Efficient Protocol Testing Under Temporal Uncertain Event Using Discrete-event Network Simulations

MINH VU and LISONG XU, University of Nebraska–Lincoln
SEBASTIAN ELBAUM, University of Virginia
WEI SUN, Facebook Inc.
KEVIN QIAO, University of Maryland

Testing network protocol implementations is difficult mainly because of the temporal uncertain nature of network events. To evaluate the worst-case performance or detect the bugs of a network protocol implementation using network simulators, we need to systematically simulate the behavior of the network protocol under all possible cases of the temporal uncertain events, which is time consuming. The recently proposed Symbolic Execution based Interval Branching (SEIB) simulates a group of uncertain cases together in a single simulation branch and thus is more efficient than brute force testing. In this article, we argue that the efficiency of SEIB could be further significantly improved by eliminating unnecessary comparisons of the event timestamps. Specifically, we summarize and present three general types of unnecessary comparisons when SEIB is applied to a general network simulator, and then correspondingly propose three novel techniques to eliminate them. Our extensive simulations show that our techniques can improve the efficiency of SEIB by several orders of magnitude, such as from days to minutes.

CCS Concepts: • **Computing methodologies → Discrete-event simulation;**

Additional Key Words and Phrases: Network protocol testing, symbolic execution, temporal uncertainty, discrete event simulator

ACM Reference format:
Minh Vu, Lisong Xu, Sebastian Elbaum, Wei Sun, and Kevin Qiao. 2022. Efficient Protocol Testing Under Temporal Uncertain Event Using Discrete-event Network Simulations. *ACM Trans. Model. Comput. Simul.* 32, 2, Article 13 (February 2022), 30 pages.
https://doi.org/10.1145/3490028

1 INTRODUCTION

Testing network protocol implementations is difficult mainly because of the temporal uncertain nature of network events. An event is called a *temporal uncertain event* (or just *uncertain event* for

This article is a journal extension of our paper published at IEEE INFOCOM 2019 [49] with reorganized pseudocode, more illustrative examples, techniques to handle TCP timeout events, and new experiment results.
This work was supported in part by NSF CNS-1526253, NSF SHF-1718040, and NSF CCF-1918204.
Authors' addresses: M. Vu and L. Xu, School of Computing, University of Nebraska-Lincoln, Lincoln, NE 68588-0115, USA; emails: minh.vu@huskers.unl.edu, xu@cse.unl.edu; S. Elbaum, Computer Science Department, University of Virginia, Rice 423, Charlottesville, Virginia 22904, USA; email: selbaum@virginia.edu; W. Sun, 1 Facebook Way, Building 21, Menlo Park, CA 94025, USA; email:wesun@fb.com; K. Qiao, Department of Computer Science, University of Maryland, 8125 Paint Branch Drive, College Park, MD 20742, USA; email: kqiao@umd.edu.

1049-3301/2022/02-ART13 $15.00
https://doi.org/10.1145/3490028

short) if it may occur at any time in an interval instead of a single time instant. For example, the arrival event of a packet at a node is an uncertain event if the packet may experience an uncertain delay in a network. Many network protocol issues are related to uncertain events and can be detected only in corner cases with low probabilities. For example, most of the recently found TCP issues [13] are related to low-probability uncertain events. As another example, many issues of wireless sensor networks [37] are related to low-probability uncertain events, and they are hard to detect and costly to fix once deployed in the fields.

In this article, we consider network protocol testing methods using network simulators, such as NS-3 [31], because network simulators are popular tools for testing network protocols. Specifically, we consider the testing methods that use network simulators to simulate and check the behaviors of a network protocol under all possible cases of uncertain events. These testing methods are useful for evaluating the worst-case performance a network protocol and detecting the design and implementation bugs of a network protocol in corner cases. It is, however, very challenging to check the behaviors of a network protocol under all possible cases of uncertain events. Let us consider a network protocol with n packets, each experiencing k possible packet delays. For this example, we need to simulate and check the behavior of the network protocol for all k^n possible uncertainty cases (i.e., packet delay combinations), which is time consuming as k is usually very large. For instance, for a packet delay in (0, 1] second, k is 10^3 with a millisecond resolution, and 10^6 with a microsecond resolution.

State of the art. There are two general classes of network protocol testing methods using network simulators: (1) *random testing* simulates randomly selected uncertainty cases, such as random packet delays according to a distribution; (2) *systematic testing* aims to enumerate and simulate all uncertainty cases. Random testing is more suitable for evaluating the performance of a network protocol in normal cases, whereas systematic testing is more suitable for checking the behavior of a network protocol under all possible cases.

Systematic testing methods can be further classified into two categories. One category is *brute force testing*, which separately simulates and checks a network protocol for each uncertainty case. It is inefficient, but it is simple and can be used for any general network simulator. A second category is *interval branching* [34], which simulates multiple uncertainty cases together by associating the timestamp of a simulation event with an interval. The interval of an event indicates the set of all possible occurrence times of the event, and two events overlap if the intersection of their intervals is not empty. When overlapping, interval branching forks to cover all possible occurrence orders of the events, and each branch continues with updated timestamp intervals for the involved events. Therefore, interval branching is more efficient than brute force testing.

Interval branching can be implemented in two different ways. The first way is *direct interval branching*. Interval branching was originally implemented by directly modifying a simulator [34]; however, it requires substantial changes to the simulator, especially nontrivial work for forking. This cumbersome implementation has considerably slowed the adoption of interval branching by the networking community. The second way is **Symbolic Execution based Interval Branching (SEIB)**: interval branching has been recently implemented [18, 45, 46] by leveraging symbolic execution [10], a popular program analysis technique in the software testing and verification community. Conceptually, a tester declares the timestamp variable of an event in a simulator as a symbolic variable that can take multiple values, then executes the simulator using a symbolic execution engine that automatically takes care of forking the simulator when comparing overlapping symbolic variables. As SEIB greatly simplifies the work to fork the simulator and manage multiple copies of the simulator, it is more likely to be widely adopted than direct interval branching.

The efficiency (e.g., testing time and consumed memory) of SEIB mainly depends on the total number of generated SEIB branches, which in turn mainly depends on the total number of

comparisons of overlapping timestamps in the simulation. Specifically, the number of branches is approximately an exponential function of the number of comparisons of overlapping timestamps, and thus it is still time consuming for SEIB to check the behavior of a network protocol under all possible cases.

Our work. In this article, we argue that the efficiency of SEIB could be significantly improved by eliminating unnecessary comparisons of overlapping timestamps. By doing so, we can significantly reduce the number of generated branches and then the testing time and consumed memory. This is because an unnecessary comparison, if not eliminated, generates a new branch that may continuously fork and generate new branches. Specifically, we summarize and present three general types of unnecessary comparisons when SEIB is applied to a general network simulator (e.g., NS-3), then correspondingly propose three techniques to modify the simulator to eliminate these unnecessary comparisons.

The first type is the unnecessary comparisons due to *simultaneous events*. We find that a simulator may compare the timestamps of two events for multiple times instead of once, to check the special case where two events happen at exactly the same time instant. These unnecessary comparisons can be eliminated by reorganizing the comparison code of the simulator.

The second type is the unnecessary comparisons due to *conditional ineffective events*. We find that a simulator may have various types of conditional ineffective events, which have no impact on the simulation result under some conditions but their timestamps are unnecessarily compared with other events, such as an uncertain event that might happen after the end of a simulation, and a TCP retransmission timeout event that might be cancelled by an uncertain ACK. These unnecessary comparisons can be eliminated by identifying when these conditional ineffective events become ineffective and then removing them from the simulation.

The third type is the unnecessary comparisons due to *independent events*. Two events on different network nodes are independent if they do not have any impact on each other. As a result, two independent events can be executed in any order in a simulation, and it is not necessary to compare their timestamps. The general idea of exploring independent events to speed up software testing and verification (e.g., in model checking) is not new. The novelty of our work is that we apply it to SEIB and propose to eliminate unnecessary comparisons of independent events on different nodes by decomposing the network simulation into multiple synchronized node simulations. Our decomposition technique is similar to and inspired by the traditional parallel simulation methods. Different from a parallel simulator that runs on multiple processors with the aim to speed up the parallel simulation, our work still runs on a single processor with the aim to reduce the number of branches.

Our contributions are threefold. First, we propose three novel techniques to significantly improve the efficiency of SEIB for testing the behavior of a network protocol under all possible cases of temporal uncertain events. For each proposed technique, its correctness and efficiency can be formally proved. Second, we modify the popular general network simulator, NS-3, to make it more symbolic-execution-friendly using our proposed techniques. To the best of our knowledge, this is the first time that SEIB is applied to a large general network simulator that has been widely used in the networking community. Third, we evaluate the efficiency of our proposed techniques by comparing the modified NS-3 with the original NS-3 using various network topologies and protocols including TCP, UDP, and IP routing. The results show that when executed by SEIB, the modified NS-3 achieves several orders of magnitude shorter testing times than the original NS-3, such as from days to minutes. Our evaluation also shows that our techniques are several orders of magnitude more efficient than traditional parallel simulation methods when executed by SEIB.

Pseudocode 1: A discrete-event network simulator

```
1:  variables for network state
2:  array: list[ ]                                          ▷ global event list
3:  variable: clock                                         ▷ global clock
4:  function Run
5:      repeat
6:          e ← FindEvent()
7:          ExecuteEvent(e)
8:      until list is empty or e is an end-of-simulation event
9:  function FindEvent
10:     e ← list[0]                                         ▷ The earliest one
11:     Remove list[0] from list
12:     return e
13: function ExecuteEvent(e)
14:     clock ← e.t                                         ▷ Advance the clock
15:     Simulate e (update state variables, generate new events, cancel events)
16:     for each new event new_e generated by e do
17:         InsertEvent(new_e)                              ▷ Insert to list[ ]
```

2 BACKGROUND

2.1 Network Simulation

Network simulation is usually conducted using a discrete-event network simulator, which simulates a network using a sequence of events and updates the simulation variables only when an event occurs.

Pseudocode 1 shows the major data structures and functions of a discrete-event network simulator. It maintains three types of data structures: (1) the network state variables, which describe the current state of the whole network; (2) the global *list* of pending events in the whole network, which are sorted in the ascending order of their timestamps; and (3) the global *clock*, which is the current time in a simulation. The simulator has three major functions. Function *Run* at line 4 repeatedly finds and executes the earliest event *e* in *list* until *e* is the last event or the end of the simulation. Function *FindEvent* at line 9 finds the first event in *list* that is the one with the earliest timestamp to avoid *causal violations*, which happen when a future event affects a past event. Function *ExecuteEvent* at line 13 advances the global *clock* to the timestamp *e.t* of event *e*, updates related network state variables, generates zero or more new events that will be inserted into *list* using function *InsertEvent*, and may cancel some existing events (e.g., timeouts). Function *InsertEvent* (not shown in this pseudocode but will be discussed later) inserts a new event *new_e* to the appropriate position of *list* using some sorting algorithm.

Current discrete-event network simulators, such as NS-3 [31], are not originally designed for symbolic execution that is described in the next section. In this article, we propose to modify discrete-event network simulators using NS-3 as an example to make them more symbolic-execution-friendly using our proposed techniques. We choose NS-3 because it is an open source discrete-event network simulator widely used in the networking community. Currently, NS-3 has approximate 350,000 lines of C++ code, and users can write their own simulation scripts/code in C++ or Python. It supports both sequential simulations on a single processor and parallel simulations on multiple processors. We consider only sequential simulations in this article.

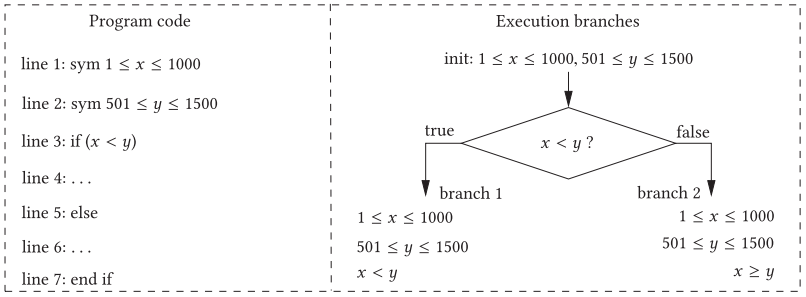

Fig. 1. A symbolic execution example with symbolic variables x and y.

2.2 Symbolic Execution

Instead of running a program directly, symbolic execution [16, 26] runs a program with symbolic variables using a symbolic execution engine. Different from normal program variables that take concrete values, a *symbolic variable* takes a symbolic value represented as symbolic constraints. In other words, a symbolic variable can take all possible values satisfying the symbolic constraints. Figure 1 shows an example. The first two lines of the program declare two symbolic variables x and y with their initial constraints. For example, x can take any integer values between 1 and 1,000. Once the execution reaches an $if(cond)$ statement involving symbolic variables, the symbolic execution engine queries a constraint solver to check the feasibility of both possibilities (i.e., $cond$ = true or false) under the current constraints. For example, for $cond$ = "$x < y$" in line 3, because both possibilities are feasible, the current execution forks into two branches. The true branch continues with additional constraint $x < y$, and the false branch continues with additional constraint $x \geq y$.

Symbolic execution is a powerful technique widely used in the software testing and verification community, because it can automatically divide all possible combinations of the symbolic variable values into equivalence classes. The combinations in the same equivalence class have the same execution path, and they are executed together using the same branch. For Figure 1, there are a total of $1{,}000 \times 1{,}000 = 10^6$ combinations of x and y. Without symbolic execution, we need to execute the program 10^6 times, one for each combination, to check all possible behaviors of the program. With symbolic execution, we execute the program using only two branches. For example, all combinations satisfying constraints $1 \leq x \leq 1{,}000$, $501 \leq y \leq 1{,}500$, and $x < y$ have the same execution path (i.e., lines 1, 2, 3, 4, 7) and are executed together as branch 1.

In this work, we use S^{2}E [14], which is a powerful symbolic execution platform that can symbolically execute NS-3 in a virtual machine. The virtual machine is emulated using the QEMU machine emulator [4], and the symbolic execution is conducted using the KLEE symbolic execution engine [9].

3 SYMBOLIC EXECUTION BASED INTERVAL BRANCHING

This section introduces basic definitions, explains how SEIB works, and discusses the advantages and limitations of current SEIB.

3.1 Definitions and Notation

In this work, we consider only temporal uncertain events caused by uncertain network delays, which are the major uncertainty source for network protocols. In the following, let us consider an example, where two nodes are connected with a link, and a node sends three packets p_i, $i \in [1, 3]$, to the other one.

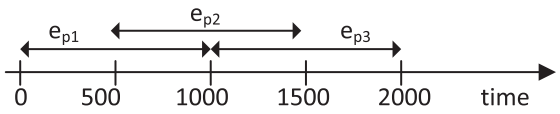

Fig. 2. Example: Three uncertain packet arrival events. The double-headed arrows indicate their timestamp intervals.

For each packet p_i, let d_{p_i} denote its *delay* over the link, and let *delay interval* D_{p_i} denote the set (or range) of all possible values of d_{p_i}. We say that delay d_{p_i} is *uncertain* if D_{p_i} contains more than one value (i.e., $|D_{p_i}| > 1$). The *delay space* $\mathbb{D}$ of a simulation is the cross product of all delay intervals in the simulation, and a vector $\vec{d} \in \mathbb{D}$ is called a *delay vector*. For the example, $\mathbb{D} = D_{p_1} \times D_{p_2} \times D_{p_3}$ is a three-dimensional space, and $\vec{d} = (d_{p_1}, d_{p_2}, d_{p_3})$. Suppose that each packet p_i has the same $D_{p_i} = [1, 1{,}000]$ ms assuming a millisecond resolution, then $|D_{p_i}| = 1{,}000$ and $|\mathbb{D}| = 10^9$. In other words, $\mathbb{D}$ has a total of 10^9 possible delay vectors.

Pseudocode 2: Part of a simulation code for the three-packet example in Section 3

```
1:  function Main
2:      sym 1 ≤ d_p1 ≤ 1000                          ▷ symbolic variable
3:      sym 1 ≤ d_p2 ≤ 1000                          ▷ symbolic variable
4:      sym 1 ≤ d_p3 ≤ 1000                          ▷ symbolic variable
5:      ...
6:      e_p1.t ← 0 + d_p1                            ▷ e_p1.t is symbolic
7:      e_p2.t ← 500 + d_p2                          ▷ e_p2.t is symbolic
8:      e_p3.t ← 1000 + d_p3                         ▷ e_p3.t is symbolic
9:      InsertEvent(e_p1)
10:     InsertEvent(e_p2)
11:     InsertEvent(e_p3)
12:     ...
13:     Run()                                        ▷ Pseudocode 1
14:     ...
15:     assert(checking simulation results)
16: function InsertEvent(new_e)
17:     for k ← 0; k < list.size; k ← k + 1 do
18:         if new_e.t < list[k].t) then             ▷ Forks into two branches, if new_e.t and list[k].t overlap
19:             Insert new_e to position k in list
20:             return
21:         end if
22:     Append new_e to the end of list
```

For each event e in a simulation, let $e.t$ denote its timestamp, and let *timestamp interval* $[e.t]$ denote the set (or range) of all possible values of $e.t$. We say that event e or timestamp $e.t$ is uncertain if $[e.t]$ contains more than one value. To simplify our discussion in this section, let us consider only the arrival events of these packets in the example. For each packet p_i, let e_{p_i} denote its *arrival event* at the destination node and then $e_{p_i}.t$ is the packet arrival time. Suppose that the three packets in the example depart from their source node at 0, 500, and 1,000 ms, respectively, and have the same $D_{p_i} = [1, 1{,}000]$ ms. We have $e_{p_1}.t = 0 + d_{p_1}$, $e_{p_2}.t = 500 + d_{p_2}$, and $e_{p_3}.t = 1{,}000 + d_{p_3}$. Therefore, $[e_{p_1}.t] = [1, 1{,}000]$, $[e_{p_2}.t] = [501, 1{,}500]$, and $[e_{p_3}.t] = [1{,}001, 2{,}000]$, as shown in Figure 2.

We say that two timestamps *overlap* if their timestamp intervals overlap (i.e., nonempty intersection). For example, $e_{p_1}.t$ and $e_{p_2}.t$ overlap because $[e_{p_1}.t] \cap [e_{p_2}.t] = [1, 1{,}000] \cap [501, 1{,}500] = [501,$

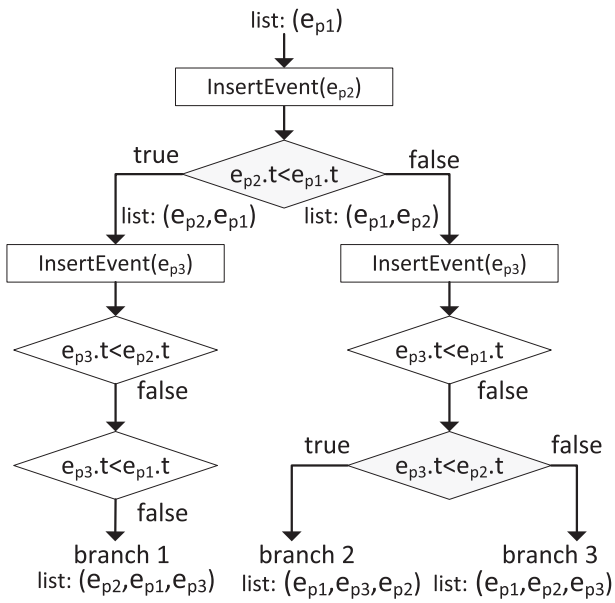

Fig. 3. Three branches generated when Pseudocode 2 and Pseudocode 1 are executed by SEIB, due to two comparisons of overlapping timestamps (shaded).

1,000]. Intuitively, this means e_{p_1} and e_{p_2} may occur in different orders. As another example, $e_{p_1}.t$ and $e_{p_3}.t$ do not overlap, and this means that e_{p_1} and e_{p_3} may occur in only one order.

3.2 SEIB

Both random testing and brute force testing directly run a network simulator, whereas SEIB runs a network simulator with symbolic delay variables using a symbolic execution platform, such as S^2E [14]. SEIB does not change the functions of a network simulator, except declaring the packet delay variables as symbolic variables. Pseudocode 2 illustrates a possible simulation code for the three-packet example with symbolic packet delay variables. The *Main* function (lines 2 through 4) declares each delay d_{p_i} as a symbolic variable with the initial constraints defined according to its delay interval D_{p_i}. As a result, all other variables depending on these symbolic variables are automatically handled as symbolic variables by the symbolic execution engine of SEIB. For example, timestamp $e_{p_1}.t$ assigned in line 6 is also a symbolic variable, and its timestamp interval $[e_{p_1}.t]$ is implicitly defined by the constraints of d_{p_1}. Lines 9 through 11 call function *InsertEvent* (defined at line 16) to insert the events to *list*. Line 13 calls function *Run* defined in Pseudocode 1 to run the simulation. Finally, line 15 checks the simulation results and detects possible bugs.

Figure 3 shows the symbolic execution of lines 10 and 11 of Pseudocode 2, when Pseudocode 2 together with Pseudocode 1 is executed by SEIB. Before executing line 10, we have $list = (e_{p_1})$. When executing line 10, function *InsertEvent* compares whether e_{p_2} happens before e_{p_1} using the if statement at line 18. The symbolic execution engine of SEIB finds out that both possibilities are feasible according to the current constraints. As a result, SEIB forks the current execution into two branches: the true branch continues to line 19 and the false branch to line 20. Each branch then continues with different *list*s (shown in Figure 3) and different updated constraints (not shown in Figure 3).

We can see that the total number of branches depends on the number of comparisons of overlapping timestamps, which are indicated by shaded diamonds in Figure 3. Finally, a total of three branches are generated due to two comparisons of overlapping timestamps. This is because a comparison of non-overlapping timestamps does not generate any new branches. For example, $[e_{p_1}.t] = [1, 1{,}000]$ does not overlap with $[e_{p_3}.t] = [1{,}001, 2{,}000]$, and thus $e_{p_3}.t < e_{p_1}.t$ is always false. Note that when $InsertEvent(e_{p_3})$ is called in branch 1, $[e_{p_3}.t]$ and $[e_{p_2}.t]$ do not overlap anymore and specifically $e_{p_3}.t < e_{p_2}.t$ is always false. This is because the constraints of branch 1 have been updated with additional constraint $e_{p_2}.t < e_{p_1}.t$ after calling $InsertEvent(e_{p_2})$.

3.3 Advantage and Limitation of SEIB

SEIB is more efficient than brute force testing when checking the behavior in all possible cases of uncertain events. For the example, brute force testing needs to run the simulation for a total of $|\mathbb{D}| = 10^9$ times, one for each delay vector by changing lines 2 through 4 of Pseudocode 2 to specific delays. In contrast, SEIB only needs to execute the simulation once with three generated branches, and the assertion at line 15 is checked for each branch. However, the number of SEIB branches is approximately an exponential function of the number of comparisons of overlapping timestamps, because SEIB forks into two branches every time the simulator compares two overlapping timestamps. As a result, the number of SEIB branches still increases quickly and consequently causes poor testing efficiency. Specifically, the more the number of SEIB branches, the longer the total testing time, and the larger the total amount of consumed memory (because each SEIB branch is a virtual machine in S^2E).

4 OUR METHOD

4.1 Overview

Current SEIB works [18, 45, 46] demonstrate the promising potential of SEIB, but they use only small and simple network simulators. For example, SPD [45] writes a toy simulator to simulate only two nodes connected by a link. In this work, for the first time, we apply SEIB to a large, general, and widely used network simulator, NS-3. Specifically, we propose multiple novel techniques to modify some functions of NS-3 to eliminate several general types of unnecessary comparisons of overlapping timestamps to significantly improve the efficiency of SEIB. Note that SEIB declares packet delay variables as symbolic variables but still uses the original NS-3 functions, whereas our proposed techniques declare packet delay variables as symbolic variables and modify some NS-3 functions. Also note that our proposed techniques do not modify the symbolic execution platform.

When describing each proposed technique, we present only the proposed modification against the original NS-3 functions, which are illustrated in Pseudocode 1 and Pseudocode 3 that together define five NS-3 functions: *Run*, *FindEvent*, *ExecuteEvent*, *InsertEvent*, and *Before*. Each proposed technique modifies one or more of these functions. Although all our proposed techniques can be combined together as shown and evaluated in Section 5, we will not present the modified NS-3 functions for the combined techniques because of the page limit and because it is straightforward to combine them.

Our proposed techniques can significantly reduce the number of branches because an unnecessary comparison, if not eliminated, forks the current branch into two branches, each of which continuously forks for all the remaining comparisons of overlapping timestamps. For each proposed technique, we prove its correctness and efficiency. First, a technique is *correct* if the modified simulator always generates the same simulation result as the original one for each delay vector in the delay space. Second, a technique is *efficient* if the SEIB branches of the modified simulator are no more than those of the original one for the same delay space.

Pseudocode 3: Original NS-3 functions: Pseudocode 1 plus the following functions

```
1:  function InsertEvent(new_e)
2:      for k ← 0; k < list.size; k ← k + 1 do
3:          if Before(new_e, list[k]) then
4:              Insert new_e to position k in list
5:              return
6:          end if
7:      Append new_e to the end of list
8:  function Before(e1, e2)
9:      if e1.t < e2.t then
10:         return True
11:     else if e1.t = e2.t then
12:         if e1.id < e2.id then
13:             return True
14:     return False
```

Pseudocode 4: Modified NS-3 function for efficiently handling simultaneous events

```
1:  function Before(e1, e2)
2:      if e1.id < e2.id then
3:          if e1.t ≤ e2.t then
4:              return True
5:      else
6:          if e1.t < e2.t then
7:              return True
8:      return False
```

4.2 Unnecessary Comparisons Due to Simultaneous Events

4.2.1 Simultaneous Events. We say that two events are *simultaneous* if they occur at the same time instant. Simultaneous events are handled differently by different network simulators. NS-3 associates each event with an event ID, which is a concrete and unique number. When comparing two simultaneous events, NS-3 puts the one with a smaller event ID before the other one in the event list. Specifically, NS-3 uses functions *InsertEvent* and *Before* defined in Pseudocode 3 to insert a new event and compare the timestamp of the new event with existing events. Function *InsertEvent* of Pseudocode 3 is the same as that of Pseudocode 2, except it compares the timestamps using function *Before* to handle simultaneous events. We can see that function *Before* defined in Pseudocode 3 compares the timestamps $e_1.t$ and $e_2.t$ of two events e_1 and e_2 twice at lines 9 and 11, respectively.

4.2.2 Our Technique. We propose to modify function *Before* of NS-3 as shown in Pseudocode 4, which compares the timestamps $e_1.t$ and $e_2.t$ only once at either line 3 or line 6. Intuitively, for each pair of $e_1.t$ and $e_2.t$ (i.e., a delay vector), our proposed technique combines potentially two comparisons of $e_1.t$ and $e_2.t$ into a single one (i.e., efficiency) without changing the simulation result (i.e., correctness).

4.2.3 Correctness. The following theorem proves the correctness of our modified NS-3 with proposed Pseudocode 4.

Theorem 1. *The modified NS-3 with Pseudocode 4 always generates the same simulation result as the original NS-3 (i.e., Pseudocode 1 and 3) for each delay vector in the delay space.*

Proof. There are four possible cases. *Case 1*: When $e_1.t < e_2.t$, both the modified and original NS-3 return true. *Case 2*: When $e_1.t > e_2.t$, both return false. *Case 3*: When $e_1.t = e_2.t$ and $e_1.id < e_2.id$, both return true. *Case 4*: When $e_1.t = e_2.t$ and $e_1.id > e_2.id$, both return false. □

4.2.4 Efficiency. For non-overlapping $e_1.t$ and $e_2.t$, both the modified and original NS-3 generate only one branch. The following theorem considers overlapping $e_1.t$ and $e_2.t$.

Theorem 2. *The modified NS-3 with Pseudocode 4 never generates more branches than the original NS-3 (i.e., Pseudocode 1 and 3) for overlapping timestamps.*

Proof. In the general case of overlapping $e_1.t$ and $e_2.t$, the original NS-3 forks twice and generates three branches. For example, if $[e_1.t] = [1, 1{,}000]$ and $[e_2.t] = [501, 1{,}500]$, the original NS-3 generates three branches corresponding to three cases: e_1 occurs before, at the same time, or after e_2. In this case, the modified NS-3 generates only two branches.

A special case for overlapping $e_1.t$ and $e_2.t$ is when one timestamp interval contains only one time instant and is the left end or right end of another timestamp interval. For example, if $[e_1.t] = [1, 1{,}000]$ and $[e_2.t] = [1{,}000]$, the original NS-3 generates two branches. In this special case, the modified NS-3 generates one or two branches depending on their event ids. □

To have a better understanding of the impact of our proposed technique for simultaneous events, let us consider a simplified simulation example, which calls function *Before* on two overlapping timestamps for N_{before} times. The original NS-3 generates $3^{N_{\text{before}}}$ branches, whereas our modified NS-3 generates $2^{N_{\text{before}}}$ branches. Note that this reduction of branches can be potentially achieved using path merging techniques [2, 23, 28, 39] that can automatically merge certain types of branches but require special support from the symbolic execution engine. Our proposed technique reduces the number of branches without requiring any special support from the symbolic execution engine.

4.3 Unnecessary Comparisons Due to Conditional Ineffective Events

A simulator may have various types of conditional ineffective events, which have no impact on the simulation results under some conditions. We have identified two major types of conditional ineffective events in NS-3. First, *an uncertain event that might happen after the end of a simulation.* NS-3 function *Simulator*::*Stop*(t) creates a special end-of-simulation event with timestamp t so that the simulation stops at time t (see line 8 in Pseudocode 1). If the timestamp interval of an uncertain event is sufficiently long, its interval might contain t. In other words, the event may happen before or after t. In the cases when the event happens after t (referred to as a *beyond-the-end event*), it has no impact on the simulation result. However, NS-3 still keeps these beyond-the-end events in $list$, which leads to unnecessary comparisons among these beyond-the-end events. Second, *a timeout event.* NS-3 simulates multiple types of timeout events, such as TCP retransmission timeout events, which might be cancelled by other events. For example, a TCP retransmission timeout event will be cancelled if the corresponding ACK arrives before the timeout. If cancelled, NS-3 only sets a flag of the timeout event to indicate that it is cancelled but still keeps the cancelled timeout event in $list$, which leads to unnecessary comparisons with other events.

4.3.1 Events That Might Happen After the End of a Simulation.

Our technique. We propose to modify function *InsertEvent* of NS-3 as shown in Pseudocode 5, which detects and then discards those beyond-the-end events. Specifically, line 6 checks whether new event *new_e* happens after the end of simulation (i.e., *new_e* happens after current event $list[k]$, and $list[k]$ is an end-of-simulation event). If so, line 7 discards event *new_e* because it has

Pseudocode 5: Modified NS-3 function for efficiently handling events that might happen after the end of a simulation

```
1:  function InsertEvent(new_e)
2:      for k ← 0; k < list.size; k ← k + 1 do
3:          if Before(new_e, list[k]) then
4:              Insert new_e to position k in list
5:              return
6:          else if list[k] is an end-of-simulation event then      ▷ new_e is a beyond-the-end event?
7:              Discard new_e                                       ▷ If so, discard it
8:              return
9:          end if
10:     Append new_e to the end of list
```

no impact on the simulation result. Note that our technique assumes that an end-of-simulation event will never be cancelled in a simulation.

Correctness. The following theorem proves the correctness of our modified NS-3 with proposed Pseudocode 5.

Theorem 3. *The modified NS-3 with Pseudocode 5 always produces the same simulation result as the original NS-3 (i.e., Pseudocode 1 and 3) for each delay vector in the delay space.*

Proof. Although the modified NS-3 discards those beyond-the-end events and the original NS-3 keeps those events in *list*, both of them run the simulation until an end-of-simulation event. As a result, neither of them executes any of those beyond-the-end events, and thus both generate the same simulation result. □

Efficiency. The following theorem proves the efficiency of our modified NS-3 with proposed Pseudocode 5.

Theorem 4. *The modified NS-3 with Pseudocode 5 never generates more branches than the original NS-3 (i.e., Pseudocodes 1 and 3).*

Proof. We consider two cases depending on the number of beyond-the-end events in a simulation. First, if there is no or only one beyond-the-end event, both the modified and original NS-3 generate the same number of branches. Second, if there are at least two beyond-the-end events, the modified NS-3 generates no more branches than the original NS-3. This is because the modified NS-3 discards all these beyond-the-end events and then these events will not be compared among themselves, whereas the original NS-3 keeps all these beyond-the-end events in *list* and then these events will be unnecessarily compared with one another, which might generate more branches. □

Note that our proposed technique does not work for the special cases where the beyond-the-end events are generated and inserted into *list* even before an end-of-simulation event is inserted into *list*. NS-3 simulations usually create an end-of-simulation event using function *Simulator*::*Stop*(t) in simulation scripts/code when initializing the simulations. Therefore, the proposed technique works well for NS-3 simulations.

Note that our proposed technique works for an event *new_e* whose timestamp overlaps with the end of simulation (i.e., might happen before or after an end-of-simulation event). When function *InsertEvent* in Pseudocode 5 handles event *new_e*, it forks into two branches at line 3 when comparing with end-of-simulation event $list[k]$: the event *new_e* associated with the true branch

Pseudocode 6: Modified NS-3 function for efficiently handling cancelled timeout events

```
1: function ExecuteEvent(e)
2:     clock ← e.t
3:     Simulate e (update state variables, generate new events, cancel events)
4:     for each timeout event cancelled_e cancelled by e do
5:         Remove cancelled_e from list
6:     for each new event new_e generated by e do
7:         InsertEvent(new_e)
```

happens before $list[k]$ and is then inserted into $list$, and the event new_e associated with the false branch is a beyond-the-end event and is then discarded.

4.3.2 Timeout Events.

Our technique. A timeout event has three possible states in a simulation: outstanding, cancelled, and expired. For example, once a TCP retransmission timeout event is created with an expiration time t, its state is outstanding until it is cancelled or expired. If TCP receives the corresponding ACK before time t, the timeout event is cancelled; otherwise, the timeout event expires at time t. We propose two techniques to eliminate unnecessary comparisons of a timeout event at the outstanding and cancelled states.

Technique to efficiently handle cancelled timeout events. We propose to modify function *ExecuteEvent* of NS-3 as shown in Pseudocode 6 to remove all cancelled timeout events from $list$. Specifically, line 4 checks whether each timeout event in $list$ has been just cancelled by the simulated event e. If so, line 5 removes the cancelled timeout from $list$. If not removed (as the original NS-3 does), a cancelled timeout remains in $list$ and will be compared with the new events, which in turn leads to unnecessary comparisons.

To better understand the difference between the original NS-3 (i.e., Pseudocodes 1 and 3) and modified NS-3 with Pseudocode 6, let us consider a cancelled timeout event $cancelled\_e$. Let t_{gen} denote the clock time (i.e., value of $clock$) when the event is generated in NS-3, let t_{exp} denote the original expiration time of the event (i.e., the event timestamp $cancelled\_e.t$), and let t_{can} denote the clock time when the event is cancelled. Both the original and modified NS-3 insert event $cancelled\_e$ into $list$ at time t_{gen}. The original NS-3 keeps event $cancelled\_e$ in $list$ until the event becomes the first one in $list$ (i.e., $list[0]$) and then removes the event from $list$ at line 11 of Pseudocode 1. In other words, the original NS-3 removes event $cancelled\_e$ at time t_{exp}. The modified NS-3 removes event $cancelled\_e$ from $list$ at line 5 of Pseudocode 6 as soon as the event is cancelled. In other words, the modified NS-3 removes event $cancelled\_e$ at t_{can}. For example, suppose that a TCP retransmission timeout event is generated at time $t_{gen} = 100$ ms with an expiration time $t_{exp} = 1{,}000$ and is cancelled at $t_{can} = 200$ ms. Both the original and modified NS-3 insert the event into $list$ at $t_{gen} = 100$ ms, but the original NS-3 removes the event at $t_{exp} = 1{,}000$ ms, whereas the modified NS-3 removes the event at $t_{can} = 200$ ms.

Correctness and efficiency. The following two theorems prove the correctness and efficiency of our modified NS-3 with Pseudocode 6.

Theorem 5. *The modified NS-3 with Pseudocode 6 always produces the same simulation result as the original NS-3 (i.e., Pseudocodes 1 and 3) for each delay vector in the delay space.*

Proof. Because a cancelled timeout event $cancelled\_e$ does not have any impact on the simulation result in both the original and modified NS-3, the original and modified NS-3 generate the same simulation result, although they remove the event $cancelled\_e$ at different times. □

Pseudocode 7: Modified NS-3 functions for efficiently handling both outstanding and cancelled timeout events

```
1: function FindEvent
2:     e ← list[0]
3:     if (e is an outstanding timeout event) and (e.waitinglist is not empty) then
4:         tmp_list ← e.waitinglist
5:         e.waitinglist ← empty
6:         for i ← 0; i < tmp_list.size; i ← i + 1 do
7:             InsertEvent(tmp_list[i], 0)                          ▷ Case 1
8:     e ← list[0]                                  ▷ list[0] may be changed
9:     Remove list[0] from list
10:    return e
11: function ExecuteEvent(e)
12:    clock ← e.t
13:    Simulate e (update state variables, generate new events, cancel events)
14:    for each timeout event cancelled_e cancelled by e do
15:        j ← the index of cancelled_e in list
16:        Remove cancelled_e from list                  ▷ Remove cancelled timeout
17:        for i ← 0; i < cancelled_e.waitinglist.size; i ← i + 1 do
18:            InsertEvent(cancelled_e.waitinglist[i], j)           ▷ Case 2
19:    for each new event new_e generated by e do
20:        InsertEvent(new_e, 0)
21: function InsertEvent(new_e, start_index)
22:    for k ← start_index; k < list.size; k ← k + 1 do
23:        if (list[k] is an outstanding timeout event) and (k != 0) then
24:            Append new_e to the end of list[k].waitinglist ▷ Delay comparisons with outstanding timeout
25:            return
26:        else if Before(new_e, list[k]) then
27:            Insert new_e to position k in list
28:            return
29:    Append new_e to the end of list
```

Theorem 6. *The modified NS-3 with Pseudocode 6 never generates more branches than the original NS-3 (i.e., Pseudocodes 1 and 3).*

Proof. For a cancelled timeout event *cancelled_e*, we consider two cases depending on the number of new events generated between t_{can} and t_{exp}. First, when there is no new event generated between t_{can} and t_{exp}, the modified and original NS-3 generate the same number of branches. Second, when there is at least one new event generated between t_{can} and t_{exp}, the modified NS-3 does not compare event *cancelled_e* with any of these new events because event *cancelled_e* has already been removed from *list* at t_{can}. The original NS-3 might unnecessarily compare event *cancelled_e* with these new events (depending on their timestamps) when inserting these new events into *list* at line 17 of Pseudocode 1 and thus possibly generates more branches. □

Technique to efficiently handle both outstanding and cancelled timeout events. Different from the previous technique that eliminates unnecessary comparisons due to cancelled timeout events, this new technique eliminates unnecessary comparisons due to both outstanding and cancelled timeout events. Let us still consider the example where a TCP retransmission timeout event is generated at time t_{gen} = 100 ms with an expiration time t_{exp} = 1,000 ms and is cancelled at t_{can} = 200 ms. The previous technique eliminates unnecessary comparisons between the cancelled timeout event and new events generated between t_{can} = 200 and t_{exp} = 1,000. However, new events may also be generated between t_{gen} = 100 and t_{can} = 200, and then may be unnecessarily compared with the timeout event that is still outstanding at that time and will be cancelled at a future time. Therefore,

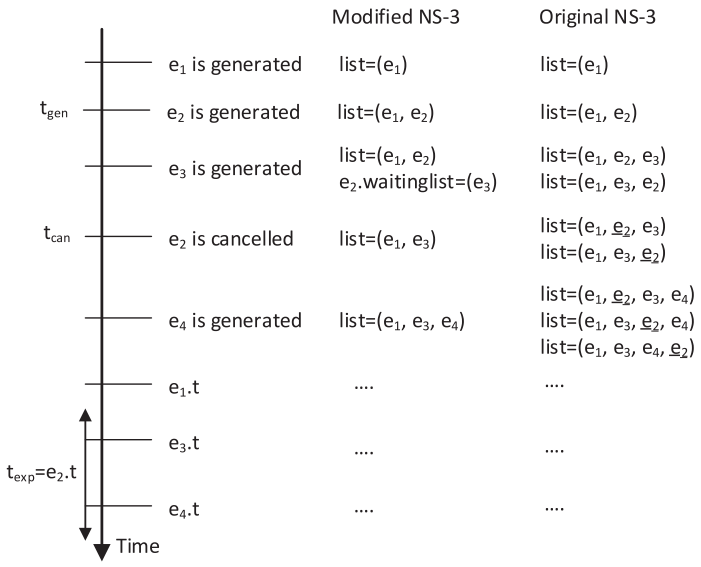

Fig. 4. An example of the modified NS-3 with Pseudocode 7. e_2 is a timeout event generated at t_{gen} with expiration time t_{exp} and cancelled at t_{can}. The timestamp (i.e., $e_2.t = t_{exp}$) of e_2 is an interval that overlaps with the timestamps of e_3 and e_4. $\underline{e_2}$ indicates cancelled e_2. Finally, the modified NS-3 has only one branch, whereas the original NS-3 has three branches.

we propose another technique to eliminate unnecessary comparisons due to both outstanding and cancelled timeout events. In other words, this new technique eliminates unnecessary comparisons between the timeout event and new events generated between $t_{gen} = 100$ and $t_{exp} = 1{,}000$ instead of just between $t_{can} = 200$ and $t_{exp} = 1{,}000$.

We propose to modify functions *FindEvent*, *ExecuteEvent*, and *InsertEvent* of NS-3 as shown in Pseudocode 7. The basic idea of this technique is to delay the comparisons involving an outstanding timeout event as late as possible until either we must compare the outstanding timeout event with other events to find the earliest event to execute (case 1) or the outstanding timeout event is cancelled (case 2). The delay of comparisons is implemented in function *InsertEvent*, case 1 is handled by function *FindEvent*, and case 2 is handled by function *ExecuteEvent*. An unsorted event waiting list $e.waitinglist$ is created for each outstanding timeout event e to keep track of all the events that have not been compared with event e.

In the following, we explain how the three functions of Pseudocode 7 work. First, function *InsertEvent* of Pseudocode 7 is similar to the original *InsertEvent* function of NS-3 (i.e., Pseudocode 3) with the following two differences. One difference is that it can insert event *new_e* to *list* from index *start_index* instead of always from 0, so as to support the modified function *ExecuteEvent*. Another difference is if it encounters an outstanding timeout event $list[k]$, it does not compare event *new_e* with $list[k]$ and just adds *new_e* to the waiting list of $list[k]$, so as to delay the comparisons with $list[k]$. Note that we do not sort the events in the waiting list and simply append *new_e* to the end of the waiting list. Also note that function *InsertEvent* does not delay the comparisons with $list[0]$ (line 23) even if it is an outstanding event, because these comparisons cannot be further delayed. Second, function *FindEvent* is modified to handle case 1 where $list[0]$ is an outstanding timeout event with a non-empty waiting list. In this case, we must insert all the events in the waiting list of $list[0]$ to *list*, to find the earliest event to execute. Note that line 8 resets e to $list[0]$ because $list[0]$ may have been changed. Third, function *ExecuteEvent* is

modified to handle case 2 where an outstanding timeout event *cancelled_e* is cancelled. In this case, we remove *cancelled_e* at line 16. If *cancelled_e* has a non-empty waiting list, the events in the waiting list are inserted to *list* starting from index *j* at line 18, where *j* is the original index of *cancelled_e* in *list*. This is because these events have already been compared with all the events before *cancelled_e* in *list*.

Figure 4 shows an example to demonstrate how modified NS-3 with Pseudocode 7 works. Event e_2 is a timeout event generated at t_{gen} with expiration time t_{exp} and cancelled at t_{can}. The timestamp (i.e., $e_2.t = t_{exp}$) of e_2 is an interval that overlaps with the timestamps of event e_3 and e_4, so events e_2 and e_3 may happen in different orders and events e_2 and e_4 may happen in different orders. For the clarity of the figure, the events that generate events e_1, e_2, e_3, and e_4 and the event that cancels event e_2 are not shown in the figure. At time t_{gen}, the modified NS-3 has the same $list = (e_1, e_2)$ as the original NS-3. When e_3 is generated between t_{gen} and t_{can}, the modified NS-3 only adds event e_3 into the waiting list of e_2, whereas the original NS-3 compares the timestamps of e_2 and e_3 and thus leads to two branches with different *list*s as shown in the figure. At time t_{can}, the modified NS-3 removes the cancelled e_2 and reinserts the events (i.e., e_3) in its waiting list to *list*, whereas the original NS-3 only marks e_2 as a cancelled event (indicated by $\underline{e_2}$) but still keeps e_2 in *list*. When e_4 is generated between t_{can} and t_{exp}, the modified NS-3 still has only one branch because it does not compare the removed e_2 with e_4, whereas the original NS-3 compares the timestamps of the cancelled e_2 and e_4 and thus leads to a total of three branches with different *list*s as shown in the figure. Therefore, we can see that the modified NS-3 with Pseudocode 7 eliminates unnecessary comparisons between event e_2 and new events generated between t_{gen} and t_{exp} (i.e., events e_3 and e_4).

Correctness and efficiency. The following two theorems prove the correctness and efficiency of our modified NS-3 with Pseudocode 7.

Theorem 7. *The modified NS-3 with Pseudocode 7 always produces the same simulation result as the original NS-3 (i.e., Pseudocodes 1 and 3) for each delay vector in the delay space.*

Proof. We prove the correctness of the modified NS-3 by showing that event *e* returned by function *FindEvent* of Pseudocode 7 is the earliest event among all the events in *list* and in the waiting lists of all outstanding timeout events in *list*. There are two cases depending on whether $e = list[0]$ assigned at line 2 of Pseudocode 7 is a timeout event with a non-empty waiting list or not.

First, if *e* assigned at line 2 is not a timeout event or is a timeout event but with an empty waiting list, its timestamp is before (defined by function *Before*) the timestamps of all other events in *list*. This is because function *InsertEvent* ensures that $list[0].t$ is before $list[k].t$ for $\forall k > 0$. In addition, its timestamp is before the timestamps of all the events in the waiting lists of all outstanding timeout events in *list*. This is because function *InsertEvent* also ensures that $list[0].t$ is before $list[k].waitinglist[i].t$ for $\forall k > 0$ and for $\forall i \geq 0$ if $list[k]$ is an outstanding timeout event. Therefore, in this case, *e* assigned at line 2 is the earliest event and is the *e* returned by function *FindEvent*.

Second, if *e* assigned at line 2 is a timeout event with a non-empty waiting list, it may or may not be the earliest event depending on the events in its waiting list. Therefore, function *FindEvent* empties its waiting list and inserts all the events in its waiting list to *list*. Note that these events will not be added back to the waiting list of *e* while *e* remains as the front event in *list* due to the condition *k* != 0 at line 23. As a result, the $e = list[0]$ reassigned at line 8 is guaranteed to be a non-timeout event or a timeout event with an empty waiting list. Thus, the *e* returned by function *FindEvent* is the earliest event. □

Theorem 8. *The modified NS-3 with Pseudocode 7 never generates more branches than the original NS-3 (i.e., Pseudocodes 1 and 3).*

Proof. Let us consider a cancelled timeout event *cancelled_e* that is generated at t_{gen} with the original expiration time t_{exp} (i.e., the timestamp) and is cancelled at t_{can}. The modified NS-3 eliminates two types of comparisons involving *cancelled_e* when compared with the original NS-3. First, when *cancelled_e* is cancelled, it is removed from *list* at line 16 of Pseudocode 7. Thus, the modified NS-3 does not compare *cancelled_e* with the new events generated between t_{can} and t_{exp} as the original NS-3. Second, when *cancelled_e* is cancelled, the events in its waiting list (i.e., the new events generated between t_{gen} and t_{can}) are inserted to *list* without comparing with *cancelled_e* at line 18 of Pseudocode 7. Thus, the modified NS-3 does not compare *cancelled_e* with the new events generated between t_{gen} and t_{can} as the original NS-3. □

To have a better understanding on the impact of our proposed techniques for the timeout events, let us consider a simplified simulation example, which has N_{timeout} back-to-back timeout events. In other words, the next timeout event is generated when the current one is cancelled (e.g., TCP usually schedules a new retransmission timer when the current one is cancelled). For each timeout event, let N_{branch} denote the number of new branches generated due to non-timeout events during the time interval from the generation to the cancellation of the timeout event, and let N_{overlap} denote the number of events overlapping with the timeout event. In Figure 4, there is $N_{\text{timeout}} = 1$ timeout event, $N_{\text{branch}} = 1$ branch generated due to non-timeout events, and $N_{\text{overlap}} = 2$ events overlapping with the timeout event. For this simulation example, the original NS-3 finally generates $(N_{\text{branch}} \times (N_{\text{overlap}} + 1))^{N_{\text{timeout}}}$ branches, whereas our modified NS-3 finally generates $(N_{\text{branch}} \times 1)^{N_{\text{timeout}}}$ branches. For Figure 4, the original NS-3 generates $(1 \times (2 + 1))^1 = 3$ branches, whereas our modified NS-3 generates only $(1 \times 1)^1 = 1$ branch.

4.4 Unnecessary Comparisons Due to Independent Events

4.4.1 Independent Events. We first define the node $e.node$ associated with an event e. There are two general types of events: link events and node events. First, a link event e simulates the propagation of a packet over a link from a source node $e.src$ to one (or more) destination node $e.dst$. Event e is usually called a packet arrival event at node $e.dst$, and we say that it is associated with node $e.dst$ (i.e., $e.node = e.dst$). Second, a node event e simulates an event at a node i, and we say it is associated with node i (i.e., $e.node = i$). For example, a timeout event at a node is a node event.

We use a general event dependency model [30] for general networking protocols. We say that two events e_i and e_j are *independent* of each other if neither $e_i \rightarrow e_j$ nor $e_j \rightarrow e_i$ holds, where $\rightarrow$ is a relation defined by the following three cases. First, $e_i \rightarrow e_j$ if $e_i.node = e_j.node$ and $Before(e_i, e_j)$. Second, $e_i \rightarrow e_j$ for a link event e_j if e_i generates e_j (then $e_i.node = e_j.src$). Third, $e_i \rightarrow e_j$ if there exists an event e_k such that $e_i \rightarrow e_k$ and $e_k \rightarrow e_j$. Intuitively, $e_i \rightarrow e_j$ means that e_i has an impact on e_j. If e_i and e_j are independent, they do not have any impact on each other. Therefore, two independent events can be executed in any order in a simulation, and it is not necessary to compare their timestamps.

4.4.2 Overview. NS-3 sorts all events using relation $Before$, which is a strict total order (i.e., irreflexive, antisymmetric, transitive, and connex). When NS-3 is executed by SEIB, the number of branches is in the order of the number of different total orders of the events with respect to relation $Before$.

We propose to modify NS-3 to sort all events using relation $\rightarrow$, which is a strict partial order (i.e., irreflexive, antisymmetric, and transitive). As result, when the modified NS-3 is executed by

SEIB, the number of branches is in the order of the number of different partial orders of the events with respect to relation $\rightarrow$.

The general idea of exploring partial ordering of event dependency to speed up software testing and verification (e.g., in model checking) is not new. The novelty of our work is that we apply it to SEIB, and we propose to achieve partial ordering for SEIB by decomposing the network simulation into multiple synchronized node simulations.

4.4.3 Differences from Traditional Parallel Simulation. Our decomposition technique is similar to and inspired by the traditional parallel simulation methods [20]. Both our decomposition technique and parallel simulation decompose the simulation of a network into multiple simulations of the nodes. Different from a parallel simulator that runs on multiple parallel processors with the aim to speed up the simulation, our work still runs on a single processor with the aim to reduce the number of branches. Thus, they have different design choices.

The first difference is about the synchronization among multiple node simulations. Parallel simulation considers how to reduce the communication overhead of the synchronization among different processors. Our decomposition technique considers how to eliminate unnecessary comparisons of independent events in the synchronization, but not about communication overhead.

The second difference is about the lookahead that is the minimum latency for an event on a node to have an impact on another node and is usually the propagation delay from the first node to the second one. Lookahead is widely used in many parallel simulation methods to improve the parallelism of different node simulations. For example, event e_i on a node can be executed before e_j on a different node if $e_i.t < e_j.t + lookahead$ (i.e., e_j has no impact on e_i). However, it is possible that $e_i.t$ and $e_j.t + lookahead$ overlaps even if $e_i.t$ and $e_j.t$ do not overlap, which leads to more branches. Thus, lookahead is not always helpful and is not used in our decomposition technique.

4.4.4 Our Technique. Pseudocode 8 shows our modified NS-3 to efficiently handle independent events, and it assumes a static network topology where the neighbors of a node do not change. By comparing the first three lines of Pseudocode 1 (i.e., original NS-3) and Pseudocode 8 (i.e., modified NS-3), we can see that the modified NS-3 still keeps the original network state variables but changes the one-dimensional array *list* to a two-dimensional array *local_list* and changes the variable *clock* to a one-dimensional array *local_clock* so that each node i has its own event list $local\_list[i]$ and its own clock $local\_clock[i]$. In the following, we use *local_list* to refer to the set of all the events in a network and $local\_list[i]$ to refer to the sorted list of all the events at node i. The modified NS-3 has the same *Run* function as the original NS-3 but has different *FindEvent*, *ExecuteEvent*, and *InsertEvent* functions, which are explained in the following.

Function *FindEvent* needs to find an event e that is safe to execute, to avoid causal violations. An event e in *local_list* is *safe* if there does not exist any event e' in *local_list* such that $e' \rightarrow e$. Because relation $\rightarrow$ is a strict partial order, there may exist multiple safe events. For a node i, its local earliest event $local\_list[i][0]$ may not be safe. There are two general ways to determine whether $local\_list[i][0]$ is safe: global synchronization using the global time information of all the nodes and local synchronization using only the local time information of the neighbors of node i. To reduce the unnecessary timestamp comparisons among different nodes, we choose local synchronization.

Function *LocalSynchronization* implements our local synchronization method, which is motivated by the local causal constraint [20] in the traditional parallel simulation. The basic idea is that the local earliest event $local\_list[i][0]$ at node i is safe if $local\_list[i]$ contains at least one packet arrival event from each neighbor and the non-decreasing arrival condition is met. The *non-decreasing arrival condition* requires that the packet arrival events from a source node j to a destination node i must be added into $local\_list[i]$ in the non-decreasing order of their timestamps.

Pseudocode 8: Modified NS-3 functions for efficiently handling independent events

```
1:  variables for network state
2:  array: local_list[node][ ]                                   ▷ local event lists
3:  array: local_clock[node]                                     ▷ local clocks
4:  function FindEvent
5:      e ← LocalSynchronization()
6:      if e = null then
7:          e ← GlobalDeadlockRecovery()
8:      return e
9:  function LocalSynchronization
10:     repeat
11:         for each node i do
12:             while local_list[i] contains at least one arrival event from each neighbor do
13:                 e ← local_list[i][0]
14:                 Remove local_list[i][0] from local_list[i]
15:                 if (e is arrival event) and (i ≠ e.dst) then
16:                     InsertEvent(e, e.dst)                    ▷ Ensures non-decreasing arrival
17:                 else
18:                     return e
19:     until no more moving of arrival events
20:     return null
21: function GlobalDeadlockRecovery( )
22:     for each node i do
23:         for each node j ≠ i do
24:             safe ← True
25:             if not Before(local_list[i][0], local_list[j][0]) then
26:                 safe ← False
27:                 break;
28:         if safe then
29:             e ← local_list[i][0]
30:             Remove local_list[i][0] from local_list[i]
31:             return e
32: function ExecuteEvent(e)
33:     local_clock[e.node] ← e.t
34:     Simulate e (update state variables, generate new events, cancel events)
35:     for each new event new_e generated by e do
36:         InsertEvent(new_e, e.node)                           ▷ Insert into the local list
37: function InsertEvent(new_e, node)
38:     for k ← 0; k < local_list[node].size; k ← k + 1 do
39:         if Before(new_e, local_list[node][k]) then
40:             Insert new_e to position k in local_list[node]
41:             return
42:         end if
43:     Append new_e to the end of local_list[node]
```

Note that because of the uncertain delay, the timestamp order of packet arrival events to node i may not be the same as the order that they are generated at source node j. To achieve the non-decreasing arrival condition, a newly generated packet arrival event is first inserted into $local\_list[j]$ of source node j (line 36). When this event becomes the local earliest event in $local\_list[j]$, it is moved to $local\_list[i]$ of destination node i (line 16).

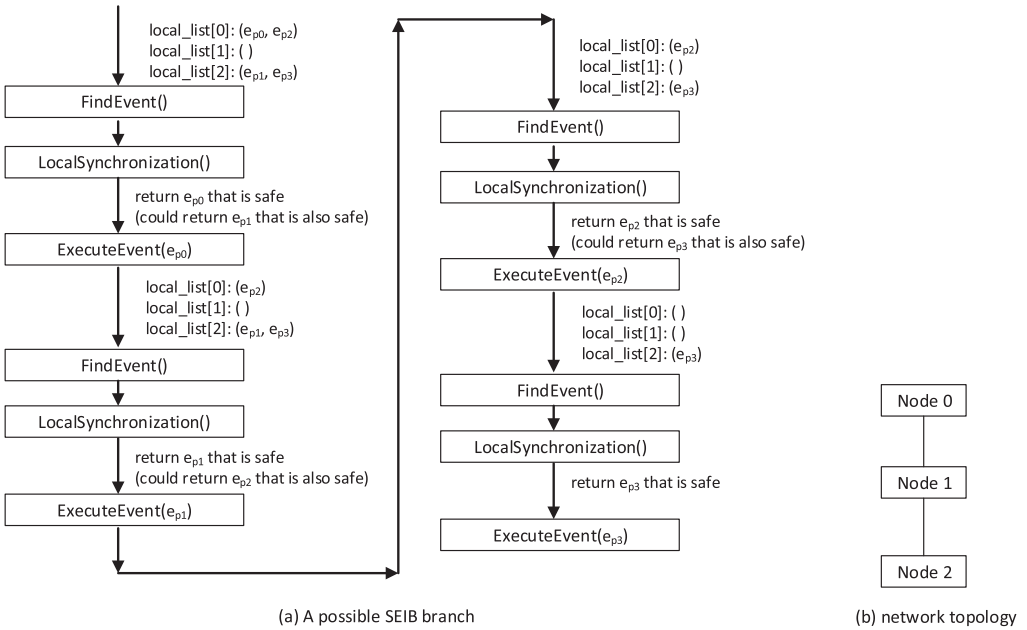

Fig. 5. One possible branch of the modified NS-3 with Pseudocode 8 for a simulation with three nodes. Suppose that in $local\_list[0]$ of node 0, event e_{p2} is an arrival event from neighbor node 1, and in $local\_list[2]$ of node 2, event e_{p3} is an arrival event from neighbor node 1. All four events can be executed without comparing the events between $local\_list[0]$ and $local\_list[2]$.

However, deadlock may occur in *LocalSynchronization*, which happens when each node is waiting for a packet arrival event from one or more of its neighbors. In this case, *LocalSynchronization* could not find any safe event and returns *null*. The deadlock can be recovered in two general ways: global recovery using global time information of all the nodes and local recovery using the local time information of the neighboring nodes. However, a limitation of local recovery (e.g., the null message method [20]) is the time-creeping problem, where the local clock of each node advances iteratively but slowly when comparing with the timestamps of its events, and leads to multiple unnecessary timestamp comparisons. Thus, we choose global recovery as explained in the following.

Function *GlobalDeadlockRecovery* finds the globally earliest event in case of the deadlock. Specifically, if the local earliest event $local\_list[i][0]$ of node i happens before the local earliest events at every other node, it is the globally earliest event and is a safe event. As explained before, we do not use the lookahead information when determining whether $local\_list[i][0]$ is safe or not.

Finally, function *ExecuteEvent* updates the local clock of node $e.node$, updates related state variables, and inserts any newly generated events to its local event list using function *InsertEvent*.

Figure 5(a) shows a possible branch of the modified NS-3 with Pseudocode 8 for a simulation with a network topology shown in Figure 5(b). There are three nodes and thus three $local\_list[]$ instead of one global $list$. Suppose that in $local\_list[0]$ of node 0, event e_{p2} is an arrival event from node 1 that is the only neighbor of node 0. Because $local\_list[0]$ contains at least one arrival event from each neighbor of node 0, its local earliest event e_{p0} is considered safe by function *LocalSynchronization*. Also suppose that in $local\_list[2]$ of node 2, event e_{p3} is an arrival event from node 1 that is the only neighbor of node 2. Because $local\_list[2]$ contains at least one arrival event from each neighbor of node 2, its local earliest event e_{p1} is considered safe by function *LocalSynchronization*. It does not matter whether function *LocalSynchronization* returns e_{p0} or

e_{p1} as both are safe. The branch in Figure 5(a) returns e_{p0}. As illustrated in Figure 5, the modified NS-3 with Pseudocode 8 finally executes all four events of nodes 0 and 2 without comparing the events between nodes 0 and 2. This is why our modified NS-3 is more efficient than the original NS-3. In general simulations, function *LocalSynchronization* sometimes may not find any safe event and then returns null. Then, function *GlobalDeadlockRecovery* compares the timestamps of the local earliest events of different nodes to find the globally earliest one.

4.4.5 Correctness. We prove the correctness of the modified NS-3 with Pseudocode 8 by proving that the events returned by *LocalSynchronization* and *DeadlockRecovery* are safe. In other words, they do not violate the causal constraints and thus do not change the simulation results.

Theorem 9. *The event e returned by function LocalSynchronization is a safe event.*

Proof. We prove that there does not exist any event e' in $local\_list$ such that $e' \rightarrow e$. Let i denote the node of event e. In other words, $e = local\_list[i][0]$.

First, let us consider all the events at node i. Because $e = local\_list[i][0]$, e has the earliest timestamp among all the events in $local\_list[i]$. Thus, there does not exist any event e' in $local\_list[i]$ such that $e' \rightarrow e$.

Second, let us consider all the events on other node $j \neq i$. Because $local\_list[i]$ contains at least one arrival event from each neighbor and the non-decreasing arrival condition is met, there does not exist any event e' in $local\_list[j]$ such that $e' \rightarrow e$. □

Theorem 10. *The event e returned by function DeadlockRecovery is a safe event.*

Proof. Let i denote the node of event e. In other words, $e = local\_list[i][0]$. Because e happens before the local earliest event at every other node j (line 25), e is the globally earliest event in $local\_list$ and is safe. □

4.4.6 Efficiency. We consider two extreme cases of the modified NS-3 with Pseudocode 8. The first case is the best case when *LocalSynchronization* never returns null. In other words, deadlock never occurs. The second case is the worst case when *LocalSynchronization* always returns null. In other words, deadlock always occurs. We prove that in both cases, the modified NS-3 with Pseudocode 8 generates no more branches than the original NS-3 (i.e., Pseudocodes 1 and 3)

Theorem 11. *In the best case, the modified NS-3 with Pseudocode 8 is more efficient than the original NS-3 (i.e., Pseudocodes 1 and 3).*

Proof. In the best case, the modified NS-3 only compares an event e with other events at the same node (line 36) when e is generated, or compares it with other events at the destination node (line 16) if e is a packet arrival event. Thus, the modified NS-3 does not have any unnecessary comparisons of events on different nodes as the original NS-3. □

Theorem 12. *In the worst case, the modified NS-3 with Pseudocode 8 has the same efficiency as the original NS-3 (i.e., Pseudocodes 1 and 3).*

Proof. In the worst case, the modified NS-3 compares the events on different nodes (lines 25 and 16) and the same node (line 36) using relation $Before$. The original NS-3 maintains only a single global event list and thus also compares the events on different nodes and same node. As a result, the modified NS-3 and original NS-3 might have different total numbers of comparisons, but they have the same number of comparisons of overlapping timestamps. □

Overall, the number of branches generated by the modified NS-3 is in the order of the number of different partial orders of the events with respect to relation $\rightarrow$ in the best case, and is in the order of the number of different total orders of the events with respect to relation $Before$ in the worst case that is the same as the original NS-3.

To have a better understanding of the impact of our proposed techniques for the independent events, let us consider a simplified simulation example based on Figure 5. Suppose that when events e_{p0} and e_{p1} are executed, each generates N_{new} new events. To simplify the analysis, we assume that all these $2 \times N_{\text{new}}$ events have the same timestamp intervals and all happen before pending events e_{p2} and e_{p3}. The original NS-3 inserts and compares all these $2 \times N_{\text{new}}$ new events into the global $list$, and thus finally generates a total of $(2 \times N_{\text{new}})!$ branches (i.e., all possible total orders of these events with respect to relation $Before$). The modified NS-3 with our proposed techniques inserts and compares the N_{new} events generated by e_{p0} into $local\_list[0]$ of node 0 and the N_{new} events generated by e_{p1} into $local\_list[2]$ of node 2. Thus, the modified NS-3 finally generates a total of $N_{\text{new}}! \times N_{\text{new}}!$ branches (i.e., all possible partial orders of these events with respect to relation $\rightarrow$).

5 EVALUATION

We evaluate the efficiency of our proposed techniques using NS-3 with various protocols and network topologies.

5.1 Simulation Setup

Table 1 lists all the testing methods to evaluate in this section. First, the *brute* force testing method directly and repeatedly runs the original NS-3 for each delay vector in the delay space (referred to as Brute). Second, the *original* SEIB method uses S^2E to execute the original NS-3 with symbolic delay variables (referred to as Original). Third, all our proposed techniques including the technique for simultaneous events (referred to as S), the technique for the conditional ineffective events that might happen after the end of a simulation (referred to as C_e), the technique for the conditional ineffective events that are cancelled timeout events (referred to as C_c), the technique for the conditional ineffective events that are outstanding or cancelled timeout events (referred to as C_{oc}), the technique for all the conditional ineffective events (referred to as C), and the technique for the independent events (referred to as I) use S^2E to execute modified NS-3 with symbolic delay variables. Fourth, we have different combinations of techniques S, C, and I. For example, SCI means that all three techniques are used. Fifth, we also use S^2E to execute NS-3 using the parallel simulation methods. NS-3 itself supports both sequential and parallel simulation methods. However, we find that the parallel simulation methods of NS-3 do not work under S^2E because the communication messages sent by their synchronization mechanisms do not support symbolic variables. Therefore, we have implemented two popular parallel simulation methods [20] using shared variables instead of communication messages for synchronization: the global safe window method (referred to as technique P_g) and the null message method (referred to as technique P_n). The source code of all our proposed techniques is available on GitHub.[1]

We run each testing method for each experiment for at most 24 hours on virtual machines configured with a 2.3-GHz four-core processor, 64 GB of RAM, and Ubuntu 14.04. The simulation scripts used in the experiments are selected from the examples provided in the NS-3. We select the following three examples, simple-error-model.cc, tcp-bulk-send.cc, and rip-simple-network.cc, corresponding to the three most important Internet protocols, UDP, TCP, and IP, respectively. We keep all the network topologies and parameter settings in the original simulation scripts, and we

[1] https://github.com/minhvu2/ns3-symbolic.

Table 1. Testing Methods to Evaluate

Method	NS-3 Functions	Symbolic Variables	Description
Brute	Original (Pseudocodes 1 and 3)	No	Brute force testing method
Original	Original (Pseudocodes 1 and 3)	yes	Original SEIB
S	Modified with Pseudocode 4	yes	SEIB + technique for simultaneous events
C_e	Modified with Pseudocode 5	yes	SEIB + technique for events that might happen after the end of a simulation
C_c	Modified with Pseudocode 6	Yes	SEIB + technique for cancelled timeout events
C_{oc}	Modified with Pseudocode 7 (contains Pseudocode 6)	Yes	SEIB + technique for both outstanding and cancelled timeout events
C	Modified with Pseudocodes 5 and 7	Yes	C_e + C_{oc} (i.e., all conditional ineffective events)
I	Modified with Pseudocode 8	Yes	SEIB + technique for independent events
SC	Modified with Pseudocodes 4, 5, and 7	Yes	S + C
SCI	Modified with Pseudocode 4, 5, 7, and 8	Yes	S + C + I
SCP_g	Modified with Pseudocode 4, 5, and 7, and global safe window [20]	Yes	S + C + Global safe window
SCP_n	Modified with Pseudocodes 4, 5, and 7, and null message [20]	Yes	S + C + Null message

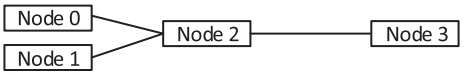

Fig. 6. Network topology of the UDP experiments.

add uncertain packet delay to a group of selected packets. Please refer to the source code of these scripts for detailed parameters, which will not be specified in the article.

5.2 UDP Experiments: Multiple Nodes

This group of experiments use the simple-error-model.cc of NS-3, which simulates a total of four nodes as shown in Figure 6 that generate a total of about 2,000 packets. Node 0 continuously sends UDP packets to node 2, and node 3 continuously sends UDP packets to node 1. We introduce uncertain delays for the last $n = 1, 2, 3, 4$ packets from node 0 to node 2. Each of these n packets has an uncertain delay in $D = [1, 1{,}024]$ ms with a millisecond resolution, and all other packets still have their delays specified in the script. The uncertain delay range D is chosen to reflect the range of most packet delays in the Internet [32].

Figure 7 shows the number of branches generated by methods Brute, Original, and SCI, and Figure 8 shows their total testing times. For Brute, the number of branches is the number of individual NS-3 simulations. For example, with $n = 1$, Brute runs 1,024 simulations and takes 25 minutes. We can see that *Original has several orders of magnitude fewer branches and shorter testing times than Brute*. For example, with $n = 2$, Brute needs to run about 10^6 simulations and could not finish in 24 hours, whereas Original takes only 7 minutes. We can also see that *SCI has even several orders of magnitude fewer branches and shorter testing times than Original.* For example, with $n = 4$, Original could not finish in 24 hours, whereas SCI takes only 94 minutes.

To understand the efficiency of each technique, Table 2 shows the number of branches generated by each technique and different combinations. The symbol ↻ means that the test could not finish in 24 hours. We can see that all our techniques are more efficient than Original. Especially, technique

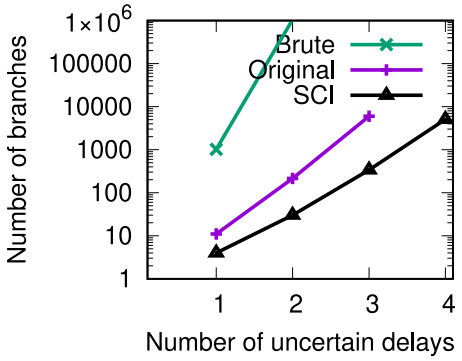

Fig. 7. Number of branches in the UDP experiments.

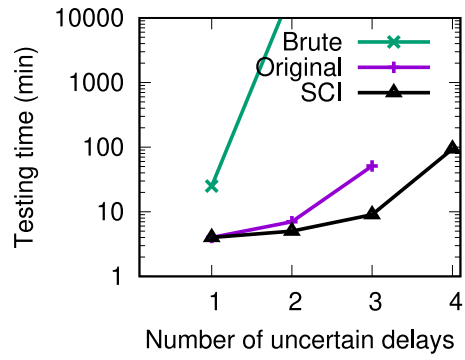

Fig. 8. Testing time of the UDP experiments.

Table 2. Number of Branches of the UDP Experiments

	$n = 1$	$n = 2$	$n = 3$	$n = 4$
Original	11	215	6,013	↻
S	8	109	2,098	↻
C	8	130	3,184	↻
I	5	41	677	13,591
SC	5	53	858	18,677
SCI	4	30	337	5,065
SCP_g	15	310	9,179	↻
SCP_n	260	↻	↻	↻

I is more efficient than techniques S and C in the UDP experiments, because there are four nodes and thus a large number of independent events on different nodes.

By comparing the results of SCI with SCP_g and SCP_n in Table 2, we can also see that our technique I has several orders of magnitude fewer branches than the two popular parallel simulation methods P_g and P_n. This is because they are designed for speeding up parallel simulation and have a large number of comparisons of timestamps.

5.3 TCP Experiments: Timeout Events

This group of experiments use the tcp-bulk-send.cc of NS-3, which simulates two nodes connected over a link, and simulates a TCP connection over the link that generates a total of about 1,500 packets. We introduce uncertain delays for the last three data packets. Each of these three packets has an uncertain delay in $D = [1, d^{max}]$ ms with a millisecond resolution, and all other packets still have the delays specified in the script. We vary the maximum uncertain delay d^{max} from 4 to 16,384 ms, which is chosen to be longer than most of the possible TCP timeout periods [38].

Figure 9 shows the number of branches generated by different testing methods. We can see that all our techniques are more efficient than Original. Technique C is more efficient than techniques S and I. This is because there are a large number of TCP outstanding timeout events, most of which are cancelled by ACK and then become ineffective events. Again we see that SCI is several orders of magnitude more efficient than Original.

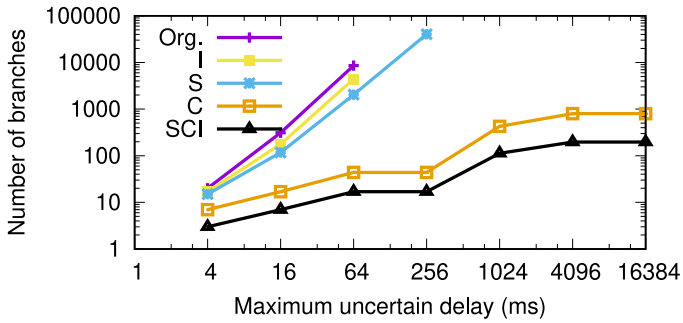

Fig. 9. Number of branches in the TCP experiments.

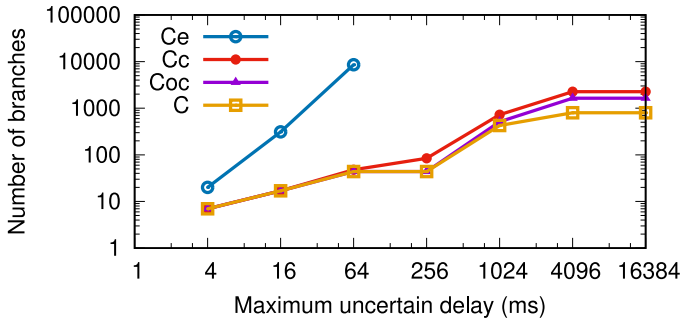

Fig. 10. Number of branches in the TCP experiment by different techniques for conditional ineffective events.

Figure 10 shows the number of branches generated by different testing methods for the conditional ineffective events. Most outstanding TCP timeouts are cancelled in the simulations, and therefore there are a large number of cancelled timeout events in the simulations. As a result, techniques C_c, C_{oc}, and C that eliminate some or all unnecessary comparisons of outstanding and cancelled timeout events are more efficient than technique C_e that only eliminates unnecessary comparisons of the beyond-the-end events.

Note that although we introduce uncertain delay to only three packets in this group of experiments, there are already a large number of branches, especially for Original (i.e., the original SEIB) as shown in Figure 9. This is because the uncertain delay of a packet has an impact not only on the packet itself but also on all the following events triggered by the packet. For example, the uncertain delay of a TCP data packet also affects the transmission event and arrival event of the ACK packet triggered by the data packet, and affects the simulation clock, the calculated round-trip time, the calculated timeout period, and then the following retransmission timeout events.

5.4 IP Routing Experiments: A Use Case

This group of experiments demonstrates a use case of our proposed techniques, where we check the behavior and evaluate the worst-case performance of a routing protocol under uncertain events. We use the rip-simple-network.cc of NS-3, which simulates a total of six nodes including four routers interconnected over a network shown in Figure 11. The routers communicate with one another to run the RIP routing protocols. A total of about 100 IP packets are generated in the simulation. The link between routers B and D is broken at 40 seconds. All the packets from routers C to A after 40 seconds have uncertain delays in $D = [1, d^{max}]$ ms, and all other packets still

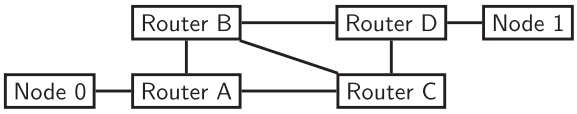

Fig. 11. Network topology of the IP routing experiments.

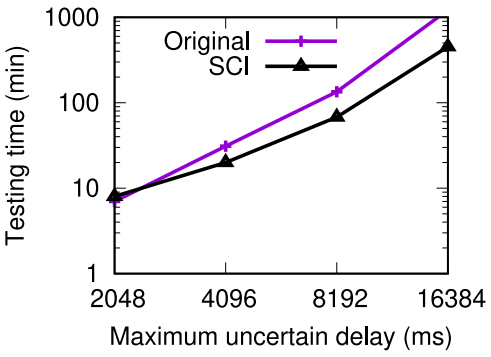

Fig. 12. Testing time of the IP routing experiments.

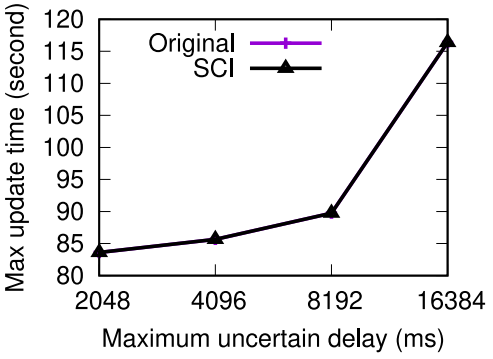

Fig. 13. Longest update time of the routing tables in the IP experiments.

have the delays specified in the script. We vary the maximum uncertain delay d^{max} from 2,048 to 16,384 ms.

First, if the routing protocol works correctly, all the routing tables will be correctly updated. Figure 12 shows the testing times for Original and SCI. After the test, every Original branch and every SCI branch report that all the routing tables have been correctly updated. In other words, the routing protocol works correctly under all possible cases of these uncertain events.

Second, for each d^{max}, we measure the longest time for correctly updating all routing tables among all possible cases of these uncertain events. Figure 13 shows the results of Original and SCI. We can see that Original and SCI report the same result. This is expected and implies that SCI generates the same simulation result as Original. Overall, we can see that SCI can be used to test the behavior and evaluate the worst-case performance of a network protocol under all possible cases of these uncertain events, and it takes significantly shorter time than Original as shown in Figure 12.

5.5 Performance Overhead in Concrete Executions

The experiments in the previous sections show the significant performance improvement of our proposed techniques in symbolic executions (i.e., SEIB), and the experiments in this section evaluate the performance of our proposed techniques in concrete executions (i.e., normal executions). If our proposed techniques do not introduce any performance overhead in concrete executions, it is more likely to be integrated into NS-3 and to be potentially used by both concrete executions and symbolic executions. We conduct two groups of experiments similar to the previous UDP and TCP experiments to measure the potential performance overhead for different network topologies and different network protocols.

The group of UDP experiments still uses the simple-error-model.cc of NS-3. Different from the UDP experiments in Section 5.2, this group of experiments runs all the methods in concrete executions, including the original NS-3 (still referred to as Original) and the modified NS-3 with our

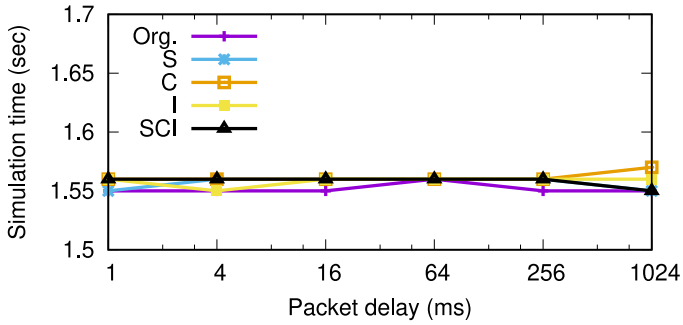

Fig. 14. The simulation time of each individual UDP experiment in concrete execution.

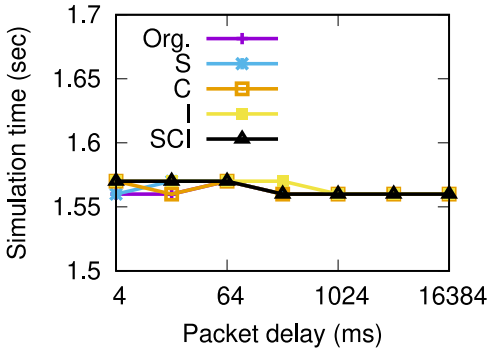

Fig. 15. The simulation time of each individual TCP experiment in concrete execution.

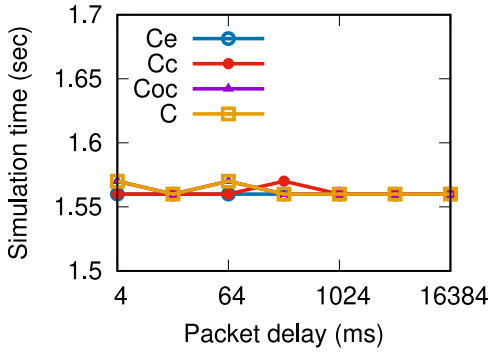

Fig. 16. The simulation time of each individual TCP experiment in concrete execution.

proposed techniques (still referred to as S, C, I, and SCI). The UDP experiments in Section 5.2 obtain the results for all these methods for up to three packets (see Table 2) with uncertain delays in $D = [1, 1024]$ ms, and thus accordingly this group of experiments varies the concrete delays of the same three packets from 1, to 4, 16, 64, 256, and 1,024 ms. We repeat the simulation for each delay and each method for 100 times, and report the average simulation time. Figure 14 shows the measured average simulation times. We can see that there is only minor difference between the simulation times of Original and all our proposed techniques. In other words, our proposed techniques do not introduce any major performance overhead into the concrete executions.

The group of TCP experiments still uses the tcp-bulk-send.cc of NS-3. Different from the TCP experiments in Section 5.3, this group of experiments runs all the methods in concrete executions, including Original, S, C, I, SCI, and these techniques for conditional ineffective events: C_e, C_c, and C_{oc}. This group of experiments varies the concrete delays of the same three packets as the TCP experiments in Section 5.3 from 4 to 16,384 ms. We repeat the simulation for each delay and each method for 100 times, and report the average simulation time. Figures 15 and 16 show the measured average simulation times. We can see that there is only minor difference between the simulation times of Original and all our proposed techniques. In other words, our proposed techniques do not introduce any major performance overhead into the concrete executions.

Compared with the experimental results in the previous sections, we can see that our proposed techniques make only minor impact on the simulation time of NS-3 in each concrete execution

(e.g., Figures 14, 15, and 16) but make a significant impact on the total testing time of NS-3 in symbolic execution (e.g., Table 2, and Figures 9 and 10). Intuitively, this is because a comparison eliminated by our techniques has a linear impact on concrete execution but a multiplicative impact on symbolic execution. For example, let us consider a simulation with a total of N_{cmp} independent comparisons of timestamps. To simplify our discussion, we assume that the simulation time of each concrete execution is proportional to the number of comparisons, and the total testing time of symbolic execution is proportional to the number of branches. With the original NS-3, the simulation time of each concrete execution is proportional to N_{cmp}, and the total testing of symbolic execution (i.e., SEIB) is proportional to $2^{N_{\text{cmp}}}$. If a proposed technique eliminates N_{eli} out of N_{cmp} comparisons, the simulation time of the modified NS-3 in each concrete execution is proportional to $N_{\text{cmp}} - N_{\text{eli}}$, and the total testing time of the modified NS-3 in symbolic execution is proportional to $2^{N_{\text{cmp}}-N_{\text{eli}}}$. In other words, a proposed technique has a linear impact on each concrete execution (i.e., from N_{cmp} to $N_{\text{cmp}} - N_{\text{eli}}$) and has a multiplicative impact on symbolic execution (i.e., from $2^{N_{\text{cmp}}}$ to $2^{N_{\text{cmp}}-N_{\text{eli}}}$). When N_{eli} is much smaller than N_{cmp} (e.g., few packets with uncertain delays in our experiments), a proposed technique has a minor impact on concrete execution but a significant impact on symbolic execution.

6 DISCUSSIONS

Are the proposed techniques specific to NS-3? We describe our proposed techniques using NS-3 [31] as an example because it is the most popular open source discrete-event network simulator. The general ideas of the three proposed techniques are not specific to NS-3, although the implementations (e.g., the pseudocode) are specific to NS-3. All three types of unnecessary comparisons discussed in the article are common in many discrete-event network simulators, such as another popular network simulator OMNeT++ [48], because these unnecessary comparisons greatly simplify the simulators and yet do not significantly affect the simulation performance of concrete executions. In other words, the general ideas of our proposed techniques can be applied to many other discrete-event network simulators.

What is the prevalence of the three types of events in general network simulations? It is hard to accurately measure the prevalence of the three types of events in general network simulations, so we intuitively explain when and why they are common in SEIB. First, simultaneous events are common in simulations with any number of nodes and any network protocol. As long as the timestamp intervals of two events overlap (note that not necessarily the same), they are likely to happen at the same time. Second, there are two types of conditional ineffective events: the events that might happen after the end of a simulation and the timeout events, both of which depend on the specific simulation. For example, if a simulation simulates TCP, then it always maintains an outstanding timeout event and potentially multiple cancelled timeout events for each TCP flow. If a simulation simulates only UDP, then it does not have any timeout events. Third, independent events are common for simulations with multiple nodes, especially among the nodes that are not neighbors.

Is SEIB scalable to a large number of packets with uncertain delays? SEIB and the proposed techniques are designed to systematically test network protocols for a small number of packets with uncertain delays and for a small number of nodes. This is based on the traditional "small scope hypothesis" [1, 25] in software testing stating that a high proportion of bugs can be detected by systematically testing a program for all possible input values in a small scope, and it is based on similar findings in network systems [29, 51, 52] in which a high proportion of bugs can be detected by systematically testing a small number of packets or nodes. Nevertheless, SEIB and our proposed techniques can be combined with other types of testing methods, such as random testing as in our

previous works [47, 50], to test network protocols with a large number of uncertain packets and/or a large number of nodes.

What are the differences between SEIB and current simulation-based performance evaluations of network protocols? SEIB and the proposed techniques are different from and complementary to the simulation-based performance evaluations. The former is used to systematically test network protocols with a small number of uncertain packets and a small number of nodes for correctness testing or worse-case performance evaluation, whereas the latter is used to evaluate the normal case or special case performance of network protocols with a large number of packets and/or a large number of nodes.

7 RELATED WORK

Network protocols have been tested in network environments with two general types of symbolic variables: (1) networks where packets have symbolic delays or loss, such as in KleeNET [37], SymTime [18, 35], SPD [45, 46], and Chiron [24]; (2) networks where packets have symbolic headers, such as in DiCE [11], SymbexNet [41], NICE [12], SOFT [29], Chiron [24], BUZZ [19], SymNet [43, 44], PIC [33], and MAX [27]. Different from the preceding works that model specific and simple network environments, our work considers general and various network environments built upon NS-3.

There are other efforts to extend network simulators for testing purpose. The J-Sim [40] network simulator is extended for model checking, and VeriSim [5] extends NS-2 for formal trace analysis. Different from these works, our work extends NS-3 for symbolic execution.

There is a large body of works to improve the efficiency of symbolic execution engines, such as compositional symbolic execution [17, 21], redundant path elimination [6, 7], path prioritization using static analysis information [3, 8], path merging [2, 23, 28, 39], state mapping methods [35, 36], and combination with random testing [15, 22, 42]. These methods are complementary to and can be used together with our proposed techniques.

8 CONCLUSION

In this article, for the first time, we apply SEIB to a large general network simulator NS-3 and propose three techniques to modify NS-3 to significantly improve the efficiency of SEIB. The efficiency of our proposed techniques depends on the simulated network protocols and topologies. For example, technique I is more efficient for simulations with multiple nodes, and technique C is more efficient for simulations with a large number of cancelled timeout events as in TCP simulations. In this work, we introduce only symbolic packet delays to NS-3 to efficiently test network protocols with temporal uncertain events. In the future, we plan to introduce symbolic packet headers to NS-3 to efficiently test network protocols with uncertain packet headers.

REFERENCES

[1] A. Andoni, D. Daniliuc, and S. Khurshid. 2002. *Evaluating the "Small Scope Hypothesis."* Technical Report, MIT CSAIL.
[2] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley. 2014. Enhancing symbolic execution with veritesting. In *Proceedings of the International Conference on Software Engineering*.
[3] D. Babic, L. Martignoni, S. McCamant, and D. Song. 2011. Statically-directed dynamic automated test generation. In *Proceedings of ACM ISSTA*.
[4] F. Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX ATC*.
[5] K. Bhargavan, C. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. 2002. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering* 28, 2 (Feb. 2002), 129–145.
[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'08)*.
[7] S. Bugrara and D. Engler. 2013. Redundant state detection for dynamic symbolic execution. In *Proceedings of USENIX ATC*.

[8] J. Burnim and K. Sen. 2008. Heuristics for scalable dynamic test generation. In *Proceedings of the IEEE/ACM Conference on Automated Software Engineering*.
[9] 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*.
[10] C. Cadar and K. Sen. 2013. Symbolic execution for software testing: Three decades later. *Communications of the ACM* 56, 2 (Feb. 2013), 82–90.
[11] M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, and O. Crameri. 2011. Toward online testing of federated and heterogeneous distributed systems. In *Proceedings of USENIX ATC*.
[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*.
[13] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert. 2013. PacketDrill: Scriptable network stack testing, from sockets to packets. In *Proceedings of USENIX ATC*. 213–218.
[14] V. Chipounov, V. Kuznetsov, and G. Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems* 30, 1 (Feb. 2012), Article 2, 49 pages.
[15] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. 2011. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Conference on Security (SEC'11)*.
[16] L. A. Clarke. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* 2, 3 (May 1976), 215–222. https://doi.org/10.1109/TSE.1976.233817
[17] M. Dobrescu and K. Argyraki. 2014. Software dataplane verification. In *Proceedings of USENIX NSDI*.
[18] O. Dustmann. 2013. Symbolic execution of discrete event systems with uncertain time. *Lecture Notes in Informatics* S-12 (2013), 19–22.
[19] 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*.
[20] R. Fujimoto. 1990. Parallel discrete event simulation. *Communications of the ACM* 33, 10 (Oct. 1990), 30–53.
[21] P. Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of POPL*.
[22] P. Godefroid, N. Klarlund, and K. Sen. 2005. DART: Directed automated random testing. In *Proceedings of ACM Programming Language Design and Implementation*. 213–223.
[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'09)*.
[24] E. Hoque, O. Chowdhury, S. Chau, C. Nita-Rotaru, and N. Li. 2017. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*.
[25] D. Jackson and C. Damon. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering* 22, 7 (July 1996), 484–495.
[26] J. King. 1976. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (July 1976), 385–394.
[27] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. 2011. Finding protocol manipulation attacks. In *Proceedings of ACM SIGCOMM*.
[28] 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*.
[29] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. 2012. A SOFT way for OpenFlow switch interoperability testing. In *Proceedings of the ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'12)*.
[30] P. Li and J. Regehr. 2010. T-Check: Bug finding for sensor networks. In *Proceedings of ACM/IEEE IPSN*.
[31] NS-3. n.d. ns-3 Network Simulator. Retrieved November 19, 2021 from https://www.nsnam.org/.
[32] V. Paxson. 1997. End-to-end internet packet dynamics. In *Proceedings of ACM SIGCOMM*.
[33] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. 2015. Analyzing protocol implementations for interoperability. In *Proceedings of USENIX NSDI*.
[34] P. Peschlow, P. Martini, and J. Liu. 2008. Interval branching. In *Proceedings of the ACM Workshops on Principles of Advanced and Distributed Simulation*.
[35] R. Sasnauskas, O. Dustmann, B. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. 2011. Scalable symbolic execution of distributed systems. In *Proceedings of ICDCS*.
[36] R. Sasnauskas, P. Kaiser, R. Jukic, and K. Wehrle. 2012. Integration testing of protocol implementations using symbolic distributed execution. In *Proceedings of IEEE ICNP*.
[37] 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*. 186–196.
[38] N. Seddigh and M. Devetsikiotis. 2001. Studies of TCP's retransmission timeout mechanism. In *Proceedings of IEEE ICC*.

[39] K. Sen, G. Necula, L. Gong, and W. Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of ESEC/FSE*.
[40] A. Sobeih, M. Viswanathan, D. Marinov, and J. Hou. 2007. J-Sim: An integrated environment for simulation and model checking of network protocols. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.
[41] 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.
[42] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbette, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of NDSS*.
[43] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. 2013. SymNet: Static checking for stateful networks. In *Proceedings of the ACM CoNEXT HotMiddlebox Workshop*.
[44] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of ACM SIGCOMM*. 314–327.
[45] W. Sun, L. Xu, and S. Elbaum. 2015. SPD: Automatically test unmodified network programs with symbolic packet dynamics. In *Proceedings of IEEE GLOBECOM*.
[46] W. Sun, L. Xu, and S. Elbaum. 2017. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*. 79–89.
[47] W. Sun, L. Xu, S. Elbaum, and D. Zhao. 2019. Model-agnostic and efficient exploration of numerical state space of real-world TCP congestion control implementations. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. 719–734.
[48] A. Varga. n.d. OMNET++ Discrete Event Simulator. Retrieved November 19, 2021 from https://omnetpp.org/.
[49] M. Vu, L. Xu, S. Elbaum, W. Sun, and K. Qiao. 2019. Efficient systematic testing of network protocols with temporal uncertain events. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'19)*. 604–612.
[50] Minh Vu, Phuong Ha, and Lisong Xu. 2020. Efficient correctness testing of Linux network stack under packet dynamics. In *Proceedings of the IEEE International Conference on Communications (ICC'20)*.
[51] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. Krishnamurthy, T. D. Braun, and K. Chan. 2020. SymTCP: Eluding stateful deep packet inspection with automated discrepancy discovery. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'20)*.
[52] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, and X. Zhao. 2014. Simple testing can prevent most critical failures. In *Proceedings of USENIX OSDI*.

Received September 2020; revised June 2021; accepted September 2021