Leveraging State Information for Automated Attack Discovery in Transport Protocol Implementations

Samuel Jero
Purdue University
sjero@purdue.edu

Hyojeong Lee
Purdue University
hyojlee@purdue.edu

Cristina Nita-Rotaru
Purdue University
cnitarot@purdue.edu

***Abstract*—We present a new method for finding attacks in unmodified transport protocol implementations using the specification of the protocol state machine to reduce the search space of possible attacks. Such reduction is obtained by appling malicious actions to all packets of the same type observed in the same state instead of applying them to individual packets. Our method requires knowledge of the packet formats and protocol state machine. We demonstrate our approach by developing SNAKE, a tool that automatically finds performance and resource exhaustion attacks on unmodified transport protocol implementations. SNAKE utilizes virtualization to run unmodified implementations in their intended environments and network emulation to create the network topology. SNAKE was able to find 9 attacks on 2 transport protocols, 5 of which we believe to be unknown in the literature.**

I. INTRODUCTION

Transport protocols provide end-to-end communication in a layered network architecture by implementing guarantees such as reliability, in-order delivery, and congestion control. They are used not only directly by applications, but also by Internet services such as BGP and secure protocols such as SSL. The most well known transport protocol is TCP, which underlies the majority of Internet communication today and provides connections, reliability, in-order delivery, flow control, and congestion control to applications that use it.

The design and implementation of transport protocols is complex, with many components, special cases, error conditions, and interacting features. Further, many implementations are written in low level languages like C for improved performance and make use of error-prone, but highly efficient, constructs like pointer manipulation and type casting. These low level constructs are difficult to address by model checking systems, making such systems of limited use beyond checking the protocol design. Unfortunately, transport protocol implementations often sacrifice simplicity and ease of understanding for improved performance, resulting in a high probability of bugs introduced during implementation.

Although there are few transport protocols in common use, because of their ubiquitous role in network communication, there are many different implementations and variants of these transport protocols. For example, the `nmap` security scanner is able to detect 3,079 distinct TCP/IP network stack configurations in its most recent version [1]. This includes printers, VoIP phones, routers, and embedded systems, along with general purpose operating systems. While many of these may be different configurations of a few common networking stacks, these variations represent different handling of particular network conditions, which often implies the exercise of different code paths.

Despite the importance of these protocols and the complexity and number of their implementations, the testing of transport protocol implementations has been mainly a manual and ad-hoc process [2], [3], [4]. This lack of systematic testing for transport protocols and their implementations has resulted in a stream of new bugs and attacks [5], [2], [6], [3]. Consider TCP, one of the most well studied and well tested network protocols; the list of discovered attacks extends from the mid-1980's to the present day [7], [8], [9], [10], [11], [12], [13]. Many of these attacks have been discovered repeatedly or rediscovered again in slightly different contexts.

Prior work in testing network protocol implementations has focused on easing the development of manual tests [2], [14] and on enabling deeper testing for crashes by using stateful fuzzing techniques [15], [16], [17]. Other work has focused on systematic testing by leveraging techniques like symbolic execution [18], [5] and dynamic interface reduction [19] in combination with concrete attack execution. Many of these techniques require access to the source code and require heuristics to efficiently handle low level constructs like type casting, pointer casting, and function pointers, which are heavily used in network protocol implementations. The major challenge faced by all of these approaches is search space explosion.

In this paper we focus on automated attack finding for transport protocol implementations. Specifically, we leverage information about the packet formats and protocol state machine to automatically create attack scenarios consisting of malicious actions performed on protocol packets in targeted protocol states. Knowledge of the packet formats enables the generation of malicious packets based on packet type while information about the state machine allows the tracking of the current state of the protocol at runtime. State tracking is achieved without code instrumentation by monitoring the packets while malicious packet manipulation is achieved using a network proxy. By inferring the current state of the protocol state machine, our method can perform malicious actions on all packets of a particular type in a particular protocol state instead of on individual packets, significantly reducing the search space. The protocol state machine also allows us to identify key points for attack injection in the transport protocol, ensuring wide coverage. Note that the state machine and packet formats are an important part of any protocol specification.

Hyojeong Lee is now with Google, Inc. This work was done while at Purdue University.

As such, they are often readily available in the specification documents themselves. For proprietary protocols where the specification of the state machine may not be available, recent work in state machine inference may be leveraged [20].

Our approach works with *unmodified* implementations irrespective of their operating system, programming language, or required libraries. It does not require access to the source code, enabling the testing of a wide range of transport protocol implementations, including proprietary, closed-source systems.

The contributions of this paper are:

- We present a new approach to search space reduction without instrumenting the code. This approach leverages the description of the protocol state machine to identify critical points in the search space for attack injection and to explore the implementation more thoroughly. We use knowledge of the packet formats to perform a variety of malicious actions, including packet field manipulation, and apply these malicious actions to packet type, protocol state pairs instead of individual packets, enabling significant state space reduction. We also use the protocol state machine to ensure that we test all protocol states, providing wide coverage.
- We demonstrate our approach with SNAKE, our new tool for finding attacks on unmodified transport layer protocol implementations running in arbitrary operating systems and in realistic networks. SNAKE (State-based Network AttacK Explorer) uses virtualization to run unmodified transport layer implementations in their intended environments and a network emulator to tie these virtual machines together into a realistic, emulated network. The network emulator intercepts and modifies packets, tracks the current protocol state during execution, and uses this information to create packet-based attacks at specific points in the state machine execution. SNAKE is general for use on many transport protocols, requiring only the description of the packet header formats and the transport protocol state machine as input.
- We use SNAKE to examine a total of 5 implementations, 2 transport protocols—TCP and DCCP—, and 4 operating systems. We find 9 attacks, 5 of which are, to the best of our knowledge, unknown in the literature. We also compare our state-based attack search with two baseline approaches and show its effectiveness in search space reduction.

The rest of this paper is organized as follows. Section II reviews related work. Section III discusses the system and attack models we consider. Sections IV and V present the design and the implementation of our system, respectively. Section VI shows our results, including the attacks we discovered, while Section VII concludes our work.

II. Related Work

A variety of other works have looked at automatically finding vulnerabilities in network protocols or distributed systems. One common method for doing this is to combine model checking techniques with actual execution. DeMeter [19] is one such tool that uses a technique called dynamic interface reduction to reduce the search space. The key insight of this technique is to hide local non-determinism by splitting a distributed system into components that communicate via message passing.

Several other systems [18], [5] leverage a technique called symbolic execution. Symbolic execution simulates code execution using symbolic variables and updates these variables with constraints as the code runs [21]. MACE [18] combines symbolic execution with concrete execution and protocol state machine inference. The inferred protocol state machine is used as a search space map to allow deep exploration and enable parallelism. From each state, a combination symbolic/concrete execution system is started. The symbolic execution identifies new code paths to explore with further concrete execution. MAX [5] utilizes symbolic execution to find manipulation attacks against network protocols. MAX takes some metric of performance and possible vulnerable lines of code and uses symbolic execution and concrete testing with a malicious proxy to attempt to repeatedly force that vulnerable statement to be executed. In contrast, SNAKE does not rely on model checking techniques that require access to the protocol source code or manual marking of vulnerable statements.

Another method for finding vulnerabilities in network protocols and their implementations is fuzzing. KiF [15] is one such fuzzer. It is designed to test SIP implementations for crashes or fatal errors and makes use of the SIP packet format as well as the SIP protocol state machine to cover deeper and more relevant portions of the search space. Interestingly, the authors infer a protocol state machine for each implementation they test instead of using the state machine from the protocol specification. SNOOZE [14] and EXT-NSFSM [16] are other network protocol fuzzers. SNOOZE [14] also targets the SIP protocol and makes use of the protocol state machine to track the target implementation. However, SNOOZE requires the user to provide a fuzzing scenario and so does not need to deal with state space reduction. EXT-NSFM [16] uses the state machine of a target protocol to enable deeper fuzzing of application protocols like FTP. In particular, it tracks the protocol state machine to determine what part of the protocol to fuzz without unnecessarily restarting the application. Both tools search for crashes or other fatal errors.

Packetdrill [2] is a framework for creating tests for network protocol stacks. It is designed to help reproduce bugs or ease the writing of regression tests. The rich environment Packetdrill provides allows sending and receiving packets using a `tcpdump`-like syntax as well as the performance of system calls and the running of arbitrary shell commands or python scripts. SNAKE is orthogonal to Packetdrill. They have different goals and can compliment each other: our approach offers great breadth for test coverage while Packetdrill provides depth for specific test cases.

Turret [6] is a platform for finding performance attacks against intrusion tolerant distributed systems. Turret inserts a malicious proxy in front of an unmodified implementation to simulate a malicious attacker and uses a greedy search strategy to look for the malicious actions that cause the largest impact in system performance. SNAKE uses a different set of malicious actions that are tailored for transport-layer, two-party protocols instead of multi-party application protocols. Further,

since intrusion tolerant distributed systems are designed to be attack resistant, Turret is able to use a greedy search strategy that looks for actions that cause performance impacts and then combines them. By contrast, SNAKE uses a search strategy based on the network protocol state machine.

III. System and Attack Model

In this section we provide an overview of transport protocols and describe the attacks that we consider in this paper. As usually such protocols are deployed in a client-server setting, we will refer to the two parties as client and server.

A. Transport Protocols

Transport layer protocols provide end-to-end communication between two applications running on two different hosts. They use the concept of a port to allow multiple applications to use the same host simultaneously and provide a checksum to protect data from accidental corruption as it travels from one host to the other. With the exception of UDP [22], which provides only unreliable data delivery, most transport protocols provide additional services such as: (1) reliability, (2) ordered delivery, (3) flow control, and (4) congestion control. Many transport protocols are connection-oriented, as both parties need to maintain state. Connection-oriented protocols consist of three phases: connection establishment, data transfer and connection tear-down.

Connection establishment. Connection establishment, typically in the form of a handshake, takes place before any data can be exchanged between the client and the server. During connection establishment the client and server exchange sequence numbers, set sequence windows, and allocate necessary buffers. Errors or delays in this phase may lead to connection termination without any data transfer.

Data transfer. Once a connection is established, data flows between the two parties. During this phase, packets may be buffered by the sender, to guarantee reliability, and by the receiver, to enforce ordered delivery to the application. Additionally, system parameters such as congestion window size and timeouts are dynamically adjusted to implement flow control and congestion control.

Reliability. Reliability is usually implemented using acknowledgments and retransmissions. The sender uses a buffer to store data that has been sent and includes a sequence number on each packet. Periodically, the receiver sends an acknowledgment to the sender. When the sender receives this acknowledgment, it determines what data has been lost and retransmits this data. Data acknowledged as received correctly is also removed from the sender's buffer. Since there is the possibility of acknowledgments being dropped by the network, the sender includes a timer to retransmit data if no acknowledgment of sent data has been received after some lengthy time interval. Note also that transport protocols are allowed to declare failure after several retransmissions and terminate the connection without having delivered the data. Failure of reliability will result in connection termination or an improper change in sending rate due to interactions with congestion control.

Ordered delivery. Ordered delivery guarantees that data sent by one application is received at the other in the same order that it was sent. This is related to reliability and the two are usually implemented together. Implementing ordered delivery also requires a packet sequence number, allowing the receiver to determine the sending order. Packets received out of order are buffered at the receiver until the missing packets are received. The packets can then be delivered to the application in order. A failure of ordered delivery would result in receiver buffer overflow or the delivery of out-of-order data to the application.

Flow control. Flow control ensures that a sender does not overwhelm a slow receiver with more data than it can buffer. The goal is for the sender to send at the same rate that the receiver is receiving. Flow control is specified as a sliding window indicating the data that the receiver can currently buffer. The sender is then limited to sending that window of data before receiving an acknowledgment indicating that the window has either slid forward or increased in size. Issues with flow control will cause unnecessarily slow throughput or receiver buffer overflows and data retransmissions.

Congestion control. Congestion control serves two related purposes. First, it protects against congestion collapse in the network, and second, it provides fairness between competing flows. Congestion collapse occurs when severe network congestion, or over-utilization, results in the network spending the majority of its time sending data that will eventually be dropped. This results in a persistent drop in throughput. Ultimately, congestion control operates by detecting indicators of congestion and slowing down the sending rate in response. The means of detecting congestion and the precise details of the response to congestion vary significantly between transport protocols and even within the same protocol. Often, dropped packets or increased RTT are used to identify congestion. Issues with congestion control will cause the sending rate to be increased or decreased improperly and unnecessarily.

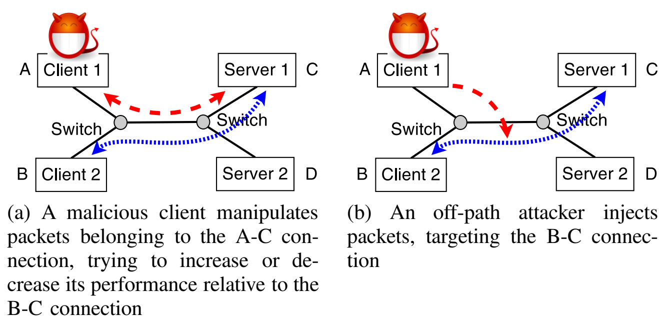
A particularly important property of congestion control is *fairness*. That is, if two flows are competing over bandwidth on a bottleneck link, they should share the bandwidth equally. The networking community has generally understood this to mean that the flows achieve throughput within a factor of two of each other [23], [24]. Issues with fairness may cause unfair competition between flows, possibly resulting in starvation.

Connection tear down. After all desired data has been transferred, there must be a way for client and server to signal this to each other and release all state associated with the connection. Like connection establishment, connection tear down takes place through a handshake in which the two hosts indicate that they are done sending data and are ready to close the connection. A failure to properly tear down the connection may result in the associated state staying around on both hosts much longer than desired and resources remaining allocated.

B. Attack Goals

We focus on attacks that target any of the phases of a transport protocol: connection establishment, data transfer, or connection tear down. In the case of data transfer, we consider attacks against all goals: reliability, ordered delivery, flow control, and congestion control (including fairness).

Connection-related attacks. An attacker can interfere with the connection establishment or connection tear down



(a) A malicious client manipulates packets belonging to the A-C connection, trying to increase or decrease its performance relative to the B-C connection

(b) An off-path attacker injects packets, targeting the B-C connection

Fig. 1. Examples of attacker location and target connections

protocols by preventing them from achieving their goals: establishing a connection or cleanly terminating a connection.

Preventing connection establishment attacks are actions taken by a malicious host to prevent some target connection from being established and transferring useful data. These actions occur at the same approximate time as the target connection attempt and target the very core of the transport protocol by preventing a user from successfully initiating a connection.

End-host resource exhaustion attacks are actions taken by a malicious host to force the other end of the connection to exhaust some limited resource (memory, sockets, etc) in order to deny service to other (legitimate) requests, creating a denial of service condition. These actions occur in already established connections and target future connections that have yet to be attempted. These attacks are conducted by malicious clients against a server in an attempt to prevent legitimate users from accessing the provided service.

We do not consider connection hijacking attacks. These attacks require sampling from the target implementation or the collusion of a low-privilege component on the victim.

Performance-related attacks. An attacker can also target the performance of an individual connection, either by seeking to degrade the throughput of some target connection or by seeking to compromise the fairness of the protocol's congestion control algorithm.

Throughput degradation attacks are actions taken by a malicious host to decrease the throughput of some target connection, often with the intention of making the connection so slow that it is useless to the initiating application. These attacks target the congestion control and flow control algorithms of the transport protocol.

Fairness attacks are actions taken by a malicious host to unfairly increase its throughput at the expense of other competing connections. This type of attack directly targets the fairness of the transport protocol's congestion control algorithm. Many of these attacks also indirectly compromise the reliability of the transport protocol.

We do not consider denial of service conditions that result from an attacker consuming all of a service's network bandwidth nor those resulting from a sheer overwhelming number of connections. The transport layer can do nothing to prevent these attacks.

C. Attacker Interaction with the Protocol

We consider a client-server setup where the attacker is either a compromised client or an off-path third party. Note that an attacker can also conduct **on-path attacks**. For example, modifying data in transit or dropping connection initiation requests. We do not consider such attacks since transport protocols such as TCP are not usually designed to address these attacks.

Malicious client. In this case, the attacker is a compromised client. As shown in Figure 1(a), the attacker is one of the endpoints so he can view all packets in the connection, create arbitrarily formed packets, and respond arbitrarily to incoming packets. This may include ignoring received packets, delaying or duplicating responses, or setting unusual field values in sent packets or sequences there of. Such an attacker can target the fairness of the network protocol by seeking to gain more than his fair share of network bandwidth. He may also seek to deny bandwidth to other flows by abusing his connection with the server or use repeated connections to cause resource exhaustion on the server.

Off-path attacker. In this case, the attacker is not one of the endpoints of the connection, but a third party, placed off-path. As shown in Figure 1(b), the attacker cannot view or modify the packets in the target connection. Instead, he is limited to spoofing packets, either individually or in sequences, such that they appear to originate at either the client or the server. An attacker in this position is likely to seek to attack the ability to establish the target connection or the congestion control of that connection.

IV. Design

In this section, we discuss the design of SNAKE. We first provide an overview of our approach, then describe how we utilize the state machine of the protocol to reduce the search space and generate attack strategies. Finally, we describe the packet-level basic attacks we consider.

A. Overview

We focus on finding attacks in unmodified implementations of transport protocols. We consider attacks on connection establishment as well as resource exhaustion attacks, throughput degradation attacks, and fairness attacks. These attacks can be identified by examining the results of an attempted data transfer. Specifically, connection establishment attacks can be identified by observing a target connection that transfers no data. Resource exhaustion attacks result in incomplete socket cleanup at the server. Throughput degradation attacks and attacks on fairness can be identified by unfair competition between a target connection and its competitor; throughput degradation attacks target the low throughput connection while attacks on fairness target the high throughput connection. All of these attacks can be detected by running the protocol for a relatively short period of time.

We select an environment that combines virtualization with network emulation. Virtualization allows us to test a wide range of implementations independent of language, operating system, or access to source code. The network emulation provides us the reproducible measurements and attack isolation

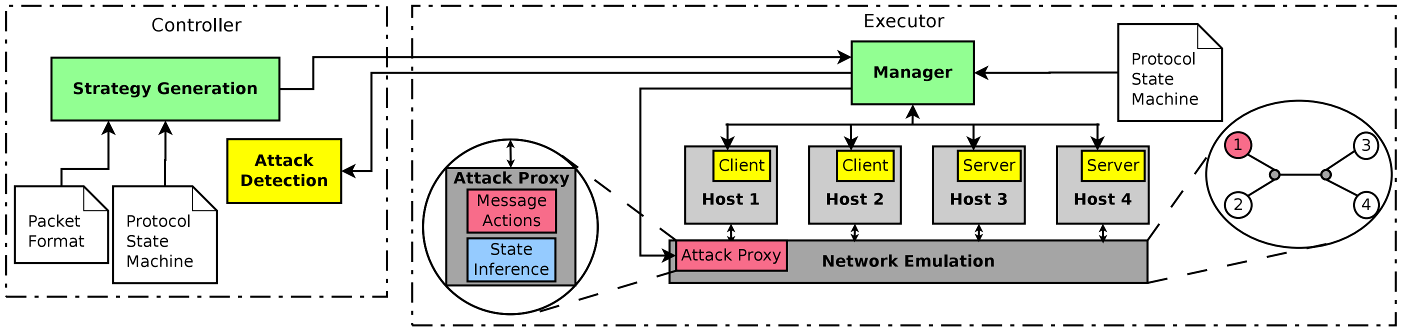


Fig. 2. Design of SNAKE

needed to detect performance-related attacks. Figure 2 presents our system design.

The attack strategies we consider can be created by packet manipulation and injection based on the packet type and the individual packet fields. These strategies are selected from a set of basic attacks derived from information about packet formats. For instance, an attack strategy may be to duplicate packets of type W ten times, or to inject a new packet of type X with field 3 set to Y, or to modify field 5 of packet type Z to 555. Each of these attack strategies are performed in particular protocol states.

To determine what kinds of basic attacks would be most useful, we performed a detailed literature study on transport protocol attacks and identified some common components, or building blocks, used in many of these attacks. Based on this study, we defined a set of packet-based basic attacks that we use to compose attack strategies.

As we do not require access to the source code, our approach relies on intercepting and modifying or injecting network traffic. We place an attack proxy between one of our test hosts and the emulated network. This attack proxy intercepts packets and can apply basic attacks such as influencing the delivery of packets or modifying the packets flowing through it. We can also use the proxy to emulate an off-path attacker that injects new packets into the network.

We detect if an attack was successful or not by comparing the connection performance under attack with a baseline generated from a test with no attacks and by checking for open sockets on the server after the test completes. Attack strategies that appear successful are tested a second time to ensure repeatability.

B. Attack Injection

An important aspect of determining an attack search strategy is identifying the attack injection points, that is, the points where attacks can be inserted into a test run.

Send-packet-based attack injection. One simple approach is to have the proxy intercept each packet generated by the client application running in the virtual machine, apply any basic attacks desired, and forward the packet on to its destination. This means that an attack injection point occurs whenever there is a send for a particular packet type.

While this approach is relatively simple and can find many attacks, it also results in repeatedly performing attacks that have the same semantics for the protocol, thus resulting in redundant executions and lengthening the time required to complete the search. In addition, this approach does not work well for off-path attackers and fails to find attacks not connected with packet send events in the code. This is particularly problematic for transport protocols because many attacks against connection establishment and tear down fall into this category.

Time-interval-based attack injection. One approach to provide support for off-path attackers and finer time granularity is to divide the running time into fixed intervals and, for each of these intervals, attempt to inject packets following all basic attacks. While this approach is also relatively simple, a small time interval must be used in order to catch many attacks. This will result in testing thousands of strategies that either do not inject attacks or inject many redundant attacks, based on the semantics of the protocol. As a result, this approach also has a high execution time overhead and can take a very long time to complete.

Protocol state aware attack injection. Our approach to eliminate some of the redundant testing scenarios, support off-path attackers, and provide finer granularity for injecting attacks is to take into account the semantics of the protocol when injecting attacks. We can obtain information about the semantics of the protocol from its state machine. Many transport protocols have well documented state machines describing their connection lifecycles, and in the absence of such documentation, work in state machine inference may be leveraged [20].

We propose a state-based search strategy that leverages several characteristics of the protocol state machine to reduce the attack search space. Specifically, we inject attacks at specific states in the protocol execution. Because the protocol state machine defines key points in the operation of the protocol, this approach allows us to quickly gain wide coverage within the search space by focusing on each of these states. We also treat all attack injection points in the same state in the same manner. This further prunes the number of search paths to be explored. The motivation behind our approach is that two packets of the same type received in the same protocol state usually cause similar results; however, an identical packet received in two different states may cause significantly different results.

In order to apply our *protocol state aware attack injection*, we need a mechanism to infer which protocol state an endpoint is in. As we do not require access to the source code, we

use packet monitoring to infer the state. This is accomplished by a state tracking component (see Figure 2) that uses a description of the protocol state machine supplied by the user. The state machine provides information about what packets determine transitions from one state to another. At run time, the state machine tracker infers changes in the state machines of each endpoint by observing the packets exchanged and matching them with the state transition rules. The state tracking component also keeps track of some basic information about each observed state, including the packet types observed in that state.

Note that this strategy assumes that implementations have correctly implemented the protocol state machine as described in their specification. Existing work on state machine verification [25] could be leveraged to overcome this limitation. However, the high granularity state machines, describing connection lifecycle, that we use are unlikely to be implemented incorrectly because of their simplicity and importance to the protocols. Taking TCP as an example, the state machine has 11 states in total and all data transfer, and associated retransmissions and congestion control, takes place in a single state [26]. A mistake in this state machine has a similar impact to getting the packet header formats wrong; while the implementation may work with itself, it will fail simple interoperability tests.

C. Attack Strategy Generation

Based on the packet types and state machine information, we automatically generate attack strategies. For each packet type we generate the basic attacks described below.

We conducted an extensive study of the literature on transport protocol attacks to develop our basic attacks. All of these attacks are conducted by our attack proxy at a packet level, either one packet at a time or considering several packets together.

Malicious client attacks. The first set of basic attacks we developed interfere with packet delivery or packet content. Packet delivery attacks model a malicious client who either ignores certain packets entirely or who delays processing packets in order to interfere with the protocol. Packet content attacks model a malicious client who sends packets that contain unexpected or invalid values.

We consider the following *packet delivery attacks*: drop, duplicate, delay, and batch.

Drop: The attack proxy intercepts and drops a packet with a given probability specified as a parameter in percent. This attack may impact many of the core features of transport protocols from connection establishment to congestion control, depending on when it is applied.

Duplicate: The attack proxy intercepts a packet and then sends multiple copies of it to the destination. The number of duplicates to inject is specified as a parameter. This attack could impact many features of a transport protocol, but fairness and congestion control are particularly vulnerable. Acknowledgment duplication, in particular, can cause fairness problems [11].

Delay: The attack proxy intercepts a packet and then inserts a delay before sending it on. The delay is specified as a parameter in seconds. Depending on the length of the delay, this attack may cause reordering or retransmission situations. It may also interfere with RTT estimation, which is usually a key component of retransmission algorithms.

Batch: The attack proxy intercepts packets and waits some amount of time before sending them all at once. The wait time is a parameter specified in seconds. This attack is designed to find attacks similar to the Shrew and Induced-Shrew attacks [9], [8].

We also consider the following *packet content manipulation attacks*: reflect and lie.

Reflect: The attack proxy intercepts a packet and sends it back to its originating host. This attack models sending an unexpected, but potentially valid, packet. It is particularly likely to disrupt connection establishment and termination. Consider, for example, the TCP Simultaneous Open Attack where an attacker responds to a SYN packet with another SYN packet [7].

Lie: The attack proxy intercepts a packet and modifies a specified field before sending it on. Modifications supported include setting particular values, setting random values, or adding/subtracting/multiplying/dividing the current value by some factor. The field and the type of modification are parameters. We use a list of modifications chosen based on the field-type to be likely to cause unexpected behavior. These include setting values like 0, the maximum value a field can handle, and the minimum value a field can handle. This attack may impact all of the core features of transport protocols from connection establishment to congestion control, depending on when and where it is applied.

Off-path attacks. The second set of attacks we developed are attacks on a connection by an off-path third party. These attacks spoof packets such that they appear to come from the client or the server in a target connection. We consider the following off-path attacks: inject and hitseqwindow.

Inject: The attack proxy injects a new packet into the network. This attack contains a number of parameters describing the fields in the packet, its source and destination, and when it should be injected (in seconds from emulation start). Many parts of a transport protocol may be affected by such an attack, from reliability to connection tear down.

HitSeqWindow: This attack is very similar to inject. Instead of injecting just one packet, the attack proxy injects a whole series of packets with their sequence numbers spanning the whole possible sequence range. This attack is designed to look for attacks similar to the Reset and Syn-Reset attacks on TCP [12], [3].

Note that one can also consider more complex attack strategies that combine the basic attacks described above into strategies consisting of sequences of actions. We currently support only the basic attacks described above.

V. Implementation

In this section, we discuss how we implement SNAKE. We first present an overview of the whole platform and then discuss our attack proxy, state tracking, and parallelism in more detail. See also Figure 2.

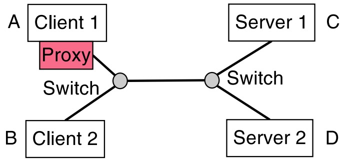


Fig. 3. Test Network Topology

A. Overview

We separate the functionality of SNAKE into two components: a *controller* that generates attack strategies and one or more *executors* that test the strategies.

The controller generates and selects the attack strategies based on the packet formats and the state machine transitions obtained from the protocol specification supplied by the user. An executor first runs a non-attack test and then, for each strategy, runs the attack scenario and reports performance information back to the executor, who determines whether an attack took place or not. SNAKE uses parallelism to run multiple executors concurrently and speed up the attack finding process.

The executor controls the execution of a testing scenario consisting of a set of four virtual machines each running an unmodified instance of the protocol under test. These virtual machines are connected in a dumbbell topology using a network emulator and tap devices. We use KVM as the virtualization environment and NS-3 for network emulation.

A dumbbell topology consists of two machines on each side of a bottleneck link as shown in Figure 3. In our setup, the two machines on one side act as servers while the two on the other act as clients. We configured our attack proxy to be between one of the clients and the bottleneck link. The other client makes a connection to a server that we refer to as *competing connection*, as it will compete with our proxy for bandwidth on the bottleneck link. This topology allows us to test both attacks that impact a connection to which the attacker is a party and attacks where the attacker is an off-path third party. The first type of attack often represents an attack on the fairness of the transport protocol or a resource-exhaustion-based denial of service attack against a server. The second type of attack often represents an off-path attacker who wishes to terminate or slow a connection between two other hosts. See Section III-C and Figure 1 for a more detailed discussion of these types of attacks.

To determine successful attacks, the controller examines the performance of the client without the attack proxy (Client 2 in Figure 3) and the number of connections the server is maintaining at the end of the test. This information is obtained by the executor. Specifically, the executor calculates performance as the quantity of data transferred during the test and queries the OS to determine the number of connections maintained by the server, for example by using the `netstat` command on UNIX-based systems. After the test completes, the executor sends these metrics to the controller, which compares the received metrics observed after the tested attack with the metrics observed in a non-attack test run.

The executor is implemented as a Perl script that listens for strategies from the controller and then initializes the virtual machines from snapshots, starts the network emulator, configures the attack proxy, and starts the test. Once the test completes, it collects the performance data and any feedback from the attack proxy and sends this back to the controller.

The controller is implemented in a combination of C and Perl and is responsible for choosing strategies to execute and determining attacks based on the performance data returned by the executor. Instead of generating all of the attack strategies at once, we implement our controller to generate them a few at a time in response to feedback about packet types and protocol states observed by the state tracking component of our attack proxy. This is equivalent to generating all the strategies at once but is a little more flexible.

B. Attack Proxy

Our attack proxy intercepts all packets along the ingress and egress paths in NS-3. We modify NS-3 to allow us to designate malicious nodes and only intercept packets to or from those nodes. The interception is done in NS-3's tap-bridge module, which connects NS-3 to outside tap-devices serving the virtual machines.

When the attack proxy receives a packet, it examines it to determine the protocol. Protocols not of interest are returned to the tap-bridge for normal processing. For packets of the target protocol, the type of the packet is examined and the sender's protocol state is identified from the state tracking system. If there is a matching strategy, the basic attack is performed on the packet. To accomplish this, our proxy needs a description of the protocol packet header format. We use a simple language to describe the header structure and then automatically generate C++ code to parse and modify this header.

Our malicious proxy is also capable of injecting packets into the network. Proper packet headers are generated from the protocol description using our automatically generated C++ protocol processing code, and the resulting packet can then be sent using standard NS-3 packet send mechanisms.

C. State Tracking

We implement our protocol state machine tracking inside the attack proxy. The tracker takes a description of the protocol state machine, written in the `dot` language [27], as input. This description contains the state transitions, including the packets or actions that cause these transitions or result from them. The use of a standardized graph language like `dot` to represent the state machine enables the use of SNAKE on a variety of two-party protocols simply by swapping out the state machine and packet header descriptions.

Our state machine tracker watches the packets that pass through the proxy and uses the state machine transition rules to infer what state the client and server are currently in. The state machine tracker also collects some useful statistics about each state in the protocol. This includes what packet types and how many packets were sent and received during each state. It also includes the amount of time the host spent in each state and the number of times it visited that state. These statistics are extracted from the attack proxy by the executor at the end of each test and then sent to the controller along with the performance information.

D. Parallelism

We have implemented SNAKE as separate controller and executor modules to enable parallelism. These modules can even reside on separate systems, as all communication is done via TCP. Because testing each strategy takes about two minutes this becomes a highly parallel problem, with linear speedup limited only by the amount of processing power that can be thrown at the problem.

Each executor requires significant resources, as it will start four virtual machines and an NS-3 instance. In practice, we found that running about one executor for every six hyperthreads resulted in good performance. The memory requirements per executor depend primarily on the demands of the implementation and operating system under test. In our tests, they ranged around 4-8GB per executor.

Our controller requires little processing power since its primary responsibility is to identify attacks based on the performance information returned by the executors and to supply new attack strategies to the executors. In our experiments, we did not find it necessary to dedicate a core to the controller.

VI. Results

We applied SNAKE to test two protocols and a total of five transport protocol implementations on four different operating systems. The two protocols we tested were TCP and DCCP. For TCP, we tested implementations in Linux 3.0.0, Linux 3.13, Windows 8.1, and Windows 95. For DCCP, we focused on the implementation in Linux 3.13. We were able to find attacks on all implementations, including several previously unknown attacks. We discuss these protocols and present our findings below, and summarize them in Tables I and II.

All of these tests were run on a hyperthreaded 16 core Intel® Xeon® 2.3GHz system with 94GB of RAM. We ran five separate executors simultaneously. Testing each implementation required about 60 hours, but this duration could be decreased by running more executors.

We define successful attacks as strategies that result in an increase or decrease in achieved throughput of at least 50% compared to the non-attack case or that cause the server-side socket to not be released normally after the connection is closed. This throughput threshold is based roughly on the notion that reasonable competition for network flows is achieving throughput within a factor of two of each other [23], [24] as well as on experience.

A. TCP

TCP [26] is the most common transport protocol today, underlying the majority of all Internet traffic. Its goal is to provide a reliable byte-stream between end hosts. As a result, it implements reliability, in-order delivery, and flow control. It also attempts to ensure fairness and prevent congestion collapse by implementing congestion control.

A TCP connection is started by a handshake between the two end hosts [26]. This allows both endpoints to inform each other of their initial sequence numbers and any important options. A similar handshake is performed at the end of the connection to make sure that all data has been delivered before the connection terminates.

Reliability is achieved by using sequence numbers and acknowledgments. The sender assigns a sequence number to each byte of data and then the receiver acknowledges the highest consecutive byte of data it has received [26]. Retransmissions are triggered either by a retransmission timeout (RTO) or by receiving three duplicate acknowledgments, indicating the reception of packets above some missing bytes [29].

TCP uses several flags in its header to indicate certain types of packets. The packets in the initial handshake are marked with the `SYN` flag; those in the final handshake with the `FIN` flag. Reset packets use the `RST` flag to abruptly terminate a connection after an error. An `ACK` flag indicates a valid acknowledgment field and is set on every packet after the initial `SYN`. An important side-effect of using a set of flags instead of a single packet-type field is that TCP implementations have to decide how to handle unusual or nonsensical flag combinations, for example `SYN+FIN+ACK`.

TCP congestion control is a complex research area in its own right; however, the basic scheme is Additive Increase, Multiplicative Decrease [29] where TCP slowly increases its sending rate by one packet per RTT in steady state and cuts the sending rate in half on packet loss.

Testing. We tested TCP in one of its most popular settings. Specifically, we utilized a large HTTP download with Apache or IIS running on the servers and `wget` for clients.

For each of our TCP implementations, SNAKE tried between five and six thousand strategies and determined that between 128 and 163 of these (depending on the implementation) resulted in significant performance degradation or potential for resource exhaustion. These attack strategies represent around 3% of the tested strategies.

On-path attacks. Some of the attacks we found, while possible, require an on-path attacker. Strategies like modifying the source or destination ports or the header size do prevent a connection from being established, but these strategies are not possible for off-path attackers and a malicious client could simply not initiate a connection. These attacks can be conducted by an on-path attacker. However, as TCP was not designed to handle such attackers, we are not interested in these types of attacks.

False positives. We found a few attacks that were false positive strategies for each implementation. These were related to the *hitseqwindow* basic attack. This attack injects numerous packets in an attempt to get one packet into the sequence window of a target connection. Unfortunately, the injection of such a large number of packets tends to slow down the target connection significantly, irrespective of whether the packets have any malicious impact. We manually inspect the packet captures for attacks using this action to determine why an attack was declared and identify false positives when the reduced performance is caused by the number of packets injected, and not by hitting the target sequence window.

Client and off-path attacks. Discarding the false positive and on-path attacks results in a set of between 17 and 48 (depending on implementation) attack strategies. However,

TABLE I. SUMMARY OF SNAKE RESULTS

Protocol	Implementation	Strategies Tried	Attack Strategies Found	On-path Attacks	False Positives	True Attack Strategies	True Attacks
TCP	Linux 3.0.0	5994	128	82	5	41	4
TCP	Linux 3.13	5717	163	105	10	48	3
TCP	Windows 8.1	5549	137	118	2	17	4
TCP	Windows 95	5013	147	122	3	22	3
DCCP	Linux 3.13	4508	67	27	2	38	3

TABLE II. SUMMARY OF ATTACKS DISCOVERED BY SNAKE

Protocol	Attack	Description	Impact	Operating System	Known
TCP	CLOSE_WAIT Resource Exhaustion	Connections hang on server if client exits and resets are dropped	Server DoS	Linux 3.0.0 / Linux 3.13	Partially [28]
TCP	Packets with Invalid Flags	The handling of invalid flag combinations could allow OS fingerprinting	Fingerprinting	Linux 3.0.0 / Windows 8.1	No
TCP	Duplicate Acknowledgment Spoofing	Frequently duplicating acknowledgments causes sender to increase window faster than normal	Poor Fairness	Windows 95	Yes [11]
TCP	Reset Attack	Brute force a sequence-valid reset	Client DoS	All	Yes [13]
TCP	SYN-Reset Attack	A sequence-valid SYN causes connection reset	Client DoS	All	Yes [3]
TCP	Duplicate Acknowledgment Rate Limiting	Occasionally duplicating acknowledgments result in indicated loss and connection slow down	Throughput Degradation	Windows 8.1	No
DCCP	Acknowledgment Mung Resource Exhaustion	Connection will hang waiting for timeouts to empty send queue if acknowledgments are disrupted	Server DoS	Linux 3.13	No
DCCP	In-window Acknowledgment Sequence Number Modification	Connection can be throttled by incrementing sequence number in an acknowledgment, resulting in a forced resync	Throughput Degradation	Linux 3.13	No
DCCP	REQUEST Connection Termination	Any packet except Response received in REQUEST state results in connection reset	Client DoS	Linux 3.13	No

many of these strategies are functionally the same attack, just performed on a different field or with a different value. Ultimately, we found a total of six unique attacks, several of which are effective against multiple implementations. We discuss each of these attacks in detail below.

1) CLOSE_WAIT Resource Exhaustion Attack: This attack results in connections staying alive on the server in the `CLOSE_WAIT` state for tens of minutes after the client closes them. An attacker can easily initiate hundreds of thousands of such connections before they begin to expire, likely rendering the server unavailable.

`CLOSE_WAIT` is the TCP state that the passive close side of a TCP connection, usually the server, remains in after receiving notification of remote close and while waiting for the local application to close the connection. After the local close, the connection must remain in this state until a `FIN` can be sent.

If a Linux TCP client exits while in the middle of a data transfer (like an HTTP download), Linux will send a `FIN` packet and then not acknowledge any more data on the connection; any further packets will generate a reset. This is valid behavior according to the RFC since the application will never receive this data [26]. If these reset packets are blocked, it will appear to the sending TCP that the whole in-flight window of packets was lost, triggering congestion avoidance and a series of retransmissions that will never succeed.

When the server application eventually closes the TCP connection, TCP will transition to the `CLOSE_WAIT` state where it needs to remain until all outstanding data is acknowledged, including the lost window of packets that were in-flight when the client exited. These packets will never be acknowledged, meaning that TCP is stuck in `CLOSE_WAIT` with (possibly significant) data queued on the socket. Linux will eventually force-close a TCP connection due to lack of delivery, but that requires 15 retries by default, which is between 13 and 30 minutes depending on the RTT [30].

To the best of our knowledge, this attack is unreported in the research literature. However, system administrators have been aware of similar problems with connections stuck in `CLOSE_WAIT` for many years [28]. SNAKE found this attack on Linux 3.0.0 and Linux 3.13.

2) Packets with Invalid Flags: Recall that the TCP header includes several flags that indicate the packet type. Not all combinations of these flags make sense. For instance, a packet with `SYN+FIN+ACK+RST` flags would indicate a packet starting a connection, closing the connection, acknowledging a packet in the connection, and resetting the connection. This is clearly a nonsensical combination. One would expect a TCP implementation to ignore such invalid packets. However, both Linux 3.0.0 and Windows 8.1 respond to such invalid packets in an active connection.

Linux 3.0.0 attempts to interpret these nonsensical flag combinations as best it can. This results in sending a duplicate acknowledgment in response to a packet with no flags set, a situation that is never valid. We have also observed Linux 3.0.0 attempting to process `SYN+FIN` and `SYN+FIN+ACK+PSH` packets. Note that Linux 3.13 appears to have fixed these problems and no longer responds to such invalid packets.

Windows 8.1 will also process and respond to invalid packets. However, it follows a different approach. If the `RST` flag is set, the connection is reset irrespective of what other flags might also be set. Otherwise, nonsensical flag combinations are ignored.

Responding to packets with invalid flag combinations is not by itself a security issue. We have found no instance where responding to invalid flag combinations achieves something that is not possible with valid flag combinations. However, a target's responses to invalid flag combinations could be used to fingerprint the particular TCP implementation in use, indicating other possible vulnerabilities to exploit. Further, packets with invalid flag combinations may be interpreted differently by end hosts and middleboxes like firewalls and intrusion

detection systems, providing a possible way to subvert such middleboxes.

3) Duplicate Acknowledgment Spoofing: This is a classic TCP attack originally discovered by Savage, et al. in 1999 [11]. This attack operates against a naïve TCP implementation where the sender increases its congestion window for every acknowledgment received, without checking for duplicates or checking how much data is currently outstanding in the network. As a result, a receiver can significantly increase its achieved throughput by simply acknowledging packets multiple times, thereby increasing the sender's congestion window much faster than normal.

This attack requires frequent duplication of acknowledgments to be meaningful, as each acknowledgment only increases the congestion window by a very small amount. In addition, if acknowledgments are duplicated more than three times, TCP will react as if a loss occurred, halve its congestion window, and enter fast recovery. However, in this mode, each acknowledgment received results in a new packet being sent. This simplifies the attack by allowing the attacker to control the sending rate by controlling the acknowledgment rate.

There are mitigations to this attack, including only allowing the congestion window to be incremented by the number of data segments outstanding in the network. Another option would be a nonce in the TCP header and a sender side register allowing acknowledgment of each nonce only once.

In our tests, SNAKE discovered this attack against Windows 95 and was able to use it to increase a malicious connection's throughput by a factor of 5. SNAKE did not find this attack against any other tested implementation, which is expected as this attack and its mitigations were well known by the time they were released.

4) Reset Attack: This attack works by spoofing a large number of resets for a target connection. If one of these resets is sequence-valid, the receiving TCP will reset the connection. The work in [13] showed this attack to be much more practical than previously supposed by pointing out that a reset packet anywhere in the receive window is sufficient to reset the connection. Thus, one could send packets at receive window intervals, greatly reducing the number of packets required.

In our testing, SNAKE discovered this attack against all of our TCP implementations. Since this attack utilizes a feature of the TCP specification itself, all implementations should be vulnerable. The only thing implementations can do to protect themselves is to keep their receive window small.

5) SYN-Reset Attack: This attack is very similar to the Reset Attack discussed above. In this case, the TCP specification says that the receipt of a sequence-valid `SYN` packet on an active connection should result in the connection being reset. As a result, an attacker can spoof a large number of `SYN` packets at receive window intervals in an attempt to slip one into the target connection's sequence window, resulting in a connection reset. This attack has been known since at least 2009 [3].

In our testing, SNAKE discovered this attack against all of our TCP implementations. Like the Reset Attack, this attack utilizes a feature of the TCP specification itself, which makes it difficult for implementations to protect against.

6) Duplicate Acknowledgment Rate Limiting: Duplicate Acknowledgment Rate Limiting is a new attack that SNAKE discovered against Windows 8.1. It operates by duplicating `PSH+ACK` packets, which occur only occasionally in the data stream, ten times. This causes duplicate acknowledgments to be sent to the sender by the receiver. After three duplicate acknowledgments, the sender halves its congestion window and retransmits the indicated packet.

So far, this is standard TCP behavior common to all TCP New Reno implementations. However, for a Windows 8.1 server and a Linux 3.0.0 client, we observe a throughput degradation of a factor of 5 compared to the competing flow. Both of the Linux implementations we tested show throughput consistent with normal TCP competition in this scenario; that is, approximately fair bandwidth sharing.

B. DCCP

The Datagram Congestion Control Protocol (DCCP) was designed for applications that wanted congestion control, but did not want the retransmissions and head-of-line-blocking associated with TCP [31]. Examples of such applications are applications that are highly latency sensitive, such as VoIP, realtime streaming video, and video gaming.

Like TCP, DCCP requires a handshake to setup a connection and another one to tear the connection down. However, DCCP uses different types of packets for these handshakes, instead of a set of flags [31]. Hence, the initial handshake consists of a `REQUEST` and a `RESPONSE` packet while the final handshake consists of a `CLOSE` and a `RESET` packet.

DCCP assigns sequence numbers to packets instead of bytes. Further, every packet increments the sequence number; even pure acknowledgments carrying no data [31]. The receiver acknowledges the highest sequence number received; since DCCP does not retransmit data, a TCP-like cumulative acknowledgment does not make sense.

However, this design means that DCCP endpoints can get out of sync after extended bursts of loss and reject valid packets as not within the current sequence window. To mitigate this issue, a third handshake–of `SYNC` and `SYNCACK` packets–is used to exchange the current sequence numbers of both parties and resynchronize the connection [31].

DCCP also features pluggable congestion control modules, known as CCIDs. Two are currently standardized: CCID 2, TCP-like Congestion Control, and CCID 3, TCP-Friendly Rate Control (TRFC). We focus on CCID 2 in this work. It follows the TCP SACK congestion control algorithm as closely as possible [32]. There are several changes in order to handle the switch from byte-based to packet-based sequence numbers.

Testing. For DCCP testing, we used `iperf` to measure throughput. Since DCCP is not a reliable protocol, we measured performance based on server goodput, or actual data received. As DCCP is currently only supported on Linux and is fairly uncommon, we focused our efforts on a single implementation, the Linux kernel 3.13 implementation.

SNAKE tried just over 4,500 strategies against DCCP. Of these, it identified 67 candidate strategies that caused significant performance issues or potential resource exhaustion. This is about 1.5% of the total strategies tested.

On-path attacks. As with TCP, DCCP was not designed to be resilient to on-path attacks. Thus, we exclude all on-path attacks found by SNAKE.

False positives. We also found 2 attacks that were false positives. As with TCP, these attacks are both *hitseqwindow* strategies that attempt to inject packets into a target connection at sequence window intervals. Injecting this quantity of packets tends to significantly slow down the competing target connection, irrespective of any malicious impact of the injected packets. Thus, these strategies tend to fall below our attack threshold.

Client and off-path attacks. Discarding the on-path attacks and the two false positives leaves us with 38 strategies that represent actual attacks. However, many of these strategies are functionally the same attack, just repeated on different fields or with different values. Ultimately, we found three attacks; none of which have been reported in the literature. We discuss each of these attacks below.

1) Acknowledgment Mung Resource Exhaustion Attack: This attack is possible because a DCCP sender will not close a connection until its send queue is empty. This send queue defaults to 10 packets, but may be much larger for applications like video streaming. As a result, if a connection's congestion control can be persuaded to send at the minimum rate, a connection can be held in an open-but-useless state for a very long time. By repeating this process, one can create an effective resource exhaustion attack that may render the target host unavailable.

Note that DCCP does not retransmit data. As a result, while similar attacks against TCP last until TCP gives up retransmitting a particular packet and resets the connection, DCCP will continue sending at its minimum rate until the application and the human trying to use it explicitly close the connection. Once the application closes the connection, DCCP will send all queued packets and then close the connection and free related resources.

There are several ways to convince DCCP's congestion control to send at its minimum rate. Most of them work by invalidating or dropping the acknowledgments from the receiver. Modifying the sequence or acknowledgment numbers are very effective because this results in an additional exchange of `SYNC` and `SYNCACK` packets.

2) In-window Acknowledgment Sequence Number Modification: This attack targets sequence numbers in the receiver's acknowledgment packets. Recall that sequence numbers in DCCP are per-packet and that every packet increments the sequence number; even pure acknowledgment packets.

If the sequence number of one of these acknowledgments is increased, such that it is still sequence valid, the sender will begin to acknowledge this bad acknowledgment number in its data packets. However, when the receiver receives these data packets it will find they acknowledge packets that have not yet been sent. As a result, it will drop these packets and send a `SYNC` in response. The `SYNC` packet will result in a `SYNCACK` packet from the sender, resynchronizing the sequence numbers and allowing the connection to proceed. However, by that point an entire window of packets will have been dropped, resulting in DCCP's congestion control reducing the connection's allowed sending rate. It may even trigger a timeout and subsequent slow start, assuming DCCP's CCID 2 congestion control is in use.

To perform this attack, an attacker does not have to be an endpoint. It suffices to be able to sniff and spoof network traffic. Such an attacker can inject an acknowledgment with a slightly higher sequence number and trigger this vulnerability.

3) REQUEST Connection Termination Attack: This attack is an effective way to terminate a connection during the connection initiation phase. A client enters the `REQUEST` state on initiating a connection, immediately after having sent a `REQUEST` packet to the server, and stays in this state until it receives a `RESPONSE` packet from the server.

The only valid packets in the `REQUEST` state are `RESPONSE` or `RESET`; any other packet results in a reset. Note that both the pseudo-code in RFC 4340 [31] and the Linux 3.13 DCCP implementation perform this packet type check *before* checking the sequence numbers. Thus, it is possible to reset a DCCP connection in the `REQUEST` state by sending *any* non-`RESPONSE` packet with *any* sequence and acknowledgment numbers.

This makes the attack exploitable by anyone who can sniff and spoof packets. An off-path, third party attacker can launch this attack, if they can guess the connection initiation time (to within an RTT) and the source port.

C. Benefits of State-based Strategy Generation

Our state-based strategy generation algorithm enabled us to find 9 attacks against 2 transport protocols and a total of 5 implementations. 5 of these attacks were previously unknown. To accomplish this, we required about 60 hours per tested implementation. Removing parallelism, this becomes 300 hours of computation per tested implementation.

By contrast, the time-interval-based attack injection approach discussed in Section IV-B requires trying our malicious strategies at intervals of 5 microseconds, which is roughly the amount of time needed to send a minimum sized TCP packet at 100Mbits/sec. Thus, there are 12 million possible injection points in a 1 minute test connection. For each of these injection points, we would have to test about 60 different malicious strategies resulting from the 8 general malicious actions and the 13 fields in the TCP header. This results in 720 million strategies to test.

At 2 minutes to test each strategy, this would require 24 million hours of computation. At an equivalent level of parallelism, this would take 548 years to complete, which is clearly impractical.

The send-packet-based attack injection approach is more practical. A one minute non-attack test with TCP results in the sending of about 13,000 packets. For each of these packets, we would need to test about 53 different malicious strategies for packet manipulation, resulting in a total of 689,000 strategies. This would require 22,967 hours of computation. At an equivalent level of parallelism, this would take about 191 days.

The send-packet-based attack injection also provides no support for packet injection attacks modeling third party, off-path attackers. As a result, it would be impossible to find the Reset and Syn-Reset attacks using this attack injection model.

VII. CONCLUSION

Transport layer networking protocols form an important part of the Internet, yet, to date, their testing has been mostly manual and ad-hoc. This has resulted in a stream of vulnerabilities stretching back to the 1980's. To help remedy this situation, we present SNAKE, a tool to allow systematic testing of unmodified transport protocol implementations, utilizing the protocol state machine to reduce the search space. We demonstrate SNAKE by testing 2 different protocols, TCP and DCCP, and 5 implementations, including both open-source and closed-source systems. We found 9 attacks, 5 of which we believe to be unknown in the literature. SNAKE requires only a description of the protocol packet headers and protocol state machine, both readily obtained from protocol specification documents. We believe SNAKE can contribute to securing the transport layer of modern network stacks.

ACKNOWLEDGMENT

This material is based in part upon work supported by the National Science Foundation under Grant Number CNS-1223834. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] G. Lyon, "Nmap," 2014. [Online]. Available: http://nmap.org/

[2] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H.-k. J. Chu, A. Terzis, and T. Herbert, "Packetdrill: Scriptable network stack testing, from sockets to packets," in *USENIX Annual Technical Conference*. USENIX, 2013, pp. 213–218.

[3] Centre for the Protection of National Infrastructure, "Security assessment of the transmission control protocol," Centre for the Protection of National Infrastructure, Tech. Rep. CPNI Technical Note 3/2009, 2009.

[4] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz, "Known TCP implementation problems," RFC 2525 (Informational), Mar. 1999.

[5] N. Kothari, R. Mahajan, T. Millstein, R. Govidan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of the ACM SIGCOMM 2011 Conference*. ACM, 2011, pp. 26–37.

[6] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2014, pp. 660–669.

[7] B. Guha and B. Mukherjee, "Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions," *IEEE Network*, vol. 11, no. 4, pp. 40–48, 1997.

[8] V. Kumar, P. Jayalekshmy, G. Patra, and R. Thangavelu, "On remote exploitation of TCP sender for low-rate flooding denial-of-service attack," *IEEE Communications Letters*, vol. 13, no. 1, pp. 46–48, 2009.

[9] A. Kuzmanovic and E. Knightly, "Low-rate TCP-targeted denial of service attacks and counter strategies," *IEEE/ACM Transactions on Networking*, vol. 14, no. 4, pp. 683–696, 2006.

[10] R. Morris, "A weakness in the 4.2 BSD unix TCP/IP software," AT&T Bell Leboratories, Tech. Rep., 1985.

[11] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, p. 71, Oct. 1999.

[12] J. Touch, "Defending TCP against spoofing attacks," RFC 4953 (Informational), Jul. 2007.

[13] P. Watson, "Slipping in the window: TCP reset attacks," CanSecWest, Tech. Rep., 2004. [Online]. Available: http://bandwidthco.com/whitepapers/netforensics/tcpip/TCPResetAttacks.pdf

[14] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmer, and G. Vigna, "SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr," in *Information Security Conference*, ser. Lecture Notes in Computer Science, S. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, Eds., vol. 4176. Springer, 2006, pp. 343–358.

[15] H. J. Abdelnur, R. State, and O. Festor, "KiF: A stateful SIP fuzzer," in *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, ser. IPTComm '07. ACM, 2007, pp. 47–56.

[16] J. Wang, T. Guo, P. Zhang, and Q. Xiao, "A model-based behavioral fuzzing approach for network service," in *Third International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC)*. IEEE, 2013, pp. 1129–1134.

[17] P. Tsankov, M. T. Dashti, and D. Basin, "SECFUZZ: Fuzz-testing security protocols," in *7th International Workshop on Automation of Software Test (AST)*, 2012, pp. 1–7.

[18] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *USENIX Security Symposium*. USENIX, 2011.

[19] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011, p. 265.

[20] Y. Wang, Z. Zhang, D. D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Proceedings of the 9th International Conference on Applied Cryptography and Network Security*, ser. ACNS'11. Springer-Verlag, Jun. 2011, pp. 1–18.

[21] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[22] J. Postel, "User datagram protocol," RFC 768 (Standard), Aug. 1980.

[23] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "TCP friendly rate control (TFRC): Protocol specification," RFC 5348 (Proposed Standard), Sep. 2008.

[24] J. Widmer and M. Handley, "TCP-friendly multicast congestion control (TFMCC): Protocol specification," RFC 4654 (Experimental), Aug. 2006.

[25] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, ser. NSDI'04. USENIX Association, 2004, pp. 155–168.

[26] J. Postel, "Transmission control protocol," RFC 793 (Standard), Sep. 1981.

[27] E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with *dot*," 2006. [Online]. Available: http://www.graphviz.org/Documentation/dotguide.pdf

[28] WebHosting Talk, "DOS attack – hosting security and technology," 2004. [Online]. Available: https://www.webhostingtalk.com/showthread.php?t=293069

[29] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," RFC 5681 (Draft Standard), p. 18, Sep. 2009.

[30] O. Andreasson, "TCP variables," 2002. [Online]. Available: https://www.frozentux.net/ipsysctl-tutorial/chunkyhtml/tcpvariables.html

[31] S. Floyd, M. Handley, and E. Kohler, "Datagram congestion control protocol (DCCP)," RFC 4340 (Proposed Standard), 2006.

[32] S. Floyd and E. Kohler, "Profile for datagram congestion control protocol (DCCP) congestion control ID 2: TCP-like congestion control," RFC 4341 (Proposed Standard), 2006.