SPD: Automatically Test Unmodified Network Programs with Symbolic Packet Dynamics

Wei Sun, Lisong Xu, Sebastian Elbaum
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA 68588-0115
Email: {wsun, xu, elbaum}@cse.unl.edu

***Abstract*—Network programs are difficult to test, especially under the large space of network program behavior defined by packet dynamics such as packet delay and packet loss. It is unlikely for common approaches using testbeds with random packet dynamics to cover the prohibitively large number of packet dynamics possibilities in a limited time.**

In this paper, we leverage symbolic execution to find the equivalence classes of packet dynamics which lead to exactly the same network program behavior, in order to efficiently check the large space of network program behavior. Specifically, we propose and develop a network program test platform, called Symbolic Packet Dynamics (*SPD*), to automatically and transparently test unmodified network programs for low-probability bugs and extreme-case performance under packet dynamics. *SPD* uses symbolic representation of packet dynamics, instead of random packet dynamics.

Our experiments show that *SPD* can achieve significantly much higher packet dynamics coverage than random packet dynamics within the same amount of time, and thus it is much easier and takes much shorter time for *SPD* to detect low-probability bugs and extreme-case performance.

I. Introduction

Network programs including the implementations of network protocols and network applications are difficult to test, especially under the large space of network program behavior defined by packet dynamics [1] such as packet delay and packet loss. This is because there are a prohibitively large number of packet dynamics possibilities, and many bugs are revealed only in corner cases with low probabilities, such as certain packet delay or reordering patterns. For example, most of the Linux TCP bugs recently found by Google [2] are related to low-probability packet dynamics. As another example, a time synchronization bug led to a three-day network outage of a volcano monitoring sensor network [3], which was caused by low-probability packet dynamics in the extreme volcano environment.

A common way to test network programs is to apply testbed experiments with emulated random packet dynamics [4], [5]. Although using random packet dynamics is effective to exercise most common situations, it is impractical to check all possible packet dynamics. For example, considering only two packets each with an independent delay range (0, 2000] ms, there are already a total of $2000 * 2000 = 4,000,000$ different packet delay combinations assuming a 1-ms time resolution [6]. It is unlikely for approaches using random packet dynamics to cover all these possibilities.

To efficiently check the large space of network program behavior defined by packet dynamics, we apply an advanced technique called symbolic execution from the software verification community. With significant recent advances [7] in constraint solving, scalable dynamic symbolic execution, and computer virtualization, symbolic execution has been successfully used in testing GNU utilities [8], Linux and Windows binaries [9], device drivers [10], and large systems such as full operating systems [11] and network systems [12]. By systematically and automatically exploring program paths, symbolic execution can generate high-coverage test cases for programs.

A simple symbolic execution example works as follows. First, a symbolic execution engine executes a tested program on an input variable, for example var, which is assigned to a "symbolic" value with an initial constraint, for example, $var \in (0, 2000]$, instead of a specific concrete value in a normal execution. Every time when the engine reaches a branching statement, for example, `if` ($var \leq 500$), it forks the current execution state into two states: one of which follows the true branching path with a new constraint $var \leq 500$, and the other one follows the false branching path with a new constraint $var > 500$. The engine repeatedly selects a state to execute and forks a new state when necessary until enumerating all possible execution paths of the program for the given symbolic variable. The engine accumulates a set of constraints along each program execution path. A set of constraints defines an equivalence class of the symbolic variable, because all values satisfying the constraints follow exactly the same execution path. Therefore, in order to check all possible execution paths of a network program, we only need one value for each equivalence class instead of an extremely large number of all possible values. This simple example is for illustration purpose, but principles are general.

In this paper, we propose and develop a network program test platform, called Symbolic Packet Dynamics (*SPD*), to automatically and transparently test unmodified network programs for low-probability bugs and extreme-case performance under packet dynamics. In order to efficiently check the large space of network program behavior defined by packet dynamics, our proposed *SPD* uses symbolic representation of packet dynamics, instead of random packet dynamics. *SPD* automatically and transparently finds the equivalence classes of packet dynamics for a network program. Our experiments also demonstrate the main feature of *SPD*: It can achieve considerably higher packet dynamics coverage than random packet dynamics within the same amount of time. Therefore, *SPD* can detect low-probability bugs and extreme-case performance much more quickly and easily.

II. Background and related work

A. Background

The core idea of symbolic execution is to run a program using a symbolic execution engine, such as KLEE [8]. The engine runs the program on inputs with symbolic values instead of concrete values. Figure 1 shows one example, where line 2 assigns variable t to symbolic value $\triangle t_1$ that has an initial constraint $\triangle t_1 \in (0, 2000]$ (i.e., line 1). Once the execution reaches the first branching statement that is the `if` ($t_1 \leq 1000$) statement at line 3, the engine checks both execution paths by forking into two copies. One follows the true branching path with an accumulative constraint $\triangle t_1 \in (0, 1000]$ (referred to as path 1 in the figure), and the other one follows the false branching path with an accumulative constraint $\triangle t_1 \in (1000, 2000]$ (referred to as path 2 in the figure). Path 1 further forks into two copies at line 4, and path 2 forks into two copies at line 11.

The symbolic execution tree in the middle of Fig. 1 shows all four possible execution paths as well as their corresponding accumulative constraints. For example, path 3 has an accumulative constraint $\triangle t_1 \in (500, 1000]$, which is the combination of the initial constraint $(0, 2000]$ at line 1, the true branching constraint of `if` ($t_1 \leq 1000$) statement at line 3, and the false branching constraint of `if` ($t_1 \leq 500$) statement at line 4. Sometimes, an accumulative constraint is infeasible (i.e., unsolvable), and then the corresponding path is killed by the engine. For example, path 2 has an infeasible accumulative constraint ($t_1 > 1000$) and ($t_1 \leq 700$).

Finally, the accumulative constraint of each feasible path describes an equivalence class of $\triangle t_1$ leading to the same execution path, and can be solved for a concrete value to generate a test case for covering the corresponding path or reproducing bugs. For example, $\triangle t_1 \in (500, 1000]$ (i.e., testcase 2 in Fig. 1) is an equivalence class following the same execution path (i.e., path 3), and 600 is a specific concrete value for traversing path 3.

B. Related Work

The correctness of the implementations of network applications and protocols (e.g., the complex TCP) can be tested using expert-written scripts (e.g., PacketDrill [2] by Google TCP experts), using rigorous protocol specifications [13], or by passively checking packet traces [14].

Given a model of a system, model checking exhaustively checks whether this model meets a given property. Due to the event-driven nature of network programs, implementation-level model checking has been used to test various types of

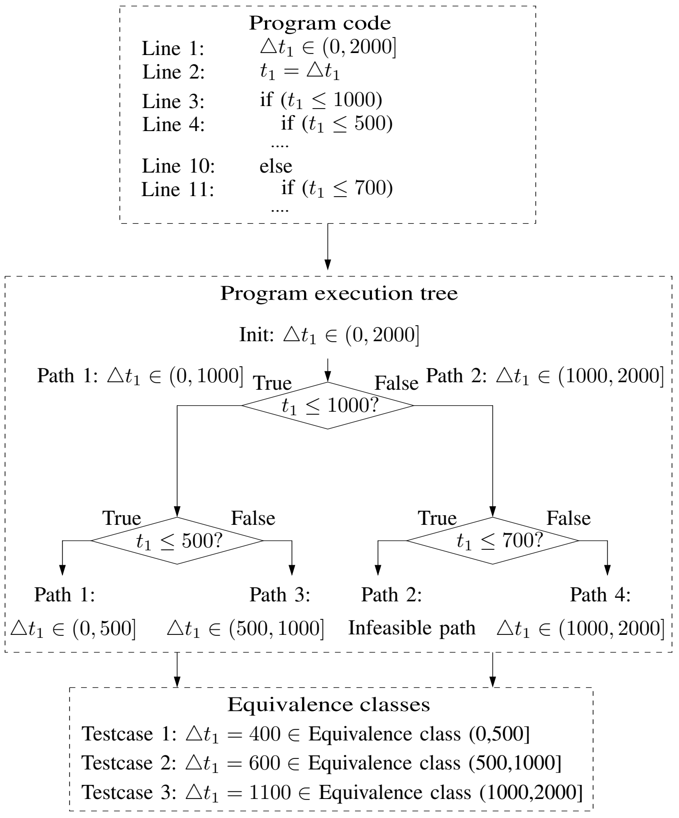
Program code
Line 1: $\triangle t_1 \in (0, 2000]$
Line 2: $t_1 = \triangle t_1$
Line 3: if ($t_1 \leq 1000$)
Line 4: if ($t_1 \leq 500$)
....
Line 10: else
Line 11: if ($t_1 \leq 700$)
....
Program execution tree
Init: $\triangle t_1 \in (0, 2000]$
Path 1: $\triangle t_1 \in (0, 1000]$
True
False
Path 2: $\triangle t_1 \in (1000, 2000]$
$t_1 \leq 1000?$
True
False
$t_1 \leq 500?$
True
False
$t_1 \leq 700?$
Path 1: $\triangle t_1 \in (0, 500]$
Path 3: $\triangle t_1 \in (500, 1000]$
Path 2: Infeasible path
Path 4: $\triangle t_1 \in (1000, 2000]$
Equivalence classes
Testcase 1: $\triangle t_1 = 400 \in$ Equivalence class (0,500]
Testcase 2: $\triangle t_1 = 600 \in$ Equivalence class (500,1000]
Testcase 3: $\triangle t_1 = 1100 \in$ Equivalence class (1000,2000]

Fig. 1. A symbolic execution example.

network programs. NICE [15] tests OpenFlow applications. CMC [16] tests Linux TCP implementation. T-Check [17] tests sensor networks for safety and liveness properties. MaceMC [18] and CrystalBall [19] test distributed systems implemented using the Mace framework. MoDist [20] and dBug [21] attempt to transparently test distributed systems.

Symbolic execution runs program code symbolically and can generate high-coverage test cases for a large input space. It has been used to detect problematic message sequences and frequencies in distributed systems [22], to discover manipulation attacks in network protocol implementations [23], and to check the data plane properties of network programs [24]. Cloud9 [25] uses parallel symbolic execution to test web servers. In addition, packets with symbolic fields in content have been used to test network programs, such as in DiCE [26], SymbexNet [27], NICE [15] and SOFT [28].

There is, however, very little work on the symbolic representation of packet dynamics. KleeNet [12] tests a sensor network under symbolic packet loss and duplication. SymTime [6] extends KleeNet to symbolic packet delay, and is an initial attempt to test network programs with symbolic time values and thus is closely related to our work. Both share the idea of executing a network event at a range of times instead of at a concrete time. Our approach, however, is based on a virtual machine based S^2E [11], which can provide a complete program running environment. That is, it requires no or minimal efforts to model an environment for the tested programs, and this is one of the reasons that our proposed

SPD can transparently test a network program. Furthermore, the selective feature of S^{2}E does not require all the code to be symbolically executed, and thus can potentially save the computing time and resources.

III. System Overview

A. Design Goal

SPD is designed to automatically find equivalence classes of packet dynamics by transparently checking unmodified application-layer network programs. A tested network program is referred to as a Network Program Under Test (NPUT). To have high usability, *SPD* is designed to be transparent to an NPUT. That is, a user does not need to modify the source code of an NPUT, and actually *SPD* does not even require the source code of an NPUT. The obtained equivalence classes of packet dynamics can be used for detecting low-probability bugs or extreme-case performance. For example, if an assertion or exception in an NPUT catches a bug or some abnormal performance, *SPD* can automatically find the equivalence class of packet dynamics leading to that assertion or exception.

Packet dynamics is represented using symbolic values, and *SPD* supports two popular types of packet dynamics: packet delay and packet loss. A user can specify the range of each individual packet using *SPD* and without modifying the NPUT. For example, the delay of some packet is within the range of $(a, b]$ ms. Packet loss is modeled as a special case of packet delay, and when the delay of a packet is longer than a threshold, it is considered as lost.

B. Overall Design

In order to automatically find equivalence classes of packet dynamics, *SPD* runs an NPUT with the Selective Symbolic Execution (S^{2}E) platform [11] and uses symbolic drivers and a central scheduler to implement symbolic representation of packet dynamics. In order to transparently check unmodified application-layer network programs, *SPD* intercepts all network-related and time-related system calls of an NPUT instead of directly modifying the source code of an NPUT.

SPD consists of four components: (1) S^{2}E; (2) symbolic drivers; (3) central scheduler; and (4) communication channels. The first component S^{2}E was proposed in [11], and the other three components are our contributions and they run in a single virtual machine managed by S^{2}E. The logical architecture of *SPD* is illustrated in Fig. 2, where we consider a simple NPUT with only one client and one server to simplify the discussion. This architecture can be easily extended to other common communication schemes (e.g., peer to peer, multiple clients and servers). Below, we explain the four components one by one.

Selective symbolic execution platform: We choose to use S^{2}E (version 1.3) [11] in *SPD*. S^{2}E is a selective symbolic execution platform, which can symbolically execute the selected code in a virtual machine. Specifically, S^{2}E runs a virtual machine that is interpreted by a Quick Emulator (QEMU) dynamic translator augmented with symbolic execution engine KLEE [8]. Any part of the virtual machine can be selected to run symbolically on specified symbolic variables. S^{2}E can automatically explore all feasible execution paths and find the equivalence classes of the symbolic variables. In *SPD*, an NPUT is executed on the virtual machine managed by S^{2}E. However, S^{2}E alone cannot test the NPUT under symbolic packet dynamics, because packet dynamics are not present in the NPUT code. Therefore, we propose and implement symbolic drivers and central scheduler, which together define symbolic variables to represent packet dynamics and emulate the impact of packet dynamics on the NPUT. The central scheduler is selected to execute symbolically.

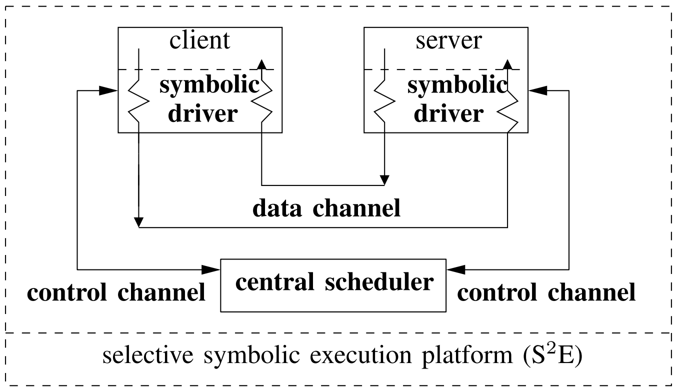

Fig. 2. Logical architecture of *SPD* where components in the bold font indicate our contribution.

Symbolic drivers: Symbolic drivers and the central scheduler together provide an NPUT with the illusion of communicating over a network with symbolic packet dynamics. There are two symbolic drivers, one for the client and the other for the server. They intercept all network-related and time-related system calls so that an NPUT can be transparently tested without the need to modify the source code. Specifically, the drivers create an event for each intercepted system call to keep track of the associated parameters and data (e.g., packet buffer, timeout duration). *SPD* currently supports three types of events: (1) packet departure; (2) packet arrival; and (3) timer expiration. The drivers send all events to the central scheduler through control channels.

Central scheduler: The central scheduler is responsible for controlling symbolic drivers and dispatching events globally. It is a generic discrete event scheduler with a global virtual clock and a global event list. It has a reliable channel to receive events from symbolic drivers. When a new event arrives, the scheduler associates it with an expected execution timestamp. For example, if a packet arrival event will occur after a symbolic packet delay of $\triangle t_1 \in (0, 2000]$ ms, its expected execution timestamp is $current\_clock\_time + \triangle t_1$ ms, where $current\_clock\_time$ is the global virtual clock. As another example, if a timer will expire after a concrete timeout period 900 ms, its expected execution timestamp is $current\_clock\_time + 900$ ms. Every time when the global event list is not empty, the central scheduler finds an event with the earliest expected execution timestamp, and then dispatches the event to the corresponding symbolic driver and updates the global virtual clock. After dispatching an event, the scheduler waits for a certain amount of time (called the synchronization waiting period) so that the event is executed and possibly new events are generated. A performance overhead for the synchronization waiting period is discussed in Section IV.

Communication Channels: As illustrated in Fig. 2, there are data channels between the client and the server for transferring data, and control channels between the central scheduler and the client or the server for central coordination. The channels should be reliable and controllable for two usages. One is that the channels are used by the client or server to notify a scheduled event to the central scheduler for updating the global event list. Another is that the central scheduler also needs to signal the dispatched event to the corresponding symbolic driver via the channel for controlling the event execution in the symbolic driver. Therefore, the speed and reliability of the channels are crucial for the performance of the whole framework. We detail on channel implementation in Section IV.

IV. Implementation and Evaluation

A. Implementation

The prototype implementation is illustrated in Fig. 3, and below we describe some implementation details of *SPD*.

System call interception: Symbolic drivers intercept all network-related and time-related system calls. Specifically, a hook (e.g., dynamical shared library) is used to intercept a library call and call other wrapper code. The wrapper code in the drivers is used to collect associated parameters and data with the calls. An event data structure is used to keep track of all information of an event. Through system call interception, *SPD* does not need to change the source code of the socket programming library or to recompile the library's source code, which is laborious and prone to error [11]. $LD\_PRELOAD$ environment variable is used to give the symbolic drivers a high priority to load before anything else. Then the binary of an NPUT is linked dynamically with the symbolic drivers. By doing so, *SPD* is transparent for the NPUT and thus can even test a binary directly without its source code.

TABLE I
INTERCEPTED NETWORK-RELATED SYSTEM CALLS

socket()	bind()	select()	poll()
sendto()	recvfrom()	listen()	accept()
connect()	write()	read()	recv()
send()	sendmsg()	recvmsg()	close()

Each type of events has a corresponding event handler implemented by the symbolic drivers. The event handlers for packet departure events and packet arrival events are to simply transfer packets through the communication channel. The intercepted network-related system calls are summarized in Table I. The event handler for timer expiration events is much more complicated as network programs may use different timer mechanisms. Three intercepted time-related system calls are summarized in Table II. Disabling timers (e.g., alarm(0)) is also supported in our prototype.

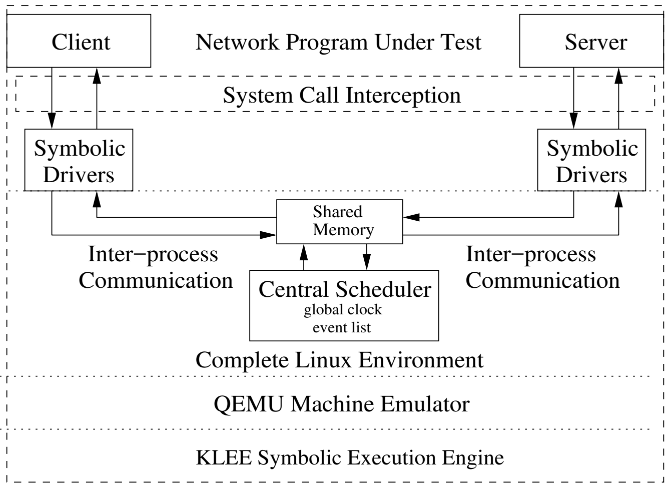

Fig. 3. Prototype implementation of *SPD*.

TABLE II
INTERCEPTED TIME-RELATED SYSTEM CALLS

Type	System call	Event handler
signal based	alarm()	generate a $SIGALRM$ signal
return value based	select()	return a value of zero
clock based	gettimeofday()	update clock later than expected

Communication channels: Inter-process communication with shared memory is used for providing communication channels between the client or server and the central scheduler for two reasons. One reason is that inter-process communication is faster than the entire TCP/IP networking stack with over 30 thousands lines of code [25]. Another reason is that the client, the server and the central scheduler can share the global clock and the global event list more easily through the shared memory, especially when to access a symbolic variable. The communication channels are essentially producer-consumer queues of events, with a function to notify an event to multiple listeners. To reduce the large overhead of using semaphores or monitors to access the shared memory, a single producer and single consumer pattern is used without the need for end-to-end atomic synchronization.

Synchronization waiting period: The length of the synchronization waiting period for the central scheduler influences the overall performance of *SPD*. A shorter period improves the performance, however, a shorter period has a higher possibility of missing new events just generated. *SPD* chooses a relatively long and conservative time, and implements an adaptive mechanism to adaptively select an appropriate value for different kinds of network programs.

B. Evaluation

We demonstrate the capability of *SPD* using an implementation[1] of Trivial File Transfer Protocol (TFTP). TFTP is a delay-sensitive protocol and is widely used for bootstrap loading. TFTP works on top of unreliable UDP, and uses the stop-and-wait method to reliably transfer a file. Fig. 4

[1]https://github.com/lanrat/tftp

demonstrates how the program works, where a file with the size of two data packets is uploaded from the TFTP client to the TFTP server. The client first sends one control packet and then two data packets to the server. After receiving the ack to the last transmitted packet, the client sends out the next packet. The timeout duration is set to 500 milliseconds and the maximum number of timeouts is set to 3.

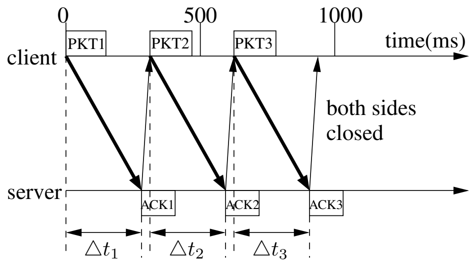

Fig. 4. Programs run correctly.

Packet dynamics coverage: We run two groups of experiments to compare the packet dynamics coverage of two different techniques: 1) experiments using *SPD*, 2) experiments using random packet dynamics (*RPD*). The delay of every packet from the server to the client is fixed to 10 ms in both groups of experiments. The delay of each packet from the client to the server is set to a symbolic value in the range of $(0, 1000]$ milliseconds in the *SPD* experiments, and it is set to a random value in the same range in the *RPD* experiments.

Fig. 5 shows the number of equivalence classes of packet dynamics covered by these two groups of experiments, as we increase the file size from 1 data packet to 5 data packets. For each file size, we repeat the *RPD* experiments for 10,000 times, and we run a single *SPD* experiment for the same amount of time as the total running time of these 10,000 *RPD* experiments.

An equivalence class of packet dynamics is defined as a set of packet dynamics all leading to the same sequences of events (i.e., packet departure, packet arrival, and timer expiration) in the TFTP execution, because network programs are event-based programs and different sequences of events generally lead to different execution paths.

We can see that *SPD* covers much more number of equivalence classes of packet dynamics than *RPD*, and the difference between them increases with the increase of the file size. We can also see that most of the *RPD* experiments cover duplicate packet dynamics equivalence classes. A detailed analysis of the *RPD* experiments shows that the second 5,000 *RPD* experiments cover only a few new equivalence classes. In addition, if we repeat the *RPD* experiments for a longer time, we can only very slowly increase the coverage, because most of those uncovered equivalence classes are low-probability equivalence classes. On the contrary, a longer running time of a *SPD* experiment can quickly increase its coverage if *SPD* has not covered all possible equivalence classes yet.

Bugs detected: For each equivalence class of packet dynamics, we compare the received file at the server with the original file at the client to detect reliability bugs due to packet dynamics. Once we have these packet dynamics, it still requires a manual process with knowledge of the source code to locate the exact lines of buggy code. Because *SPD* can cover more number of equivalence classes of packet dynamics than *RPD* within the same amount of time, it is easier and faster to locate the exact lines of buggy code for developers with the help of *SPD*. Table III summarizes the reliability bugs that we have found for the TFTP implementation. These bugs appear to be real bugs, we have contacted the developer for correction.

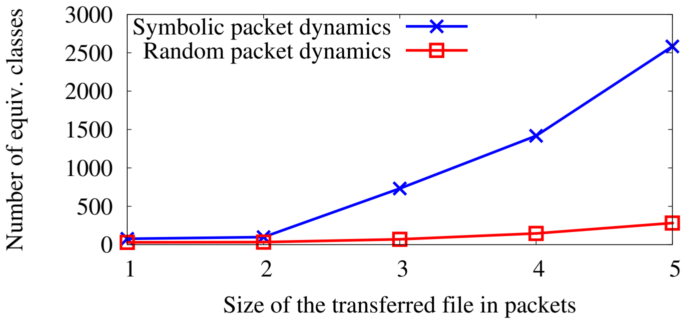

Fig. 5. Equivalence classes of packet dynamics covered by *SPD* and *RPD* within the same amount of time.

TABLE III
BUGS SUMMARY

Bugs	Reasons and impacts
1	a local variable $timeout\_counter$ in $sendfile$ function is not initialized to zero and different compilers have uncertain default values, which fails the retransmission mechanism.
2	an empty pointer $filename$ is passed when the server calls $recvfile$ function, which fails the establishment of connection.
3	the client cannot correctly handle the last data packet with the size $n\ ! = MAX\_DATA\_SIZE$ when it is lost, which fails the termination of connection.

Extreme-case performance: In these groups of experiments, we demonstrate that *SPD* can be used to detect extreme-case performance. We measure the elapsed time that the server stays in a connection, which is the time duration between the arrival time of the first incoming packet and the departure time of the last outgoing packet from the server, in the same experiment scenario as above. This information is important for capacity planning of the TFTP server. We first run the *RPD* experiments for 10,000 times and find the longest elapsed time. Then we add an assertion into *SPD* to check whether the elapsed time is less than or equal to the longest one found by *RPD*. If there exists an equivalence class of packet dynamics leading to an elapsed time longer than that of *RPD*, *SPD* generates a concrete test case for it. Fig. 6 shows that except the initial case with the same result, *SPD* can always find longer elapsed time than *RPD*, and their difference increases with the increase of the file size.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented *SPD* to transparently check unmodified application-layer network programs and automat-

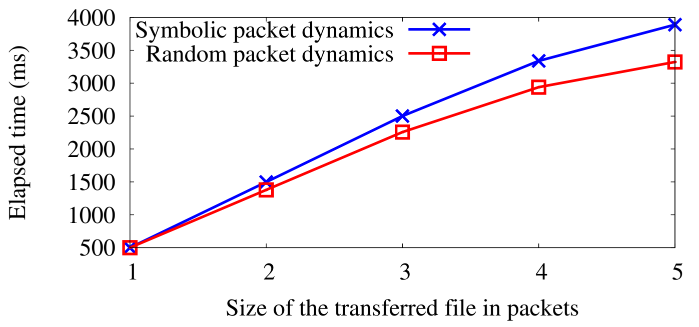

Fig. 6. Longest server elapsed time measured by *SPD* and *RPD* within the same amount of time.

ically find equivalence classes of packet dynamics. The core idea of *SPD* is to test network programs under the symbolic representation of packet dynamics instead of random packet dynamics. Our experiments also illustrate *SPD*'s ability to obtain considerably higher coverage of packet dynamics equivalence classes than random packet dynamics within the same amount of time. Thus, it is much easier and faster to detect low-probability bugs and extreme-case performance with the help of *SPD*.

In the future, we plan to further improve the scalability of *SPD*, so that we can test much larger network programs with a large number of packets. Specifically, we plan to use path reduction and state merging techniques to alleviate the path explosion problem [7] of symbolic execution. That is, the number of paths explored by symbolic execution is usually exponentially proportional to the number of branching statements in the code. This problem becomes worse for the virtual machine based S^2E as more information (e.g., virtual machine state) needs to be stored for each different path. We used a machine with 16-GB RAM in our experiments, and it can only explore about 25,000 different paths before it is terminated due to memory outage.

VI. Acknowledgement

The work presented in this paper was supported in part by NSF 1526253 and UNL Research Council Grants-in-Aid.

References

[1] V. Paxson, "End-to-end Internet packet dynamics," in *Proceedings of ACM SIGCOMM*, France, September 1997.

[2] 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.

[3] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of USENIX OSDI*, Seattle, WA, November 2006.

[4] S. Hemminger, "Network emulation with NetEm," in *Proceedings of the 6th Australia's National Linux Conference*, Australia, April 2005.

[5] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Measurement and Modeling of Computer Systems*, 1998, pp. 151–160.

[6] O. Dustmann, "Symbolic execution of discrete event systems with uncertain time," *Lecture Notes in Informatics*, vol. S-12, pp. pp. 19–22, 2013.

[7] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. pp. 82–90, February 2013.

[8] 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.

[9] 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.

[10] M. Renzelmann, A. Kadav, and M. Swift, "SymDrive: testing drivers without devices," in *Proceedings of USENIX OSDI*, Hollywood, CA, October 2012.

[11] 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.

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

[13] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets," in *Proceedings of ACM SIGCOMM*, Philadelphia, PA, August 2005.

[14] P. Mouttappa, S. Maag, and A. Cavalli, "Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols," *Computer Networks*, vol. 57, no. 15, pp. pp. 2992–3008, October 2013.

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

[16] M. Musuvathi and D. Engler, "Model checking large network protocol implementations," in *Proceedings of USENIX USDI*, San Francisco, CA, March 2004.

[17] P. Li and J. Regehr, "T-Check: Bug finding for sensor networks," in *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, April 2010.

[18] C. Killian, J. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proceedings of USENIX NSDI*, Cambridge, MA, April 2007.

[19] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems," in *Proceedings o USENIX NSDI*, Boston, MA, April 2009.

[20] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MoDist: Transparent model checking of unmodified distributed systems," in *Proceedings of USENIX NSDI*, Boston, MA, April 2009.

[21] J. Simsa, R. Bryant, and G. Gibson, "dBug: Systematic evaluation of distributed systems," in *Proceedings of USENIX SSV*, Canada, October 2010.

[22] C. Lucas, S. G. Elbaum, and D. S. Rosenblum, "Detecting problematic message sequences and frequencies in distributed systems," in *Proceedings of ACM OOPSLA*, Tucson, AZ, October 2012.

[23] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of ACM SIGCOMM*, Toronto, Canada, August 2011.

[24] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of USENIX NSDI*, Seattle, WA, April 2014.

[25] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of EuroSys*, Salzburg, Austria, April 2011.

[26] M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, and O. Crameri, "Toward online testing of federated and heterogeneous distributed systems," in *Proceedings of USENIX ATC*, Portland, OR, June 2011.

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

[28] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A SOFT way for OpenFlow switch interoperability testing," in *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Nice, France, December 2012.