Efficient Correctness Testing of Linux Network Stack under Packet Dynamics

Minh Vu, Phuong Ha, Lisong Xu
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA 68588-0115
Email: {mvu, pha, xu}@cse.unl.edu

***Abstract*—Network protocols are challenging to test for correctness due to the huge number of packet dynamics possibilities. Network simulators are popular in evaluating the performance of network protocols but unable to test the correctness under different packet dynamics efficiently. Random testing and symbolic execution are two effective automated correctness testing techniques that can explore different program execution possibilities. Random testing is simple and scalable in checking typical cases but often misses corner ones with low probabilities. Symbolic execution is more efficient in exploring these corner cases but suffers from the scalability problem. In this paper, we propose a testing platform built upon a network simulator by implementing a combination of symbolic execution and random testing to mitigate their limitations. Then we evaluate the efficiency of different techniques in testing Linux network stack under multiple possibilities of packet dynamics.**

***Index Terms*—Network protocol testing, symbolic execution, network simulator.**

I. Introduction

Testing network protocols for correctness is important because they are the core components of many services. Unfortunately, they are very difficult to test due to the unpredictable nature of the underlying network. Packets can arrive out of order or be lost during transmission. Packet transmission related parameters such as delay, jitter and loss can be considered as packet dynamics because these values are often different for each packet. Testing network protocols under all possible combinations of packet dynamics is essential because many TCP bugs [1] are only detected in corner cases with low probabilities.

Network simulators such as NS-3 [2] are widely used in studying network protocol performance due to the capability to simulate different cases of packet dynamics, topologies and links. However, these network simulators are inefficient in testing the correctness of network protocols under multiple packet dynamics possibilities because they do not implement automated correctness testing techniques.

In this paper, we consider the packet dynamics caused by network delay, which directly affects the behavior of network protocols. For example, TCP running on an end node adjusts the transmission rate based on the transmission time of previously received packets. In order to guarantee the correctness, we need to test all the possibilities of packet dynamics. Automated testing techniques are popular choices for this job because of their capability to explore different cases systematically. However, it is very challenging due to the extremely large number of packet dynamics possibilities.

Random testing and symbolic execution are two widely used automated testing techniques in the software testing community. Random testing is cost-effective at checking common cases, but it has trouble reaching branches with strict conditionals (e.g., $if(x == 500)$). On the contrary, symbolic execution can effectively explore these branches. The main idea is to execute the programs with symbolic variables that represent multiple values. However, symbolic execution does not scale well because its complexity is approximately an exponential function of the number of branching statements and symbolic variables. Network protocol testing systems based on symbolic execution suffer from similar limitations and are only capable of testing simple protocols [3], [4], [5] or limited packet dynamics [6].

To address these problems, we propose to combine symbolic execution with random testing to test network protocols. By doing so, we take advantage of the ability to explore low probability cases of symbolic execution while partially mitigate its scalability problem. The general idea of combining symbolic execution with random inputs is not new. The novelty of our work is that we apply it to the large and general network simulator NS-3, and evaluate its efficiency in testing network protocols under packet dynamics while current works [7], [8] focus on improving the performance of symbolic execution.

Our contributions: 1) We propose a testing platform based on NS-3 to test network protocols for correctness. To efficiently explore packet dynamics, we implement the combination of symbolic execution and random testing in our platform. To the extent of our knowledge, this paper is the first to apply both symbolic execution and random testing in a network simulator for correctness testing of real-world network stack under packet dynamics. 2) We present our comprehensive evaluation of different techniques on Linux TCP and Multipath TCP (MPTCP) [9] stack and investigate the trade-off between scalability and coverage for the combination technique. Using symbolic execution, we can test an equivalence class of inputs simultaneously, instead of just a single input. However, the number of equivalence classes is roughly an exponential function of the number of symbolic variables, and our experiments show that the coverage does not have a linear increase with the number of equivalence classes created in a simulation run. Therefore, it is more efficient to explore equivalence classes of a single symbolic execution at a time and merge the coverage results of different runs than to increase the number of symbolic variables in one run.

II. Background

A. Network simulation

Network protocols are usually tested using discrete-event network simulators. These simulators maintain a list of pending events, and each event has a *timestamp* value, which is its occurrence time. During the simulations, the system variables only change at the occurrence times of events. Instead of advancing the system clock by one time unit each step, the simulators directly jump to the execution time of the next event. For example, if the current event is completed at time 5 seconds and the next event is scheduled at 20 seconds, the simulators will set the clock to 20 and immediately process the next event. Note that the currently processing event must have the lowest timestamp value to preserve the causal relationship, which means future events cannot affect past events.

In this paper, we consider NS-3 [2], which is a discrete-event network simulator widely used in the networking community. To run the Linux network protocol stack, we use the DCE module [10].

B. Random testing

In random testing, a number of test inputs are generated randomly according to some distribution, and these inputs are used to run the program to check for correctness. Random testing generally can check common cases with a relatively small number of inputs compared to the total number of all possible inputs. The main problems are that it often misses corner cases with low probabilities, and many test inputs can be redundant because they lead to the same program behaviors. As an example, let us consider the conditional statement $if(x == 500)$ where $x \in [1, 10^6]$, random testing with a uniform distribution only has a probability of $1/10^6$ to pick the correct x value to satisfy this statement.

C. Symbolic execution

Symbolic execution [11] becomes practical in recent years due to the significant advancement of constraint solvers. Instead of executing programs with actual concrete data, this technique uses symbolic input values and represents the values of variables as symbolic expressions. Each symbolic expression is a collection of constraints for a variable along an execution path. Figure 1 shows a simple example of symbolic execution. Lines 1 and 2 initialize two symbolic variables t_1 and t_2 with their constraints. The constraints mean that the symbolic variables can be any integer value in their respective intervals. Once the execution reaches the branching if statement at line 3, it checks the condition $t_1 < t_2$. In this case, the condition can be either true or false. Thus the execution forks into two paths, and each path has an additional constraint. For example, path 1 adds $t_1 < t_2$ to its corresponding accumulative constraints.

Even though there are $1000 \times 2000 = 2 \times 10^6$ distinct combinations of t_1 and t_2, symbolic execution only generates two different paths, each path corresponds to multiple values of t_1 and t_2. We can also see that the number of satisfying values for each variable has no impact on the number of paths.

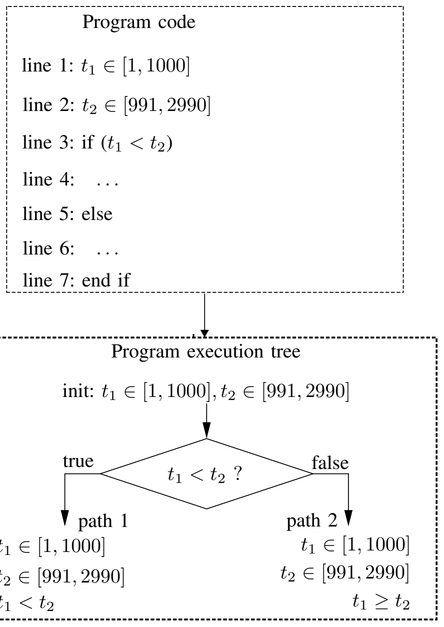

Fig. 1. A symbolic execution example.

In this example, the number of values of t_1 for the false branch is much lower than that for the true branch.

In this paper, we use S^{2}E [12], which is a powerful symbolic execution platform that can symbolically execute NS-3 in a virtual machine. The virtual machine is emulated using the QEMU machine emulator [13], and the symbolic execution is conducted using the KLEE symbolic execution engine [7].

III. Testing platform

A. Overview

In this paper, we focus on the packet delays, which are a major factor causing network protocols to change their behaviors. Protocols such as TCP and MPTCP [9] change their behavior based on the arrival time of packets. For example, the arrival time of a packet can change the estimated round-trip time (RTT), or an acknowledgment (ACK) packet arriving after a predetermined timeout can trigger a re-transmission of the corresponding data packet. In order to comprehensively test these protocols, we have to try all possible combinations of packet delays. Consider a series of n packets $\{p_0, p_1, ..., p_n\}$, each has an independent delay d_{p_i} in the range $[1, t]$. Each test case is a delay vector $\vec{d} = (d_{p_1}, d_{p_2}, ..., d_{p_n})$. Thus the total number of unique delay vectors or test cases is t^n. For just a simple experiment with three packets having the same delay interval $[1, 1000]$ ms, there would be $1000^3 = 10^9$ delay vectors.

With symbolic execution, we can group multiple delay vectors into an equivalence class. We define an equivalence class as a group of delay vectors that follows the same execution path. Because network simulators such as NS-3 are event-based, an execution path generally corresponds to a distinct sequence of events. Therefore, we only need to test one delay vector in each equivalence class, and the number of equivalence classes is usually several orders of magnitude lower than the total number of delay vectors.

Pseudocode 1 Three different types of delay values

```
 1: sorted list: $events[\,]$                         ▷ list of events
 2: variable: $packetDelay$                          ▷ concrete value
 3: variable: $maxDelay$                             ▷ delay interval
 4: function $TransmitStart(p, transmitTime)$
 5:     if $p.type == symbolic$ then
 6:         $delay = symbolic(1, maxDelay)$
 7:     else if $p.type == random$ then
 8:         $delay = rand() \;\%\; maxDelay + 1$
 9:     else
10:         $delay = packetDelay$
11:     $e = Event()$                                ▷ create a new event
12:     $e.timestamp = transmitTime + delay$
13:     $InsertEvent(e)$
14: function $InsertEvent(e)$
15:     for $i = 0$; $i < events.size$; $i = i + 1$ do
16:         if $e.timestamp < events[i].timestamp$ then
17:             Insert $e$ to position $i$
18:             return
19:     Append $e$ to the end of $events$
```

B. Implementation

Pseudocode 1 shows the process of injecting different types of delay values. We have three types of packets: random packets that have random delay values, symbolic packets that have symbolic delay values, and concrete packets that have traditional concrete delay values. The variable $packetDelay$ at line 2 specifies the concrete delay value, which could be different for different concrete packets in general. The variable $maxDelay$ at line 3 defines the interval of all delay values.

Function $TransmitStart$ sets delay values for each packet according to its type. The delay of each random packet is a random integer in the interval $[1, maxDelay]$ while the concrete packets have delays specified by $packetDelay$. For a symbolic packet, its delay is defined by the constraint $1 \leq delay \leq maxDelay$. That is, the delay is a symbolic variable is the interval $[1, maxDelay]$. Consequently, the arrival time of this symbolic packet $e.timestamp$ is a symbolic variable in the interval $[transmitTime + 1, transmitTime + delay]$. The arrival event e is then inserted into the global list of events sorted by timestamp in ascending order.

Function $InsertEvent$ compares the timestamp of event e with timestamps of other events in the list. We say that the timestamps of two events overlap if the intersection of their intervals is nonempty. This means two events may occur in different orders and comparing their overlapping timestamps at line 16 causes the execution to fork into two branches. Note that, the execution only forks when at least one timestamp is symbolic because each concrete or random delay contains exactly a single value.

C. Testing techniques

The testing platform[1] supports four testing techniques. Figure 2 shows an example of transmitting two packets using different techniques.

[1]The source code is available at https://github.com/minhvu2/packet-dynamics-platform

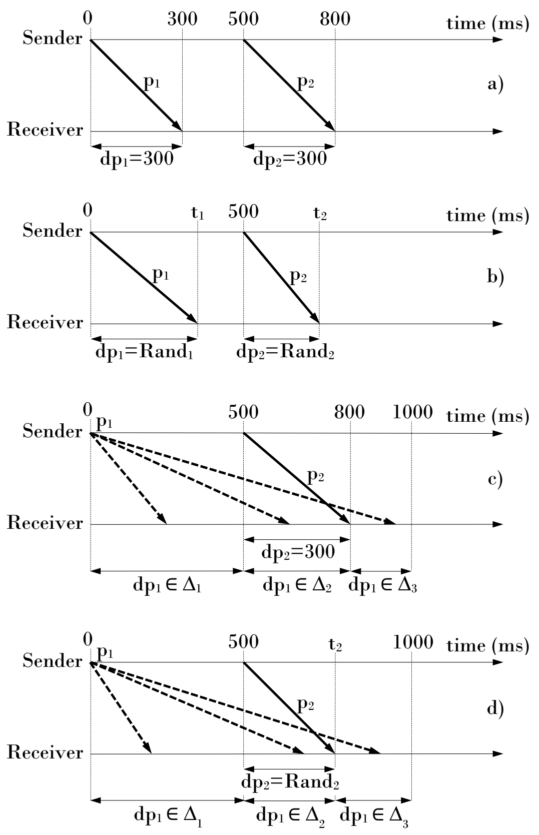

Fig. 2. Example of transmitting two packets. a) Original. b) Random. c) Symbolic execution. d) Symbolic execution plus random

1) Orginal: Figure 2a shows an example of sending two packets p_1, p_2 with $packetDelay = 300$ which is equivalent to the delay vector $\vec{d} = (300, 300)$. Both packets have the same delay in this example and in Pseudocode 1, but in general they may have different concrete delays. Each execution follows a sequence of events, and repeated executions output the same results unless we change the delay vector. There is also no guarantee that a new delay vector produces a new sequence of events because a sequence can correspond to multiple delay vectors. To test a protocol for correctness using this method, we need to run the simulator for all distinct delay vectors. If each packet has 1000 different delay values, there are 1000^2 delay vectors to test.

2) Random: Similar to Original, each execution corresponds to one delay vector. The main difference is that packets can have different random delay values in different simulator runs. With this method, we can often test common sequences of events with a much lower number of runs when compared to Original. The problem of Random is that delay vectors are not distributed evenly among equivalence classes. Thus, it cannot guarantee to cover all possible sequences even if we run a large number of simulations. For example, two packets p_1 and p_2 are sent at time 0 and 500 respectively as in Figure 2b. If both packet delays are random with $maxDelay = 1000$, the probability of p_1 arriving after p_2 is lower than the inverse case, assuming a uniform distribution.

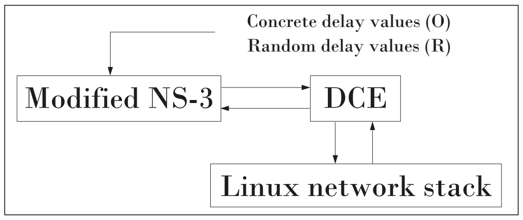

Fig. 3. System implementation for Original and Random

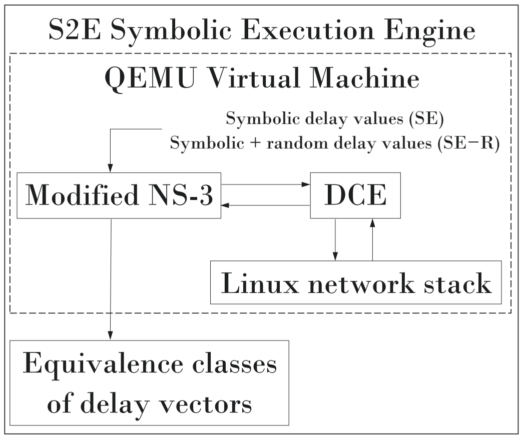
S2E Symbolic Execution Engine
QEMU Virtual Machine
Symbolic delay values (SE)
Symbolic + random delay values (SE−R)
Modified NS-3
DCE
Linux network stack
Equivalence classes of delay vectors

Fig. 4. System implementation for Symbolic Execution and Symbolic Execution plus Random

3) Symbolic Execution: A symbolic delay associates with a set of constraints, and it represents multiple delay values instead of a single delay value. Let's consider the example in Figure 2c where d_{p_1} is a symbolic delay in the interval $[1, maxDelay] = [1, 1000]$. Because p_1 departs at time 0, the timestamp interval of p_1 arrival is $[1, 1000]$ which overlaps with the departure and arrival of p_2 occurring at time 500 and 800. Thus, there are three set of constraints $\{1 \leq d_{p_1} \leq 500\}$, $\{501 \leq d_{p_1} \leq 800\}$ and $\{801 \leq d_{p_1} \leq 1000\}$, which define three sub-intervals $\Delta_1 = [1, 500]$, $\Delta_2 = [501, 800]$ and $\Delta_3 = [801, 1000]$, respectively. As a result, three equivalence classes corresponding to three distinct sequences of events are generated. Non-symbolic packets (if any) have the specified concrete delays.

4) Symbolic Execution plus Random: The only difference between this technique and Symbolic Execution is that non-symbolic packets can have random delay values instead of concrete ones in different simulation runs. Note that, Figure 2d only represents a single random value of d_{p_2}. During symbolic execution, the interval Δ_2 and Δ_3 can vary among different execution paths.

Figure 3 illustrates the implementation of Original and Random. NS-3 communicates with the Linux network protocol stack through the DCE module. For these two techniques, each execution of the simulator covers only one delay vector.

With Symbolic Execution and Symbolic Execution plus Random techniques, we set selected packets to be symbolic and execute NS-3 and DCE in a QEMU virtual machine using the S^{2}E symbolic execution engine, as shown in Figure 4. For these two techniques, each execution of the simulator covers multiple equivalence classes, and each equivalence class covers multiple delay vectors corresponding to a same sequence of events.

IV. Evaluation

A. Simulation setup

We demonstrate and compare the following testing techniques: Original (referred to as *O*), Random (referred to as *R*), Symbolic Execution (referred to as *SE*), Symbolic Execution plus Random (referred to as *SE - R*). For O, $packetDelay$ is specified in the simulation scripts. For SE and SE - R, we use n to indicate the number of symbolic packets and set $maxDelay = 16384 \times 10^6$ ns.

In this paper, we use code coverage to evaluate the efficiency of different techniques. Code coverage is a popular metric in software engineering community to show which parts of the source code of a program under test are executed. In particular, we measure the line, function and branch coverage on the TCP and MPTCP implementations in the Linux kernel.

We choose TCP because it is responsible for the majority portion of Internet traffic. Popular applications such as email, web browsing, and video streaming are transported over TCP. Recently, MPTCP [9] has been proposed to improve the performance of TCP and implemented in Linux kernel [14] and Apple Mac OS.

For O and R, we repeatedly run the simulator for different delay vectors. For SE and SE - R, we first execute the simulator symbolically and generate one delay vector for each equivalence class. These delay vectors are then used to run the simulator to collect the coverage.

We evaluate different techniques with respect to the following questions:

1) What is the scalability of SE in exploring packet dynamics?
2) Does SE - R improve code coverage when compared to SE and R running separately?
3) What is the most effective way to apply SE - R for different types of network protocols?

The experiments run on machines configured with a 2.3GHz 4-Core processor, 64GByte of RAM and Ubuntu 16.04. We run the simulation scripts configured with TCP and MPTCP. These simulation scripts are selected from examples provided with the DCE module.

B. MPTCP experiments

This group of experiments uses the dce-cradle-mptcp.cc script of the DCE module. There are two nodes communicating through a MPTCP connection as shown in Figure 5. There are two paths between the two nodes corresponding to two TCP subflows. About 50 packets are transmitted during a simulation.

1) Scalability evaluation: Table I shows the testing time and numbers of delay vectors using O and SE, where a selected packet has a delay in the range of $[1, 16380 \times 10^6]$ and all other packets have a concrete delay (i.e., a total of 16380×10^6 delay vectors to test). When using SE, there is one symbolic packet (i.e., the selected packet) and all other packets are concrete

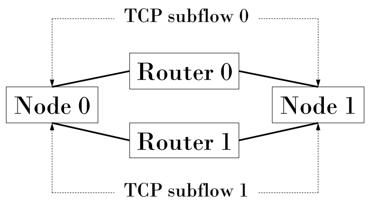

Fig. 5. Network topology of the MPTCP experiments

TABLE I
NUMBER OF DELAY VECTORS AND TESTING TIME OF DIFFERENT TECHNIQUES

	Number of delay vectors	Testing time
O	16384×10^6	17747 years
R	156	1.48 hours
R	1029	9.76 hours
SE (n=1)	156 equivalence classes	2.25 hours
SE - R (n=1)	1029 equivalence classes	12.82 hours

packets, and we run the simulator with SE only once. When using O, all packets are concrete packets, and we need to repeatedly run the simulator for 16380×10^6 times, each time with a different concrete delay for the selected packet. As a result, O takes about 21248 years. In contrast, SE finishes the test in just 2.55 hours, and reports 156 equivalence classes of delay vectors. This result clearly shows the effectiveness of SE in testing network protocols.

While R is faster than SE and SE - R when running with the same numbers of delay vectors, multiple delay vectors in R can belong to an equivalence class because they are chosen randomly. SE - R outputs a higher number of equivalence classes than SE due to the fact that the set of constraints for each equivalence class must be unique. The delay for a packet remains the same for all paths in SE while it varies among different paths in SE - R. Thus, additional feasible branches are forked, leading to more execution paths.

However, symbolic execution does not scale well as we increase the number of symbolic packets. We run the simulator and introduce symbolic delay values for $n = 2$ packets. In this case, the execution is unable to finish within 110 hours for just two symbolic packets, as summarized in Table II. Moreover, the code coverage does not increase linearly with the number of delay vectors given the limited testing time. As we increase the number of symbolic packets, the code coverage of different delay vectors overlaps with each other. Thus, instead of increasing the number of symbolic delays in one run, we can execute the simulator multiple times, each time only one packet has symbolic delay. In each of these runs, we can choose a new type of packet to have symbolic delay.

TABLE II
TESTING TIME AND CODE COVERAGE FOR SE - R

Number of symbolic packets	$n = 1$	$n = 2$
Testing time in hour	14.75	stop at 110
Number of delay vectors	1029	6627
Line coverage for MPTCP	100 %	82.0 %
Function coverage for MPTCP	100 %	87.8 %
Branch coverage for MPTCP	100 %	81.0 %

TABLE III
MPTCP TESTING SCENARIOS FOR PARTS UNAFFECTED BY DELAYS

	Testing scenarios
mptcp_fullmesh.c	1) More routers and paths.
mptcp_input.c	1) Unable to establish MPTCP connection and falling back to TCP. 2) Fast open option.
mptcp_output.c	1) Paths with highly different capacities. 2) Keep alive connection option.
All files	1) Removing and adding subflows.

TABLE IV
LINE COVERAGE FOR THE MPTCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
mptcp_binder.c	100 %	100 %	100 %	100 %
mptcp_ctrl.c	5.0 %	80.7 %	83.0 %	100 %
mptcp_fullmesh.c	56.0 %	89.3 %	89.3 %	100 %
mptcp_input.c	0.0 %	80.8 %	81.0 %	100 %
mptcp_ipv4.c	12.8 %	36.4 %	36.4 %	100 %
mptcp_ndiffports.c	100 %	100 %	100 %	100 %
mptcp_ofo_queue.c	0.0 %	6.7 %	6.7 %	100 %
mptcp_output.c	0.0 %	91.2 %	90.2 %	100 %
mptcp_pm.c	72.2 %	100 %	100 %	100 %
mptcp_sched.c	28.0 %	100 %	98.8 %	100 %

Takeaway: *(Results for Question 1)* The increased testing time of using SE is acceptable when compared to R running the simulator for the same number of times as the number of equivalence classes. Increasing the number of symbolic packets may produce lower coverage due to the fact that the simulation is unable to finish within the allotted time, and the code covered by different delay vectors becomes overlapped. In cases where one symbolic packet is insufficient, it is more efficient to repeat the simulation with different symbolic packets than to introduce additional symbolic packets.

2) Efficiency evaluation: We have thoroughly analyzed the source code of MPTCP to the best of our understanding, and identified all possible parts of the MPTCP code that can be covered by the dce-cradle-mptcp.cc script with dynamic packet delays. Below, we only report the coverage for these parts. Table III lists further testing scenarios in order to cover the other parts which are unrelated to packet delays.

Tables IV, V and VI show the line, function and branch coverage, respectively, on the MPTCP implementation by different techniques. For O, we run the simulator only once with the original dce-cradle-mptcp.cc script. For R, we repeatedly run the simulator for 1029 times, to be consistent with SE ($n = 1$) that reports 1029 equivalence classes. The symbolic packet in SE (n=1) and SE-R (n=1) is the SYN-ACK packet

TABLE V
FUNCTION COVERAGE FOR THE MPTCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
mptcp_binder.c	100 %	100 %	100 %	100 %
mptcp_ctrl.c	4.3 %	87.3 %	89.4 %	100 %
mptcp_fullmesh.c	61.2 %	96.1 %	96.1 %	100 %
mptcp_input.c	0.0 %	86.9 %	91.2 %	100 %
mptcp_ipv4.c	10.0 %	50.0 %	50.0 %	100 %
mptcp_ndiffports.c	100 %	100 %	100 %	100 %
mptcp_ofo_queue.c	0.0 %	16.7 %	16.7 %	100 %
mptcp_output.c	0.0 %	95.4 %	95.4 %	100 %
mptcp_pm.c	66.7 %	100 %	100 %	100 %
mptcp_sched.c	28.5 %	100 %	100 %	100 %

TABLE VI
BRANCH COVERAGE FOR THE MPTCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
mptcp_binder.c	100 %	100 %	100 %	100 %
mptcp_ctrl.c	7.7 %	75.4 %	80.0 %	100 %
mptcp_fullmesh.c	56.4 %	83.5 %	83.5 %	100 %
mptcp_input.c	0.0 %	76.1 %	74.5 %	100 %
mptcp_ipv4.c	7.0 %	21.5 %	21.5 %	100 %
mptcp_ndiffports.c	100 %	100 %	100 %	100 %
mptcp_ofo_queue.c	0.0 %	4.5 %	4.5 %	100 %
mptcp_output.c	0.0 %	93.8 %	88.8 %	100 %
mptcp_pm.c	80.0 %	100 %	100 %	100 %
mptcp_sched.c	17.1 %	100 %	89.5 %	100 %

TABLE VII
COVERAGE FOR THE MPTCP IMPLEMENTATION USING SE-R (N=1) WITH DIFFERENT SYMBOLIC PACKETS

	SYN-ACK	Data	FIN-ACK	Combined cov
Line	100 %	82.0 %	81.4 %	100 %
Function	100 %	87.8 %	87.8 %	100 %
Branch	100 %	81.0 %	78.2 %	100 %

of the connection establishment that is sent through TCP subflow 0. We can see that SE-R achieves the highest coverage (actually perfect 100% coverage) across all experiments. We can also see that the coverage improves from O to SE-R for several files: mptcp_ipv4.c that initializes the connection, files mptcp_ctrl.c, mptcp_input.c and mptcp_output.c that deal with the transmission and receipt of packets, file mptcp_ofo_queue.c that handles packet reordering, and file mptcp_sched.c that manages congestion control. This is because packet delays directly affect these tasks, and thus these files see significant improvement from O to other techniques.

To see how the technique SE-R performs with different symbolic packets, we set various types of packets to be symbolic, as shown in table VII. We can see that SYN-ACK symbolic packet achieves the highest coverage, which is expected because of two reasons. First, MPTCP essentially consists of multiple TCP subflows, and MPTCP is mainly responsible for the connection establishment and management. Handling the transmission and receipt of packets is the task of TCP. Second, the symbolic delay of a packet can affect the timestamps of subsequent events that are triggered by the arrival of the packet. That is, these timestamps also become symbolic variables. In this group of experiments, we can see that the coverage of the i^{th} symbolic packet is a subset of that of the $(i+1)^{th}$ symbolic packet due to the impact of symbolic delay on subsequent events.

Takeaway: *(Results for Question 2)* SE-R reports the highest coverage based on our experiment results. For MPTCP, one simulator run in which the SYN-ACK packet of the connection establishment phase is set to symbolic is sufficient to cover all possible behaviors. It is unnecessary to repeatedly running the simulator with other packets because the symbolic delay of SYN-ACK packet can propagate to later packets.

C. TCP experiments

This group of experiments test Linux TCP code using the dce-cradle-simple.cc script of the DCE module which contains two nodes connecting over a wired link. Table VIII, IX and X show the line, function and branch coverage for TCP implementation by different techniques. Again we have thoroughly analyzed the source of TCP and identified all possible parts of the TCP code that can be covered by the dce-cradle-simple.cc script with dynamic packet delays. The coverage is calculated for only these parts. In addition, we exclude file tcp_fastopen.c because NS-3 and DCE do not support fast open mechanism. Files that implement different congestion control algorithms such as tcp_bic.c, tcp_cubic.c, tcp_highspeed.c, and tcp_dctcp.c are also excluded because a TCP connection can only use one congestion control algorithm. We can see that SE-R still gets the highest coverage.

TABLE VIII
LINE COVERAGE FOR THE TCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
tcp.c	47.6 %	47.6 %	98.3 %	98.3 %
tcp_cong.c	67.4 %	67.4 %	85.3 %	85.3 %
tcp_input.c	37.0 %	53.2 %	83.3 %	91.8 %
tcp_ipv4.c	62.3 %	64.0 %	83.4 %	100 %
tcp_minisocks.c	66.1 %	71.2 %	92.1 %	99.1 %
tcp_output.c	38.5 %	43.1 %	94.6 %	96.9 %
tcp_timer.c	27.2 %	31.2 %	45.3 %	95.1 %

TABLE IX
FUNCTION COVERAGE FOR THE TCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
tcp.c	33.3 %	33.3 %	100 %	100 %
tcp_cong.c	63.6 %	63.6 %	91.0 %	91.0 %
tcp_input.c	42.4 %	62.6 %	87.9 %	96.0 %
tcp_ipv4.c	83.9 %	83.9 %	91.9 %	100 %
tcp_minisocks.c	70.0 %	70.0 %	90.0 %	100 %
tcp_output.c	37.0 %	38.9 %	96.3 %	98.2 %
tcp_timer.c	33.3 %	44.4 %	55.6 %	100 %

Table XI shows the line coverage for different symbolic packets for SE-R. We only set one packet to be symbolic in each run and also report their combined coverage. In contrast to MPTCP, we can see that there are improvements in the combined coverage because TCP handles the transmission and receipt of packets on its own. Even though the symbolic delay can affect subsequent events, it is still bounded by the existing constraints which can limit the interval of these subsequent events. Thus, there can be new sequence of events when setting different packets to be symbolic.

Takeaway: *(Results for Question 3)* SE-R still achieves the best coverage. Contrary to MPTCP, TCP requires several simulator runs with different symbolic packets to cover all cases. The difference is that TCP manages the whole process, including sending and receiving packets, which are directly affected by the delays. While the symbolic delay propagates to subsequent packets, its accumulative constraints can prevent valid execution paths, leading to some undiscovered equiva-

TABLE X
BRANCH COVERAGE FOR THE TCP IMPLEMENTATION

	O	R	SE (n=1)	SE - R (n=1)
tcp.c	43.9 %	43.9 %	97.3 %	97.3 %
tcp_cong.c	71.3 %	71.3 %	81.1 %	81.1 %
tcp_input.c	27.3 %	45.6 %	72.4 %	85.7 %
tcp_ipv4.c	63.8 %	68.1 %	85.6 %	100 %
tcp_minisocks.c	51.8 %	66.8 %	79.1 %	96.5 %
tcp_output.c	26.6 %	33.6 %	88.8 %	94.4 %
tcp_timer.c	17.4 %	18.8 %	28.5 %	91.0 %

TABLE XI
LINE COVERAGE FOR THE TCP IMPLEMENTATION USING SE-R (N=1) WITH DIFFERENT SYMBOLIC PACKETS

	SYN-ACK	Data	FIN-ACK	Combined cov
tcp.c	98.3 %	98.3 %	97.2 %	100 %
tcp_cong.c	85.3 %	85.3 %	100 %	100 %
tcp_input.c	91.8 %	98.9 %	76.4 %	100 %
tcp_ipv4.c	100 %	99.2 %	83.9 %	100 %
tcp_minisocks.c	99.1 %	94.9 %	96.7 %	100 %
tcp_output.c	96.9 %	99.8 %	89.0 %	100 %
tcp_timer.c	95.1 %	97.1 %	73.6 %	100 %

lence classes.

D. Limitations

One limitation of our platform is that manual effort is required to analyze the code. Another limitation is that we only consider the packet delay in our study. It can potentially limit parts of network protocols that can be tested. In future work, we plan to add support for more types of packet dynamics.

V. RELATED WORK

Fuzzing is a type of random testing that provides randomly mutating inputs to programs. SAGE [8], QSYM [15] and Driller [16] follow the hybrid approach by using symbolic execution to guide the fuzzers to alternate paths. Their aim is to improve the performance of fuzzing whereas we evaluate the effectiveness of the combination approach in testing network protocols.

Concolic testing combines concrete and symbolic execution to overcome the limitations of symbolic execution. This technique has been implemented in various tools such as KLEE [7], Dart [17], CUTE [18] and CREST [19] to find vulnerabilities in software. In this technique, the execution starts with randomly generated concrete inputs, collects symbolic constraints along the execution path, and then uses constraint solvers to generate inputs to drive the next execution to unexplored parts of the code. Hybrid concolic testing [20] interleaves random testing and symbolic execution. The technique starts with random testing until no new coverage point is discovered. Then it switches to symbolic execution to find uncovered parts of the code. When one is found, the execution reverts to random. Selective symbolic execution [21] exploits the fact that parts of the code do not need to be executed symbolically such as system calls or calls to external libraries. MergePoint [22] performs dynamic path merging to mitigate the path explosion problem. These techniques focus on improving the performance of symbolic execution engine. On the contrary, our testing platform is the first attempt to introduce both symbolic execution and random testing into the popular network simulator NS-3 in order to efficiently test different packet dynamics.

VI. CONCLUSIONS

In this paper, we present our testing platform which implements both symbolic execution and random testing technique. Our preliminary experiments show the efficiency of our platform in testing real-world network stack under multiple possible packet dynamics.

ACKNOWLEDGMENT

The work presented in this paper was supported in part by NSF CNS-1526253 and CCF-1918204.

REFERENCES

[1] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert, “PacketDrill: Scriptable network stack testing, from sockets to packets,” in *Proceedings of USENIX ATC*, San Jose, CA, June 2013.

[2] Network Simulator 3, https://www.nsnam.org/.

[3] W. Sun, L. Xu, and S. Elbaum, “SPD: Automatically test unmodified network programs with symbolic packet dynamics,” in *Proceedings of IEEE Globecom*, San Diego, CA, December 2015.

[4] ——, “Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics,” in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Santa Barbara, CA, July 2017.

[5] M. Vu, L. Xu, S. Elbaum, W. Sun, and K. Qiao, “Efficient systematic testing of network protocols with temporal uncertain events,” in *Proceedings of IEEE INFOCOM*, Paris, France, April 2019, pp. 604–612.

[6] J. Song, C. Cadar, and P. Pietzuch, “SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, July 2014.

[7] C. Cadar, D. Dunbar, and D. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of USENIX OSDI*, San Diego, CA, December 2008.

[8] P. Godefroid, M. Levin, and D. Molnar, “Automated whitebox fuzz testing,” in *Proceedings of Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2008.

[9] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, “TCP extensions for multipath operation with multiple addresses,” *RFC 6824*, January 2013.

[10] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous, “Direct code execution: revisiting library OS architecture for reproducible network experiments,” in *Proceedings of ACM CoNEXT*, Santa Barbara, CA, December 2013.

[11] J. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[12] V. Chipounov, V. Kuznetsov, and G. Candea, “The S2E platform: design, implementation, and applications,” *ACM Transactions on Computer Systems*, vol. 30, no. 1, February 2012.

[13] F. Bellard, “Qemu, a fast and portable dynamic translator.” in *USENIX Annual Technical Conference, FREENIX Track*, Anaheim, CA, April 2005, pp. 41–46.

[14] IP Networking Lab, UCL, “Linux kernel implementation of MultiPath TCP,” http://multipath-tcp.org/.

[15] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “Qsym: A practical concolic execution engine tailored for hybrid fuzzing,” in *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, August 2018, pp. 745–761.

[16] N. Stephens, J. Grosen, C. Salls, and A. D. et al., “Driller: augmenting fuzzing through selective symbolic execution,” in *Proceedings of NDSS*, San Diego, CA, February 2016.

[17] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of ACM Programming Language Design and Implementation*, Chicagi, IL, June 2005.

[18] K. Sen, D. Marinov, and G. Agha, “CUTE: a concolic unit testing engine for C,” in *Proceedings of European Software Engineering Conference held jointly with ACM Symposium on Foundations of Software Engineering*, Paris, France, February 2005.

[19] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *Proceedings of IEEE/ACM Conference on Automated Software Engineering*, L’Aquaila, Italy, September 2008.

[20] R. Majumdar and K. Sen, “Hybrid concolic testing,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*, Minneapolis, MN, May 2007, pp. 416–426.

[21] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, “Selective symbolic execution,” in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, no. DSLAB-CONF-2009-002, 2009.

[22] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley, “Enhancing symbolic execution with veritesting,” in *Proceedings of International Conference on Software Engineering*, Hyderabad, India, June 2014.