StateDiver: Testing Deep Packet Inspection Systems with State-Discrepancy Guidance

Zhechang Zhang[1,2,3,4]
zhangzhechang@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Bin Yuan[1,2,3,4,6*]
yuanbin@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Kehan Yang[1,2,3,4]
u201812080@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Deqing Zou[1,2,3,4]
deqingzou@hust.edu.cn
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Hai Jin[3,4,5]
hjin@hust.edu.cn
School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

Deep Packet Inspection (DPI) systems are essential for securing modern networks (e.g., blocking or logging abnormal network connections). However, DPI systems are known to be vulnerable in their implementations, which could be exploited for evasion attacks. Due to the critical role DPI systems play, many efforts have been made to detect vulnerabilities in the DPI systems through manual inspection, symbolic execution, and fuzzing, which suffer from either poor scalability, path explosion, or inappropriate feedback. In this paper, based on our observation that a DPI system usually reaches an abnormal internal state before a forbidden packet passes through it, we propose a fuzzing framework that prioritizes inputs/mutations which could trigger the DPI system's abnormal internal states. Further, to avoid deep understanding of the DPI systems under inspection (e.g., to identify the abnormal states), we feed one pair of inputs to multiple DPI systems and check whether the state changes of these DPI systems are consistent — an inconsistent internal state change/transference in one of the DPI systems indicates a new abnormal state is reached in the corresponding DPI system. Naturally, inputs that trigger new abnormal states are preferentially selected for mutations to generate new inputs. Following this idea, we develop StateDiver, the first fuzzing framework that uses the state discrepancy between different DPI systems as feedback to find more bypassing strategies. We make StateDiver publicly available online. With the help of StateDiver, we tested 3 famous open-source DPI systems (Snort, Snort++, and Suricata) and discovered 16 bypass strategies (8 new and 8 previously known). We have reported all the vulnerabilities to the vendors and received one CVE by the time of paper writing. We also compared StateDiver with Geneva, the state-of-the-art fuzzing tool for detecting DPI bugs. Results showed that StateDiver outperformed Geneva at the number and speed of finding vulnerabilities, indicating the ability of StateDiver to detect strategies bypassing DPI systems effectively.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems**; **Network security**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Deep packet inspection, TCP, fuzzing, StateDiver

ACM Reference Format:
Zhechang Zhang[1,2,3,4], Bin Yuan[1,2,3,4,6*], Kehan Yang[1,2,3,4], Deqing Zou[1,2,3,4], and Hai Jin[3,4,5]. 2022. StateDiver: Testing Deep Packet Inspection Systems with State-Discrepancy Guidance. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564625.3564650

1 INTRODUCTION

Deep Packet Inspection (DPI) is a powerful data-processing technique that inspects transport-level or application-level network packets and may take necessary actions such as logging or blocking. For example, if string "bad_url" is in the HTTP block list, DPI system will tear down any HTTP connection that contains "bad_url". This technique has been widely used in modern network architectures for various security purposes, including but not limited to

[1]Hubei Key Laboratory of Distributed System Security, Wuhan, China.
[2]Hubei Engineering Research Center on Big Data Security, Wuhan, China.
[3]National Engineering Research Center for Big Data Technology and System, Wuhan, China.
[4]Services Computing Technology and System Lab, Wuhan, China.
[5]Cluster and Grid Computing Lab, Wuhan, China.
[6]Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen, China.
*Bin Yuan is the Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ACSAC '22, December 5–9, 2022, Austin, TX, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9759-9/22/12…$15.00
https://doi.org/10.1145/3564625.3564650

malware detection [28], phishing-attack detection [32], remote exploits prevention [43], data leakage prevention [45], advertisement injection [50], copyright enforcement [24] and even statistics [37].

Unfortunately, DPI systems contain security vulnerabilities that allow attackers to bypass the detection. For example, due to CVE-2019-18792 [7], Suricata [20], one of the most popular open-source DPI systems, will accept malformed HTTP connections that contain forbidden keywords. The vulnerabilities stem from the fact that DPI systems usually implement customized network protocols for high throughput and wide adoption [17, 47]. Due to the complexity in network protocols, the customization is inevitably different from the standards specified in hundred-page RFCs, which may lead to severe security breaches [5, 6, 8]. It is emerging to detect these vulnerabilities to enhance the strength of DPI systems.

Previous techniques for detecting DPI vulnerabilities fall into three categories: manual construction, symbolic execution, and fuzzing. Many well-known DPI vulnerabilities are identified through manual efforts [38, 46]. This method relies on human experience and cannot effectively handle new DPI systems. Automatic methods based on symbolic execution can systematically explore the protocol state space to identify differences between DPI systems and other applications [47]. However, symbolic execution consumes heavy resources and suffers from the path-explosion problem [30, 33]. This hinders large-scale adoption of related tools to test DPI systems.

In recent years, fuzzing has become a popular technique to verify program functionalities and detect vulnerabilities [1, 10, 12]. The key idea of fuzzing is to utilize proper dynamically collected feedback to identify promising mutations from randomly generated inputs. By accumulating promising mutations, the fuzzer will likely trigger program abnormal behaviors, like crashes or assertion failures. Fuzzing techniques have been widely used to test a large range of applications and reported thousands of bugs [13, 31, 42].

Researchers have built several fuzzing tools to help find vulnerabilities in DPI systems [25, 41, 52]. However, their designs lack proper feedback mechanisms and may diminish the power of fuzzing. For example, DPIFuzz is a generation-based fuzzer that does not use any feedback mechanism [41]. TCPFuzz relies on low-level code coverage as feedback which is too sensitive to identify valuable mutations from less promising ones [52]. Geneva prioritizes inputs that trigger different server-side or DPI-side responses, and may miss intermediate mutations that are necessary for final bypasses [25]. We need a balanced feedback to highlight the most important changes while not early dropping useful mutations.

Our analysis on previously known bypassing strategies reveals that, before the packet containing forbidden keywords finally passes through DPI systems, its predecessors in the mutation chain usually have rendered the DPI system to reach an abnormal internal state. A safe DPI version (i.e., free from the bug or with the bug fixed) will not show such early changes in its internal state. Therefore, any changes that trigger abnormal internal state should be prioritized to increase the chance of reaching the final bypass. The state discrepancy between multiple DPI systems or multiple DPI versions could be a good feedback to help find more bypassing strategies.

In this paper, we propose StateDiver, a fuzzing framework that automatically discovers DPI-bypass strategies using state discrepancies. We make all the code of StateDiver publicly available on GitHub [19]. Similar to previous tools, StateDiver relies on random mutations of existing network packets to generate new inputs. The key difference is that StateDiver performs state instrumentation in DPI systems to track their internal state transference. When one new input renders the DPI system entering an abnormal state, StateDiver will prioritize the input for further mutations.

However, without a deep understanding of the tested DPI system, it is challenging for us to tell which of its internal states are abnormal. Our solution to this problem is to monitor the execution of multiple DPI systems on processing, feed one pair of inputs (instead of just one input) to multiple DPI systems (instead of just one DPI), and check whether the internal state changes of different DPI systems are consistent. If one DPI system shows different internal state transference for the given two inputs while all other DPI systems show the same transference, we will treat the new state reached by the first DPI system as abnormal. For the pair of inputs, one is generated from another through random mutations. In this way, we successfully use the state-discrepancies to guide our further mutations, decide and select the strategies for the following generations.

To understand the strength of state-discrepancy guidance, we applied StateDiver on 3 of the most famous open-source DPI systems, specifically, Snort, Snort++[1], and Suricata. StateDiver successfully reproduced 8 previously known bypass strategies, and detected 8 previously unknown ones. We have reported all our findings to the corresponding vendors. At the time of paper writing, we received one CVE number for all reported issues. We compared StateDiver with Geneva, the state-of-the-art fuzzing tool for detecting DPI bugs. Experiments show that during 24-hour evaluation, StateDiver can detect 2× more unique bugs, and explore 1.2× more unique state transitions than the previous tool. StateDiver is 5× faster in finding the first bypass strategies than Geneva. The result confirms that StateDiver can effectively detect strategies bypassing DPI systems.

We summarize our contributions as follows:

- **New Feedback Mechanism**. We propose a novel feedback method, state discrepancy, to guide fuzzing DPI systems. Our method achieves a good balance on highlighting the most interesting mutations while avoiding dropping necessary ones.
- **End-to-end System**. We implement StateDiver, the first end-to-end fuzzing platform that utilizes the state discrepancy to guide network packet prioritization and mutation.
- **New DPI Vulnerabilities**. We apply StateDiver to 3 most famous open-source DPI systems, and discover 16 bypass strategies (8 new and 8 previously known). We have reported all findings and received one CVE.

2 BACKGROUND

2.1 Deep Packet Inspection

DPI is an advanced method of examining and managing network traffic [23]. Conventional packet filtering (firewall like iptables [11]) can only analyze packets at or below the transport layer. DPI, however, is powerful enough to analyze contents in application layer

[1]Notably, compared to Snort, Snort++ has rewritten its TCP protocol stack (see Section 5.1). Therefore, we treat Snort and Snort++ as two different DPI systems.

or even encrypted data, providing a wider range of detectable protocols and finer identification granularity. DPI system rebuilds data streams from network packets before performing application-layer payloads examination [36]. The data streams are dynamically parsed by the assigned protocol parser after the DPI system recognizes the application layer protocol used in the packets. Then, the DPI system performs pattern recognition on the results output by the protocol parser. DPI system usually uses rule files to define the contents (protocols, fields, and values) to identify and actions to take. Specifically, the contents can be defined as the "www.example.com" in Host field of HTTP protocol, the "PASV" in Command field of FTP protocol, and so on. Actions are used to define how the DPI system processes the packets it receives. For example, Suricata [20] defines the following four actions: 1) *Reject.* Send RST error packets to both sides of the connection. 2) *Drop.* Drop the packet. 3) *Alert.* Generate an alert. 4) *Pass.* Stop further inspection on the connection. The following Suricata rule — *reject tcp any any -> any 80 (msg: "Bad keyword detected"; content: "bad_url"; http_uri; sid: 1;)*, specifies to reject the connection (a.k.a., send RST packets to both sides of the connection) if Suricata identifies that the HTTP packet contains "bad_url" in its URL buffer.

2.2 Customized Protocol Stacks in DPI Systems

In order to rebuild data streams from network packets, DPI systems needs specific protocol implementation to track the state of each connection and perform reconstruction at the right time. DPI vendors usually customize their implementations on the protocol stacks, for the following reasons.

• *Generality Requirement.* A DPI system usually works as a middlebox between the clients and the servers, whose implementations of protocol stacks are different (e.g., Windows and Linux operating systems, different versions of Linux). Hence, it is necessary for DPI vendors to implement their own customized universal protocol stacks to cope with different systems.

• *High Throughput Requirement.* DPI systems need to continuously process high-traffic data. Hence, DPI vendors normally would implement simplified protocol stacks, instead of complete and complex standard ones as specified by RFCs.

However, customizing the protocol stacks may introduce security risks. First, the DPI vendors usually implement their own protocol stacks based on their human interpretation of the RFCs (written in natural language). The incorrect human interpretation would result in vulnerable implementation. Second, when simplifying the protocol stacks, DPI vendors might ignore certain security checks defined in the RFCs intentionally or unintentionally. Therefore, the processing logic for the same packet sequences might be different in the DPI systems and the clients/servers. Consequently, it is possible to evade the DPI system's inspection (e.g., blocking certain packets) with carefully constructed packet sequences [47].

2.3 Fuzzing

Fuzzing is a test technique to verify program functionalities and detect vulnerabilities automatically [1, 10, 12, 21]. It involves inputting massive amounts of random data to the test target trying to make it behave abnormally (e.g., crash or error execution). Fuzzing has found thousands of vulnerabilities in various applications [2, 4, 10]. There are different fuzzing strategies, which can be roughly classified as follows [49]:

Generation-Based and Mutation-Based Fuzzing. Generation-based fuzzing generates inputs from scratch based on grammars or valid corpus. However, mutation-based fuzzing starts with randomly generated inputs, then it mutates existing inputs to get new inputs. Mutation-based fuzzing relies on properly feedback mechanism to identify promising mutations from randomly generated inputs.

Black-Box, Grey-Box, and White-Box Fuzzing. Based on the amount of information observed during execution. Black-box fuzzing does not have any knowledge about the internal states. White-box fuzzing knows all the internal knowledge to explore more state space of target programs. Grey-box fuzzing obtains the knowledge between these two techniques.

3 THREAT MODEL AND MOTIVATION

In this section, we present the threat model we considered to bypass a DPI system and the flaw example that motivates our work.

3.1 Threat Model

As shown in Figure 1, a DPI system is usually deployed between the user/client and the server to monitor all the transferred packets. As we discussed in Section 2.1, the DPI system tracks the TCP states of a session to determine how to process the packets it receives. In specific, the DPI system internally records the TCP states of both the client and the server, and decides to block or forward certain packets under different states based on the configured rules.

Figure 1: Threat Model

In this paper, we assume the DPI system is configured with a rule to block packets containing certain sensitive string (a.k.a. the keyword) in the application layer (e.g., HTTP, FTP). That is, once a DPI system receives a packet containing the keyword, it would drop the packet immediately. Such a configuration is usually used to block access to sensitive (probably malicious) URLs. We consider the adversary as a malicious user or an attacker that attempts to bypass the restriction of the DPI system. The adversary would send arbitrary packets to the DPI system to disrupt the DPI system's internal state to render the DPI system entering an abnormal state under which a packet containing the forbidden keyword would be forwarded to the server by the DPI system (instead of blocking the packet).

3.2 The Motivating Example

Figure 2 illustrates the RST_Bad_Timestamp bypass strategy that motivates us to systematically study the relationship between the DPI system's internal state transference and the bypass of the blocking rule. Specifically, a malicious user can leverage this strategy to

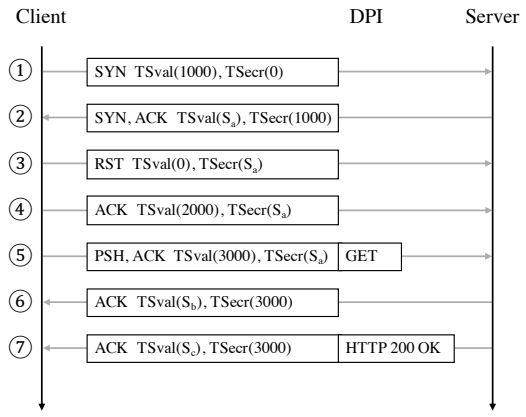

Figure 2: Packet Sequences of RST_Bad_Timestamp
S_a, S_b and S_c are local timestamp in the server-side when the server sends the packet. 1000, 0, 2000, 3000 are values that we artificially constructed to bypass the DPI system.

construct packets to access the network services that are meant to be forbidden by Snort.

The Different Actions of Snort and Suricata under the Same Strategy. Snort and Suricata are two of the most popular open-source DPI systems. We applied the RST_Bad_Timestamp strategy to both of them. As shown in Figure 2, the strategy inserts a RST packet with a corrupted TCP timestamp option (Packet ③) during the 3-way handshake right after receiving SYN-ACK packet from the server. Result shows that Snort is bypassed, while Suricata can successfully detect and drop the packet containing the forbidden keyword.

Root Cause Analyses. To figure out the reason why Snort and Suricata act differently under the same RST_Bad_Timestamp strategy, we manually investigated the TCP implementations in Snort, Suricata, and the server (e.g., the Linux kernel's TCP stack), as well as the TCP RFC documents. As documented in RFC1323 [14] and RFC7323 [15], TCP timestamp option contains two 32-bit timestamp field: TS Value (TSval) represents the local timestamp of the sender when sending the packet, and TS Echo Reply (TSecr) represents the value of TSval in the sender's last received packet. If we craft packets according to the RST_Bad_Timestamp strategy, that is, sending a RST packet with a smaller TSval value (0 in Figure 2), the server would treat the packet as invalid as the TSval is smaller than 1000 (Packet ①) and stays at the state to wait for the ACK packet of the 3-way handshake. Similarly, Suricata, which is implemented with correct timestamp verification, would ignore the above RST packet like the server does. Hence, Suricata cannot be bypassed. However, Snort is implemented to assume the server would close the connection after receiving the RST packet. Therefore, Snort tears down the TCB (TCP control block) of the connection, and stops further detection on the connection, which opens a door for bypassing attack — the attacker sends a packet containing the forbidden keyword after Snort tears down the TCB and successfully reaches/accesses the forbidden server/service.

The Different State Transference under the Same Strategy. Observing the different actions in the two DPI systems, we, then, take a closer look at the internal state transference of the two DPI systems when each packet goes through. As Table 1 shows, when the first two packets of the 3-way handshake (Packet ① and Packet ②) go through, the two DPIs have the same state transference. However, when the RST packet with corrupted timestamp goes through (Packet ③), Snort changes Sn_c from SYN_SENT to CLOSED, while Suricata remains unchanged. By the time the ACK of 3-way handshake arrives (Packet ④), Snort is in unestablished state (Sn_c is CLOSED and Sn_s is SYN_RCVD) while Suricata reaches established state.

Table 1: State Transference of Snort and Suricata under the RST_Bad_Timestamp Strategy

No.	Packet	Direction	Snort	Suricata
①	SYN TSval(1000) TSecr(0)	To Server	Sn_c: SYN_SENT Sn_s: LISTEN	Su_{st}: SYN_SENT
②	SYN, ACK TSval(S_a) TSecr(1000)	To Client	Sn_c: SYN_SENT Sn_s: LISTEN⇒SYN_RCVD	Su_{st}: SYN_SENT⇒SYN_RCVD
③	RST TSval(0) TSecr(S_a)	To Server	Sn_c: SYN_SENT⇒CLOSED Sn_s: SYN_RCVD	
④	ACK TSval(2000) TSecr(S_a)	To Server		Su_{st}: SYN_RCVD⇒ESTABLISHED
⑤	GET TSval(3000) TSecr(S_a)	To Server		
⑥	ACK TSval(S_b) TSecr(3000)	To Client		
⑦	HTTP 200 OK TSval(S_c) TSecr(3000)	To Client		

Snort uses two variables to represent endhosts' (Client and Server) states. For convenience, we call them Sn_c and Sn_s, respectively. Suricata only uses one to represent state, we call it Su_{st}.
An empty cell indicates no change in the values of the variables.

To summarize, Packet ③ causes Snort to reach an abnormal internal state, which finally leads to the evasion. However, a safe DPI (e.g., Suricata) does not show such changes in its internal state. Therefore, if we could identify and prioritize the packets that trigger the DPI's internal state transfer to abnormal internal state, we could increase the chances of reaching the final bypass.

The question now is, without requiring a deep understanding of the tested DPI system (e.g., to identify the DPI system's internal abnormal states) and the protocol details (e.g., to determine the correct protocol state transference), how can we effectively generate packets that could drive the DPI system's internal state transfer to abnormal state?

Our analyses on the RST_Bad_Timestamp strategy (e.g., inconsistent state transference in different DPI systems under the same strategy) inspired us to include multiple DPI systems for vulnerability discovery. That is, using the differential testing technique to help us overcome such challenges.

Specifically, we can feed one pair of inputs (instead of just one input) to multiple DPI systems (instead of just one DPI system), and check whether the internal state changes of different DPI systems are consistent. If one DPI system shows different state transference for the given two inputs while all other DPI systems show the same transference, we could treat the new state reached by the first DPI system as abnormal, without the need of further investigation on whether the state is benign or abnormal. For the pair of inputs, one is generated by the other through random mutation. In specific, we leverage the state-discrepancy to guide our further mutations — prioritizing the aforementioned inputs/strategies that trigger

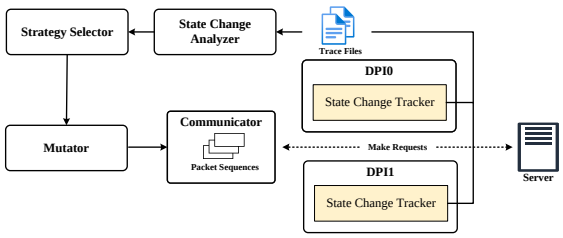

Figure 3: The Architecture of StateDiver

inconsistent state transference for further mutations to generate the following generations.

4 SYSTEM DESIGN AND IMPLEMENTATION

4.1 Overview

Figure 3 shows the architecture of StateDiver, which contains the following modules:

- **State Change Tracker.** We perform state instrumentation in DPI systems. It tracks the state transition in DPI systems and generates trace files. Details are defined in Section 4.2.
- **Mutator.** This module mutates the strategy we select from the *Strategy Selector*. Mutations combine both packet-level mutations and sequence-level mutations. Details are defined in Section 4.3.
- **Communicator.** This module tries to communicate with the server under the surveillance of DPI systems. It tries to make a HTTP request to the server containing keyword in DPI block list. If *Communicator* receives reset packet, that could mean the DPI system recognizes the forbidden keyword and tears down the connection (by sending RST reset packets to each endhost), or it could mean that the server cannot process the packet sequences and establish the TCP connection (i.e. the server sends the RST reset packet to the client representing the failure of making an established TCP connection). We don't make a special distinction between these two possibilities. If *Communicator* receives HTTP Status 200 (OK) status code, we consider it as a successful bypass strategy.
- **State Change Analyzer.** This module receives trace files from DPI systems and evaluates strategies (i.e., the mutations). In the process of mutation, it leverages trace files from previous and current generation, using state-discrepancies as standard to judge whether the mutation is positive, and guide our further mutations. Details are defined in Section 4.4.
- **Strategy Selector.** We implement *Strategy Selector* following the ideas of the queue schedule algorithm in AFL [1], one of the most popular and successful grey-box fuzzer.

Network Topology. Our development involves the participation of two DPI systems, for the convenience of description, we name them DPI0 and DPI1. We set DPI0 inline IPS (Intrusion Prevention System) mode, which is directly in the traffic path and has the ability to manipulate the packet (e.g. drop the packet or send RST packets and reset the connection). We set DPI1 IDS (Intrusion Detection System) mode, which will receive a copy of the packet sequences going through DPI0. But it cannot modify the packet, just raise the alert. Both DPI systems are able to fully see the travel-through packets and perform application-level detection based on respective TCP implementations. They are given the same keyword to detect. The only difference is their action as detecting the forbidden behavior.

Once DPI0 detects the threat, the connection will be terminated immediately. It first intuitively tells us whether the strategy has caused a bypass attack in DPI0. We don't need to check DPI system's log to find out whether it is alerted, which improves the test speed. Secondly, it freezes the state transitions from the initialization of the connection until the connection is blocked. We can take the trace files to perform discrepancies-guided fuzzing.

Differential testing requires both systems to receive the same input. This is inherently difficult to implement in our live-packets network scenario, because if a request is actually sent multiple times, it cannot be guaranteed to be exactly the same due to the disturbance and uncertainty of the network connection. The elements of TCP, such as sequence number, checksum etc., are random-generated or related to each part of the packet. Previous works make a lot of effort on eliminating the non-determinism factors in different protocols [41, 47]. In our network topology, all DPI systems capture the same sequence of travel-through packets. It is unnecessary making the "same" request again, which improves system efficiency.

Workflow of StateDiver. Our system starts with *Strategy Selector*. At the beginning, it generates some strategies randomly, adds to its seed corpus and performs *dry run*, where mutation is not involved, just crafting packets and sending them to the server, in order to get trace files of each prime strategy. After that, the main loop starts, with one strategy from seed corpus starting to mutate and generating its offspring-strategies in *Mutator*. Each of the offspring-strategies forms packet sequences in *Communicator*. These packet sequences are sent to the server under the surveillance of DPI systems. When each tested packet sequences are finished sending, trace files containing DPIs inner state transitions will be generated from *State Change Tracker* module and will be gathered in *State Change Analyzer*. *State Change Analyzer* gives an evaluation of the strategy according to the state transitions of the parent strategy as well as the current strategy in different DPI systems. The result mainly determines the priority and the frequency of the next mutation, and whether the current strategy should be retained. It then adds the current strategy to the seed corpus if necessary. After completing the evaluation of all offspring-strategies, *Strategy Selector* will select the next promising strategy, and repeat the previous actions.

4.2 State Instrumentation

In AFL [1], instrumentation is performed in compilation time, and is injected into compiled programs to capture branch (edge) coverage as well as coarse branch-taken hit counts. Similar to this, State instrumentation aims at getting DPI's inner state transitions, and is performed in preprocessing stage. To accomplish this, we leverage debugging information in open-source DPI systems. The debug option is designed for developers in open-source DPI systems and is used to debug different modules of DPI, which precisely reflects changes of variables, including variables representing the states of endhosts. The similar setting exists in all the open-source DPI systems we investigate. To get more subtle state changes, we manually

add more codes in TCP state processing functions, especially in packet accept point, packet drop point, and function return point, where state changes are most likely to occur. This modification only involves light manual effort and will be discussed in more detail in Section 6. Finally, we compile the DPI system with specific parameters set. Our module will extract the state changes and output a trace file containing them.

4.3 Mutations

We refer to Geneva [25] and use its mutation methods for packets. Here we briefly introduce its composition. Strategies in Geneva comprise a set of (Trigger, Action Tree) pairs. It is a description that tells our system how it should manipulate network traffic. Packets that match a given Trigger are modified using the corresponding sequence of actions in an Action Tree, and then sent on wire.

Triggers. Trigger describes which packets the action tree should run on. The syntax of Triggers is: [PROTOCOL:FIELD:VALUE]. For example, [TCP:flags:PA] represents that the follow-up Action Tree will manipulate the packet where the TCP filed flags are set to PSH-ACK. Triggers perform exact match, so packets with ACK flag will not match this trigger, only packets with both ACK flag and PSH flag.

Actions. Action Tree specifies the manipulation that happens to the packets which fire the Trigger. There are four mutate actions and their respective syntaxes, which are the basic elements to make up the Action Tree. Different mutate actions can be nested within each other and form the action sequence. We use A_1, A_2 to represent different action sequences:

- ***duplicate*(**A_1,A_2**)**: takes one packet and returns two copies of the packet. Action sequence A_1 will be applied to the first packet and A_2 will be applied to the second one.
- ***drop***: takes one packet and drops the packet.
- ***tamper*{protocol:field:mode}(**A_1**)**: takes one packet and returns the modified packet. *tamper* has two modes: *replace* and *corrupt*. *replace* sets the given field of the packet to a specified value, while *corrupt* generates a random value to fill the given field. Action sequence A_1 can be applied to the packet.
- ***fragment*{protocol:offset:order}(**A_1,A_2**)**: takes one packet and returns two fragments or two segments (depending on the given protocol is TCP or IP). *offset* represents where to split the packet and *order* represents whether the two data packets after cutting are in order or reverse order. Action sequence A_1 will be applied to the first packet and A_2 will be applied to the second one.

Example. Here we use the following example to illustrate Geneva's syntax:

Strategy 1 RST_Bad_MD5

[TCP:flags:A]-duplicate(,tamper{TCP:options-md5header:corrupt}(tamper{TCP:flags:replace:R},))-| \/

This example strategy duplicates the outgoing ACK packets, sends the first copy of the packet unaltered, modifies the duplicate one's TCP option MD5 field with a random number and replaces its TCP flag to RST before sending it.

4.4 State Discrepancy Guidance

The algorithm is shown in Algorithm 1. As the previous subsection 4.1 described, *State Change Analyzer* performs differential analysis with four different trace files, that is, two trace files generated by the current strategy in two DPIs (*currDPI0Stat* and *currDPI1Stat*), and two trace files generated by the parent of the current strategy (*parDPI0Stat* and *parDPI1Stat*). We use the complete packet sequences caused by the strategy as the smallest unit of comparison. The algorithm consists of two main parts. First, we perform *CmpDPI* in the same DPI but in different generation strategies. We use the result to confirm whether the newly mutated strategy has caused new state transference in the same DPI compared to the old one, and the result is stored in bool value *DPI0StatAlt* and *DPI1StatAlt*. For example, if the parent strategy and the current strategy cause the same state transference in DPI0, then the *DPI0StatAlt* is set as *False*. That means DPI1 has the same state transference before and after mutation. Then, we evaluate the current strategy based on the results of the first step (line 3-9). If one of their values is *True* and the other is *False*. It reveals the fact that, as the mutation happens from parent strategy to current strategy, in one DPI system the state transference stays the same, but in the other DPI it changes. This may happen due to different processing logic and judging criteria in different DPI systems' TCP implementations, which is exactly what we hope to happen. We prioritize the current strategy for future generations. If in the second branch (line 5-6), it reveals that our mutation is making progress and DPI systems reach further different stages. We consider this kind of mutation is slightly weaker than the top-rated mutation, but still a good mutation. In the last scenario (line 7-8), we consider the mutation is moderate, since it doesn't arouse changes in DPI systems. At this point, *State Change Analyzer* completes the evaluation of the current strategy. If the strategy gets best score, the system will take actions to make it mutate earlier or mutate more times in the next round.

Moderate strategy needs to be analyzed to determine whether to add it to the seed corpus or discard it directly. For example, we will discard the strategy if the mutation just changes the action *fragment* offset value from one-third to one-half. That means purely numerical changes in some fields cannot cause the discrepancy of state transitions.

A slight challenge arises from the fact that what if the mutation proceeds backward in the original direction. We handle it by using hash values of joint state transitions between different generations, which prevents past strategies from being preserved as new strategies.

5 RESULTS AND EVALUATION

Our evaluation tries to answer the following questions:

- Can StateDiver generally apply to different real-world DPI systems and discover bypasses? (Section 5.2)
- Can state-discrepancy guidance improve fuzzing effectiveness? (Section 5.3)
- How does StateDiver perform compared with the previous state-of-the-art evasion works? (Section 5.4)

Algorithm 1 Differential Analysis

```
DPI0StatAlt ← CmpDPI(parDPI0Stat, currDPI0Stat)
DPI1StatAlt ← CmpDPI(parDPI1Stat, currDPI1Stat)
if DPI0StatAlt is True xor DPI1StatAlt is True then
    evaluation ← best_score
else if DPI0StatAlt is True and DPI1StatAlt is True then
    evaluation ← good_score
else if DPI0StatAlt is False and DPI1StatAlt is False then
    evaluation ← moderate_score
end if
return evaluation
```

5.1 Experiment Setup

Our evaluation of StateDiver runs on a machine with 8 cores Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz , and 32GB memory. We use VMware Workstation Pro 16 and employ four virtual machines. The settings details are in Table 9. The OS in virtual machines is Ubuntu 18.04.5 LTS 64-bit, and the Linux kernel version is 5.4.0-81-generic. We experiment locally to reduce the effects of network latency and ensure no machines will be exposed to a real attack.

We applied our system to 3 famous open-source DPI systems, Snort [18], Snort++ [17], and Suricata [20]. As the website [17] illustrated, Snort++ has rewritten TCP handling and has a fully stateful HTTP inspector. It also has changed the programming language from C to C++. So we treated Snort and Snort++ as two different DPI systems.

We downloaded the latest version of Snort (2.9.19), Snort++ (3.1.31), and Suricata (6.0.3) at the time of writing. We first performed state instrumentation. It took us less than two hours to do the state instrumentation in each DPI system, and should take less efforts for someone with experience. We selected DPIs, configured our network topology, and ran the tests. StateDiver tries to make HTTP request with bad keyword "bad_url". DPIs enable rules to detect "bad_url" keyword in HTTP layer. We ran each test for 24h and repeated this process five times.

5.2 Identified Bypasses by StateDiver

As shown in Table 4, StateDiver has successfully identified 16 bypass strategies among 3 DPI systems, including 10 from Snort, 12 from Snort++, and 1 from Suricata. At the time of paper writing, 1 CVE is assigned. In the following case studies, we discuss some of the representative bugs to understand how StateDiver can find these bypasses and how these bypasses work in DPI systems.

Case Study 1: SYN_Bad_MD5. StateDiver identified this bypass strategy in Snort, shown in Strategy 2. It has two Action Trees, both triggered on packets with the ACK flag. The first one duplicates the packets and sends the original and its copy. The second Action Tree first modifies TCP MD5 option field with a random md5 value, then it changes TCP flags from ACK to SYN and sends it on the wire. This strategy can elude Snort successfully. Our deeper analysis of this strategy reveals the reasons for its success. Snort will stop further detection when it gets SYN packet on established stream and consider all the SYN packet on established stream will cause RESET on the other endhost, which is not true in all situations. The server does not accept the SYN packet once the connection is established, however, it performs an MD5 check and ignores packets with invalid MD5 value, while Snort doesn't. This strategy manipulates the last packet of 3-way handshake with the ACK flag set. Sending the original and its copy, which completes 3-way handshake. Then it sends the packet with SYN flag and random TCP MD5 value, which Snort accepts but the server ignores. The connection is still intact but Snort will not perform further detection, which leads to the success evasion.

Strategy 2 SYN_Bad_MD5

[TCP:flags:A]-duplicate-| [TCP:flags:A]-tamper{ TCP:options-md5header:corrupt }(tamper{TCP:flags:replace:S},)-| \/

Now we try to illustrate why our state-discrepancy guidance is effective in discovering this bypass. During the mutation process, we record the mutation sequence of the same strategy and trace files for analysis. After refinement, the strategy and trace files of each strategy are in Table 2. We use A, A', and A" to represent different strategies in the mutation chain. The process of mutation over time is from A through A' to A" (i.e., A generates A', and A' generates A"). As defined in Table 2, we use Sn_c and Sn_s to represent Snort state variables, use Su_{st} to represent Suricata state variable. For example, row 1 in Table 2 means, that when strategy A is being tested, the state transference chain in Snort is, Sn_c: SYN_SENT to CLOSED. Sn_s: LISTEN to SYN_RCVD. The state transference chain in Suricata is SYN_SENT to SYN_RCVD. When strategy A generates strategy A' through mutation, it mutates the strategy's second Trigger from SYN flag to ACK flag. Different state transference occurs at both DPIs when compared with strategy A (We marked in Table 2 with a different typeface). Our system then saves strategy A' for further mutation, since it causes different state transference in both DPIs. We think it is a positive sign to go deeper into the internal state of the DPI system and fully explore it. When strategy A' generates strategy A", the strategy's Action Tree is mutated from corrupting TCP timestamp option to corrupting TCP MD5 options, it leads Sn_c to reach CLOSED after ESTABLISHED. At this point Suricata's state transference Su_{st} is not changed at all, which represents Snort has different behavior than Suricata when facing the same sequences of packets generated by strategy A", and finally leads to evasion. Strategy A" will also be saved for future mutation. Our solution contributes to the mutation process by saving strategies like A' and A". Looking forward to finding strategies causing different state transference.

Case Study 2: FIN_With_Data. StateDiver identified this strategy in Snort, described in Strategy 3. It duplicates the PSH-ACK packet, which contains the HTTP GET request. The strategy changes the first packet's TCP flags from PSH-ACK to FIN, fragments it into two TCP segments, remains the second packet unaltered, then sends these three packets on wire.

Strategy 3 FIN_With_Data

[TCP:flags:PA]-duplicate(tamper{ TCP:flags:replace:F}(fragment{tcp:-1:True},),)-| \/

As RFC793 [16] mentions, once a TCP connection is established, the ACK flag is always sent. The server will ignore all packets with

Table 2: Mutation Process and Trace Files of SYN_Bad_MD5

Strategy	Strategy Code	Snort Trace	Suricata Trace
A	[TCP:flags:A]-duplicate-| [TCP:flags:S]-tamper{TCP:options-timestamp:corrupt} (tamper{TCP:flags:replace:S},)-| \/	Sn_c: SYN_SENT⇒CLOSED Sn_s: LISTEN⇒SYN_RCVD	Su_{st}: SYN_SENT⇒SYN_RCVD
A'	[TCP:flags:A]-duplicate-| [TCP:flags:**A**]-tamper{TCP:options-timestamp:corrupt} (tamper{TCP:flags:replace:S},)-| \/	Sn_c: SYN_SENT⇒**ESTABLISHED** Sn_s: LISTEN⇒SYN_RCVD⇒**ESTABLISHED**	Su_{st}: SYN_SENT⇒SYN_RCVD⇒**ESTABLISHED**
A"	[TCP:flags:A]-duplicate-| [TCP:flags:A]-tamper{TCP:**options-md5header**:corrupt} (tamper{TCP:flags:replace:S},)-| \/	Sn_c: SYN_SENT⇒ESTABLISHED⇒**CLOSED** Sn_s: LISTEN⇒SYN_RCVD⇒ESTABLISHED	Su_{st}: SYN_SENT⇒SYN_RCVD⇒ESTABLISHED

FIN flag but no ACK flag set. Snort, however, will change the inner TCP state once it sees the FIN flag with or without the ACK flag. This leads to the following bypass actions shown in Figure 4. After the 3-way handshake (Packets ①-③), the client sends a FIN packet containing half of the HTTP request in Packet ①, which turns the DPI inner state from ESTABLISHED to FIN_WAIT_1, but the application-layer payload in Packet ① is still added to Segment List. Segment List is designed for reconstructing application-layer payload to perform checks (e.g., HTTP inspection) if TCP segmentation occurs. Data that arrives first will enter Segment List and wait for subsequent data's arrival. Packet ⑤ doesn't contribute because DPI doesn't accept the second FIN packet and discards it. Packet ⑥ is a PSH-ACK packet containing exactly the same HTTP payload as Packet ④. DPI treats it as overlapping with data in Packet ④ and does not add it to the Segment List. Packet ⑦ means the server confirms the arrival of Packet ⑥. As RFC793 [16] describes, when the state in FIN_WAIT_1 receives a packet with the ACK flag, it will change itself to FIN_WAIT_2, representing the client is done talking. DPI follows RFC793's definition, and will not process any further data carried by the following packets or add data to the Segment List, believing that the server will not process them either. Packet ⑧, carries with the rest of the data, can successfully evade DPI. Since the server ignores packet ④ and packet ⑤, its process sequence is ①-③-④(discard)-⑤(discard)-⑥-⑧. The connection is still intact and the server can successfully answer the client's GET request.

Previous work [47] has disclosed a similar vulnerability in older versions of Snort. Our research finds that, for evasion attacks related to FIN packet in ESTABLISHED state, the conditions are relatively harsh. In this strategy, FIN packet must carry certain data to prevent DPI from entering the following situations: 1. Mark the connection as CLOSED and drop the rest of the attack packets. 2. Still performing HTTP level detection when the packet goes through DPI right after FIN packet.

We perform the same mutation process analysis as Strategy 2, shown in Table 3. As the mutation proceeds. The strategy's Action Tree is changed by adding a duplicate action. This action causes a new state (FIN_WAIT_1) to be added to the Sn_c. We highlight B' for the reason that it raises state discrepancies between Snort and Suricata, since Snort states migration changes but Suricata remains the same. B' reaches a stage where this bypass strategy is nearly a success. As we can see from the root cause analysis, this attack is very subtle to perform and not only depends on turning Sn_c to FIN_WAIT_1. However, our state-discrepancy guidance captures this tiny success factor and prioritizes the strategy for further mutations. Finally finding this evasion strategy in various cases.

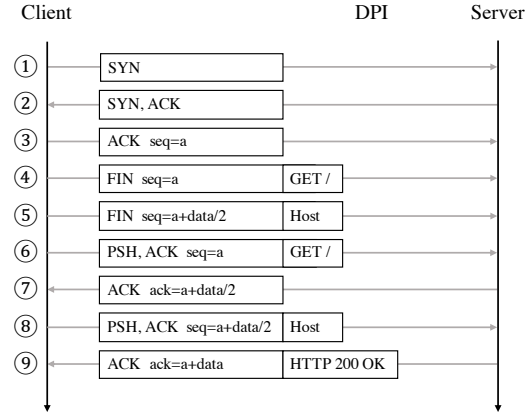

Figure 4: Packet Sequences of FIN_With_Data

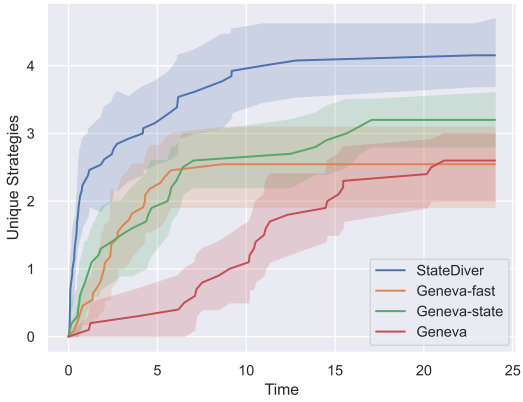

Figure 5: Unique Bypasses Founded by Evaluated Fuzzers for 24h in Snort

5.3 Contributions of State Discrepancies

To understand the contributions of state-discrepancy guidance in fuzzing, we perform unit tests by comparing StateDiver, Geneva, Geneva-*fast*, and Geneva-*state*. Geneva [25] is designed by Bock et al. to perform fuzzing on real-network DPI leveraging DPI-side or server-side responses. Since we test locally, we modify the codes to make it run faster and suitable for local tests, we name it Geneva-*fast*. Geneva-*state* is Geneva with responses guidance removed

Table 3: Mutation Process and Trace Files of FIN_With_Data

Strategy	Strategy Code	Snort Trace	Suricata Trace
B	[TCP:flags:PA]-tamper{TCP:flags:replace:F}-| ∨	Sn_c: SYN_SENT⇒ESTABLISHED Sn_s: LISTEN⇒SYN_RCVD⇒ESTABLISHED	Su_{st}: SYN_SENT⇒SYN_RCVD⇒ESTABLISHED
B'	[TCP:flags:PA]-**duplicate**(tamper{TCP:flags:replace:F}**,)**-| ∨	Sn_c: SYN_SENT⇒ESTABLISHED⇒**FIN_WAIT_1** Sn_s: LISTEN⇒SYN_RCVD⇒ESTABLISHED	Su_{st}: SYN_SENT⇒SYN_RCVD⇒ESTABLISHED
B"	[TCP:flags:PA]-duplicate(tamper{TCP:flags:replace:F} **(fragment{tcp:-1:True},)**,)-| ∨	Sn_c: SYN_SENT⇒ESTABLISHED⇒FIN_WAIT_1⇒**FIN_WAIT_2** Sn_s: LISTEN⇒SYN_RCVD⇒ESTABLISHED	Su_{st}: SYN_SENT⇒SYN_RCVD⇒ESTABLISHED

Table 4: Bypass Identified by StateDiver

Strategy Name	Strategy Code	Illustration	Affected DPI		
			Snort	Snort++	Suricata
△ RST_Bad_Timestamp	[TCP:flags:A]-duplicate(tamper{TCP:options-timestamp:corrupt} (tamper{TCP:flags:replace:R},),)-| ∨	Send RST with invalid TCP timestamp option	✓	✓	✓
△ RST/ACK_Bad_Timestamp	[TCP:flags:A]-duplicate(tamper{TCP:options-timestamp:corrupt} (tamper{TCP:flags:replace:RA},),)-| ∨	Send RST/ACK with invalid TCP timestamp option	✓	✓	✓
△ RST_Bad_MD5	[TCP:flags:A]-duplicate(tamper{TCP:options-md5header:corrupt} (tamper{TCP:flags:replace:R},),)-| ∨	Send RST with invalid TCP MD5 option	✓	✓	
△ RST/ACK_Bad_MD5	[TCP:flags:A]-duplicate(tamper{TCP:options-md5header:corrupt} (tamper{TCP:flags:replace:RA},),)-| ∨	Send RST/ACK with invalid TCP MD5 option	✓	✓	
△ Timestamp_Gap	[TCP:flags:PA:1]-fragment{tcp:-1:True} (,tamper{TCP:options-timestamp:add:2147483648})-| ∨	Send partial request with TCP timestamp option, then send the remaining request with TCP timestamp = last_timestamp + long gap (2147483648)	✓		✓
△ RST/ACK_Bad_ACK_Number	[TCP:flags:A]-duplicate(tamper{TCP:flags:replace:RA} (tamper{TCP:ack:corrupt},),)-| ∨	Send RST with corrupted ACK number in ESTABLISHED state	✓	✓	✓
△ Multiple_SYNs	[TCP:flags:A]-tamper{TCP:flags:replace:S}(tamper{TCP:seq:corrupt},)-| ∨	Send another SYN with corrupted SEQ number in 3-way handshake		✓	
△ TCB_Turnaround	[TCP:flags:S]-fragment{tcp:-1:False}(tamper{TCP:flags:replace:SA},)-| ∨	Send SYN/ACK before sending the SYN packet		✓	
★ FIN_With_Data	[TCP:flags:PA:1]-duplicate(tamper{TCP:flags:replace:F} (fragment{tcp:-1:True},),)-| ∨	Send FIN with junk data in ESTABLISHED state, then send the request in TCP segments	✓		
★ SYN_Bad_MD5	[TCP:flags:A]-duplicate-| [TCP:flags:A]-tamper {TCP:options-md5header:corrupt}(tamper{TCP:flags:replace:S},)-| ∨	Send SYN with invalid TCP MD5 option in ESTABLISHED state, then send the request	✓	✓	
★ SYN_Fragment	[TCP:flags:PA]-duplicate(fragment{tcp:-1:True}(,tamper{TCP:flags:replace:S}),)-| ∨	Send junk data in ESTABLISHED state, then send SYN with junk data, finally send the request	✓		
★ Fragment_And_Segment	[TCP:flags:PA]-fragment{tcp:7:True}(fragment{ip:-1:True} (,duplicate),fragment{ip:-1:True}(duplicate,))-| ∨	Combination of multiple IP fragmentations and TCP segmentations	✓		
★ FIN_Bad_ACK_Number	[TCP:flags:A]-tamper{TCP:ack:corrupt}(duplicate(tamper{TCP:flags:replace:F},),)-| ∨	In SYN_RECV state, send FIN and ACK with same corrupted ACK number successively, then send the request		✓	
★ ACK_Bad_ACK_Number_And_ Data_With_ Smaller_Timestamp	[TCP:flags:A]-fragment{tcp:-1:False}(,tamper{TCP:ack:corrupt} (tamper{TCP:options-timestamp:add:-10},))-| ∨	Send ACK with corrupted ACK number in ESTABLISHED state, then send request with TCP timestamp = last_timestamp - gap (10)		✓	
★ ACK_Bad_MD5_And_ Data_With_Smaller_ Timestamp	[TCP:flags:A]-fragment{tcp:-1:True}-| [TCP:flags:A]-tamper{TCP:options-md5header:corrupt} (tamper{TCP:options-timestamp:add:-10},)-| ∨	Send ACK with invalid TCP MD5 option in ESTABLISHED state, then send request with TCP timestamp = last_timestamp - gap (10)		✓	
★ PSH_Before_SYN	[TCP:flags:S]-duplicate(tamper{TCP:flags:replace:P},)-| ∨	Send PSH without data before 3-way handshake		✓	

△ means previous known strategies, and ★ means new strategies.

and add our state-discrepancy guidance. We compare them in three different metrics: the number of unique bypasses, the speed of bypass discoveries, and state transference collected through trace files. We evaluate the number of unique bypasses as it can reflect bypass finding capabilities. The speed can inform us of the efficiency of fuzzing. For state transference collected through trace files, we consider it can help us explain why our solution works better.

Unique Bypasses. We manually count the unique bypasses of each run in 24h, and show the result in Figure 5, Figure 6, and Figure 7. To save space, we put Figure 6 and Figure 7 in an appendix. We repeat the experiments 5 times and each time lasts 24h. The solid dot lines represent the mean of the result and the shadow around lines are confidence intervals for five runs with 95% confidence level. StateDiver identifies 62%, 380% more unique bypasses than Geneva in Snort, Snort++ respectively. Even when compared with Geneva-*fast* which generates mutations at the same speed, StateDiver still identified 160%, 243% more unique bypasses than Geneva-*fast*. Our experiment shows that Suricata takes better actions to prevent bypass, where all test tools can only find the same bypass. However, with state-discrepancy as guidance, StateDiver and Geneva-*state* are more stable to rediscover the bypass strategy in every test. StateDiver performs 901% and 248% faster finding the first bypass than Geneva-*fast* in Snort and Suricata respectively. In Snort++ our solution is slightly slower, with an average of 0.18h finding the first bypass than 0.08h in Geneva-*fast*. But our solutions can find more unique bypasses over time, whereas the original solution can only find a limited number of repeated strategies.

Unique State Transitions. As discussed in previous sections, all DPIs we investigate have variables reflecting endhosts' TCP state. Their possible values are, or can be converted to eleven optional states of TCP [16] (i.e., LISTEN, SYN_SENT, SYN_RCVD, ESTABLISHED, CLOSE_WAIT, LAST_ACK, FIN_WAIT_1, FIN_WAIT_2, TIME_WAIT, CLOSING and CLOSED). We use dots to represent states, and use arrows to represent state transition (e.g., an arrow from dot SYN_SENT to dot ESTABLISHED represents a packet that makes the state variable of DPI shift from SYN_SENT to ESTABLISHED), then it is possible for us to represent the transitions with state model. For example, Figure 8 shows the state model of Snort Sn_c in a single 24h test with Geneva, Geneva-*fast*, Geneva-*state*, and StateDiver respectively. In each figure we merge all state transitions produced by all strategies in 24h, then we calculate the unique state transition in each figure. Whether comparing Fig. 8c with Fig. 8a, or comparing Fig. 8d with Fig. 8b, state-discrepancy guidance both performs richer state transitions. Note that our state model is the transformation chain of the DPI internal state variable, so it doesn't exactly match standard TCP state changes due to different vendors' designs and modifications.

We repeat the experiments 5 times and each time lasts 24h, then we calculate the average number of unique state transference during the tests (i.e. the number of directed edges in each state model).

Table 5: Numbers of Average Unique State Transference

Target DPI	Variables	GENEVA	GENEVA-*state*	GENEVA-*fast*	STATEDIVER
Snort	Sn_c	15.8	19.8	25.0	29.4
	Sn_s	22.0	24.4	28.6	32.2
Snort++	Sp_c	12.6	20.4	15.8	25.4
	Sp_s	20.8	25.4	27.2	31.4
Suricata	Su_{st}	13.6	18.8	26.4	34.0

Table 5 shows the result. Our solution can cause more unique state transitions in all DPI systems. For example, when testing Snort, the average number of unique state transference of Sn_c reaches 29.4 using STATEDIVER, compared with 25.0 using GENEVA-*fast*. A similar result appears in different state variables and different DPI systems. With more unique state transitions, we may have more potential chances to explore areas that may cause state discrepancies, which may lead to bypass vulnerabilities.

5.4 Comparison with State-of-the-art Elusion Works

We conduct a detailed evaluation to compare STATEDIVER with state-of-the-art results in evading DPI-like systems. Including INTANG [46], lib·erate [38], Geneva [25], SymTCP [47] and Themis [48]. In order to perform the evaluation, we use Wang's attack dataset [22], which implemented all the TCP-related evasion strategies discovered in former works. We apply them to our target DPI systems and summarize methods that can cause successful evasion. Then we compare STATEDIVER with their works manually.

In Table 6, Table 7, and Table 8. ★ means new strategies. △ means previous known strategies. ✓ means their paper mention the strategy in their tool. ✓♯ means their paper do not mention, but we ran out of this strategy in our experiments.

Table 6: Prior Work's TCP-based Strategies and STATEDIVER Found on Snort

Strategy Name	*INTANG*	*lib·erate*	*Geneva*	*SymTCP*	*Themis*	STATEDIVER
△ RST_Bad_Timestamp	✓		✓	✓		✓
△ RST/ACK_Bad_Timestamp	✓		✓	✓		✓
△ In_Window_FIN				✓		
△ RST_Bad_MD5	✓		✓	✓		✓
△ RST/ACK_Bad_MD5	✓		✓	✓		✓
△ Timestamp_Gap				✓		✓
△ Urgent_Data				✓		
△ In_Window_RST				✓		
△ MD5_FIN_ACK				✓		
△ MD5_FIN_Bad_ACK				✓		
△ Multiple_SYNs	✓			✓		
△ RST/ACK_Bad_ACK_Number			✓	✓		✓
△ RST_Bad_SEQ				✓		
△ No_ACK_Flag_FIN				✓		
△ RST_After_FIN					✓	
△ SYN+FIN		✓				
★ FIN_With_Data						✓
★ SYN_Bad_MD5						✓
★ SYN_Fragment			✓♯			✓
★ Fragment_And_Segment			✓♯			✓

Table 7: Prior Work's TCP-based Strategies and STATEDIVER Found on Snort++

Strategy Name	*INTANG*	*lib·erate*	*Geneva*	*SymTCP*	*Themis*	STATEDIVER
△ RST_Bad_Timestamp	✓		✓	✓		✓
△ RST/ACK_Bad_Timestamp	✓		✓	✓		✓
△ RST_Bad_MD5	✓		✓	✓	✓	✓
△ RST/ACK_Bad_MD5	✓		✓	✓	✓	✓
△ RST/ACK_Bad_ACK_Number			✓♯	✓		✓
△ Timestamp_Gap				✓	✓	
△ In_Window_RST				✓	✓	
△ RST_After_FIN					✓	
△ No_ACK_Flag_FIN				✓	✓	
△ In_Window_SYN					✓	
△ TCB_Turnaround	✓		✓		✓	✓
△ Multiple_SYNs	✓			✓	✓	✓
★ SYN_Bad_MD5						✓
★ FIN_Bad_ACK_Number						✓
★ ACK_Bad_ACK_Number_And_ Data_With_Smaller_Timestamp						✓
★ ACK_Bad_MD5_And_Data_ With_Smaller_Timestamp						✓
★ PSH_Before_SYN						✓

Table 8: Prior Work's TCP-based Strategies and STATEDIVER Found on Suricata

Strategy Name	*INTANG*	*lib·erate*	*Geneva*	*SymTCP*	*Themis*	STATEDIVER
△ RST_Bad_MD5	✓		✓	✓	✓	✓
△ RST/ACK_Bad_MD5	✓		✓	✓	✓	✓
△ SEQ_Number_Before_ISN					✓	

As Table 6, Table 7, and Table 8 describe, from all previously published strategies work on Snort, Snort++, and Suricata, STATEDIVER discovered 6, 7, and 2 of them, respectively. STATEDIVER also discovered 4 new strategies in Snort and 5 new strategies in Snort++ that were not previously outlined.

6 DISCUSSION

Manual Effort to Analyze a New DPI. As described in Section 4.2, STATEDIVER performs state instrumentation and extracts trace files from DPIs, using state-discrepancy as feedback to guide our mutation. When facing a new DPI, in order to output trace files, we enable debug options during the configuration process. In most cases, the native debugging information can cover our needs. To make it more elaborate, we add more codes in TCP processing functions, especially in packet accept points, packet drop points, and function return points, where state transference is most likely to occur. It took us around 1 hour, 1.5 hours, and 1 hour to perform state instruction in our test DPIs. It will take less time if the developer is experienced. We believe such manual efforts are acceptable.

Limitations. STATEDIVER is designed to leverage TCP-layer state discrepancy to guide fuzzing DPI systems, it may be less effective when finding bypass leveraging other layers' discrepancies (e.g., HTTP-layer). It may also be hard to find a strategy that doesn't lead to the difference in TCP state transference. For example, the urgent pointer strategy [47] sends a data packet with TCP URG flag and urgent pointer set. Snort will ignore all application data before the urgent pointer but Linux will only consume 1 byte of urgent data and handle the rest of the data. While these packets are being sent, it will not cause TCP-layer state discrepancies. It may depend on higher network level discrepancies and additional

research. TCPFuzz [52] relies on code coverage as feedback to find semantic vulnerabilities. We tried a similar method and found the following problems that are hard to balance. First, we performed branch level, basic block level, and function level instrumentation in DPIs, all of which appeared sensitive to identifying valuable mutations from less promising ones. Second, unlike TCPFuzz tests on TCP implementations which is easy to instrument, DPIs are complex software containing detection logic, protocol logic, and so on. Full instrumentation will bring many disturbances. So we chose state discrepancy as the balanced feedback mechanism in DPI testing. Last, we admit that the strategies discovered in this paper work in limited circumstances. Wang et al. [48] pointed out different behaviors between different versions of Linux kernels. So some strategies may fail with different versions or systems. However, state-discrepancy guidance and different versions are not in conflict, because we focus on discrepancies between DPIs, not DPI and endhosts. We can still find new bypass strategies with state-discrepancy guidance.

Ethical Considerations. We performed the experiment locally so no machines were exposed to a real attack. Besides, we announced our findings only after contacting the vendors and hearing back from the vendors. We acknowledge that finding new strategies may increase potential attack opportunities for other untested DPIs in this paper, since these strategies are likely to be effective on many DPI systems at the same time, we believe that the value in publishing these strategies will make the DPI systems we test more robust and complete, and guide follow-up updates and fixes from other DPI vendors.

7 RELATED WORK

Fuzz Testing. Fuzz testing [9] is an automated software testing technique that provides invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Fuzz testing plays an important role in the discovery of different kinds of vulnerabilities in different fields, like the well-known American Fuzzy Lop (AFL) [1] and its successor AFLGo [26] in software vulnerabilities, Syzkaller [21] in kernel vulnerabilities, and FIRM-AFL [51] in IoT devices threats. We found that fuzzing for stateful network protocols is relatively rare, but has grown in abundance in recent years. Boofuzz [3] is a Python framework that allows users to specify protocol formats and perform fuzzing, which is generation-based fuzzing. But Boofuzz can not perform transport layer mutation and is mostly designed to fuzz application layer, which is not in our field. Joeri de Ruiter and Erik Poll [35] used state machine learning to infer state machines from protocol implementations, and then manually inspected the inferred state machines to look for spurious behavior which might be an indication of flaws. Also, they aimed at high-level protocol. Their fuzzing only does part of the work, and the rest relies on human observation of the state machine model. Pham et al. [39] expanded AFL using server state as feedback to perform coverage-guided grey-box fuzzing for protocol implementations in AFLnet. They explored FTP and RTSP implementations using their work, which highly relies on response codes to get the current protocol state. It is not practicable for protocols that don't have obvious response codes (e.g., TCP). Zou et al. designed TCPFuzz [52] which is also an extension of AFL. It uses a new code coverage metric named branch transition as program feedback and leverages differential testing to find semantic bugs. We find that low-level code coverage as feedback is too sensitive to identify valuable mutations from less promising ones in DPIs.

Deep Packet Inspection Evasion. Ptacek et al. [40] suggested that middleboxes like DPIs are not capable of perfectly reconstructing the data flow in the exact way that the endhost performs. Over the years, many works have been working in this field [38, 46]. Wang et al. [47] used symbolic execution to collect all execution paths to accept points and drop points in Linux TCP implementations, selected and generated packets that may lead to a discrepancy between DPI and the tested server. Finally, they leveraged these sequences to real DPI systems and outputted the valid sequences. It requires manual analysis of Linux TCP stack and is heavily limited by path explosion when performing symbolic execution. Reen and Rossow [41] proposed a differential fuzzing framework to detect strategies to elude stateful DPI systems for QUIC. It is a generation-based fuzzer that does not use any feedback mechanism. Bock et al. [25] used genetic algorithm to automatically detect evasion strategies. Their tool prioritizes inputs that trigger different server-side or DPI-side responses, which may miss intermediate mutations that are necessary for final bypasses. These approaches by Bock et al. [25] and Zou et al. [52] inspire us to find balanced feedback to highlight the most important changes while not dropping necessary mutations early. To the best of our knowledge, we are the first to leverage state discrepancy to guide fuzzing DPI systems.

Protocol Reverse Engineering. There has been considerable work on automated reverse engineering protocol state machines to perform a large number of security tasks, from bot detection to spam detection [27, 29, 34, 44]. They propose languages to describe protocol specifications, handle multiple messages and recover the protocol's state machine. However, protocol reverse engineering tends to reveal a full image of the protocol's state machine, which is not necessary for our scenario. We are trying to uncover the discrepancies between two different state machines and leverage them as guidance to construct evasion packet sequences. Besides, comparing state machines after reverse engineering requires heavy manual work to identify possible evasion routes, which hinders large-scale adoption from testing DPI systems [35]. In contrast, our solution is automated to uncover evasion packet sequences.

8 CONCLUSION

In this paper, we propose StateDiver, the first end-to-end fuzzing framework that uses state discrepancy between different DPI systems as feedback to discover strategies to bypass DPI systems. We used StateDiver to test 3 popular open-source DPI systems and discovered 16 bypass strategies. Our new understanding on the root causes of bypassing DPI systems, a new feedback mechanism for DPI fuzzing, and newly identified DPI vulnerabilities would help enhance not only the current DPI systems but also those to be built in the years to come.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their insightful comments, and a sincere thank you to Dr. Hong Hu for his constructive criticism and suggestion of this paper. This work was supported by the National Natural Science Foundation of China (No. 61902138), the Hubei Province Key R&D Technology Special Innovation Project (No. 2021BAA032), the Wuhan Applied Foundational Frontier Project (No. 2020010601012188), and the Guangdong Provincial Key R&D Plan Project (No. 2019B010139001).

REFERENCES

[1] 2022. AFL (american fuzzy lop) — AFL 2.53b Documentation. https://afl-1.readthedocs.io/en/latest/. (Accessed: 2022-05).
[2] 2022. afl-cve: A Collection of Vulnerabilities Discovered by the AFL Fuzzer (afl-fuzz). https://github.com/mrash/afl-cve. (Accessed: 2022-05).
[3] 2022. Boofuzz: A Fork and Successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz. (Accessed: 2022-05).
[4] 2022. Bugs Found by Syzkaller. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md. (Accessed: 2022-06).
[5] 2022. CVE-2018-14568. https://nvd.nist.gov/vuln/detail/CVE-2018-14568. (Accessed: 2022-06).
[6] 2022. CVE-2019-1010279. https://nvd.nist.gov/vuln/detail/CVE-2019-1010279. (Accessed: 2022-06).
[7] 2022. CVE-2019-18792. https://nvd.nist.gov/vuln/detail/CVE-2019-18792. (Accessed: 2022-06).
[8] 2022. CVE-2021-1236. https://nvd.nist.gov/vuln/detail/CVE-2021-1236. (Accessed: 2022-06).
[9] 2022. Fuzzing - Wikipedia. https://en.wikipedia.org/wiki/Fuzzing. (Accessed: 2022-06).
[10] 2022. Honggfuzz. https://google.github.io/honggfuzz/. (Accessed: 2022-06).
[11] 2022. Iptables(8) - Linux Man Page. https://linux.die.net/man/8/iptables. (Accessed: 2022-06).
[12] 2022. LibFuzzer - A Library For Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html. (Accessed: 2022-06).
[13] 2022. OSS-Fuzz - Continuous Fuzzing For Open Source Software. https://github.com/google/oss-fuzz. (Accessed: 2022-06).
[14] 2022. RFC 1323: TCP Extensions for High Performance. https://www.rfc-editor.org/rfc/rfc1323.html. (Accessed: 2022-06).
[15] 2022. RFC 7323: TCP Extensions for High Performance. https://www.rfc-editor.org/rfc/rfc7323.html. (Accessed: 2022-06).
[16] 2022. RFC 793 - Transmission Control Protocol. https://datatracker.ietf.org/doc/html/rfc793. (Accessed: 2022-06).
[17] 2022. Snort++. https://snort.org/snort3. (Accessed: 2022-05).
[18] 2022. Snort Rules and IDS Software Download. https://snort.org/downloads#snort-downloads. (Accessed: 2022-05).
[19] 2022. StateDiver. https://github.com/CGCL-codes/StateDiver. (Accessed: 2022-10).
[20] 2022. Suricata. https://suricata.io/. (Accessed: 2022-05).
[21] 2022. Syzkaller: An Unsupervised Coverage-Guided Kernel Fuzzer. https://github.com/google/syzkaller. (Accessed: 2022-05).
[22] 2022. Themis Attacks. https://github.com/seclab-ucr/Themis/tree/main/attacks/single. (Accessed: 2022-06).
[23] 2022. What Is Deep Packet Inspection (DPI)? https://www.techtarget.com/searchnetworking/definition/deep-packet-inspection-DPI. (Accessed: 2022-06).
[24] Ralf Bendrath and Milton Mueller. 2011. The End of the Net as We Know It? Deep Packet Inspection and Internet Governance. *New Media & Society* 13, 7 (2011), 1142–1160.
[25] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. 2019. Geneva: Evolving Censorship Evasion Strategies. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*. 2199–2214.
[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[27] Nikita Borisov, David Brumley, Helen J Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. 2007. Generic Application-Level Protocol Analyzer and Its Language. In *Proceedings of the 14th Network and Distributed System Security Symposium*.
[28] Amine Boukhtouta, Serguei A Mokhov, Nour-Eddine Lakhdari, Mourad Debbabi, and Joey Paquet. 2016. Network Malware Classification Comparison Using DPI and Flow Packet Headers. *Computer Virology and Hacking Techniques* 12, 2 (2016), 69–100.
[29] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security*. 621–634.
[30] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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*, Vol. 8. 209–224.
[31] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*. 711–725.
[32] Tommy Chin, Kaiqi Xiong, and Chengbin Hu. 2018. Phishlimiter: A Phishing Detection and Mitigation Approach Using Software-Defined Networking. *IEEE Access* 6 (2018), 42516–42531.
[33] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. *ACM Sigplan Notices* 46, 3 (2011), 265–278.
[34] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. 110–125.
[35] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Security Symposium*. 193–206.
[36] Holger Dreger and Anja Feldmann. 2006. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *Proceedings of the 15th USENIX Security Symposium*. 257–272.
[37] Reham Taher El-Maghraby, Nada Mostafa Abd Elazim, and Ayman M Bahaa-Eldin. 2017. A Survey on Deep Packet Inspection. In *Proceedings of the 12th International Conference on Computer Engineering and Systems*. 188–197.
[38] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David R. Choffnes, Phillipa Gill, and Alan Mislove. 2017. lib•erate, (n): A Library for Exposing (Traffic-Classification) Rules and Avoiding Them Efficiently. In *Proceedings of the 2017 Internet Measurement Conference*. 128–141.
[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification*. 460–465.
[40] Thomas H Ptacek and Timothy N Newsham. 1998. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Technical Report. Secure Networks inc Calgary Alberta.
[41] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 332–344.
[42] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code. USENIX Association, San Francisco, CA.
[43] Daniel Smallwood and Andrew Vance. 2011. Intrusion Analysis With Deep Packet Inspection: Increasing Efficiency of Packet Based Investigations. In *Proceedings of the 2011 International Conference on Cloud and Service Computing*. 342–347.
[44] Gianluca Stringhini, Manuel Egele, Apostolis Zarras, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. 2012. B@bel: Leveraging Email Delivery for Spam Mitigation. In *Proceedings of the 21st USENIX Security Symposium*. 16–32.
[45] Radwan Tahboub and Yousef Saleh. 2014. Data Leakage/Loss Prevention Systems (DLP). In *Proceedings of the 2014 World Congress on Computer Applications and Information Systems*. 1–6.
[46] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Your State Is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *Proceedings of the 2017 Internet Measurement Conference*. 114–127.
[47] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. 2020. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium*.
[48] Zhongjie Wang, Shitong Zhu, Keyu Man, Pengxiong Zhu, Yu Hao, Zhiyun Qian, Srikanth V Krishnamurthy, Tom La Porta, and Michael J De Lucia. 2021. Themis: Ambiguity-Aware Network Intrusion Detection Based on Symbolic Model Comparison. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security*. 3384–3399.
[49] Zhu Xiaogang, Wen Sheng, Camtepe Seyit, and Xiang Yang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (2022), 1–36.
[50] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. 2016. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2991–3029.
[51] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *Proceedings of the 28th USENIX Security Symposium*. 1099–1114.
[52] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 489–502.

APPENDIX

Table 9: Environment Settings

VM's No.	VM's Name	Illustration	RAM	Cores
1	StateDiver	Run our tool	16GB	4
2	*DPI0*	Run DPI0	6GB	2
3	*DPI1*	Run DPI1	6GB	2
4	*Server*	Run a server	4GB	1

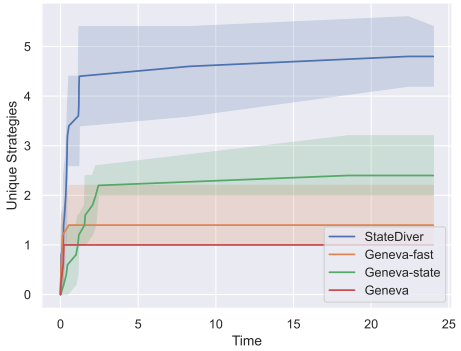

Figure 6: Unique Bypasses Founded by Evaluated Fuzzers for 24h in Snort++

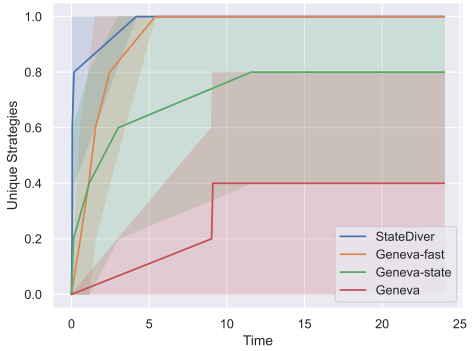

Figure 7: Unique Bypasses Founded by Evaluated Fuzzers for 24h in Suricata

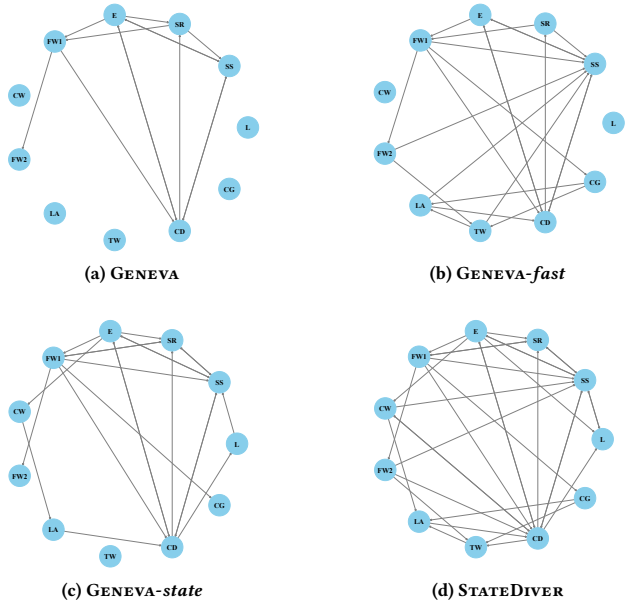

Figure 8: Examples about Unique State Transitions during the Test

TCP State: L - Listen, SS - SYN_SENT, SR - SYN_RECV, E - ESTABLISHED, FW1 - FIN_WAIT_1, CW - CLOSE_WAIT, FW2 - FIN_WAIT_2, LA - LAST_ACK, TW - TIME_WAIT, CD - CLOSED, CG - CLOSING.