets. Different operating systems (OSes) and different versions thereof, all differ in their implementations. Even subtle discrepancies have been shown to lead to an evasion attack against the NIDS [126]. Importantly, our observation is that NIDSs usually use much simpler network protocol implementations as compared to endhosts to reduce their overhead, which widen the gap between their implementations and those of the endhosts.

Researchers have leveraged these discrepancies to design numerous evasion strategies that can bypass state-of-the-art NIDSs [125, 74, 126, 20]. For example, if a RST packet is accepted by the NIDS but not by the endhosts, the NIDS will consider the connection to be terminated and lose track of the connection. On the other hand, if a RST packet is accepted by the endhosts but not the NIDS, the NIDS will keep track of this terminated connection and if a later connection reuses the same 4-tuple, the NIDS will fail to track the new connection. Similar strategies have been crafted by manipulating control packets such as SYN or FIN packets, as well as data packets. On the defense side, mitigations such as traffic normalization [61] and Active Mapping [104] have been proposed. However, they either cannot eliminate all ambiguities or require additional information from endhosts and thus, still leave opportunities for attackers.

The root cause of the problem is that a traditional NIDS applies a specific network protocol implementation, but there are many different implementations running on the endhosts, all compliant with the same protocol specification. Thus, the NIDS cannot always recover the same information from the network traffic as that by an endhost. Blindly running all different implementations on the NIDS can be prohibitive in terms of overhead. In order to solve this problem, we propose an NFA-based NIDS that forks the connection state only when ambiguities are encountered. Our approach enables the NIDS to explore the appropriate possibilities, while introducing relatively low overhead. However, this requires prior knowledge of existing implementations running on the endhosts. To enable this, we employ symbolic execution to extract high-fidelity models from implementations, and empower the NIDS with these models.

4.2.2 Selective Symbolic Execution and State Merging

Symbolic execution is a formal program verfication technique to systematically find bugs or verify properties in software programs [69, 23]. With promising breakthroughs in automatic reasoning via SAT and SMT solvers, researchers are now widely adopting symbolic execution. Due to its heavyweight analysis, symbolic execution can still only be applied to a small scope of the program, and has to be carefully tuned to avoid uncontrollable path explosions. In addition, practical programs may contain external code not traceable by the symbolic executor, or complex constraints involving non-linear arithmetic or transcendental functions [13]. To make symbolic execution more practical, researchers have proposed "concolic" execution [77], a mixture of concrete execution and symbolic execution, which allows concrete execution to kick in when symbolic execution is incapable or inefficient in dealing with certain parts of the program.

Selective symbolic execution [37] is an innovative form of "concolic" execution that allows switching between symbolic execution and concrete execution at code boundaries. This will restrict symbolic execution only within the scope of interest, while running other parts of the code (e.g., libraries and system calls) with the much faster concrete execution. Defining the boundary between symbolic execution and concrete execution is usually tricky. Exhaustive symbolic execution is theoretically both sound and complete. Here soundness means all inputs derived are guaranteed to yield expected outcomes, i.e., no false positives; completeness means all inputs are covered, i.e., no false negatives. Selective symbolic execution may have impacts on both soundness and completeness while improving performance, because it doesn't completely model all possible outcomes of the code being executed concretely. Therefore, we need to carefully define the scope of symbolic execution to make sure no side effects that may impact the main logic will be introduced by the code out of scope. If any side effects were introduced, we may miss them and be subject to loss of soundness and completeness (by introducing false positives and false negatives). In the Linux kernel, we only run symbolic execution on the TCP core logic, while leaving other parts of the kernel as out of scope.

Numerous works [25, 32, 126] have used symbolic execution to verify properties or discover bugs in software programs. However, they randomly explore only parts of the program, and get partial coverage. These approaches aim at opportunistically finding bugs rather than achieving complete coverage. This causes loss of both completeness and soundness, and leads to false negatives and false positives. Differently, our goal is to extract a complete model of the target code we are interested in, so that we can retain completeness and soundness, which means no false negatives or false positives. As discussed, scalability is known to be the biggest challenge in symbolic execution. The problem worsens when running symbolic execution on binaries rather than on source codes, since more branches could be introduced into the low-level assembly code after compilation.

To achieve scalability, researchers have proposed state merging [62], which can reduce the number of execution paths in symbolic execution, but at the cost of introducing harder-to-solve constraints for the constraint solver. We use an example in Figure 4.1 to illustrate the rationale of state merging. A symbolic execution state is defined as a 3-tuple (ℓ, σ, π). ℓ denotes the current program location; σ denotes the symbolic store that stores all symbolic and concrete values associated with the current state; π denotes the path constraints associated with the current state. In Figure 4.2 and Figure 4.3, we demonstrate the process of symbolic execution without and with state merging respectively; here, each block represents a state. Without state merging, a state forks when a conditional branch is encountered and both branches are feasible. The number of states doubles each time and so, there will be 4 states after two branches. With state merging, two states at the same ℓ can be merged by: 1) combining their paths constraints π with a logical OR; 2) merging their symbolic stores σ with if-then-else (ITE) expressions. For example, when two states $(\ell : 8, \sigma : a = 5, \pi : x > 10)$ and $(\ell : 8, \sigma : a = -5, \pi : x \leq 10)$ meet at ℓ 8, their paths constraints are merged into $x > 10 \vee x \leq 10$, which can be simplified to $true$, and the value of variable a becomes $ite(x > 10, 5, -5)$.

```
1   int foo(int x, int y) {
2       int a = 0;
3       if (x > 10) {
4           a = 5;
5       } else {
6           a = -5;
7       }
8       if (y == 1) {
9           ++a;
10      } else {
11          --a;
12      }
13      return a;
14  }
```

Figure 4.1: Sample code snippet for state merging

After two rounds of state merging, we have only one state. By comparing the results with and without state merging, we find that in the latter case, there are an exponential number of states, concrete values for variable a, and complex path constraints. In contrast, with state merging, there are much fewer states, much simpler path constraints, but com-

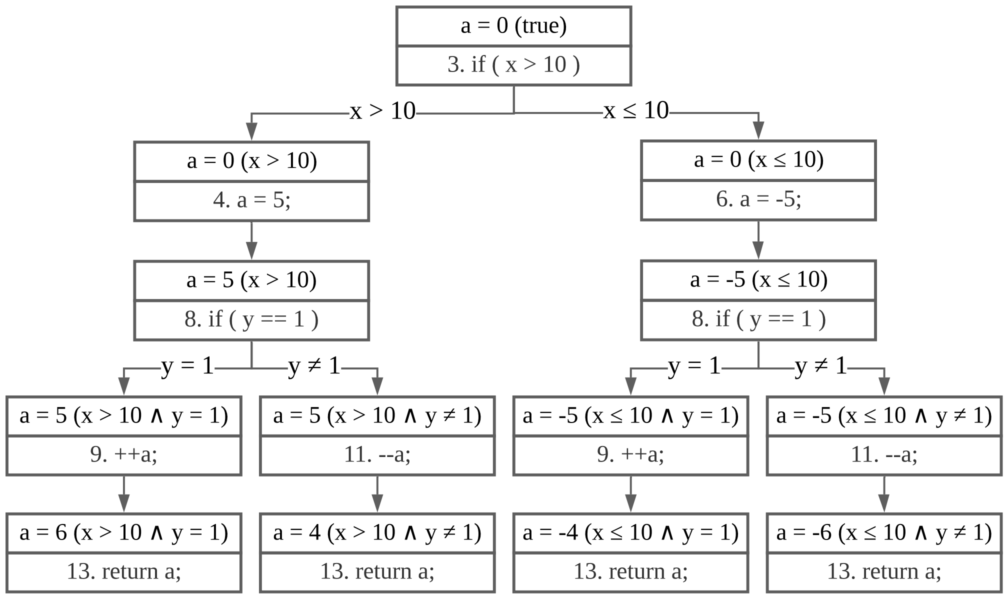


Figure 4.2: Symbolic execution without state merging for the example in Figure 4.1

plex expressions for symbolic variables. Because ITE expressions introduce more complex expressions that are translated into disjunctions, they can cause significant overhead in constraint solving, and eventually negate the benefits from state merging [62]. Essentially, we are shifting the burden from the symbolic executor to the constraint solver, and thereby need a good balance. Kuznetsov et al. [73] provide insights into the problem and show that two states should be merged if the variables they differ in, are less frequently used in later queries to the constraint solver.

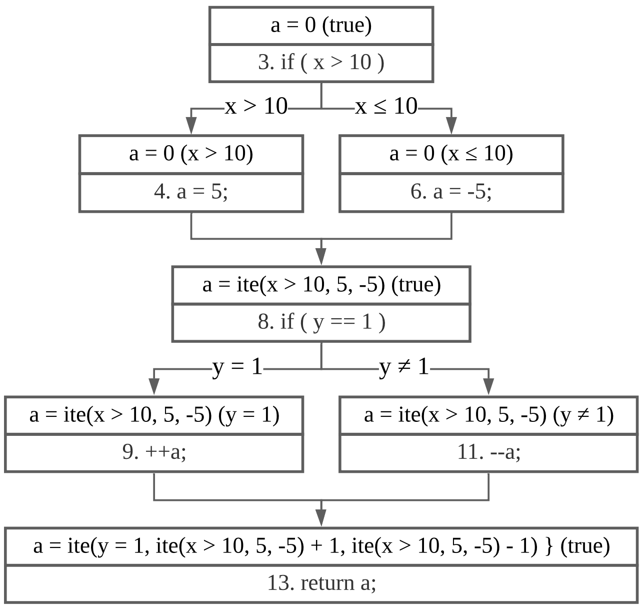

a = 0 (true)
3. if (x > 10)
x > 10
x ≤ 10
a = 0 (x > 10)
4. a = 5;
a = 0 (x ≤ 10)
6. a = -5;
a = ite(x > 10, 5, -5) (true)
8. if (y == 1)
y = 1
y ≠ 1
a = ite(x > 10, 5, -5) (y = 1)
9. ++a;
a = ite(x > 10, 5, -5) (y ≠ 1)
11. --a;
a = ite(y = 1, ite(x > 10, 5, -5) + 1, ite(x > 10, 5, -5) - 1) } (true)
13. return a;

Figure 4.3: Symbolic execution with state merging for the example in Figure 4.1

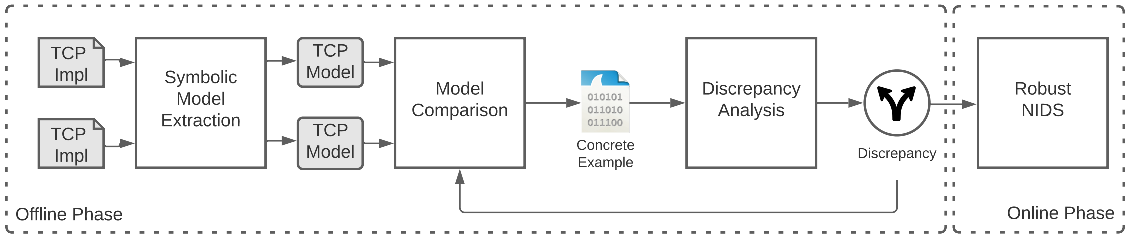


Figure 4.4: System Overview of Themis

4.3 Offline Phase: Discovering TCP Implementation Discrepancies

The offline phase of Themis that finds discrepancies between any two TCP implementations has three key components, as shown in Figure 4.4. The first component is "Symbolic Model Extraction", which runs symbolic execution exhaustively on different versions of TCP implementations and extracts high-fidelity models to accurately reflect detailed behaviors of each implementation. The second is called "Model Comparison," which compares two symbolic models and automatically generates concrete examples that will trigger the discrepancies between them. The last is "Discrepancy Analysis," which empirically analyzes the execution traces corresponding to the concrete examples and determines the root cause of a discrepancy in the behaviors between the two implementations. The process is iterative in that we feed the discrepancies summarized from "Discrepancy Analyis" back to the "Model Comparison" to exclude them from the models in the next round of concrete example generation, until there are no discrepancies between the two models. These discrepancies will also be integrated into the NIDS to enable online operations of THEMIS as discussed in §4.4.

4.3.1 Symbolic Model Extraction

Finding low-level discrepancies between two TCP implementations is a daunting task, because of the huge number of possible states in the program. Such discrepancies could be buried deep, in some rarely visited states. To formalize, discrepancies occur when two implementations generate different outputs given the same input. With respect to the NIDS evasion attacks, a discrepancy occurs when two TCP implementations produce different reassembled data streams, when they receive the same sequence of TCP packets. To aid the discovery of discrepancies, we first define a set of critical states S as intermediate states that precede our target output (which is the reassembled data in the TCP receive buffer). A critical state s consists of a set of state variables and specific values. $s = \{s_i = d_i\}$. We consider two types of critical states as follows: (a) TCP states, e.g., LISTEN, SYN_RECV, ESTABLISHED, CLOSE, and (b) receive buffer events, e.g., whether a packet enters the

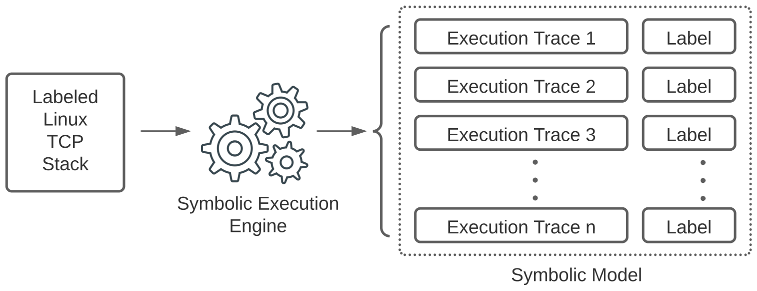


Figure 4.5: Symbolic Model Extraction

TCP in-order queue or out-of-order queue. Intuitively, if the same sequence of input packets drives two TCP implementations into two different TCP states or into accepting different payloads in the TCP receive buffers, such discrepancies are bound to be exploitable.

As shown in Figure 4.5, via exhaustive symbolic execution, we extract a mapping M between the path constraints Π and the critical states S, denoted as $M : \Pi \rightarrow S$. Since path constraints are constraints on the inputs, this translates to summarizing the relationships between inputs and critical states. By combining all the path constraints that lead to a critical state with disjunction, we can automatically obtain the *weakest precondition* [46] of the critical state. The weakest precondition, denoted as $wp(S, R)$, is the condition that characterizes all possible initial states making a system S terminate in a final state that establishes the truth of an assertion (post-condition) R. The term "weak" or "strong" allude to how general or specific a condition is. The weakest precondition is basically the most general constraints that should be satisfied in order to satisfy a given postcondition. Weakest precondition is commonly used in the generation of verification conditions [46]. In our setting, the postconditions are the critical states that we label.

Therefore, we have the following equation, in which I denotes the TCP implementation:

$$wp(I, s) = \bigvee_{M(\pi_i)=s} \pi_i \tag{4.1}$$

To provide a concrete example of a discrepancy leading to differing path constraints in different versions of the TCP implementation, we show in Listing 4.1 and Listing 4.2 how Linux validates incoming RST packets in different versions. In Linux kernel versions before 3.6, when in the ESTABLISHED state, it accepts a RST packet as long as its sequence number is within the current receive window (Line 6); it then resets the connection and enters the CLOSE state (Line 13). In versions after 3.6, Linux developers implemented the defense mechnism from RFC 5961 [94], which performs a much stricter check on RST packets. These versions only accept a RST packet if its sequence number exactly matches the next expected sequence number *rcv_nxt* (Line 19). In this case, the two implementations differ in the path constraints relating to the CLOSE state. In the earlier versions, the differing path constraints include $rcv\_nxt < seq\_num < rcv\_nxt + window\_size$; in the latter versions, the differing path constraints include $seq\_num = rcv\_nxt$.

```
static int tcp_validate_incoming(struct sock *sk, struct sk_buff *skb,
                  struct tcphdr *th, int syn_inerr)
{
    ...
    /* Step 1: check sequence number */
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        ...
        goto discard;
    }

    /* Step 2: check RST bit */
    if (th->rst) {
        tcp_reset(sk);
        goto discard;
    }
    ...
}
```

Listing 4.1: Validation of RST packets in Linux kernel versions before 3.6

```
static bool tcp_validate_incoming(struct sock *sk, struct sk_buff *skb,
                  const struct tcphdr *th, int syn_inerr)
{
    ...
    /* Step 1: check sequence number */
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        ...
        goto discard;
    }

    /* Step 2: check RST bit */
    if (th->rst) {
        /* RFC 5961 3.2 :
         * If sequence number exactly matches RCV.NXT, then
         *     RESET the connection
         * else
         *     Send a challenge ACK
         */
        if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
            tcp_reset(sk);
        else
            tcp_send_challenge_ack(sk, skb);
        goto discard;
    }
    ...
}
```

Listing 4.2: Validation of RST packets in Linux kernel versions after 3.6

4.3.1.1 Eliminating Non-determinism in TCP Processing

To extract a deterministic model from a TCP implementation, we need to first eliminate any non-determinism in TCP. Non-determinism can cause symbolic execution to explore different parts of the code in different runs, and thus causes false positives, i.e., "discrepancies" found between non-deterministic models may not exist between the actual implementations. Note that non-determinism is always introduced by concrete inputs, because symbolic inputs will enable forking in symbolic execution and exploration of all feasible paths. One example of such concrete inputs is a random number generated during execution, e.g., the initial sequence number in TCP. Since we model the server-side logic of a TCP implementation, we assume the client-side sequence number is controllable by the attacker and thus symbolize it. Meanwhile, we hook the random number generator for the server-side initial sequence number and coerce it to always return a fixed number to eliminate non-determinism.

Other non-determinism may be introduced due to variations in the execution times of symbolic execution; this would influence factors such as (but not limited to): 1) Timeouts (e.g., TCP connection timeout, packet transmission timeout); 2) Congestion control window size computations; 3) Round-trip time (RTT) calculations; 4) Receive buffer size computations; 5) Delayed ACK computations; 6) MTU probing; 7) Rate-limits (e.g., out-of-window ACKs, challenge ACKs); 8) Socket locking by the user thread (affected by the timing of kernel and user thread switching). To eliminate the non-determinism introduced by the variation of execution time, we hook the TCP access to the system clock and always return deterministic values. This could potentially lead to reduced code coverage (e.g., no timeouts). We argue that this is a reasonable decision because even if a discrepancy exists in such timing-related code blocks, it can be unreliable to use such a discrepancy to perform an evasion attack. In fact, we have not seen any report of such discrepancies leveraged to that effect. In our experiments, we freeze the clock by always returning the same exact value. Interestingly, we also tried using monotonically increasing values for the clock, but it resulted in even lower code coverage.

4.3.1.2 State Merging to Achieve Scalability

To handle path explosion in symbolic execution, we adopt the idea of state merging from [73]. There is a gamut of state merging options on a program, from complete separation of individual execution traces (no merging) to aggressively merging two states whenever their execution traces join; the latter is also called static state merging. As discussed in §4.2, state merging reduces repetitive work (of executing the same code blocks) in symbolic execution at the cost of introducing harder-to-solve symbolic formulas in constraint solving. Aggressive state merging may even harm performance rather than improve it [62]. Hence, we employ state merging following the general suggestions from [73] as well as the domain knowledge of TCP.

Specifically, we first collect a list of fork points during symbolic execution with an initial run. Then we mark *merge range candidates* with the fork points as the starting points and their immediate post-dominators as the ending points. The start and end points form candidate merge ranges. After that, we manually inspect each candidate and the variables being modified within it, and decide whether to merge based on the following heuristics: 1) the critical state variables should not be modified within the merge range; otherwise, the critical states may become symbolic after merging and thus, it becomes complicated to group execution traces by critical states in the later phase; 2) no new packets should be generated/sent, and no dynamic memory allocation or deallocation should occur within the candidate range; otherwise, additional symbolic memory will be created inducing extra overheads subsequently; 3) no excessive number of variables modified within the candidate range (especially if there are TCP-related state variables that will be used heavily subsequently); otherwise, extra complexity will be introduced in constraint solving later. Note that these heuristics can be potentially automatically applied with the help of static analysis. Nevertheless, we consider it an orthogonal component which can be improved upon separately. We leave the automation of the merge range determination as a future work.

Finally, during the actual symbolic execution, the labelled merge ranges will be applied accordingly. Note that the merge ranges can be nested. We will always merge the innermost ranges and then the outer ones.

4.3.2 Model Comparison

The symbolic model extracted in the previous step is in the form of a mapping from path constraints on inputs, to critical states, as shown in Figure 4.5. Here the critical states can be considered as the intermediate states that are directly related to the output state, which is the reassembled data stream that will be passed to the application layer. Instead of finding the differences in critical states given the same input, we try to find the

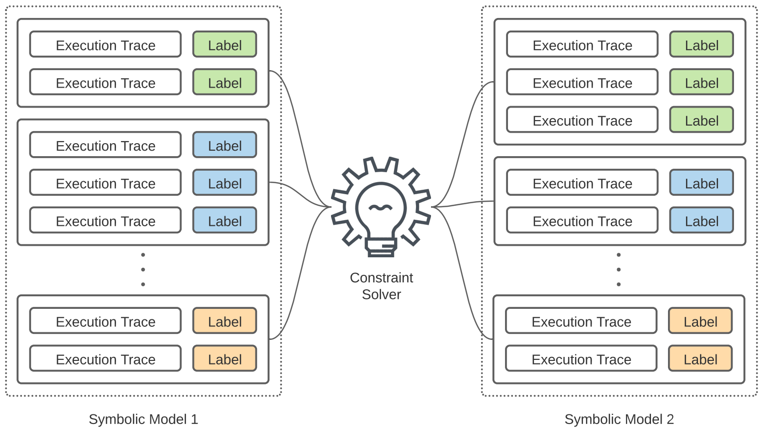


Figure 4.6: Symbolic Model Comparison

differences in inputs given the same critical state. Specifically, we group the exectuion traces by critcal states, and then combine their path constraints with disjunction. The combined path constraints reflect all possible inputs that will drive the TCP implementation into the critical state, which is also equivalent to the weakest precondition of the critical state. Then we compare the combined path constraints from the two different TCP implementations, for each of the critical states, as shown in Figure 4.6. Note that the path constraints are represented in a format (SMT-LIB [108]) that can be directly processed by constraint solvers; thus, we can easily test if two path constraints are equivalent using state-of-the-art constraint solvers such as Z3 [136]. The constraint solver will either prove that the two path constraints are equivalent or generate a concrete counterexample that is accepted by one of the path constraints but not the other.

4.3.3 Discrepancy Analysis

From the counterexample generated from the last step, we can craft TCP packets that will trigger the discrepancy between the two TCP implementations. However, our goal is to learn a class of packets that belong to a specific discrepancy (e.g., RST packets with in-window sequence number) and then, summarize the discrepancy in general. To this end, we first feed the TCP packets generated from a counterexample, to both TCP implementations and record the execution traces, respectively. Then, we manually reason about the root cause of the difference between the two execution traces, to determine the critical conditions that trigger the difference. Since the execution traces are from two different implementations, we cannot directly compare them to find the discrepancy. Further, they are at the binary level and little information is provided. To ease the analysis, we translate the binary-level execution traces to source-code-level execution traces. From there, we can focus on the symbolic branch traces and easily identify the differences and the critical branches. Subsequently, we summarize the difference in a symbolic formula, which can be used to identify the discrepancy. The symbolic formula is fed back to the model comparison phase, to exclude the discrepancies that have already been found; the process continues to generate new counterexamples to discover new discrepancies until none exist. The algorithm is shown in Algorithm 1. For example, with regard to the discrepancy discussed in §4.3.1, the symbolic formula is a constraint that matches all packets with the RST flag set and the sequence number in window but does not match the exact rcv_nxt in the ESTABLISHED state. We exclude this constraint from all execution traces starting with the ESTABLISHED state by performing a conjunction with the negation of the constraint on the original path constraints. This fulfills the goal of exclusion of the discrepancy.

An alternative workflow is to find a counterexample, exclude the path constraints corresponding to that counterexample from both symbolic models, and then find the next

Algorithm 1 Finding discrepancies between two implementations

```
1: function FindAllDiscrepancies($I_1, I_2$)
2:     $AllDiscrepancies \leftarrow \emptyset$
3:     $M_1 \leftarrow$ ExtractSymbolicModel($I_1$)
4:     $M_2 \leftarrow$ ExtractSymbolicModel($I_2$)
5:     for $s \in$ all critical states in $M_1$ or $M_2$ do
6:         $Discrepancies \leftarrow$ CompareSymbolicModels($M_1, M_2, s$)
7:         $AllDiscrepancies = AllDiscrepancies \cup Discrepancies$
8:     end for
9:     return $AllDiscrepancies$
10: end function
11: function CompareSymbolicModels($I_1, I_2, M_1, M_2, s$)
12:     $Discrepancies \leftarrow \emptyset$
13:     $\Pi_1 \leftarrow \bigvee_{M_1[\pi]=s} \pi$
14:     $\Pi_2 \leftarrow \bigvee_{M_2[\pi]=s} \pi$
15:     $Result, CounterExample \leftarrow$ SolveConstraints($\Pi_1 = \Pi_2$)
16:     while $Result = unsat$ do
17:         $Discrepancy \leftarrow$ DiscrepancyAnalysis($I_1, I_2, CounterExample$)
18:         $Discrepancies.insert(Discrepancy)$
19:         $\Pi_1 \leftarrow \Pi_1 \backslash Discrepancy.constraints$
20:         $\Pi_2 \leftarrow \Pi_2 \backslash Discrepancy.constraints$
21:         $Result, CounterExample \leftarrow$ SolveConstraints($\Pi_1 = \Pi_2$)
22:     end while
23:     return $Discrepancies$
24: end function
25: function DiscrepancyAnalysis($I_1, I_2, CounterExample$)
26:     $ExecTrace_1 \leftarrow$ TraceExecution($I_1, CounterExample$)
27:     $ExecTrace_2 \leftarrow$ TraceExecution($I_2, CounterExample$)
28:     $Discrepancy \leftarrow$ RootCauseAnalysis($ExecTrace_1, ExecTrace_2$)
29:     return $Discrepancy$
30: end function
```

counterexample. This would decouple the Symbolic Model Comparison from the Discrepancy Analysis, and make the former fully automated. However, there could be a large number of execution traces corresponding a single discrepancy, and so it could take much longer to exclude a discrepancy than using the feedback from the Discrepancy Analysis directly (our initial studies indicate this is the case). Besides, an execution trace may cover more than one discrepancy and thus the exclusion of an entire trace may remove more than the current discrepancy. Thus, we did not pursue this second approach in our work.

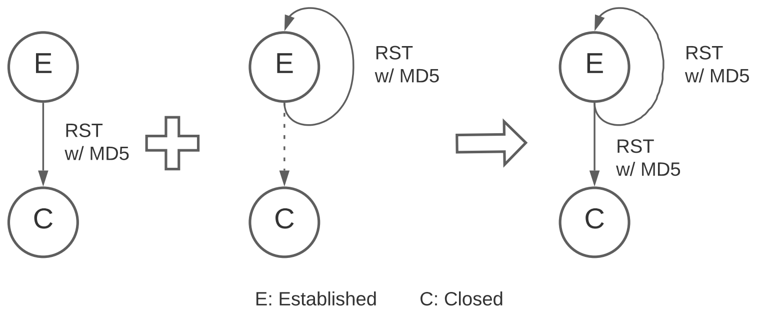


Figure 4.7: Merging DFAs into NFA

4.4 Online Phase: Ambiguity-Aware NIDS

4.4.1 NFA-Based Model for NIDS

Stateful network protocols are usually modeled as Deterministic Finite Automata (DFA) to make sure that different parties associated have deterministic behaviors and are well-synchronized. NIDSs also use DFA-based network protocol implementations (as do the endhosts). However, this makes them conform to a specific version of a protocol implementation, and have discrepancies with other versions. This leaves opportunities for attackers to evade them. In order to ensure compatibility with different versions of the network protocol implementation, we propose a novel, NFA-based model for the NIDS. In Nondeterministic Finite Automata (NFA), upon receiving an input, the state could transition into any of multiple different new states non-deterministically. If there are any possibilities that the state transitions into an accepting state, the input is accepted. In practice, an NFA ”clones” its state when there are multiple possible next states. Thus, it enables the NIDS to handle packets with ambiguities i.e., compatible with different packet handling logics of different versions.

Discrepancies across TCP implementations project ambiguities to the NIDS when handling packets. We define an ambiguity as a 3-tuple, $(\varphi, \lambda_1, \lambda_2)$, derived from the discrepancies found by THEMIS as discussed in §4.3. φ denotes the symbolic formula characterizing all possible inputs that trigger the behavioral differences between the two implementations. λ_1 and λ_2 denote the differing behaviors of the two respective implementations. We integrate the ambiguities into the existing DFA of the NIDS and turn it into an NFA. φ is the guard or predicate of the transition. There are two output states, and λ_1 and λ_2 are the corresponding transition functions. In this way, we are merging multiple DFAs into an NFA while reusing the common parts in the DFAs to the maximum extent possible. An example is shown in Figure 4.7. When processing a RST packet with a TCP MD5 option, an earlier version of Linux accepts it while a later version discards it. After merging the two DFAs, the NFA will explore both possibilities in parallel. Rather than running two DFAs side by side, THEMIS allows maximized reusability of the code and expedites packet processing.

When processing a packet that causes an ambiguity, the NFA-based NIDS will fork its currently maintained state for the TCP connection and process it with behavior λ_1 and behavior λ_2, respectively. Note that for each connection and ambiguity, we only fork once and remember the behavior associated with the copy of the connection state. Assuming there are k ambiguities, then the upper bound of the number of forked connection states is 2^k. If the attacker is also aware of the ambiguities, then he can maliciously inject them into the traffic and cause an exponential growth in the number of connection states on the NIDS, as a resource exhaustion attack. As a further optimization, we take version coherence of the behaviors into account and reduce the growth rate to a linear rate.

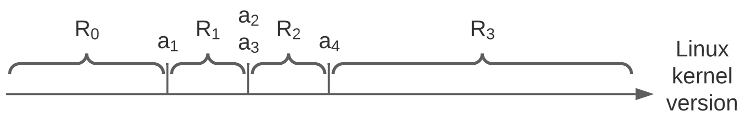


Figure 4.8: Example: version ranges defined by ambiguities

4.4.2 Version Coherence

In this work, we focus on the Linux TCP stack and try to find discrepancies between TCP stacks of different versions of the Linux kernel. We choose n Linux kernel versions to analyze and sort them by version numbers, denoted as $v_i, i \in [1, n]$. Then we compare each pair of adjacent Linux kernel versions using the approach described in §4.3, and summarize the discrepancies into ambiguities. For a specific ambiguity $(\varphi, \lambda_1, \lambda_2)$, we associate λ_1 with the earlier version of the Linux kernel, e.g., v_1, and λ_2 with the later version, e.g., v_2. Then, we know that the changes made to the TCP stack causing the ambiguity was introduced between v_1 and v_2. An important insight here is that **all versions before the ambiguity was introduced, should all conform with the old behavior, and all versions after the ambiguity was introduced, should all conform with the new behavior.** Thus, all versions before v_1 must conform to λ_1, and all versions after v_2 must conform to λ_2.

Ambiguities divide the version space into version ranges. All kernels in the same version range behave the same with respect to all ambiguities. We sort the ambiguities by the versions wherein they are introduced, and denote them as $a_j, j \in [1, m]$. Note that we do not know the exact version in which an ambiguity was introduced; we can only approximate, using the versions we have analyzed. We analyzed every pair of adjacent kernel versions, and for each pair we list the ambiguities found between the two versions. The ambiguities found between the same pair have the same order. The ambiguities found between different pairs are sorted by the version pairs. For example, let us say a_1 is found between v_1 and v_2, a_2, a_3 are found between v_2 and v_3, and a_4 is found between v_3 and v_4. Then their relationship would be $v_1 < a_1 < v_2 < a_2 = a_3 < v_3 < a_4 < v_4$. We use R_i to denote the version range defined by two adjacent ambiguities, viz., a_i and a_{i+1}. R_0 is the version range before a_1 and R_m is the version range after a_m. For example, in Figure 4.8, there are 4 ambiguities, $a_1 < a_2 = a_3 < a_4$, dividing the version space into 4 version ranges. Version range R_0 should conform to the old behaviors for all ambiguities, denoted as $(\lambda_{1,1}, \lambda_{2,1}, \lambda_{3,1}, \lambda_{4,1})$. Version range R_2 should have the old behavior for ambiguity 4 and new behaviors for ambiguities 1, 2, and 3, denoted as $(\lambda_{1,2}, \lambda_{2,2}, \lambda_{3,2}, \lambda_{4,1})$. k ambiguities can define at most $k+1$ version ranges. Note that we only need to maintain one connection state copy for each version range, as the versions in the same version range all behave the same regarding all ambiguities. Thus, we have at most $k+1$ forked states for each connection.

In Algorithm 2, we show the algorithm used in THEMIS for handling packets with ambiguities in the NIDS.

4.5 Evaluation

In this section, we first evaluate THEMIS's symbolic-execution-based discrepancy discovery, and list the discrepancies found between different versions of the Linux kernel. Next, we evaluate THEMIS augmented NIDS by integrating all the discrepancies discovered,

Algorithm 2 Handling packets with ambiguities in NIDS

```
procedure OnPacketReceived($Packet$)
    $NewConnStates \leftarrow \emptyset$
    for $ConnState \in AllConnStates$ do
        $ConnState.Ambiguities \leftarrow$ CheckAmbiguities($Packet, ConnState$)
        for $AmbiguityID \in ConnState.Ambiguities$ do
            if $ConnState.Behaviors[AmbiguityID] = Undefined$ then
                $NewConnState \leftarrow$ Fork($ConnState$)
                for $i \in [AmbiguityID, MaxAmbiguityID)$ do
                    $NewConnState.Behaviors[i] \leftarrow Old$
                end for
                for $i \in [0, AmbiguityID)$ do
                    $ConnState.Behaviors[i] \leftarrow New$
                end for
                $NewConnStates.insert(NewConnState)$
            end if
        end for
    end for
    $AllConnStates \leftarrow AllConnStates \cup NewConnStates$
    for $ConnState \in AllConnStates$ do
        HandlePacketWithAmbiguities($Packet, ConnState$)
    end for
end procedure
procedure HandlePacketWithAmbiguities($Packet, ConnState$)
    for $AmbiguityID \in ConnState.Ambiguities$ do
        if $ConnState.Behaviors[AmbiguityID] = Old$ then
            Implementation of the old behavior
        else if $ConnState.Behaviors[AmbiguityID] = New$ then
            Implementation of the new behavior
        end if
    end for
    Processing packets without ambiguities
end procedure
```

and demonstrate its (1) effectiveness in defending against all existing and newly discovered evasion attacks and (2) performance overhead at runtime.

4.5.1 Symbolic-execution-based Discrepancy Discovery

Themis's offline component described in §4.3, is built upon S2E [37] and Z3 [136]. We implemented our TCP symbolic execution as S2E plugins, and implemented model comparison and deviation analysis with Python scripts using Z3 as the underlying constraint solver. We run Themis on a machine with an AMD EPYC 7542 32-core 64-thread CPU, and run symbolic execution with 64 processes in parallel.

4.5.1.1 Performance of Symbolic Execution and Model Comparison.

In our exhaustive symbolic execution, we bound the input to 3 symbolic TCP packets with payloads and run until all execution paths finish. We symbolize the TCP header fields except the TCP checksum. This can cover all passive TCP states, including LISTEN, SYN_RECV, NEW_SYN_RECV, ESTABLISHED, CLOSE_WAIT, CLOSE, and critical states related to receive buffer (e.g., receipt of in-order data or out-of-order data). Other TCP states require the endhost to actively initiate or close a connection. Since we are modeling the servers' behaviors, we leave the exploration of these other TCP states to future work.

In our initial attempt of exhaustive symbolic execution without state merging on the Linux TCP stack (on version 4.4), we send 3 symbolic packets without any TCP options; it took 13.5 hours on average to finish. The total number of execution paths is 1,219,938. After enabling state merging, it takes less than 4 minutes to finish, and the total number of execution paths decreases to 1386. Using the heuristics provided in §4.3, we labeled 24 merge ranges. They are mainly related to ECN (Explicit Congestion Notification), window size, MSS (Maximum Segment Size), and urgent pointer. It takes approximately two days on average, for a domain expert to do the manual labeling for a Linux kernel version. In addition, without state merging, it takes more than a week to compare the rather huge models (with more than one million paths) extracted from Linux kernel versions 4.4 and 5.4. Instead, it takes only 15 seconds to compare two models (with about 1000$\sim$2000 paths) after enabling state merging.

Table 4.1: Number of execution paths grouped by critical states in different versions of Linux kernel

Version	All	SR	EST	CW	CL	IO	OOO
3.0	2966	2841	919	588	84	372	1454
3.10	2344	2128	574	350	40	284	796
4.4	1386	1170	302	181	20	149	410
5.4	2214	1998	729	650	49	428	1140
5.10	2314	2250	863	678	51	440	1322

* SR - SYN_RECV/NEW_SYN_RECV; EST - ESTABLISHED; CW - CLOSE_WAIT; CL - CLOSE; IO - In-order data; OOO - Out-of-order data.
* The results are accquired with state merging enabled.

4.5.1.2 Themis's Discrepancy Discovery

We analyzed 5 major LTS versions of the Linux kernel from 3.0 to 5.10, viz., 3.0, 3.10, 4.4, 5.4, and 5.10. For each version, we run exhaustive symbolic execution with state merging and send 3 symbolic packets, without and with TCP options. Since TCP options are usually not correlated with each other, we try each TCP option individually instead of combining them. We also symbolize the values in each TCP option. For the experimental results without TCP options, the numbers of execution paths relating to each critical state are shown in Table 4.1. Note that, these numbers are based on merged execution paths, and are affected by the labeled merge ranges.

We list all the discrepancies found by THEMIS in Table 4.2. We also validate our findings with the commit history of the Linux kernel, and note the date and version when a discrepancy was first introduced. As one can see, most of the discrepancies were introduced around 2012, while some newer ones were introduced around 2017. A major reason contributing to these discrepancies, is the change proposed in RFC 5961 [94], a mitigation against blind in-window attacks. The RFC introduces stricter checks on the sequence and acknowledgment numbers in SYN, RST and data packets. This leads to the Discrepancies 2, 3, and 4. A second reason is buggy implementations when validating a TCP packet. Discrepancy 1 is caused by the older versions not doing a propoer validation on TCP flags in the LISTEN state, and accepting invalid TCP flag combinations, i.e., SYN+FIN. Discrepancy 5 is due to older versions not checking ACK flags when processing data packets. Discrepancy 9 is due to older versions mistakenly bypassing the acknowledgment number checking in certain states, e.g., CLOSE_WAIT, CLOSING, LAST_ACK. There are also other reasons stemming from performance improvements and compatibility with other operating systems. Discrepancy 6 was introduced by a fix to the implementation of the Fast Retransmit/Fast Recovery algorithm. Discrepancy 7 was introduced for performance optimization when SACK is enabled and packet loss happens frequently. Discrepancy 8 was introduced to handle an idiosyncrasy associated with Mac OSX clients, which may leave a connection that is supposed to be closed, in a lingering state.

Table 4.2: Discrepancies found between different versions of Linux kernels (from v3.0 to v5.10)

Discrepancy No.	1
Condition	In LISTEN state, received a SYN+FIN packet
Old Behavior	Initiate a connection
New Behavior	Discard
Date	12/3/2011
Version	3.3
Commit	`https://github.com/torvalds/linux/commit/fdf5af0daf8019cec2396cdef8fb042d80fe71fa`

Discrepancy No.	2
Condition	In ESTABLISHED state, received a SYN packet with in-window SEQ number
Old Behavior	Reset the connection
New Behavior	Discard and send a challenge ACK
Date	7/17/2012
Version	3.6
Commit	`https://github.com/torvalds/linux/commit/0c24604b68fc7810d429d6c3657b6f148270e528`

Discrepancy No.	3
Condition	In ESTABLISHED state, received a RST packet with in-window SEQ number but doesn't match the exact next expected SEQ number (rcv_nxt)
Old Behavior	Reset the connection
New Behavior	Discard and send a challenge ACK
Date	7/17/2012
Version	3.6
Commit	`https://github.com/torvalds/linux/commit/282f23c6ee343126156dd41218b22ece96d747e3`

Discrepancy No.	4
Condition	In ESTABLISHED state, received a packet with ACK number < prior_snd_una - max_window
Old Behavior	Accept the packet
New Behavior	Discard and send a challenge ACK
Date	12/22/2012
Version	3.8
Commit	`https://github.com/torvalds/linux/commit/354e4aa391ed50a4d827ff6fc11e0667d0859b25`

Discrepancy No.	5
Condition	Data packets without ACK flag
Old Behavior	Accept the payload
New Behavior	Reject the payload
Date	12/26/2012
Version	3.8
Commit	`https://github.com/torvalds/linux/commit/c3ae62af8e755ea68380fb5ce682e60079a4c388`

Discrepancy No.	6 (New)
Condition	In LISTEN state, received a SYN packet and created a new child socket
Old Behavior	The initial receive window size is 14600
New Behavior	The initial receive window size is 29200
Date	6/13/2013
Version	3.11
Commit	`https://github.com/torvalds/linux/commit/85f16525a2eb66e6092cbd8dcf42371df8334ed0`

Discrepancy No.	7 (New)
Condition	When SACK is enabled, received a RST packet with SEQ number = end of previously received rightmost SACK block
Old Behavior	Discard and send a challenge ACK
New Behavior	Reset the connection
Date	6/8/2016
Version	4.8
Commit	`https://github.com/torvalds/linux/commit/e00431bc93bb48c650273be4a00007b2a392d32a`

Discrepancy No.	8 (New)
Condition	In one of the closing states (CLOSE_WAIT/CLOSING/LAST_ACK), received a RST packet with SEQ number = rcv_nxt - 1
Old Behavior	Discard
New Behavior	Enter CLOSE state
Date	1/17/2017
Version	4.11
Commit	`https://github.com/torvalds/linux/commit/0e40f4c9593ba2c7c30150ed669da97bd581c0cd`

Discrepancy No.	9 (New)
Condition	In one of the closing states (CLOSE_WAIT/CLOSING/LAST_ACK), received a data packet with SEQ number < rcv_nxt and ACK number < prior_snd_una, but with partial in-window payload
Old Behavior	Discard
New Behavior	Enter CLOSE state
Date	5/25/2017
Version	4.13
Commit	`https://github.com/torvalds/linux/commit/d0e1a1b5a833b625c93d3d49847609350ebd79db`

We have manually inspected changes to the TCP stack in the Linux kernel from version 3.0 to 5.10, and confirmed that the discrepancies listed in Table 4.2 are true, and did not find any new discrepancies. In addition, we measured these discrepancies on Alexa's top 1 million websites from the client side, by sending probe packets to the servers and collecting responses. Although Linux-based servers have the largest market share, there are still other operating systems and variations. We take them as equivalent if they have the same behavior as a specific Linux version. We find that both older and newer behaviors are observed for all discrepancies, which means today's NIDSs can only either incorporate the older or the newer version but not both. This leaves the remaining servers vulnerable to attacks.

In addition to implementation-level discrepancies, we also found some discrepancies in default configurations. Although the TCP MD5 option was introduced in version 2.6.20 in 2006, it was an experimental feature and by default disabled until version 3.9 in 2013. The initial window sizes are different between versions 4.4 and 5.4; this is caused by different configuration values of the TCP receive buffer size, i.e., net.ipv4.tcp_rmem. The default values for minimum, default, and maximum size of the TCP receive buffer are (4096, 87380, 1887552) in version 4.4, but are (4096, 131072, 1772832) in version 5.4, as a result of an increasing demand in throughput.

4.5.1.3 Case Studies

In this section, we choose three of the newly discovered discrepancies from our analysis, and describe how to exploit them in today's NIDSs. Different from evasion attacks in previous works, we creatively re-use the four-tuple of a connection to exploit some of these discrepancies.

RST rightmost SACK (leading to Discrepancy 7) was introduceds in 2016 [99], as a performance optimization that allows a connection to be closed by a RST promptly, when packet losses or out-of-order packets are experienced. When packet losses or out-of-order packets occur, the rcv_nxt stays at the end of the previously received in-order data. After RFC 5961, TCP only accepts a RST packet if its sequence number is equal to rcv_nxt. So in this case, if an RST is sent after some lost or re-ordered segment, the server's rcv_nxt doesn't match the sequence number in the RST and the server will respond with a challenge ACK. In a lossy situation, the challenge ACK may be lost as well, and the connection will stay alive for a while. Therefore, the newer versions accept a RST packet as long as its sequence number matches the right edge of the right-most SACK block previously received. One can exploit this discrepancy via two possibilities: 1) if the server accepts such RST packets and the NIDS rejects them, then we can send such a RST packet to tear down the connection on the server, and then re-use the four-tuple to build a new connection with a different initial sequence number (ISN), which will not be tracked by the NIDS; 2) if the NIDS accepts such RST packets and the server rejects them, then we can simply craft such a RST packet to tear down the connection on the NIDS.

RST after FIN (leading to Discrepancy 8) is an optimization to handle compatibility issues with Mac OSX [98]. In Mac OSX, when some applications are abruptly terminated, a RST packet is sent after a FIN packet with the same sequence number as the FIN packet. When a Linux server receives the FIN packet, it advances the rcv_nxt by one; this causes the following RST packet to be rejected because of an out-of-window sequence number, and a challenge ACK to be sent. The MAC OSX client may not reply with any further RST packets, and the connection on the Linux server will be left in a closing state (e.g., CLOSE_WAIT). To prevent connections from staying in closing states in such cases, the newer versions of the Linux kernel also accepts RST packets with a sequence number equal to rcv_nxt - 1, when in a closing state. One can exploit this discrepancy via two possibilities: 1) if the server accepts such RST packets and the NIDS does not, then we can send such a RST to tear down the connection on the server, and then re-use the four-tuple to build a new connection; because the NIDS has not torn down the old connection, it will not be able to track the new connection; 2) if the NIDS accepts such RST packets and the server does not, then we can send such a RST to tear down the connection on the NIDS, and then re-use the four-tuple to send a SYN packet which will create a half-open connection on the NIDS; after that, we send a legitimate RST packet to tear down the connection on the server, and then re-use the four-tuple to create a new connection with a different ISN; because the NIDS already has a half-open connection, it will miss the new connection.

Data in closing states (leading to Discrepancy 9) will reset the connection in older versions of the Linux kernel because of a buggy implementation [43]. In older versions, when in one of the closing states (e.g., CLOSE_WAIT, CLOSING, LAST_ACK), a data packet with stale sequence and acknowledgment numbers but partial-in-window data will cause the connection to be reset. Although the acknowledgment number is checked, the result is not used, and the packet is not discarded immediately but processed further. In new versions, this bug was fixed, and such data packets will be discarded and trigger a challenge ACK or duplicate ACK. In order to exploit this discrepancy, there are two possibilities: 1) if the server accepts such data packets and the NIDS does not, then we can send such a data packet to reset the connection on the server, and then re-use the four-tuple to build a new connection; assuming the NIDS has not torn down the old connection, the new connection will not be tracked by the NIDS; 2) if the NIDS accepts such data packets and the server does not, then we can send such a data packet to reset the connection on the NIDS, and then send a SYN packet with the same four-tuple to create a new half-open connection on the NIDS; after that, we send a legitimate RST packet to tear down the connection on the server, and then re-use the four-tuple to create a new connection with a different ISN; the new connection will not be tracked by the NIDS.

4.5.2 Themis Online Evaluations

In order to understand the effectiveness and efficiency of a NIDS empowered with Themis, we conduct two evaluations. Specifically we assess its robustness against evasion strategies and overhead performance in runtime, and compare our results with a state-of-the-art defense (very recent) [138] against such attacks; this recent approach is based on Deep Learning (DL) models and has disclosed its pipeline implementation and dataset [57].

4.5.2.1 NIDS Implementation

As mentioned previously, our NIDS should behave according to real TCP implementations in Linux instead of over-simplified implementations as found in today's NIDSs. However, for ease of implementation, we chose to develop our NIDS on top of Zeek (formerly Bro) [87], one of the most popular open-source general-purpose NIDSs in the market. First, we have to realign its behaviors to the common behaviors of the Linux versions we support; Second, we implement the different behaviors regarding each of the discrepancies we discovered; Third, we implement the logics of connection state forking and ambiguity detection.

Overall, we extend Zeek version 4.0.0 with only 1970 lines of C++ code to handle 8 of the discovered discrepancies listed in Table 4.2 [1]. Note that realigning Zeek to a Linux implementation is relatively straightforward and introduces negligible overhead.Hereon, we refer to the realigned version of Zeek as ambiguity-agonistic which is the baseline in our overhead evaluation, to be distinguished from Themis which is ambiguity-aware. We will open source our implementation and associated datasets for reproducibility and future extensions, at the time of publication.

4.5.2.2 Effectiveness

First, we evalute the effectiveness of Themis in defending against evasion attacks. Over the past years, there are a number of evasion strategies proposed in [92, 68, 125, 74, 126, 20]. In order to maximize the coverage of our evaluation, we thoroughly analyze all attacks presented in these works and picked strategies that are related to ambiguities in TCP. We summarize and implement 34 different strategies after merging redundant ones, including strategies that leverage the new discrepancies discovered by Themis. A detailed list of all implemented strategies can be found in the Appendix. We even design composite strategies that leverage multiple discrepancies in a single connection. Note these strategies fully cover the evaluated attacks in [138], and thus, we are able to conduct an apples-to-apples comparison. Our robust NIDS can detect these attacks with a success rate of 100%

[1]Discrepancy #6 is excluded because the ambiguity can be easily eliminated by looking at the advertised window size in the server's response packet.

Table 4.3: Breakdown of ambiguities present in the 8-day MAWI dataset used in evaluations

Ambiguity No. (from Table 4.2)	Connections	Ratio	Ambiguity No. (from Table 4.2)	Connections	Ratio
1	5	0.000007%	5	3	0.000004%
2	0	0%	7	20	0.00002%
3	31,149	0.043%	8	34,343	0.00047%
4	4,723	0.0065%	9	0	0%
No Ambiguity	72,383,094	99.903%	Total	72,453,189	100%

Discrepancy #6 is excluded as discussed in §4.5.2.1

(i.e., malicious payloads that are veiled by evasion attacks, can elude the ambiguity-agnostic Zeek but not our robust version). In comparison, [138] reports an Area Under the Receiver Operating Characteristic Curve (AUC-ROC) of 0.963 in detecting state-of-the-art NIDS evasion attacks, meaning it still produces a considerably large number of false positives as well as negatives.

4.5.2.3 Operational Runtime Overhead

In addition, we evaluate the overheads incurred due to Themis at runtime, compared to the ambiguity-agnostic Zeek when no malicious evasion attacks are present. Note that even without malicious evasion attacks, there can be a number of ambiguous packets observable in natural network traffic. This is because in a wild Internet environment, various implementations may exist and the packets exchanged across them may satisfy the conditions associated with ambiguities. In such cases, these ambiguities are not actively exploited for malicious purposes, but can still cause overhead since Themis would still fork states on such bases. We refer to this overhead as operational runtime overhead associated with Themis when it is deployed in real network environments.

The key to accurately estimating the operational overhead is finding representative network traffic captures to evaluate Themis. For this, we use the MAWI Traffic Archive [54] as the base dataset. It provides PCAP dumps from a backbone network located in Japan, and is thus, considered sufficiently large and representative. We pick 7-day recent traces captured from April 25 to May 1, 2021, in additon to the trace on April 7, 2020 (i.e., the dataset used in [138]), and filter out any non-TCP connections to forge a test set of 72,453,189 TCP connections. Table 4.3 shows the statistics of different ambiguities present in the trace. Overall, only a very small fraction of natural/benign traffic contain packets that cause ambiguities. Specifically, there are only 69,994 (0.097% of the connections) connections with exactly 1 ambiguity, 131 with 2 different ambiguities, 1 with 3 different ambiguities, and no connections with more than 3 ambiguities. As for the resulting operational overhead, we find that compared to ambiguity-agnostic Zeek, our robust version incurs only about 1.07% additional processing time, indicating only negligible levels of operational cost. The average processing time is 69400.5 packets per second. In comparison, the state-of-the-art defense from [138], can only process less than 2200 packets per second (due to the computational cost of the deep learning model), which is more than 30 times slower than Themis.

4.5.2.4 Overhead Growth with Multiple Ambiguities

In addition to the operational overhead incurred on benign traffic traces, we are also interested in evaluating the runtime overhead that Themis imposes in the presence of multiple different ambiguities in a single connection. Although, natural traffic rarely includes more than one ambiguity in the same connection (only ∼0.0002% in our 8-day MAWI dataset), we are interested in knowing how would the overhead of Themis grow with repsect to the number of ambiguities, for understanding how vunlunerable Themis is agaisnt Denial-of-Service attacks (i.e., deliberately injected multiple ambiguities for slowing down NIDS processing). To maliciously induce extremely high overhead, an attacker could aim to trigger as many state forkings as possible for each connection. Because each packet will be processed by all forked states, the overhead is proportional to the number of forked states. Furthermore, the attacker may want to use long-lived connections, because after reaching the maximum number of forked states, every packet sent by the attacker will cause significant extra overhead on the NIDS. Based on this reasoning, we manually inject ambiguities into normal connections, and measure the processing time growth of Themis along with number of ambiguities. The results are shown in Figure 4.9, and suggest that the overhead growth is in principle linearly proportional to the number of ambiguities. Note that the overhead stops growing after 6 ambiguities. This is because some ambiguities, specifically 7, 8, and 9, cannot coexist in the same connection due to version coherence and connection being reset. For example, versions before ambiguity 7 all conform with the old behavior for ambiguity 7, 8 and 9, and for versions after ambiguity 7, the connection will be reset after seeing ambiguity 7. In addition, based on the results in §4.5.2.3, it's rare to encounter a connection with more than 3 ambiguities, therefore, we could safely mark a connection as suspicious if it has more than 3 ambiguities.

4.6 Discussion and Limitations

Completeness and Soundness of Symbolic Modeling. By performing exhaustive symbolic execution, we achieve a complete coverage of feasible execution paths by

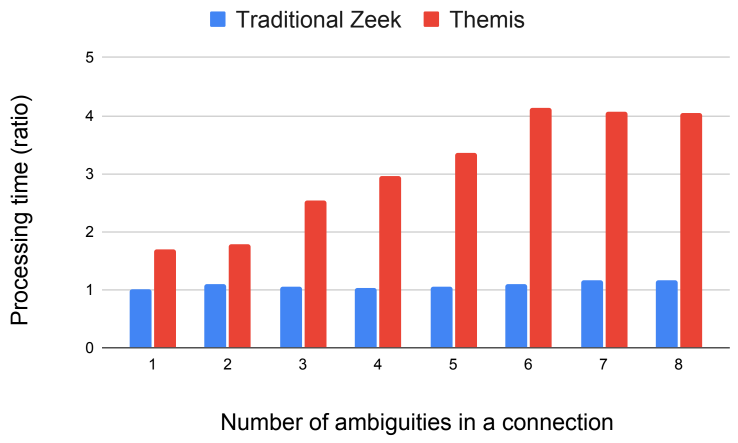


Figure 4.9: Overhead growth in a connection by number of ambiguities

finishing all execution states. However, there still could be potential cases missing. First, we use a specific configuration of Linux (e.g, kernel compilation configuration, Linux sysctl settings), and thus, cannot guarantee all TCP logic are covered (e.g, some features may not be enabled). Second, we eliminate some non-determinism in TCP (e.g., we freeze the CPU clock), and this could also cause incomplete coverage because we only explore one possibility instead of all. Further, some TCP logics need to be triggered by user space applications, which can set TCP socket options, or call certain system calls like `connect()`, `send()`, `recv()`, etc. We only use a simple server-side application with default options, passively listening on a socket and receiving packets.

In addition, we note that there is a well-known data overlapping evasion strategy which is missed by Themis. Basically, an attacker can craft two data packets with overlapping sequence numbers and result in ambiguities regarding which of the two copies of the overlapped portion will be accepted. Most of the OSes, such as Linux, favor the last segment upon receiving the overlapped data, while Windows favors the first segment.

Since THEMIS currently analyzes only Linux implementations, it cannot discover this particular discrepancy. Nevertheless, this discrepancy can still be easily incorporated into the ambiguity-aware NIDS.

Extending to Other Operating Systems. Although Linux enjoys a major market share among the server OSes, there are also other OSes such as Windows and FreeBSD. THEMIS uses S2E [37] as the symbolic execution engine to extract the TCP model from an OS. S2E works on the binary level and runs the entire OS in QEMU, and does not require source code. So in principle, we could extend THEMIS to all other OSes, including those that are closed-source (e.g., Windows). However, we will need to label the critical states and merge ranges to scale up symbolic execution. Without access to source code, this process can be more time-consuming.

Extensions to Model Client Behaviors. In this work, we focus on modeling servers' behaviors. Although the TCP stack of the client and the server are the same, we do not explore TCP states exclusively related to the client, e.g., TCP_SYN_SENT. Our motivation stems from the fact that NIDS are typically deployed as safeguards against servers in corporations as opposed to individual clients which could be anywhere in the world. In order to model the clients' behaviors, we need to run a client application and actively initiate connections and send packets. Conceivably, THEMIS can protect a client from being exploited by malicious content sent from a server. However, we leave this possibility to future work.

Ethical Considerations. We acknowledge that improving the robustness of NIDS has an unintended consequence of improving the robustness of censorship firewalls too, as they both need to keep track of TCP connection states and reassemble TCP data packets. Similar to other technologies such as encryption that can be used for both good (e.g., protecting our privacy) and bad purposes (e.g., plotting a terrorist attack), we believe the value in preventing malicious attacks generally outweighs the collateral damage of disrupting censorship circumvention.

4.7 Related Work

Finding Discrepancies between Implementations. Discrepancies between implementations are usually good indicators of implementation bugs. They can also be used to fingerprint implementations or evade detection (as considered here) leveraging semantic gaps [67, 126]. There is work aiming at finding discrepancies between different implementations of the same target; examples include network protocols [25, 34, 126], parsers [88, 67, 30], libraries [120, 112], etc. A common way to find discrepancies is differential testing combined with random input generation or fuzzing. Brubaker et al.[24], generate synthetic X.509 certificates by randomly mutating fields in a real certification, and then feed them to different certificate validation programs in order to find bugs from discrepancies. Jana et al. [67] also employ differential fuzzing but against malware detectors. They discover novel attacks that exploit the discrepancies between parsers of the malware detectors and actual applications, and can evade the detection. This approach treats the target as a black-box and does not require any internal information, and is therefore easy to apply. However, the coverage is usually low because it can only explore the search space near the seed input.

Some other works use static analysis to extract semantic information from binary or source code. Min et al. [78], target the Linux file systems and extract high-level semantic information from the source code; they then do a statistical comparison to discover deviant behaviors. Srivastava et al. [112] conduct flow- and context-sensitive interprocedual static analysis on Java API implementations, and produce context-sensitive security policies for every API entry point, and then compare the policies to find discrepancies. Static analysis can leverage semantic information and is scalable, but also suffers from false positives.

Symbolic execution is also used to extract a more accurate semantic representation from the source code or binary. Brumley et al. [25], extract symbolic formulas by replaying captured network traces against different implementations, and then compare the symbolic formulas with a constraint solver. But due to limited coverage, they suffer from false positives. Similarly, Chau et al [33] feed symbolic X.509 certificates to certificate validation implementations, and extract constraints relating to certificate "accept" and "reject" paths. They then use a constraint solver to find discrepancies. Wang et al. [126] combine symbolic execution with black-box differential testing, and use symbolic execution as a guide to group equivalence inputs by execution paths and therefore, largely reduce the search space. However, all these works only achieve partial coverage and compare indiviual execution paths to discover discrepancies opportunistically. Our approach aims to systematically discover all discrepancies between two implementations, as it relates to NIDS evasion. To achieve this goal, we need to run symbolic execution exhaustively to traverse all feasible execution paths in each of our target implementations, and then calculate the weakest preconditions based on the entire program.

Defenses against NIDS Evasion Attacks. Zhu et al.[138] present a deep learning based solution for detecting and localizing DPI evasion attacks by learning the so-called packet context (i.e., inter-relationships of header fields within and across packets) from benign traffic traces. It then uses the learnt model on unseen network connections to spot anomalies in terms of deviations from the benign context distribution. As discussed in §4.5, compared to Themis, [138] falls behind in terms of both the detection accuracy (0.963 vs. 1.00 in AUC-ROC) and runtime overhead (2162.2 vs. 69400.5 packets processed per second under the same single-core CPU setup). This is because any DL-based defense, unlike Themis, always will generate some incorrect classifications and require relatively heavy computations in their inference phase.

Traffic normalization [61] takes a different approach in defending NIDS against evasion attacks. A normalizer sits in the path and patches up the packets passing through to eliminate potential ambiguities, before they are seen by the NIDS. It relies on a manually curated list of potential ambiguities in basic network protocols such as TCP, UDP, IP, and ICMP. However, unfortunately, it cannot safely remove all possible ambiguities in the absence of detailed knowledge about the various implementations on the endhosts, and could even disrupt the communications since it alters the traffic.

Active Mapping [104] builds a profile for each endhost and actively maintains a profile database. This is apt for a small network with relatively stable members, and not for a large scale network with dynamic members. Usually, it takes time to build profiles, and they need to be updated often. Moreover, the NIDS needs to be agnostic to the details and configurations of software on the endhosts.

Paxson [87] proposes to use bifurcating analysis to explore all different possibilities of packet reassembly. However, without the knowledge of ambiguities, it will lead to exponential growth in state forking overhead in practice.

4.8 Conclusions

In this chapter, we aim to defend against attacks that seek to evade network intrusion detection systems, by exploiting the discrepancies between its TCP implementation and that at a targeted end server. These discrepancies are commonplace, and, thus these threats are very real. We design a novel lightweight system Themis which is extremely effective in defending against such attacks. It contains an offline phase, where it identifies and models discrepancies in TCP implementations across OS versions using symbolic execution. The models are then employed at runtime, and by applying a non-deterministic automaton the proper implementation versions are forked to handle packets correctly and block evasion attempts. Themis is extremely effective and is able to block all known evasion attempts to date with negligible additional overhead on a NIDS. In developing Themis we also discover multiple brand new discrepancies, that are exploitable as it relates to current NIDS.

Chapter 5

Conclusions

My work focuses on understanding the potential threats caused by discrepancies in implementations of stateful network protocols, such as the Transmission Control Protocol (TCP), developing systematic and automated approaches to discover discrepancies, and proposing defenses that mitigate these threats. First, we conduct a comprehensive study of the evadability of the largest censorship/network intrusion detection system on today's Internet, viz., the Great Firewall of China (GFW). Our evasion strategies are based on TCP-layer discrepancies between the target and the servers on the Internet. We treat the target as a blackbox and infer its TCP model by sending probe packets. Our inferred model leads us to the discovery of novel evasion strategies designed based on unique behaviors of the target. We also build an extensible measurement tool INTANG that incorporates all existing and newly discovered evasion strategies. Our results show that our discrepancy-based evasion strategies can achieve extremely high success rates of close to 100%, thus, validating our understanding of TCP-layer discrepancies in the context of NIDS evasion.

Subsequently, we seek an automated approach to boost the efficiency of discovery of TCP-layer discrepancies. We develop SymTCP, an automated tool that employs the symbolic execution technique to analyze the Linux TCP stack and generate candidate insertion and evasion packets that may trigger discrepancies, and then use differential testing to validate those candidates to find real discrepancies between the TCP stacks of NIDSs and of Linux servers. SymTCP is demonstrated to be capable of finding highly effective evasion strategies, against three state-of-the-art NIDS systems, Zeek, Snort, and the GFW, in a short time, significantly outperforming previous works based on manual approaches. Our methodology developed can be easily adapted and extended to other TCP implementations and other network protocols.

Finally, we aim at defending the network intrusion detection systems against discrepancy-based evasion attacks, by incorporating the knowledge of discrepancies into the design of NIDS. We design Themis, a system composed of an offline phase and an online phase. In the offline phase, we employ exhaustive symbolic execution to systematically discover discrepancies between various versions of the Linux TCP stack, and generate a comprehensive list of discrepancies, including previously unknown ones. In the online phase, we propose a novel design of NIDS based on nondeterministic finite automata (NFA), and empower it with the knowledge summarized from discovered discrepancies. Our NIDS is robust and immune to discrepancy-based evasion attacks. We demonstrate that Themis is highly effective in detecting all known evasion attacks to date, while introducing negligible overhead comparing to a traditional DFA-based NIDS.

Bibliography

[1] DNSCrypt. `https://dnscrypt.org/`.

[2] libev. `http://software.schmorp.de/pkg/libev.html`.

[3] Giuseppe Aceto and Antonio Pescapé. Internet censorship detection: A survey. volume 83, pages 381–421, New York, NY, USA, June 2015. Elsevier North-Holland, Inc.

[4] Daniel Anderson. Splinternet behind the great firewall of china. volume 10, pages 40:40–40:49, New York, NY, USA, November 2012. ACM.

[5] Anonymous. Evaluation and problems of intrusion detection system. `http://www.chinagfw.org/2009/09/gfw_21.html`, 2009.

[6] Anonymous. The Collateral Damage of Internet Censorship by DNS Injection. volume 42, pages 21–27, New York, NY, USA, June 2012. ACM.

[7] Anonymous. Towards a Comprehensive Picture of the Great Firewall's DNS Censorship. In *4th USENIX Workshop on Free and Open Communications on the Internet*, FOCI '14, San Diego, CA, August 2014. USENIX Association.

[8] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1690–1701, New York, NY, USA, 2016. ACM.

[9] George Argyros, Ioannis Stais, Aggelos Kiayias, and Angelos D Keromytis. Back in black: towards formal, black box analysis of sanitizers and filters. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 91–109. IEEE, 2016.

[10] Big brother on a budget: How internet surveillance got so cheap. `https://arstechnica.com/information-technology/2012/09/big-brother-meets-big-data-the-next-wave-in-net-surveillance-tech/`.

[11] Nebuad, isps sued over dpi snooping, ad-targeting program. `https://arstechnica.com/tech-policy/2008/11/nebuad-isps-sued-over-dpi-snooping-ad-targeting-program/`.

[12] Pablo Neira Ayuso. Netfilter queue project. `http://www.netfilter.org/projects/libnetfilter_queue/`.

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[14] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Ralf Bendrath. Global technology trends and national regulation: Explaining variation in the governance of deep packet inspection. In *International Studies Annual Convention*, volume 15, 2009.

[16] Ralf Bendrath and Milton Mueller. The end of the net as we know it? deep packet inspection and internet governance. *New Media & Society*, 13(7):1142–1160, 2011.

[17] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. *Commun. ACM*, 60(2):99–107, January 2017.

[18] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave Levin. Detecting and evading censorship-in-depth: A case study of iran's protocol whitelister. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association, August 2020.

[19] Kevin Bock, George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, and Dave Levin. Come as you are: Helping unmodified clients bypass censorship with server-side evasion. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 586–598, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2199–2214, New York, NY, USA, 2019. Association for Computing Machinery.

[21] boofuzz: Network protocol fuzzing for humans. `https://github.com/jtpereyda/boofuzz`.

[22] Amine Boukhtouta, Serguei A. Mokhov, Nour-Eddine Lakhdari, Mourad Debbabi, and Joey Paquet. Network malware classification comparison using dpi and flow packet headers. *Journal of Computer Virology and Hacking Techniques*, 12(2):69–100, May 2016.

[23] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

[24] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129. IEEE, 2014.

[25] David Brumley, Juan Caballero, Zhenkai Liang, and James Newsome. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *16th USENIX Security Symposium (USENIX Security 07)*, Boston, MA, August 2007. USENIX Association.

[26] 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, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[27] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

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

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

[30] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[31] 5 things you need to know about deep packet inspection. `https://www.cavium.com/pdfFiles/CSS-DPI-White-Paper.pdf`.

[32] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. Symcerts: Practical symbolic execution for exposing noncompliance in X.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 503–520. IEEE Computer Society, 2017.

[33] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification. In *NDSS*, 2019.

[34] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1516–1527, New York, NY, USA, 2016. Association for Computing Machinery.

[35] 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, pages 388–400, New York, NY, USA, 2015. ACM.

[36] Tommy Chin, Kaiqi Xiong, and Chengbin Hu. Phishlimiter: A phishing detection and mitigation approach using software-defined networking. *IEEE Access*, 6:42516–42531, 2018.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[38] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. volume 30, pages 2:1–2:49, New York, NY, USA, February 2012. ACM.

[39] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[40] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson. Ignoring the great firewall of china. In *Proceedings of the 6th International Conference on Privacy Enhancing Technologies*, PET '06, pages 20–35, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] Jedidiah R. Crandall, Daniel Zinn, Michael Byrd, Earl Barr, and Rich East. Conceptdoppler: A weather tracker for internet censorship. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 352–365, New York, NY, USA, 2007. ACM.

[42] Exploiting dpi surveillance for advertising will track if you surf for work or fun. `https://www.csoonline.com/article/2227882/`.

[43] tcp: better validation of received ack sequences.

[44] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015. USENIX Association.

[45] Sarang Dharmapurikar and Vern Paxson. Robust tcp stream reassembly in the presence of adversaries. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[46] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[47] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[48] Haixin Duan, Nicholas Weaver, Zongxu Zhao, Meng Hu, Jinjin Liang, Jian Jiang, Kang Li, and Vern Paxson. Hold-on: Protecting against on-path dns poisoning. In *Workshop on Securing and Trusting Internet Names (SATIN)*, 2012.

[49] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the great firewall discovers hidden circumvention servers. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 445–458, New York, NY, USA, 2015. ACM.

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

[51] Graham Finnie. Isp traffic management technologies: The state of the art. *Heavy Reading. Report for the CRTC*, 2009.

[52] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Learning fragments of the tcp network protocol. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, pages 78–93, Cham, 2014. Springer International Publishing.

[53] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 454–471, Cham, 2016. Springer International Publishing.

[54] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *Proceedings of the 6th International COnference*, pages 1–12, 2010.

[55] Christian Fuchs. Implications of deep packet inspection (dpi) internet surveillance for society. PACT, 2012.

[56] Phillipa Gill, Masashi Crete-Nishihata, Jakub Dalek, Sharon Goldberg, Adam Senft, and Greg Wiseman. Characterizing web censorship worldwide: Another look at the opennet initiative data. volume 9, pages 4:1–4:29, New York, NY, USA, January 2015. ACM.

[57] Inc. GitHub. Source code and dataset for clap. `https://github.com/seclab-ucr/CLAP`, 2020.

[58] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[59] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166. Citeseer, 2008.

[60] F. Gont and A. Yourtchenko. On the Implementation of the TCP Urgent Mechanism. Internet Requests for Comments, January 2011.

[61] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 9–9, Berkeley, CA, USA, 2001. USENIX Association.

[62] Trevor Hansen, Peter Schachte, and Harald Søndergaard. *State Joining and Splitting for the Symbolic Execution of Binaries*, page 76–92. Springer-Verlag, Berlin, Heidelberg, 2009.

[63] Andy Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385, RFC Editor, August 1998.

[64] E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 627–638, June 2017.

[65] Transmission control protocol (tcp) parameters: Tcp option kind numbers. `https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1`.

[66] OpenNet Initiative. China — oni country profile. `https://opennet.net/research/profiles/china`, 2012.

[67] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, page 80–94, USA, 2012. IEEE Computer Society.

[68] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. Towards illuminating a censorship monitor's model to facilitate evasion. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*, FOCI '13, Washington, D.C., August 2013. USENIX.

[69] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, page 228–233, New York, NY, USA, 1975. Association for Computing Machinery.

[70] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[71] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 26–37, New York, NY, USA, 2011. ACM.

[72] Andreas Kuehn and Milton Mueller. Profiling the profilers: deep packet inspection and behavioral advertising in europe and the united states. *Available at SSRN 2014181*, 2012.

[73] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 193–204, New York, NY, USA, 2012. Association for Computing Machinery.

[74] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. Lib·erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 128–141, New York, NY, USA, 2017. ACM.

[75] Llvm language reference manual. `https://llvm.org/docs/LangRef.html`.

[76] Graham Lowe, Patrick Winters, and Michael L Marcus. The great dns wall of china. Technical report, December 2007.

[77] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[78] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 361–377, New York, NY, USA, 2015. Association for Computing Machinery.

[79] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 102–115, Berlin, Heidelberg, 2006. Springer-Verlag.

[80] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 699–718, Berkeley, CA, USA, 2019. USENIX Association.

[81] Milton L Mueller and Hadi Asghari. Deep packet inspection and bandwidth management: Battles over bittorrent in canada and the united states. *Telecommunications Policy*, 36(6):462–475, 2012.

[82] Steven J. Murdoch and Stephen Lewis. Embedding covert channels into tcp/ip. In *Proceedings of the 7th International Conference on Information Hiding*, IH'05, pages 247–261, Berlin, Heidelberg, 2005. Springer-Verlag.

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

[84] Daiyuu Nobori and Yasushi Shinjo. Vpn gate: A volunteer-organized public vpn relay system with blocking resistance for bypassing government censorship firewalls. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 229–241, Berkeley, CA, USA, 2014. USENIX Association.

[85] Decryption. `https://www.paloaltonetworks.com/features/decryption`.

[86] Jong Chun Park and Jedidiah R. Crandall. Empirical study of a national-scale distributed intrusion detection system: Backbone-level filtering of html responses in china. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 315–326, Washington, DC, USA, June 2010. IEEE Computer Society.

[87] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, 31(23-24):2435–2463, December 1999.

[88] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[89] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 543–553, New York, NY, USA, 2016. Association for Computing Machinery.

[90] Jon Postel. Transmission control protocol. RFC 793, RFC Editor, September 1981.

[91] The Tor Project. The tor project. `https://www.torproject.org`.

[92] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Envasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, SECURE NETWORKS INC CALGARY ALBERTA, 1998.

[93] Alan Quach, Zhongjie Wang, and Zhiyun Qian. Investigation of the 2016 linux tcp stack vulnerability at scale. *SIGMETRICS Perform. Eval. Rev.*, 45(1):8–8, June 2017.

[94] Anantha Ramaiah, R Stewart, and Mitesh Dalal. Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, RFC Editor, August 2010.

[95] Redis. The redis project. `http://redis.io/`.

[96] Gaganjeet Singh Reen and Christian Rossow. Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic. In *Annual Computer Security Applications Conference*, ACSAC '20, page 332–344, New York, NY, USA, 2020. Association for Computing Machinery.

[97] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 156–, Washington, DC, USA, 2000. IEEE Computer Society.

[98] tcp: accept rst for rcv_nxt - 1 after receiving a fin.

[99] tcp: accept rst if seq matches right edge of right-most sack block.

[100] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 186–196, New York, NY, USA, 2010. ACM.

[101] scholarzhang. West chamber project. `https://code.google.com/p/scholarzhang/`, 2010.

[102] Zain Shamsi, Ankur Nandwani, Derek Leonard, and Dmitri Loguinov. Hershel: Single-packet os fingerprinting. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 195–206, New York, NY, USA, 2014. ACM.

[103] U. Shankar and V. Paxson. Active mapping: resisting nids evasion without altering traffic. In *2003 Symposium on Security and Privacy, 2003.*, pages 44–61, May 2003.

[104] U. Shankar and V. Paxson. Active mapping: resisting nids evasion without altering traffic. In *2003 Symposium on Security and Privacy, 2003.*, pages 44–61, May 2003.

[105] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[106] Daniel Smallwood and Andrew Vance. Intrusion analysis with deep packet inspection: increasing efficiency of packet based investigations. In *2011 International Conference on Cloud and Service Computing*, pages 342–347. IEEE, 2011.

[107] Mathew Smart, G. Robert Malan, and Farnam Jahanian. Defeating tcp/ip stack fingerprinting. In *USENIX Security*, 2000.

[108] Smt-lib the satisfiability modulo theories library.

[109] Snort - network intrusion detection & prevention system. `https://www.snort.org/`.

[110] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 262–271, New York, NY, USA, 2003. ACM.

[111] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Trans. Softw. Eng.*, 40(7):695–709, July 2014.

[112] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. A security policy oracle: Detecting security holes using multiple api implementations. *ACM SIGPLAN Notices*, 46(6):343–354, 2011.

[113] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[114] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 314–327, New York, NY, USA, 2016. ACM.

[115] Symantec data loss prevention. `https://www.symantec.com/products/data-loss-prevention`.

[116] Radwan Tahboub and Yousef Saleh. Data leakage/loss prevention systems (dlp). In *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, pages 1–6. IEEE, 2014.

[117] Michael Carl Tschantz, Sadia Afroz, David Fifield, and Vern Paxson. Sok: Towards grounding censorship circumvention in empiricism. In *2016 IEEE Symposium on Security and Privacy (SP)*, number Section V, pages 914–933, May 2016.

[118] twilde. Knock knock knockin' on bridges' doors. `https://blog.torproject.org/blog/knock-knock-knockin-bridges-doors`, January 2012.

[119] Rohit Tyagi, Tuhin Paul, BS Manoj, and B Thanudas. A novel http botnet traffic detection method. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2015.

[120] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. Discovering discrepancies in numerical libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 488–501, New York, NY, USA, 2020. Association for Computing Machinery.

[121] John-Paul Verkamp and Minaxi Gupta. Inferring mechanics of web censorship around the world. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet*, FOCI '12, Bellevue, WA, 2012. USENIX.

[122] Vex intermediate representation. `https://github.com/angr/vex/blob/dev/pub/libvex_ir.h`.

[123] VPNanswers.com. Bypass the great firewall and hide your openvpn in china. `https://www.vpnanswers.com/bypass-great-firewall-hide-openvpn-in-china-2015/`, 2015.

[124] Mythili Vutukuru, Hari Balakrishnan, and Vern Paxson. Efficient and robust tcp stream normalization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 96–110, Washington, DC, USA, 2008. IEEE Computer Society.

[125] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Your state is not mine: A closer look at evading stateful internet censorship. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 114–127, New York, NY, USA, 2017. ACM.

[126] 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 *Network and Distributed System Security Symposium (NDSS)*, 2020.

[127] Every click you make. `http://www.washingtonpost.com/wp-dyn/content/article/2008/04/03/AR2008040304052.html`.

[128] Nicholas Weaver, Robin Sommer, and Vern Paxson. Detecting forged tcp reset packets. In *NDSS*, 2009.

[129] Deep packet inspection. `https://www.wikiwand.com/en/Deep_packet_inspection`.

[130] Philipp Winter and Stefan Lindskog. How the great firewall of china is blocking tor. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet*, FOCI '12, Bellevue, WA, 2012. USENIX.

[131] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 29–35, New York, NY, USA, 2016. ACM.

[132] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Conference on Security*, SEC '11, pages 30–30, Berkeley, CA, USA, 2011. USENIX Association.

[133] Eva Xiao. Behind the scenes: Here's why your vpn is done in china. `http://technode.com/2016/03/17/behind-scenes-heres-vpn/`, 2016.

[134] Xueyang Xu, Z. Morley Mao, and J. Alex Halderman. Internet censorship in china: Where does the filtering occur? In *Proceedings of the 12th International Conference on Passive and Active Measurement*, PAM '11, pages 133–142, Berlin, Heidelberg, 2011. Springer-Verlag.

[135] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association.

[136] The z3 theorem prover. `https://github.com/Z3Prover/z3`.

[137] Ying Zhang, Zhuoqing Morley Mao, and Ming Zhang. Detecting traffic differentiation in backbone isps with netpolice. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 103–115, New York, NY, USA, 2009. ACM.

[138] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V. Krishnamurthy, Kevin S. Chan, and Ananthram Swami. You do (not) belong here: Detecting dpi evasion attacks with context learning. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 183–197, New York, NY, USA, 2020. Association for Computing Machinery.

Appendix A

List of Implemented Evasion Strategies for Themis Evaluation

We have implemented all the TCP-related evasion strategies presented in previous works [92, 68, 125, 74, 126, 20], after merging redundant strategies. There are in total 34 evasion stategies as listed in Table A.1. We apply those stategies to a HTTP connection, in which the client sends a malicious request with a bad keyword. THEMIS can successfully detect the bad keyword in all the attacks.

Table A.1: List of implemented evasion strategies (strategies in bold are new)

No.	**Strategy**	**Description**
1	Bad checksum data	In ESTABLISHED state, send junk data with bad checksum, and then send the request
2	Bad checksum RST	In ESTABLISHED state, send partial request, then send a RST packet with bad checksum, and then send the remaining request
3	No ACK flag data	In ESTABLISHED state, send junk data without ACK flag, and then send the request
4	No ACK flag FIN	In ESTABLISHED state, send partial request, then send a FIN packet without ACK flag, and then send the remaining request
5	SYN with data	In LISTEN state, send a SYN packet with payload, then send the request
6	Bad ACK number data	In ESTABLISHED state, send junk data with out-of-window ACK number, and then send the request
7	Bad ACK number RST/ACK	In ESTABLISHED state, send partial request, then send a RST/ACK packet with out-of-window ACK number, and then send the remaining request
8	Small data offset header	In ESTABLISHED state, send junk data with TCP data offset <5, and then send the request
9	Large data offset header	In ESTABLISHED state, send junk data with TCP data offset >actual packet size / 4, and then send the request
10	Bad MD5 data	In ESTABLISHED state, send junk data with TCP MD5 option, and then send the request

No.	Strategy	Description
11	Bad MD5 RST	In ESTABLISHED state, send partial request, then send a RST packet with TCP MD5 option, and then send the remaining request
12	Bad TCP timestamp data	In ESTABLISHED state, send junk data with bad TCP timestamp, and then send the request
13	Bad TCP timestamp RST	In ESTABLISHED state, send partial request, then send a RST packet with bad TCP timestamp, and then send the remaining request
14	Bad SEQ number data	In ESTABLISHED state, send junk data with out-of-window SEQ number, and then send the request
15	Bad SEQ number FIN	In ESTABLISHED state, send partial request, then send a FIN packet with out-of-window SEQ number, and then send the remaining request
16	Bad SEQ number RST	In ESTABLISHED state, send partial request, then send a RST packet with out-of-window SEQ number, and then send the remaining request
17	Invalid TCP flags	In ESTABLISHED state, send junk data with flags FRAPUN set, and then send the request
18	Multiple SYNs	In SYN_RECV or ESTABLISHED state, send a SYN packet with out-of-window SEQ num, and then send the request
19	Big gap in data	In ESTABLISHED state, send junk data with SEQ = rcv_nxt + max_gap_size (16384), and then send the request
20	SEQ number before ISN	In ESTABLISHED state, send the request with SEQ <ISN (initial sequence number) but partial-in-window data

No.	**Strategy**	**Description**
21	In-window SYN	In ESTABLISHED state, send partial request, then send a SYN packet with SEQ >rcv_nxt but in window, and then send the remaining request
22	In-window FIN	In ESTABLISHED state, send partial request, then send a FIN packet with SEQ >rcv_nxt but in window, and then send the remaining request
23	In-window RST	In ESTABLISHED state, send partial request, then send a RST packet with SEQ >rcv_nxt but in window, and then send the remaining request
24	Partial in-window RST	In ESTABLISHED state, send partial request, then send a RST packet with SEQ <rcv_nxt but partial data in window, and then send the remaining request
25	Urgent data	In ESTABLISHED state, send the request with urgent pointer and URG flag set, also need to insert one byte urgent data into the payload
26	Time gap	In ESTABLISHED state, send partial request with timestamp, and then send the remaining request with timestamp = last_timestamp + 0x80000000
27	Small segments	In ESTABLISHED state, send the request in small segments (size = 4)
28	TCB Turnaround	In LISTEN state, send a SYN/ACK packet before sending the SYN packet, then establish the connection and send the request
29	Muti-segmentation	In ESTABLISHED state, send the request in segments, 1st segment size 8, 2nd segment size 4, and then send the remaining request

No.	**Strategy**	**Description**
30	Simple TCB Desynchronization	In SYN_RECV state, send a SYN packet with junk payload, and then send the request
31	SYN+FIN	In LISTEN state, send a SYN+FIN packet, then establish the connection with a different ISN and send the request
32	**RST rightmost SACK**	SAckOK option in SYN packet. In ESTABLISHED state, send partial request with a SEQ gap, then send a RST packet with SEQ = SEQ end of last packet, and then send the remaining request
33	**RST after FIN**	In ESTABLISHED state, send a FIN/ACK packet, then send a RST packet with SEQ = SEQ of the FIN packet, and then reuse the 4-tuple to established a new connection and send the request
34	**Data in closing states**	In ESTABLISHED state, send a FIN/ACK packet, then send junk data with SEQ <rcv_nxt but in-window data and ACK <previous ACK, and then reuse the 4-tuple to establish a new connection and send the request