Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics

Authors

ABSTRACT

The majority of Internet traffic is transferred by transport protocols. The correctness of these transport protocol implementations is hard to validate as their behaviors depend not only on their protocols but also on their network environments that can introduce dynamic packet delay and loss. Random testing, widely used in industry due to its simplicity and low cost, struggles to detect packet delay related faults which occur with low probability. Symbolic execution based testing is promising at detecting such low probability faults, but it requires large testing budgets as it attempts to cover a prohibitively large input space of packet dynamics. To improve its cost-effectiveness, we propose two domain-specific heuristic techniques, called packet retransmission based priority and network state based priority, which are motivated by two common transport protocol properties. In our experiments using the Linux TFTP programs, our techniques improve the cost-effectiveness of symbolic execution based testing for transport protocols, detecting three times as many faults when the budget is in the range of minutes and hours.

CCS CONCEPTS

•Networks →Protocol testing and verification; •Software and its engineering →Dynamic analysis;

KEYWORDS

Network protocol validation, testing packet dynamics, automated testing, symbolic execution

ACM Reference format:

Authors. 2016. Improving the cost-effectiveness of symbolic testing techniques

for transport protocol implementations under packet dynamics. In *Proceedings of ACM International Symposium of Software Testing and Analysis, Santa Barbara, CA, USA, July 2017 (ISSTA2017),* 12 pages.

DOI: 10.475/123_4

1 INTRODUCTION

The majority of Internet traffic is transferred by transport protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Trivial File Transfer Protocol (TFTP), and Real-Time Transport Protocol (RTP). Validating the implementation of such transport protocols is difficult because their behaviors depend not only on the protocol specifications but also on the network environment that can include different packet dynamics [41] such as packet delay, reordering, loss, and duplication. The prohibitively large number of packet dynamics possibilities contributes to many latent faults in popular implementations of these protocols [13].

As an example, let's consider a simplified scenario trying to explore the packet delay dynamics with tests consisting of only two packets, each with a delay range $(0, \infty)$. The input space is then a 2-dimensional vector space $(\triangle t_0, \triangle t_1)$, where $\triangle t_i \in (0, \infty)$ is the delay of packet i. If each delay is described by a 16-bit unsigned integer, there are $65,535 * 65,535 = 4,294,836,225$ potential test inputs $(\triangle t_0, \triangle t_1)$.

Current automated testing techniques to explore packet dynamics range from random input generation to the most systematic approaches. Random techniques are the most ubiquitous in industry due to their low-cost and they mostly vary based on whether they are driven by predefined distributions [39, 41] or by cross traffic [22, 56]. More exhaustive techniques such as those based on model-checking have also been used. For example, CMC [40], a C model checker, was used to test the Linux kernel TCP under packet dynamics, and Java Pathfinder [44], a Java model checker, has been extended to test UDP programs under packet dynamics. These techniques do not scale well, but are effective at finding faults when bounded. More recently, supported by symbolic execution engines such as S^2E [14] and KLEE [10], symbolic execution based testing techniques for the network domain have emerged such as SymTime [18] and SPD [55]. These techniques attempt to group test inputs into equivalence classes based on the paths they force the program to take. The idea is that by selecting only one input per path one can reduce the exploration cost while retaining the space explored [11]. This approach is illustrated in Figure 1 where constraints on each program path define the input classes for $(\triangle t_0, \triangle t_1)$. Such symbolic techniques' cost-effectiveness falls somewhere in between random and exhaustive.

In this work we argue that, in the context of *packet dynamics*, symbolic execution based testing could be more cost-effective by leveraging transport protocol properties. That is, the space of packet dynamics can be reduced in lieu of the properties of these transport protocols. Then, we propose two domain-specific heuristics, called packet retransmission based priority and network state based priority, aimed at improving the cost-effectiveness of symbolic execution based testing:

1) Packet retransmission based priority. Packet retransmission is a common component of many transport protocols. We find that many tests differing in just their packet retransmission patterns have the same or repetitive protocol behaviors. Thus, we propose to prioritize the generated tests before execution based on their packet retransmission patterns to increase the exposure of different protocol behaviors within a limited testing budget.

2) Network state based priority. Many transport protocols work as an event-driven finite state machine. In this machine, the future behavior of a protocol on a test input depends on the current network state. We find that many test inputs with initially different

ISSTA2017, Santa Barbara, CA, USA

DOI: 10.475/123_4

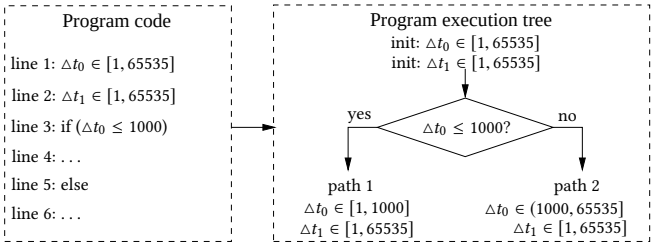


Figure 1: Symbolic execution finds equivalence classes of symbolic variables leading to the same path.

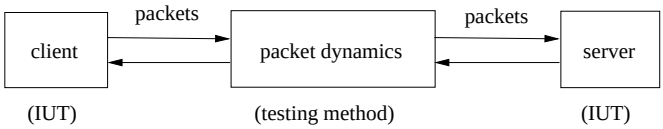


Figure 2: TFTP IUT and packet dynamics

protocol behaviors reach the same network state and subsequently have the same protocol behavior. Therefore, we propose to prioritize the executions of the test inputs based on their inferred network states, delaying the execution of tests that reach a known network state to increase the exposure of different protocol behaviors within a limited testing budget.

The contributions of this paper are as follows. First, it introduces two domain-specific heuristic techniques, packet retransmission based priority and network state based priority, to improve the cost-effectiveness of symbolic execution based testing. Second, it provides an assessment of the cost-effectiveness of applying the proposed heuristics. The experiments involve hundreds of mutants and also real faults for the Linux TFTP, last 700 CPU days, and show that the techniques improve the cost-effectiveness of symbolic execution based testing by about three times when the testing budget is in the range of minutes and hours. Our techniques also find a new fault on the Linux TFTP.

2 BACKGROUND ON TFTP

To illustrate and assess the proposed techniques we selected the widely used Trivial File Transfer Protocol (TFTP) [7, 51]. There are two reasons for choosing TFTP to showcase the proposed work. First, TFTP is simpler than other transport protocols but it still contains all the essential transport protocol components, such as connection establishment and termination, timeout mechanism and packet retransmission. For example, an entity (client or server) associates each of its transmitted packets with a timer. If it does not receive the corresponding packet from the other entity within the timeout period, a timeout occurs and then it retransmits the same packet until it reaches the maximum number of allowed retransmissions. Second, its implementation is already difficult to test using the current testing methods, as we will show in Section 5.

TFTP consists of two entities: the client and server, as illustrated in Fig. 2. The client and server programs together are the Implementation Under Test (IUT), and they follow the TFTP specification to communicate with each other using network packets. These network packets experience packet dynamics, which are defined by the input space and whose exposure is controlled by a testing method. TFTP works on top of unreliable UDP, and uses the stop-and-wait method to reliably transfer a file.

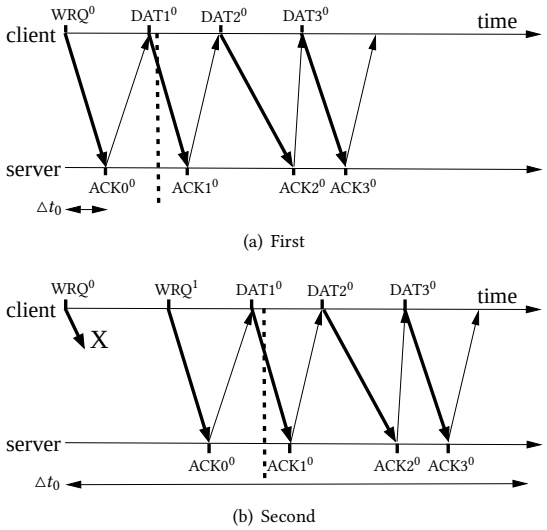


Figure 3: The TFTP client uploads a file with 3 data packets to the server. X: a lost packet. The dotted lines are explained in Section 4.2.

Fig. 3 illustrates how TFTP works, where a file with the size of three data packets is uploaded from the client to the server. Starting with Fig. 3(a), the client first sends a Write Request (WRQ) packet to establish the connection with the server, and then sends three data packets to the server. The first two data packets (DAT1 and DAT2) each contains exactly 512 bytes of the file data, and the third data packet (DAT3) contains less than 512 bytes of data and indicates that this is the final data packet. For each received client packet, the server sends back an acknowledgment (ACK), and the sequence number of the ACK indicates the sequence number of the received client packet. For example, the sequence number of WRQ is 0, and thus the sequence number of the corresponding ACK0 is 0. Superscript 0 indicates an original packet, and superscript i indicates the i-th retransmitted packet. Fig. 3(b) presents an alternative scenario where the original WRQ^0 is lost (label "X" indicates that $\triangle t_0$ is very long (e.g., T_∞) so that WRQ^0 is considered lost) and thus retransmitted (WRQ^1 is the first retransmitted WRQ packet). Note that each packet experiences packet dynamics. For example, the delay of the first packet (i.e., WRQ^0) is $\triangle t_0$.

3 PROTOCOLS' INPUTS AND OUTPUTS

In this section we identify the inputs and outputs that we will use as part of our tests. We also discuss how to assess the correctness of a protocol. Last, we briefly describe the symbolic testing platform on which we will later incorporate our techniques.

3.1 Space of Packet Dynamics

In this work we consider three types of packet dynamics: packet delay, loss, and reordering. We use a variable packet delay model to describe packet dynamics: packet i can have any positive delay $\triangle t_i \in (0, T_\infty]$, where $\triangle t_i$ is the time interval from the packet departure time at its source to the packet arrival time at its destination, and T_∞ is the maximum packet delay in the test and should be longer than the total running time of the tested transport protocol.

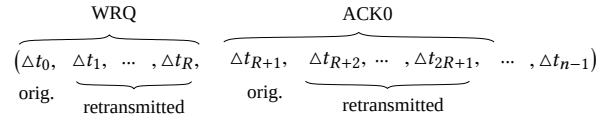


Figure 4: A point in the TFTP input space is an $n = P(R+1)$ dimensional delay vector.

This variable packet delay model requires only one parameter $\triangle t_i$ to describe packet i. Yet it is powerful, as it can also describe packet loss and reordering. For example, if $\triangle t_i$ of packet i is sufficiently long so that it arrives at its destination after the connection termination (e.g., T_∞), packet i has the same effect on the protocol as a lost packet. As an example of packet reordering, let's consider two packets: packet i is sent first, and then packet $i+1$ is sent next from the same source to the same destination. If $\triangle t_i$ is sufficiently longer than $\triangle t_{i+1}$, then packet $i+1$ will arrive at the destination earlier than packet i.

With the above packet dynamics model, *a test input* is a point $\overrightarrow{\triangle t}$ in the input space, which is an n-dimensional delay vector $\overrightarrow{\triangle t}$= $(\triangle t_0, \triangle t_1, ..., \triangle t_{n-1})$, where n is the total number of packets and vector element $\triangle t_i \in (0, T_\infty]$ is the delay of packet $i \in [0, n-1]$.

Each point in the TFTP input space model is an $n = P(R+1)$ dimensional delay vector as shown in Fig. 4, where P is the maximum number of original packets in a test. For example, there are $P = 8$ original packets in Fig. 3 (including the four ACKs). R is the maximum number of allowed retransmissions for each original packet in case of timeout. For example, delay $\triangle t_{R+1}$ is for ACK0^0, and delays $\triangle t_{R+2}$, ..., $\triangle t_{2R+1}$ are for the R retransmitted ACK0^1, ..., ACK0^R, respectively, as shown in Fig. 4. Thus, there are a total of at most $P(R+1)$ packets, which include P original packets and $P \times R$ retransmitted packets.

3.2 Protocol Behavior

To characterize a protocol implementation we consider three types of events generated by the client and server: 1) *Packet events*: a packet departs from its source (called a packet departure event), and a packet arrives at its destination (called a packet arrival event). 2) *Timer events*: a timer is expired (called a timeout event), and a timeout period is changed (called a timeout period change event). 3) *Program events*: the client program exits (called a client exit event), and the server program exits (called a server exit event).

In the case of TFTP, the IUT execution on a test input $\overrightarrow{\triangle t}$ starts from the WRQ^0 departure event at the client. The occurrence of this event generates a WRQ^0 arrival event at the server that will occur after $\triangle t_0$, and generates a timeout event at the client which will either occur after the timeout period or be disabled if ACK0 is received by the client before the timeout period. The occurrence of the next earliest event may generate one or multiple new events. The execution continues, until both the client and server programs exit (i.e., client and server exit events).

For each test input, there is then a corresponding sequence of IUT events starting from the WRQ^0 departure event at the client until the client and server program exit events, and this sequence of IUT events defines the IUT behavior. Note that this sequence is not just a sequence of packets, but also includes a varying number of packet, timer, and program events, all of which depend on test input $\overrightarrow{\triangle t}$.

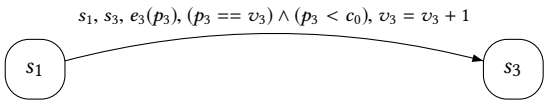


Figure 5: A small portion of the TFTP EEFSM

3.3 Assessing Behavior Correctness

To check the correctness of an IUT sequence we encode the transport protocol specification in an event-driven extended finite state machine (EEFSM) as proposed by Lee *et al.* [30]. An EEFSM is an 8-tuple $\langle S, s_0, s_a, s_r, \overrightarrow{\mathbf{v}}, \Sigma, \overrightarrow{\mathbf{p}}, \mathbb{T} \rangle$, where S=$\{s_0, s_1, \ldots, s_a, s_r\}$ is a finite set of protocol states with state s_0 as the initial state, state s_a as the accepting state, and state s_r as the rejecting state, vector $\overrightarrow{\mathbf{v}}$=$(v_0, v_1, \ldots)$ consists of a finite number of EEFSM variables, Σ=$\{e_0(\overrightarrow{\mathbf{p}}), e_1(\overrightarrow{\mathbf{p}}), \ldots\}$ is a finite set of IUT events, vector $\overrightarrow{\mathbf{p}}$=$(p_0, p_1, \ldots)$ consists of a finite number of event parameters, and $\mathbb{T}$ is a finite set of transitions. A transition $\langle s, s', e(\overrightarrow{\mathbf{p}}), predicate(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{p}}), assignment(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{p}}) \rangle \in \mathbb{T}$ means that if IUT event $e(\overrightarrow{\mathbf{p}})$ happens and $predicate(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{p}})$ is true, the EEFSM moves from the current state s to the next state s' and then updates the EEFSM variables by assignment $\overrightarrow{\mathbf{v}} := assignment(\overrightarrow{\mathbf{v}}, \overrightarrow{\mathbf{p}})$.

Fig. 5 shows a small portion of an EEFSM for the TFTP specifications [7, 51]. It has two states: s_1 where the client is waiting for an ACK from the server, and s_3 where the client is transmitting a packet to the server. The transition from state s_1 to s_3 is triggered when an ACK arrival event e_3 occurs at the client and when predicate $(p_3 == v_3) \wedge (v_3 < c_0)$ is true. The predicate means that the sequence number p_3 carried by the ACK in event e_3 equals the expected ACK sequence number v_3 at the client, and v_3 is less than the sequence number c_0 (a constant in a test) of the final ACK. If the EEFSM takes this transition, assignment $v_3 = v_3 + 1$ is performed. That is, the client is now expecting the next ACK sequence number.

The current status of the EEFSM is described by its *configuration*, which is a 2-tuple $\langle s, \overrightarrow{\mathbf{v}} \rangle$ containing its current state s and variable values $\overrightarrow{\mathbf{v}}$. For each test input, when we run the EEFSM on its corresponding sequence of IUT events, the EEFSM always starts from the initial configuration with the initial state s_0 and initial variable values. The EEFSM will finally stop on either the accepting state s_a or the rejecting state s_r. If the EEFSM stops on the accepting state, the sequence of IUT events is accepted by the EEFSM; otherwise, is rejected. *An IUT passes a test if the sequence of events is accepted by the EEFSM, and fails on the test otherwise.*

3.4 Symbolic Execution Platform

Our work on testing protocols under different packet dynamics builds on the symbolic based testing approaches and engines of SPD [55] and S^{2}E [14]. Fig. 6 shows the architecture of the testing platform, where modules with label 1 are the IUT, modules with label 2 are SPD [55], modules with label 3 are S^{2}E [14], module with label 4 implements an EEFSM, and modules with labels 5 and 6 implement the proposed heuristics to be discussed later.

1) IUT is the implementation of a transport protocol. In our studies that corresponds to TFTP discussed, but others can be checked in the same fashion. The testing platform currently supports an IUT in the user space.

2) SPD (Symbolic Packet Dynamics) proposed by Sun *et al.* [55] implements a discrete-event network emulator with packet delay

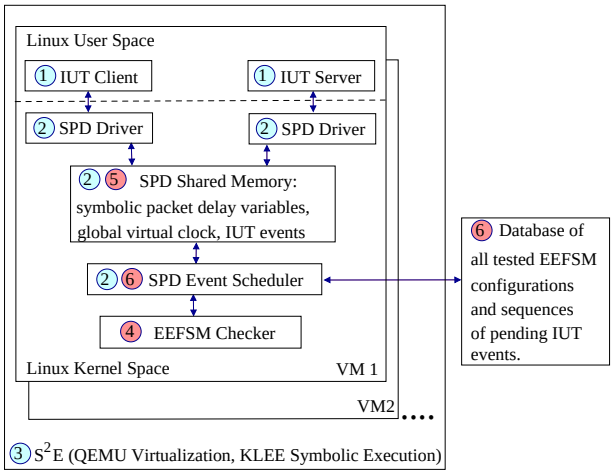


Figure 6: Testing platform architecture. 1: IUT; 2: SPD; 3: S^2E; 4: EEFSM; 5: packet retransmission priority; 6: network state priority. Our contributions include: 4, 5, and 6.

represented by symbolic variables. The IUT client and server communicate through SPD within a single virtual machine (VM), and SPD provides the IUT with an illusion of communicating over a network while using a symbolic packet delay like the one in Section 3.1. SPD consists of four main components: client driver, server driver, event scheduler, and shared memory.

The client and server drivers are implemented in the Linux kernel space and intercept all network-related and time-related system calls of the IUT client and server, respectively, so that the IUT can be tested directly without any modification. For these system calls, the drivers create the corresponding IUT events, such as packet departure, packet arrival, and timeout events. The event scheduler schedules all the IUT events in a VM by maintaining a global virtual clock and a global sequence of pending IUT events. The drivers and the event scheduler coordinate with one another using shared memory, which contains the global virtual clock for the emulated network, the pending IUT events, and symbolic packet delay variables. Note that, the symbolic packet delay variables are introduced in the shared memory of SPD. As a result, *all the code dependent on the shared memory, including the SPD event scheduler, SPD client and server drivers, IUT client, and IUT server, will be executed symbolically.*

3) S^2E (Symbolic Selective Execution) proposed by Chipounov *et al.* [14] is a symbolic execution platform, which can symbolically execute code in the user space or kernel space of a VM. Specifically, the VMs are emulated using QEMU virtualization, and symbolic execution is conducted using the KLEE symbolic execution engine [10]. We use all default S^2E parameters and configurations.

S^2E starts executing the user-space IUT and kernel-space SPD in a single VM (called the starting VM). *The initial constraints of the symbolic packet delay variables of the starting VM define the input space.* Every time the execution reaches a branching statement (in the SPD event scheduler, SPD drivers, or IUT code) dependent on symbolic variables, such as the `if` statement in Fig. 1, S^2E checks both branches by forking into two corresponding VMs. We choose the default depth-first order to execute these branches. At any time, each VM is associated with a set of accumulative constraints for the symbolic packet delay variables, which describes an equivalence class of test inputs with the same sequence of IUT events so far. Note that, *all the test inputs in the same equivalence class lead to not only the same execution path of the IUT client and server, but also the same execution path of the SPD including the SPD event scheduler and SPD drivers.*

4) EEFSM checker determines whether the sequence of IUT events of a VM is accepted by the EEFSM or not, and is marked with label 4 in Fig. 6. The EEFSM checker works in parallel with the IUT and SPD execution. Every time the SPD event scheduler finds the next earliest IUT event, the EEFSM checker checks that event immediately. There are three cases depending on the next EEFSM state. First, if the next state is the rejecting state, the IUT fails on the corresponding equivalence class of test inputs and the current VM stops. Second, if the next state is the accepting state, the IUT passes the corresponding equivalence class of test inputs and the current VM stops. Third, if the next state is not the rejecting or accepting state, the current VM continues.

4 PROPOSED TECHNIQUES

We propose two domain-specific heuristic techniques for symbolic execution based testing, packet retransmission based priority and network state based priority, which prioritize test inputs to increase the exposure of different IUT behaviors within a limited testing budget. By doing so, they can improve the cost-effectiveness of symbolic execution based testing. The techniques prioritize the inputs into high and low priority based on the likelihood of revealing a new IUT behavior, where high-priority points are exercised before all low-priority points in the input space.

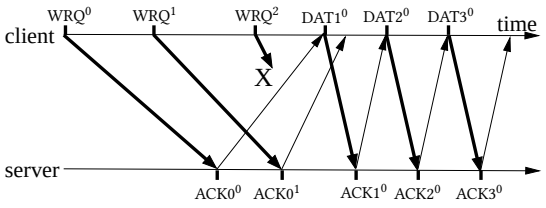
Note that through the proposed techniques, we delay the execution of low-priority test inputs but do not drop them. The reason is that our heuristic techniques are based on observations of a fault-free IUT where these low-priority test inputs do not reveal any new behaviors for a fault-free IUT, but may reveal some new behaviors for faulty IUTs.

4.1 Packet retransmission based priority

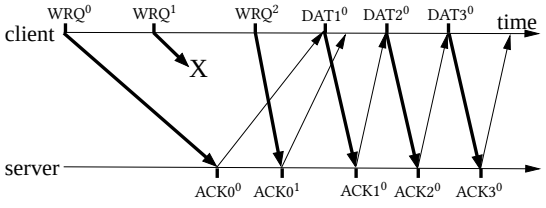
This prioritization is motivated by the observation that packet retransmission is a common component of many transport protocols. For example, both TFTP (studied here) and TCP (dominating Internet transport protocol) retransmit the same packet if it has been delayed for a long time or lost. In general, we find that *many test inputs with different packet retransmission patterns (thus in different equivalence classes as per their packet dynamics) have the same or repetitive IUT behaviors.* Hence, we execute only a small number of these tests with a high priority, and the rest with a low priority.

4.1.1 Motivating example 1. To simplify the discussion, let's consider a fault-free IUT. Fig. 7 shows the IUT behaviors on three test inputs with different packet retransmission patterns. The three inputs have three identical WRQ packets: WRQ^0, WRQ^1, and WRQ^2, but since different WRQ packets are lost (delay is T_∞) they have different sequences of IUT events.

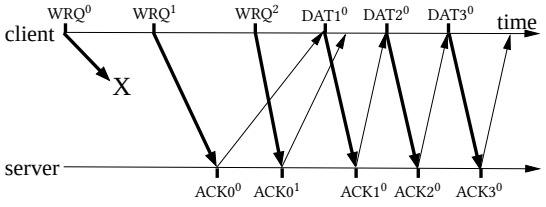
The three test inputs belong to different equivalent classes, because they traverse distinct SPD execution paths (i.e., label 2 in Fig. 6), specifically, different SPD event scheduler paths and different SPD driver paths. But we observe that the three test inputs



(a) First



(b) Second



(c) Third

Figure 7: Three test inputs with different packet retransmission patterns have the same IUT behavior. Label X: delay is T_∞ and thus the packet is lost.

have completely the same IUT behavior in both the client and the server (i.e., label 1 in Fig. 6). 1) In spite of having different retransmission patterns, the three test inputs have exactly the same sequence of events occurring at the client, such as WRQ^0 departure, WRQ^0 timeout, WRQ^1 departure, WRQ^1 timeout, WRQ^2 departure, and $ACK0^0$ arrival. Therefore, we observe the exact same client behavior under the three test inputs. 2) At the server side we observe something similar to the client. The three test inputs lead to different sequences of events reaching the server. Specifically, when WRQ^0 arrives at the server in Figs. 7(a) and 7(b), WRQ^1 arrives at the server in Fig. 7(c). However, from the server IUT perspective, there is no difference between these inputs, in fact the IUT server executes exactly the same sequence of program statements when receiving these different but identical WRQ packets. Therefore, all three inputs, although exploring different packet dynamics, cause the same IUT behavior in the client and server, so we argue it is only necessary to test one of them with a high priority.

Motivating example 2. Fig. 8 shows a test input similar to that from Fig. 7(a) but with WRQ^2 successfully arriving to the server. This leads to Fig. 8 having three more events than Fig. 7(a): 1) the WRQ^2 arrival at the server, 2) the $ACK0^2$ departure from the server, 3) the $ACK0^2$ arrival at the client.

As per the TFTP specifications, TFTP has the same behavior in the presence of received identical packets after a threshold $\alpha = 2$ (i.e., client dropping these packets, and server retransmitting the last ACK packet). Therefore, the server in Fig. 8 executes exactly

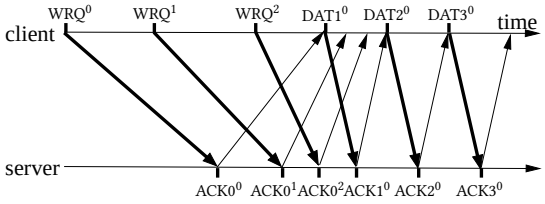


Figure 8: This test input and the one in Fig.7(a) have different packet retransmission events but repetitive IUT behaviors.

the same sequence of program statements when WRQ^1 (i.e., the second received identical packet) and WRQ^2 (i.e., the third received identical packet) arrive at at the server. That is, the behavior of the IUT server on the WRQ^2 arrival is repetitive to that on the WRQ^1 arrival. Similarly, the two other extra events (i.e., the $ACK0^2$ departure from the server and the $ACK0^2$ arrival at the client) in Fig. 8 also lead to repetitive IUT behaviors.

Consequently, although the test input in Fig. 8 leads to more events than that in Fig. 7(a), the behavior triggered by the extra events in Fig. 8 is redundant. Therefore, between these two test inputs, we only need to test one with high priority. Note that, this example assumes that all identical packet arrivals beyond a threshold α are treated similarly. This is true not only for TFTP (studied in this work with $\alpha = 2$) but also for many other transport protocols, such as TCP which uses the same code to process the fourth and other subsequently received identical packets. (e.g., dup ACKs with $\alpha = 4$).

4.1.2 Proposed technique. Motivated by the above two examples, we propose *packet retransmission based prioritization* for the points in the input space. Specifically, the priority of a test input $\overrightarrow{\triangle t}=(\triangle t_0, \triangle t_1, ..., \triangle t_{n-1})$, with R being the maximum number of retransmission per original packet, and $\alpha \in [1, R+1]$ as the threshold in the second motivating example, is assigned as follows, where predicate $P(i)$ denote statement $(i\%(R+1) \geq \alpha) \rightarrow (\triangle t_i = T_\infty)$

$$\begin{cases} \text{high}, & \text{if } \forall i\ P(i) \text{ is true} \\ \text{low}, & \text{otherwise} \end{cases} \quad (1)$$

Let's calculate the priority of Figs. 7(a), 7(b), 7(c), and 8 using priority assignment (1) to understand how it works. Supposing that $P = 8$ and $R = 4$, there are at most $P(R+1) = 40$ packets in a test and the input space is $n = 40$ dimensional. For example, Fig. 7(a) shows the delays of 11 packets, and the other $40 - 11 = 29$ packets (e.g., WRQ^3, $ACK0^2$, and $DAT1^1$) are not transmitted in this test. For these untransmitted packets, our implementation treats their delays to be T_∞ in the test input and other delay values do not need to be checked in the test. Specifically, the test input of Fig. 7(a) is a 40-dimensional delay vector: $\triangle t_0 < T_\infty$ (i.e., WRQ^0), $\triangle t_1 < T_\infty$ (i.e., WRQ^1), $\triangle t_2 = T_\infty$ (i.e., lost WRQ^2), $\triangle t_3 = T_\infty$ (i.e., untransmitted WRQ^3), $\triangle t_4 = T_\infty$ (i.e., untransmitted WRQ^4), $\triangle t_{R+1=5} < T_\infty$ (i.e., $ACK0^0$), $\triangle t_6 < T_\infty$ (i.e., $ACK0^1$), $\triangle t_7 = T_\infty$ (i.e., untransmitted $ACK0^2$), ..., , $\triangle t_{10} < T_\infty$ (i.e., $DAT1^0$), ..., $\triangle t_{35} < T_\infty$ (i.e., $ACK3^0$), ..., $\triangle t_{39} = T_\infty$ (i.e., untransmitted $ACK3^4$). This is illustrated in Table 1, together with the prioritization values for each input. Therefore, if $\alpha = 2$, the condition $\forall i\ P(i)$ in priority assignment (1) is true for Fig. 7(a) (i.e., high priority), but false for Figs. 7(b) 7(c), and 8 (i.e., low priority).

Table 1: Applying Packet Retransmission Prioritization to the Inputs from Figs. 7 and 8. Assume $R = 4$ and $\alpha = 2$.

		Fig. 7(a)		Fig. 7(b)		Fig. 7(c)		Fig. 8	
i	i%(4+1)≥ 2	$\triangle t_i = T_\infty$	$P(i)$	$\triangle t_i = T_\infty$	$P(i)$	$\triangle t_i = T_\infty$	$P(i)$	$\triangle t_i = T_\infty$	$P(i)$
0	0 ≥ 2 = False	False	True	False	True	True	True	False	True
1	1 ≥ 2 = False	False	True	True	True	False	True	False	True
2	2 ≥ 2 = True	True	True	False	False	False	False	False	False
3	3 ≥ 2 = True	True	True	True	True	True	True	True	True
4	4 ≥ 2 = True	True	True	True	True	True	True	True	True
5	0 ≥ 2 = False	False	True	False	True	False	True	False	True
...	...	...	...	...	...	...	...	...	...
$\forall i\ P(i)$			True		False		False		False

The proposed technique could be extended to more than two levels of priority. For example, the test inputs shown in Figs. 7(b) 7(c), and 8 could have different levels of priority.

4.1.3 Cost-effectiveness analysis. Each retransmitted packet adds one more dimension to an input space, and thus a significant portion of the input space is caused by packet retransmission. Let's still consider the motivating example with $P = 8$, $R = 4$, and an $n = P(R + 1) = 40$ dimensional input space. If the range of each dimension contains 65535 delay values (e.g., if a delay is represented by a 16-bit unsigned integer), there are $65535^n = 65535^{40}$ points in the input space. Symbolic execution based testing struggles to tests all these points with the same priority.

For each of the P original packets (e.g., WRQ^0), our technique gives a high priority to the points where the first α identical packets (e.g., WRQ^0, ..., $WRQ^{\alpha-1}$) can take all possible delays but the rest $R+1-\alpha$ identical packets (e.g., WRQ^α, ..., WRQ^R) can take only delay T_∞. Therefore, a total of $65535^{P\times\alpha}$ points are set to a high priority. For example, if $\alpha = 2$, $65535^{P\times\alpha} = 65535^{8\times2} = 65535^{16}$ points have a high priority. Our technique first tests these 65535^{16} high-priority points since they are more likely to reveal new IUT behaviors, and then continues to test the remaining $65535^{40}-65535^{16}$ low-priority points that are less likely to reveal new IUT behaviors. Therefore, given a limited testing budget, this technique is more cost-effective than traditional symbolic execution based testing.

4.1.4 Implementation. Our prioritization technique builds on the symbolic testing platform introduced in Fig. 6, and the specific technique is marked with label 5 in the same figure. The priority of each point is assigned at the test design time (before test execution like most traditional test case prioritization techniques [19]), where we divide the whole input space into subspaces using Eq. 1. This particular implementation uses two subspaces: the subspace of high-priority points and then the subspace of low-priority points. Each subspace is separately tested using the symbolic execution platform shown in Fig. 6. This is implemented by running the symbolic execution platform two times using the starting VM with different initial constraints for the symbolic packet delay variables. In the first time, the initial constraints are the constraints of the high-priority points, and in the second time, the initial constraints are the constraints of the low-priority points.

4.2 Network state based priority

As we have mentioned, this technique is motivated by the observation that many transport protocols work as an event-driven finite state machine. Given a test input, the future of the protocol IUT depends on the current *network state*, which consists of not only the current IUT state (i.e., IUT client and server state variables), but also the sequence of pending IUT events and the remaining packet dynamics of the test input (i.e., delays of the outstanding and future packets). We find that *the executions of many test inputs visit the same network state*, and thus many test inputs with initially different IUT event sequences reach the same network state and subsequently have the same IUT behavior (i.e., *partially the same IUT behavior*). This heuristic is meant to assign low-priority to those inputs leading to the same IUT behavior. Different from the prior technique, this implementation determines the priority of each point at the test execution time instead of the test design time, delaying the symbolic exploration of paths rendering an already observed network state.

4.2.1 Motivating example. Let's consider a fault-free IUT again, and the two test inputs shown in Fig. 3. The two inputs lead the IUT to the same network state at the time specified by the dotted lines. At that time, they have the same IUT state, such as the client expected ACK sequence number, the server expected DAT sequence number, the client retransmission count, and so on. Note that prior to that time, they have different IUT states, because they have different client retransmission counts (i.e., WRQ is retransmitted for zero time in Fig. 3(a) but once in Fig. 3(b)). The client retransmission count is reset to zero after the client sends out $DAT1^0$ (i.e., the dotted lines). They also have the same sequence of pending IUT events: first the pending server $DAT1^0$ arrival event, then the pending server timeout event for $ACK0^0$, and finally the pending client timeout event for $DAT1^0$. Fig. 3(b) also has a pending server WRQ^0 arrival event that has no impact on the IUT behavior since WRQ^0 is lost. Last, they have the same remaining packet dynamics: each of the remaining packets (i.e., DAT1, DAT2, DAT3, ACK1, ACK2, and ACK3) has the same packet delay in both test inputs.

Thus the two test inputs in Fig. 3 lead to initially different but subsequently the same sequences of future IUT events after the same network state (at the dotted lines) is reached.

4.2.2 Proposed technique. Our proposed *network state based priority* assigns the priority of a test input $\overrightarrow{\triangle t}$ as follows.

$$\begin{cases} \text{high}, & \text{if every network state of } \overrightarrow{\triangle t} \text{ is new} \\ \text{low}, & \text{otherwise} \end{cases} \quad (2)$$

Identifying the relevant IUT state variables of a general transport protocol implementation is likely to require expert participation (they are often encoded throughout multiple variables and code constructs across a large code base). Instead, to make the process fully automated, we abstract the IUT state using an EEFSM configuration. We set the priority of $\overrightarrow{\triangle t}$ to low, as soon as we find that the current network state has already been executed by another test input. The execution of the IUT on $\overrightarrow{\triangle t}$ is then suspended, and will be resumed later after all high-priority test inputs have been checked. Just like the prior technique, we delay instead of terminating the executions of these low-priority test inputs.

Let $\overrightarrow{\triangle t}_{3(a)}$ and $\overrightarrow{\triangle t}_{3(b)}$ denote the test inputs of Figs. 3(a) and 3(b), respectively, and calculate their priorities using the priority assignment (2) to understand how it works. Because the priority of each test input depends on the execution order of all test inputs, to simplify the explanation, we suppose that we execute first $\overrightarrow{\triangle t}_{3(a)}$, next $\overrightarrow{\triangle t}_{3(b)}$, and then all other test inputs. 1) Execution of $\overrightarrow{\triangle t}_{3(a)}$: After executing each event, the EEFSM configuration is updated according to the event. The current network state is inferred using the current EEFSM configuration, the pending IUT events, and the remaining delays of $\overrightarrow{\triangle t}_{3(a)}$. Because this is the first test input, $\overrightarrow{\triangle t}_{3(a)}$ will be executed completely (i.e., high priority). 2) Execution of $\overrightarrow{\triangle t}_{3(b)}$: After executing each event, the EEFSM configuration is updated according to the event. The current network state is inferred and compared with all network states already visited. After executing the client DAT1^0 departure event, we find that the current network state is the same as a network state already visited before (i.e., when executing $\overrightarrow{\triangle t}_{3(a)}$). As a result, the execution of $\overrightarrow{\triangle t}_{3(b)}$ will be suspended (i.e., low priority). 3) Execution of all other test inputs: they will be executed as high-priority until they reach a seen network state at which point they will be marked as low priority and their execution will be suspended. 4) The executions of all low-priority test inputs will be resumed.

4.2.3 Cost-effectiveness analysis. Due to the finite-state-machine nature of transport protocols, this technique has the potential to greatly improve the cost-effectiveness of symbolic execution based testing. The exact cost-effectiveness improvement depends on the EEFSM and IUT and is studied in Section 5.

4.2.4 Implementation. Different from the prior technique, this implementation sets the priority of each point at the test execution time instead of the test design time. The implementation of network state based priority (*label 6* in Fig. 6) checks for the same network states among VMs (i.e., equivalence classes of test inputs instead of a single test input). It maintains a network state database containing the checked EEFSM configurations and the sequences of pending IUT events, which is stored outside S^{2}E and can be accessed by any VM emulated by S^{2}E via real UDP packets. The database does not need to maintain the information of remaining packet delays because the delay of each remaining packet is described by a symbolic variable, whose constraint is still the same initial constraint for every VM. Thus, as long as two VMs have the same EEFSM configuration, they have the same remaining packet delay constraints. After the SPD event scheduler executes each IUT event in a VM, we check whether the current EEFSM configuration and the current sequence of pending IUT events of the VM are the same as any of those in the database. If so, the current network state has already been executed by another VM, and thus the current VM is suspended. Otherwise, the current EEFSM configuration and sequence of pending IUT events are added into the database, and the current VM continues its execution.

5 EXPERIMENTS

This section evaluates the cost-effectiveness of our techniques when applied to the symbolic execution testing platform described in Section 3.4 and the techniques from Section 4. Through these experiments we measure the number of faults or mutants that can be killed by the proposed and existing techniques over units of time, and explore the impact of the parameters provided as part of the techniques.

5.1 General Setup

IUTs. We choose tftp-hpa [1] as the IUTs in the experiments, because it is currently the most widely used TFTP application in Linux. It consists of the tftp-hpa server and tftp-hpa client. We use two versions of tftp-hpa: the latest version tftp-hpa 5.2 released in December 2011, and a previous version, tftp-hpa 0.24, with two known confirmed faults released in November 2001.

Faults. The first experiment aims to assess cost-effectivenesss on real faults in both tftp-hpa 0.24 and 5.2, and the second experiment uses mutation to generate faulty versions of tftp-hpa 5.2.

EEFSM. Following the process outlined in Section 3.3, we carefully examined the required TFTP specifications [7, 51], searching for the sentences with the keywords such as "must" and "must not", and then constructing the EEFSM representing these sentences. We excluded the optional TFTP specifications, such as RFC 2347 [36], 2348 [35], and 2349 [37]. The constructed EEFSM is used in both experiments. Because the EEFSM is large (10+ states and 35+ transitions), it will be given in a technical report to be cited.

Input space. We test the IUTs using a file with three data packets. The last data packet is used to test connection termination, and the other two data packets are used to test the TFTP connection. Therefore, there are a total of $P = 5 + 3 = 8$ original packets as shown in Fig. 3. Since the tftp-hpa server and client have different maximum numbers of retransmissions (server is five and client is four), to simplify our discussion, we change the maximum number of server retransmissions to be the same as that of the client (i.e., $R = 4$). For the rest of the parameters we use the default tftp-hpa values. As a result, there are a total of at most $P(R + 1) = 40$ packets, and the input space is an $n = 40$ dimensional vector $\overrightarrow{\triangle t}$=($\triangle t_0, \triangle t_1, ..., \triangle t_{39}$) as shown in Fig. 4.

Each packet delay $\triangle t_i$, $i \in [0, 39]$, is represented by an unsigned 16-bit integer in the C language, and can take any positive delay in $[1, 65535]$. The packet delay unit is 4 ms, which is the default Linux time granularity (i.e., jiffies, corresponding to default Linux Kernel constant HZ=250). Therefore, the delay range is $[1 \times 4,\ 65535 \times 4]$=$[4, 262140]$ ms (i.e., $T_\infty = 262140$ ms). This range is much longer than any possible running time of our tftp-hpa experiments, and

thus packets with a long delay can be considered as lost in the experiments.

Process. The cost-effectiveness of a testing method is measured by the number of faults detected within a testing budget. Because the testing platform shown in Fig 6 will run out of memory after 10 hours of symbolic execution on our 32GB memory computers, the maximum testing budget is set to 10 hours when testing an IUT on an input space.

Testing infrastructure. Supported by our university cloud computing platform, we use a total of 17 virtual machines configured with Ubuntu 14.04.3, 2.3 GHZ Intel 4-Core Haswell Processor, and 32 GB memory.

5.2 Treatments

We evaluate the following testing methods.

1) Symbolic Packet Dynamics (SPD) proposed by Sun *et al.* [55], which is a symbolic execution based testing method that considers packet dynamics and will be used as a reference baseline method.

2) SPD using the proposed Network State based priority (NS), Packet Retransmission based priority (PRα), and both of them (NSPRα). We evaluate NS, PR2, NSPR2, as well as PR1 and NSPR1. Methods PR1 and NSPR1 are used to study the impact of α with less than the recommended value. Because the maximum value of α is $R+1=5$ and packet retransmission based priority with $\alpha = 5$ set all points to a high priority, methods PR5 and NSPR5 should have the same performance as SPD and NS, respectively, so we use them to study the impact of α with greater than the recommended value.

3) RANdom testing (RAN), which keeps randomly selecting test inputs from an input space within the testing budget, and is implemented using the SPD platform. Specifically, the value of each symbolic packet delay variable is randomly generated in the input space according to a shifted Gamma distribution [39, 41] (with parameters [24], specifically scale=1, shape=2.5, shift=107.5 with additional 1% packet loss) which is often used to model the Internet packet delay and loss.

5.3 Exp#1: Cost-effectiveness with Real Faults

This experiment evaluates cost effectiveness using real faults.

5.3.1 IUTs. We use both tftp-hpa [1] 0.24 and 5.2 as the IUTs. tftp-hpa 0.24 has two known packet dynamics faults, both fixed in the latest version tftp-hpa 5.2. *Fault 1* [33]: Adaptive timeout: RFC 1123 [7] requires that "A TFTP implementation MUST use an adaptive timeout". However, tftp-hpa 0.24 implements only a fixed timeout mechanism. *Fault 2* [32]: Sorcerer's Apprentice Syndrome: RFC 1123 requires that "Implementations MUST contain the fix for this problem: the sender must never resend the current DAT packet on receipt of a duplicate ACK". tftp-hpa 0.24 does not fix the Sorcerer's Apprentice Syndrome problem.

5.3.2 Process. We run each method on the IUT with the default input space for at most 10 hours. If a testing method detects a fault, we fix the fault and then re-run the testing method on the modified IUT to detect the next fault.

5.3.3 Results. Tables 2 and 3 shows the results for tftp-hpa 0.24 and 5.2, respectively, where "No" indicates that the method fails to detect the fault in 10 hours, and a number indicates the running time (in seconds) of the method to successfully detect the fault.

Table 2: Running time (sec) to detect faults of tftp-hpa 0.24

Result	SPD	RAN	NS	PR1	NSPR1	PR2	NSPR2
Fault 1	12	57	13	11	12	11	12
Fault 2	No	46	24639	No	No	27332	297
Fault 3	No	No	No	No	No	No	851

Table 3: Running time (sec) to detect faults of tftp-hpa 5.2

Result	SPD	RAN	NS	PR1	NSPR1	PR2	NSPR2
Fault 3	No	No	No	No	No	No	863

NSPR2 successfully detects both known faults of tftp 0.24. In addition, it detects a new fault (called fault 3) of both tftp-hpa 0.24 and 5.2. *Fault 3*: RFC 1350 [51] requires that "the host sending the final ACK will wait for a while before terminating in order to retransmit the final ACK if it has been lost. The acknowledger will know that the ACK has been lost if it receives the final DATA packet again." However, both tftp 0.24 and 5.2 retransmit the final ACK if they receive any (not just the final) DATA packet again. As a result, they sometimes send an extra useless ACK to the network, which wastes the precious network bandwidth. We have reported this fault and our fix to the tftp-hpa developers and are waiting for their confirmation.

SPD successfully detects fault 1 using 12 seconds, but could not detect the other two faults. RAN successfully detects faults 1 and 2, but could not detect fault 3 because the probability of fault 3 is very low (about 10^{-9}). NSPR2 is the only one that can detect all three faults. Therefore, in this experiment, *NSPR2 is more cost-effective than both SPD and RAN for all testing budgets up to the maximum 10 hours.*

5.4 Exp#2: Cost-effectiveness with Mutants

5.4.1 IUTs. To comprehensively assess cost-effectiveness, we need a large number of faulty protocol implementations. Since the versions we studied only have three faults, we use mutation to provide us with a larger number of faults to better characterize the performance of the proposed techniques.

Since tftp-hpa [1] is written in the C language, we use the MiLu [25] mutation tool which works in that language. Since the latest version tftp-hpa 5.2 has one fault (i.e., fault 3), we first fix the fault to get a fault-free tftp-hpa, and then use MiLu to modify the fault-free tftp-hpa. Because we only care about faults related to packet dynamics, we use MiLu to modify just tftp-hpa functions related to packet dynamics. Finally, MiLu generates a total of 679 mutants, including 316 mutants that change the client and 363 mutants that change the server.

We can classify the generated mutants into three groups after manually checking these mutants. First, there are 25 *invalid mutants* that do not build. Second, there are a total of 281 mutants that appear to have the same network behaviors as the fault-free tftp-hpa (i.e., no packet dynamics faults) and could be considered *equivalent mutants*. However, given the complexity of the code, we are not certain about their equivalence. Thus we tested each of these 281 mutants using all 7 testing methods, and none of these 281 mutants was killed by any of these testing methods. Third,

there are a total of 373 *faulty mutants* that cause different network behaviors than the fault-free tftp-hpa when exposed to certain test inputs. These 373 faulty mutants are the IUTs used to evaluate cost-effectiveness.

5.4.2 Process. The cost-effectiveness of a testing method is measured by its *mutation score* achieved with a testing budget (up to 10 hours for each mutant), which is the percentage of 373 faulty mutants killed by the method within the given testing budget. A mutant is killed by a method if the mutant fails on a test input generated by the method. With the same testing budget, the higher the mutation score of a testing method, the more cost-effective the testing method. We run each of the seven testing method on each of these 373 faulty mutants for up to 10 hours. As soon as a method finds that a mutant fails on a test input, the method stops checking the remaining inputs on that mutant. The total time to test these 373 faulty mutants and to confirm the 281 equivalent mutants together took about 700 CPU days.

5.4.3 Overall Results. Fig. 9 shows the cost-effectiveness results (we removed PR1 and NSPR1 from this figure to make it more readable but their final numbers are provided next anyway). When the testing budget is under 100 seconds, RAN achieves the highest cost-effectiveness. However, its mutation score remains the same after the testing budget increases beyond 10 seconds, as it repeatedly selects inputs that do not reveal new IUT behaviors, and thus does not detect more faults. When the testing budget is under 100 seconds, our proposed techniques, NS, PR2, and NSPR2, achieve similar cost-effectiveness as SPD. Beyond 100 seconds, NS, PR2, and NSPR2 all improve the cost-effectiveness of SPD, and NSPR2 combining both priority techniques kills about three times as many mutants as SPD. Overall, *RAN is the most cost-effective when the testing budget is very limited (e.g., under 100 seconds), and NSPR2 is the most cost-effective when the testing budget is longer, significantly improving on SPD.*

Fig. 10 shows the mutation score of each method with the maximum testing budget (i.e., 10 hours) in increasing score. The proposed techniques, including PR1 and NSPR1, achieve higher mutation scores than SPD, going from 30.29% to 81.77%∼100% of faulty mutants killed. *Especially, NSPR2 kills all 373 faulty mutants.* We notice that NSPRα achieves a higher mutation score than NS and PRα. That is, when packet retransmission based priority and network state based priority are used together (i.e., NSPRα) their performance is better. Since RAN uses a random packet delay model that models the Internet delay and loss pattern, we conjecture that the percentage of faulty mutants killed by RAN (82.57%) may be a good estimate of the percentage of faulty mutants killed in real field tests (i.e., when directly running them in the Internet). RAN kills many mutants, but as shown later in this subsection, RAN misses some low-probability ones.

5.4.4 Impact of α. To study the impact of α, we re-plot Fig. 10 in two figures. Fig. 11 shows the impact of α on PRα, where PR5 has the same mutation score as SPD. Fig. 12 shows the impact of α on NSPRα, where NSPR5 has the same mutation score as NS. PR2 achieves a higher mutation score than both PR1 and PR5 in Fig. 11, and this illustrates the importance of choosing α carefully. When α (e.g., 1) is less than that recommended value (i.e., 2), the

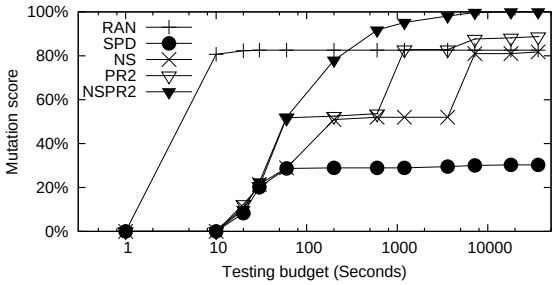


Figure 9: Cost-effectiveness: Mutants killed by each method over time (up to 10 hours = 36000 seconds).

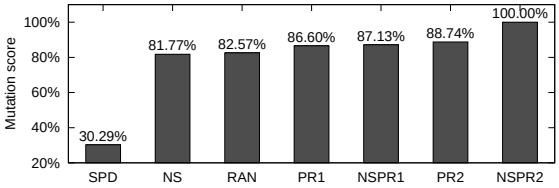


Figure 10: Mutants killed by each method in 10 hours.

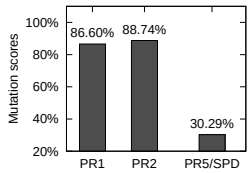


Figure 11: Impact of α on PRα.

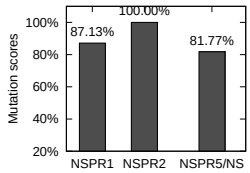


Figure 12: Impact of α on NSPRα.

faults triggered only by two identical packet arrivals will be tested with a low priority and thus require a longer testing budget to be exposed. When α (e.g., 5) is much greater than the recommended value, too many points in the input space are set to a high priority, and thus it also requires a longer testing budget to exercise just the high-priority points. For the same reason, NSPR2 achieves a higher mutation score than both NSPR1 and NSPR5 in Fig. 12.

5.4.5 Pairwise comparison. Table 4 shows the pairwise comparison between each pair of testing methods with the maximum testing budget. A number at row A and column B shows $|A-B|$ where $A - B = \{x \mid x \in A, \text{and } x \notin B\}$, and intuitively is the number of mutants that can be killed by A but not by B. For example, |SPD−RAN|=13. That is, SPD can kill 13 mutants not killed by RAN with the maximum testing budget. We now summarize the major findings revealed by this data, providing examples of the faulty behaviors that differentiate the techniques.

RAN ⊂ NSPR2, because |RAN−NSPR2|=0 and |NSPR2−RAN|=75. We find that NSPR2−RAN contains the mutants that show faulty behaviors with very low probability packet delays. For example, Fig. 13 shows the faulty behavior of a mutant, where the client mistakenly retransmits WRQ again after already reaching the maximum number of allowed retransmissions (i.e. 4). This mutant is killed by NSPR2, but not by RAN, because the probability of 5 consecutive timeouts in RAN is extremely low (about $10^{-2\times5} = 10^{-10}$).

Table 4: Pairwise comparison of testing methods (with the maximum 10 hours)

$\|A - B\|$	SPD	RAN	NS	PR1	NSPR1	PR2	NSPR2
SPD	0	13	0	2	0	0	0
RAN	208	0	35	0	0	15	0
NS	192	32	0	17	15	0	0
PR1	212	25	35	0	0	15	0
NSPR1	212	27	35	2	0	15	0
PR2	218	38	26	23	21	0	0
NSPR2	260	75	68	50	48	42	0

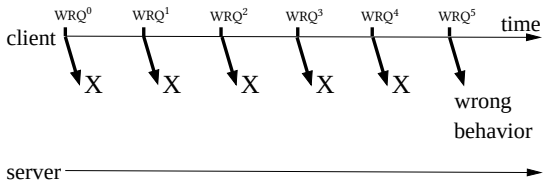


Figure 13: Fault exposed by NSPR2, but not by RAN.

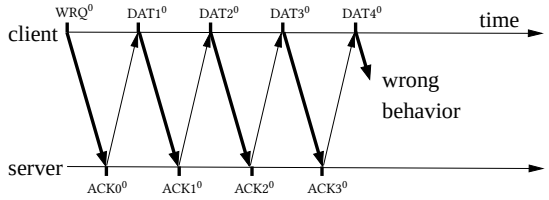


Figure 14: Fault exposed by NSPR2, but not by SPD.

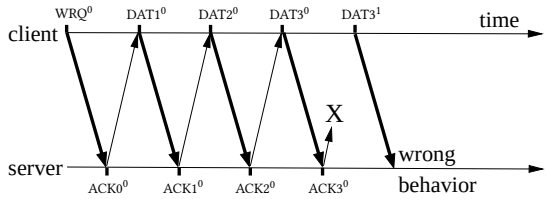


Figure 15: Fault exposed by NSPR2, but not by NSPR1.

SPD ⊂ NSPR2. Fig. 14 shows the faulty behavior of a mutant, where the client mistakenly sends an extra data packet (i.e., $DAT4^0$) after sending all three data packets. This mutant can be killed by NSPR2, but not by SPD because it spends all the 10 hours of testing budget on the test inputs with all possible packet retransmission patterns leading to aborted connections, but not on the test input shown in Fig. 14. Instead, NSPR2 first checks the high-priority test inputs including the one shown in Fig. 14 within the testing budget.

NSPR1 ⊂ NSPR2. Fig. 15 shows the faulty behavior of a mutant, where the server should retransmit ACK3 after receiving the retransmitted DAT3 (last data packet), but the server mistakenly does not do anything. Note that there are two successful DAT3 arrivals (i.e., $DAT3^0$ and $DAT3^1$) in Fig. 15. As a result, the test input shown in Fig. 15 is tested with a low priority by NSPR1, and thus have not been exercised by NSPR1 yet within the testing budget. Instead, the test input shown in Fig. 15 is exercised with a high priority by NSPR2 within the testing budget.

6 RELATED WORK

There exist a rich body of work on protocol conformance testing [5, 15, 17, 28, 60], which analyzes the FSM of a protocol and generates a set of packet sequences to test the conformance of one or multiple entities in an IUT. In contrast, our work studies how to test the specific impact of packet dynamics, and it always considers the interaction among all the entities of an IUT.

Transport protocols implementations have been validated and verified using a spectrum of techniques ranging from expert-written scripts [13], to trace [4, 29, 30, 38] and static analyses [57]. Within the symbolic approaches, we highlight NICE [12], KleeNET [48], and SymbexNET [52], which test an IUT using packets with symbolic data fields but with limited packet dynamics which is our focus. Closest to our work is SPD [55], which we build on as shown in Section 3.4, but with a focus on becoming more cost-effective by leveraging established properties of transport protocols.

Symbolic execution has also been used to check the data plane properties of an IUT [16, 54], find protocol manipulation attacks [26] of an IUT, analyze the network configurations policies [20, 53], and check for interoperability problems [42]. We refer to Quadir et al. [43] for a recent survey of network testing methods. Our contribution over this line of work is the focus on packet dynamics through a symbolic approach that can scale to detect faults that can be exposed only through longer packet sequences.

The scalability issues associated with symbolic execution are still central to realizing the potential of line of work. Several approaches have been proposed such as compositional symbolic execution [16, 21], redundant path elimination [6, 8, 45], path prioritization using static analysis information [3, 9], path merging [2, 23, 27, 49], and state mapping methods [46, 47]. These methods are are complementary to the domain-specific ones we are proposing in this work for transport protocol implementations.

The techniques proposed are related to two general approaches for improving testing cost-effectiveness. The retransmission technique is a kind of test case prioritization [19, 58]. The network-state prioritization also performs test prioritization but it does so at test execution time, so it is more related to approaches to guide and direct symbolic execution (e.g.,[31, 34, 50, 59]). Both proposed techniques are unique in that they are driven by protocol-based instead of code-based heuristics.

7 CONCLUSIONS

We have proposed two techniques to improve the cost-effectiveness of symbolic execution based testing for network transport protocols in the context of packet dynamics. Although we have shown the techniques' potential only on TFTP, we anticipate that the proposed techniques generalize to other transport protocols such as TCP, so part of our ongoing work consists of performing that assessment which requires a more sophisticated testing infrastructure. We are also investigating whether a family of heuristics such as the ones proposed could be used to provide feedback to the random method that is so cost-effective with smaller testing budgets. Last, we are exploring how to exploit our knowledge of these protocol packet dynamics properties to more quickly guide symbolic execution towards paths that may contradict them.

REFERENCES

[1] H. P. Anvin. TFTP HPA. https://www.kernel.org/pub/software/network/tftp/tftp-hpa/.

[2] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of International Conference on Software Engineering*. Hyderabad, India.

[3] D. Babic, L. Martignoni, S. McCamant, and D. Song. 2011. Statically-Directed dynamic automated test generation. In *Proceedings of ACM ISSTA*. Toronto, Canada.

[4] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. 2006. Engineering with Logic: HOL Specification and Symbolic-Evaluation Testing for TCP Implementations. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*. Charleston, SC.

[5] G. Bochmann and A. Petrenko. 1994. Protocol testing: review of methods and relevance for software testing. In *Proceedings of ACM ISSTA*. Seattle, WA.

[6] P. Boonstoppel, C. Cadar, and D. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Hungary.

[7] R. Braden. 1989. Requirements for Internet Hosts – Application and Support. *RFC 1123* (October 1989).

[8] S. Bugrara and D. Engler. 2013. Redundant state detection for dynamic symbolic execution. In *Proceedings of USENIX ATC*. San Jose, CA.

[9] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of IEEE/ACM Conference on Automated Software Engineering*. L'Aquaila, Italy.

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

[11] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (February 2013), 82–90.

[12] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. 2012. A NICE way to test OpenFlow applications. In *Proceedings of USENIX NSDI*. San Jose, CA.

[13] N. Cardwell, Y. Cheng, and et al. 2013. PacketDrill: Scriptable Network Stack Testing, from Sockets to Packets. In *Proceedings of USENIX ATC*. San Jose, CA.

[14] V. Chipounov, V. Kuznetsov, and G. Candea. 2012. The S2E platform: design, implementation, and applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. DOI: http://dx.doi.org/10.1145/2110356.2110358

[15] T. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering* 4, 3 (May 1978), 178–187.

[16] M. Dobrescu and K. Argyraki. 2014. Software dataplane verification. In *Proceedings of USENIX NSDI*. Seattle, WA.

[17] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, and N. Yevtushenko. 2010. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology (Elsevier)* 52, 12 (December 2010), 1286–1297.

[18] O. Dustmann. 2013. Symbolic Execution of Discrete Event Systems with Uncertain Time. *Lecture Notes in Informatics* S-12 (2013), 19–22.

[19] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Software Eng.* 28, 2 (2002), 159–182. DOI: http://dx.doi.org/10.1109/32.988497

[20] S. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. 2016. BUZZ: Testing context-dependent policies in stateful networks. In *Proceedings of USENIX NSDI*. Santa Clara, CA.

[21] P. Godefroid. 2007. Compositional Dynamic test generation. In *Proceedings of POPL*. Nice, France.

[22] S. Ha, I. Rhee, and L. Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review* 42, 5 (July 2008), 64–74.

[23] T. Hansen, P. Schachte, and H. Sondergaard. 2009. State Joining and Splitting for the Symbolic Execution of Binaries. In *Proceedings of Runtime Verification (RV)*. France.

[24] IEEE Working Group 802.20. 2005. 802.20 Evaluation Criteria - Ver. 1.0. *802.20 Working Group Permanent Documents* (September 2005).

[25] Y. Jia and M. Harman. 2008. MiLu: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Proceedings of Testing: Academic & Industrial Conference Practice and Research techniques*. Windsor, United Kingdom.

[26] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. 2011. Finding Protocol Manipulation Attacks. In *Proceedings of ACM SIGCOMM*. Toronto, Canada.

[27] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of ACM Programming Language Design and Implementation*. Beijing, China.

[28] R. Lai. 2002. A survey of communication protocol testing. *Journal of Systems and Software (Elsevier)* 62, 1 (May 2002), 21–46.

[29] F. Lalanne and S. Maag. 2013. A Formal Data-Centric Approach for Passive testing of Communication Protocols. *IEEE/ACm Transactions on Networking* 21, 3 (June 2013), 788–801.

[30] D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. 2006. Network protocol system monitoring - a formal approach with passive testing. *IEEE/ACM Transactions on Networking* 14, 2 (2006), 424–437.

[31] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 19–32. DOI: http://dx.doi.org/10.1145/2509136.2509553

[32] Linux tftp-hpa commit. 2001. Sorcerer's Apprentice bug. https://goo.gl/Nu3bsc.

[33] Linux tftp-hpa commit. 2002. Adaptive Timeout. https://goo.gl/PUOwXK.

[34] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 95–111. http://dl.acm.org/citation.cfm?id=2041552.2041563

[35] G. Malkin and A. Harkin. 1998. TFTP Blocksize Option. *RFC 2348* (May 1998).

[36] G. Malkin and A. Harkin. 1998. TFTP Option Extension. *RFC 2347* (May 1998).

[37] G. Malkin and A. Harkin. 1998. TFTP Timeout Interval and Transfer Size Options. *RFC 2349* (May 1998).

[38] P. Mouttappa, S. Maag, and A. Cavalli. 2013. Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols. *Computer Networks* 57, 15 (October 2013), 2992–3008.

[39] A. Mukherjee. 1994. On the dynamics and significance of low frequency components of Internet load. *Internetworking: Research and Experience* 5 (December 1994), 163–205.

[40] M. Musuvathi and D. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proceedings of USENIX NSDI*. San Francisco, CA.

[41] V. Paxson. 1997. End-to-End Internet Packet Dynamics. In *Proceedings of ACM SIGCOMM*. France.

[42] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. 2015. Analyzing protocol implementations for interoperability. In *Proceedings of USENIX NSDI*. Oakland, CA.

[43] J. Qadir and O. Hasan. 2015. Applying formal methods to networking: theory, techniques, and applications. *IEEE Communication Survey and Tutorials* 17, 1 (First Quarter 2015), 256–291.

[44] W. Rathje and B. Richards. 2014. A framework for Model checking UDP network programs with Java Pathfinder. In *Proceedings of ACM High Integrity Language Technology (HILT) International Conference*. Portland, OR.

[45] Eric F. Rizzi, Matthew B. Dwyer, and Sebastian G. Elbaum. 2014. Safely reducing the cost of unit level symbolic execution through read/write analysis. *ACM SIGSOFT Software Engineering Notes* 39, 1 (2014), 1–5. DOI: http://dx.doi.org/10.1145/2557833.2560580

[46] R. Sasnauskas, O. Dustmann, B. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. 2011. Scalable Symbolic Execution of Distributed Systems. In *Proceedings of ICDCS*. Minneapolis, MN.

[47] R. Sasnauskas, P. Kaiser, R. Jukic, and K. Wehrle. 2012. Integration testing of protocol implementations using symbolic distributed execution. In *Proceedings of IEEE ICNP*. Austin, TX.

[48] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. 2010. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of ACM/IEEE IPSN*. Stockholm, Sweden.

[49] K. Sen, G. Necula, L. Gong, and W. Choi. 2015. MultiSE: Multi-Path Symbolic Execution using Value Summaries. In *Proceedings of ESEC/FSE*. Italy.

[50] Hyunmin Seo and Sunghun Kim. 2014. How We Get There: A Context-guided Search Strategy in Concolic Testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 413–424. DOI: http://dx.doi.org/10.1145/2635868.2635872

[51] K. Sollins. 1992. THE TFTP PROTOCOL (REVISION 2). *RFC 1350* (July 1992).

[52] J. Song, C. Cadar, and P. Pietzuch. 2014. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering* 40, 7 (July 2014), 695–709.

[53] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. 2013. SymNet: Static Checking for stateful networks. In *Proceedings of ACM CoNEXT HotMiddlebox Workshop*. Santa Barbara, CA.

[54] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of ACM SIGCOMM*. Brazil.

[55] W. Sun, L. Xu, and S. Elbaum. 2015. SPD: Automatically Test Unmodified Network Programs with Symbolic Packet Dynamics. In *Proceedings of IEEE Globecom*. San Diego, CA.

[56] K. Tan, J. Song, Q. Zhang, and M. Sridharan. 2006. A Compound TCP Approach for High-speed and Long Distance Networks. In *Proceedings of IEEE INFOCOM*. Barcelona, Spain.

[57] O. Udrea, C. Lumezanu, and J. Foster. 2006. Rule-based static analysis of network protocol implementation. In *Proceedings of USENIX Security Symposium*. Vancouver, Canada.

[58] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120.

DOI: http://dx.doi.org/10.1002/stv.430

[59] Pingyu Zhang, Sebastian G. Elbaum, and Matthew B. Dwyer. 2011. Automatic generation of load tests. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*. 43–52. DOI: http://dx.doi.org/10.1109/ASE.2011.6100093

[60] M. Zhigulin, S. Prokopenko, and M. Forostyanova. 2012. Detecting faults in TFTP implementations using Finite State Machines with timeouts. In *Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering*. Russia.