Symbolic ns-3 for Efficient Exhaustive Testing: Design, Implementation, and Simulations

Jianfei Shao
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
jianfei.shao@huskers.unl.edu

Minh Vu
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
minh.vu@huskers.unl.edu

Mingrui Zhang
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
mzhang23@huskers.unl.edu

Asmita Jayswal
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
asmita.jayswal@huskers.unl.edu

Lisong Xu
University of Nebraska-Lincoln
Lincoln, Nebraska, USA
xu@unl.edu

ABSTRACT

Exhaustive testing is an important type of simulation, where a user exhaustively simulates a protocol for all possible cases with respect to some uncertain factors, such as all possible packet delays or headers. It is useful for completely evaluating the protocol performance, finding the worst-case performance, and detecting possible design or implementation bugs of a protocol. It is, however, time consuming to use the brute force method with current ns-3 for exhaustive testing. In this paper, we present our work on sym-ns-3 for more efficient exhaustive testing, which leverages a powerful program analysis technique called symbolic execution. Intuitively, sym-ns-3 groups all the cases leading to the same simulator execution path together as an equivalence class, and simulates a protocol only once for each equivalence class. We present our design choices and implementation details on how we extend ns-3 to support symbolic execution, and also present several exhaustive testing results to demonstrate the significantly improved testing speeds of sym-ns-3 over current ns-3.

CCS CONCEPTS

• **Networks** → **Network simulations**.

KEYWORDS

Exhaustive testing, Symbolic execution, Packet dynamic and semantics

ACM Reference Format:
Jianfei Shao, Minh Vu, Mingrui Zhang, Asmita Jayswal, and Lisong Xu. 2022. Symbolic ns-3 for Efficient Exhaustive Testing: Design, Implementation, and Simulations. In *Proceedings of the WNS3 2022 (WNS3 2022), June 22–23, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3532577.3532604

WNS3 2022, June 22–23, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9651-6/22/06…$15.00
https://doi.org/10.1145/3532577.3532604

1 INTRODUCTION

ns-3 [14] is a popular network simulator that has been widely used in the networking community. It is usually used to evaluate the normal-case or special-case performance of a network protocol, where a user simulates the protocol for some normal or special cases. In this paper, we consider an important type of simulation (referred to as *exhaustive testing* hereinafter), where a user exhaustively simulates a protocol for all possible cases with respect to some uncertain factors, such as all possible packet delays or headers. Exhaustive testing is useful for completely evaluating the performance of a protocol for all possible cases, finding the worst-case performance of a protocol, and for detecting possible design or implementation bugs of a protocol as many bugs happen only in corner cases.

It is, however, time consuming to use the current ns-3 for exhaustive testing, because a user needs to enumerate and simulate a protocol for each possible case (referred to as the *brute force* method). Let's consider a simple motivating example, where we need to exhaustively test a protocol in the network shown in Figure 1, where the propagation delay d_i of link $i = 0, 1$ could be any value of between 1 ms and 1000 ms with a resolution of 1 ms. Thus, the test space (d_0, d_1) contains a total of $10^3 \times 10^3 = 10^6$ possible testing cases. The brute force method runs ns-3 to enumerate and individually simulate each of the 10^6 cases in the test space, and thus takes a long time. The details can be found in Section 2.

In this paper, we present our work on *Symbolic ns-3* (sym-ns-3 for short), which extends ns-3 to support the *symbolic execution method* [3, 11] that is a powerful and popular program analysis technique widely used in the software testing and verification community. Intuitively, the symbolic execution method divides the test space (d_0, d_1) into equivalence classes, each equivalence class containing all the cases leading to the same simulator execution path. The symbolic execution method simulates all the cases in an equivalence class together instead of individually as the brute force method. By doing so, the symbolic execution method can more efficiently exhaustively test the same test space (d_0, d_1) than the brute force method. The details can also be found in Section 2.

Symbolic execution has been used to test network protocols. KleeNET [16], SymTime [7], SPD [20], and Chiron [9] test network protocols with symbolic packet delays or loss. DiCE [4],

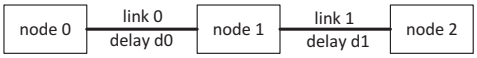

Figure 1: Network Topology of the Motivating Example

SymbexNet [18], NICE [5], SOFT [13], Chiron [9], BUZZ [8], SymNet [19], PIC [15], and MAX [12] test network protocols with symbolic packet headers. Different from these works that consider only simple and specific network environments, sym-ns-3 attempts to support general network environments by leveraging ns-3.

We make the following contributions in this paper: 1) *Symbolic variable management*: We present our design choices and implementation of symbolic variables in sym-ns-3 that are the foundation of symbolic execution. A symbolic variable takes a set of values instead of a single value as a normal variable. We have explored multiple different ways to introduce symbolic variables to sym-ns-3 so that sym-ns-3 users can easily manage symbolic variables and sym-ns-3 developers can easily maintain and upgrade sym-ns-3.

2) *More efficient packet semantics testing*: There are two types of exhaustive testing: 1) packet dynamic testing: checking a protocol with all possible packet dynamics (e.g., delays), 2) packet semantics testing: checking a protocol for all possible packet header and payload semantics. We have already presented several techniques to further improve the testing speed of sym-ns-3 for packet dynamic testing in our previous work [21], and we present several techniques to further improve the testing speed of sym-ns-3 for packet semantics testing in this paper.

3) *Simulations*: We present several exhaustive testing simulations to demonstrate how to use sym-ns-3 and the significantly improved testing speeds of the symbolic execution method using sym-ns-3 compared to the brute force method using current ns-3.

2 MOTIVATING EXAMPLE

In this section, we present a simple exhaustive testing example to illustrate the difference between the brute force method of current ns-3 and the symbolic execution method of our proposed sym-ns-3.

2.1 An Exhaustive Testing Problem

Let's consider a network shown in Figure 1, where three nodes are connected by two point-to-point links. Nodes 0 and 2 each simultaneously sends a UDP packet to node 1. The propagation delay d_i of link $i = 0, 1$ could be any value between 1 ms and 1000 ms. An exhaustive testing problem is to find the range of all possible *diff* among a total of 10^6 testing cases (i.e., combinations) of d_0 and d_1, where *diff* is the arrival time difference at node 1 between the packets from node 0 and node 2.

In order to make it easier for the readers to understand, this exhaustive testing problem is very simple as we can manually infer that the range of *diff* should be [-999, 999] ms. More realistic and complicated examples are demonstrated in Section 6.

2.2 Brute Force using Current ns-3

To find the range of *diff* using the brute force method with the current ns-3, we write shell script `repeatCurrentDemo.sh` as shown in Code 1 to enumerate all possible 10^6 cases of d_0 and d_1, and run an ns-3 simulation for each case.

Code 1: Shell Script repeatCurrentDemo.sh to Enumerate All Possible Cases

```
#!/bin/bash
for delay0 in {1..1000}
do
  for delay1 in {1..1000}
  do
    ./waf --run "currentDemo --d0=$delay0 --d1=$delay1"
  done
done
```

We also write ns-3 script `currentDemo.cc` as shown in Code 2 to simulate the network according to the link delays specified in the arguments. To simplify the example, we calculate *diff* directly in this script instead of modifying file `udp-server.cc` or creating a trace sink to measure the packet arrival times at the receiver and then calculate *diff*.

Code 2: ns-3 Script currentDemo.cc to Simulate Each Case

```
... // Other setup code
p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
p2p[1].SetChannelAttribute("Delay",TimeValue(Time(d1)));
... // Simulation execution
Time diff = Time(d0)-Time(d1);
std::cout<<"diff is "<<diff<<std::endl;
```

It takes about a half second to run a single case and thus a total of about six days to run all the cases. The total reported range of *diff* is [-999, 999] ms.

2.3 Symbolic Execution using sym-ns-3

To find the range of *diff* using the symbolic execution method with our sym-ns-3, we write only one script `symDemo.cc` as shown in Code 3 to simulate the network with two symbolic link delays, each in the range of [1, 1000] ms. Also, to simplify the example, we calculate *diff* directly in the script.

Code 3: sym-ns-3 Script symDemo.cc

```
... // Other setup code
Ptr<Symbolic> sym0 = CreateObject<Symbolic>();
sym0->SetMinMax(1, 1000);
uint32_t d0 = sym0->GetSymbolicUintValue();
Ptr<Symbolic> sym1 = CreateObject<Symbolic>();
sym1->SetMinMax(1, 1000);
uint32_t d1 = sym1->GetSymbolicUintValue();
p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
p2p[1].SetChannelAttribute("Delay",TimeValue(Time(d1)));
...  // Simulation execution
Symbolic diff=sym0-sym1;
diff.PrintRange("diff");
```

It takes a total of about a minute to execute the code using a symbolic execution engine, which is several orders of magnitude faster than the brute force method. The total reported range of *diff* is also [-999, 999] ms, the same as the range reported by the brute force method.

3 OVERVIEW

3.1 Architecture of sym-ns-3

Figure 2 illustrates the different architectures of the brute force method with current ns-3 and the symbolic execution method with sym-ns-3. Different from the brute force method that directly executes ns-3, the symbolic execution method uses symbolic execution platform S^2E [6] to symbolically execute sym-ns-3 in virtual machines. The reason that we choose S^2E is that it supports virtual

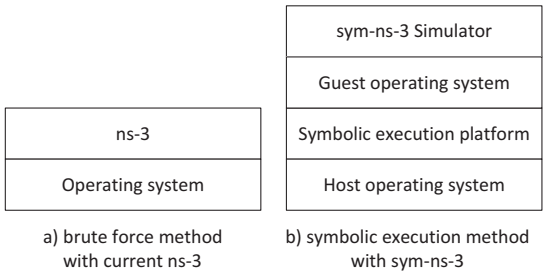

Figure 2: Architectures of the Brute Force and Symbolic Execution Methods

machine symbolic execution and thus is easier to symbolically execute big software systems, such as ns-3, involving multiple languages, multiple dependent packages, and heavy operating system interactions. Specifically, S^{2}E emulates the virtual machines using the QEMU machine emulator [1] and conducts symbolic execution using the KLEE symbolic execution engine [2]. Different from the brute force method where each variable can take only a value at a time, the symbolic execution method introduces symbolic variables, each of which can take a set of values described by a group of constraints.

3.2 Symbolic Execution

We use the C-like pseudocode shown in Code 4 as an example to explain how S^{2}E works. S^{2}E initially runs the code on a single virtual machine. Lines 1 and 2 define two symbolic variables d_0 and d_1 with the same initial constraints, and thus each of them initially takes a set of values in the range of 1 and 1000. When S^{2}E reaches an `if` statement involving symbolic variables, such as lines 3 and 5, it calls a constraint solver to determine which branch is feasible. If both branches are feasible, it conceptually forks the current virtual machine into two virtual machines (called *branches*) using lightweight snapshots and backtracking. For example, when the `if` statement at line 3 is executed, S^{2}E forks the current virtual machine into two virtual machines, where the true branch continues to line 4 with additional constraint $d_0 > d_1$ and the false branch continues to line 5 with additional constraint $d_0 <= d_1$. Similarly for the `if` statement at line 5.

Code 4: An Example for Symbolic Execution

```
1  sym 1<= d0 <= 1000
2  sym 1<= d1 <= 1000
3  if (d0 > d1){
4    ... //simulate accordingly
5  }else if (d0==d1){
6    ... //simulate accordingly
7  }else{
8    ... //simulate accordingly
9  }
10 diff = d0 - d1;
```

Finally, S^{2}E stops with three branches as illustrated in Figure 3, where the final constraints for each branch are also listed. Using these final constraints, S^{2}E can then calculate the possible range of variable *diff* defined in line 10. Specifically, the range of *diff* is [1, 999] ms for branch 1, [0, 0] ms for branch 2, and [-999, -1] ms for branch 3. The total range of *diff* is the union of these ranges and thus is [-999, 999] ms.

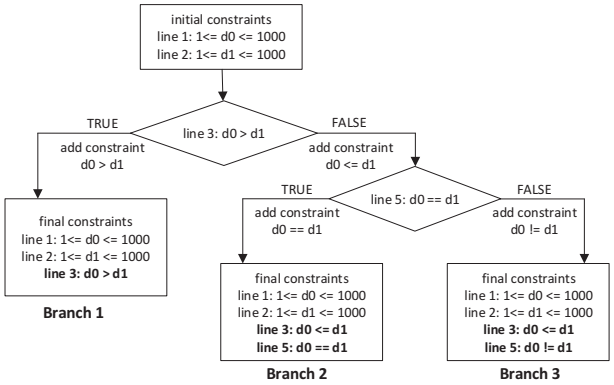

Figure 3: Three Branches Generated during the Symbolic Execution of Code 4

Symbolic execution consumes more CPU and memory resources than normal execution. For example, for the motivating example in Section 2, the symbolic execution method using sym-ns-3 consumes a total of about 2 GBytes of memory, whereas the brute force method using current ns-3 consumes only about 200 MBytes of memory.

3.3 Design Goals

sym-ns-3 is designed with the following design goals.

(1) *Easy to use*: It is easy for current ns-3 users to use sym-ns-3 for exhaustive testing. Specifically, a sym-ns-3 user writes a sym-ns-3 testing script in a way very similar to an ns-3 user writing an ns-3 testing script.
(2) *Easy to develop*: It is easy for sym-ns-3 developers to maintain and upgrade sym-ns-3. Specifically, sym-ns-3 makes as little change as possible to current ns-3, especially, to existing ns-3 modules.
(3) *Efficient*: It is more efficient to conduct exhaustive testing using sym-ns-3 than current ns-3. Although symbolic execution already makes sym-ns-3 more efficient for exhaustive testing than brute force with current ns-3, we propose several techniques to further improve the efficiency of sym-ns-3.

4 SYMBOLIC VARIABLE MANAGEMENT

In this section, we describe how to design sym-ns-3 so that a sym-ns-3 user can easily use symbolic variables (i.e., the first design goal) and a sym-ns-3 developer can easily develop sym-ns-3 (i.e., the second design goal).

4.1 Managing Symbolic Variables

Exhaustive testing simulates a network for all possible cases with respect to some uncertain factors, which can be tested using symbolic variables in sym-ns-3. For example, the motivating example simulates a network for all possible link delays d_0 and d_1, which are tested using symbolic variables.

There are two different ways to change a normal variable to a symbolic variable in sym-ns-3. 1) *Direct Symbolization*: We can

use S^{2}E functions to make a normal variable symbolic. For example, function `s2e_make_symbolic(&x,sizeof(x),"x")` makes variable `x` a symbolic variable by marking `sizeof(x)` number of bytes at address `&x` symbolic. We can make most basic data types, such as integers and strings, symbolic, except floating-point data types because S^{2}E (and KLEE) currently supports limited symbolic floating-point computation. For an ns-3 class, such as `Time`, we can make its data members symbolic. 2) *Assignment Symbolization*: If a normal variable `y` is set to the value of an expression involving a symbolic variable `x`, variable `y` also becomes a symbolic variable. For example, if symbolic variable `x` has a symbolic value between 1 and 1000, variable `y` will have a symbolic value between 2 and 1001 after executing assignment `y = x+1`. Specifically, S^{2}E marks variable `y` symbolic and associates it with symbolic expression `x+1`.

We need to provide users with the following types of functions to manage the symbolic variables. 1) *Initialization functions*: We need to provide users with functions to set the initial constraint of a symbolic variable. For example, d_0 in the motivating example should be defined as a symbolic variable with an initial constraint of between 1 and 1000 ms. 2) *Operation functions*: We need to provide users with functions to operate on symbol variables, such as functions to perform math operations (e.g., addition, subtraction) of symbolic variables, and functions to change the type of a symbolic variable (e.g., change from `unsigned` to `Time`). 3) *Inquiry functions*: Different from a normal variable that takes a single value, a symbolic variable takes a set of values described by a group of constraints. Furthermore, the constraints of a symbolic variable may change as the simulation continues. For example, the three branches in Figure 3 each has a different group of constraints for symbolic variables d_0 and d_1. Thus, we need to provide functions for users to inquire and print out the current range of a symbolic variable, such as the max, min, and sample values.

We have explored three different methods to manage the symbolic variables in sym-ns-3. Below we explain these methods using the propagation delay of link 0 in the motivating example, which is a point to point channel `PointToPointChannel`. Its propagation delay is defined as a `Time` variable named `m_delay`. We have explored three different methods to make `m_delay` symbolic.

4.2 Method 1: In-module Direct Symbolization

This method directly modifies existing ns-3 modules that are involved in an exhaustive test. For the motivating example, this method directly modifies `PointToPointChannel` by adding new channel attributes and modifying its code accordingly. For example, Code 5 shows a part of a different script `symDemo.cc` implemented using this method. Specifically, we add three new channel attributes `SymbolicMode`, `DelayMin`, and `DelayMax`. If `SymbolicMode` is true, `s2e_make_symbolic(&m_delay,sizeof(m_delay) ,"m_delay")` is called to make `m_delay` symbolic, and then its minimum and maximum values are set to `DelayMin`, and `DelayMax`, respectively.

Code 5: Method 1

```
1 p2p[0].SetChannelAttribute("SymbolicMode",BooleanValue(true));
2 p2p[0].SetChannelAttribute("DelayMin",TimeValue(Time("1ms")));
3 p2p[0].SetChannelAttribute("DelayMax",TimeValue(Time("1000ms")
      ));
```

We originally adopted this method at the beginning of our project, because the advantage of this method is that it can flexibly implement more module-specific functionalities. For example, instead of all the packets on the link experiencing the same symbolic delay `m_delay`, we can introduce a new attribute to specify that only a certain type of packets experience the symbolic delay (e.g., only data packets of a specific TCP flow), and a new attribute to specific that different packets experience different symbolic delays (e.g., to introduce packet reordering).

We later explored other methods, because this method has two disadvantages that make it hard to develop (i.e., the second design goal). First, this method makes a big change to an ns-3 module, because we need to modify and add many functions to the ns-3 module in order to add and implement all the new channel attributes and to implement the initialization, operation, and inquiry functions to manage the symbolic variables. Second, this method makes big changes to many ns-3 modules, because we have to modify each ns-3 module for which we need to add symbolic variables. For example, if we want to exhaustively test other channels, such as wireless channels, we need to modify all these channels.

4.3 Method 2: Assignment Symbolization using New Attributes

Method 2 makes less change to existing ns-3 modules than method 1 by defining and managing symbolic objects using a new class called `Symbolic`, which we have developed for sym-ns-3. For example, Code 6 shows a part of a different script `symDemo.cc` implemented using this method. It first creates a symbolic variable `symObj0` which is a `Symbolic` class variable. The `Symbolic` class implements all the functions to manage symbolic variables, such as initialization, operation, and inquiry functions, so that existing ns-3 modules do not need to implement these functions as in Method 1. For example, line 2 sets the initial range of `symObj0` using function `SetMinMax`.

Method 2 still changes `PointToPointChannel` by adding a new attribute `SymbolicDelay`, which takes a pointer value of `symObj0` whose value is then assigned to variable `m_delay`.

Code 6: Method 2

```
1 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
2 symObj0->SetMinMax(1, 1000);
3 p2p[0].SetChannelAttribute ("SymbolicDelay", PointerValue (
      symObj0));
```

The advantage of this method is that it can flexibly implement the same module-specific functionalities as Method 1 while making less changes to existing ns-3 modules than Method 1. For example, similar to Method 1, instead of all the packets on the link experiencing the same symbolic delay `m_delay`, Method 2 can also introduce a new attribute to specify only a certain type of packets to experience the symbolic delay (e.g., only TCP data packets). But different from Method 1, Method 2 does not need to change `PointToPointChannel` to implement the initialization, operation, and inquiry functions to manage symbolic variables.

The disadvantage of this method is that it still makes changes (although just adding new attributes) to many ns-3 modules, because we have to modify each ns-3 module for which we need to add symbolic variables.

4.4 Method 3: Assignment Symbolization using Existing Attributes (Motivating Example)

This method does not modify existing ns-3 modules at all. Because it does not add any new attributes to an ns-3 module, we have to use the existing attributes. For example, Code 7 shows a part of script `symDemo.cc` implemented using this method, which is just the one illustrated in the motivating example in Section 2.3. It first creates the same symbolic variable `symObj0` as Method 2. Then it gets the corresponding symbolic unsigned integer `d0` from `symObj0`, and then passes `d0` to `p2p[0]` (i.e., link 0) using the existing `PointToPointChannel` attribute `Delay`. Note that variable `d0` is also a symbolic variable with the same set of values as `symObj0`, as explained in the assignment symbolization in Section 4.1.

Code 7: Method 3

```
1 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
2 symObj0->SetMinMax(1, 1000);
3 uint32_t d0 = symObj0->GetSymbolicUintValue();
4 p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
```

The advantage of this method is that it does not make any changes to the existing ns-3 modules. As a result, it is easy for sym-ns-3 developers to maintain and upgrade sym-ns-3 (i.e., the second design goal), and it is easy to apply this method to any ns-3 modules, in addition to `PointToPointChannel` used in the example.

The disadvantage of this method is that it uses only the existing attributes of ns-3 modules, and thus supports only limited module-specific functionalities. For example, all the packets on the link have to experience the same symbolic delay, and we cannot specify only a certain type of packets to experience the symbolic delay.

4.5 Comparison of the Three Methods

Figure 4 compares the methods to manage symbolic variables. Method 1 makes the biggest changes to existing ns-3 modules, whereas Method 3 does not make any changes. On the other side, Methods 1 and 2 support more module-specific functionalities than Method 3 that supports only basic functionalities.

While all methods make sym-ns-3 easy to use for current ns-3 users (i.e., design goal 1), Method 3 is the easiest for sym-ns-3 developers to develop (i.e., design goal 2). Therefore, for the current release of sym-ns-3, we choose Method 3 so that symbolic variables can be used with all current ns-3 modules. One possible method that we plan to explore in the future is to combine these methods. For example, we use Method 3 for most ns-3 modules, but use Method 2 for some ns-3 modules requiring module-specific functionalities.

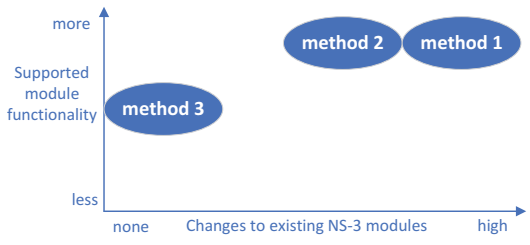

Figure 4: Methods to Manage Symbolic Variables

5 MAKING SYM-NS-3 MORE EFFICIENT

Although the methods proposed in the previous section already make sym-ns-3 more efficient for exhaustive testing than ns-3, we have noticed that we can make sym-ns-3 even more efficient by redesigning some of ns-3 modules (i.e., the third design goal). Intuitively, this is because ns-3 was not originally designed and implemented for symbolic execution, and thus we have proposed some techniques to redesign and make it symbolic execution friendly.

We have proposed two types of techniques for two general types of exhaustive testing using sym-ns-3. 1) *Exhaustive packet dynamic testing*: It tests a network protocol in a network with all possible packet dynamics, such as all possible packet delays in the motivating example. For this type of testing, sym-ns-3 changes some time-related variables to symbolic variables, such as d_0 in the motivating example. As a result, the timestamps of events become symbolic. In our previous work [21], we have proposed several techniques to redesign the event schedulers of sym-ns-3 so that it can more efficiently compare the symbolic timestamps of the events.

2) *Exhaustive packet semantics testing*: It tests a network protocol for packets with all possible header and payload semantics, such as all possible destination IP addresses. For this type of testing, sym-ns-3 changes the packet header fields or packet payload to symbolic variables. In this paper, we consider the destination IP address field of a packet, which is one of the most important fields of a packet header. Below, we propose two techniques to redesign the IP routing protocol of sym-ns-3 so that it can more efficiently handle packets with symbolic destination IP addresses.

5.1 Symbolic IP Address

A symbolic IP address can be used for exhaustive testing of a packet with a set of destination IP addresses. For example, an IP reachability test [10, 19] checks whether a packet from a node can reach another node. However, it is time consuming to find all possible nodes that can be reached by a packet from a node, if we exhaustively try all possible destination IP addresses for the packet. With the help of a symbolic destination IP address, we can more efficiently find all possible nodes that can be reached from a node. Code 8 shows how a symbolic IP address can be defined with an initial range of IP addresses 10.1.0.0 to 10.2.255.255 in sym-ns-3.

Code 8: Defining a Symbolic IP Address in sym-ns-3

```
1 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
2 symObj0->SetMinMax(0xa010000, 0xa02ffff);
3 Ipv4Address symIP0 = symObj0->GetSymbolicIpv4Add();
```

5.2 How Current ns-3 Simulates IP Routing?

We describe how current ns-3 simulates an IP routing table in this subsection, and then explain why it is not friendly to symbolic execution and how we modify it in the following subsections. Below we briefly demonstrate how current ns-3 maintains a routing table, checks a table entry for possible match, and finds the best match for the whole table, as each of them will be redesigned in sym-ns-3.

ns-3 maintains an unsorted routing table, where a new table entry is added to the end of the table. For example, Table 1 shows a possible routing table at a node. For a table `entry` with network destination `entry.ip` and mask `entry.mask`, ns-3 checks whether

Table 1: Routing Table Example of Current ns-3

Destination	Mask	Interface	Metric
127.0.0.0	255.0.0.0	0	1
10.1.0.0	255.255.0.0	2	10
10.2.0.0	255.255.0.0	2	10
0.0.0.0	0.0.0.0	1	1

the destination IP address `dst` of a packet matches the entry using function `IsMatch` as illustrated in Pseudocode 9.

Code 9: Pseudocode of Function IsMatch in ns-3

```
1 IsMatch (IP Address dst, Table Entry entry)
2   if ((dst & entry.mask) == (entry.ip & entry.mask))
3       return true;
4   else
5       return false;
```

ns-3 checks every table entry to find the best match, which is the entry with the longest mask among all matching entries. If there are multiple matching entries with the same longest length of masks, the one with the shortest metric is the best match. The code is illustrated as function `Lookup` in Pseudocode 10. For example, if `dst`=10.1.0.1, there are two matching entries in Table 1: the entry for 10.1.0.0 and the entry for 0.0.0.0. Because the former has a longer mask than the latter, the best match is the former.

Code 10: Pseudocode of Function Lookup in ns-3

```
1 Lookup (IP Address dst, IP Table table)
2   for each entry in the table
3       if IsMatch(dst, entry)
4           if (entry.masklen > bestmatch.masklen)
5               bestmatch = entry;
6           else if ((entry.masklen = bestmatch.masklen) and
7                 (entry.metric < bestmatch.metric))
8               bestmatch = entry;
9   return bestmatch;
```

5.3 Why Current ns-3 is not Symbolic Execution Friendly?

The efficiency of symbolic execution mainly depends on the number of *symbolic comparisons* that are the conditional statements involving symbolic variables, such as lines 3 and 5 in Code 4. There are two reasons. First, each symbolic comparison takes a non-trivial amount of time for the constraint solver of symbolic execution to determine whether the symbolic comparison is true or false or both with the current branch constraints. Second, if the symbolic comparison could be both true and false, symbolic execution forks the current branch (i.e., virtual machine) into two branches, a true branch and a false branch with correspondingly updated constraints. However, branch forking (i.e., virtual machine forking) takes a significant amount of time and space.

The IP routing simulation of current ns-3 is not friendly to symbolic execution, because it compares a symbolic destination IP address with each entry of a routing table. As an example, if we symbolically run Pseudocode 10 on Table 1 with a symbolic destination IP address `dst`, then there are 4 symbolic comparisons in Pseudocode 10 because it calls line 2 of Pseudocode 9 for each of 4 table entries. If the range of `dst` is from 10.1.0.0 to 10.2.255.255, there are two best matches: entry 10.1.0.0 and entry 10.2.0.0. As a result, there are finally two branches:

- one branch returns entry 10.1.0.0 (interface 2) for `dst` between 10.1.0.0 and 10.1.255.255,
- the other branch returns entry 10.2.0.0 (interface 2) for `dst` between 10.2.0.0 and 10.2.255.255.

5.4 Proposed Techniques for more Efficient IP Simulations in sym-ns-3

We redesign the simulation of IP routing in sym-ns-3 to make it more friendly to symbolic execution and thus more efficient. Specifically, we propose two techniques: 1) the table sorting technique that reduces the number of symbolic comparisons and thus reduces the number of times to call the constraint solver, and 2) the group comparison technique that reduces the number of branches (i.e., virtual machines).

The *table sorting technique* sorts a routing table according to the priority of each table entry. An entry with a longer mask is given a higher priority. If tie occurs, a shorter metric is given a higher priority. If tie still occurs, the interface is used to break the tie. All the entries with the same length of masks, same metric, and same interface belong to the same priority group. For example, Table 2 shows the sorted result of Table 1. With a sorted routing table, sym-ns-3 only needs to find the first matching priority group, but does not need to check all the remaining entries. By doing so, sym-ns-3 can reduce the number of symbolic comparisons.

Table 2: Sorted Routing Table in sym-ns-3

Destination	Mask	Interface	Metric	Priority group
10.1.0.0	255.255.0.0	2	10	1
10.2.0.0	255.255.0.0	2	10	1
127.0.0.0	255.0.0.0	0	1	2
0.0.0.0	0.0.0.0	1	1	3

The *group comparison technique* checks all the table entries within the same priority group together using only one symbolic comparison. Specifically, it replaces function `IsMatch` in Pseudocode 9 with Pseudocode 11, which returns the same result but without using a symbolic comparison. The actual code implements the logical not operator `!` at line 2 in the pseudocode using a sequence of bit-wise operations. It also replaces function `Lookup` in Pseudocode 10 with Pseudocode 12, which checks whether there is at least one matching entry in a priority group using only one symbolic comparison (i.e., line 6). Thus it generates at most one new branch for all the entries in a priority group.

Code 11: Pseudocode of Function IsMatch in sym-ns-3

```
1 IsMatch (IP Address dst, Table Entry entry)
2   return !((dst&entry.mask)^(entry.ip&entry.mask));
```

Code 12: Pseudocode of Function Lookup in sym-ns-3

```
1 Lookup (IP Address dst, IP Table table)
2   for each priority group in the table
3       flag = false;
4       for each entry in the priority group
5           flag = flag | IsMatch(dst, entry);
6       if (flag)           // check the whole group together
7           return the group;
```

We can see that these two techniques (i.e., Pseudocode 12, Pseudocode 11, and sorted Table 2) generate the same simulation results as current ns-3 (i.e., Pseudocode 10, Pseudocode 9, and unsorted Table 1). Both of them find the best match among all the table entries. For example, if `dst`=10.1.0.1, both return the entry for 10.1.0.0.

We can also see that sym-ns-3 with these two techniques has less symbolic comparisons and less branches than sym-ns-3 without these two techniques. Let's consider the same example in Section 5.3, where we search for a symbolic `dst` ranging from 10.1.0.0 to 10.2.255.255. If we symbolically run Pseudocode 12 on Table 2, then there is only 1 symbolic comparison in Pseudocode 12 because it calls line 6 only once for the two entries in priority group 1 and then returns priority group 1 (i.e., interface 2). Also note that the remaining priority groups (i.e., last two entries) of Table 2 are not checked. That is, our proposed techniques reduce the number of symbolic comparisons from 4 (see Section 5.3) to 1. In addition, there is finally only one branch as follows,

- one branch returns priority group 1 (interface 2) for `dst` between 10.1.0.0 and 10.2.255.255,

because although `dst` matches both entry 10.1.0.0 and entry 10.2.0.0, Pseudocode 12 checks their match results together only once at line 6. That is, our proposed techniques reduce the number of branches from 2 (see Section 5.3) to 1.

The complete proof of correctness and efficiency can be found in [17]. Note that although these two techniques need additional time to sort a routing table, the time to call constraint solvers and fork virtual machines is significantly more than the time to sort a routing table.

Our techniques are inspired by SymNet [19], which proposes a new symbolic execution friendly language to model computer networks including routing tables, whereas our techniques modify ns-3 code to be friendly to symbolic execution.

6 EXPERIMENTS

6.1 Simulation Setup

We evaluate the following three methods. 1) Brute force using current ns-3, which is referred to as *Brute Force*. 2) Symbolic execution using sym-ns-3 with the symbolic variable management proposed in Section 4, which is referred to as *Basic SymEx*. 3) Basic SymEx enhanced with the techniques proposed in Section 5 for more efficient IP simulations, which is referred to as *IP-Efficient SymEx*.

All the experiment results were obtained using a Ubuntu desktop with an Intel Core i5-8600 processor and 16 GigaByte of memory. The source code of sym-ns-3 and all the experiments is available at https://github.com/JeffShao96/Symbolic-NS3. More information and documents can be found at https://symbolicns3.github.io.

6.2 Exhaustive Testing on TCP Performance

In this group of simulations, we exhaustively test the TCP performance in a network shown in Figure 5, which has different and independent delays in different directions between nodes 0 and 1. Delay d_0 from node 0 to node 1 is in the range of [1, 1000] ms, and delay d_1 from node 1 to node 0 is also in the range of [1, 1000] ms. Node 0 starts to establish a TCP connection to node 1 at time 0 with an initial congestion window of 1 segment, and then sends a total

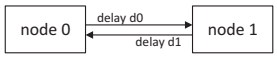

Figure 5: Network Topology of the TCP Performance Testing

of 2 data segments to node 1. The TCP performance is measured by the number of data segments received by node 1 within 2000 ms.

Brute Force runs a total of $1000 \times 1000 = 10^6$ ns-3 simulations for all possible combinations of d_0 and d_1. The simulation of each combination takes about a half second, and thus the total simulation is estimated to take about 6 days. Basic SymEx uses two symbolic variables, one for d_0 and the other for d_1, and takes about 3 hours. Basic SymEx finally generates about 140 branches (i.e., equivalence classes of d_0 and d_1 values leading to the same simulator execution paths). To help us verify the correctness of the reported TCP performance, we also print out the ranges of $2d_0 + d1$ (i.e., the time for the three-way handshake) and $3d_0 + 2d1$ (i.e., one round-trip time after the three-way handshake) for each branch. The result of all the branches is summarized and aggregated in Table 3. Note that the link data rate is 5 Mbps, and the transmission time of a segment is slightly less than 1 ms. We can see that *Basic SymEx efficiently and exhaustively reports the TCP performance for all possible combinations of d_0 and d_1 in the specified ranges, and such information is very time consuming to obtain using Brute Force with ns-3.*

Table 3: Exhaustive TCP Performance Testing by SymEx

$2d_0 + d_1$ (ms)	$3d_0 + 2d_1$ (ms)	Number of received segments
[**1999**, 3000]	[2999, 5000]	0
[1000, **1998**]	[**1999**, 3497]	1
[3, 1331]	[5,**1998**]	2

6.3 Exhaustive Testing on IP Reachability

In this group of simulations, we exhaustively test the IP reachability from node 0 to all other nodes in a network shown in Figure 6. Specifically, node 0 sends a ping packet with a destination IP in the range of 10.0.0.0 and 10.255.255.255, and reports the round-trip time (RTT) if it receives a reply. The routing table of each node is automatically created by ns-3 function `PopulateRoutingTables`.

Brute Force runs a total of 256^3 ns-3 simulations for all possible destination IP addresses. The simulation of each IP address takes about a half second, and thus the total simulation is estimated to take about 100 days. Basic SymEx uses one symbolic variable for all the destination IP addresses, and takes about 15 minutes.

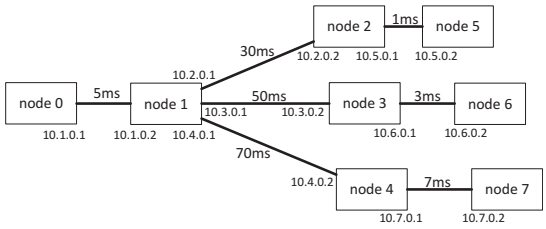

Figure 6: Network Topology of the Reachability Testing

Table 4: Exhaustive IP Reachability Testing by SymEx

Destination IP	Ping RTT (ms)
10.1.0.1	0
10.1.0.2, 10.1.255.255, 10.2.0.1 10.2.255.255, 10.3.0.1, 10.3.255.255 10.4.0.1, 10.4.255.255	10
10.2.0.2, 10.5.0.1, 10.5.255.255	70
10.5.0.2	72
10.3.0.2, 10.6.0.1, 10.6.255.255	110
10.6.0.2	116
10.4.0.2, 10.7.0.1, 10.7.255.255	150
10.7.0.2	164
All other IP addresses	No reply

Basic SymEx finally generates about 30 branches. The result of all branches is summarized in Table 4. An interesting case is that if the destination is broadcast 10.5.255.255, the ping RTT is 70 ms that is replied by node 2, and we got the same result using ns-3. We can see that *Basic SymEx efficiently and exhaustively reports the ping RTTs for all possible destination IP addresses in the specified range.*

6.4 Evaluating IP-Efficient SymEx

The previous two groups of simulations have relatively small routing tables, so there is not much difference between Basic SymEx and IP-Efficient SymEx. In this group of simulations, we use a big routing table to demonstrate the different performance of Basic and IP-Efficient SymEx. Specifically, we manually add the following n additional entries to the routing table of node 2 in Figure 6 for the reachability simulations.

Table 5: Additional Table Entries for Node 2

Destination	Mask	Interface	Metric
10.5.1.0	255.255.255.0	2	default
...	...	...	...
10.5.n.0	255.255.255.0	2	default

The simulation results shown in Figure 7 indicate that the number of branches generated by Basic SymEx increases proportionally as the number n of additional table entries increases, whereas that of IP-Efficient SymEx remains unchanged. Accordingly, the testing time of Basic SymEx increases proportionally as n increases, whereas that of IP-Efficient SymEx increases only slightly.

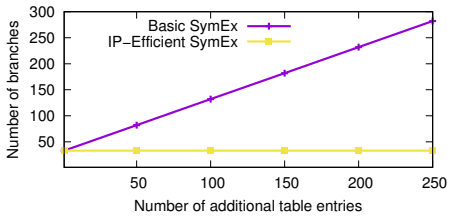

Figure 7: IP-Efficient is More Efficient than Basic SymEx

7 CONCLUSIONS

In this paper, we present our current progress on sym-ns-3 for more efficient exhaustive testing. Specifically, we present our design choices and implementation details on how we extend ns-3 to support symbolic execution, and also the significantly improved testing speeds of sym-ns-3 over ns-3. In the future, we plan to study and improve the performance of sym-ns-3 for other types of symbolic variables, such as symbolic data rates of channels.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of USENIX ATC*. Anaheim, CA.
[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of USENIX OSDI*. San Diego, CA.
[3] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (February 2013), 82–90.
[4] Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. 2011. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *Proceedings of USENIX ATC*.
[5] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of USENIX NSDI*. San Jose, CA.
[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems* 30, 1 (February 2012).
[7] Oscar Dustmann. 2013. Symbolic Execution of Discrete Event Systems with Uncertain Time. *Lecture Notes in Informatics* S-12 (2013), 19–22.
[8] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *Proceedings of USENIX NSDI*. Santa Clara, CA.
[9] Endadul Hoque, Omar Chowdhury, Sze Yiu Chau, Cristina Nita-Rotaru, and Ninghui Li. 2017. Analyzing Operational Behavior of Stateful Protocol Implementations for Detecting Semantic Bugs. In *Proceedings of IEEE/IFIP DSN*.
[10] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Proceedings of USENIX NSDI*.
[11] James King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.
[12] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. 2011. Finding Protocol Manipulation Attacks. In *Proceedings of ACM SIGCOMM*. Toronto, Canada.
[13] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano, and Dejan Kostic. 2012. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proceedings of ACM CoNEXT*. Nice, France.
[14] Network Simulator 3. . A Discrete-Event Network Simulator for Internet Systems. https://www.nsnam.org/.
[15] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. Analyzing Protocol Implementations for Interoperability. In *Proceedings of USENIX NSDI*. Oakland, CA.
[16] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment. In *Proceedings of ACM/IEEE IPSN*. Sweden, 186–196.
[17] Jianfei Shao. 2022. Symbolic ns-3 for Efficient Exhaustive Testing. Master Thesis, School of Computing, University of Nebraska-Lincoln.
[18] JaeSeung Song, Cristian Cadar, and Peter 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.
[19] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of ACM SIGCOMM*. Brazil, 314–327.
[20] Wei Sun, Lisong Xu, and Sebastian Elbaum. 2015. SPD: Automatically Test Unmodified Network Programs with Symbolic Packet Dynamics. In *Proceedings of IEEE GLOBECOM*. San Diego, CA.
[21] Minh Vu, Lisong Xu, Sebastian Elbaum, Wei Sun, and Kevin Qiao. 2022. Efficient Protocol Testing with Temporal Uncertain Events using Discrete Event Simulator. *ACM Transactions on Modeling and Computer Simulation* 32, 2 (April 2022), 1–30.