Symbolic Execution of Distributed Software

MEng Final Report

Written by Milen Dzhumerov

Supervisor: Peter Pietzuch

Second Supervisor: Cristian Cadar

Contents

Abstract

Verification of program correctness is one of the most important aspects of software development. One particular approach to automatically testing programs is symbolic execution. This report documents the development of a system that enables symbolic execution of distributed software which uses network sockets for communication.

We explore the design and implementation of the essential facilities to simulate distributed software – namely, the networking and the filesystem layers. Furthermore, we examine the effects and performance of automatically injecting system call failures and constraining symbolic input. We also present Minvariant, a new language used to express invariants over network nodes. Finally, we evaluate the performance of our system using real-world production software[3].

Acknowledgments

First and foremost, I would like to thank my parents for their continuous support and encouragement to pursue my dreams throughout my life. I would like express my deepest gratitude towards my two supervisors, Peter Pietzuch and Cristian Cadar, for their relentless guidance, ideas, advice and constructive criticism without which this project would not have been possible. I will also be forever grateful to Tony Field, Susan Eisenbach, Ian Hodkinson and Iain Phillips for "opening" my eyes and completely changing the way I see the world. Finally, I'd like to thank Chris Emery for always challenging my views, providing me with invaluable guidance and being there along every step of the journey.

1 Introduction

In this section, we provide the motivation behind this project, clarify our aims and approach, provide an overview of our contributions and finally present the structure of the report.

1.1 Motivation

Ever since the beginning of the software industry, computer programs have been used in an increasing number of industries. The majority of devices that we use on an everyday basis are powered by software – whether that would be cars, phones or TVs, it is virtually impossible to find places where software is not being used. Together with the increasing usage of software, there has been an explosion in its complexity, both internal and external.

The exponential increase in software complexity brings almost as many problems as it solves. It has become particularly hard to ascertain the correctness of software – even for mission-critical systems. In 1996, the Ariane 5[4] rocket suddenly changed its flight direction path about half a minute into flight and had to self-destruct itself due to very high aerodynamic forces. The problem originated from the software – there was a data conversion overflow that was not handled which cascaded in the upper layers and started a chain reaction that eventually led to the destruction of the rocket. Those accidents are not isolated cases. Another high profile software disaster was the Therac-25[9] radiation therapy machine. During its use, patients were given enormous overdoses of radiation.

Contributing even more to the already complex software landscape was the introduction of computer networking, whereby programs that run on different machines (virtual or real) can communicate with each other. One of the challenges when it comes to verification of distributed software is the non-deterministic behaviour of the network nodes and the virtually infinite interaction combinations. The non-determinism creates an explosion in possible state combinations which in turn affects the complexity of the software.

Given the challenges that the computer industry has always been facing, it was obvious at an early stage that programs need to be verified for correctness, especially in mission-critical environments. There are two broad types of software verification that are currently in use:

- **Dynamic Verification**

  Dynamic verification involves running the software in question. There are several ways dynamic verification is performed – either manually by testers or in an automated fashion, but in all cases the actual behaviour of the program is checked.

- **Static Verification**

  In contrast to dynamic verification, static verification does not run the software but only analyses the source code or an intermediate representation of it. One of the most time-consuming and expensive methods of static verification is formal verification whereby the behaviour of the software is mathematically modelled and then properties are proven based on that model.

In this report, we focus our attention on software testing and some of its associated issues. There has been an enormous increase in automated software testing over the last 10 years, which gave birth to a development processes named Test-Driven Development[1] (TDD). Writing unit tests is a common practice in most companies, especially when working on green-field projects because it is a lot easier to write unit tests when the code is designed with testability in mind as opposed to trying to retrofit unit testing into an existing architecture. Even though unit testing is widely practised in the industry[5], the vast majority of software in current use does

not have any automated test suites and relies on quality assurance (QA) teams to ensure its correct operation. There are even cases where very old software keeps getting used without modification due to the fragility of the code and the potential disruption to business.

A particularly hard class of applications to test, either via automated tests or manually, are distributed systems. There have been no general solutions to the problem of testing arbitrary distributed systems – the reason comes from the complexity due to multitude of network topologies and the non-deterministic interaction of all the participants. It is particularly hard to automatically test those systems due to several reasons:

- **Test Environment**

  There is an inherent complexity in setting up various network topologies, distributing the final product and testing it. For many projects, the costs of the infrastructure and its maintenance outweigh the benefits and the software might only be tested as a single entity (and its internal subcomponents).

- **Code Coverage**

  Setting up multiple test environments does not guarantee that all code-paths will be tested. In distributed systems, there are many potential edge-cases that rarely occur in practice (excessive packet loss, packet corruption, etc.) which have the potential to bring the whole system down. If the testing environment does not have the ability to simulate those, there will be important code paths that would have never been tested properly.

- **Distributed Properties**

  Distributed systems usually interact to achieve a common goal and certain properties of the system exist that should be invariant across the nodes. Unless there are specific mechanisms exposed to access the necessary information for verification, their truth value cannot be verified. It should be noted that even if a protocol that establishes certain properties over the system might be proven to be correct, its implementation might have bugs which invalidate the properties. In practice, a lot of security-related issues are due to the implementation details and not the protocol.

Manually testing distributed systems is a virtually impossible task for any non-trivial piece of software, mainly due to the enormously high number of possible configurations and interactions. Another problem associated with the correctness of distributed software is that the correctness of a single entity or a particular configuration does not give any guarantees for the system correctness in other settings.

1.2 Aims

The main goal of this project is to investigate the feasibility of automatically testing distributed software by building a system that allows to symbolically execute programs that communicate over a network. One of the crucial requirements for the system is that it should be able to test arbitrary programs without the need to modify them in any significant ways. In addition, any testing methodologies that are introduced should be orthogonal to any pre-existing testing practises.

1.3 Approach

We will be building the system by modifying an existing symbolic virtual machined named KLEE[2]. Building upon the foundations provided by another system allows us to focus on the issues relevant to symbolic execution of distributed software.

One of the reasons for taking a symbolic approach is due to the inherent capabilities of the technique to provide very high code coverage without manually writing tests. In particular, KLEE was chosen as a foundation for the following reasons:

- **Built on LLVM**

  Because KLEE actually works with LLVM bitcode, the source language of the distributed system can vary. Thus we can test any system as long as there is a compiler from the source language to LLVM bitcode. At the the time of writing, C, C++ and Objective-C can all be compiled to LLVM bitcode.

- **Unmodified Source Code**

  Because KLEE is essentially a symbolic execution engine that interprets LLVM bitcode, there is no need to modify the original source code. Intrusive testing systems that require changes to the source code (or complete rewrites) have an associated cost and can inadvertently modify the actual behaviour of systems.

1.4 Contributions

This project made the following contributions:

- **Symbolic Execution of Distributed Software** We designed and implemented a system which can be used to symbolically execute arbitrary distributed software which uses network sockets for communication.

- **Replay Framework** We designed and implemented a replay framework that can be used to reproduce issues found by our system.

- **Failure Model** We designed and implemented a failure injection model while also evaluating its effects on code coverage metrics.

- **Distributed Invariants** We designed a language to express invariants over network nodes and implemented support for it as part of our system.

- **Web Server Evaluation** We evaluated the performance of our system on a minimalistic production web server[3]. We uncovered 2 critical bugs which lead to the server becoming inoperative. While evaluating the effects of symbolic data in HTTP requests, we managed to achieve 99.8% of the code coverage upper bound for our test scenario.

- **Scalability Evaluation** We evaluated the scalability characteristics of our system by performing a variety of synthetic tests whose aim was to quantify the practical limits when symbolically executing distributed software.

1.5 Report Structure

We continue with the Background (section 2) which provides introduction to concepts that are essential to the project. We also provide an overview of any relevant previous work in the symbolic execution and software verification space, making sure provide some details about LLVM itself.

Afterwards, in section 3 we take a deeper look at the specifics of how our system works from a high level and provide the rationale for our design decisions. Specific issues pertaining the limits of our system are also discussed.

In Replay Framework (section 4) we present a way for issues found by our system to be reproduced by running the software under test natively instead of simulating it symbolically. In Failure Model (section 5) we present a way to artificially inject low-probability events in order to increase code coverage. In Distributed Invariants (section 6), we introduce a framework and domain-specific language that allow the expression and verification of invariants across distributed systems.

In Implementation (section 7), we reveal the most important details about how our system works. We take a look at how we have implemented networking, our own filesystem and how we have integrated our changes in a backwards-compatible way.

In Evaluation (section 8), we try to quantify the performance of our system in a series of tests. We look at symbolically testing a web server and the inherent challenges involved in symbolically executing software that interacts with many subsystems, include the network and filesystem. We also cover a range of synthetic tests in order to asses the hard limits of our implementation. Furthermore, we discuss fundamental limitations that affected the utility of our system.

In Development Methodology (section 9), we take a quick look of the development process and the testing methodology used. Finally, in Conclusion (section 10), we present a retrospective look of the project goals, the targets that we hit, the lessons learned and provide an overview of areas for future work.

2 Background

Symbolic execution of distributed application is an active research area and before we look into previous work, we will cover the basics of how symbolic execution works.

2.1 Symbolic Execution

When it comes to assuring the quality and behaviour of software, the methods vary between the two extreme points: completely formal static analysis and exhaustive manual testing. Each end of the spectrum has its own set of advantages and disadvantages:

- **Formal Verification**

  Formally proving the correctness of programs has the advantage that we can be absolutely sure of the properties that we can prove – there is no doubt whether those properties will hold only sometimes and might be invalidated in certain cases. On the other hand, providing formal proofs is not only impractical due to costs for most non-mission-critical software but also requires a complete formal specification so that we can determine whether the software satisfies it.

- **Manual Testing**

  Most, if not all, programs are tested before being released for production use. The amount of testing and the code covered by this technique greatly varies between software vendors and products. One of the advantages of testing the products manually is low cost of the method – all that is needed is a Q&A team. On the other hand, a lot of resources can be wasted by testing the same features on every release to ensure that there are no regressions.

  Moreover, there is no reliable way to determine how many of the possible test cases have been covered and how much of the code was tested – manual testers usually focus on the most common use cases for the software in question.

Using symbolic execution for testing stands in-between formal verification and manual testing. It is a testing approach that has been in increasing use over the past decade while also finding real bugs and vulnerabilities in applications.

The basic premise of symbolic execution is as follows: instead of executing programs on real values, we use symbolic values which represent arbitrary values, possibly constrained. If we compare it with manual testing, one of the advantages of symbolic execution is that we do not only explore the behaviour of the program for particular values but for a set of classes of values.

One of the most important aspects of symbolic execution is the handling of branching instructions which involve symbolic values. Because the symbolic value represents a class of values, it is possible that both paths can be taken. A symbolic execution engine would in this case would try to explore both code paths, adding the constraints implied by the branch instruction to each execution path.

The symbolic execution of the program can be visualised as a tree which defines the execution path being simulated. Each leaf node represents a program state while non-leaf nodes represent branching[1] due to executing branch instructions on symbolic values. By definition, the leaf nodes are characterised by the tree path followed to reach them – each time a decision is made to follow a particular tree child node, an additional constraint is added to the current path condition. In order to demonstrate the concepts in a more straightforward, a symbolic execution example follows.

[1] Also referred to as forking, which should not be confused with the `fork` call. Unless explicitly stated otherwise, forking refers to branching.

2.1.1 Example

We will illustrate how symbolic execution works by walking through a piece of code that caused all Microsoft's Zunes to be stuck in an infinite loop on the 31st December 2008. The code has been modified so that a symbolic execution engine can find the error, although the change is immaterial to understanding the principle of symbolic execution.

Before we take a look at the execution, we need to point out the important aspects of the code. Lines 7–9 were added so that the symbolic execution engine can catch the bug. The issues lies in the fact that no progress is made in the loop for the case when line 8 gets executed, so the code will be stuck in an infinite loop.

Listing 1: Showing an infinite loop in the clock driver on the Microsoft Zune.

```
1  while (days > 365) {
2    if (IsLeapYear(year)) {
3      if (days > 366) {
4        days -= 366;
5        year += 1;
6      }
7      else {
8        assert(0);
9      }
10   }
11   else {
12     days -= 365;
13     year += 1;
14   }
15 }
```

Assume that the symbolic execution engine is about to execute the code on line 1 and that the variables `days` and `year` are both symbolic without any constraints. Figure 1 shows a graphical representation of the execution.

1. `while (days > 365)`

   This line of code gets translated as a branch instruction when compiled to machine code. In this case, the symbolic execution engine would see a comparison based on a symbolic variable. Execution would need to branch in two – one where `days` is greater than 365 and the other where `days` is less than or equal than 365.

2. `if (IsLeapYear(year))`

   The branch which follows the path where `days` is greater than 365 will encounter the next branching instruction that checks whether the year is leap. Again, there are two possible outcomes and both paths will be followed.

3. `if (days > 366)`

   The execution path where the year is leap will need to execute the above branch instruction. Again, both paths will be explored. We are only interested in the path where `days` is less than or equal to 366.

4. `assert(0)`

   Following the path where the `if` check fails leads to a failing assert. The execution engine has recorded the constraints for all the symbolic variables that lead to the assertion to be reached. A regression test case can be generating by solving the constraints and picking any sets of values for the variables that satisfy the constraints.

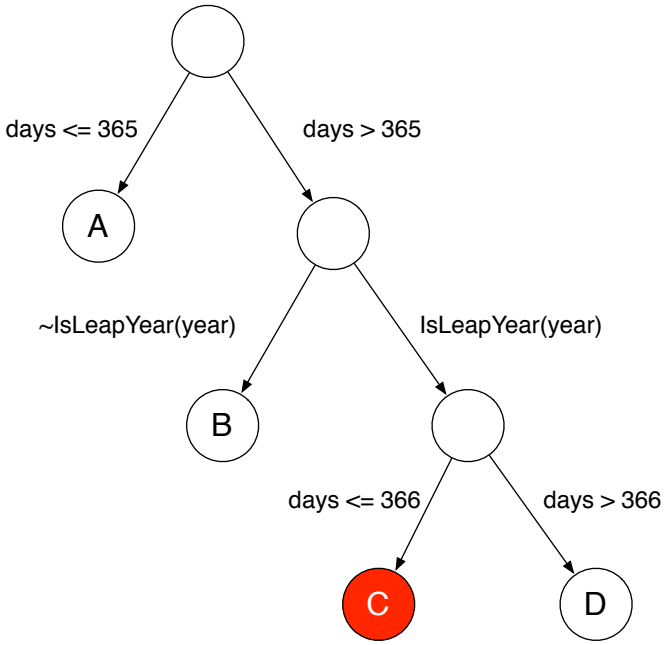


Figure 1: Symbolic execution tree for the clock driver code. Note state C contains a failing `assert` that will trigger the generation of a test case. In order to generate concrete values the symbolic variables, all constraints along the path must be satisfied. For example, to reach state C `days` must equal 366 and the `year` must be leap.

2.2 LLVM

LLVM[8] stands for Low Level Virtual Machine and is a flexible compiler infrastructure that was started in 2000. Since its inception, the project's popularity and use increased manyfold and currently underpins the development tools for Mac OS X and iOS.

One of the advantages of using LLVM are the great number of front-ends available which will only be increasing in the future. One of the consequences is that tools built on top of the LLVM infrastructure can be used in conjunction with a variety of source languages.

At the core of LLVM sits the intermediate representation (IR). It has three isomorphic forms:

- **Assembly**

  The assembly format is user-readable and very similar to assembly code for traditional processors.

- **Bitcode**

  The bitcode form is a tightly-packed binary form which is mainly used when the IR has to be exchanged between different programs. Tools built on LLVM usually work with bitcode.

- **Internal Representation in C++**

  LLVM itself is written in C++ and there is a class hierarchy that represents the IR – it also has support for easy conversion between the various forms. Tools usually read in bitcode and then transform it to C++ objects which represent it and perform their work on that representation.

The LLVM intermediate representation is fairly high-level – for example, function calls are not expanded to pushing & popping stack frames. The representation also is heavily typed and can include metadata that is used during the optimisation passes.

Listing 2 shows a simple program written C that contains a function to check whether a character is an ASCII digit. The program's entry point just calls the function and prints the results to the standard output.

Listing 2: C program to check if a character is an ASCII digit.

```c
#include <stdio.h>
#include <string.h>

int is_ascii_digit(char ch) {
  if(ch >= '0' && ch <= '9')
    return 1;
  return 0;
}

int main(int argc, char* argv) {
  int is_ascii = is_ascii_digit('5');
  printf("ASCII: %d\n", is_ascii);
  return 0;
}
```

Listing 3: LLVM assembly for the program in Listing 2.

```
define i32 @is_ascii_digit(i8 signext %ch) nounwind readnone ssp {
entry:
  %0 = add i8 %ch, -48                            ; <i8> [#uses=1]
  %1 = icmp ult i8 %0, 10                         ; <i1> [#uses=1]
  %.0 = zext i1 %1 to i32                         ; <i32> [#uses=1]
  ret i32 %.0
}

define i32 @main(i32 %argc, i8* nocapture %argv) nounwind ssp {
entry:
  %0 = tail call i32 @is_ascii_digit(i8 signext 53) nounwind ; <i32> [#uses=1]
  %1 = tail call i32 (i8*, ...)* @printf(i8* noalias ..., i32 %0) nounwind ; <
      i32> [#uses=0]
  ret i32 0
}

declare i32 @printf(i8* nocapture, ...) nounwind
```

Listing 3 shows the code compiled to LLVM assembly. There are several notes of interest:

- **Types**

  Function definitions contain the return types and also the types of its arguments, together with any metadata. In addition, function calls also contain the types of their arguments and the return type – the intermediate representation can very easily be verified for type-safety.

- **Metadata**

  Function declarations include metadata (known as function attributes) which allows the compiler to emit more efficient code.

– **nounwind**

Indicates that the function never returns with an unwind or other exceptional control flow.

– **readnone**

Indicates that the function only needs its arguments to compute the result – it does not dereference any pointers and does not access any mutable state, as far as the caller is concerned.

– **ssp**

Indicates that during code generation, a "stack smashing protector" should be emitted. It is usually a random value pushed on the stack before the local variables that is checked on function return whether it was overwritten (thus detecting memory corruption).

2.3 KLEE

KLEE[2] is a symbolic execution engine that is built on top of the LLVM infrastructure. KLEE can automatically test software and also has the ability to generate regression test cases for any bugs that are found.

2.3.1 Overview

KLEE is one of the latest tools in the symbolic execution area that has produced tangible results and found bugs in already heavily tested software. It can achieve very high code coverage at a fraction of the costs if it had been done manually and can potentially find serious security bugs.

2.3.2 Operation & Architecture

At its core, KLEE is a symbolic interpreter for LLVM bitcode which has two fundamental goals: to cover as many execution paths as possible and to detect any illegal operations (for example, dereferencing `NULL` pointers). Internally, each symbolic process (referred to as *state*) represents an instance of the application being tested – it has its own program counter, registers, heap, stack and quite importantly – a path condition. The path condition is a set of constraints on the symbolic variables which lead to the current process state.

When KLEE encounters conditional branches involving boolean expressions, it needs to decide how to proceed with the execution. Firstly, it asks the constraint solver whether the condition is either provably true or false – in which case, the instruction pointer is adjusted appropriately. If the value of the branch condition cannot be determined, KLEE will clone the current state and then explore both paths, corresponding to the two possible branch conditions. While interpreting the bitcode, safety checks are performed – for example, loads and stores are bound-checked, while any arithmetic is checked for division by zero.

Given that every branch instruction that operates on symbolic data can result in the cloning of a state, symbolic execution and KLEE suffer from the exponential state explosion problem. In order to minimise the impact of cloning and allow for a large number of states to be resident in memory, aggressive copy-on-write is employed at the memory object level (as opposed on memory page level). Due to the implementation of the heap as an immutable map, parts of the heap can also be shared across states which makes it possible to clone in $O(1)$.

2.3.3 Query Optimisation

During the evaluation stage of KLEE, it was revealed that the majority of the time is spent solving constraints due to the fact that the logic produced by KLEE is NP-complete. Some of the optimisation techniques that were employed follow:

- **Expression Rewriting**

  Expressions can usually be rewritten when they involve constants or can be reduced to less complex ones. For example, $x + 5 + 3 < 9$ can be rewritten as $x < 1$.

- **Constraint Set Simplification**

  While KLEE executes programs symbolically, it adds constraints each time branch instructions necessitate following multiple execution paths. In practice, multiple constraints refer to the same symbolic variable and the constraints usually become more concrete which allows for the wider constraints to be eliminated. For example, a constraint $y > 6$ could have been added and subsequently $y = 10$ added as well. Substituting the value of 10 into the first constraints makes it evaluate to *true*, thus it can be eliminated.

- **Implied Value Concretisation**

  It is possible that certain constraints imply actual values for symbolic variables. In that case, the concrete value inferred can be written back to memory which will result in very fast execution for any expressions that reference the symbolic variable later during the execution. For example, given the expression $x - 3 = 10$, $x = 7$ can be inferred and then substituted by a constant expression.

- **Constraint Independence**

  The constraint solver is used in a specific way: it is given a set of constraints and a query. Minimising the number of constraints can significantly speed up the time spent evaluation queries. It should be noted that not all constraints are needed when computing the answer – only constraints that refer, directly or indirectly, to the variables that are part of the query. Thus eliminating any irrelevant constraints provides significant time savings. For example, lets assume that the current path condition contains the following 3 constraints: $\{a + b < 20, b < 5, c > 15\}$ and the query is whether $a = 4$ – the third constraint, $c > 15$, is irrelevant in this situation while the other 2 are needed.

Query optimisations are very important because they provide significant savings to the run time. Running the coreutils suite symbolically under KLEE revealed the following data:

- Without any optimisations, constraint solving absolutely dominates the run time – accounting for 92% of the execution time.

- Turning on all optimisations reduces the the overall run time by almost 300% and minimises the time spent solving constraints to about 41%.

2.3.4 State Scheduling & Environment

When KLEE symbolically executes a particular program, it is simultaneously exploring many execution paths and conceptually running multiple instances of the program with different inputs. It is very important to correctly schedule the execution of those states, so that it results in increased code coverage and bugs found. KLEE combines two strategies in a round-robin fashion to provide the best possible exploration pattern:

- **Random Path Selection**

  The random path selection algorithm works by starting at the root of the state tree and selecting a random path to follow at each branch, thus the probability of selection of each branch is 50%. The strategy has two properties that make it better than random state selection: firstly, states higher in the tree are more likely to be selected and secondly, it avoids starvation if a subtree starts to rapidly fork. States higher in the tree are generally more desirable because they have less constraints, thus can potentially cover more execution paths.

- **Coverage-Optimised Search**

  KLEE also uses some heuristics to select states that are more likely than others to increase the code coverage ratio. One of those heuristics is the minimum distance to the next uncovered instruction.

Most applications interact with the environment to some extent or other, whether that would be reading files, performing networking or getting the values of environmental variables. Ideally, when system calls are symbolically executed, we want to return all possible values that can be produced in order to explore as many execution paths. KLEE handles environmental functions by using *models* which are written in C to simulate the behaviour of the functions they implement. The execution engine also has the ability to simulate the failure of system calls, thus exercising code paths that very rarely get tested.

2.4 KleeNet

KleeNet[13] is a system, built on top of KLEE, designed to execute unmodified programs written for sensor networks. Wireless Sensor Networks (WSN) are usually deployed in remote places without any infrastructure and left for long periods of time. Thus it is very important that the system can operate reliably without any intervention, in an environment where the networking can be very lossy and sensors can fail at any point in time. Due to the usually high costs associated with maintaining and repairing WSNs, it is very desirable to achieve very high code coverage.

Bugs that are found in WSNs are usually caused by low probability non-deterministic events that are hard to simulate in a controlled test environment, such as:

- **Corrupted Network Packets**

  In a real environment where WSNs are deployed, the nodes are usually connected using unreliable ad-hoc infrastructure where packet corruption does occur and can cause faulty node behaviour.

- **Complex Node Interaction**

  Due to the non-deterministic nature of networking events and the resulting node interaction, only a very limited fraction of the possible scenarios can be tested in practice. Symbolic execution tries to explore as many combinations as possible.

- **Non-Deterministic Events**

  When WSNs are deployed in remote locations, a number of events can occur, like sporadic node reboots or complete node loss, which lead to scenarios which have not been thoroughly tested and can have unpredictable results on the network behaviour.

2.4.1 Contributions

KleeNet provided four important contributions for testing WSNs:

- **Coverage**

  KleeNet allows for the symbolic execution of unmodified network applications. By driving the execution by using symbolic inputs from the environment, a much higher code coverage can be achieved.

- **Non-Determinism**

  KleeNet simulates the loss, corruption and duplication of network packets during the symbolic execution which increases the chance of finding corner-case bugs.

- **Distributed Assertions**

  KleeNet has the ability to perform assertions on the distributed state of the system. This is useful when the correctness and convergence of network protocols needs to be tested. The only disadvantage is that the assertions have to be written manually by people who understand the software and protocols being tested.

- **Repeatability**

  KleeNet has the ability to generate and replay test cases whenever any bugs are found. This feature provides immense help when trying to localise the issue and fix the root cause.

2.4.2 Concept & Design

The basic operation of KleeNet can be demonstrated with a simple example. We can assume the existence of different network nodes: A, B and C. When a packet is sent from a node A to a node B, there are several possible outcomes, all of which are simulated:

- **Invalid Packet**

  When a network packet is received, it is checked for validity. It is possible that the packet is malformed and thus discarded.

- **Local Delivery / Forwarding**

  After a packet has passed the validity check, it might either be destined for the current node, B, or needs to be forwarded to another node.

KleeNet also injects non-deterministic events during symbolic execution. It is possible that in any of the possible execution paths that were explored, a node shutdown or reboot be injected, consequently exploring a yet another possible event combination.

Symbolic input plays a very important role in achieving high code coverage when symbolically executing programs. KleeNet users have the ability to mark variables as being symbolic (a feature inherited from KLEE). Since distributed network applications are mostly driven by input from the network, a lot of scenarios can be explored by marking network packets as symbolic. For example, programs usually have logic that uses the packet header in order to properly handle the data that has arrived. Making the header symbolic will exercise the code to ensure that all possible headers, including malformed ones, are properly handled. One important observation that KleeNet revealed was that the execution time, in practice, did not grow exponentially because the sensor network applications were designed to work in a resource constrained environment which kept the possible execution paths to a minimum.

KleeNet features a node model that simulates low-probability non-deterministic events, such as node reboots and node outages. These can occur in WSNs deployments due to bugs in the operating systems or due to hardware failures. Such failures are usually very hard to test against and generally guide the code into corner cases that might not be handled properly, consequently revealing serious flaws in the software. A node reboot event is implemented by branching the state selected for reboot in two: in the first state, execution continues as normal while the second state is reset and reinitialised so that it simulates relaunching the application. In terms of complexity, node reboots do not have direct impact on the number of execution paths because rebooting resets the state to the initial state, which has already been visited.

KleeNet also includes a network interaction model that increases the overall code coverage. It can simulate three types of non-deterministic events:

- **Packet Loss**

  Packet loss usually occurs on unreliable networks and is especially common in WSNs. The system can inject symbolic packet losses where the packets never reach their destination, thus testing the execution paths that deal with those infrequent events.

- **Packet Duplication**

  Similar to packet loss, packet duplication happens quite often in deployed WSNs. In practice, it was found that packet duplication can lead to disastrous behaviour and uncover complex interaction bugs.

- **Packet Corruption**

  Packet corruption attempts to discover new execution paths by randomly corrupting arbitrary packets. Corruption usually reveals bugs in the packet parsing and verification logic.

The number of packets that are dropped, corrupted or lost can be configured before symbolic execution begins. In practice, it was sufficient to set the number of packet failures to a relatively low number (e.g., 20) in order to cover all possible execution paths that deal with such circumstances. It was revealed that increasing the amount of such events did not lead to discovery of further bugs because it did not give rise to uncovered distributed application behaviour.

It is usually very hard to analyse the consistency of distributed state in conventional testing environments. KleeNet provides the ability to specify assertions across the distributed application state which makes it easy to find bugs in protocols or protocol implementations. It should be noted that distributed assertions must be manually written by people who have domain specific knowledge of the code that is being tested.

2.4.3 Evaluation

KleeNet was successful in finding critical bugs in a TCP/IP stack that were caused by non-deterministic events, such as packet loss and packet duplication. One of the bugs found can be rated as highly severe and resulted in refusal to accept any further connections.

Several limitations were identified while trying to test and debug applications:

- **Symbolic Input**

  KleeNet suffers from the state explosion problem with relatively small-sized symbolic inputs, even with a low number of network nodes. The problem is amplified further by the injection of non-deterministic events, such as packet loss, duplication and so on. In

order to work around the problem, domain-specific knowledge of the application being tested can be used to minimise the amount of symbolic input in order to avoid simulating an excessive number of execution paths.

- **Automation**

  Symbolic inputs and distributed assertions have to be manually specified. Knowledge about the application that is being tested is required in order to introduce symbolic input that will result in producing relevant test traces and provide a high code coverage ratio. On the other hand, the injection of non-deterministic events is automatic and there is no need for user intervention.

- **Application Domain**

  KleeNet is designed for symbolically testing applications by implementing application-level networking primitives, as opposed to being designed for MAC-level debugging.

2.5 MoDist

MoDist[15] is a model checker that was designed to test unmodified distributed applications which run on unmodified operating systems. In contrast to KleeNet, it does not use symbolic execution to guide the execution but instead influences the system's behaviour by simulating non-deterministic events, such as packet reordering and node outages.

Model checking tools work on a simple principle – they try to test applications by trying to exhaustively traverse all execution paths. In order for model checkers to be able to guide the execution, the system under test needs to expose a set of possible actions that can be taken at certain points during run time. Usually, manual modifications to the systems are needed to be able to test the applications and in some cases, complete rewrites in domain-specific languages are necessary. Performing such modifications, especially on large-scale production software, places an enormous burden on the developers.

MoDist takes a different approach and does not require any modifications to the systems under test or the operating system. It achieves this by inserting a layer between the OS and the application code (transparent interposition) which allows it to infer the set of possible actions. Once the actions have been inferred, a model checking engine makes decisions about which one to take.

2.5.1 Overview

MoDist is tailored towards testing distributed systems that run as separate OS processes and communicate using network sockets.

It consists of two logical components: an interposition layer that is injected into every process that is being tested and a separate backend (another OS process) that communicates with the interposition layers via remote procedure calls. The interposition layer is kept as simple as possible to avoid modifying the original behaviour of systems under test. The backend is composed of five logical parts: a dependency tracker, a failure simulator, a virtual clock, a model checking engine and a global assertion module.

The interposition layer's aim is to determine the set of possible actions dynamically at runtime and let the backend have the ability to schedule them. This is achieved in two steps, by firstly suspending the processes just before they execute actions while also notifying the backend and secondly, by letting the backend control the outcome of the action – whether it will succeed or fail. Even though the interposition layer depends on the specific OS, the backend does not and can be reused regardless of the underlying platform.

The dependency tracker in MoDist has the responsibility to control the action dependencies between the different processes. It can compute the set of possible actions for a particular process which is needed to guide the execution – if the backend instructs the interposition layer to perform an action that will result it to block in the OS, then the target system will be deadlocked. The failure simulator's role is to inject rare events that usually do not occur very often and are very hard to produce in a test environment. The failures are injected deterministically so that any program errors can be reproduced with ease. MoDist features a virtual clock that resides in the backend which has two responsibilities:

- Heuristically find timers and fire them as instructed by the model checking engine. In practice, timeouts are checked soon after calls to `gettimeofday`.
- Ensure the consistency of time across the distributed system.

The model checking engine sits at the core of the backend and drives the execution of distributed system. Search heuristics and optimisations to reduce the state space are employed to provide adequate performance and provide high code coverage. MoDist also has the ability to check assertions across the state of the distributed system which can uncover interaction bugs that cannot be found by using assertions on a per node-basis.

In addition to providing global assertions, MoDist detects two more types of errors:

- **Fail-Stop Errors**

  Those are errors that are triggered by the operating system when the program tries to access invalid memory, divides by 0 or encounters a failing `assert`.

- **Divergence Errors**

  MoDist keeps track if the systems under test become unresponsive for an extended period of time (10 seconds by default) due to a deadlock or an infinite loop. If it determines that a system has stopped responding, it will report it as a divergence error.

2.5.2 Implementation

The implementation of MoDist is based on intercepting calls to the WinAPI, which was chosen due to its ubiquitous use by libraries and application. Two main goals were set from the start:

- **Deterministic Execution** The execution traces that MoDist produces should be deterministic so that errors can be reliably reproduced and to avoid any false positives.
- **Tailored towards Distributed Systems** MoDist was tailored towards distributed systems which made it easier make certain assumptions and avoid more general solutions that would have suffered in efficiency.

The interposition layer sits between the application and the OS by intercepting function calls to WinAPI and allows the backend to control their behaviour. It has two main aspects: its implementation complexity plays an important role because it needs to avoid changing the behaviour of the host application and also be deterministic & consistent. The second aspect of the layer is the IO abstraction: in order to make the backend more portable and also simpler, it needs to abstract the specifics of the IO subsystem in WinAPI. Because MoDist is tailored for testing distributed systems, only the relevant subset of the WinAPI functions had to be interposed, namely the networking, time, filesystem, memory and thread related sets. Most of the WinAPI wrappers followed a simple model: send a message using RPC to the backed and wait for a reply whether to inject a failure or proxy the call to the actual

WinAPI function. Abstracting the Windows Network IO was a complex task due to several factors: there is a relatively high number of functions dealing with IO, they are asynchronous and are in themselves non-deterministic due to network failures. MoDist addresses those issues by abstracting related network operations in a single entity in order to reduce the size and complexity of the implementation while implementing asynchronous IO by using a synchronous API in a proxy thread. In addition, extra care was taken with the placement of error injections to ensure error reproducibility.

The dependency tracker in MoDist plays a very important role by trying to prevent the deadlock of the target systems. It does so by defining whether any two actions are dependent as follows: two actions are dependent if and only one can enable the other or executing them in a different order leads to different states. MoDist can compute the set of possible actions at any point and exactly one of them is chosen to preserve determinism (all other actions are paused).

MoDist also features a virtual clock manager that has the ability to simulate timeouts and accelerate the passage of time. In practice, most programs use *implicit* timers whereby they read the current time and perform some arithmetic on it to check for a timeout. Analysing Berkeley DB revealed that in over 92% of the cases, only a small number of instructions separate the function call to query the current time and the check for a timeout. Handling implicit timers follows a 3 step process:

1. **Static Analysis** The code is statically analysed for all the function call sites that return the current time. Doing the analysis statically means that there is no associated run time penalty.

2. **Data Flow Analysis** The data flow of the returned time value into variables is tracked.

3. **Time Value Branching** If there is a branch instruction involving the time value, a constraint solver is run in order to determine the values that need to be returned by the function that returns the current time so that both branches are traversed. Later during execution, the virtual clock manager will try to cover as many possible execution paths by making sure it returns values that will cover both branches (on separate occasions).

2.5.3 Evaluation

MoDist was applied to three real-world distributed production systems:

- **Berkeley DB**

  Berkeley DB is a library that provides an embedded key-value database. It is widely used, most notably by Subversion and MySQL.

- **MPS**

  MPS is an implementation of the Paxos algorithm by a Microsoft product team. Paxos allows a group of unreliable network nodes to achieve agreement.

- **PacificA**

  PacificA is a replication framework for log-based database systems that was designed by the team behind MoDist.

Evaluating MoDist on the above systems revealed a total of 35 bugs that have not been reported before. More importantly, 10 of those (about 29%) were protocol-level bugs which result from unexpected network communication interleaving and node outages. The discovery

of the protocol-level bugs is especially important as they reveal flaws in the communication protocol which are almost impossible to be found by testing a single node in isolation.

MoDist made several interesting findings. First and foremost, all systems were found to exhibit protocol-level bugs that were the result of implementation details for unspecified parts of the specifications that are left to the implementors – even though the distributed protocols are theoretically sound, their implementations were not. Furthermore, in order to be able to reliably reproduce any errors found, non-determinism has to be eliminated. Domain knowledge plays an important role in guiding the execution into corner cases and as a result provides better code coverage.

2.6 Model Checking Without a Network

In this section, we provide a overview of an alternative approach to model checking networking systems, described by Guerraoui, et al[7].

2.6.1 Overview

The main idea behind the paper is to follow a local approach where the network is ignored – only the local nodes' states are explored, separately. The reason for choosing this alternative is to avoid the rapid explosion of the global system state. Notably, the set of valid global system states is just a subset of all combinations of the states of local nodes. By exploring the behaviour of each node on its own, it is possible to reach states that might not be a valid at the global level – this is checked for after any issues have found in order to avoid false positives. While the approach does not eliminate the exponential state explosion problem, it postpones it until deeper levels.

As already noted, all combinations of local node states do not result in valid global states due to ignoring of the network element. The approach taken results in **completeness** – any violation of a global state invariant that can be detected by a global approach can be detected by the local approach presented in the paper. On the other hand, it is **unsound** meaning that it detects invariant violations on invalid global states. The issue is addressed by verifying that any violations that are found are violations on valid global states. The importance of this method is that the check needs to be performed only after any violations have been found – if that number is low, as it turned out for a particular Paxos test described in the paper, it results significant speed-ups.

2.6.2 Evaluation

The performance of the system was evaluated against a classic global approach when trying to find bugs in Paxos and its variant 1Paxos. The tests were performed in a network setup with three nodes where one proposes a value and the others react to the proposition as defined by the Paxos protocol. During the tests, 3 systems were compared – a classical system that explores global states, a general system which implements the local approach and a specially optimised local approach system tuned for verification of Paxos.

Speedup The speed of the approach was measured when only one value was proposed. The state space has a depth of 22 which is relatively small. Using the classic global system, exploring the state space takes 1514 s while the optimised local system takes only 189 ms (8,000 times faster) and the general local system takes 5.16 s (300 times faster compared to the global approach). The number of transitions that were performed by the global and the general local systems, were respectively 157,332 and 1,186 (132 times less). The reason for the large difference

is because the global method redundantly executed transitions multiple times which the local approach only explored once.

Scalability Limits In order to test the scalability of the local approach, the depth of the state space was increased to 41 events where two nodes each propose two values. During the tests, the global system and the general local system both could not finish the state exploration after several hours of running. The reason for the slowdown in the local approach is due to the expensive soundness verification when evaluating whether any invariant violations are on valid global states – the number of different event sequences that must be considered when verifying increases exponentially with the search depth.

2.6.3 Conclusion

The paper[7] presented a novel approach to model checking distributed software by removing the network and only considering the states of each local node separately. While the approach is complete, it is not sound. The new approach provides significant speedups for moderate exploration depths although it does not eliminate the exponential state space increase.

2.7 Summary

In this section, we provided an overview of how symbolic execution works. We covered LLVM and KLEE, a symbolic execution engine built on top of LLVM. Afterwards, we reviewed 3 related pieces of work – KleeNet, designed to execute programs for sensor networks, and another 2 model-checking systems – one of which follows a global approach while the other follows a novel local approach.

3 Architecture, Networking & Filesystem Design

In this section, we describe the architecture of our system from a high-level point of view because it allows us to temporarily ignore implementation details. Fundamentally, our goal is to design a tool that enables the automatic testing of distributed software and in addition, provide facilities for asserting the correctness of the software at higher abstraction levels.

As we are building on top of KLEE, the symbolic execution engine engine described in Background (section 2), our changes amount to a set of extensions that enhance its modelling and execution capabilities in a backwards compatible way.

First and foremost, it is important to highlight the two facets of KLEE and how they enable the symbolic execution of programs. The tool can be split into two logical parts – **system** and **runtime**. The **system** represents the interpreter which manages the symbolic execution – it has global knowledge about all virtual processes, handles their memory address spaces and acts as a virtual process scheduler. It is built on top of LLVM and it is written in C++.

On the other hand, the processes that being simulated are compiled down to LLVM bitcode which then the interpreter runs symbolically. In order to bridge the gap between the interpreter, a **runtime** is used. The **runtime** represents a set of C functions that are compiled to LLVM bitcode and linked with the simulated bitcode before symbolic execution begins. The two primary functions of the runtime are to override the behaviour of system calls and to provide certain necessary primitives to the user programs being tested, such as marking variables as symbolic.

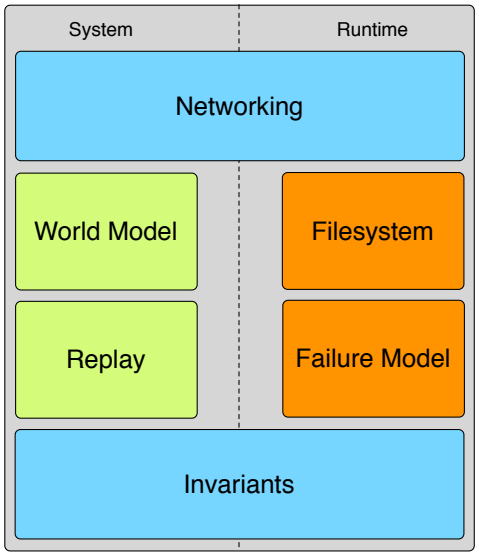


Figure 2: Showing the high-level view of the extensions provided by our system. Note how Networking and Invariants require changes on both "sides" – **system** and **runtime**.

In the following subsections, we describe the components shown in Figure 2 in greater detail. The **Networking** extension provides the ability for processes to communicate while the **Invariants** component allows the verification of user-defined properties over processes running in a network. In order to be able to execute most non-trivial programs, we provide a rudimentary **Filesystem**. By providing a **Failure Model**, we can inject system call failures and steer the software under test into rarely explored code-paths. Reproduction of any issues that are found is made possible by the **Replay** framework. Finally, changes to the internal execution model are necessary, as shown by **World Model**, to provide the capability to represent multiple processes

connected via a network.

3.1 Requirements

Before we delve into the particulars of our system, it is essential to take a step back and analyse the requirements for tools that aim to automatically test distributed software.

Wide Software Set Tools for checking correctness should be able to test the widest possible variety of software. This requirements puts multiple restrictions with regards to the possible approaches that can be taken. Choosing a method that only enables the testing of a very narrow range of software would severely limit its utility.

Symbolic Network Data The power of symbolic execution lies in the fact that programs are tested by operating on symbolic values which consequently results in exploration of multiple code paths. Enabling the ability to treat network data as symbolic would facilitate a deeper examination of the communicative behaviour of distributed software and the effects network participants can have on the global interaction.

Network Topology As the primary purpose of a correctness-verification tools are to be used on real-world systems, any tool that attempts to be used on networked software must have the capabilities to completely simulate any required network topology. The main reason for this requirement is that different network topologies can give rise to varying behaviour. In addition, specific applications might depend on specific network setups.

Performance Scaling symbolic execution to cope with large pieces of software is a complex problem on its own. Extending it to distributed software only makes it a much harder undertaking and thus the performance of the system should be of paramount importance.

Reproducibility Finding any issues in software is only the starting point. In order for problems to be resolved, the ability to deterministically reproduce any faulty behaviour is an absolute must. Providing a reproducibility framework is a very important requirement and essentially a must-have when it comes to providing real-world utility.

Higher-Level Correctness Issues can be broadly classified into two categories: low-level programming errors, such as segmentation faults, and logical errors. Low-level errors are a lot easier to find and correct as they are clearly manifested in most cases. On the other hand, logical errors are much harder to discover because their effect might be "hidden" due to the enormous complexities of large software. Providing tools, such the ability to express invariants over a set of network nodes, is an example of one approach that can aid in discovering such issues.

Low-Probability Events Networks do not provide many reliability guarantees, albeit in practice, their operation is highly reliable. In turn, this causes the code paths which handle network and low-level failures to be relatively untested. Facilities that automatically inject such failures can be beneficial in ensuring the robustness of software.

Throughout the development this project, we guided all of our design choices in ways that would maximise the satisfaction of the requirements outlined above.

3.2 Approach

The most important aspect of our system is to allow the symbolic execution of the distributed software. In particular, we wanted to provide that ability at the lowest level in the software stack, which meant at the network sockets layer. The main reason for this decision was that higher level APIs that engineers and developers are using are all based on the primitives we chose. As a consequence, the array of software eligible for testing is greatly increased.

In essence, our system is an extension of KLEE that models multiple processes connected via a network. Some of the reasons for choosing to extend KLEE include:

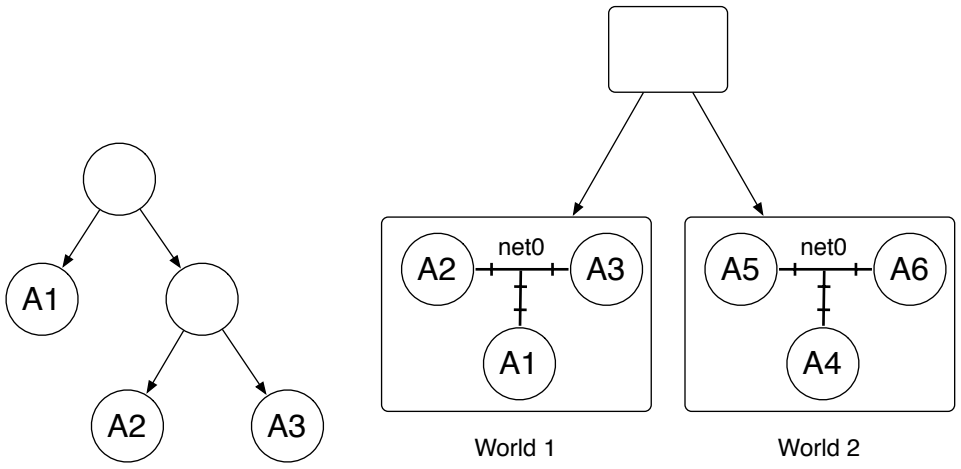
Solid Base KLEE itself is a significant step towards making symbolic execution practical for real world applications. It is well-tested and has been used around the world for several years. This is gives us an excellent starting point which to built upon instead of reinventing the basics.

It also allows us to focus on what is the most important aspect of this project – how to make distributed software testable using symbolic execution.

Open Source KLEE is open source which makes it possible to extend and modify in any way that we deem necessary. Proprietary systems would have had a limiting effect on how far we can take things as they would have only provided a fixed set of extension hooks.

3.2.1 World Model

The way KLEE sees the world is as a single process which has branched at different points due to bitcode branch instructions whose conditions can be both true and false (due to operating on symbolic values). We have extended KLEE such that its model can be seen as the simulation of a single process running simultaneously on different network nodes which are connected in some way. The rest of this section will cover the core features of the system that were necessary in order to provide the basic ability to symbolically execute networked programs.



(a) Single Process Model (b) Distributed Network Model

Figure 3: Showing the differences between the single-process model and distributed network model. 3a shows the same process which has branched due to symbolic branch conditions. Each process has no awareness of any of the others. In 6a, there are two alternate worlds, each containing processes which are connected by a virtual network and can communicate with each other.

3.3 Architecture

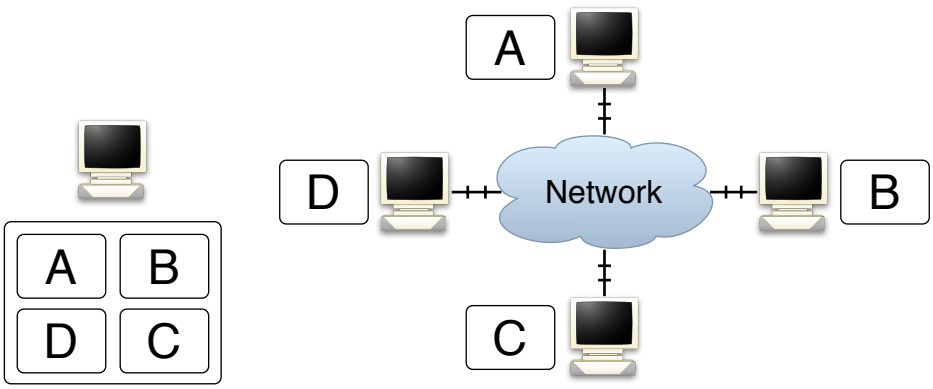
In this subsection, we aim to provide an overview of our basic assumptions and design choices in our system together with their rationale. It is important to note that some design choices are consequences of the practical nature of this project, as we are trying to both explore the limits and utility of symbolic execution of distributed software while trying attain adequate performance in real-world applications. We attempt to answer questions that bound the modelling power of the system, such as the set of the network topologies that can be explored, the semantics of simulating multiple worlds, the interaction with the operating system and several others.

3.3.1 Single Process

Simulating the execution of distributed software naturally raises the important question whether the simulation itself should be distributed. The two basic approaches that can be taken can be summarised as:

1. Re-create the networking setup and execute each process symbolic on a single node (running multiple instances of KLEE). Support would have to be added in order to connect up all the different instances of KLEE that are running on the different machines.

2. Model the execution of the whole system (i.e., all possible worlds) within one instance of KLEE.



(a) Single Instance (b) Multiple Instances

Figure 4: Showing the differences between the single-instance and multiple instances architectural choices. 4a shows a single instance of KLEE running on a computer which holds all the information. In 4b, the global state is distributed across 4 instances of KLEE connected via a network. Each instance contains only part of the state.

At first glance, option one might seem a good approach. It possesses one very desirable property – by design, it is concurrent and automatically utilises the resources of all the machines that it is being run on. Unfortunately, this design has some severe shortcomings which made us choose the second alternative. The more notable ones include:

Setup Complexity Having to set up complex network topologies adds a significant burden to the potential utility of the system. For example, if we want to explore the behaviour under

a network configuration of 50 nodes, recreating that setup, distributing the LLVM bitcode, configuration files and any other necessary bits suddenly becomes a significant entry barrier and considerably lowers the usability of the tool.

Network Topology Scalability Even more importantly, this approach suffers from serious scaling difficulties when the network topologies become increasingly large. In the alternative approach, the network topology has virtually zero cost – the costs are proportional to the number of nodes being simulated.

State Access Now that the state of each process in the network is resident in a different process address space, a single instance of KLEE does not have access to the memory contents of the all the worlds. This ability is crucial for providing higher level correctness verification tools, like invariants written over all the processes being simulated in a particular world.

Synchronisation Most importantly of all becomes the issue of actual synchronisation of the simulation. Fundamentally, all nodes needs access to the global branching history and current state of the system in order to correctly branch and preserve the semantics of isolation between the different worlds. The core problem lies in the fact that this state is distributed across the address spaces of multiple KLEE instances. Branching would require a global lock across all participating processes. This incurs significant complexity and verifying the correctness of any such architecture would not be a simple task.

Due to the drawbacks outlined above, we choose to go with a single-process design.

3.3.2 Symbolic Network Topology

We briefly touched on the subject of network topologies in the above section when it comes to scalability. Another important facet of distributed software is that in many cases it depends on the particular network topology that it is interacting with. For example, routing protocols exclusively deal with issues of topologies and how to most efficiently route packets.

It is thus desirable to automatically explore the behaviour of software when run on different topologies. One way to think about it would be to have a "symbolic" node connected to the network which represents multiple network nodes arranged in different topologies beyond that point. Consequently, if we run tests on such a meta-topology (which represents a set of topologies), we can make much stronger statements about the correctness of the software with respect to the network configurations it can handle.

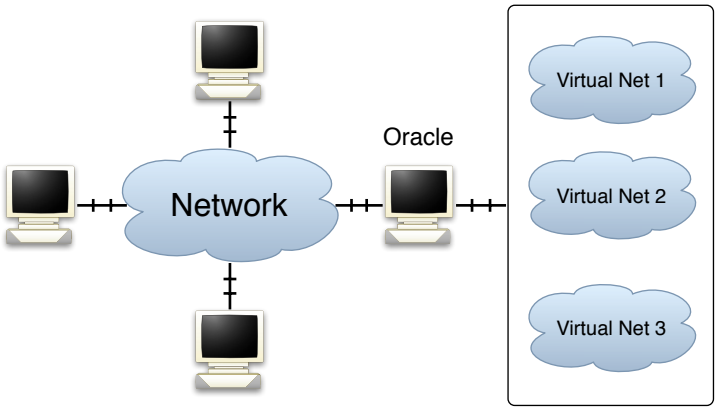


Figure 5: Showing a Oracle node which represents a set of network topologies.

While providing an incredible opportunity to prove correctness over a multitude of network setups, there are very serious issues in realising that ability. Firstly, it should be noted that in order for the feature to be useful, the nodes that are being simulated need to be able to communicate with the symbolic part of the network. The question arises of how would the node, the oracle node, which sits on the border of the symbolic and non-symbolic networks respond to any network requests without actually simulating the different possible topologies. If on one hand, the oracle just acted as a black hole by not replying to any network requests, the scenario that we are testing degenerates into a network without the symbolic part. On the other hand, the oracle node has inherent way to know how a particular network of processes would have replied to a sequence of packets.

As a consequence, we have deemed the feature of a symbolic network topology as non-practical at present and our system would only explore the behaviour of the software under test under a specific topology in a single run.

3.3.3 Copy on Send Branching

Semantically, each world can be viewed as an entity that contains multiple processes, each of which executes on a separate network node. It is crucial to preserve the semantics of world separation when it comes to network node communication – that is, data is received by a process in world if and only if that data would have been received if we run the software natively.

This becomes important when we consider what happens when a single process in a world branches due to operations on symbolic values. Assume that we have three processes - A, B and C and that C encounters a branch which can be both true and false. This means that there are now two possible parallel worlds:

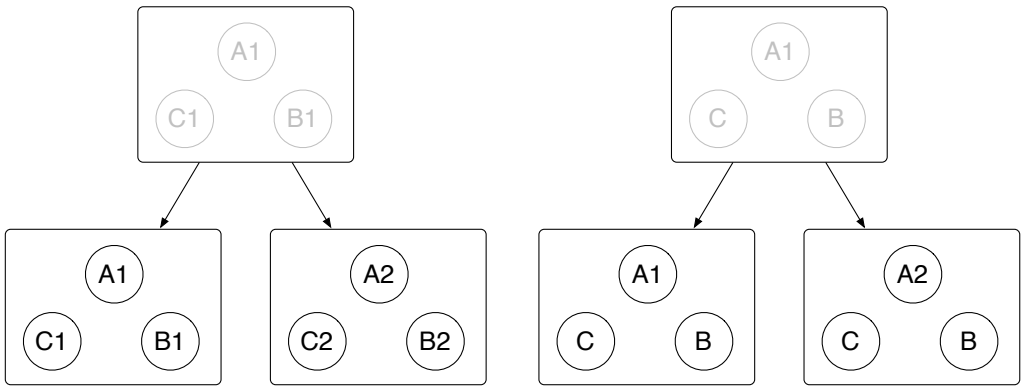
1. The world where the branch condition of the expression was true.

2. The world where the branch condition of the expression was false.

So now we have 6 processes in 2 worlds – one triplet in each. For example, we have two copies of A – one in the world where its neighbouring node branched with a particular condition being false and another being true. It is important to realise the need for having two semantic copies of A – if now the C process which took the true branch decided to send a UDP packet to A, that UDP packet should only be received in the world containing A in the same logical world. If we instead did not treat the branching as branching all the states contained in a world, process A could receive a network packet from C that originated by taking the true branch, even though C took the false branch – which clearly breaks the semantics of world separation.

The naive way to preserve the semantics would be to just copy all processes in a world whenever one of the processes branches. While technically correct, this would be not be a smart choice. Crucially, it can be noticed that we can defer the copying only when there is communication between the nodes that will violate the separation, which can be easily checked for.

In order to avoid a premature optimisation, a dynamic instrumentation tool[10] was used to compare the number of branch instructions (which are used to branch processes) versus the number of system calls (which provides an upper bound on instructions that can trigger a branch to preserve isolation) when running a routing daemon[6]. The average over 9 runs revealed that the number of branch instructions is about 325 as many as system calls. Consequently, it makes sense to implement the optimisation to only branch on demand as opposed to conservatively.



(a) Conservative Distributed Branching (b) Optimised Distributed Branching

Figure 6: Showing the differences between conservative and optimised branching. 6a shows how all states get branched whenever one of them (in this case A1) needs to branch due to a branch instruction on a symbolic value. Note that in 6a, states B1 and B2, C1 and C2 would be identical and would not have any additional constraints added to them. On the other hand, in 6b only state A1 is branched and B, C are shared between the distributed systems – they will be branched on demand to preserve the isolation property whenever one of the other network nodes tries to send them data over the network.

3.3.4 Boot-Strapping

As a consequence of our decision to use a single instance of KLEE to symbolically simulate all processes in the network, the binary that gets simulated must contain the code for all binaries across the network that will be simulated. That does not present a problem because most cases fall under the following two scenarios:

1. **Identical Binary**

   The test usually consists of running the same program on all the machines and testing how it interacts with different copies of itself. For example, routing daemons fall into that category.

2. **Client-Server**

   In the majority of other cases, the relationship is a client-server one where the code being tested resides in the server while the client is just used to initiate the test sequence. For example, testing a Web server falls into that category where the client can consist of 40 lines of C code which just creates a `GET` request and sends it to the server.

In extreme cases where the binary size becomes a limiting factor in initialising the simulation, the system can be modified to allow the loading of separate binaries. As it is highly unlikely that we will have to deal with such cases, we chose the simple approach of linking up the final binary which would contain the combined code of several logical programs.

Another aspect of boot-strapping the simulation is the order of initialisation. For example, if we were simulating a client making an HTTP request to a server, we would want to the server to be already listening for connections by the time the client makes an attempt to connect. In order to enforce such relations, our systems provides the ability to specify the order of initialisation of processes.

3.3.5 OS State & Interaction

Every program interacts with the operating system to varying degrees – networked applications even more so. By default, KLEE routes any functions that have not been specifically modelled to the OS. This gives rise to two problems:

- **Symbolic Interaction**

  Given that values can now be symbolic, it is important to decide what should happen when an OS function (a so called "external") gets called with a symbolic value. There are three ways to handle it:

  1. Disallow the execution of external functions with symbolic values. Taking this approach can defeat the purpose of testing the software if a large proportion of function calls are externals.
  2. Concretise the value to one particular instance that satisfies the constraints and pass it to the OS. This option can be even more dangerous than the previous one as it can give rise to false positives. If the program reads back the values that it passed to the OS, it will not necessarily retrieve a equal values.
  3. Explicitly model such functions in a way that preserves their semantics.

  We have opted for a combination of the first and last option – we disallow the execution of external functions that are deemed unsafe and explicitly model all essential functions that are necessary for the operation of distributed programs.

- **Isolation**

  Calling externals which operate on shared data kept at the OS level can lead to unexpected problems. Our system is simulating the execution multiple processes on different network nodes and we need to ensure that any interaction with externals cannot affect any of the other processes.

  For example, it is unsafe to redirect the usage of `chdir()`[2] to the OS – not only will it change the current working directory for all worlds that are being simulated, in addition it will change the working directory for KLEE itself.

  In order to properly handle such cases, we need to model and provide isolation by storing such state ourselves on a per simulated-process basis.

3.3.6 Scheduling

In its original form, KLEE simulates the execution of a single process but with the twist that it explores multiple possible execution paths. This means that at any given point in time, KLEE has to decide which particular execution path to follow. The scheduler (or "searcher" in KLEE parlance) is responsible for that role.

By modelling the simulation of multiple processes running on different network nodes and by providing the ability to communicate using the POSIX sockets API, our system introduces the need for changes to the scheduler in order to implement the semantics of the relevant functions.

[2] Changes the working directory of the calling process.

Blocking By default, reading from the network is a blocking operation. This implies that if a process tries to `read()` from a socket and there is no data, this particular process becomes blocked and ineligible for selection to be run. In addition to being blocked, a process can also set a timeout after which it becomes unblocked – for example, the `select()` function accepts a timeout and returns `0` if the timeout expires with no activity on a set of sockets.

This creates an additional level of complexity in the scheduler because it does not have the freedom to choose any arbitrary process to run next (while that was indeed the case in KLEE in its original form).

Interleaving Of even greater importance is the behaviour of the scheduler when it operates at the scale of a world. Assume that the scheduler has selected one of the worlds and now needs to choose which process within that world to run. The core issue is that there is no "correct" choice – the scheduler is essentially taking the role of deciding how the processes running on multiple network nodes are to be interleaved.

The importance of this issue is that the behaviour of networked software can depend on the sequence of packets it receives. As a simple example, assume we have a server that arbitrates the access to a shared resource on a first come, first served basis. Furthermore, two clients want to request the resource and are just about to send the network packet requesting the resource. The outcome of who gets the token is non-deterministic if we run the programs natively – it will at least depend on factors like network congestion and delay. Our system will non-deterministically pick one of the clients and we will not explore all possible interleaving as that will create a significant increase in possible states.

Our Design From a very high level, we can view the state of all worlds as a tree where the leaves contain the worlds and the structure of the tree reflects the branching history. In addition, each node in the tree knows whether it is blocked or unblocked – for the leaves, it would be equal to whether there is at least one state that is runnable. For an intermediate node, the node is unblocked if and only if either the left or the right node is unblocked.

This design allows us to choose a state by starting at the top of tree and traversing the tree down until we reach a leaf (an invariant maintained throughout the execution is that a node is a leaf if and only if it contains at least one process) – there is no need to backtrack by getting stuck if we reach a world which is blocked. In addition, it allows us to instantly determine whether there are any unblocked states at all across all worlds by just expecting the root of the tree.

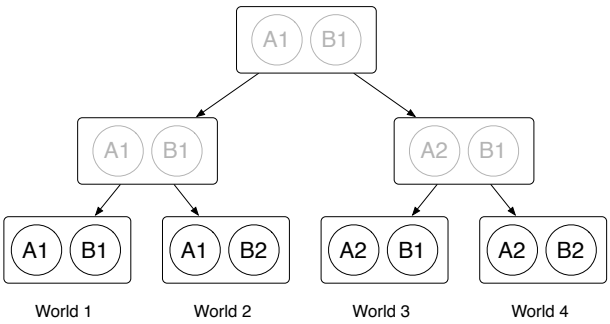


Figure 7: Showing a tree composed of 4 worlds. Initially, there were two worlds A1 and B1. Firstly, A1 branched and later B1. Note that we only have 2 processes even though there are 4 worlds – this is because no communication has taken place and we branch on demand as explained before.

When it comes down to actually choosing which state to run, we follow a two stage process:

1. **World Choice**

   In the first stage, we choose a world. We do this by following a random path starting at the root of the world tree. At each node where both paths can be followed, the set of worlds in each subtree has equal probability of being selected. We inherit the same properties as in the original implementation of KLEE[2], namely:

   - It favours worlds that are higher in tree – i.e., the ones that have less constraints and more freedom to explore different code paths.
   - It avoids starvation in cases where a tight loop in some world is creating an excessive amount of processes by branching.

2. **Process Choice**

   Once a world has been chosen, we need to choose a specific process within that world. We do that in a round-robin fashion making sure we skip any blocked processes.

3.3.7 Deadlock Detection

Another feature of our system that is closely related to scheduling is the ability to detect a deadlocked world. When a process becomes blocked waiting for network events, it can either specify a timeout or it can wait infinitely. We use the term "stalled" to indicate a process that has blocked and does not have a timeout.

Once a process becomes stalled, it can only become unblocked if and only if it receives a network packet from another process in the same world. Deadlock detection uses this fact to detect the case when all processes in a particular world have "stalled" – as all of them are waiting for network packets from each other but none of them can send any before being unblocked, the whole world is deadlocked and can be terminated to save resources. Deadlocks can arise in normal testing and do not always indicate faults in the implementation. For example, if we were to test how a server handles a single HTTP `GET` request, after the client process exits, the system will be deadlocked as the server usually runs in an infinite loop waiting for more connections.

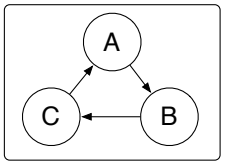


Figure 8: A deadlocked world – A is waiting for network events from B, B is waiting for events from C and C is waiting for events from A.

3.3.8 Closed World

Another design choice that we have made was to model our system as a closed world. We do not support and model any interaction with entities that are not being simulated. Having the ability to interact with any parts outside of the simulated world raises multiple issues, include but not limited to:

Symbolic Data In order to preserve the semantics of symbolic execution, the external entities would need to have the ability to accept symbolic data. This precludes the usage of systems without any modifications and it can be argued that extending KLEE itself might be a better option in such scenarios.

Shared Entities Unless the state of any shared external entities is read-only, additional support would need to be added to correctly handle any communication such that it preserves the semantics of world separation.

As we did not encounter any cases where interaction with the outside "world" was absolutely necessary during the the preliminary stages of this project, we deemed the closed world assumption to be a practical one that should not affect the utility of the system.

3.4 Networking

The networking subsystem lies at the heart of our extensions. Its aim is to provide an efficient internal mechanism (via a set of functions) that can be used to implement the POSIX sockets API. In terms of extending the current system, there are two types of additions that are required to enable inter-process communication of simulated processes:

1. **Internal**

   Processes have no access or awareness of any others. We need to expose an API which provides the ability to exchange information with the other network nodes. KLEE has a mechanism to allow "special function" which are implemented within KLEE itself. For example, that is how primitives, such as marking data as symbolic, are implemented. Only KLEE has access to the address space of all processes, so we have added primitives to allow exchange of data within worlds.

2. **Runtime**

   The runtime implementation gets compiled to LLVM bitcode and is linked during initialisation with the rest of the program. We have to implement the POSIX sockets APIs in the runtime by using the "special functions" that we have exposed.

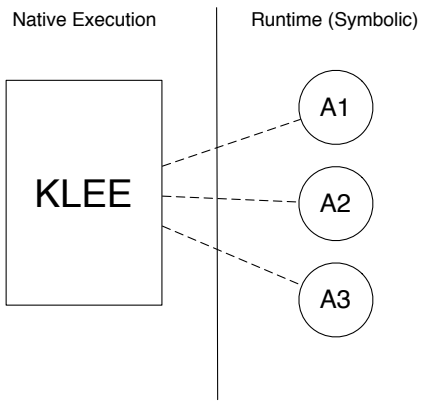


Figure 9: Showing the distinction between the address spaces of the simulated processes and the simulator. Note that our system has full access to the internal address spaces of A1, A2 and A3 while those virtual processes are do not have the ability to access any other address spaces.

There is a fine line between choosing which parts should go into the runtime and which ones should go into the system itself. It usually depends on the functionality that is being implemented. The benefits of each approach include:

Internal Only the system itself has a complete global picture of all the worlds and processes they contain. If there is a need to access that information, the only option would be to implement it internally. Another benefit of internal implementations is that they run natively which provides significant performance advantages (in general, code runs between 10-100 times slower when interpreted).

Runtime Implementing parts in the runtime provides benefits when it comes to using symbolic data. Symbolic values are essentially first-class citizens, as the code does not differentiate whether something is symbolic or not – it is just plain C code. In addition, it makes it easy to implement any behaviours that are related to branching or deal with symbolic data. For example, if we want to explore two code-paths at the same time, all we have to do is mark a variable as symbolic and write an `if` statement whose condition tests whether the variable is equal to some particular value.

The networking subsystem can be subdivided into two separate parts – one that deals with transferring data between processing and the other is concerned with defining the topology of the world.

3.4.1 Event System

Our design includes an extensible way to transfer "events" between states. An "event" is the most important abstraction and underlies the basis for all inter-process communication. An "event" has several properties:

- **Type**

  Each event has a type which allows any receiver to determine whats actions to take. There are currently 4 types in use: `data`, `request`, `reply` and `close` used to support data transfer and TCP connection establishment & termination.

- **Identifier**

  Each event features a unique identifier. The reason for including identifiers to be able to detect and correctly handle duplicate events.

- **Source & Destination**

  Each event has a source and destination pairs of IPv4 address and port.

- **Request & Connection Identifiers**

  Those two identifiers are optional and can be used to identify long-lasting (in terms of event sequences) "conversations" between processes.

- **Data**

  Optionally, each event can be associated with opaque data – a sequence of bytes (which can be symbolic). It is very important to note when any symbolic data is transferred, we transfer all the constraints that it is associated with.

Coupled with the notion of an event, our system defines a set of internal functions available to processes to interact with other processes via events. Those functions can be divided into two categories:

1. **"Inbox" Inspection**

   Each process has an "inbox" that collects and stores pending events. There are two **very** important functions that allow the inspection of the inbox:

   (a) **Pending Events**

   A process can check whether there are any pending events that match certain criteria. This functionality is essential as it allows the implementation of non-blocking POSIX socket APIs.

   (b) **Event Blocking**

   Even more importantly, there is a function that allows the calling process to indicate that it should become blocked unless certain events are contained in its "inbox", optionally specifying a timeout. This is the primitive that is used to implement all the blocking behaviour in our system. For example, the primitive can be used to implement a blocking `recv()`, `select()` and even `sleep()`.

2. **Event Transfer**

   In addition to the "inbox" inspection functions, a set of functions that enable the retrieval, removal and sending of events is provided. Events which have been sent reside in the address space of our system until they are retrieved – the "inbox" represents events that are buffered by the OS and only get moved into the address space of processes by using the retrieval functions.

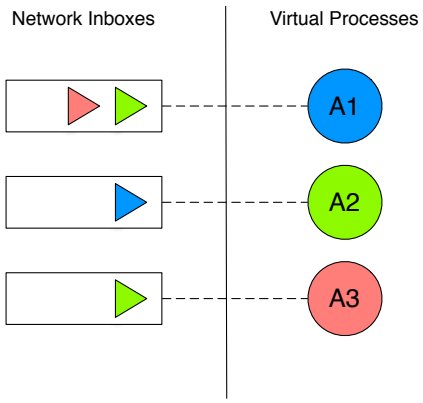


Figure 10: Showing the network inboxes of each virtual process. Note that the inboxes reside in the address space of the system. In this figure, A2 has sent two events – one to A1 and one to A3, A1 has sent an event to A2 and A3 sent an event to A1.

One of the reasons we have chosen an event-based systems is due to the flexibility and extensibility it provides. The event based system is general enough such that it is trivial to provide implementations of other interprocess primitives. One such example would be the implementation of process signals – a signal can just be implemented as a new type of event which has its IP source and destination zeroed out, the request identifier can be used to carry the process ID of the recipient and the connection identifier can be used to store the signal.

3.4.2 Network Topology

Providing the ability to send and receive events leaves one important detailed completely unspecified – the routing of the events from the source to the destination.

Our system exposes another set of "special" functions that can be used to configure the network topology. The advantage of exposing the functionality to the process under simulation as a set of functions means that the network topology can be configured during initialisation and actually be dependent on arguments passed to the program. Our model is identical to a configuring a real system – we provide five primitives that are used to configure all aspects:

1. **Networks**

   Ability to add named networks which this can be viewed as defining a physical LAN which nodes can be attached to.

2. **Routers**

   Ability to create routers which are virtual entities that allow traffic to flow between networks.

3. **Interfaces**

   Ability to add network interfaces to routers or to the network node simulating the current calling process.

4. **Addresses**

   Ability to assign IPv4 addresses (together with a subnet mask) to network interfaces.

5. **Routes**

   Ability to specify the routes which the virtualised OS should use to route packets. Routes can be specified for both virtualised routers and for the network node that is running the calling process.

Our system provides virtualised routers whose only role is to forward packets to the destination processes that are being simulated – the routers themselves are not simulated in the same way as the processes under testing.

The advantage of following the networking model as used by real system is that it allows a transparent mapping between the models. This property will later be exploited by other parts of our system.

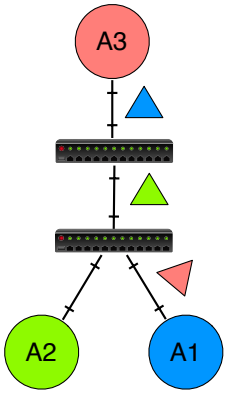


Figure 11: Showing a network composed of two routers and three processes. There are three events being transmitted.

3.5 Filesystem

Most distributed software relies on the filesystem for various reasons, most commonly to read configuration files or to store results from its computation. Thus it is essential that a rudimentary filesystem is provided that respects the semantics of the system calls that interact with it. The design of any such filesystem should favour efficiency and speed for the common cases over completeness of support of POSIX functions.

Our filesystem can be seen as dual system which supports fast access to Operating System-backed files and also virtual ("symbolic") files. It has the ability to correctly handle the semantics of all commonly used filesystem functions – for example, creation, reading and writing to files are all supported. When we refer to files, we also implicitly include directories as both are handled in a uniform way.

When a request is made to `open` a file, what actions are taken depend on several factors but primarily it depends on whether the filename is symbolic or concrete. If the filename is symbolic, the request can be considered to a simultaneous request for multiple files. There are two ways in which we can handle symbolic path names:

1. **State Exploration**

   We can try to explore all possible combinations that the pathname can take, branch for each one and re-route them to the operating system. While a simple solution this would have terrible performance and create an explosion of states, most of which would end being non-existent.

2. **Symbolic Filesystem Subset**

   Another approach would be to have a very small set of "symbolic" (in memory) files that is used to represent the contents of the whole filesystem when the filename is symbolic – in a way, it provides an symbolic-reality filesystem. The advantages of this approach is that it is efficient in terms of branching as the symbolic pathname will only be compared against a very small number of files. Even more importantly, the "symbolic" filesystem can be chosen such that it covers all "interesting" cases, such as empty files, regular files and directories, as well as a non-existent files.

The following flow chart provides a high level overview of the logic used when opening files.

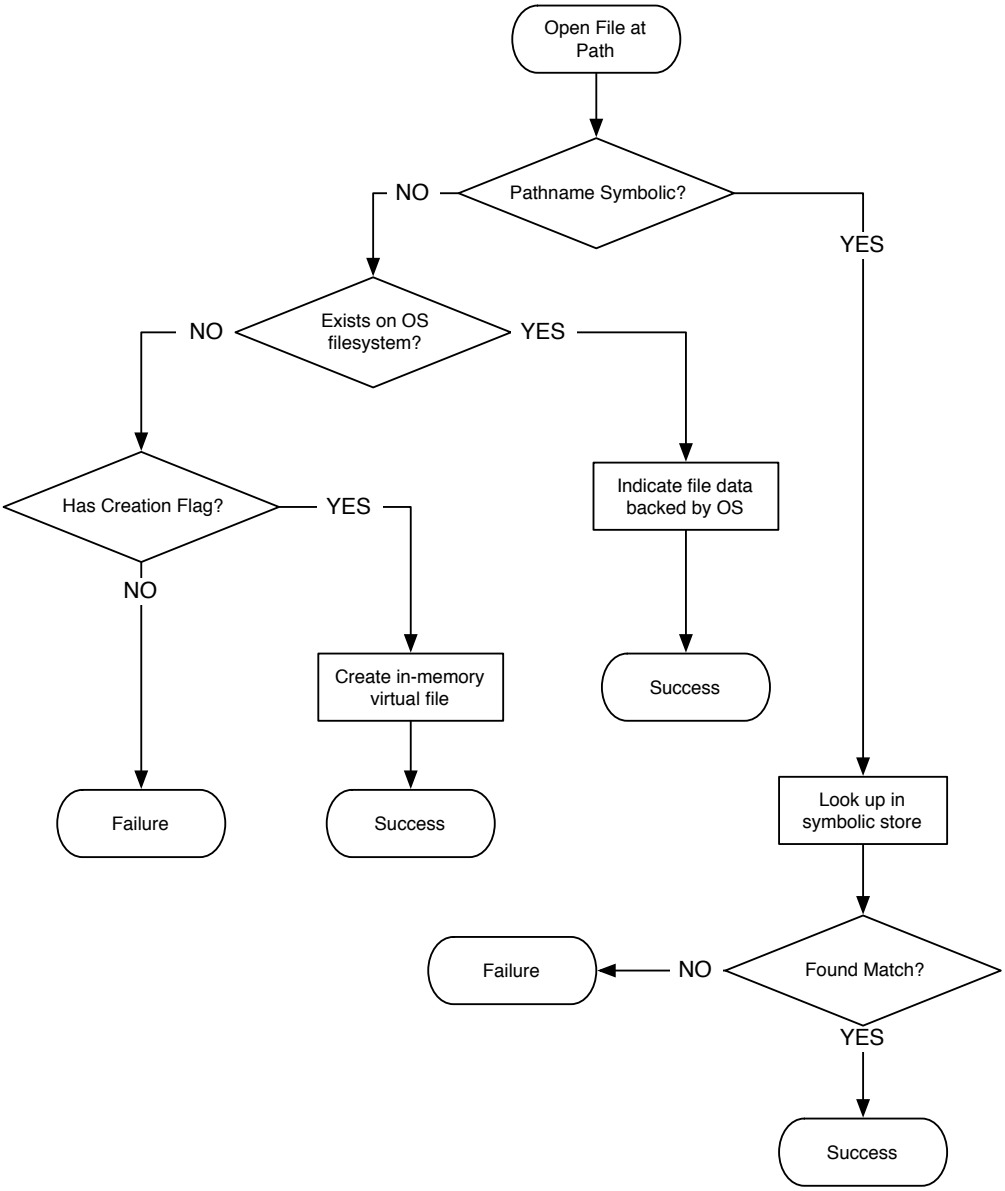


Figure 12: Showing the high-level logic when opening a file.

3.5.1 OS-backed Files

One of the most common cases that would arise when testing a programs would be that of reading configuration files from the filesystem, opening log files and outputting data to the standard streams (`stdin` and `stdout`). By design, our files can be backed by the OS which provides enormous savings versus having to store the data in memory.

Writing to OS-backed Files The issue arrises about how our system should handle writes to files that are backed by the real filesystem. We cannot just relay any writes because that will inadvertently affect the other processes that are being simulated. There are several approaches that can be taken.

In-Memory When a request is made to write to a file, we could read the whole file in memory and then perform the writes in memory – this will not disrupt any other processes as we have turned the shared file into a private memory region. The biggest disadvantage of this approach is the enormous memory penalty for large files.

Versioned A more efficient approach would be to have a versioned file system whereby `read()`s and `write()`s go via a special function API which is provided by our system that does the versioning efficiently on a per process bases. This solution easily lends itself to optimisations such as copy-on-write.

Restrict A further approach would to be put a restriction such that OS-backed files cannot be written to.

As we guided the development of our system by the needs that arose in order to be able to run production software, we choose the last option – currently, writing to files that are backed by the OS is unsupported.

3.5.2 Extra Features

Our filesystem also handles some special cases that arise in practice often enough to warrant special handling.

Standard Streams There is special support for `stdout`, `stderr` and `stdin` such that all operations correctly work on them.

/dev/null One common thing that happens in web servers is the redirection of various streams to `/dev/null`. Support for `/dev/null` is present with all the correct semantics.

Finally, we also have the ability to persist any in-memory files at the end of the run to disk for further investigation of the behaviour of the software.

3.6 Summary

In this chapter, we looked at the high-level design of our system. We discussed limitations to the set of network topologies that can be tested, provided description of how we can bootstrap the simulation process using existing infrastructure and explored the responsibilities & requirements placed on the process scheduler. We provided an overview of a event-based system which is general enough to allow the implementation of the POSIX networking APIs. Finally, we looked at the requirements for our filesystem and outlined various approaches on how we can satisfy them.

4 Replay Framework

Finding issues using our system is only half the story. The first step in fixing any bugs requires the ability to reproduce the problem. In its original form, KLEE generates "test cases" when a particular execution path terminates. These files provide the information that can make the software follow the exact same code paths that it did while being simulated. It achieves that by instantiating any symbolic variables to concrete values that satisfy the constraints for that execution sequence.

There are two reasons why the existing replay system does not work when we run distributed software:

1. **Network Configuration**

   The network topology used by the simulation would rarely coincide with the network topology of the host that is running the simulation.

2. **Initialisation Sequence**

   The order of initialisation of the network nodes usually plays an important role in recreating the issue.

We resolved the two issues outlined above by re-using the existing test system while also including more data in a separate test file. We refer to the original test files as ktests while the additional tests generated by our system as dtests.

In order to reproduce an issue found by our system, we have to know two things:

1. Which processes (and consequently which ktests) were part of the world where the issue appeared.

2. What was the network configuration and initialisation order of the processes within the world.

Both of the above are recorded in the new dtest files – in essence, a dtest file composes (by referencing) multiple ktest files while additionally including information about the network configuration and initialisation order. An example dtest file is shown below.

Listing 4: dtest test case

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
  <network name="net0"/>
  <network name="net1"/>
  <router hostname="router">
    <interface identifier="1" network="net0">
      <address ip="10.0.0.2" mask="24" primary="true"/>
    </interface>
    <interface identifier="2" network="net1">
      <address ip="10.0.1.2" mask="24" primary="false"/>
    </interface>
  </router>
  <node filename="test000002.ktest" hostname="client" filesystem="" server="
      false" sequence="0">
    <interface identifier="1" network="net1">
      <address ip="10.0.1.1" mask="24" primary="true"/>
    </interface>
    <route destination="null" mask="0" gateway="10.0.1.2"/>
  </node>
```

```
19    <node filename="test000003.ktest" hostname="server" filesystem="" server="true
          " sequence="1">
20      <interface identifier="1" network="net0">
21        <address ip="10.0.0.1" mask="24" primary="true"/>
22      </interface>
23      <route destination="null" mask="0" gateway="10.0.0.2"/>
24    </node>
25  </test>
```

The dtest file contains all the information to completely reproduce any issues found. Finally, we use VNUML[11] to create a virtualised network of systems that exactly replicates the configuration specified in the dtest file. We have written a tool that generates a VNUML configuration from a dtest file.

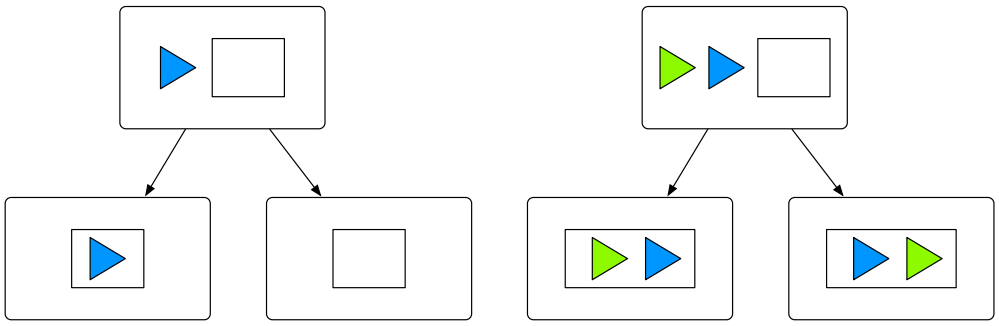
5 Failure Model

When testing distributed software, low probability events are usually very hard to simulate and thus execution paths which handle exceptional errors can end up being untested. Consequently, it is very important to non-deterministically fail system calls, like `read()`, `write()`, etc., in order to achieve higher code coverage. As demonstrated by both KleeNet[13] and MoDist[15], once a very small number of such failures has been injected, any additional ones do not lead to exposing any additional distributed behaviour.

We provide four ways to automatically inject various events, in hopes of steering the system under test into unexplored code paths.

5.1 Packet Loss & Re-Ordering

Communication over UDP does not guarantee reliability, which makes it possible to lose packets and re-order them while still preserving the semantics of communication over UDP sockets. Our system has the ability to define the maximum number of packets that were are lost along any execution path. In addition, UDP packets can be dynamically re-ordered at runtime – there are two parameters that control re-ordering behaviour: the number of times packets will be re-ordered and the re-order window size. The window size specifies how many packets to wait for and then to perform all possible re-ordering on that sequence. It should be noted that for a window size of n, there are $n!$ possible re-orderings which makes large reorder window sizes impractical.



(a) Packet Loss (b) Packet Re-ordering

Figure 13: 13a shows how our system explores both possible outcomes when a packet is sent – both being received and dropped. In 13b, we explore all possible re-ordering combinations, which in this case is 2.

5.2 Symbolic Automark

Another feature of our system is the ability to automatically mark parts of the data sent across the network as symbolic. The number of times this is performed, the amount of bytes and the offset within the data can be specified. This ability can be viewed in two ways: either as an easy way to mark data as symbolic without performing any code changes or as a mechanism to explore how the software handles arbitrary data (e.g., malicious clients).

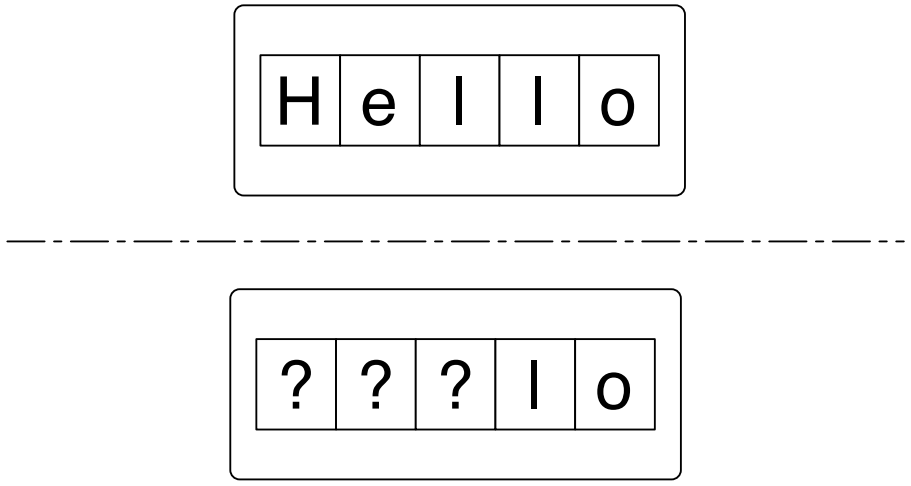


Figure 14: The code sends the bytes *Hello* while our system marks the first 3 bytes as symbolic and the recipient will transparently receive the symbolic data.

5.3 System Call Failures

Finally, we can choose to fail common system calls, such as `read()`, `write()`, `accept()`, etc. The user can specify the maximum number of failures along any single execution path.

6 Distributed Invariants

As previously seen in KleeNet[13] and in MoDist[15], distributed invariants can find bugs at the protocol level that cannot be caught otherwise. It is entirely possible that programs can achieve 100% code coverage but still have bugs which result from untested node interaction and could either be caused by faulty protocol implementation or due to the protocol being unsound in itself. Checking the distributed state across all the network nodes can reveal such situations. The specification of distributed invariants has several important aspects, outlined below.

Syntax The invariants have to be expressed in some form and there are two main choices: either a domain specific language (DSL) or the native language of the software under test. The use of a domain specific language would make it easier to express certain properties while on the other hand it would require the infrastructure for its interpretation.

Instead, if the invariants are expressed as functions in the language of the target software, they could be compiled to LLVM bitcode and then interpreted at runtime. This will provide the freedom to use the target system's internal APIs to define arbitrarily complex assertions. One necessary change would be to include functions that can used to return handles to the nodes that are being tested, so that the invariant checking code can iterate over them and execute functions in the processes' contexts.

Data Access The invariants themselves need to access state from each process in a world. If a DSL is used, there needs to a way to specify how the memory / state is "seen" from the within the language. On the other hand, if the native language is used as a syntax, things become a little bit easier as we can just re-use the semantics defined for that language.

In both cases we would still need to define the context within which the invariants are evaluated.

Evaluation Point The evaluation point of the distributed invariants has to be well defined in order to avoid false positives. For example, loop invariants only hold at the loop start point. Ideally, programs should have the ability to specify at which points the assertions should hold.

6.1 Minvariant

Based on the advantages of each approach, we choose to design our own domain-specific language. Two of the main reasons included:

1. **Expressive Power**

   DSLs are custom designed to perfectly fit their target usage. In turn, this implies that by design they will have a very high expressive power which would result in ease of usage.

2. **Extensibility**

   Designing a custom language gives us the freedom to extend it without worrying about breaking any backwards compatibility.

While running preliminary tests of our system, all the scenarios that we encountered where we could apply invariants could be seen as a two stage process:

1. **Collect** In the collect stage, opaque data (sequences of bytes) is collected from each process in a world.

2. **Operate** In the operate stage, the set of sequences of bytes are being used to return a boolean value.

One of the most important aspects of our design is how processes expose data. Firstly, we need to define three primitives:

- **Node** – a node stands for a process running on a simulated network node. It is represented by a string.
- **State** – a state is an abstraction of a particular user-defined internal state of their system. It is represented by a integer.
- **Key** – a key is used in the same way as in a dictionary / map, as an index. It is represented by string.

In our design, nodes can expose data for a particular combination of state and key. The state is usually meant to be used to distinguish between different phases a program goes through and the key is used as an index to locate the data for a particular key. A pair of state and key can be thought of as a unique way to identify a sequence of bytes within a node.

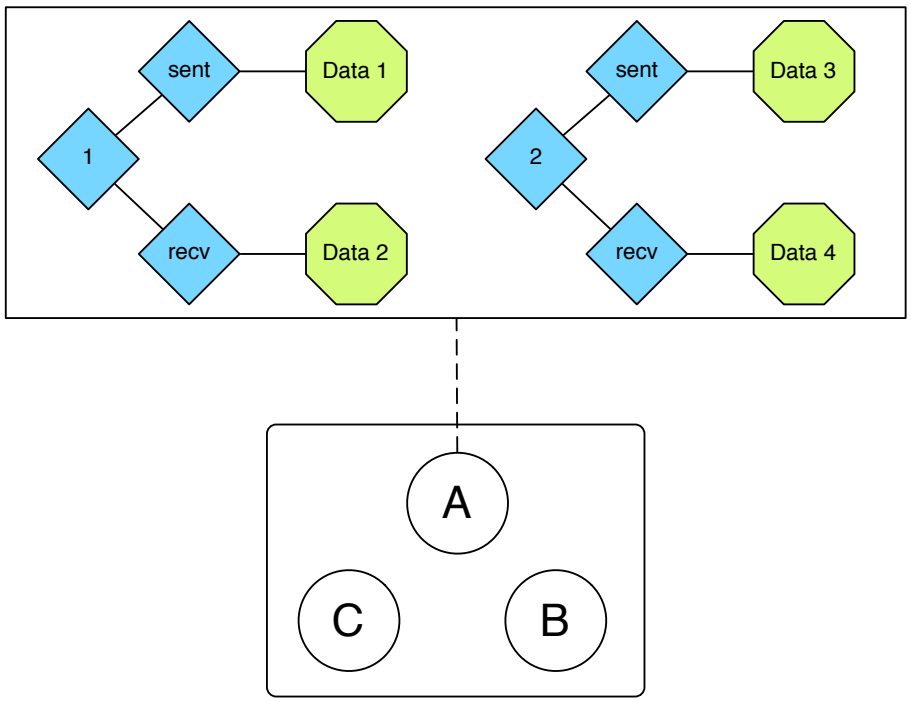


Figure 15: Showing data exposed by process A for invariant checking. For example, *Data 2* corresponds to using a state value of *1* and key *recv*. Note that the the data exists outside its address space once exposed. Invariant data is shown in green and can be reached by a "composite key" (shown in blue) which is a combination of state and key.

Before showing how the *collect* and *operate* stages are semantically defined, we present a Minvariant program that checks whether all participants in a 2-Phase Commit have reached the same decision.

Listing 5: Checking for decision consistency in 2PC.

```
1  invariant decisionConsistency(data[] d) : nodes, states, keys {
2    return d.equalElements();
3  }
4
5  string[] nodes() {
6    return sys.nodes();
7  }
8
9  int[] states(string node) {
10   return int[1] ;
11 }
12
13 string[] keys(string node, int state) {
14   return string["decision"];
15 }
```

The key components in this Minvariant program are the three functions (`nodes`, `states`, `keys`) and the invariant `decisionConsistency`. It should be noted that each invariant accepts an array of `data` (representing opaque sequences of bytes) and in its definition needs to specify 3 functions – one which returns an array of nodes, another which given a node returns an array of states and finally a function that given a node and a state, returns a set of keys.

In order to evaluate the invariant `decisionConsistency`, our system performs the following steps:

1. It evaluates the `nodes` function which returns an array of nodes. Note that `sys` is a special global object and `sys.nodes()` returns an array of all nodes in the current world.

2. For each of the nodes that were returned, it will call the `states` function, passing the current state. This will return an array of states.

3. For each of the states returned in the step above, the function `keys` will be executed collecting the results.

4. At this point, our system will have a list of triplets – node, state and key which should be used to collect the data which has been previously exposed by processes.

5. After collecting all data, the invariant `decisionConsistency` is evaluated by passing the collected sequences of bytes. In this case, it checks whether all the sequences of bytes are equal (i.e., all decisions are the same).

6. If the return value of the invariant is `false`, then it is deemed violated.

One of the important features of Minvariant is that the *collect* stage is defined as a set of functions (which can be of arbitrary complexity) while the *operate* stage is the invariant code, which again can be of arbitrary complexity. Notably, functions can be easily re-used when defining multiple invariants.

7 Implementation

In this section, we provide implementation-specific details about the various subsystems that comprise our software. Our system's architecture is shown again in figure 16.

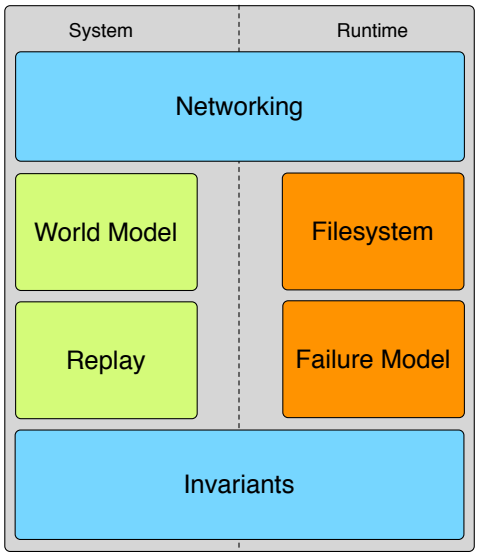


Figure 16: Showing the high-level view of the extensions provided by our system. Networking and Invariants require changes on both "sides" – **system** and **runtime**.

The section is split into two parts, reflecting the changes that were made to the **system** and the **runtime**. One very important aspect of this architecture is that the **runtime** is executed symbolically which gives it the ability to branch and use symbolic values transparently. The specifics of uninteresting or trivial parts have been left out.

System Changes These changes were modifications made to KLEE itself needed to provide the infrastructure necessary to implement several primitives to allow for communication between the simulated processes. In addition, the changes in the **system** were necessary to extend the world model to accommodate the testing of distributed software. From figure 16, **Networking** and **Invariants** span both logical components and shows that those features require special primitives in order to be implemented.

Runtime Changes The **runtime** is composed of C implementations of the POSIX filesystem and socket APIs and gets compiled to LLVM bitcode. During initialisation, the **runtime** is dynamically linked with the software under test so that any system calls get re-routed to our custom implementations. As shown by figure 16, the **Filesystem** and **Failure Model** do not require any special support while **Networking** and **Invariants** require a certain set of primitives.

7.1 Overview

First and foremost, we provide details about the modifications that were necessary to introduce a multi-world model (section 7.2.2) that allows us to represent the execution of multiple processes across a network. We also cover the bootstrap (section 7.2.3) process which re-uses pre-existing infrastructure to provide a simple solution. Afterwards, we take a look into the specifics of how

networking is implemented on top of a general event system (section 7.2.4). Crucially, in section 7.2.5, we illustrate the operation of the algorithm that preserves the semantics of world isolation when processes exchange events. In section 7.2.6, we show the additional imposed restrictions on the scheduler as a result of our modifications and how we provided a very fast solution. 7.2.7 covers the model and implementation of the invariants framework while 7.2.8 briefly describes the operation of the replay framework.

After covering the **system** changes, we move on to the **runtime** extensions. 7.3.1 provides details on the primitives provided by by the **system** side to enable implementation of the networking APIs, which itself is looked at in section 7.3.2. Section 7.3.3 explains how our filesystem works while section 7.3.4 provides a deeper look into how we inject failures.

From the onset of the project, it was crucial for all of our modifications to maintain full backwards compatibility with the existing system for two reasons.

Complexity KLEE, together with LLVM, are very large and complex pieces of software. They deal with intricate details that might not be apparent by glancing over the code without having intimate knowledge of the problems being addressed. If we decided to not retain backwards compatibility by making fundamental changes, we would have had lower confidence of the correctness of our system.

Test Suite Furthermore, retaining backwards compatibility allows us to re-use the already existing test suite to check for any potential breakages. This raises our confidence in the correctness of our implementation

All of the added features of our system are enabled by adding command-line arguments when starting the simulation, with the two most important ones being `--distributed-mode` and `--distrib-runtime`.

7.2 System

We made two important changes to KLEE so that it can handle the simulation of networked software:

1. **Process Model**

   We modified KLEE so that its model represents a set of parallel worlds, each of which contains multiple network nodes executing a single process.

2. **Event System**

   In addition, we added primitives to allow for inter-node communication within each world.

Before we describe the modification, we take a look at the existing structure.

7.2.1 Existing Model

One of the most important classes in KLEE is `ExecutionState` which represents the state of a process being executed. Initially, when KLEE starts running the software under test, there is a single execution state. During the simulation, when a branch instruction whose condition can be both true and false, the current `ExecutionState` being run is "branched" (two copies created) and each copy is modified by adding its respective constraint – for the execution path taking the true branch, the constraint would be that the branch condition evaluated to true (and analogously for the false copy).

During the branching process, KLEE keeps a record of the branching history as a tree where all intermediate nodes contain no data and the leaves contain references to a `ExecutionState`. The reason why the tree represents the branching history is that on every branch, the left subtree contains the state which has the constraint that the branch condition is false while the right that it is true. This provides the "decision" history for the nth symbolic branch.

`ExecutionState` is composed of the following important sub-components:

- **Program Counters** Both the current program counter and the previous program counter are recorded.
- **Stack** The current execution stack is recorded as a vector of stack frames. A stack frame keeps track of the caller, the function being executed, local variables and any passed arguments.
- **Constraints** Each state has a constraints manager which efficiently keeps track of all the constraints applicable to the state.
- **Address Space** The address space of the state maintains all information pertinent to modelling its memory. It maintains a mapping of "memory objects" to their particular values. A "memory object" represents a specific allocation point (for example, `int a;`).

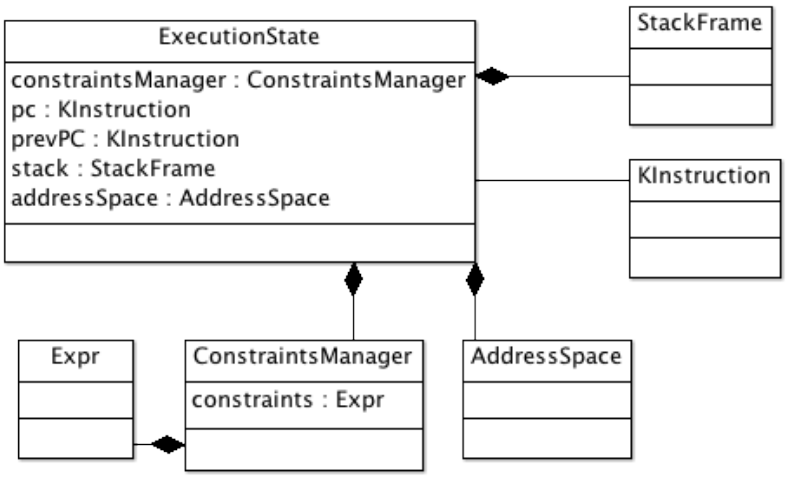


Figure 17: Showing the class diagram for ExecutionState and the most important related classes.

7.2.2 Modified Model

The first and most important change was to modify the model to represent the execution of multiple processes in a world. We added two important structures to KLEE

- **World**

  A world is represented by the `DistributedSystem` class. It has three responsibilities: to "bundle" up a set of execution states, to manage the information relevant to generating distributed test cases and to manage the data exposed for invariant checking.

- **Distributed Tree**

  The distributed tree (implemented by `DTree`) is a dual of the `ExecutionState` tree (implemented by `PTree`) – it keeps track of the worlds' branching behaviour during execution.

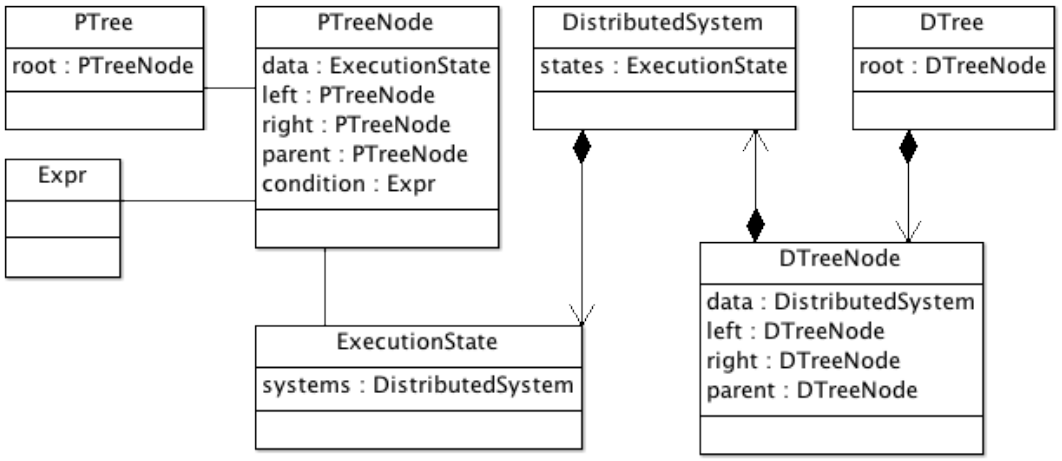
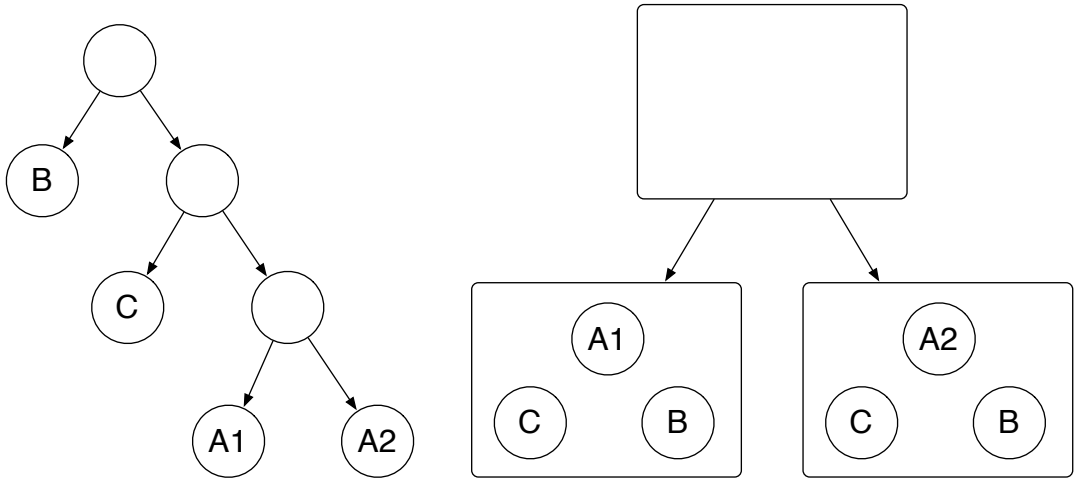


Figure 18: Showing the class diagram for DistributedSystem.



(a) Process Tree (b) Distributed Tree

Figure 19: Showing an instance diagram for processes from two different perspectives. 19a shows how the processes in the way they are organised as a process tree ignoring the modelling of worlds. In 19b, processes are grouped into worlds.

It is of great importance to note the relationship between the `DistributedSystem` and `ExecutionState` classes – many-to-many. It is obvious why a world can contain many processes, as this was one of the aims of the new model. The reason why a process (`ExecutionState`) can belong to multiple worlds is due to our on-demand copy strategy – we only branch processes on network transfer whenever absolutely necessary to preserve the semantics of world

separation.

7.2.3 Bootstrapping

When it comes to starting a simulation, the natural question arises how the initial world gets initialised. This is equivalent to asking when and how the initial "bundling" of states into a `DistributedSystem` happens.

When running under distributed mode (i.e., our system is trying to simulate distributed software), it is initially in a state which can be described as waiting to reach point the point of initial "bundling" (that is, creation of the initial world). While in this state, most of the code related to handling distributed aspects does not get executed. By definition, the point of initial world creation is reached when all existing processes are "network-ready". A process becomes "network-ready" by executing the function `klee_net_host_setup`. Crucially, "network-ready" processes are not selected by the scheduler during the initialisation stage. Usually, a wrapper `main()` functions acts as the entry point for symbolic execution. Furthermore, a variable is made symbolic and branched on it, thus creating several processes where each will represent a process running on a different network node. The code below demonstrates how to bootstrap a client and a server process.

Listing 6: Bootstrapping the initial world.

```
1  int main(int argc, char* argv[]) {
2    klee_net_add_network("net0");
3
4    int machine = 0;
5    klee_make_symbolic(&machine, sizeof(machine), "machine");
6
7    if(machine == 1) {
8      uint32_t ip = klee_net_make_ipv4(10,0,0,1);
9      klee_net_add_interface(1, "net0", NULL);
10     klee_net_add_ipv4_address(ip, 0, 1, 1, NULL);
11     klee_net_host_setup("server", 1, 1);
12     return main_server();
13   }
14
15   uint32_t ip = klee_net_make_ipv4(10,0,0,2);
16   klee_net_add_interface(1, "net0", NULL);
17   klee_net_add_ipv4_address(ip, 0, 1, 1, NULL);
18   klee_net_host_setup("client", 0, 0);
19   return main_client();
20 }
```

Our system will execute the program as follows:

1. Execution starts at line 3. This function will add a network named "net0" which network nodes can attach themselves to. This network name would be used as an argument in later functions. Only networks added by calling `klee_net_add_network` can be used when attaching interfaces to networks. If a network has not been added previously, the functions will return an error code.

2. At line 6, the variable `machine` is marked as symbolic thus making it possible for it to take any value in the range of `int`. It is important to note that if the variable is compared against a value, both outcomes are possible (equal and non-equal), thus creating two branches (or processes).

3. At line 7, the execution needs to decide which path to take – since the variable `machine` can take any value, the system will branch the current process into two and explore both paths (while adding the constraint `machine == 1` to one process and `machine != 1` to the other).

4. Assume that the scheduler always picks the process with the constraint `machine == 1`. Lines 8, 9 and 10 are executed which set up the network interface on the calling process. The effect of the functions is that the calling process will be connected to the network `net0` with an IPv4 address of 10.0.0.1 and the default subnet mask of 255.255.255.0.

5. After executing line 12, the scheduler will no longer pick the process that went down that path. The function call will also set the hostname of the calling process to `server` and the second argument indicates that the process needs to be executed until it blocks – essentially, defining our initialisation order.

   As there is only one other process left (as the one that blocked at line 12 is now excluded from being considered), the scheduler will start picking the process which took the `machine != 1` path.

6. Lines 16, 17 and 18 get executed which set up the network interface for the second process – it will be connected to the same network as the server process but with an IPv4 address of 10.0.0.2 (same subnet mask of 255.255.255.0).

7. After line 20 is executed, the system is in a state where all existing processes are "network-ready". At this point, a world is created (`DistributedSystem`) and all states added to it. This concludes the bootstrapping phase. Figure 20 shows how the world is seen by our system at this point.

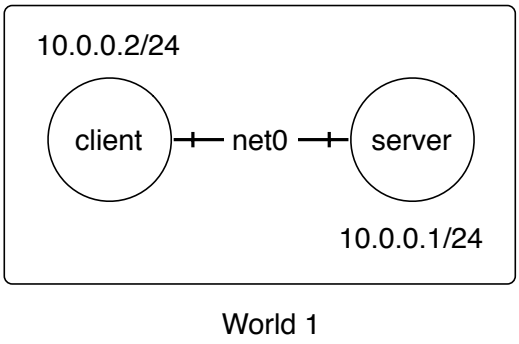


Figure 20: Showing the initial world after bootstrap.

7.2.4 Event System

The event system is responsible for all exchanges of events between processes. It is implemented as an extension of the tool itself as it is the only place that has global knowledge about all processes. In order to make it possible to transfer events, each process (`ExecutionState` class) is extended to contain a network "inbox" (`NetworkStack` class). Its two primary responsibilities are to manage the network configuration of the current process and the management of events that it receives.

Event When our system deals with events internally, it uses the `NetworkEvent` class. A `NetworkEvent` class represents the composition of a primitive event and the symbolic data associated with it. A primitive event is defined as:

Listing 7: Primitive event definition.

```
enum net_event_type {
  net_event_type_none    = (0),
  net_event_type_data    = (1 << 0), // data packet
  net_event_type_request = (1 << 1), // request for connection
  net_event_type_reply   = (1 << 2), // reply to a request
  net_event_type_close   = (1 << 3)  // connection closed
};

struct net_event {
  uint16_t type;
  uint64_t identifier;
  uint32_t from_ip, to_ip;
  uint16_t from_port, to_port;
  uint32_t req_id;
  uint32_t con_id;
} __attribute__ ((__packed__));
```

There are several important details to notice. The values for the `type` of a primitive event can be used as a bit-mask due to their definition. This allows us to re-use the `net_event` structure when searching for events by specifying to match a set of types. For example, if a process wants to wait for either data packets or for a signal that the connection has closed, it can specify a type of
`(net_event_type_data|net_event_type_close)`.

Furthermore, note that each event has a 64bit unique identifier – this is so that processes can detect duplicates if needed. The `req_id` and `con_id` fields serve the role of keeping track of "conversations", that is, sequences of related events. For example, when a process tries to connect to a server via `connect()`, it will set `req_id` not a non-zero value and wait for replies (of type `net_event_type_reply`) only to that `req_id`.

Data Transfer The networking system is responsible for the transfer of data between processes. We can see from the definition of a primitive network event that it has no reference to any data. The reason for that is that a primitive event represents a kind of "signal". When processes send primitive events, they can pass a pointer to an opaque sequence of data which will internally get associated with the event (on the **system** side in the `NetworkEvent` class). Processes use the function `klee_net_event_put(net_event* inEvent, uint8_t* data)` to send data across the network. This is a "special" that gets handled by the system as follows:

1. The primitive event is checked for validity (i.e., it must contain a well-known type, etc.).
2. If `data` is not a `NULL` pointer, it is looked up in the address space of the calling process. The value of memory objects is seen internally as instances of the `ObjectState` class. If the pointer points to a valid `ObjectState`, it is cloned so that any modifications to it at later times should not affect the recipient of the data (essentially, we are deep-copying the memory contents).
3. The set of destination processes is computed. If necessary to preserve the semantics of world separation, processes are cloned.

4. The event (represented by the `NetworkEvent` class) is added to the network inbox of each destination.

One subtlety during the second step is that it requires extra care if the transfer involves any symbolic data. One of the fundamental building blocks of the memory model is the "array". Each "array" has a unique global identifier and can be one of the following:

- **Concrete**

  A concrete "array" is composed of a sequence of constant expressions (`ConstantExpr` class). In turn, a `ConstantExpr` represents a constant expression – an arbitrary precision integer (backed by the LLVM class `APInt`).

- **Symbolic**

  A symbolic array is not composed of any sub-values. In essence, an instance of a symbolic array is used as a unique in-memory identifier for that array. The array is subsequently used in expressions which are added as constraints to the process. For example, an "array" can be created with the identifier `arr27` and we can add a constraint that says "the value at index 0 in arr27 is not equal to 3".

If an `ObjectState` instance contains symbolic data, it will contain reference to "arrays" that are symbolic. As a consequence, if we just cloned the `ObjectState` when symbolic data is present, the recipient process would receive the data unconstrained. In order to correctly transfer the data that the sender we need to:

1. Compute the set of symbolic arrays that are referenced by the `ObjectState` instance.
2. Compute the transitive closure of all constraints referring to the set of symbolic arrays.
3. Add the constraints to the destination state.

Listing 8 shows the code that computes the transitive closure of the constraints that refer to symbolic arrays in the transferred data. Line 1 declares the set of constraints should be added to the destination state. The `if` statement on line 2 is needed because some events do not carry data in which case there is no work to do. The for loop that starts on line 5 will add the symbolic arrays contained in the data to be transferred to the set `worklist`. The `while` loop on line 12 will iterate over a work list until it becomes empty. We also keep track of symbolic arrays that we have processed using the set `processed` in order to avoid infinite looping due to cycles in the constraints. During each iteration of the while loop, we pick a symbolic array (line 13) from the work list and we iterate over each constraint in the sender process (line 17). If a particular constraint contains a reference to the symbolic array (line 22), we add the constraint to the set of constraints (line 23) and then add any symbolic arrays to the work list contained in the constraint (lines 24, 25, 26). At the end of the while loop, the set `constraints` will contain all the constraints that need to be added to the destination state to transfer symbolic data.

Listing 8: Computation of the transitive closure of constraints.

```
1  std::set<ref<Expr> > constraints;
2  if(writableData) {
3    // first, we need to find all symbolic arrays that this data refers to
4    std::set<const Array*> worklist;
5    for(unsigned i = 0, count = writableData->size; i < count; ++i) {
6      ref<Expr> expr = writableData->read8(i);
7      Expr::getSymbolicArrays(expr, worklist);
8    }
9
10   // now we have to find all constraints that contain those symbolic arrays (
         transitively)
11   std::set<const Array*> processed;
12   while(!worklist.empty()) {
13     const Array* array = *worklist.begin();
14     worklist.erase(array);
15     processed.insert(array);
16
17     for(ConstraintManager::const_iterator cit = state->constraints.begin(), cend
           = state->constraints.end(); cit != cend; ++cit) {
18       ref<Expr> c = *cit;
19       std::set<const Array*> constraintArrays;
20       Expr::getSymbolicArrays(c, constraintArrays);
21
22       if(constraintArrays.count(array) > 0) {
23         constraints.insert(c);
24         for(std::set<const Array*>::iterator arIt = constraintArrays.begin(),
               arEnd = constraintArrays.end(); arIt != arEnd; ++arIt) {
25           if(processed.count(*arIt) == 0)
26             worklist.insert(*arIt);
27         }
28       }
29     }
30   }
31 }
```

Blocking In order to be able to implement any blocking behaviour required by the POSIX APIs, the networking stack keeps track on the current wait condition that determines whether a process is blocked. The blocking condition is composed of:

- **Events Mask**

  The events mask is defined as a set of `net_event` structures (as they have dual roles – both as representing a "signal" and also a search criteria for "signals").

- **Minimum Count** The minimum number of events.

- **Timeout** A timeout specified in seconds since the Unix epoch.

A process is deemed blocked if and only if the number of events matching the event mask is less than the minimum count and the time of timeout (if present) has not been reached. On top of the notion of process being blocked, we define two derived states for a process:

- **Runnable** A process is runnable if and only if it is not blocked or timed out.

- **Stalled** A process is stalled if and only if it is blocked and does not have a timeout.

7.2.5 World Branching

Our system requires modification to the branching behaviour so that whenever a single process branches, we need to record the branch in the distributed tree and correctly maintain information about the worlds. There are two reasons why states would need to branch: due to a LLVM bitcode branch whose condition can be both true and false or due to preservation of the semantics of world isolation whenever processes are communicating.

Code Branch In the case where a branch occurs due to the code, all we have to do is split the current world into two – in one of the resulting worlds we add all previous processes and one of the branched ones. Analogously, we do this the same for the other world except we add the other copy of the branching process. For a visualisation of the process, please refer to Figure 7 on page 27.

World Isolation This case occurs when processes within a world are sending data and we need to ensure that world isolation is separated. We provide a high-level description of the algorithm as the actual implementation contains too many implementation artefacts that are irrelevant. The following algorithm branches any necessary processes – its inputs are a source process and an event to be sent.

1. **Destination Calculation**

   In this step, we calculate all processes in all worlds that the source belongs to that should receive the event. We call those the destination processes and we record two bits of information: the destination process and a pair of the world and the destination process (as we iterate over the worlds that the source belongs to).

   At the end of this step, we have a set of destination processes and a set of pairs of destination process & world.

2. **World Separation**

   In this step, we iterate over each destination process. During each iteration we calculate two sets of worlds – the worlds that should receive the event and the worlds that should not receive the event. We calculate the worlds that should receive the event by iterating over the pairs of destination processes & worlds and looking for matches. Note that the worlds contained in the set contain both the destination and source processes.

   We also calculate the set of worlds that should not receive event but would have received it if we just sent it to the current destination process – we do this by just subtracting the set of worlds which should receive the event from all the worlds that the destination belongs to.

3. **World Branching**

   The set of worlds which should not receive the event are precisely the set of worlds which would violate world separation semantics if we sent the event to the current destination. We just create a single copy of the current destination and exchange it with the destination process in all the worlds which should not receive the event.

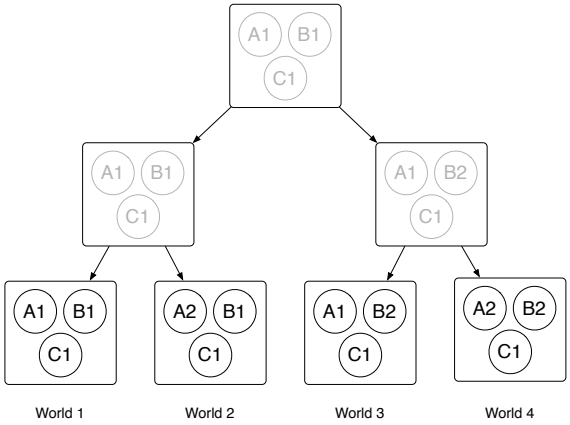


Figure 21: Showing the initial configuration before the algorithm has started executing.

In order to provide a better representation of the algorithm, we demonstrate how it works in a specific situation. In this particular example, we have 3 processes in each world - A, B and C. Let B have an IPv4 address of 10.0.0.1 and let C have an IPv4 address of 10.0.0.2. Assume that that from the initial state, B branched and then A branched – we end up with 4 worlds and 5 distinct processes (3 initial ones and 2 copies for the 2 branches). Assume that process B1 (i.e., process B which branched at took the false branch condition) wants to send an event to the process with IPv4 address 10.0.0.2.

1. **Destination Calculation**

   In this step, the algorithm will iterate over World 1 and World 2 (as those are all the worlds that B1 belongs to) and compute the destination states by using the rules for IPv4 routing of network packets. The result of this step is shown in the figure below.

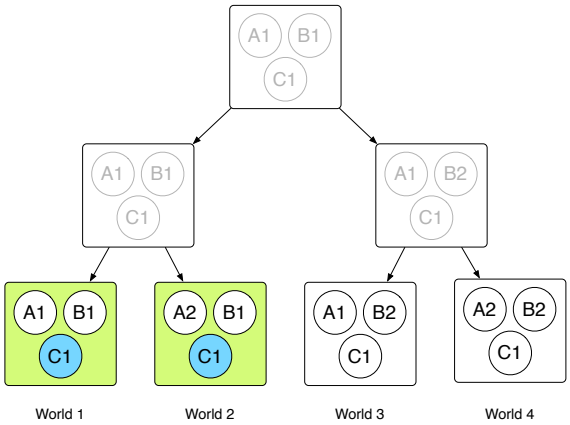


Figure 22: Showing the state after the first step. The processes in blue (only C1) are the destination processes and together with the worlds (coloured in green) form the pairs – two pairs to be exact, (World 1, C1) and (World 2, C1).

2. **World Separation**

   In this step we start by iterating over all destinations – in this case, only C1. Then we calculate the worlds which should receive the event by going over the pairs that we computed in the step above – in this case, the worlds that should receive the event are

World 1 and World 2 (coloured in green). The worlds which should not receive the event are defined as all the worlds C1 belongs to minus the worlds that should receive it – in this case, World 3 and World 4 (coloured in red).

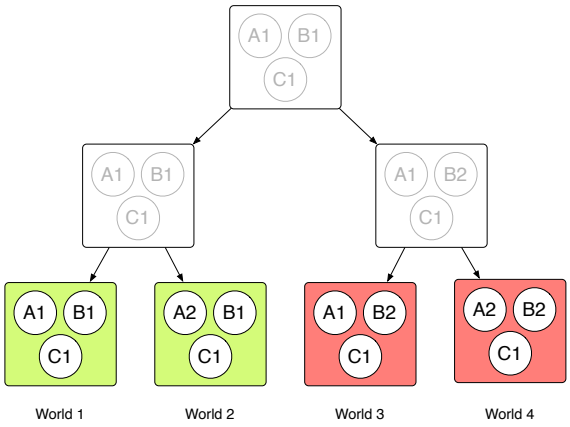


Figure 23: Showing the worlds that should receive the event in green (1 and 2) and the worlds that should not in red (3 and 4).

3. **World Branching**

   In this step, we create a copy of C1 and replace its occurrence in World 3 and World 4 (which is shown as C2 coloured in blue). It now safe to add the event to the network inbox of C1 as it will not break the semantics of world separation.

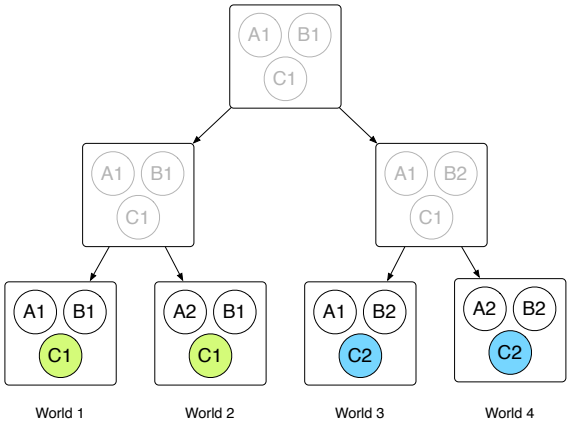


Figure 24: Showing C1 in green which will receive the event and the copy of C1 (C2 in the diagram) that was created to preserve world separation semantics.

7.2.6 Scheduler

The scheduler is responsible for choosing the next process to execute. It is a very critical part of the system for two reasons:

- **Code Coverage**

  The searcher has one of the greatest impacts on the code covered as it has to make the decision which processes to run. If, for example, it chooses paths that lead to very heavy

branching in tight loops, our system would run out of memory before exploring much of the code.

- **Performance**

  The performance of the searcher directly affects the performance of the whole system as the searcher is asked to select a process for execution after every instruction.

There are two key additional restrictions when scheduling processes in distributed mode:

- **Eligibility**

  If a process is blocked, then it cannot be chosen by the scheduler. This creates a situation whereby a backtracking scheduler would be penalised for making choices which end requiring it to backtrack.

- **Timeout**

  The scheduler is also responsible for keeping track of any timeouts for processes. Any inefficiencies have a direct impact on performance.

Our solution runs in O(1) time during selection of states – no computations whether a state is blocked are performed during scheduling. We achieve this by:

- **Tree Augmentation**

  Each node in the distributed tree (world tree) gains two additional boolean variables – runnable and stalled. For the leaf nodes (which represent the worlds), the values for the booleans are just copies for the world's runnable and stalled boolean values. In turn, a world is runnable if and only if it contains at least one process that is runnable. A world is stalled if and only if all the processes it contains are stalled.

  For intermediate tree nodes, the values are defined as follows:

  - **Runnable** A node is runnable if and only if at least one of its children is runnable.
  - **Stalled** A node is stalled if and only if all of its children are stalled.

- **Flag Caching & Propagation**

  Augmenting the tree is the first step in achieving high performance. In addition, the tree flags are always conservatively updated and propagated as soon as the flags change for a single process. The reason why this is beneficial and necessary for attaining a very fast searcher is due to the fact that the ratio of scheduling processes to changes in the flags is incredibly high – scheduling happens after every instruction while changes in the flags happen only when the software calls a blocking system call.

By the design of our data structure, the searcher does not need to backtrack as it can just inspect a node's runnable property to find out whether at least one process exists down the subtree that is eligible for execution.

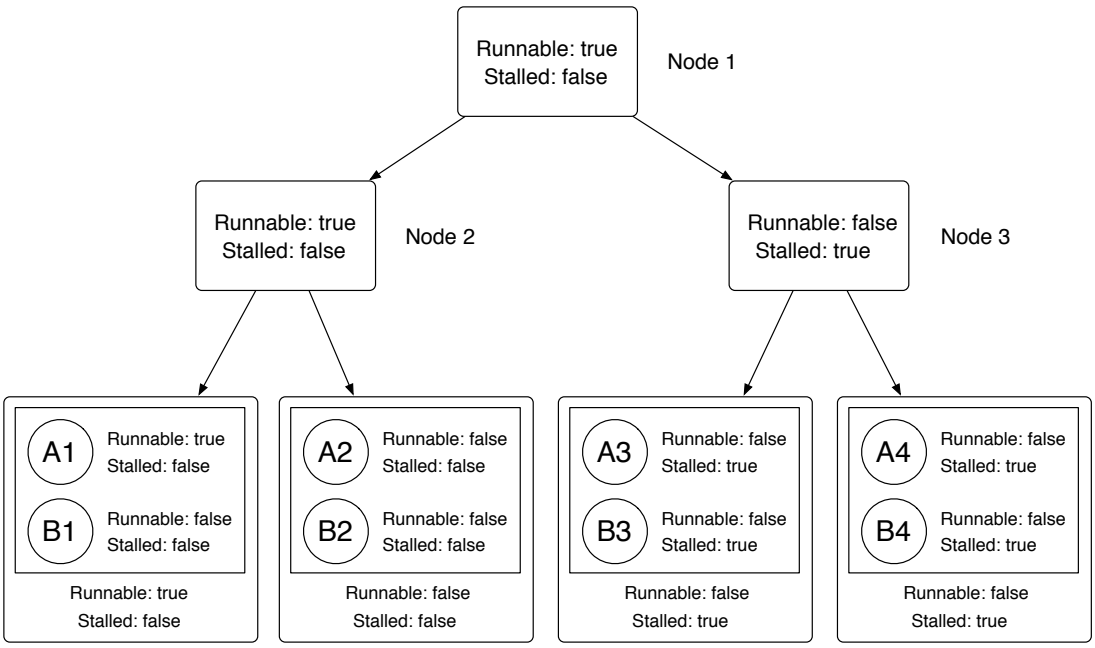


Figure 25: Showing the augmented world tree. Note how Node 3's stalled flag is true – by our definition, that means that all worlds that can be reached from that node will be stalled.

Timeout The ability to specify timeouts of of blocking system calls (e.g., `select()`) means that our system needs to handle the case where all processes are blocked and waiting on timeout to expire. The distributed tree makes it easy to detect that case by just looking at the root node – if runnable is false and stalled is false, that means that all processes are blocked waiting for their timeouts to expire.

We needed to extend the base scheduler class to include two methods that are called during simulation:

- `addWaitingStateWithTimeOut(ExecutionState* state, uint32_t absSeconds)`

  This method is called whenever a process became non-runnable (i.e., runnable is false) with a specific timeout. It signifies that the scheduler is responsible for managing the timing out of the process.

- `removeWaitingState(ExecutionState* state)`

  This method is called whenever the scheduler is no longer responsible for managing the timing out of a particular process. This usually happens under two circumstances – when the process receives a message that it is blocked on or when the world becomes deadlocked and all processes are terminated early.

Our scheduler implements those methods by keeping track of the states ordered by their timeout – this gives us $O(lg(n))$ insertion and search time. More importantly, it allows us to time out multiple processes in one go – all we need to do is find the index of the first process whose timeout is greater than the current time and then time out all processes that appear before that index in the sorted list.

Deadlock Another case that arises when simulating distributed software is the possibility of universe deadlock – all processes across all worlds are waiting to receive events from each other

and none of them have any timeouts. In this case, the scheduler cannot choose any state that can be run – although this is an assumption in the scheduler interface (the method which returns the state to run must always return a valid value). We modified the main simulation loop to call a method on the scheduler named `isDeadlocked()` which returns whether the universe became deadlocked after the execution of the last instruction. If the method returns `true`, then all existing processes are forcefully terminated and the simulation ends. We chose to add an additional method in order to keep the backwards compatibility of all existing schedulers without having to modify them.

It is crucial to note that the `isDeadlocked()` method is called after every instruction and has to be be very fast – any on-the-fly computation would seriously impact the performance. By the design of our distributed tree, universe deadlock can be detected by just inspecting the root node's runnable and stalled property – if stalled is true, then, by the definition of how the value is computed, we have universe deadlock.

Sequencing Finally, the scheduler also has the responsibility to respect the initialisation order within each world. The sequence number that users can assign to processes is interpreted as "initialise the processes in that order until they block". This is performed by keeping a flag as part of each process whether it has blocked during its lifetime and taking it into account when performing round-robin scheduling at the world level.

7.2.7 Invariants Framework

The invariants framework is composed of three parts: the parser, the runtime and the interpreter.

Parser The parser is a hand-written recursive-descent parser that reads an invariant program passed as a command line argument. The output of the parser is an intermediate representation of the language.

Runtime The runtime provides a memory model to support the execution of the program. It primarily composes a heap and a stack where the heap is just a mapping of addresses to objects. The language itself is fully object oriented and the runtime has built-in support for strings, integers and sequences of bytes.

Interpreter The interpreter runs every time there is new invariant data exposed. It creates a new runtime (which is very cheap operation) and evaluates all invariants, reporting any violations. The executor (represented by the class `Executor` and responsible for driving the symbolic executor) keeps a flag whether any new invariant data has been exposed after the last instruction executed. This means that invariant evaluation happens only as necessary and has a virtually zero cost as data for invariant checking does not get exposed very often.

Interface There are two abstract classes that are used in the invariants framework whose primary role is to provide the flexibility to use the subsystem in a variety of applications (and not just being tied to our distributed symbolic execution engine).

Listing 9: Abstract data source class declarations.

```
1 class DataProvider {
2 public:
3   virtual void* retrieveData(const std::string& node, int state, const std::
        string& key, int& size) = 0;
```

```
4   virtual void releaseData(void* data) = 0;
5 };
6
7 class NodeProvider {
8 public:
9   virtual void retrieveNodes(std::vector<std::string>& outNodes) = 0;
10 };
```

The `Node` provider "interface" is needed to provide the implementation of the `sys.nodes()` method that can appear in Minvariant programs. The `DataProvider` "interface" is used to retrieve the opaque sequences of bytes for the triplets of node, state and key that are specified the Minvariant programs themselves. When retrieving the data for a specific composite key (a combination of node, state and key), the size of the data is returned by reference. In addition, the caller assumes ownership of the opaque byte sequence and must release it when not needed by calling the `releaseData` method.

7.2.8 Replay Framework

The primary role of the replay framework is to be able to generate dtest files which compose (by referencing) existing ktest files and including network topology information. Each world (i.e., `DistributedSystem` class) features an object that represents a distributed test case (`DistributedTest` class). A distributed test just composes a set of process tests – each of which contains the filename of the process' ktest file, its network interfaces and routes together with a few auxiliary book-keeping properties.

When the last process that is part of world terminates (either in a controlled or uncontrolled manner), the world's distributed test file is written to disk, in similar fashion to process test files.

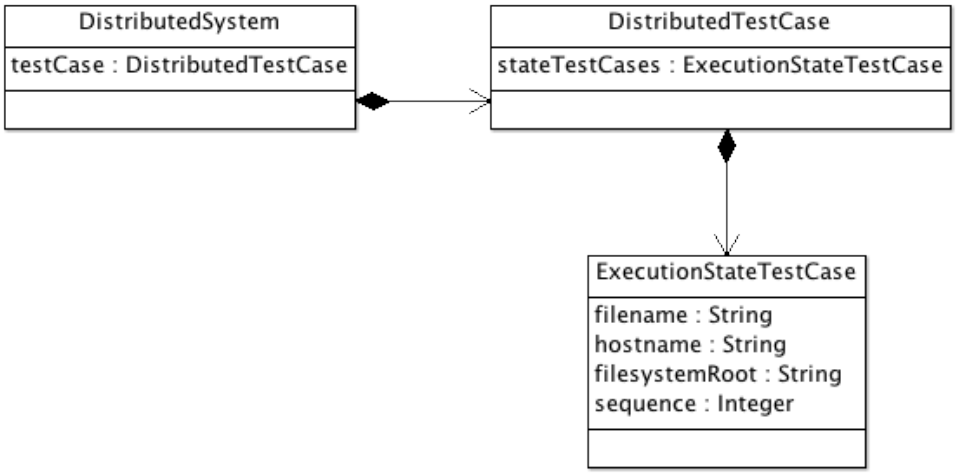


Figure 26: Showing the class diagram for DistributedTest and its most important relationships.

7.2.9 Code Coverage

Another necessary addition was the ability to track code coverage of the LLVM bit code on a per-function basis across multiple runs. KLEE already had most of the infrastructure to track coverage at the function level albeit with two limitations – it tracked coverage at the program level and included all functions when calculating the total coverage. We modified the system

so that a filename can be passed as a command line which includes a list of function and only those functions will be included as part of the coverage metric. In addition, we added the ability to track instruction coverage on a per function basis so that we get coverage percentages for each function. We do this by storing which instructions (for example, the 5th, 6th, etc.) of a function were executed and this is persisted. A tool was written (`klee-cov`) which can combine the information from multiple runs and provide a final aggregate coverage metric on a per function-basis.

7.3 Runtime

The rest of the system is implemented in C and is responsible for providing implementations of the POSIX APIs. The implementation can be split into two parts:

- **Networking**

  The networking part provides an implementation of both UDP and TCP sockets using the special internal APIs exposed by our system to communicate with other processes.

- **Filesystem**

  The filesystem part implements a rudimentary system has has uniform support for for files and directories and can support symbolic pathnames.

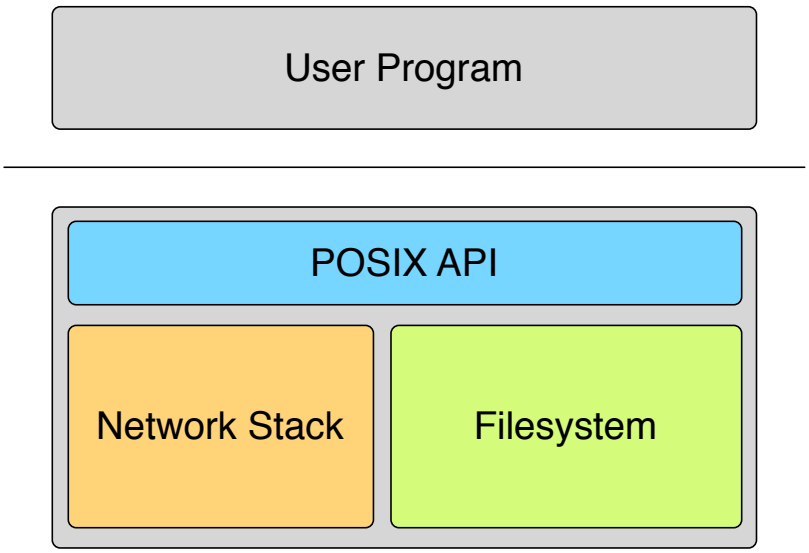


Figure 27: Showing the logical organisation of the user program being simulated with relation to the runtime which we implement. The functions defined by the POSIX standard are implemented using our internal network and filesystem subsystems.

In both cases, the subsystems provide their own custom APIs to manage them internally and then the functions at the POSIX level make use of the aforementioned subsystems. In order for our system to be able to fully run a minimalistic web server[3], we had provide non-stub implementations for the following set of functions:

7.3.1 Special Functions

Most of the runtime, especially the networking part, depend on special primitives provided by our extensions which are implemented outside the runtime. All of them are part of the

open	write	read	close	opendir	closedir	readdir
rewinddir	seekdir	telldir	dup	dup2	lseek	fcntl
fstat	select	getcwd	chdir	fchdir	sleep	socket
connect	bind	listen	accept	recv	recvfrom	send
sendto	getsockopt	setsockopt	getsockname	getpeername		

Table 1: Set of POSIX functions implemented.

`netklee.h` header file and the most important ones include:

- `klee_net_will_get_events`

  This is one of the most important special functions that are used to implement blocking behaviour. It accepts a set of event masks, a minimum event count and a timeout. The scheduler will ensure that the calling process is blocked at this function call until either there are enough events matching the event masks or the time out has expired.

- `klee_net_has_pending_events`

  This function allows the calling process to inspect its network "inbox" without blocking – it accepts a set of event masks and returns the number of events that match the set.

- `klee_net_events_get`, `klee_net_event_put` & `klee_net_event_remove`

  These three functions allow the calling process to send events to other processes and also retrieve & remove events from its "inbox". Note that retrieving an event does not implicitly remove the event from the "inbox", it has to be removed explicitly with a call to `klee_net_event_remove`.

- `klee_net_event_data_alloc` & `klee_net_event_data_free`

  This pair of functions are similar to `malloc` and `free` but with a twist – instead of passing the number of bytes to allocate, `klee_net_event_data_alloc` accepts an event identifier and it will return the byte sequence associated with the event (if any). The caller owns the memory and is responsible to dispose of it by calling `klee_net_event_data_free`.

- `klee_invariant_expose_data`

  This functions allow the calling process to expose data for invariant verification.

- `klee_net_write_filesystem`

  This function will persist any current in-memory files to the disk for later inspection.

7.3.2 Networking

The networking APIs are implemented on top of an internal "net stack". The "net stack"'s responsibility is to model network sockets and provide the ability to read and write data to the sockets. It has no awareness of file descriptors or how it fits into the bigger picture – it is meant to be small and re-usable. There are two central data structures – socket and socket data. A socket has a pointer to a linked list of socket data structures – each socket data structure represents an individual data packet at the network level in the OSI model. There is a set of high level functions that allow for efficient manipulation and use of the data structures without managing any intricate details. Using a linked-list makes sense in this case as sockets do not provide random access and data is always read as a stream.

Listing 10: Network Stack structure definitions.

```
enum NetStackSocketType {
  kNetStackSocketTypeUndefined,
  kNetStackSocketTypeUDP,
  kNetStackSocketTypeTCP
};

enum NetStackSocketState {
  kNetStackSocketStateUndefined,
  kNetStackSocketStateListening,
  kNetStackSocketStateConnected,
  kNetStackSocketStateClosed
};

enum NetStackSocketFlag {
  kNetStackSocketFlagNone        = (0),
  kNetStackSocketFlagNonBlocking = (1 << 0)
};

enum NetStackSocketInternalFlag {
  kNetStackSocketInternalFlagNone      = (0),
  kNetStackSocketInternalFlagSeenClose = (1 << 0)
};

struct NetStackSocketData {
  uint32_t remote_ip;
  uint16_t remote_port;

  uint8_t* data;
  // size is total size of data
  uint32_t size;
  // cursor points to the beginning of unread data
  uint32_t cursor;

  // linked list next
  struct NetStackSocketData* next;
};

struct NetStackSocket {
  // internal state
  uint8_t active;
  enum NetStackSocketType type;
  enum NetStackSocketState state;
  enum NetStackSocketFlag flags;
  enum NetStackSocketInternalFlag iflags;

  uint32_t local_ip, remote_ip;
  // ports in native order
  uint16_t local_port, remote_port;
  // for stream connections
  uint32_t connection_id;

  // the head of the linked list
  // invariant: input_head is not NULL, then always has data
  struct NetStackSocketData* input_head;
};
```

Each socket has a flag (`active`) whether it is currently in use. The `type` field indicates whether it is a TCP or UDP socket (the only two types of sockets supported so far). TCP

sockets can also have an associated state – for example, a socket which has been used in a `listen()` function call be in a listening state. In order to correctly implement non-blocking behaviour of sockets, we need to record the fact that it is a non-blocking socket which is done in the `flag` field. The field `iflags` is a shorthand for internal flags. Currently, we only have one internal flag which indicates whether the socket has already told the caller that it has been closed. This is used to implement the correct semantics when reading from an already closed socket. In addition, the socket stores any IPs and ports that it has been bound to and for TCP connections, the connection identifier.

Each data packet (`struct NetStackSocketData`) needs to keep the remote IP and address because a UDP socket can receive packets from multiple sources. For TCP clients, the values of those fields across all packets will be the same. Finally, each data packet has a cursor field which indicates how much data has been read, as the caller might request to read less bytes than available on the network.

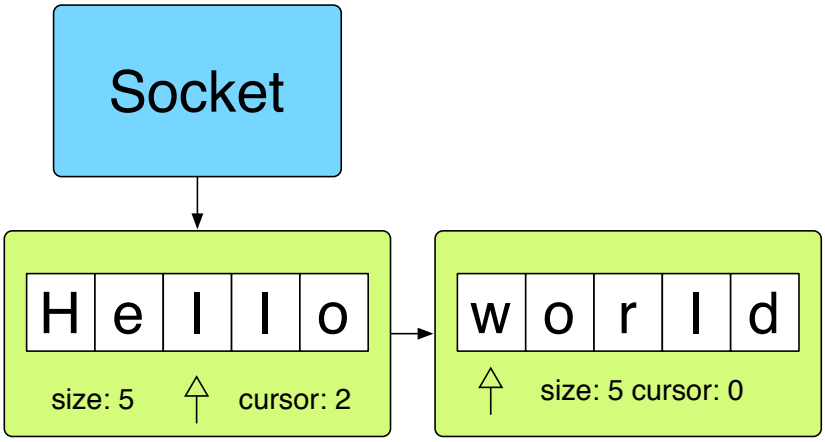


Figure 28: Showing an instance of a socket where the sender has sent two packets – the first containing the bytes **Hello** and the second **world**. The receiver has so far consumed 2 bytes from the received data. After the remaining 3 bytes from the packet are consumed, the first data packet will be discarded and the socket will point directly to the second packet.

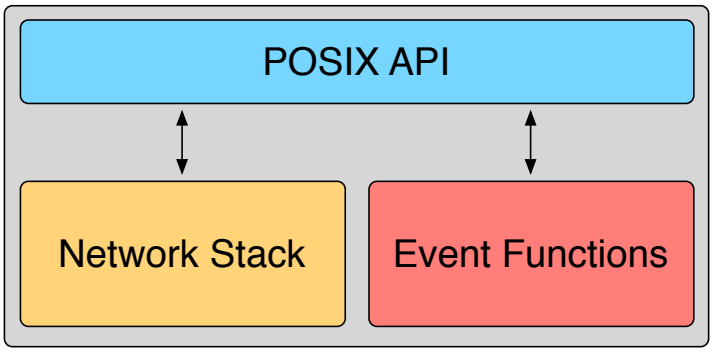


Figure 29: Showing the relationship between the POSIX function APIs, our internal network stack and the "special" event functions that expose event transfer between processes. The arrows indicate API usages. Note that there is no connection between the network stack and the special event functions – this provides separation, modularity and reusability as the network stack does not depend on a particular source of data.

We will now explain how the important POSIX socket APIs are implemented by using the our internal network stack and the special functions. We omit error handling and any code related to providing additional features, like failure injection which is part of the actual implementation.

accept() & connect() are used to establish a TCP connection between two processes. The functions use two event types to perform their function:

- `net_event_type_request` represents a request to establish a TCP connection.
- `net_event_type_reply` represents a reply to the a request to establish a TCP connection.

`accept()` blocks waiting for a connection request and sends a reply once it receives one. `connect()` is the dual – it sends a connection request and waits for a reply to its request.

Listing 11: accept() and connect() implementation with error handling omitted.

```
1  int connect_tcp(struct NetStackSocket* s, const struct sockaddr *serv_addr,
       socklen_t addrlen) {
2    // send the connection request
3    struct net_event request = {net_event_type_request, 0, s->local_ip, to_ip, s->
         local_port, to_port, NetStackNextRequestID++, 0};
4    klee_net_event_put(&request, NULL);
5
6    // wait for the connection reply
7    struct net_event reply_mask = {net_event_type_reply, 0, to_ip, s->local_ip,
         to_port, s->local_port, request.req_id, 0};
8    klee_net_will_get_events(&reply_mask, 1, 1, 0, 0);
9
10   struct net_event reply;
11   if(klee_net_events_get(&reply, 1, &reply_mask, 1) == 1) {
12     // remove reply event
13     klee_net_event_remove(reply.identifier);
14
15         // setup socket
16         ...
17     return 0;
18   }
19
20   errno = ETIMEDOUT;
21   return -1;
22 }
23
24
25 int accept_tcp(struct NetStackSocket* s, struct sockaddr *addr, socklen_t *
       addrlen) {
26   // make a mask and either block or check if we will block
27   struct net_event event_mask = {net_event_type_request, 0, 0, s->local_ip, 0, s
         ->local_port, 0, 0};
28   if((s->flags & kNetStackSocketFlagNonBlocking) && !klee_net_has_pending_events
         (&event_mask, 1, 1)) {
29     errno = EWOULDBLOCK;
30     return -1;
31   }
32
33   // possibly block
34   klee_net_will_get_events(&event_mask, 1, 1, 0, 0);
35
```

```
36   struct net_event event;
37   if(klee_net_events_get(&event, 1, &event_mask, 1) == 1) {
38     // remove event
39     klee_net_event_remove(event.identifier);
40
41     uint32_t con_id = NetStackNextConnectionID++;
42     // reply to the request
43     struct net_event reply = {net_event_type_reply, 0, s->local_ip, event.
           from_ip, s->local_port, event.from_port, event.req_id, con_id};
44     klee_net_event_put(&reply, NULL);
45
46     // try to get a socket + fd and associate with socket
47     struct PosixDescriptor* client_descriptor = posix_layer_allocate_socket();
48         ...
49
50     // return client socket
51     return client_descriptor->fd;
52   }
53
54   errno = ETIMEDOUT;
55   return -1;
56 }
```

`connect()` starts off (line 3) by creating a connection request event and appropriately setting the receiver and sender IPs and ports. Notably, it assigns a locally unique request identifier to the event and at the same time increments it so that the invariant is preserved – the value of `NetStackNextRequestID` always contains the next identifier that can be used. On line line 4, the event is sent on its own without any additional data. On line 7, an event mask (that is, a search criteria) is created that will match the possible reply from the other end – note the reversal of to / from IPs & ports. In addition, the code correctly sets the request identifier to the value that was sent. On line 8, the calling process will be blocked until a reply arrives. If the process reaches line 11, it means that it must have been unblocked (that is, received a reply), so it tries to receive the reply from the other send. `klee_net_events_get` will return the number of events retrieved and as we are asking for a maximum of one, checking for success amounts to checking for equality with 1. If we managed to retrieve an event, we need to remember to remove it from the network "inbox" which is done at line 13. Finally, after setting up internal bookkeeping information (omitted in the listing), we return to the caller at line 17.

`accept()` begins by checking whether it is a non-blocking socket (on line 28). If it is a non-blocking socket and there are no pending connection requests, it must return immediately (line 30) as otherwise it will block on line 34. When line 34 is reached, either the socket is a blocking one or it is a non-blocking one but with a pending request event in the network "inbox". In all cases, once execution reaches line 37, the process has a connection request waiting in its "inbox" which it tries to retrieve. On successful retrieval (line 39), the request is removed, a connection reply constructed (line 43) and sent to the sender of the request (line 44). Afterwards, an unused file descriptor is allocated (47) and we return the file descriptor to the caller (line 51).

sendto() has one of the simplest implementations. As the majority of the code is error handling, we only show one of the internal function that perform the data send.

Listing 12: Sending data to another process.

```
1 static uint32_t send_socket_data_raw(const void* buf, size_t len, struct
      NetStackSocket* socket, uint32_t remote_ip, uint16_t remote_port) {
```

```
2   struct net_event event = {net_event_type_data, 0, socket->local_ip, remote_ip,
        socket->local_port, remote_port, 0, socket->connection_id};
3   uint8_t* data = malloc(len);
4   memcpy(data, buf, len);
5   uint32_t status = klee_net_event_put(&event, data);
6   free(data);
7
8   return status;
9 }
```

`send_socket_data_raw` starts off by creating the primitive event to send by filling in the type, sender and recipient IPs & ports and setting the connection identifier. Then it creates a temporary buffer and copies the data that was requested to be sent into that buffer (lines 3 and 4) The reason for this implementation artefact is that `klee_net_event_put` can only send a complete memory object and not partial ranges. After the event and the data are sent (line 5), the temporary buffer is freed (line 6). Finally, we return the status of the send to the caller (line 8).

recvfrom() is the primitive that allows processes to receive data from the network. Its implementation is considerably longer and more complicated thus we will only show the most interesting part – retrieving data from the network by using the special functions and piping into our internal network stakc.

Listing 13: Receiving data from a socket.

```
1 // Returns if the socket was closed (with no other data).
2 inline static void process_network_events(struct NetStackSocket* socket, struct
      net_event* events, uint32_t count) {
3   // 1) find the first close event (+ update state of socket to closed)
4   // 2) read the data from all data events beforehand
5   // 3) remove all events that we had to process
6   // 4) if we had a close event, remove all events on that socket
7
8
9
10  // 1
11  ..
12
13  // 2
14  for(i = 0; i < close_event_idx; ++i) {
15    struct net_event* event = &events[i];
16    if(event->type == net_event_type_data) {
17      uint32_t data_size = 0;
18      uint8_t* data = klee_net_event_data_alloc(event->identifier, &data_size);
19      if(data != NULL) {
20        net_stack_socket_write(socket, data, data_size, event->from_ip, event->
              from_port);
21        klee_net_event_data_free(data);
22      }
23    }
24  }
25
26  // 3
27  for(i = 0; i < count; ++i) {
28    struct net_event* event = &events[i];
29    klee_net_event_remove(event->identifier);
30  }
31
```

```
32    // 4
33    if(close_event_idx < count)
34      sockets_remove_all_events(socket);
35  }
36
37  inline static void receive_data_from_network_single_packet(struct net_event*
        event_mask, struct NetStackSocket* socket) {
38    // block waiting for events
39    klee_net_will_get_events(event_mask, 1, 1, 0, 0);
40
41    // read the events
42    struct net_event event;
43    if(klee_net_events_get(&event, 1, event_mask, 1) == 1)
44      process_network_events(socket, &event, 1);
45  }
```

This internal function `process_network_events` receives an array of primitive network events that were pulled from the network "inbox" and its role is to read data from them. The loop on line 14 iterates over all data packet events. For each of them, it tries to retrieve the data that the sender has sent (line 18). Any data that is received is piped into the socket (line 20) and released (line 21). Note the high-level usage of the function that writes data to the socket – we do not have to worry about managing the details of how that is stored efficiently. The internal implementation of how data is stored can completely change without having to modify the upper layers. Once any data has been read and piped into the the network stack, we proceed to remove all events from the "inbox" (lines 28, 29). Finally, if a close event was received, we remove all pending events for that socket (lines 33, 34).

`receive_data_from_network_single_packet` tries fetches one event from the network by firstly possibly blocking (line 39). After the system has unblocked the process, it tries to retrieve the event matching the mask (line 43) and if successful, processes the event (line 44).

select() is one of the most important functions that is used in networked systems – it used to perform I/O multiplexing by observing the status of multiple file descriptors. Servers usually have a `select()` loop which listens for any activity on the socket that accepts new connections and also for any activity on any open connections. Due to the importance of this system call and despite its implementation length, we have included the source code in Listing 14.

The code starts off by checking whether it should simulate a failure (lines 2-12) and if it should not, then execution continues on line 18. The code creates copies of any file descriptor sets passed in – this is needed because the function parameters are both used as inputs and outputs and since we need to zero them out, we firstly copy their values. As the file descriptor sets can be `NULL`, we semantically treat it as if the user passed an empty set (lines 25, 33, 41). When we reach line 45, we have zeroed out any parameters and retained their original values. On lines 45-49, we declare several variables which will be needed later on – `socket_masks` will store the masks for any sockets that we might have to block on while `socket_masks_fds` will store the corresponding file descriptor for each mask (this is required so that we know which bit to set in the file descriptor sets given a particular event mask). `socket_mask_index` stores an index to the next unused socket mask index (and also provides the number of socket masks in use). As `select()` splits events into three types (read, write and except), we create two mask types on lines 48 and 49 (for read and except). The reason why we do not create a mask for write is due to the fact that, by design, our sockets have unlimited buffer sizes, so data can always be written to them.

One of the most important sections in the function is the for loop that starts on line 53 and finishes on line 132 – its purpose is to find whether any file descriptors are already "ready" and

turn on the appropriate bits in the file descriptor sets. Lines 54-56 read the input parameters which tell us what activity the caller is interested for the current file descriptor (variable `i`). Lines 59-63 handle the case where an invalid file descriptor was passed. Afterwards, there are two possibilities – a file descriptor can either represent a file or a socket. If it is a file (lines 65-81), then the file is ready for all all operations that were requested (we need to ensure that we only enable bits that the user requested to be checked, as exemplified by the `if` statements on lines 67, 72 and 77).

The other alternative is that the file descriptor represents a socket (lines 82-130). The easiest case is when the caller wants to write to a socket (lines 125-129) which we always allow. On other hand, if the caller is interested whether the socket can be read from or whether it was closed, things become more complex. We need to do two things to be able to answer those queries – we need to check whether we already have not received an appropriate event beforehand (e.g., there might be unconsumed data in the socket buffer) and also check the network "inbox" for any pending events. These tasks are performed in lines 95-103 and 105-114. Note that there is a variable `count` which keeps the total number of enabled file descriptors and a local variable `enabled_bit` that records whether we can potentially block on the current socket. On lines 116-122 we save the current event search mask for later – this happens if the caller requested either read or close events although none of those are available at present.

After we have processed all descriptors, execution continues on line 135. If we have found at least once "ready" descriptor or we should not block, we return immediately (line 136). Otherwise, no file descriptors were "ready" and we had a valid timeout specified. In this case, we sanity check the number of masks that we can wait on, on line 139-141 to ensure that if we continue, we do not pass invalid values to the special functions.

On line 153, we block waiting for any events that we previous saved in the for loop. Once execution reaches 156, either the timeout expired or a matching event arrived in the network "inbox". In either case, we need to loop over all event masks and check whether there is any new activity (lines 156-175). If so, we enable the corresponding bit in the file descriptor sets (lines 164-172). Finally, we return the total number of "ready" descriptors on line 177.

Listing 14: Implementation of select() system call.

```
1  int select(int nfds, fd_set *read, fd_set *write, fd_set *except, struct timeval
       *timeout) {
2    static unsigned n_selects = 0;
3    static int next_errno = EINTR;
4    unsigned sfail_interval = posix_layer.failures.select_fail_interval;
5    if(sfail_interval && (n_selects % sfail_interval == 0)) {
6      ++n_selects;
7      errno = next_errno;
8      if(next_errno == EINTR)
9        next_errno = EIO;
10     else
11       next_errno = EINTR;
12     return -1;
13   }
14
15   ++n_selects;
16
17   // firstly, copy in all fds + zero out the arguments
18   fd_set in_read, in_write, in_except;
19
20   if(read) {
21     in_read = *read;
22     FD_ZERO(read);
23   }
```

```
24    else {
25      FD_ZERO(&in_read);
26    }
27
28    if(write) {
29      in_write = *write;
30      FD_ZERO(write);
31    }
32    else {
33      FD_ZERO(&in_write);
34    }
35
36    if(except) {
37      in_except = *except;
38      FD_ZERO(except);
39    }
40    else {
41      FD_ZERO(&in_except);
42    }
43
44    // we will store masks for sockets which the caller has requested but no
          events are available
45    struct net_event socket_masks[MAX_SOCKET_BLOCK];
46    int socket_masks_fds[MAX_SOCKET_BLOCK];
47    int socket_mask_index = 0;
48    enum net_event_type read_mask_type = (net_event_type_data|
          net_event_type_request|net_event_type_reply);
49    enum net_event_type except_mask_type = (net_event_type_close);
50
51    int count = 0;
52    int i;
53    for(i=0; i<nfds; ++i) {
54      int read_bit = FD_ISSET(i, &in_read);
55      int write_bit = FD_ISSET(i, &in_write);
56      int except_bit = FD_ISSET(i, &in_except);
57
58      if(read_bit || write_bit || except_bit) {
59        struct PosixDescriptor* descriptor = posix_layer_find_descriptor_for_fd(i)
              ;
60        if (descriptor == NULL) {
61          errno = EBADF;
62          return -1;
63        }
64
65        if(descriptor->file) {
66          // file io never blocks
67          if(read_bit) {
68            FD_SET(i, read);
69            ++count;
70          }
71
72          if(write_bit) {
73            FD_SET(i, write);
74            ++count;
75          }
76
77          if(except_bit) {
78            FD_SET(i, except);
79            ++count;
80          }
```

```
81        }
82        else if(descriptor->socket) {
83          if(read_bit || except_bit) {
84            // we need to consult KLEE to find out whether there are pending
                  events
85            struct NetStackSocket* socket = descriptor->socket;
86            enum net_event_type mask_type = net_event_type_none;
87            if(read_bit)
88              mask_type |= read_mask_type;
89            if(except_bit)
90              mask_type |= except_mask_type;
91            struct net_event event_mask = {mask_type, 0, socket->remote_ip, socket
                  ->local_ip, socket->remote_port, socket->local_port, 0, 0};
92            uint8_t pending_events = klee_net_has_pending_events(&event_mask, 1,
                  1);
93
94            int enabled_bit = 0;
95            if(read_bit) {
96              int has_data = net_stack_socket_has_data(descriptor->socket);
97              int pending_data = pending_events && (event_mask.type &
                    read_mask_type);
98              if(has_data || pending_data) {
99                FD_SET(i, read);
100               ++count;
101               ++enabled_bit;
102             }
103           }
104
105           if(except_bit) {
106             int closed = (descriptor->socket->state ==
                    kNetStackSocketStateClosed);
107             int seen_closed = (descriptor->socket->iflags &
                    kNetStackSocketInternalFlagSeenClose);
108             int pending_closed = pending_events && (event_mask.type &
                    except_mask_type);
109             if((closed && !seen_closed) || pending_closed) {
110               FD_SET(i, except);
111               ++count;
112               ++enabled_bit;
113             }
114           }
115
116           if(!enabled_bit) {
117             // no pending events at the moment, so we can definitely block on
                    that mask
118             assert(socket_mask_index < MAX_SOCKET_BLOCK && "Run out of socket
                    block masks");
119             socket_masks[socket_mask_index] = event_mask;
120             socket_masks_fds[socket_mask_index] = i;
121             socket_mask_index++;
122           }
123         }
124
125         if(write_bit) {
126           // we can always write
127           FD_SET(i, write);
128           ++count;
129         }
130       }
131     }
```

```
132   }
133
134   // if we found at least one event or we shouldn't block, then return
          immediately
135   if(count != 0 || (timeout && timeout->tv_sec == 0 && timeout->tv_usec == 0))
136     return count;
137
138   // there's nothing we can wait on
139   if(socket_mask_index == 0) {
140     errno = EINVAL;
141     return -1;
142   }
143
144   // so we've got no fds that are ready and we have to block
145   uint32_t secsTimeout = 0;
146   uint32_t usecsTimeout = 0;
147   if(timeout) {
148     secsTimeout = timeout->tv_sec;
149     usecsTimeout = timeout->tv_usec;
150   }
151
152   // block waiting for events
153   klee_net_will_get_events(socket_masks, socket_mask_index, 1, secsTimeout,
          usecsTimeout);
154
155   // now, lets re-evaluate the situation
156   for(i = 0; i < socket_mask_index; ++i) {
157     struct net_event mask = socket_masks[i];
158     if(klee_net_has_pending_events(&mask, 1, 1)) {
159       int fd = socket_masks_fds[i];
160       if(mask.type != net_event_type_none) {
161         int read_bit = FD_ISSET(fd, &in_read);
162         int except_bit = FD_ISSET(fd, &in_except);
163
164         if(read_bit && (mask.type & read_mask_type)) {
165           FD_SET(fd, read);
166           ++count;
167         }
168
169         if(except_bit && (mask.type & except_mask_type)) {
170           FD_SET(fd, except);
171           ++count;
172         }
173       }
174     }
175   }
176
177   return count;
178 }
```

7.3.3 Filesystem

The filesystem is implemented in a similar fashion to the network stack – independent of its users and without any external dependencies. The filesystem implementation uses two structures:

- **FileSystemHandle**

  A handle represents a currently open file and it has two main responsibility: to maintain the state information for the open file description and to contain a pointer to the actual

data. The importance of separating this structure from the data is important because it allows us to correctly implement file descriptor duplication – two file descriptors can just point to the same handle. Reading from either of them will advance a shared cursor, as required by the POSIX standard.

- **FileSystemData**

  This structure represents the data for an item on the filesystem. The actual fields that are in use depend on the type of item – we support symbolic files and directories and also native files and directories. Unifying the handling of both native and symbolic files & directories simplifies the code and makes it easier to extend.

The definition of the two structures is shown below:

Listing 15: Filesystem structures.

```
1  // A handle is unused iff refcount == 0 && data == NULL;
2  // Invariant: data != NULL iff refcount > 0
3  struct FileSystemHandle {
4    unsigned refcount;
5    unsigned offset;
6    enum FileSystemFileMode mode;
7    enum FileSystemFileOptions opts;
8    struct FileSystemData* data;
9  };
10
11 struct FileSystemData {
12   enum FileSystemDataFlag flag;
13   char* path;
14
15   // When flag:
16   //  - File: the logical size of the file
17   //  - Directory: the number of contained items in the directory
18   unsigned contents_size;
19
20   // The following fields are only valid for symbolic files.
21   // invariants:
22   //  - contents_size <= buffer_size
23   //  - contents == NULL iff buffer_size = 0
24   unsigned buffer_size; // the real size of the malloced region
25   char* contents;
26
27   // The following fields are only valid for symbolic directories.
28   // invariants:
29   //  - contents_size == 0 iff subitems == NULL
30   struct FileSystemData** subitems;
31 };
32
33 // Represents the access mode of a file (as a bitmask).
34 enum FileSystemFileMode {
35   kFileSystemFileModeUndefined = 0,
36   kFileSystemFileModeRead      = (1 << 0),
37   kFileSystemFileModeWrite     = (1 << 1),
38   kFileSystemFileModeReadWrite = (kFileSystemFileModeRead|
         kFileSystemFileModeWrite)
39 };
40
41 enum FileSystemFileOptions {
42   kFileSystemFileOptionsNone     = 0,
43   kFileSystemFileOptionsNullSink = (1 << 0),
```

```
44   kFileSystemFileOptionsCreate   = (1 << 1)
45 };
46
47 enum FileSystemDataFlag {
48   kFileSystemDataFlagUnused,
49   kFileSystemDataFlagSymbolicFile,
50   kFileSystemDataFlagNativeFile,
51   kFileSystemDataFlagSymbolicDir,
52   kFileSystemDataFlagNativeDir,
53   kFileSystemDataFlagStdOutput,
54   kFileSystemDataFlagStdError
55 };
```

FileSystemHandle The structure contains a reference count field which keeps track of how many file descriptors are using the handle. This is needed so that we know when it is safe to re-use a handle. In addition, the handle keeps the current file offset. It also specifies the file mode – read, write or both so that we do not allow callers to write to a file that they have opened as read-only. It contains a set of options in the `opts` field – currently, one option is the null sink option and the other is flag to create a file if it does not exist already. We require the null sink option so that we can essentially nop operations on such handles but pretend that we have written any bytes (and always return EOF when reading). Finally, a handle contains a pointer to the data behind the handle (which can be `NULL` for special files like `stdin` and `/dev/null`).

FileSystemData This structure's two most important fields are `path` and `flag`. The path for any piece of data is stored so that all handles link to the same backing store – if a program opens the same file path and writes from one while reading from the other, the one reading should not see the "world" differently than the one that has been writing. The `flag` field tells us whether the structure is in use and if so, what kind of item it is. If the flag is one of the native ones, we know that when trying to open that data item, the native `open()` call must have succeeded. We also have two special flags for standard output and standard error – this is so that we can find them and persist them when the simulation ends. There are also two symbolic flags – one for a symbolic file and one for a symbolic directory. The rest of the fields' values are only defined when we have symbolic items. For example, when we have a symbolic directory, `subitems` points to the beginning of an array of pointers to file system data structures. The number of items in the array is specified by the field `contents_size`. For symbolic files, the data stored is pointed by `contents` and `buffer_size` specifies to size of the `malloc()`ed region – the reason why we need it, is because we usually allocate buffers slightly larger than the data the user wants to write, so that we do not `realloc()` on every write to a symbolic file.

OS File Descriptors One important aspect is that we do not keep a file descriptor around when the file is backed by the OS – instead, we `open()` the path when needed. The reason is that if a state has branched many times and all of them are trying to open the same file at some later point, we would end up with thousands of native open file descriptors to the same file – a wasteful use of file descriptors which can actually starve our system and other states depending on the amount of branching that is happening.

API The filesystem part provides 4 high-level functions that are used to implement the POSIX functions: `filesystem_handle_open` which handles all the complexities of opening both

symbolic and non-symbolic files & directories, `filesystem_handle_read`, `filesystem_handle_write` which are used to read and write data to files, respectively and finally `filesystem_handle_close` which closes a file system handle.

Duplication This particular choice of data structures makes it very easy to implement the required POSIX functions. For example, in order to implement file descriptor duplication (`dup()` and `dup2()`) all we have to do is increment the reference count for the relevant handle.

Listing 16: Implementing file descriptor duplication.

```
1  static int duplicate_descriptor(struct PosixDescriptor* descriptor) {
2    assert(descriptor != NULL && descriptor->file != NULL && "Invalid file
         descriptor");
3
4    struct PosixDescriptor* unused = posix_layer_find_unused_fd();
5    if(unused == NULL) {
6      errno = EMFILE;
7      return -1;
8    }
9
10   struct FileSystemHandle* handle = descriptor->file;
11   handle->refcount++;
12   unused->file = handle;
13
14   return unused->fd;
15 }
```

One of the relatively more complex functions is responsible for opening a path. To illustrate the function's increased complexity relative to the rest of the higher-level functions, the source is reproduced in Listing 17.

Lines 4-13 perform sanity checks for invalid parameters. On line 15, we try to find an unused handle – it is possible that we have run out of file handles if the user program opens too many files. In this case, there is nothing we can do and we return an error. Crucially, on line 22 we check whether the path name is concrete – we cannot pass symbolic data to the OS as the behaviour is undefined. `filesystem_handle_open_symbolic_file` will try to match the path name against a very limited set of pre-defined entities on the filesystem, thus avoiding the creation of a lot of processes. On line 26, we try to find if we already know about the path. Lines 27-31 deal with the case when the caller is trying to open a file for writing that it backed by OS. In order for processes not to be able to affect each other, we do not support this option at present time. The issue is covered in more detail in Evaluation (section 8). On the other hand, if the caller is not trying to write to a native file, we successfully setup an unused handle and associate it with the already existing data item (line 33).

If we reached line 36, that means that we need to create a new filesystem data item as one for the path name was not found. Line 44 calls a utility function that ensures that we can safely pass `pathname` to the OS – it does not cause any branching, despite what its name might suggest (furthermore, when we reach line 44, we know the path name does not contain any symbolic characters due to the test on line 22). On line 47, we bypass our system and perform a direct system call to the OS, trying to open the path name. Lines 49-55 handle the case where we do not have a backing OS entry – we create a symbolic in-memory file if the creation flag has been specified.

If the OS file exists, we continue on line 59 where we have to perform an `fstat()` to be able to tell whether the entry is a directory or a file. An implementation detail is that the `fstat()` system call operates on a different structure than the library function `fstat()`. Lines 70-72

again handle the case with writing to OS backed files. Finally, lines 77-80 handle the cases for opening a directory or a file. The reason for the code on lines 82-83 is because the user can try to open special files, like `/proc/cpuinfo` and others, which are unsafe because we do not model their behaviour correctly.

Listing 17: Implementing file descriptor duplication.

```
1  struct FileSystemHandle* filesystem_handle_open(const char* pathname, enum
       FileSystemFileMode mode, enum FileSystemFileOptions opts) {
2    assert(pathname != NULL);
3
4    if(mode == kFileSystemFileModeUndefined) {
5      errno = EINVAL;
6      return NULL;
7    }
8
9    unsigned pathlen = strlen(pathname);
10   if(pathlen == 0) {
11     errno = EINVAL;
12     return NULL;
13   }
14
15   struct FileSystemHandle* unusedHandle = filesystem_find_unused_handle();
16   if(unusedHandle == NULL) {
17     errno = EINVAL;
18     return NULL;
19   }
20
21     // firstly, we need to check if the pathname is symbolic
22   if(!klee_is_concrete(pathname, pathlen))
23     return filesystem_handle_open_symbolic_file(pathname, pathlen, unusedHandle,
           mode, opts);
24
25   // pathname is concrete, so let's try to find the data already
26   struct FileSystemData* existingData = filesystem_find_existing_data_for_path(
       pathname);
27   if(existingData) {
28     if(existingData->flag == kFileSystemDataFlagNativeFile && (mode &
         kFileSystemFileModeWrite)) {
29       assert(0 && "Unsupported writing to native files");
30       return NULL;
31     }
32
33     return filesystem_setup_handle(unusedHandle, mode, opts, existingData);
34   }
35
36   struct FileSystemData* unusedData = filesystem_find_unused_data();
37   if(unusedData == NULL) {
38     errno = EINVAL;
39     return NULL;
40   }
41
42   // concrete pathname with no existing data, we will go to the OS
43   // it's concrete, we need to go to the OS and check if the file is there
44   pathname = __concretize_string(pathname);
45   int native_flags = O_RDONLY;
46   mode_t native_mode = 0;
47   int os_fd = syscall(__NR_open, pathname, native_flags, native_mode);
48
49   if(os_fd == -1) {
```

```
50      // there's no OS file, we only allow to proceed if we have the O_CREAT flag
51      if(opts & kFileSystemFileOptionsCreate)
52        return filesystem_setup_handle_with_data(unusedHandle, mode, opts,
              unusedData, kFileSystemDataFlagSymbolicFile, pathname, pathlen);
53
54      // OS doesn't exist but they want to open, so no-go
55      return NULL;
56    }
57
58    // fstat and close the file
59    struct kernel_stat st;
60    int fstat_retval = syscall(__NR_fstat, os_fd, &st);
61    syscall(__NR_close, os_fd);
62    if(fstat_retval == -1) {
63      // some error occured, we can't proceed
64      errno = EIO;
65      return NULL;
66    }
67
68
69    // OS-backed file exist, we can only proceed if they don't want to write to it
70    if(mode & kFileSystemFileModeWrite) {
71      assert(0 && "Unsupported writing to OS-backed files");
72      return NULL;
73    }
74
75    // it's possible that they actually opened a directory, so we need to check
76    enum FileSystemDataFlag dataFlag;
77    if(S_ISDIR(st.st_mode))
78      dataFlag = kFileSystemDataFlagNativeDir;
79    else if(S_ISREG(st.st_mode))
80      dataFlag = kFileSystemDataFlagNativeFile;
81    else {
82      errno = EIO;
83      return NULL;
84    }
85
86    return filesystem_setup_handle_with_data(unusedHandle, mode, opts, unusedData,
          dataFlag, pathname, pathlen);
87  }
```

7.3.4 Failure Model

The failure model's role in our system is to artificially introduce failures of system calls. The user specifies various failure options on the command line that get parsed and values get assigned to the failure model structure. Its definition is shown in Listing 18. There are several notes of interest.

enum FailureFlags The flags defined in the enumeration will be stored in the `flags` field of `struct Failures`. The first two flags exists so that whenever we inject a failure for `open()` or `chdir()`, we also explore different return codes. Even though this increases the number of states, it also allows us to cover code which otherwise would not be covered – some callers take a different action depending on the particular value of `errno`. For example, if `open()` returns `ENOENT` (does not exist), then a web server would return a 404 Not Found while if `open()` returns `EACCES` (permission denied), the client will receive 403 Forbidden. The last flag provides the ability for `fstat()` to return a date that is far back in the past and also a

date that is far into the future. The reason for not completely marking the dates as symbolic is because in preliminary testing, code that analyses timestamps caused the constraint engine to be stuck at 100% usage for long periods of time without making any progress. On the other hand, providing the dates illustrated above resulted in covering all code paths that depended on their values.

socket_block_interval field This field can be used to set an interval of how often `write()` would return `EWOULDBLOCK` – this error is returned whenever the OS buffer is filled up. As we do not define the capacity of any such buffers, this option is the only way to induce a `write()` failure with that error.

select_fail_interval field Similar to the field above, this defines an interval of how often `select()` would just return a failure.

Listing 18: Failure model structure.

```
1  enum FailureFlags {
2    // whether open() should fail in multiple ways (diff errno)
3    kFailureFlagsOpenMultipleErrno  = (1 << 0),
4    kFailureFlagsChdirMultipleErrno = (1 << 1),
5    kFailureFlagsFstatSymTimes      = (1 << 2),
6  };
7
8  struct Failures {
9    // various failure options
10   unsigned flags;
11   // write() will block every X times (0 for no blocking)
12   unsigned socket_block_interval;
13   // select() will fail every X times (0 for no failures)
14   unsigned select_fail_interval;
15   // total across all calls
16   unsigned maximum_failures;
17 };
```

In addition, any places that need to simulate failure will have to use a symbolic variable as a branch condition so that both paths are explored. This is illustrated in the code snippet below.

Listing 19: Injecting failure.

```
1 int fail = klee_int("open:fail");
2 if(posix_layer.failures.maximum_failures && fail) {
3   --posix_layer.failures.maximum_failures;
4   errno = EIO;
5   return -1;
6 }
```

An important aspect that needs highlighting is the fact that the number of maximum failures is decremented only along the path which failed the system call. This is because, by definition, the user specifies the number of maximum failures that can occur along a path of execution. If the value was decremented beforehand, then exploring failures would only be *attempted* that many times – it would mean that in a sequence of two possibly failing system calls and a maximum failure of one, the scenario of the second one failure would never be explored.

Finally, the order of the conditions is important as well – if we checked the symbolic variable first, we would be unnecessarily branching every single time the code is encountered, even if the maximum limit has been reached.

7.3.5 Runtime Structures

Finally, we statically allocate structures for sockets, files and descriptors. There are also various utility functions provided to deal with file descriptors. Some of them include `posix_layer_find_socket_fo` `posix_layer_find_file_handle_for_fd`, `posix_layer_find_unused_fd` which are used by the implementations of the POSIX functions. The runtime structure is defined as:

Listing 20: Runtime data structure.

```
1  struct PosixLayer {
2    struct NetworkConfig net_config;
3    struct Failures failures;
4    struct ProcessInfo info;
5    struct PosixDescriptor descriptors[MAX_KLEE_FDS];
6    struct NetStackSocket sockets[MAX_KLEE_SOCKETS];
7    struct FileSystemHandle handles[MAX_KLEE_FILE_HANDLES];
8    struct FileSystemData files[MAX_KLEE_FILES];
9    struct FileSystemData symFiles[MAX_KLEE_SYM_FILES];
10 };
11
12 struct PosixDescriptor {
13   int fd;
14   struct NetStackSocket* socket;
15   struct FileSystemHandle* file;
16 };
17
18 struct ProcessInfo {
19   mode_t umask;
20   uid_t uid;
21   gid_t gid;
22   char* cwd;
23 };
24
25 struct NetworkConfig {
26   // Packer Ordering
27   size_t reorder_count, reorder_window_size;
28   // Packet Loss
29   size_t lost_packet_count;
30   // Packet Corruption
31   size_t corrupt_packet_count;
32   size_t corrupt_data_offset, corrupt_data_size;
33 };
```

In order to illustrate the simplicity of implementation of most high level POSIX APIs, Listing 21 shows the complete code for `write()`. Firstly, on line 2 we try to find an entry for the file descriptor that was passed. If no valid entry was found, an error is returned (lines 3-5). Lines 8-12 can be ignored as they are related to the failure model and only inject a failure. Finally, lines 15-18 appropriately route the request either to `send()` if it is a socket or to our internal filesystem if it is a file.

Listing 21: Implementation of write()

```
1 ssize_t write(int fd, const void *buf, size_t count) {
2   struct PosixDescriptor* descriptor = posix_layer_find_descriptor_for_fd(fd);
3   if(descriptor == NULL) {
4     errno = EBADF;
5     return -1;
6   }
7
```

```
 8   int fail = klee_int("write:fail");
 9   if(posix_layer.failures.maximum_failures && fail) {
10     --posix_layer.failures.maximum_failures;
11     errno = EIO;
12     return -1;
13   }
14
15   if(descriptor->socket)
16     return send(fd, buf, count, 0);
17   else if(descriptor->file)
18     return filesystem_handle_write(descriptor->file, buf, count);
19
20   errno = EBADF;
21   return -1;
22 }
```

Finally, during initialisation of the simulation, our system automatically inserts a function call to our own initialisation function in the user defined `main()` function. The initialisation function has two roles: to initialise all data structures shown in Listing 20 with appropriate initial values (this includes setting up `stdin`, `stdout`, etc.) and to parse the command line arguments for any options – any command line switches that it recognised, are removed which the user `main()` function would not receive. Listing 22 shows the code for the function. Line 1 initialises all structures defined in 20. We implemented our own way of specifying arguments, as shown by lines 4-10. The function call on line 13 parses the argument specification passed and appropriately removes any recognised arguments.

Listing 22: Initialisation of the runtime environment.

```
1  void klee_init_env(int* argcPtr, char*** argvPtr) {
2    posix_layer_init();
3
4    struct Argument args[6];
5    UNSIGNED_ARG(args, 0, "--max-sys-fail", &posix_layer.failures.maximum_failures
         );
6    FLAG_ARG(args, 1, "--open-errno", 0, &posix_layer.failures.flags);
7    UNSIGNED_ARG(args, 2, "--socket-block-interval", &posix_layer.failures.
         socket_block_interval);
8    FLAG_ARG(args, 3, "--chdir-errno", 1, &posix_layer.failures.flags);
9    UNSIGNED_ARG(args, 4, "--select-fail-interval", &posix_layer.failures.
         select_fail_interval);
10   FLAG_ARG(args, 5, "--fstat-sym-time", 2, &posix_layer.failures.flags);
11
12   int count = sizeof(args) / sizeof(args[0]);
13   parse_arguments(args, count, argcPtr, argvPtr);
14 }
```

7.4 Summary

Starting with the system itself, we described the necessary changes we had to make to accommodate a modified world model. We also took a look at how the event system works and reviewed the algorithm which ensures the preservation of world separation semantics. Afterwards, an efficient implementation of a process scheduler was presented. We also touched on the topic of how the invariants framework works which lets processes expose data for verification purposes.

After covering the changes to the system, we turned our attention to the runtime, which gets loaded during initialisation. In particular, we provided a review of the special primitive functions

that were necessary to implement the POSIX APIs. Subsequently, we looked into the specifics of implementing the sockets APIs while providing implementations of `accept()`, `connect()` and `select()` amongst others. We reviewed how the filesystem works and provided the details behind the most important function which handles the opening of paths. Then we showed how easy it is to inject failures into system calls. Lastly, we revealed how the POSIX data structures get initialised at the beginning and how the various runtime subsystems fit together.

8 Evaluation

One of the most important aspects in the life cycle of any project is evaluation. It is important to appraise the performance of the system and to take a step back in order to reflect on the lessons learned. In the rest of this section, we will take a look at both real-world scenarios as well synthetic ones. We will also cover any limitations that our system possesses and reflect on their effects on the overall utility of the software.

8.1 Goals and Methodology

In this section, we try quantify the following metrics:

- **Performance on real-world software**

  One of the major goals of this project was to investigate the ability to find bugs and improve the quality of software. Thus it is very important to try to evaluate its performance on a piece of software that is being used in a production environment.

  The main reason is that non-synthetic software usually has a much higher complexity and consequently will push the bounds a lot further. It will also provide us with a set of core functionality that is very likely to be used across all distributed software.

- **Capabilities of the invariants framework**

  We have seen an exponential growth in network-connected software and it is an absolute certainty that its proportion will only be increasing. Unfortunately, current software systems have grown significantly more complex over the past decade, especially in terms of internal states. It is very important to be able to analyse how systems interact with each other, as in certain cases correctness cannot be evaluated only based on local information.

  The invariants framework tries to solve that problem by providing the ability to write invariants over the global "world" state. It makes it possible to catch logical errors – even though systems might not crash, they can still have faults with grave consequences.

- **Class of bugs that can be discovered**

  Computer networks provide a certain set of guarantees about delivery of packets, data corruption detection and various other facilities. In most cases, networks behave without many errors and the code paths that result from failures are usually not thoroughly tested – one of the main contributing factors is the lack of ease in setting up a testing environment that simulates those failures.

  Furthermore, the number of failure combinations is so high that it is virtually impossible to be absolutely certain that certain cases would not have been missed. Symbolically executing networked software and injecting such failures provides an easy way to explore the behaviour of the code under such circumstances, and, hopefully reveal issues.

- **Synthetic performance**

  Another aspect that we want to cover is the absolute limits of our system. Pushing the system to its absolute boundaries until it breaks down gives us an indication of how practical it would be if we wanted to use it to test other pieces of software.

Lastly, we will cover any limitations of the system, both inherited from KLEE and any additional ones.

8.1.1 Test Configurations

We provide all the necessary details to reproduce the tests described in this section in Appendix A.

8.2 Boa Web Server

In order to evaluate the real-world performance of our system, we used the Boa web server. Some of the main reasons it was chosen for evaluation were:

- **Production Use**

  As previously outlined, it is essential to evaluate the performance of the system on real-world software. Boa is most notably used by two very large websites to serve images – Slashdot[3] and Fotolog[4]. Fotolog on its own has more 30 million registered users.

- **Stability**

  The Boa web server is an old piece of software – initially written in 1995, it has had some 16 years to mature. Given that it is a minimalistic web server, we would expect it to be virtually bug free. If some bugs were to be found, it would serve as a good indication that symbolically executing other distributed software should yield results.

- **Simplicity**

  Furthermore, its minimalistic feature set has a knock-on effect on the code – it is simple and easy to understand. This has proven very useful in allowing us to more carefully understand the performance of our system.

- **Written in C**

  Crucially, it is only written in C (and using the POSIX standard APIs) which means that we can easily run it under KLEE.

- **Web Server**

  Even though it is a minimalistic and small server, it actually depends on a very large amount of system APIs to work correctly. For example, it deals with the filesystem, the network and the OS. This means that in order for us to even process a simple `GET` request, all of those APIs have to be correctly implemented by our software such that they honour the semantics as mandated by the standard. In essence, the POSIX API utilisation footprint of a web server is very high.

- **Symbolic Data**

  Testing a web server allows us to easily quantify how symbolic input affects code coverage – the behaviour of a web server is, by and large, dependent on the requests it receives. Thus, we can measure the effectiveness of our system by varying the amount of symbolic data in HTTP requests.

[3] http://slashdot.org
[4] http://fotolog.com

8.2.1 Code Coverage

There are a variety of ways to measure code coverage and consequently it is essential to specify exactly what is being measured and how.

The way code coverage is generally measured when testing software with KLEE is as follows:

- Test the program using KLEE which would generate test files.
- Recompile the software under test with gcov support.
- Run the software natively guided by the generated tests.
- Analyse the coverage as reported by gcov.

Instead of following the above method, we take a different approach. We will be measuring the code coverage of the LLVM bitcode that KLEE runs symbolically. The reasons for taking the aforementioned approach include:

- **Replay Framework**

  Due to a limitation of the distributed replay framework, our system will currently generate test cases that cannot be replayed by running the software under test natively. We will provide further details about this limitation at the end of this section.

- **Selective Coverage**

  Due to limitations in KLEE, such as the inability to use the `fork()` system call, there are parts of the codebase that are impossible for us to test. We need to able to specify exactly which functions are to be counted towards the percentage.

  We have modified KLEE to accept a list of functions that represent the functions that we are interested in and it will only track their coverage.

- **Multiple Runs**

  During preliminary testing of the system performance, we noticed an explosion of states without any increase of coverage. The reason was that the ratio of useful `GET` requests to malformed was excruciatingly small – the system was spending a lot of time covering the same code over and over again. It became obvious that we need to concentrate on running multiple tests, each exercising a different combination of symbolic characters and then combining the coverage. We do this by recording which instructions in the relevant functions have been executed and then running a tool to aggregate all the information.

Code Size Boa's number of source code lines across its implementation files amounts to 7021 lines (according to `sloccount`[14]). In terms of LLVM bitcode instructions, after optimisation, this amounts to 5669 instructions. Of those 5669, we track the coverage for 4761 of them as only those can in theory be reached by our system. The reasons for not being able to test approximately 16% of the instructions are covered in Section 8.2.5.

8.2.2 Evaluation Tests

There are four main questions that we set ourselves when evaluating the system performance:

1. What code coverage percentage can be achieved by making non-symbolic HTTP requests?
2. How does making parts of the HTTP request symbolic affect the code coverage?

3. What are the costs of making parts symbolics and when does the system break down?

4. What is the effect of the additional facilities provided (failure injection, etc.) on the performance and coverage?

In order to systematically explore those scenarios, we created a set of test suites, running a total of 112 different configurations:

- **Non-Symbolic** A set of 29 test configurations which exercise the web server without any symbolic data.

- **Symbolic** A set of 50 test configurations with varying amounts of symbolic characters in a `GET` request.

- **Constrained Symbolic** A set of 31 test configurations which turn on an option to make the symbolic characters in a `GET` request to be only printable ASCII characters (in the range 32 up to 127, inclusive).

- **Failure Injected** A set of 2 test configurations with failure injection turned on.

In order to gather accurate code coverage percentages, all Boa tests were run optimised with inlining disabled, because library functions get inlined into the Boa code which lowers the accuracy of the code coverage that we are interested in. Unless otherwise noted, we report the aggregate code coverage, that is, across multiple runs of our system as our test suites are composed of multiple samples.

The test suites were run on Ubuntu 10.10 running in VMware Fusion on a Xeon X5500 (Nehalem-based) clocked at 2.27GHz.

8.2.3 Bugs Found

During our testing, we found 2 critical bugs that result in the web server being rendered inoperative. Both bugs are related to the support of directories.

- **Bug 1**

  This bug results in a crash of the web server by trying to dereference a `NULL` pointer. It requires 3 conditions to be met:

  1. The configuration file should not contain a name for a default directory index.
  2. The configuration file should contain a directory list cache path.
  3. The client must send a `GET` request to a directory path.

  This results in a `strcpy` with a `NULL` second argument causing a segmentation fault. The bug was discovered by trying out various configurations and sending a symbolic requests to the web server – one particular code path resulted in requesting a directory.

- **Bug 2**

  During the initialisation stage of the web server, it reads a configuration file which specifies the root directory and makes it the current working directory via `chdir()`. During the processing of a `GET` request for a directory, the code temporarily changes the current working directory to the directory being requested and restores it at the end. If a system call fails while reading the contents of the directory, the function returns early and does not restore the working directory to the web root. This results in the server failing to properly serve any further requests.

The bug was discovered by turning on failure injection into system calls. The high-level code flow which performs directory indexing is shown in Listing 23.

Listing 23: Generating a directory index.

```
1  static int index_directory(request * req, char *dest_filename)
2  {
3      if (chdir(req->pathname) == -1) {
4          ...
5          return -1;
6      }
7
8      request_dir = opendir(".");
9      if (request_dir == NULL) {
10         // early return without restoring server root
11         return -1;
12     }
13
14     fdstream = fopen(dest_filename, "w");
15     if (fdstream == NULL) {
16         // early return without restoring server root
17         return -1;
18     }
19
20     ...
21
22     closedir(request_dir);
23     chdir(server_root);
24
25     return 0;                    /* success */
26 }
```

8.2.4 Results

We have split the experiments into 4 parts which try to identify various performance characteristics of the system:

- **No Symbolic Data** In these tests, our goal was to explore the behaviour of the system without any symbolic data.
- **Symbolic Data** In these tests, we investigate what effects symbolic data has on various aspects of the system, namely code coverage, time, number of processes and memory usage.
- **Symbolic Constraining** In this part we investigate how constraining the symbolic input affects the code coverage and resource usage.
- **Failure Injection** Finally, we look at system call failure injection and the ramifications of its use.

No Symbolic Data We handcrafted a set of 29 HTTP (section A.1.1) requests that were designed to maximise the code coverage. We managed to a achieve an aggregate code coverage, on the functions that are theoretically testable, of 65.01%. Turning on failure injection of maximum 1 failure increases that number to 67.93%.

The specific HTTP requests were selected incrementally in such a way as to cover all reachable code paths only by varying the HTTP request. At each step, we would analyse the results

of the previous runs and increase the set of requests so that we force Boa to follow the remaining unexplored code paths. We look at the reasons why we can only achieve such a seemingly low-number in Section 8.2.5.

There are two very important consequences of this test result. Firstly, it provides us with an upper bound on the coverage for **any** HTTP request that can be constructed as we ensured maximum possible coverage by only varying the requests. Secondly, it also provides us with upper bounds for certain types of requests by only selecting the subset that matches the types of requests.

Symbolic Data In this test, we try to quantify the effects of symbolic data on the code coverage and its associated costs. We chose one `GET` request that is capable of covering multiple code paths if we made parts of it symbolic. The characters that were potential candidates (a total of 11) for being marked as symbolic are underlined – GET /sym1.html HTTP/1.1\r\n\r\n.

A **very important** fact that needs to be highlighted is that this request can only ever succeeds in two ways – when the URI is `/sym1.html` or `/sym2.html`. Any other combinations would make the URI either malformed or it would request a non-existent file. The reason for choosing those specific positions is that the requests will internally fail in different ways depending on the position of the malformed character. For example, marking the first character as symbolic would produce failures due to an unrecognised HTTP method while marking the HTTP version as symbolic would result in the code which checks the HTTP version to return an error. From the point of view of the client, the invalid position of the character does not make a difference, although the different ways to fail should manifest themselves as higher code coverage.

Upper Bound We can compute an approximate upper bound for a simple `GET` request (i.e., without any request options) by running all simple `GET` requests from our suite of handcrafted 29 – it contains a total of 16 such requests. The aggregate code coverage of those 16 requests is **51.46**% – **more importantly**, this provides an **approximate upper bound** for any instantiation of the symbolic characters in the request. This is the case because any instantiation of the symbolic characters would either be a malformed request or a valid one for one of the two files (`sym1.html` and `sym2.html`) and the 16 requests explore all possible ways for a simple `GET` request to fail and it also include requests to both `sym1.html` and `sym2.html` (requests #19, #20 and #21 in section A.1.1).

Methodology We performed tests to explore the system behaviour with 0, 1, 2 and 3 symbolic characters. The natural question arises how do we choose which positions to mark as symbolic – for 0 and 1, we can afford to test all combinations (for 0 we have only 1 possible combination, i.e., the get request without any symbolic data and for 1 symbolic characters we can try all 11 positions). For 3 symbolic characters, we have 165 combinations ($\binom{11}{3}$). Testing all of them instead of a random sample cannot significantly decrease our results due to a property of our test scenario – increasing the number of symbolic inputs (1 versus 2 versus 3 etc.) can only increase aggregate coverage by very small amounts.

It is **crucial** to realise why that is indeed the case – we can think of increasing the symbolic input size as increasing the number of requests sent. For example, if we made the first character symbolic, we can think of it as testing the behaviour of the server under 256 `GET` requests. 255 of them would be malformed and 1 would result in a `200 OK` response. Now, if we made the `T` in HTTP symbolic, we can think of it as sending 65536 different requests – of those, exactly one will return `200 OK`, 255 would start with a `G` but not have a `T` in HTTP, another 255 would have a `T` but not start with a `G` and the rest (65025) would not start with a `G` and not have

a `T` in H<u>T</u>TP. It is **essential** to realise that in terms of code paths, the case with 2 symbolic characters would explore 4 logical cases: succeeding, failing due to the `G`, failing due to the `T` and failing due to both the `G` and the `T`. But since we run **all** possible combinations for 1 symbolic character, it will cover all code cases **except** the last one (remember that we care about **aggregate coverage**). But in practice, in Boa there is not much code that has different behaviour if there are 2 unexpected characters instead of 1 – most functions would return a failure as soon as they detect an error, as the presence of more errors, does not generally, alter the outcome.

We can summarise the above as: **the <u>aggregate</u> coverage can only increase for configurations with more symbolic characters if, and only if, the additional degree of freedom enables the execution of code paths that were previously impossible to reach**. When it comes to Boa and the specific `GET` request, there is exactly one such case when we consider the case from going from 1 symbolic character to 2 symbolic characters – the reason why we know about it is because as part of handcrafting the 29 non-symbolic requests, we acquired intimate knowledge of how to exercise all possible code paths. The case arrises when requesting an empty file since the response depends on whether the HTTP version is 1.1 or 1.0 – so we require at least 2 symbolic characters in order to have the URI be `/sym2.html` and the HTTP version be `1.0`, as this requires two changes to the `GET` request that we chose for our test.

In summary, we choose 20 random combinations to form our samples for symbolic inputs of sizes 2 and 3. We will firstly analyse the results of the coverage, as it the focal point of those tests and then take a look at the resources used.

Code Coverage We present the code coverage results below. We report both aggregate and maximum metrics.

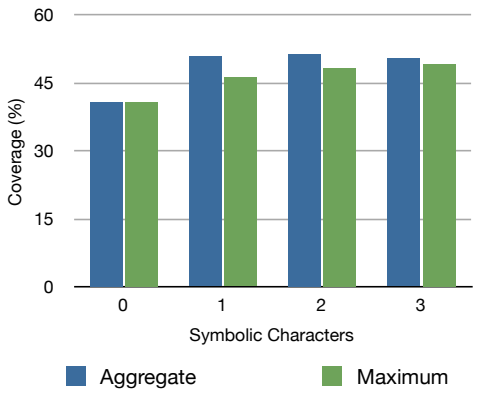


Figure 30: Showing what effect marking parts of a GET request as symbolic has on code coverage.

Sym Chars	Agg (%)	Max (%)
0	40.83	40.83
1	50.89	46.19
2	51.38	48.27
3	50.39	49.13

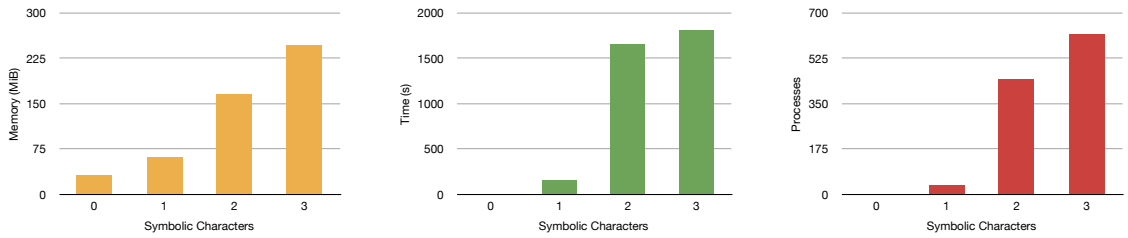
Figure 31: Showing what effect marking parts of a GET request as symbolic has on code coverage.

From Figure 31, we can see that making a single character symbolic has the greatest effect on the code coverage – the aggregate coverage increases by 24.6% between 0 and 1 symbolic characters. Most of the aggregate coverage increase would be due to different ways to fail the request. The aggregate coverage increases slightly between 1 and 2 characters – this means that there is some code path that cannot be traversed using only 1 symbolic character. When we increase the number of characters to 3, we see a decrease in aggregate coverage – this can be caused due to our sampling, as we do not cover all possible 165 combinations for 3 characters.

Another point of interest is the increase of the coverage each test configuration achieves on its own (Max column) – it confirms the intuition that more symbolic characters should result in exploring more code paths in a single run.

Note the absolute difference between the maximum and aggregate metrics for 1, 2 and 3 symbolic characters (4.7%, 3.11% and 1.26%) – the more symbolic characters we have, the more degrees of freedom each single test configuration has and consequently a single run can explore many more code paths on its own when compared to the aggregate of all runs. This is exactly what we would expect from increasing the symbolic input size and comparing it to the aggregate.

Most importantly, using 2 symbolic characters, we achieved a code coverage of 51.38% which represents **99.8%** of our approximate upper bound of 51.46% computed on page 85. Next, we look at the resource usage profile of our system for the different number of characters. When it comes to resource usage, we are interested in worst-case numbers, so the following graphs and tables show the maximum values across all runs.



(a) Memory Usage (b) Run Time (c) Processes

Figure 32: Showing what effect marking parts of a GET request as symbolic has on resource usage.

Sym Chars	Memory (MiB)
0	31.46
1	61.21
2	165.16
3	246.65

(a) Maximum Memory Usage

Sym Chars	Time (s)
0	1.16
1	157.79
2	1658.44
3	1813.91

(b) Maximum Run Time

Sym Chars	Processes
0	2
1	35
2	442
3	618

(c) Maximum Processes

Figure 33: Showing what effect marking parts of a GET request as symbolic has on resource usage.

From the data, we can see that increasing the number of symbolic characters leads to linear increases in memory usage and number of processes. On the other hand, the run time does not increase significantly once we have 2 or more symbolic characters (an increase of 9.4% when going from 2 to 3 characters compared to an increase of 951% between 1 and 2 characters). The reason we identified for the heavy branching (as exemplified by Figure 32c) was due to the unconstrained nature of the symbolic input. The issue is addressed on page 88.

Fully Symbolic GET Request Before moving on to constraining the input, we run two tests by marking the whole `GET` request as symbolic and constraining the maximum amount of memory to 1.4GiB. In the first case, our tool run for approximately 1hr28mins and achieved a coverage of 50.45%. The second run lasted 2hr34mins and achieved a coverage of 49.21%. This represents 98% and 95.6% of our approximate upper bound. It should be noted that those coverage values were achieved within about 40minutes of running. The code coverage barely changed for the rest of time. Additionally, this coverage is for each run (not aggregated) and it is higher than any of the individual maximum coverage values for up to 3 characters.

We believe that with a more advanced scheduler, a fully marked `GET` request can achieve our approximate upper bound. The reason why we **cannot** expect fully marking our example request to achieve a much higher coverage lies in the fact the specific length of our HTTP request allows a very small number of valid requests. For example, it is impossible to test the behaviour of the `If-Modified` option header as there is no valid HTTP request of the given size that also contains the aforementioned header.

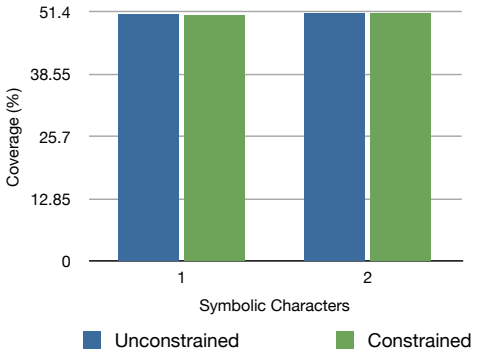
Symbolic Constraining While investigating the behaviour of Boa when run with symbolic data, we noticed an abnormal amount of branching even when just the first symbolic character was marked as symbolic. Upon further investigation, we stumbled upon a `while` loop that performed the following check for every single character in the request header.

Listing 24: Checking the request for invalid characters.

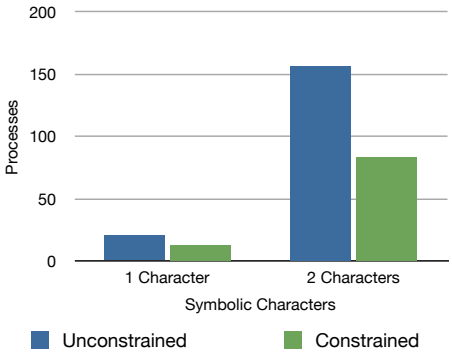
```
1  while (check < (buffer + bytes)) {
2      /* check for illegal characters here
3      * Anything except CR, LF, and US-ASCII - control is legal
4      * We accept tab but don't do anything special with it.
5      */
6      if (uc != '\r' && uc != '\n' && uc != '\t' && (uc < 32 || uc > 127)) {
7      ...
8      send_r_bad_request(req);
9      return 0;
10     }
11 }
```

With an unconstrained symbolic character, the check will add an additional 5 branches (since the expression can be true in 1 way and false in 4 ways). More importantly, all branches except 1 for which the if statement evaluates to false will lead to the sending of the exact same reply (`send_r_bad_request`) because `\rET`, `\nET`, `\tET` and x`ET` where x ! = G are all malformed initial character sequences.

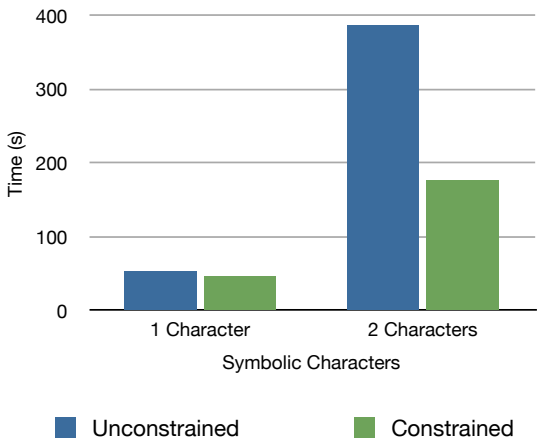
To understand the impact of constraining the characters, we run our samples with 1 and 2 symbolic characters again but instead constrained the symbolic character to be a printable character (i.e., between 32 and 127). In the graphs and tables below, we report the aggregate code coverage and the mean values for the number of processes and time taken.



(a) Aggregate Code Coverage



(b) Processes



(c) Time

Figure 34: Showing the effect of constraining symbolic characters to be in the printable character range.

Symbolic Chars	Unconstrained Coverage (%)	Constrained Coverage (%)
1	50.89	50.64
2	51.38	51.12

Figure 35: Showing how code coverage changes when constraining the symbolic input.

Symbolic Chars	Unconstrained Processes	Constrained Processes
1	20.5	12.8
2	156.7	83.3

Figure 36: Showing how the number of processes changes when constraining the symbolic input.

Symbolic Chars	Unconstrained Time (s)	Constrained Time (s)
1	53.2	47.2
2	387.0	176.8

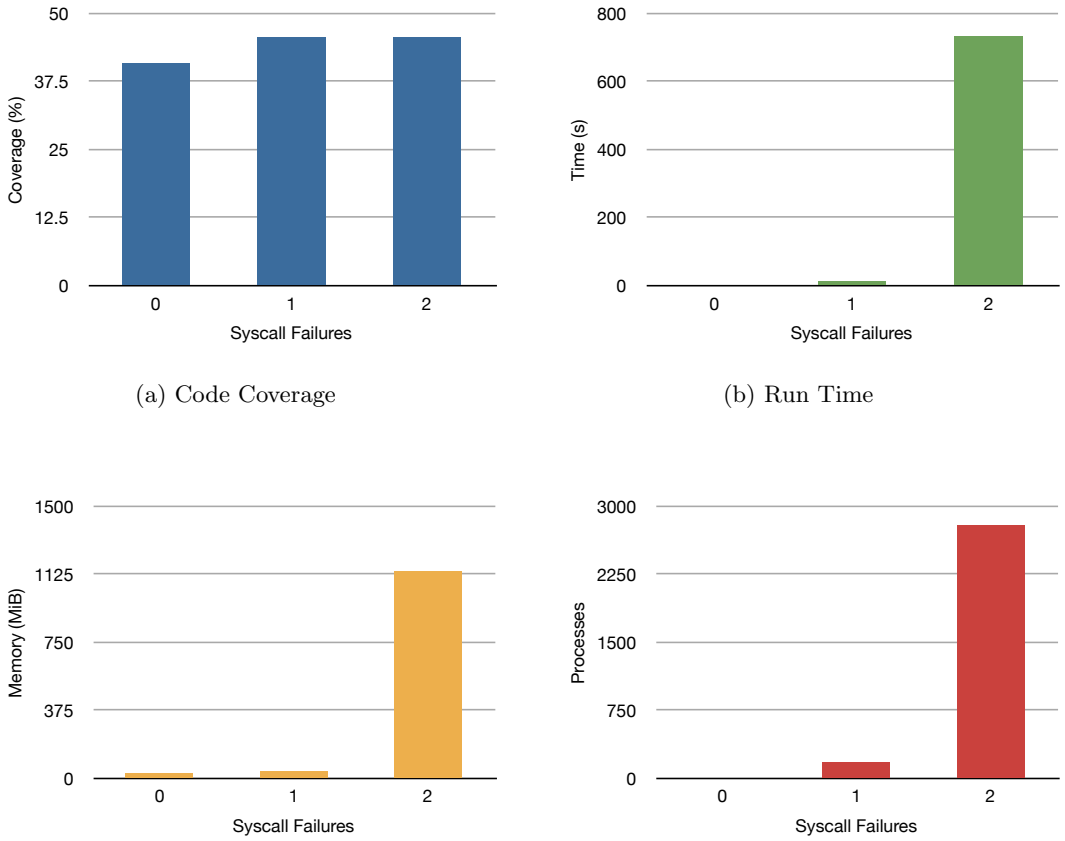
Figure 37: Showing how the time changes when constraining the symbolic input.

As can be seen, constraining the value provides great resource savings at virtually no expense in the aggregate coverage – it decreased, on average, by 0.5% while the number of processes decreased, on average, by 42.2% and time, on average, by 32.8%. The only caveat of using symbolic constraining is the need for domain-specific knowledge of exactly how to constrain the symbolic network packets.

Failure Injection Injecting failure into system calls is a way to explore code paths that would not be otherwise reached during normal execution. Failure injection works by exploring the case where a system call succeeds and also fails. Interestingly, there are cases where the error (as specified by `errno`) matters – not only do we have to explore the failure of the `open()` system calls but if we want to make Boa return an HTTP code 403 (Access Forbidden), we have to set `errno` to `EACCES` while on the other hand, returning `ENOENT` would result in a 404 (Not Found).

As previously explained, the user can specify the maximum number of system call failures along each execution path. This creates an exponential increase of states depending on the number of system calls made. In our testing, failing 2 system calls resulted in significant increases of memory and CPU usage – it is easy to see why. Assume that there are 30 system calls made during the run. If we allow up to 2 failures, that means that we will explore both paths having 1 failure, 2 failures or no failures. Thus we will have $\binom{30}{2} + \binom{30}{1} = 465$ branches. It should be noted that 30 system calls is completely unrealistic for a piece of software like a web server and that the above calculation is only for a single state – not including multiple nodes in the network that we are simulating.

In order to get a feel of the real world performance of system failures, we run the same request used in our previous tests with system call failures set to 1 and subsequently to 2.



(a) Code Coverage

(b) Run Time

(c) Memory Usage

(d) Processes

Figure 38: Showing what effect injecting system call failures has on code coverage in 38a, on run time in 38b, on the memory used in 38c and the number of processes in 38d.

Failures	Code Coverage (%)	Memory (MiB)	Processes	Time (s)
0	40.83	31.5	2	1.2
1	45.66	38.4	177	11.6
2	45.7	1143.9	2792	732.9

Figure 39: Showing metrics for different amounts of system call failures.

The staggering increase in the resources used when failing 2 system calls is clearly evident – memory increases by 2879%, processes by 1477% and time by 6218%. It is also reasonable if we consider what the program is doing – if a system call fails during the processing of an HTTP request, it will return an error to the client. The only additional code coverage that can be expected from failing more than 1 system call is code whose behaviour is dependent on multiple failures. In Boa, the increase is roughly 0.04% which would imply somewhere between 1 or 2 LLVM instructions. On the other hand, injecting 1 system call failure increases the coverage by 11.2% compared with no automatically induced failures. The costs of 1 system call failure are reasonable for the amount of coverage increase. In conclusion, injecting 2 or more failures would generally result in no increase of the final code coverage by any significant amounts and would be very costly in terms of system resources. On the other hand, injecting a single system

call failure can provide substantial increases – even more so in software that deals with a lot of error conditions.

8.2.5 Untestable Code

Even though Boa has a minimal feature set, there were parts of the code base that we did not test. We can split the untestable parts into two broad categories depending on the reason why there were untested:

1. Impossible to test due to inherent limitations in the our system (mainly un-modelled / unsupported APIs)
2. Code that we could not reach by varying the HTTP request

The list below provides an overview of the parts that are impossible to test due to limitations. Unless otherwise noted, the limitations can be overcome by implementing support for missing features by extending KLEE.

- **CGI / Scripts**

  In order to execute CGI scripts, the web server depends on the availability of the `fork()` system call which is currently unsupported.

- **Process Signals**

  KLEE's model assumes the simulation of a single process and consequently, process signals are unsupported.

- **Memory-mapped File I/O**

  Boa has multiple ways to read files that it needs to serve and one of the fastest ways is `mmap()`ing them. Currently, there is no support for the `mmap()` system call. It should be noted that implementing support would require changes in both the runtime and the system itself. The reason is that, usually, the runtime handles the filesystem but in order to efficiently implement `mmap()`, it would need to able to trap any memory reads / writes – and the "kernel" in this case is software itself.

- **Process Pipes**

  This is closely related to `fork()`ing processes and connecting their standard streams. This functionality is required by Boa to execute CGI scripts.

- **OS User Interaction**

  The web server needs the ability to find out the user groups during initialisation and give up its privileges if it is running as `root`. The OS user system is currently not modelled.

- **Network Properties**

  In real networks, the arrival and transfer rates of packets is non-deterministic. When running under our system, transfers are instant and TCP traffic does not get fragmented. As there is a very large number of possible fragmentations, naively simulating all of them is impractical. There is no general solution to the problem and currently, we exercise parts depending on the timing by inserting `sleep()` calls into the client.

- **Hardware Limits**

  Boa has code that deals with exceptional circumstances, like reaching the maximum number of connections. Those portions of the code base are not very easy to get triggered and would require a specific scheduler that can accept hints from the system on how to drive the execution.

- **Malloc failures**

  Currently, `malloc()` failures are not simulated.

In addition to code that we could not test due to missing features, there were parts that we should, theoretically, be able to hit but we did not. The reasons are outlined below.

- **Debug Functions**

  Boa includes debug functions that are used while developing. In normal builds, there are no function calls to them and as such we excluded them from the list of functions that are included in the coverage metric.

- **Unused Functions**

  There are also functions that were unused for several reasons – some functions were unused because they were still being worked on, others because they were deprecated internally but not removed from the code base and some were meant for future use. All of those were excluded from our code coverage metric.

- **Server Configuration**

  The server's behaviour is also highly dependent on the configuration. Thus, achieving higher coverage necessitates the testing of the server under various configuration. It also requires domain specific knowledge about which parts of the configuration have the biggest impact on code coverage.

  There are several ways to attack the problem. One would to be test multiple configurations over separate runs – although while this is certainly practical, it is a very inefficient way to go about it.

  Alternatively, one might try to mark parts of the configuration as symbolic. Unfortunately, this is an even less efficient way to explore the possible configurations due to the fact that the parser compares the file contents against a certain set of known keys (via `strcmp`). If we marked a particular 5 byte range as symbolic printable characters, it would be matched against all possible keys which would end up creating a very large amount of branches, most of which would not match a well known key.

  The approach we take is that we use a single configuration file across all runs. This is sufficient for our purpose because we are trying to evaluate the effect of symbolic data contained HTTP requests. The code that depends on different configuration values showed up in our results as a lower overall code coverage.

- **Internal Error Handling**

  Boa's functions are written in a very defensive manner (as they should be). We have observed that the code includes checks for invalid parameters that cannot actually occur – all the existing places that call those functions, ensure the parameters that are passed satisfy any preconditions. The only way for us to trigger that code would be to write additional code that calls Boa's internal functions with invalid values. As the aim of the

test is to evaluate the effects of symbolic HTTP requests, we have ignored this issue. This appeared in the results as a lower overall code coverage.

- **Interleaved Code**

  As outlined in the list above, there are several features that cannot be tested due to missing functionality. Some parts of the code that handle such features are mixed with code for features that we are testing (usually guarded by large `if` statements). We did not want to make any significant changes to the code base which might affect its correctness and thus have left the parts intact. This showed up in our results as a lower code coverage.

8.2.6 Summary

Performing evaluation on Boa has revealed several limitations of our systems and highlighted the fact that symbolically testing distributed software is a hard problem – approaching it with brute-force would rarely yield satisfactory results.

Nevertheless, we managed to find find 2 critical bugs in a very well-tested piece of software. Both of them are related to directories probably because those features not being used as actively as the others. The software has had about 15 years to mature and it is our belief that this has contributed to the low number of issues found.

Another important aspect of testing distributed software is the more pronounced problem of state explosion. This is due to the fact that not only does such software depend on the filesystem and its arguments, it now depends on the behaviour of the network nodes which it interacts with. This creates a very large increase in the number of "worlds" that need to be simulated. This is the reason why we had to split testing into separate cases and run them concurrently, as opposed to trying to simulate all the cases at the same time in one long run.

One of the reasons for excessive branching in Boa was the parsing of the request – it expects every character and checks whether it is a valid ASCII printable character or a whitespace character. Passing a completely unconstrained character actually results in significantly more processes (74% on average) but almost no increase in coverage (0.5% on average). In essence, most of the interesting code paths only happen when we have correctly formed HTTP requests.

Finally, a web server usually handles multiple connections at the same time and if we used symbolic data while processing one of the many requests, we will explore multiple paths. While semantically correct, the processing of a single client's request is virtually independent from all other requests – some web servers process those in separate threads or spawn multiple processes. This was the reason why we chose to only test a single GET request originating from a single client.

8.3 Invariants Framework

The invariants framework provides a high level tool to detect bugs at the state level. The performance of the invariant network is proportional to the rate at which new data is exposed to the system. In order to evaluate its performance, we wrote a 2-Phase Commit sample program and explored the behaviour of the program with 5 committers and completely symbolic data (so we are exploring all possible combinations of votes and decisions). More importantly, the rate at which data for invariant checking is exposed is extremely high – nevertheless, it provides us with an upper limit on the performance. During the test, we were simulating a total of 3472 processes across a multitude a worlds. The test concluded in 17 seconds and memory peaked at 135MiB. The test was run under Intel VTune Amplifier XE and the amount of time spent evaluating invariants was 5.8% – a total of 1675 invariant violations were detected during the run.

After we found bug #2 in Boa (page 83), we tested the invariants framework by writing an invariant that expressed the property that the current working directory should be the same before a request is processed and afterwards. It successfully managed to find the violation.

It is important to note the power that distributed invariants provide – in non-trivial applications, e.g. implementations of Paxos, the only way to check for correctness is to write checks over the whole network. As we are using a custom language and syntax, it should be very easy to customise and extend in order to perfectly suit our future needs.

8.4 Synthetic Scenarios

While testing our system on a production piece of software provides valuable information, it is important to know what are the classes of bugs that can be discovered. We illustrate 3 scenarios in distributed programs that will lead to the discovery of issues in the code.

8.4.1 Deadlock via Packet Loss

We have the ability to turn on non-deterministic loss of UDP packets, so that we explore both scenarios – what happens if the packet gets sent and what happens otherwise. In addition, KLEE was extended to detect when a "world" has deadlocked – this occurs when all nodes are blocked in a system call without a timeout.

8.4.2 Fragile Parsing Code

We also have the ability to automatically mark parts of network packets as symbolic without any source code changes in the program under test. This feature can be effectively used to find bugs in code that parses network packets – usually, the packets are structured by including separators or specifying the length of various structures. A symbolic value would effectively check that the code handles all cases correctly and does not try to blindly follow the contents of packets. Another way to look at it is from a security point of view – the packets could have been maliciously crafted to induce a denial of service.

8.4.3 Fault Tolerance

If we couple together the ability to inject failures (e.g., system call failures) and the invariants framework, we can automatically detect how the software under test handles those failures at the protocol level. For example, even though the software might not crash, it could be considered "faulty" due to inconsistent state across the network nodes.

A classic such example is the implementation of a two phase commit. The algorithm is quite simple and we will illustrate it in the case with 2 committers and one coordinator.

1. Both committers send a decision whether to commit (either `yes` or `no`).
2. If a committer's own decision is `no`, it can abort immediately and not wait for an answer from the server.
3. The coordinator waits to receive all votes. If all of them are `yes`, it sends `yes` to all committers, otherwise sends `no`.
4. If a committer does not receive the decision from the server within a certain timeout, it decides abort.

The invariant across all committers should be that they all decide the same value - either all of them decide `yes` or all of them decide `no`. Unfortunately, there is a case that can lead to the violation of that invariant.

1. Both committers send `yes` to the coordinator.
2. The coordinator only manages to send `yes` to the first committer.

In this case, the first committer will decide `yes` while the second will decide `no`. This is easily induced and detected by our system by enabling the failure of system calls and using the invariants framework. Note that this can also be viewed as the coordinator crashing after the first send (because from the point of view of the second committer, the coordinator has stopped responding).

8.5 Scalability

In order to assess limits of the system, we designed a set of synthetic tests that exercises various aspects of the system. An implementation of an echo server and client were written which could be configured to send a varying number of 4KiB UDP packets. In the following tests, we show the data from all the runs without any aggregation as their number was small enough to not require aggregation.

Deadlock Detection In order to determine the impact deadlock detection, we choose three particular runs of Boa that had varying amounts of processes. We run each test twice – once with deadlock detection turned on and then turned off. The memory usage in each case is shown below.

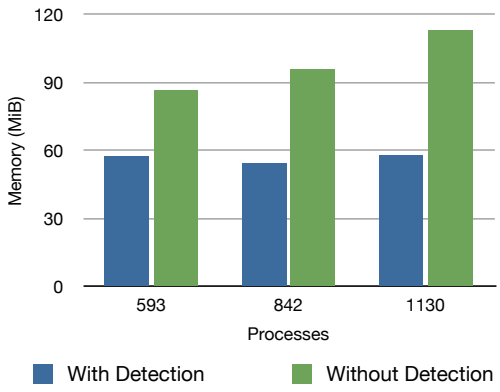


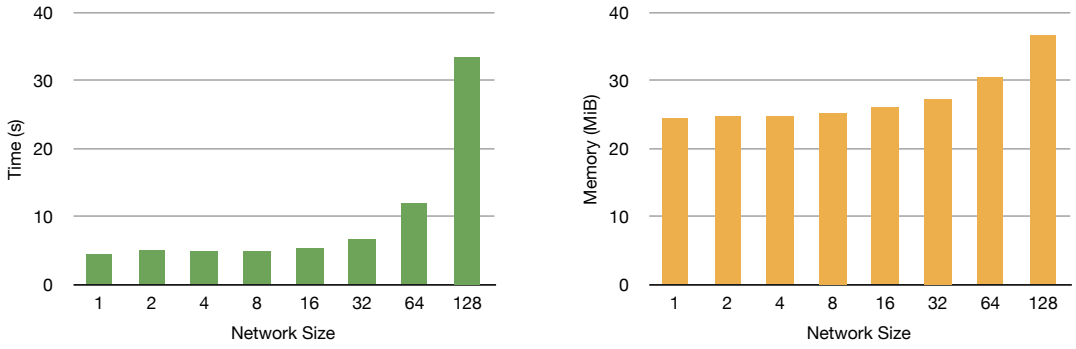
Figure 40: Showing how the number of processes affects the benefits of deadlock detection.

Branches	With Detection (MiB)	Without Detection (MiB)
593	57.4	86.4
842	54.4	95.8
1130	57.9	113.1

Figure 41: Showing how the number of processes and deadlock detection affects memory usage.

Deadlock detection can be seen to provide significant benefits when it comes to memory usage – in this case, the memory usage remains relatively constant when deadlock detection is turned on, while on the other hand, memory usage grows significantly with the number of branches / processes.

Network Size In this test, we use an echo server and send the same amount of data (32KiB each way) using different number of clients. For example, in one run, 1 client sends 128 packets of length 256 bytes. In the final run, 128 clients in the same subnet each send 1 packet of size 256 bytes.



(a) Network Size & Run time

(b) Network Size & Memory

Figure 42: Showing the memory usage and run time when transferring a total of 64KiB (32KiB each way) using a varying number of echo clients.

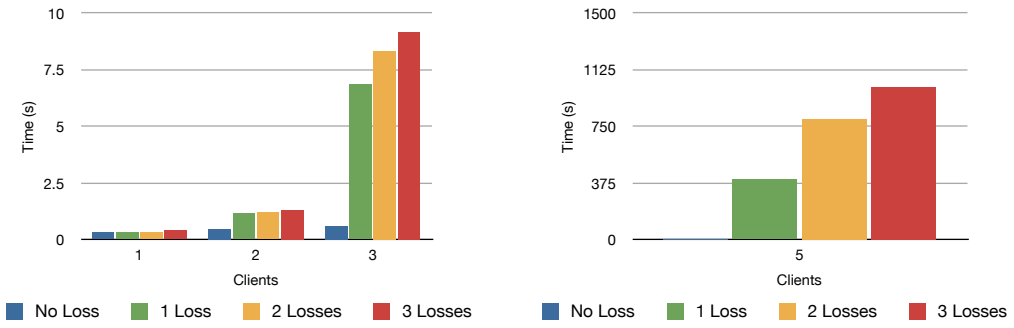
Network Size	Time (s)	Memory (MiB)
1	4.5	24.5
2	5.1	24.8
4	5.0	24.8
8	5.0	25.3
16	5.4	26.0
32	6.6	27.3
64	12.0	30.4
128	33.5	36.6

Figure 43: Showing the effect the size of the network has on run time and memory usage.

From the results, we can see that up to network sizes with 32 nodes, the increase in memory usage and run time is insignificant – for example, increasing the size from 1 node to 8 nodes results only in a approximately 10% increase in run time. On the other hand, doubling from 64 to 128 increases the time by almost 180%.

Packet Loss In this test, we aim to quantify the cost of packet loss. We run the echo program with varying number of clients and varying amounts of packet losses – specifically, we tested the effect of packet loss for 1, 2, 3 and 5 clients where each client sends 3 packets. The increase in

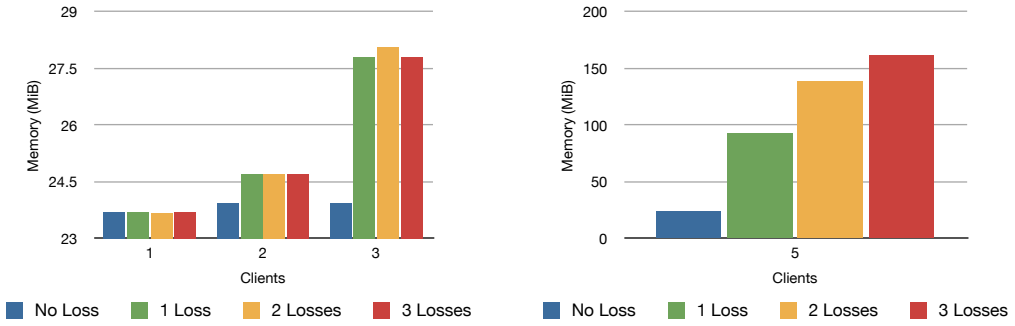
resource usage for 5 clients is so much larger than the rest that we had to separate the graphs as otherwise the bars for 1, 2 and 3 clients would be invisible.



(a) Packet Loss & Time

(b) Packet Loss & Time

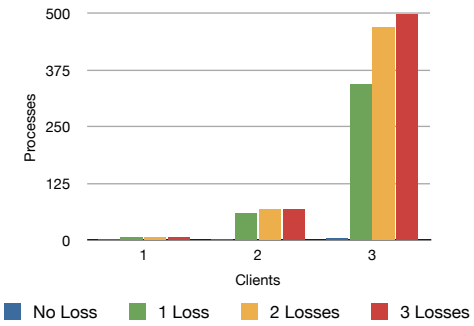
Figure 44: Showing how the run time (in seconds) varies with the number of clients and number of packets lost.



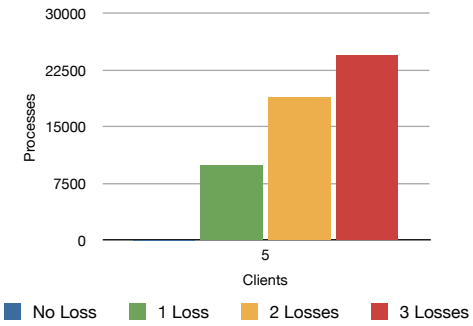
(a) Packet Loss & Time

(b) Packet Loss & Time

Figure 45: Showing how the memory usage (in MiB) varies with the number of clients and number of packets lost.



(a) Packet Loss & Time



(b) Packet Loss & Time

Figure 46: Showing how the number of processes varies with the number of clients and number of packets lost.

Clients	No Loss	1 Packet Loss	2 Packet Loss	3 Packet Loss
1	0.3	0.3	0.3	0.4
2	0.5	1.17	1.2	1.3
3	0.6	6.9	8.3	9.2
5	0.9	401.0	801.7	1010.8

Figure 47: Showing how the run time varies with the number of clients and number of packets lost.

Clients	No Loss	1 Packet Loss	2 Packet Loss	3 Packet Loss
1	23.7	23.7	23.7	27.7
2	23.9	24.7	24.7	24.7
3	23.9	27.8	28.1	27.8
5	24.2	93.1	138.3	161.6

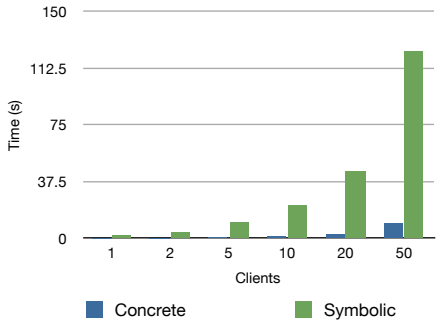
Figure 48: Showing how the memory usage varies with the number of clients and number of packets lost.

Clients	No Loss	1 Packet Loss	2 Packet Loss	3 Packet Loss
1	2	8	8	8
2	3	59	68	68
3	4	344	469	500
5	6	9940	18917	24510

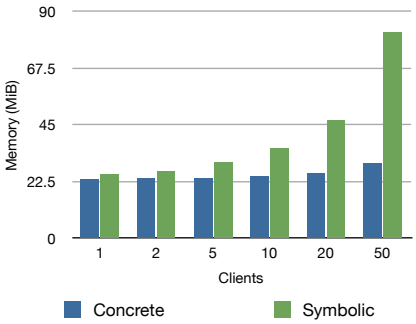
Figure 49: Showing how the number of processes varies with the number of clients and number of packets lost.

The results shown above clearly demonstrate the high cost associated with exploring large amounts of packet loss – the number of processes with 5 clients and 3 packet losses is a staggering 24510. Consequently, using packet loss should be restricted to small number of packets or smaller network setups.

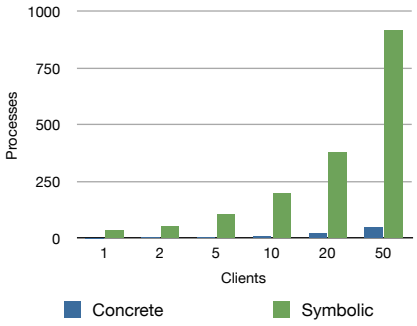
Symbolic Communication The goal of this test is to quantify the effects of excessive symbolic communication. We use the echo program to send 3 packets of data while varying the network size. In this particular setup, the server will try to match the received data (using a call to `memcmp`) against a set of predefined values. We run both tests twice – once sending concrete data and once sending completely symbolic packets.



(a) Symbolic Communication & Time



(b) Symbolic Communication & Memory



(c) Symbolic Communication & Processes

Figure 50: Showing the effects of symbolic network communication.

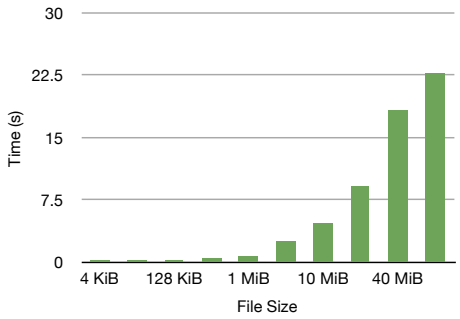
Clients	Time/C	Time/S	Mem/C	Mem/S	Procs/C	Procs/S
1	0.32	2.0	23.5	25.3	2	36
2	0.4	4.1	23.7	26.8	3	54
5	0.8	10.4	24.0	30.4	6	108
10	1.57	21.62	24.8	35.8	11	198
20	3.06	44.4	26.0	46.9	21	378
50	10.1	123.9	29.7	81.8	51	918

Figure 51: Showing the effect of network communication when using symbolic data. C stands for **C**oncrete while S for **S**ymbolic. Time is shown in seconds while memory is shown in MiB.

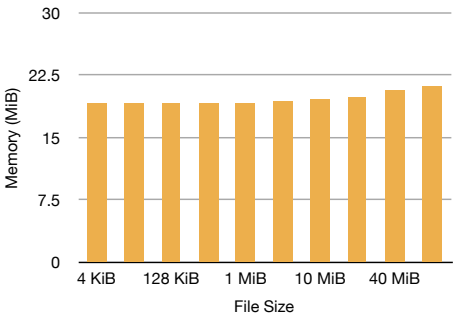
We can see from the graphs that excessive symbolic communication has significant costs on the run time and the number of processes, especially for larger networks. On the other hand, memory usage does not grow as fast as the run time and process count, mainly due to deadlock

detection.

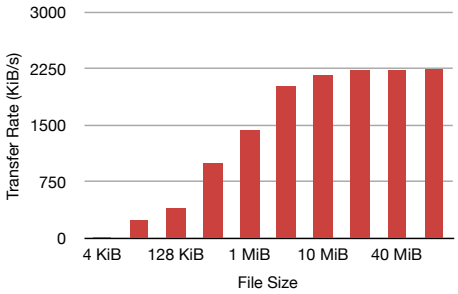
Filesystem Transfer Rate In this test, we investigated how our system performs when reading OS-backed files. We read increasingly large files in a loop, each time reading 4KiB of data and then discarding it. This setup would reveal the cost of `open()`ing the OS-backed file every time we need to read it which is done to prevent exhaustion of file descriptors due to heavy branching of processes which subsequently perform file operations.



(a) Packet Loss & Time



(b) Packet Loss & Memory



(c) Packet Loss & Rate

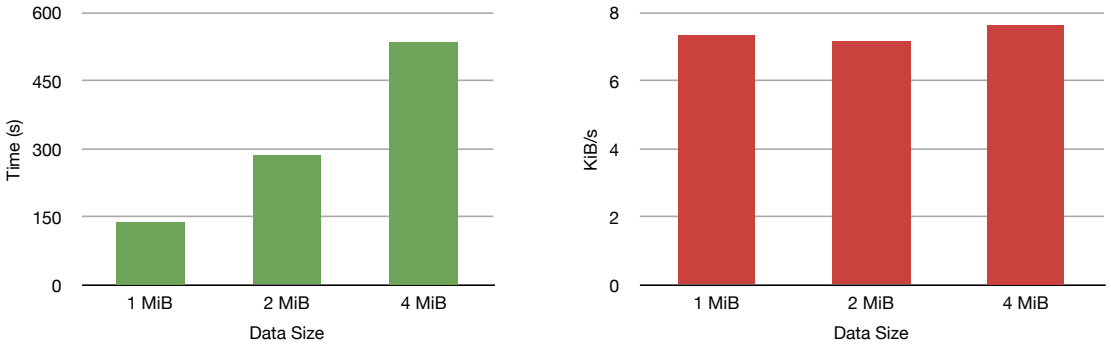
Figure 52: Showing the performance of reading files of varying sizes.

File Size	Time (s)	Memory (MiB)	Rate (KiB/s)
4 KiB	0.2	19.1	16.7
64 KiB	0.3	19.2	246.2
128 KiB	0.3	19.1	400.0
512 KiB	0.5	19.1	1003.9
1 MiB	0.7	19.1	1442.3
5 MiB	2.5	19.4	2015.7
10 MiB	4.7	19.7	2164.9
20 MiB	9.1	19.9	2238.3
40 MiB	18.3	20.7	2234.6
50 MiB	22.8	21.2	2248.6

Figure 53: Showing the effect of reading files of varying sizes on the system performance.

We can infer a few interesting facts from the graphs. Firstly, as expected, reading larger files takes longer as shown by Figure 52a. The fact that memory usage stays almost constant throughout all runs, as shown by Figure 52b is a good indication that we are not inadvertently wasting resources. Most interestingly, Figure 52c tells us that the file transfer rate from the OS effectively peaks for files larger than 5 MiB.

Network Transfer Rate In this test, we wanted to estimate the costs of transferring large amounts of data in small packets. We transfer 1, 2 and 4 MiB in 256 byte packets and record the time used so that we can calculate an effective rate.



(a) Packet Loss & Time

(b) Packet Loss & Time

Figure 54: Showing the effect of transferring large amounts of small data packets across the network.

Data Size (# of Packets)	Time (s)	Rate (KiB/s)
1 MiB (4096)	139.5	7.3
2 MiB (8192)	285.8	7.2
4 MiB (16384)	536.0	7.6

Figure 55: Showing the effect of transferring large amounts of small data packets across the network.

As expected, the time taken is linearly proportional to the amount of data sent and the rate of transfer stays fairly constant. It should be noted that the actual transfer rate is quite low – this can be explained due to the overheads of transferring data over the network which requires "context switches" between the processes being run symbolically and the system itself.

8.6 Limitations

As every piece of software, our system has a set of limitations that limit its utility. The more notable ones include:

- **Scheduling**

  As outlined in section 7.2.6, we use a random path scheduler to pick the world to simulate next (and round-robin within the world of network nodes). This only explores one way of performing scheduling between network nodes. There is a large array of distributed

software whose behaviour depends on the order of networking packets that it receives. Naively exploring all possible interleavings will further deepen the exponential state explosion problem and at present, there are no known solutions.

- **Single-Thread Single Process**

  One limitation we inherited from KLEE was the inability to use the `fork()` system call. This means that we cannot test any software systems that depend on that functionality. The solution to the problem would be to model cloning of processes in a way that preserves all the correct semantics of `fork()`.

- **Reproducibility & Non-Determinism**

  We have three main sources of non-determinism in our system that can cause problems when trying to reproduce any issues that were discovered.

  – Non-determinism inherited from the host OS – e.g., reading from `/dev/urandom`. One possible solution to this problem would be to support the recording of non-deterministic data that the program gets from the OS and make sure the same data is returned during replay. This can achieved by dynamically instrumenting the programs during replay. Pin[10] is a free tool by Intel that allows the easy instrumentation of processes.
  – If we use fault injection, the POSIX functions that we implement become non-deterministic. Thus in order to replay a test case, we will need to instrument those functions during the replay process in a similar fashion as described above.
  – It is entirely possible that bugs that were discovered depend on the particular scheduling performed by our system. Consequently, the need to follow the exact same scheduling arises. Solving the problem would involve writing a kernel extension which guides the scheduling of a certain set of processes as defined by some file. Our scheduler's role would then also require the ability to generate files that describe the decisions it made during simulation.

It is also important to notice one of the reasons state space explosion becomes an even bigger issues when simulating distributed software. In our system, network nodes are shared and only branched on-demand in order to meet the semantics of isolation within each possible "world". Even with that optimisation, chatty network protocols can results in excessive branching.

More importantly, it is possible that we might be still be branching unnecessarily. For example, suppose that two nodes (server and client) are communicating and the client branches into two due to some test on symbolic data. At this point of time, there is only one instance of the server process. But when one of the branched client tries to send a piece of data to the server, we need to branch the server as we are exploring two possible worlds. Crucially, it might be the case that two messages, even though different when compared as a sequence of bytes, might lead to the exact same execution sequence in the server.

One possible optimisation would be to determine whether that is indeed the case. Unfortunately, performing that check is highly non-trivial – if we just simulated both messages, we are back to square one and we did not make any savings. Furthermore, it is completely unknown whether the savings of any such optimisations would outweigh the associated costs.

8.7 Summary

Symbolic execution of distributed software can yield satisfactory results although it is not yet practical for large-scale programs. Blindly marking network data as symbolic does not automatically translate to achieving higher code coverage in a short amount of time. Domain-specific

knowledge of the code under test and the structure of the network traffic can be used to attain significantly better results.

9 Development Methodology

During the development of the project, we were faced with several technical challenges that influenced our development efforts. In this section, we provide a short overview of the main issues encountered and any steps we took to mitigate their effects.

9.1 Code Base

KLEE itself is built on top of LLVM and depends on its presence in source form to be able to compile. In addition, at the time of writing there were incompatibilities between KLEE and the latest version of LLVM (2.8) which meant that we had to use version 2.7. In addition, even though the build process is documented fairly well, we still believe it can be made a lot simpler as we see it as a barrier for people to experiment with the software. The situation improved right at very end of this project's timeline when a self-contained package of KLEE and all the necessary dependencies was made available on the official website.

In addition, the sheer size of the code base presents several challenges – the raw number of lines of code in KLEE and LLVM combined is close to 200,000 (the vast majority which is C++). Consequently, there is a very steep learning curve when trying to understand the how the various subsystems fit in. At times, it was necessary to not only understand the high-level design but also the intricate implementation details in various components which can become very time-consuming. On multiple occasions, the implementation of a feature that we deemed trivial at the conceptual level, turned out disproportionally time-consuming in practice.

On the other hand, it is our feeling that both KLEE and LLVM are well-designed from an engineering point of view which greatly helped us achieve our planned targets.

9.2 Tools & Language

As engineers, the tools that are available can have a very significant impact on our productivity. Our complete unfamiliarity with the code base meant that we needed appropriate tools in order to be effectively working with the code base.

Initially, we started off by using `vim` with several extensions but it quickly became obvious that it was not practical with our level of knowledge of the codebase. After researching the available tools that would allows us to navigate the large codebase effectively, we settled on using Eclipse with the CDT extension. Overall, coming from a platform which provides an integrated set of development and performance-related tools, we felt that the development platform can be significantly improved to allow for a more streamlined and efficient workflow.

KLEE and LLVM are themselves written in C++. Due to the broad feature-set and paradigm support of the language, there are many ways to perform particular tasks. Most of the codebase follows the best practices for development in C++ albeit one place that was inconsistent throughout the code base was the managing of the lifetime of objects. Some places would use custom reference counting while others would assume implicit ownership of the object graph in specific ways – the worst part being that the implicit conventions were not documented which resulted in time spent tracking down the responsibility of managing the lifetime of a particular object. The number of times we had to debug memory-related issues (most commonly segmentation faults) was very small and in all cases it was due to either oversights in the lifetime management or due to undocumented object ownership.

9.3 Testing

The complexity of symbolic execution is very high and KLEE deals with many intricate issues at various levels of abstraction. Thus modifying the behaviour of the system introduces a possibility of accidentally introducing bugs as a side effect of our changes. More importantly, one of the worst ways for us to break the system is to introduce subtle issues which lead to incorrect semantics of execution but does not lead any clearly visible effects for us to notice. It is of utmost importance for a tool, whose goal is to prove correctness, to be itself correct.

The original creators of KLEE were aware of that fact and provided an extensive suite of test cases that ensure that the system is operating correctly. It uses DejaGnu as a testing framework. The testing suite consists of little C programs (usually with a single `main()` function) that exercise various aspects of the software and check for the expected behaviour.

The existence of the test suite allowed us to be more confident about any changes that we were making and provided the freedom to refactor subsystems without fear of indadvertedly breaking the code. As an internal rule to avoid breakages, we always run the test suite before committing. In addition to using the test suite provided by KLEE, we had our own set of tests that we were continuously writing to test the features that we were implementing. Initially, they were not integrated as part of the build process but subsequently we cleaned them up and moved them into test suite. One of the challenges we faced when writing our own test cases was the inability to verify the correct operation of some features, as the verification required a global access to the states across all processes, which is not available to programs that are being run symbolically. Overall, throughout the lifetime of the project, the test suite found a total of 2 regressions. Even though the number is very low, it is very important to realise that if the test suite was inexistent, we would have introduced issues that we might not have detected otherwise and which would have directly impacted the operational correctness. In summary, we added an additional 20 test cases which exercise the additions that we made to the system for a total final count of 126, while at the same time retaining full backwards compatibility of the system. It should be noted that those are not unit tests but system tests which check the behaviour of the whole tool.

9.3.1 Test Example

Listing 25 shows an example of one our tests that check for the correct behaviour of file descriptor duplication. This particular test exercises many aspects of the filesystem. On line 15, the call to `open()` implicitly tests for support for relative pathnames – note that any relative pathnames are relative to the current process' working directory and not the working directory of our system. Line 18 creates a new file descriptor by duplicating an existing one while line 21 replaces the standard output stream with `file_1`. The file in question contains the data `contents of file_1`. Lines 24-28 read the first 8 bytes from the original file descriptor and compare it against what is expected. Later, on lines 30-33, data is read from the first duplicate descriptor. **Crucially**, duplicate file descriptors are defined to share the current offset into the file which is checked by comparing whether we have read what we expect (line 33). Finally, we try to read from the standard output stream (line 35-38) and we expect to instead read from the file since we replaced `stdout` with `file_1` on line 21.

Listing 25: Verifying the correct behaviour of file descriptor duplication.

```
1 // RUN: %llvmgcc %s -emit-llvm -O0 -c -o %t2.bc
2 // RUN: %klee --distrib-runtime --libc=uclibc --exit-on-error %t2.bc
3
4 #include <string.h>
5 #include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <assert.h>
#include <stdio.h>

#define BUF_LEN (128)

int main(int argc, char* argv[]) {
  int fd = open("../testing-dir/file_1", O_RDONLY);
  assert(fd != -1 && "Couldn't open file_1");

  int dupfd1 = dup(fd);
  assert(dupfd1 != -1 && "Couldn't duplicate file descriptor");

  int dupfd2 = dup2(fd, STDOUT_FILENO);
  assert(dupfd2 != -1 && "Couldn't duplicate stdout");

  char buf[BUF_LEN];
  int bytes = read(fd, buf, 8);
  assert(bytes == 8 && "Couldn't read the first 8 bytes");
  buf[bytes] = 0x0;
  assert(strcmp(buf, "contents") == 0 && "Didn't read what was expected");

  bytes = read(dupfd1, buf, 4);
  assert(bytes == 4 && "Couldn't read the next 4 bytes");
  buf[bytes] = 0x0;
  assert(strcmp(buf, " of ") == 0 && "Didn't read what was expected");

  bytes = read(STDOUT_FILENO, buf, 6);
  assert(bytes == 6 && "Couldn't read the next 6 bytes via stdout");
  buf[bytes] = 0x0;
  assert(strcmp(buf, "file_1") == 0 && "Didn't read what was expected");

  return 0;
}
```

10 Conclusion

Working on extending KLEE to support the execution of distributed software has been a challenging and rewarding experience in many different ways. It provided the opportunity to work on a project that not only presented a challenge from a engineering point of view but also dealt with a very hard problem – the automatic testing of distributed software via symbolic execution. Furthermore, it provided the chance to explore the limits of what is practical and theoretically possible when it comes to providing a testing framework for networked programs.

We faced a variety of challenges throughout the project. The engineering problems that cropped up throughout the project lifetime highlighted several aspects when it comes down to producing large software system. For example, when developing software, one of the most important aspects should be to ensure ease of future maintenance and extension as those two phases are a certainty in the vast majority of software. In practice, this translates into the following proper engineering practices and adhering to guidelines which ensure consistency throughout the codebase. Consistency is desirable, especially more so on large projects, as it lowers the learning curve and allows faster comprehension of the code base. Furthermore, we have found that carefully placed comments can be very beneficial – as long as the comments reveal knowledge that is not clearly apparent to professional engineers. From our experience, those comments usually reveal invariants and properties about the code at a wider scope and how it fits with the rest of the structures as opposed to explaining the operations performed by the code.

The importance of having a testing framework to ensure the quality of software cannot overstated enough. It also increases the freedom engineers have to restructure the software without fear of breaking it. Using revision control tools, preferably distributed, provides the ability to easily backtrack in the cases where regressions have been introduced. It also provides a complete history of the project and data derived from it can be used for analytical purposes.

In this project, we managed to design and implement, via a set of extensions, a system which can symbolically execute distributed software. In addition, we provide mechanisms for automatically injecting low-probability events, such as packet loss and system call failures, which lead to exploration of rarely taken code paths in the hope of finding bugs. If any issues are found, we provide a replay framework that allows the reproduction of any bugs found so that the cause can be localised and rectified. We also introduced a distributed invariants framework that allows the expression and verification of properties over the states of all participants in a network. This mechanism enables the ability to find logical errors at a much higher level – for example, it can check the implementation of specific network protocols.

Even though we made great strides in making symbolic execution of arbitrary distributed software a reality, it is still out of practical reach for the majority of networked production software. Achieving reasonable code-coverage requires careful choice of which parts should be marked as symbolic. If we indiscriminately mark all network traffic as symbolic, the system becomes completely overwhelmed with the explosion of states and quickly runs out resources. Furthermore, the usage of the distributed invariants functionality requires domain specific knowledge about what properties actually hold.

10.1 Future Work

Despite achieving our initially set goals and going beyond, there are still many areas of improvement. We outline some ideas for potential extensions.

- **Process Persistence**

In the current implementation, our system keeps all states in memory and there is a mechanism to avoid running out of memory – either processes can be randomly terminated or any further branching can be restricted. In both cases, we forcefully stop exploring certain paths due to resource constraints. One possible solution would be to provide the ability to persist processes so that they can be offloaded to the disk. Coupled with the increase use of Solid State Drives, this solution can provide practical benefits in the exploration of code paths.

- **Scheduling**

  Currently, when it comes to timing and choosing the next process to run, we follow only a single strategy (round-robin within a world). Providing the mechanisms to vary the behaviour of the scheduler such that multiple interleavings can be tested without a huge performance penalty is another interesting area of research.

- **Distributed Invariants**

  Our current implementation of distributed invariants is quite simple – in essence, it provides the foundation for future work. The ability to have finer control over the timing of evaluation and extensions to the language are two possible extensions. One way to attack this would be to pick several larger pieces of software[12] and try to provide any necessary extensions such that the programs' invariants can be expressed and tested.

- **Network Extensions**

  Our system does not simulate any properties of the networks themselves, such as delay, capacity and others. Additions which allow the exploration of the behaviour of software under different network loads can reveal additional behaviour and possibly reveal issues. For example, sockets currently have unlimited buffer space – this can easily be changed so that the behaviour of systems can be tested for proper handling of high transfer rates where the OS buffers quickly fill up.

- **fork() Support**

  One significant restriction at the moment inherited from KLEE itself is the inability to use the `fork()` system calls. Support for `fork()` would allow the testing of a much wider variety of networked programs.

- **Replay Extensions**

  The replay framework currently suffers from two main problems: issues found by automatic failure injection cannot be reproduced and non-deterministic sources of data (e.g., `/dev/urandom`). Both of those issues can be fixed by adding support for additional information to be saved during symbolic simulation and then instrumenting[10] the process under test when it runs natively so that the behaviour of any system calls matches the behaviour when run by our system.

- **Parallelism**

  Currently, our system does not take any advantage of parallelism offered by multi-core processors. In addition, ability to cluster multiple instances in a way that can run in the cloud could provide the ability to scale enough such that it becomes practical to test much bigger pieces of software.

References

[1] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[3] Larry Doolittle and Jon Nelson. Boa webserver. `http://www.boa.org/`.

[4] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22:84–, March 1997.

[5] Michael Ellims, James Bridges, and Darrel C. Ince. The economics of unit testing. *Empirical Softw. Engg.*, 11:5–31, March 2006.

[6] Ondrej Filip, Libor Forst, Pavel Machek, Martin Mares, and Ondrej Zajicek. The bird internet routing daemon project. `http://bird.network.cz/`, 2011.

[7] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[8] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26:18–41, July 1993.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[11] Technical University of Madrid. Vnuml - virtual network user mode linux. `http://www.uni-koblenz.de/~vnuml/`.

[12] Marco Primi. Libpaxos: Open-source paxos. `http://libpaxos.sourceforge.net/`.

[13] Raimondas Sasnauskas, Jó Ágila Bitsch Link, Muhammad Hamad Alizai, and Klaus Wehrle. Kleenet: automatic bug hunting in sensor network applications. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 425–426, New York, NY, USA, 2008. ACM.

[14] David Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.

[15] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

A Evaluation Test Configurations

This sections provides enough detail in order to reproduce all the tests performed in section 8. We omit any source code to the programs as it is included in the project archive.

A.1 Boa

In this subsection, we provide full details about all tests performed when analysing the performance of our system with Boa.

A.1.1 GET Requests

Listing 26: Showing the 29 handcrafted GET requests.

```
1  const char* REQUESTS[] = {
2      "", // 0
3      "GET / HTTP/1.1\r\n\r\n", // 1
4      "GET //files/./more/../nesting/index.html HTTP/1.1\r\n\r\n", // 2
5      "GET /bar/index.html HTTP/1.1\r\n\r\n", // 3
6      "GET /in<de&>\\x.html HTTP/0.9\r\n\r\n", // 4
7      "GET /in<de&>\\x.html HTTP/1.1\r\n\r\n", // 5
8      "GET /ind??##ex.html HTTP/1.1\r\n\r\n", // 6
9      "GET /index.html HTTP/1.1\r\nHost: klEE.com:8080\r\n\r\n", // 7
10     "GET /index.html HTTP/1.1\r\nRange: bytes=0-1, 3-4\r\n\r\n", // 8
11     "GET /index.html HTTP/1.1\r\nRange: bytes=0-2\r\n\r\n", // 9
12     "GET /index.html HTTP/1.1\r\nRange: bytes=500-550\r\n\r\n", // 10
13     "GET /index.html HTTP/1.1\r\n\r\n", // 11
14     "GET /index.html HTTP/2.3\r\n\r\n", // 12
15     "GET /index.html HT\rTP/\n1.1\r\n", // 13
16     "GET /index_non_exist.html HTTP/1.1\r\n\r\n", // 14
17     "GET /no_dir/ HTTP/1.1\r\n\r\n", // 15
18     "GET /programs/index.html HTTP/1.1\r\n\r\n", // 16
19     "GET /sym1.html HTTP/1.1\r\nIf-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
           \r\n\r\n", // 17
20     "GET /sym1.html HTTP/1.1\r\nIf-Range: Sat, 29 Oct 1994 19:43:31 GMT\r\nRange
           : bytes=0-20\r\n\r\n", // 18
21     "GET /sym1.html HTTP/1.1\r\n\r\n", // 19
22     "GET /sym2.html HTTP/1.0\r\n\r\n", // 20
23     "GET /sym2.html HTTP/1.1\r\n\r\n", // 21
24     "GET /~/index.html HTTP/1.1\r\n\r\n", // 22
25     "GET /~md207/index.html HTTP/1.1\r\n\r\n", // 23
26     "GET sym1.html HTTP/1.1\r\n\r\n", // 24
27     "SET /index.html HTTP/1.1\r\nRange: bytes=0-2\r\n\r\n", // 25
28     "HEAD /sym1.html HTTP/0.9\r\n\r\n", // 26
29     "HEAD /sym1.html HTTP/1.1\r\n\r\n", // 27
30     "POST /index.html HTTP/1.1\r\nContent-Length: 5\r\n\r\nHello" // 28
31 };
```

A.1.2 Function List

Listing 27: Showing the list of functions used to compute the code coverage metrics.

```
1 #----------
2 # alias.c |
3 #----------
4 get_alias_hash_value
5 add_alias
6 find_alias
```

```
translate_uri

#--------
# boa.c |
# -------
__user_main
main_boa_server
parse_commandline
create_server_socket
drop_privs
fixup_server_root

#-----------
# buffer.c |
#-----------
req_write
req_write_escape_html
req_flush
escape_string

# ----------
# config.c |
# ----------
c_set_string
c_set_int
c_set_unity
c_add_mime_type
c_add_mime_types_file
c_add_alias
lookup_keyword
apply_command
trim
parse
read_config_files

# disabled as we don't support access / cgi
#c_add_access
#c_add_cgi_env

# disabled due to no support of changing of users etc
#c_set_user
#c_set_group

#--------
# get.c |
#--------
init_get
get_dir
get_cachedir_file
index_directory

# disabled as not called
#process_get

#---------
# hash.c |
#---------
boa_hash
fnv1a_hash
hash_insert
```

```
hash_find
add_mime_type
get_mime_hash_value
get_mime_type
get_homedir_hash_value
get_home_dir


#-------
# ip.c |
#-------
bind_server
ascii_sockaddr
net_port

#--------
# log.c |
#--------
open_logs
log_access
log_error_doc
boa_perror
log_error_time
log_error
log_error_mesg
log_error_mesg_fatal

#----------
# queue.c |
#----------
block_request
ready_request
dequeue
enqueue
range_pool_pop
range_pool_push
range_add
ranges_fixup
range_parse

# disabled as only triggered on high limits
#range_pool_empty
#range_abort
#ranges_reset

#---------
# read.c |
#---------
read_header

# never triggered as it's for POST
#read_body
#write_body

#------------
# request.c |
#------------
new_request
get_request
sanitize_request
```

```
127 free_request
128 process_requests
129 process_logline
130 process_header_end
131 process_option_line
132
133 #-------------
134 # response.c |
135 #-------------
136 http_ver_string
137 print_content_type
138 print_content_length
139 print_last_modified
140 print_http_headers
141 print_content_range
142 print_partial_content_continue
143 print_partial_content_done
144 complete_response
145 send_r_request_ok
146 send_r_no_content
147 send_r_partial_content
148 send_r_moved_perm
149 send_r_moved_temp
150 send_r_not_modified
151 send_r_bad_request
152 send_r_unauthorized
153 send_r_forbidden
154 send_r_not_found
155 send_r_request_uri_too_long
156 send_r_invalid_range
157 send_r_error
158 send_r_not_implemented
159
160 # disabled due to CGI
161 #send_r_bad_gateway
162 # disabled as sent when exceeding number of connections
163 #send_r_service_unavailable
164
165 # disabled as never used
166 #send_r_bad_version
167 #send_r_precondition_failed
168 #send_r_length_required
169 #send_r_continue
170
171 #-----------
172 # select.c |
173 #-----------
174 loop
175 fdset_update
176
177 #-------------
178 # timestamp.c|
179 #-------------
180 timestamp
181
182 #---------
183 # util.c |
184 #---------
185 clean_pathname
186 get_commonlog_time
```

```
187 month2int
188 date_to_tm
189 modified_since
190 to_upper
191 unescape_uri
192 rfc822_time_buf
193 simple_itoa
194
195 # disabled as they require POST + CGI
196 #create_temporary_file
197 #boa_atoi
```

A.1.3 Non-Symbolic Runs

Listing 28: Non-symbolic runs.

```
1 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 0
2 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 1
3 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc --split 1
4 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 2
5 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 3
6 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 4
7 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 5
8 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 6
9 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 7
10 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 8
11 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 9
12 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 10
13 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 11
14 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 12
15 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 13
16 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 14
17 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 15
18 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 16
19 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 17
20 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 18
21 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 19
22 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc --split 19
```

```
23 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 20
24 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 21
25 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 22
26 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 23
27 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 24
28 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 25
29 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 26
30 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 27
31 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 28
```

A.1.4 Symbolic Runs

Note that the `--split` switch was used on two requests in order to increase the code coverage by a small fraction – the switch sends the HTTP request in 3 chunks, waiting 1 second between each send.

Listing 29: Symbolic runs.

```
 1 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0
 2 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4
 3 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8
 4 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9
 5 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10
 6 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14
 7 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17
 8 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 19
 9 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 20
10 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 22
11 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 25
12 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 25
13 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 22
14 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 22
15 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 14
16 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 19
```

```
17 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 19
18 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 10
19 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 14
20 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 10
21 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 22 25
22 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 17
23 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 19
24 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 17
25 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 22
26 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 10
27 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 20
28 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 25
29 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 20
30 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 22
31 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 19
32 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 4 8
33 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 19 20
34 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 20 22
35 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 19 25
36 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 9 25
37 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 17 20
38 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 9 14
39 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 14 25
40 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 19 20
41 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 17 20
42 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 4 14
43 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 8 25
44 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 22 25
45 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 19 22
46 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 9 17
```

```
47 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 17 19
48 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 17 25
49 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 9 14
50 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 14 19
```

A.1.5 Constrained Symbolic Runs

Listing 30: Constrained symbolic runs.

```
 1 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 --printable
 2 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 --printable
 3 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 --printable
 4 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 --printable
 5 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 --printable
 6 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 --printable
 7 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 --printable
 8 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 19 --printable
 9 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 20 --printable
10 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 22 --printable
11 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 25 --printable
12 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 0 25 --printable
13 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 22 --printable
14 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 22 --printable
15 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 14 --printable
16 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 17 19 --printable
17 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 19 --printable
18 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 4 10 --printable
19 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 14 --printable
20 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 10 --printable
21 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 22 25 --printable
22 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 9 17 --printable
23 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 19 --printable
```

```
24 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 17 --printable
25 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 22 --printable
26 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 10 --printable
27 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 20 --printable
28 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 14 25 --printable
29 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 20 --printable
30 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 8 22 --printable
31 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
       uclibc boa.bc 19 10 19 --printable
```

A.1.6 Failure Injected Runs

Listing 31: Failure injected runs.

```
1 klee --optimize --disable-inlining --distributed-mode --distrib-runtime --libc=
      uclibc boa.bc 19 --max-sys-fail 1
2 klee --optimize --disable-inlining --max-memory=1200 --max-memory-inhibit --
      distributed-mode --distrib-runtime --libc=uclibc boa.bc 19 --max-sys-fail 2
```

A.2 Invariants

For the invariants framework performance assessment, we run the simplistic implementation of a 2-Phase Commit using the command shown in Listing 32.

Listing 32: Command used to run the 2PC program.

```
1 klee --distributed-mode --distrib-runtime --libc=uclibc --invariants-path=source
      .minv 2pc.bc 5 1 --corrupt-packet-count 1 --corrupt-data-size 1
```

Listing 33: Invariant used in 2PC run.

```
1  invariant decisionConsistency(data[] d) : nodes, states, keys {
2    return d.equalElements();
3  }
4
5  string[] nodes() {
6    return sys.nodes();
7  }
8
9  int[] states(string node) {
10   return int[1] ;
11 }
12
13 string[] keys(string node, int state) {
14   return string["decision"];
15 }
```

A.3 Scalability Tests

In the rest of this subsection, we give full details about the scalability test runs.

A.3.1 Deadlock Detection Runs

Listing 34: Commands used to assess deadlock detection effects.

```
1 klee --optimize --distributed-mode --distrib-runtime --libc=uclibc  boa.bc 19 0
      --max-sys-fail 1
2 klee --optimize --distributed-mode --distrib-runtime --libc=uclibc  boa.bc 19 4
      --max-sys-fail 1
3 klee --optimize --distributed-mode --distrib-runtime --libc=uclibc  boa.bc 19 22
       --max-sys-fail 1
4 klee --detect-stalled=0 --optimize --distributed-mode --distrib-runtime --libc=
      uclibc  boa.bc 19 0 --max-sys-fail 1
5 klee --detect-stalled=0 --optimize --distributed-mode --distrib-runtime --libc=
      uclibc  boa.bc 19 4 --max-sys-fail 1
6 klee --detect-stalled=0 --optimize --distributed-mode --distrib-runtime --libc=
      uclibc  boa.bc 19 22 --max-sys-fail 1
```

A.3.2 Network Size Runs

Listing 35: Commands used to assess network size effects.

```
1 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      4096
2 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      8192
3 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      16384
```

A.3.3 Packet Loss Runs

Listing 36: Commands used to assess packet loss effects.

```
1 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1 3
2 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 2 3
3 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 5 3
4 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 3 3
5 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1 3
      --lost-packet-count 1
6 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 2 3
      --lost-packet-count 1
7 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 5 3
      --lost-packet-count 1
8 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 3 3
      --lost-packet-count 1
9 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1 3
      --lost-packet-count 2
10 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 2 3
      --lost-packet-count 2
11 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 5 3
      --lost-packet-count 2
12 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 3 3
      --lost-packet-count 2
13 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1 3
      --lost-packet-count 3
14 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 2 3
      --lost-packet-count 3
15 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 5 3
      --lost-packet-count 3
```

```
16 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 3 3
       --lost-packet-count 3
```

A.3.4 Symbolic Communication Runs

Listing 37: Commands used to assess symbolic communication effects.

```
1 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 1 3
2 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 2 3
3 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 5 3
4 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 10 3
5 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 20 3
6 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      match 50 3
7 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 1 3
8 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 2 3
9 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 5 3
10 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 10 3
11 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 20 3
12 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc --
      symbolic --match 50 3
```

A.3.5 Filesystem Transfer Rate

Listing 38: Commands used to assess file transfer rate.

```
1 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      128bytes
2 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      1024bytes
3 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      4096bytes
4 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      128kbytes
5 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      512kbytes
6 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      64kbytes
7 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      1mbytes
8 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      5mbytes
9 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      10mbytes
10 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      20mbytes
11 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      40mbytes
12 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc file_read.bc
      50mbytes
```

A.3.6 Network Transfer Rate

Listing 39: Commands used to assess network transfer rate.

```
1 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      4096
2 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      8192
3 klee --no-output --distributed-mode --distrib-runtime --libc=uclibc echo.bc 1
      16384
```