SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery

Zhongjie Wang*, Shitong Zhu*, Yue Cao*, Zhiyun Qian*, Chengyu Song*,
Srikanth V. Krishnamurthy*, Kevin S. Chan†, and Tracy D. Braun†
*Department of Computer Science and Engineering, University of California, Riverside,
{zwang048, szhu014, ycao009}@ucr.edu, {zhiyunq, csong, krish}@cs.ucr.edu
†U.S. Army Research Lab, {kevin.s.chan.civ, tracy.d.braun.civ}@mail.mil

***Abstract*—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.**

I. 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 [10], remote exploit prevention [41], phishing attack detection [16], data leakage prevention [46], government network surveillance [7], [6], targeted advertising [28], [3], and traffic differentiation for tiered services [51], [32], [20].

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 [36]) 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 [42], [13], [38]. 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 [37], [48], [29], 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) [37] 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 [37], [48], [29]. 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 paper, 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

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24083
www.ndss-symposium.org

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.

II. 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.

A. 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 [19]. 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 [44]. 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 [37], [23]. An example is that Snort [43] 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 [48], [29] 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.

B. Symbolic Execution vs. Concolic Execution vs. Selective Symbolic Execution

Symbolic execution [26] 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 [11] or VEX [40], or at the binary level [49]), 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 [11]. Moreover, the total number of feasible execution

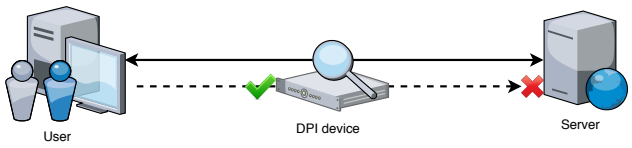

Fig. 1. Threat Model

paths in a common size modern software can be huge, leading to the notorious path explosion problem.

Concolic execution [12] 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 [4]. Most of the state-of-the-art symbolic execution engines like Angr [40] and S2E [17] support concolic execution.

Selective symbolic execution [17] 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 [17] 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.

III. 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.

A. Threat Model

The threat model that we consider is depicted as in Figure 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 paper, 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 [33].

B. 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,
- $T : Q \times \Sigma \rightarrow Q$ is the state transition function, and
- $G : Q \times \Sigma \rightarrow \Lambda$ 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 [13]).

$$T(q, P) = q \wedge G(q, P) = \varepsilon \tag{1}$$

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 §VIII). 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:

$$T(q, P) \neq q \vee G(q, P) \neq \varepsilon \tag{2}$$

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.,

$$\mathcal{G}_{M_s}(q_0, P_{1...n}) = \mathcal{G}_{M_d}(q_0, P_{1...n}) \tag{3}$$

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 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.

$$F(G(q, P)) = \begin{cases} 1 & \text{if G(q,P) contains any bad keyword} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

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 $P_{1...n}$ (Definition 2). ② When handling $P_{1...n-1}$, the state machine of the server and the DPI engine are synchronized (Definition 3). ③ Once P_n is sent, the two state machines would be "de-synchronized" as the DPI engine will drop P_n (Definition 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:

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

[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.

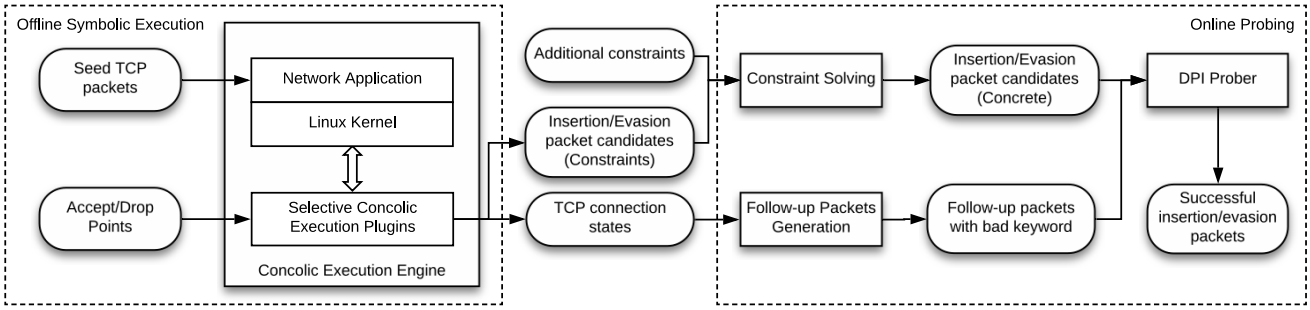

Fig. 2. Overview of SYMTCP's Workflow

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:

$$F(G_{M_s}(\mathcal{T}_{M_s}(q_0, P_{1...n+r-1}), P_{n+r})) = 1 \wedge$$
$$F(G_{M_d}(\mathcal{T}_{M_d}(q_0, P_{1...n+r-1}), P_{n+r})) = 0 \quad (5)$$

Note that our definition of evasion is purely based on the outputs to the application layer and thus, is more strict. Specifically, $P_{1...n-1}$ 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 $P_{1...n-1}$ already ends with an evasion or insertion packet.

Definition 6: Insertion Packet. Given a sequence of packets $P_{1...n} \in \Sigma^*$, 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 $P_{1...n-1}$ but will drop P_n (Definition 1). ② When handling $P_{1...n-1}$, the state machine of the server and the DPI engine are synchronized (Definition 3). ③ P_n will "de-synchronize" the two state machines as the DPI will accept P_n (Definition 2), which has to be inferred through some follow-up probe packets $P_{n+1...n+r}$ where the last packet P_{n+r} contains bad keywords ($r = 1, 2, ...$) (same as Equation 5). $P_{n+1...n+r-1}$ 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 $P_{1...n}$ where the last packet P_n is an evasion/insertion packet.

IV. WORKFLOW OF SYMTCP

An overview of SYMTCP's workflow is depicted in Figure 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 §III-B) 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 illustrates some example packets that reach drop points (Definition 1: the packets do not have any effect and are simply discarded and optionally ACKed) *and* some example packets that reach accept points (Definition 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 [48] 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 [37], making this packet an evasion packet (Definition 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 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

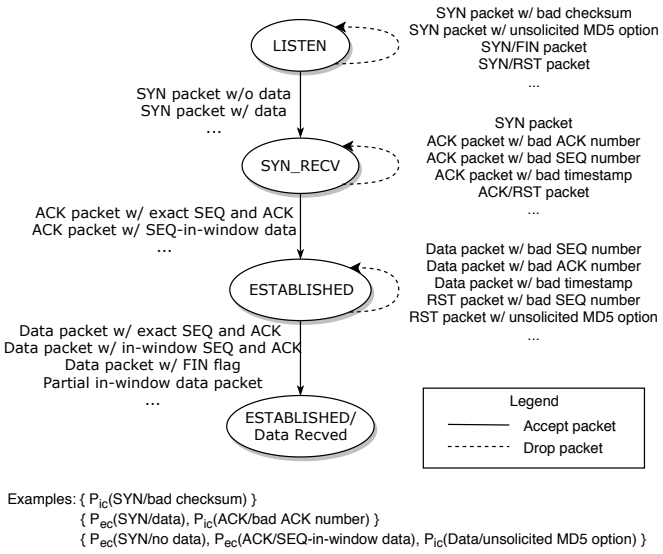

Fig. 3. Candidate packet generation with symbolic execution. P_{ic} denotes candidate insertion packet, P_{ec} denotes candidate evasion packet

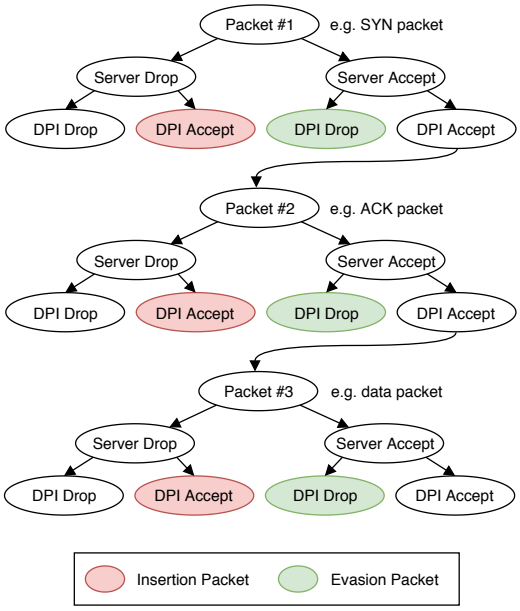

Fig. 4. Evaluation of insertion/evasion packet candidates

in §V); 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, 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 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 §VI). 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 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 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 (Definition 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 [37]), 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).

V. The Offline Phase: Practical Concolic Execution on the TCP Implementation

Our solution is built on top of the popular concolic execution engine S2E [17] 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 §V-A, we describe how we employ selective concolic execution to bound the symbolic execution space. In §V-B, we describe how we symbolize the input, i.e., the fields in the TCP header and options. In §V-C, we discuss how we abstract checksum functions in TCP. Finally, in §V-D, we discuss how to deal with server-side inputs (specifically, the sequence number used by the server) that are not known a priori.

A. 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 [4]). 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 §II), 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.

B. Symbolizing the TCP Header and Options

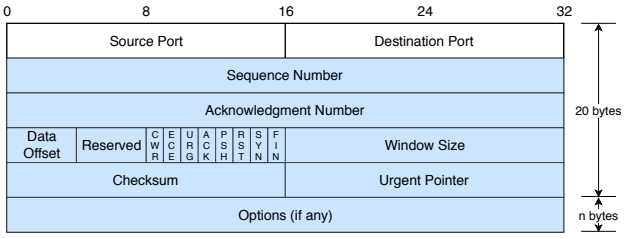

Fig. 5. Symbolized TCP header and options.

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 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 [24], 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.

C. 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.

D. 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 (§VI), 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.

E. Multi-round Symbolic Execution

As mentioned earlier in §IV, 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 §VIII, 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.

VI. Generating online evasion attacks

By means of the offline concolic execution phase described in §V, 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.

A. Constructing insertion/evasion packet candidates

Armed with the constraints relating to each execution path collected during the symbolic execution, as described in §V, 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, $P_{1...n}(n \leq 3)$, to probe the DPI.

There are two additional constraints. The first is the server's initial sequence number (ISN) as mentioned in §V-D. 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.

B. Constructing follow-up probe packets

As mentioned in §III-B, 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.

VII. Implementation

Our system is built upon S2E 2.0 [17], 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.

A. 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.

B. Online Constraint Solving

We use the state-of-the-art Z3 [50] 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.

VIII. 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.

A. Experiment Setup

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 §III-B).

TABLE I. 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

TABLE II. 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.

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 I. To save space, we put the raw table in an appendix.

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:

```
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 §III-B 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.

B. 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 §V-B, 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 II, 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).

TABLE III. 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

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 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 III, 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.

C. 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 VI, 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 [48] 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.

D. Zeek (formerly known as Bro)

Zeek [34] is very liberal in accepting incoming packets.[2] It is therefore relatively easy to bypass using insertion packets. We list some strategies in Table IV. 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 [37], [25], [48]. 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 [37]) 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.

E. Snort

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 V. 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 [37] (though not all of them are tested in practice), and the usage of TCP MD5 option was done in [48].

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 the appendix.

The second novel strategy is related to the urgent pointer processing logic, which is notoriously ambiguous [36] and often implemented incorrectly, even in major OSes such as Linux [22]. 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 [22]). 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,

TABLE IV. SUCCESSFUL STRATEGIES ON ZEEK V2.6

Strategy	TCP state	Insertion/Evasion packet	Linux	Zeek
† SYN with data	L/SR/E	(I) SYN packet with data	Ignore data	Accept data
† Multiple SYN	SR/E	(I) SYN packet with out-of-window SEQ num	Discard and send ACK	Reset TCB
† Pure FIN	E	(I) Pure FIN packet without ACK flag	Discard (may send ACK)	Flush and reset receive buffer
† Bad RST/FIN	SR/E	(I) RST or FIN packet with out-of-window SEQ num	Discard (may send ACK)	Flush and reset receive buffer
† Data overlapping	SR/E	(I) Out-of-order data packet, then overlapping in-order data packet	Accept in-order data	Accept first data
† Data without ACK	SR/E	(I) Data packet without ACK flag	Discard	Accept
† Data bad ACK	E	(I) Data packet with ACK $>$ snd_nxt or $<$ snd_una - window_size	Discard	Accept
$*$ Big gap	SR/E	(I) Data packet with SEQ $>$ rcv_nxt + max_gap_size (16384)	Accept	Ignore later data
$*$ SEQ $<$ ISN	SR/E	(E) Data packet with SEQ num $<$ client ISN and in-window data	Accept in-window data	Ignore

* TCP State: L - Listen, SR - SYN_RECV, E - ESTABLISHED. (I) - Insertion, (E) - Evasion. † - Old strategy, $*$ - New strategy.

TABLE V. SUCCESSFUL STRATEGIES ON SNORT V2.9.13

Strategy	TCP state	Insertion/Evasion packet	Linux	Snort
† Multiple SYN	E	(I) SYN packet with in-window SEQ num	Discard and send ACK	Teardown TCB
† In-window FIN	E	(I) FIN packet with SEQ num in window but $\neq$ rcv_nxt	Ignore FIN (may accept data)	Cut off later data
† FIN/ACK bad ACK	E	(I) FIN/ACK packet with ACK num $>$ snd_nxt or $<$ snd_una - window_size	Discard (may send ACK)	Cut off later data
† FIN/ACK MD5	SR/E	(I) FIN/ACK packet with TCP MD5 option	Discard	Cut off later data
† In-window RST	E	(I) RST packet with SEQ num $\neq$ rcv_nxt but still in window	Discard and send ACK	Teardown TCB
† RST bad timestamp	SR	(I) RST packet with bad timestamp	Discard	Teardown TCB
† RST MD5	SR/E	(I) RST packet with TCP MD5 option	Discard	Teardown TCB
† RST/ACK bad ACK num	SR	(I) RST/ACK packet with ACK num $\neq$ server ISN + 1	Discard	Teardown TCB
$*$ Partial in-window RST	E	(I) RST packet with SEQ num $<$ rcv_nxt but partial data in window	Discard	Teardown TCB
$*$ Urgent data	SR/E	(E) Data packet with URG flag and urgent pointer set	Consume 1 byte urgent data	Ignore all data before urgent pointer
$*$ Time gap	SR/E	(E) Data packet timestamp = last timestamp + 0x7fffffff/0x80000000	Accept	Ignore

* TCP State: L - Listen, SR - SYN_RECV, E - ESTABLISHED. (I) - Insertion, (E) - Evasion. † - Old strategy, $*$ - New strategy.

Snort discards all the data before the urgent pointer offset and fails to reconstruct the HTTP request.

F. Great Firewall of China

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 [48]. 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 [48]. 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 [48].

Comparing our strategies with previous works on GFW with manually crafted packets [48], [29], we have rediscovered all the TCP-layer strategies used in [29] (except the IP and HTTP layer strategies which are beyond our scope). In addition, we have rediscovered all the primitive strategies used in [48], 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.

TABLE VI. SUCCESSFUL STRATEGIES ON THE GFW

Strategy	TCP state	Insertion/Evasion packet	Linux	GFW
† Bad RST	SR/E	(I) RST packet with bad checksum or TCP MD5 option	Discard	Teardown TCB
† Bad data	SR/E	(I) Data packet with bad checksum or TCP MD5 option or bad timestamp	Discard	Accept
† Data without ACK	SR/E	(I) Data packet without ACK flag	Discard	Accept
∗ SEQ $\leq$ ISN	SR/E	(E) Data packet with SEQ num $\leq$ client ISN and in-window data	Accept in-window data	Ignore
∗ Small segments	SR	(E) Data packet with payload size $\leq$ 8 bytes	Accept	Ignore
∗ FIN with data	SR/E	(I) FIN packet with data and without ACK flag	Discard	Teardown TCB
∗ Bad FIN/ACK data	E	(I) FIN/ACK packet with data and bad checksum or TCP MD5 option or bad timestamp	Discard	Teardown TCB
∗ FIN/ACK data bad ACK	E	(I) FIN/ACK packet with data and ACK num $>$ snd_nxt or $<$ snd_una - window_size	Discard	Teardown TCB
∗ Out-of-window SYN data	SR	(I) SYN packet with SEQ num out of window and data	Discard and send ACK	Desynchronized
∗ Retransmitted SYN data	SR	(I) SYN packet with SEQ num = client ISN and data	Discard	Desynchronized
∗ RST bad timestamp	SR	(I) RST packet with bad timestamp	Discard	Teardown TCB
∗ RST/ACK bad ACK num	SR	(I) RST/ACK packet with SEQ num $\neq$ server ISN + 1	Discard	Teardown TCB

* TCP State: L - Listen, SR - SYN_RECV, E - ESTABLISHED. (I) - Insertion, (E) - Evasion. † - Old strategy, ∗ - New strategy.

IX. 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 §IV and §V): 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 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 Protocols / 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 §VIII. 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 [5], [30]. 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 [23], [18], [47], 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 [23] 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 [39], 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.

X. 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. [37]. 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. [25] and Wang et al. [48] followed the same principle to study evasions against the Great Firewall of China and demonstrated their effectiveness in practice. Li et al. [29] 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. [9] 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 [14], [15], 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. [27] 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. [45] 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. [2], [1] 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. [31] 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 [8], [21], [35]. 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.

XI. Conclusion

In this paper, 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.

Acknowledgment

We would like to thank Muhammad Faizan Ul Ghani, who helped us with data analysis, and Hang Zhang for his helpful comments. We thank the anonymous reviewers for their insightful feedback. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. It was also partially supported by NSF award CNS-1652954, CNS-1619391, CNS-1718997, and ONR under grant N00014-17-1-2893.

References

[1] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1690–1701. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978383

[2] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis, "Back in black: towards formal, black box analysis of sanitizers and filters," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 91–109.

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

[4] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, May 2018. [Online]. Available: http://doi.acm.org/10.1145/3182657

[5] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC 16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189200. [Online]. Available: https://doi.org/10.1145/2991079.2991114

[6] R. Bendrath, "Global technology trends and national regulation: Explaining variation in the governance of deep packet inspection," in *International Studies Annual Convention*, vol. 15, no. 18, 2009.

[7] R. Bendrath and M. Mueller, "The end of the net as we know it? deep packet inspection and internet governance," *New Media & Society*, vol. 13, no. 7, pp. 1142–1160, 2011.

[8] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," *Commun. ACM*, vol. 60, no. 2, p. 99107, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3023357

[9] K. Bock, G. Hughey, X. Qiang, and D. Levin, "Geneva: Evolving censorship evasion strategies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 21992214. [Online]. Available: https://doi.org/10.1145/3319535.3363189

[10] A. Boukhtouta, S. A. Mokhov, N.-E. Lakhdari, M. Debbabi, and J. Paquet, "Network malware classification comparison using dpi and flow packet headers," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 69–100, May 2016. [Online]. Available: https://doi.org/10.1007/s11416-015-0247-x

[11] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[12] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[13] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, L. M. Marvel, Y. Cao, T. Dao, L. M. Marvel, Z. Wang, Z. Qian, and S. V. Krishnamurthy, "Off-path tcp exploits of the challenge ack global rate limit," *IEEE/ACM Trans. Netw.*, vol. 26, no. 2, pp. 765–778, Apr. 2018. [Online]. Available: https://doi.org/10.1109/TNET.2018.2797081

[14] S. Y. Chau, O. Chowdhury, M. E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. 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*. IEEE Computer Society, 2017, pp. 503–520. [Online]. Available: https://doi.org/10.1109/SP.2017.40

[15] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, "Analyzing semantic correctness with symbolic execution: A case study on pkcs# 1 v1. 5 signature verification." in *NDSS*, 2019.

[16] T. Chin, K. Xiong, and C. Hu, "Phishlimiter: A phishing detection and mitigation approach using software-defined networking," *IEEE Access*, vol. 6, pp. 42 516–42 531, 2018.

[17] V. Chipounov, V. Kuznetsov, and G. 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*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950396

[18] S. Dharmapurikar and V. Paxson, "Robust tcp stream reassembly in the presence of adversaries," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251403

[19] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer, "Dynamic application-layer protocol analysis for network intrusion detection," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267354

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

[21] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, no. 6. ACM, 2008, pp. 206–215.

[22] F. Gont and A. Yourtchenko, "On the implementation of the tcp urgent mechanism," RFC 6093, Jan. 2011.

[23] M. Handley, V. Paxson, and C. 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*, ser. SSYM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267612.1267621

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

[25] S. Khattak, M. Javed, P. D. Anderson, and V. 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*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/foci13/workshop-program/presentation/Khattak

[26] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[27] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New

York, NY, USA: ACM, 2011, pp. 26–37. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018440

[28] A. Kuehn and M. Mueller, "Profiling the profilers: deep packet inspection and behavioral advertising in europe and the united states," *Available at SSRN 2014181*, 2012.

[29] F. Li, A. Razaghpanah, A. M. Kakhki, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove, "Lib·erate, (n): A library for exposing (traffic-classification) rules and avoiding them efficiently," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: ACM, 2017, pp. 128–141. [Online]. Available: http://doi.acm.org/10.1145/3131365.3131376

[30] I. Mironov and L. Zhang, "Applications of sat solvers to cryptanalysis of hash functions," in *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 102115. [Online]. Available: https://doi.org/10.1007/11814948_13

[31] S.-J. Moon, J. Helt, Y. Yuan, Y. Bieri, S. Banerjee, V. Sekar, W. Wu, M. Yannakakis, and Y. Zhang, "Alembic: Automated model inference for stateful network functions," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'19. Berkeley, CA, USA: USENIX Association, 2019, pp. 699–718. [Online]. Available: http://dl.acm.org/citation.cfm?id=3323234.3323291

[32] M. L. Mueller and H. Asghari, "Deep packet inspection and bandwidth management: Battles over bittorrent in canada and the united states," *Telecommunications Policy*, vol. 36, no. 6, pp. 462–475, 2012.

[33] S. J. Murdoch and S. Lewis, "Embedding covert channels into tcp/ip," in *Proceedings of the 7th International Conference on Information Hiding*, ser. IH'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 247–261. [Online]. Available: http://dx.doi.org/10.1007/11558859_19

[34] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(99)00112-7

[35] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 543553. [Online]. Available: https://doi.org/10.1145/2970276.2970316

[36] J. Postel, "Transmission control protocol," RFC 793, Sep. 1981.

[37] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," SECURE NETWORKS INC CALGARY ALBERTA, Tech. Rep., 1998.

[38] A. Quach, Z. Wang, and Z. Qian, "Investigation of the 2016 linux tcp stack vulnerability at scale," *SIGMETRICS Perform. Eval. Rev.*, vol. 45, no. 1, pp. 8–8, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3143314.3078510

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

[40] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[41] D. Smallwood and A. Vance, "Intrusion analysis with deep packet inspection: increasing efficiency of packet based investigations," in *2011 International Conference on Cloud and Service Computing*. IEEE, 2011, pp. 342–347.

[42] M. Smart, G. R. Malan, and F. Jahanian, "Defeating tcp/ip stack fingerprinting," in *USENIX Security*, 2000.

[43] Snort - network intrusion detection & prevention system. [Online]. Available: https://www.snort.org/

[44] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 262–271. [Online]. Available: http://doi.acm.org/10.1145/948109.948145

[45] J. Song, C. Cadar, and P. Pietzuch, "Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE Trans. Softw. Eng.*, vol. 40, no. 7, pp. 695–709, Jul. 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2323977

[46] R. Tahboub and Y. Saleh, "Data leakage/loss prevention systems (dlp)," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE, 2014, pp. 1–6.

[47] M. Vutukuru, H. Balakrishnan, and V. Paxson, "Efficient and robust tcp stream normalization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 96–110. [Online]. Available: https://doi.org/10.1109/SP.2008.27

[48] Z. Wang, Y. Cao, Z. Qian, C. Song, and S. V. Krishnamurthy, "Your state is not mine: A closer look at evading stateful internet censorship," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: ACM, 2017, pp. 114–127. [Online]. Available: http://doi.acm.org/10.1145/3131365.3131374

[49] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[50] The z3 theorem prover. [Online]. Available: https://github.com/Z3Prover/z3

[51] Y. Zhang, Z. M. Mao, and M. Zhang, "Detecting traffic differentiation in backbone isps with netpolice," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 103–115. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644905

APPENDIX

A. A complete list of drop points in Linux kernel v4.9.3

All of the source-code-level drop points we labeled can be found in Table VII.

B. TCP PAWS checking in Linux and Snort

As shown in the pseudo-code Listing 1 and Listing 2, the acceptable TSval ranges of Linux and Snort are "off by 2".

```
1  if ((signed int)(last_packet->tsval -
   current_packet->tsval) <= 1) {
2      // PAWS check succeeded
3  }
```

Listing 1. Pseudo-code of Linux PAWS (timestamp) check

```
1  if ((signed int)((current_packet->tsval -
   last_packet->ts_val) + 1) < 0) {
2      // PAWS check failed
3  }
```

Listing 2. Pseudo-code of Snort PAWS (timestamp) check

TABLE VII. 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