STANDALONE CONSTRAINT SOLVER FOR ASYNCHRONOUS SYMBOLICEXECUTION

by

Jiakuan Li

(Under the Direction of Kang Li)

Abstract

Symbolic execution is a technique to analyze a binary. It symbolizes the input of a program and try to generate testcases that covered all of the feasible path. The constraint solver is one of the indispensable and the most critical components in a symbolic execution engine. All of the symbolic execution engines rely on solvers to conduct testcases and guide the exploration during the operation process. Currently, and engine regularly spends comparatively long time solving the constraints. To shorten the time for analyzing a binary, using a standalone solver is one of the practical strategies. In my research, I implemented a standalone solver that takes in kquery strings and generating testcases according to the constraints in that kquery it receives. Moreover, to improve the performance of the solver, the solver now can share caches.

Index words: KLEE, Symbolic Execution, Constraint Solver

STANDALONE CONSTRAINT SOLVER FOR ASYNCHRONOUS SYMBOLICEXECUTION

by

Jiakuan Li

B.S., University of Georgia, 2016

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2018

STANDALONE CONSTRAINT SOLVER FOR ASYNCHRONOUS SYMBOLICEXECUTION

by

Jiakuan Li

STANDALONE CONSTRAINT SOLVER FOR ASYNCHRONOUS SYMBOLICEXECUTION

Jiakuan Li

August 2018

For My Parents

Acknowledgments

I would like to thank my thesis advisor Professor Kang Li at The University of Georgia. Professor Li always answered my questions patiently whenever I ran into trouble of my research or had problems of implementing the projects. He consistently allowed this thesis to be my own work but steered me it in the right direction. More important, I learned a lot of research skills as well as presenting skills from him. I am sure that these skills will benefit me in my future career.

I would also like to thank my colleagues in Network System and Security Lab. Working with such a great team will always be my pleasure. We worked together to solve engineering problems also discussed our research together, which are mind refreshing and incredibly inspiring for me and even this thesis.

Finally, I must express my very profound gratitude to my girlfriend Yunying Xu for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without her.

I hope this thesis will not be my final destination of academic research, and hope it is not just a hope.

Contents

List of Figures

List of Tables

List of Code Snippet

Chapter 1

INTRODUCTION

Symbolic Execution King [1976] is a technique that can explore all of the possible paths of a binary using a constraint solver. The method came out a long time ago, but until recentdays Cadar and Sen [2013], it has gathered significant attention. People implemented several tools and found bugs and security vulnerabilities that buried very deep in the code in different software applications Cadar et al. [2011].

The reason of why symbolic execution gathered significant attention in the recent few years is because of the development of the SMT solver De Moura and Bjørner [2011]. But still, constraints solver remains one of the main challenges of symbolic execution Palikareva and Cadar [2013]. As a result, to increase the speed of a symbolic execution engine, we can either improve the solving speed of the solver or optimize how the symbolic execution uses the engine. Moreover, let the solver uses previous constraint result is another option.

1.1 Problem Statement

All of the current symbolic execution engines can systematically cover all the feasible path for a binary during its analysis process. However, it has its disadvantages, that is symbolic execution engine are slower than another type of dynamic analysis tools. So, can we use some other techniques to help the analysis process go faster? I target my research on one of the popular symbolic execution tool, KLEE Cadar et al. [2008], and try to shorten the analysis time for a program.

1.2 Research Method

The symbolic execution engine KLEE is a synchronous engine. During its execution, it is always waiting for the response from the solver to continue the exploration. While asking the solver to generate testcases, that means the current path reaches the end, and the rest of the path exploration does not rely on that result. Therefore, waiting for the solver's response is wasting time.

Hypothesis 1 *Making a KLEE in asynchronous mode can significantly reduce the time that is required for covered all the path in a program.*

In addition to providing KLEE an asynchronous mode, solver optimization is one of the research's goal. KLEE's implementation allows the solver using cache, so let the standalone solver using cache is one of the optimization methods.

Hypothesis 2 *Allowing the standalone solver using cache can reduce the number of counterexamples that generated by the solver.*

1.3 Contribution

In this research, I have the following contributions:

- Allow KLEE to analysis the program in asynchronous mode.
- Adding feature of produce testcases for every branch node that has a possible path.
- Implementing a server to receive query command that KLEE sends out during the analysis process.
- Wrapping up a solver that checks the cache for the query before throwing it into the SMT solver to generate a counterexample.

1.4 Outline

This thesis is consist of the following chapters.

- Chapter 2: Background Knowledge and Related Work
- Chapter 3: Implementation
- Chapter 4: Experimental Evaluation
- Chapter 5: Summary and Conclusion

Chapter 2

BACKGROUND KNOWLEDGE AND RELATED WORK

In this chapter, I will introduce the background knowledge of symbolic execution and the related work of my research.

2.1 What is Symbolic Execution?

Symbolic Execution is one of the program analysis techniques whose ultimate goal is to systematically cover all the feasible paths by solving the paths' constraints by using an SMT (Satisfiability Modulo Theory). In order to get the idea of what symbolic execution really is, I will use Snippet 2.1 as an example to explain the detail of symbolic execution.

Snippet 2.1: A Simple Program

```
1  int x = 0, y = 0, z = 0;
2  if (a) {
3      x = -2;
4  }
5  if (b < 5) {
6      if (!a && c) {
7          y = 1;
8      }
9      z = 2;
10 }
11 assert(x+y+z != 3)
```

In static analysis, the value of `x`, `y` and `z` will be assigned a variable, then walk through the program to see what part of the code is covered. Then reassign values to `x`, `y`, and `z` and repeat this step until all the possible paths are covered. Apparently, this method is a very low-efficiency method because there is no guarantee that every execution will have a new path be covered.

Unlike static analysis, symbolic execution is way more efficient. In symbolic execution, no values will be assigned to `x`, `y`, and `z`. Instead, the variables `x`, `y`, and `z` are treated as "symbol." When the execution hit line 2, two constraints will be added. They are `a == 1` and `a != 1` representing the `true` and `false` of the if-statement respectively. These constraints will be added to two buckets. Each bucket is representing a possible path of this program. Then the engine will continue executing. Once it hits line 5, like the if-statement in line2, two constraints will be produced and added to the buckets. And finally, connecting all the constraints in each bucket with the `AND` logical operator, an assignment for that path is generated.

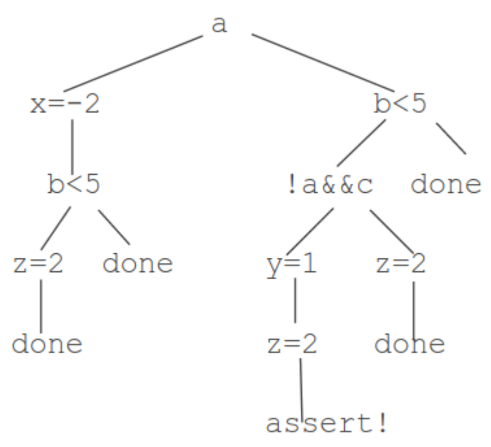
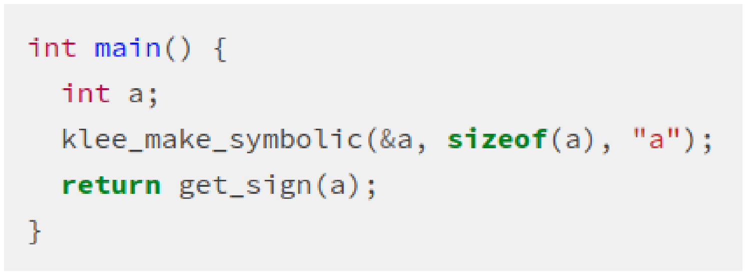


Figure 2.1: Path Conditions

Figure 2.1 is a tree of path condition of the Snippet 2.1. Each of the nodes is a line of code. If a tree node has two children, it means the there is a possible path at that node. Also, the left child representing the `true` path and the right child representing the `false` path. To activate the left most path, the testcase should fulfill the constraints of `(a == 1 AND b < 5)`. To activate the second from the left path, the testcase should fulfill the constraints of `(a == 1 AND b >= 5)`. To assert, the constraints is `(a != 1 AND b < 5 AND c == 1)`. The constrains for the rest of two paths is `(a != 1 AND b < 5 AND c != 1)` and `(a != 1 AND b >= 5)` respectively.

Comparing static analysis and symbolic execution, the main difference is that symbolic execution get the assignment according to the path it explored, and static analysis gets the assignment before exploring the path. As we can see, there are five possible paths in the program. Therefore, five testcases will be generated to cover all the paths after the symbolic execution process. But for static analysis, probably more than five testcases will be generated to cover all of the feasible paths.

2.2 KLEE



```c
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Figure 2.2: Make Input As Symbolic

KLEE is a symbolic execution tool that can generate testcases that achieve high coverage on a variety of complex and environmentally-intensive programs automatically Cadar et al. [2008]. Using the GNU `COREUTILS` as a benchmark, they covered 90% of the code on average for all 89 utilities in the package. KLEE can only test the programs that are written in C/C++ and have the source code available. First, a function `klee_make_symbolic` need to be added to the code to mark the input as symbol. Figure 2.2 demonstrates how to make the input symbolic. Then compile the code into LLVM bytecode. The reason for doing so is because KLEE simulates the system environment to handle environmentally-intensive programs. KLEE will execute every line of bytecode to maintain the stack, heap, and etc. for the environment.

To understand how KLEE works, a critical structure that KLEE implemented need to be introduced. It is `ExecutionState`. Snippet 2.2 shows the detail of `ExecutionState`. `pc` is the pointer to an instruction to be executed after the current instruction, `prevPC` is the pointer to the instruction which is currently executed, `stack` representing the current instruction stream, and `constraints` collects the constraints so far. Each `ExecutionState` is representing a valid path. KLEE define a set of operations for each of LLVM instructions. Modifying the data in `ExecutionState` is completed in those operations. Everytime KLEE hit an if-statement/while-loop/switch-statement or any other instructions that can possibly have a new path; it checks the feasibility of the instruction. If the path is feasible, KLEE will make a copy of the current `ExecutionState`, and added positive constraints and negative constraints to current `ExecutionState` and the duplicate. KLEE called this operation `fork`. After the `fork` operation complete, KLEE will assign new `pc` to the current `ExecutionState` and its copy based on their constraints. Then both of the current `ExecutionState` and its copy will be put back to the pool and waiting for next round of execution.

Talking at a high level, KLEE completes the analysis task in the following manner.

1. Select one `ExecutionState` from the pool.

2. Execute the instruction that `prevPC` pointed to.

3. Update the `ExecutionState` pool.

4. Repeat step 1.

Snippet 2.3 indicate this logic. `states` is the pool of `ExecutionState`, `searcher` is responsible for selecting state to execute from `states`. By default, KLEE will randomly select one `ExecutionState` from the pool, but it can be depth-first search or a breadth-first search as user's preference.

Snippet 2.2: ExecutionState

```
class ExecutionState{
...
KInstIterator pc;
KInstIterator prevPC;
stack_ty stack;
...
ConstraintManager constraints;
...
}
```

Snippet 2.3: KLEE's Main Logic

```
while (!states.empty() && !haltExecution) {
    ExecutionState &state = searcher->selectState();
    KInstruction *ki = state.pc;
    ...
    executeInstruction(state, ki);
    ...
    updateStates(&state);
}
```

While the `ExecutionState` reaches the end or `pc` points to an invalid address, KLEE will generate a testcase for that `ExecutionState` using the solver. The solver consumes the `constraints` of the `ExecutionState` and generate a testcase. After receiving the testcase from the solver. The `ExecutionState` object will be deleted.

2.3 Solver

By default, KLEE uses STP Ganesh and Dill [2007] as the internal solver, but Z3 De Moura and Bjørner [2008] can be an option. In general, STP solver emerges as a winner over Z3. By benchmarking [, `base64`, `chmod`, `comm`, `csplit`, `dircolors`, `echo`, `env`, `factor`, `join`, `ln` and `mkfifio`, STP consumes 1,713s to complete, and Z3 takes 3609s Palikareva and Cadar [2013]. Although KLEE has already got a robust solver, they still have a lot of optimization on the solver side to help the solver run faster. KLEE developed two optimization method on the solver side – Independent Constraint and Cache. Figure 2.3 shows they process of optimization developed by KLEE.

A query is consist of constraints. Among those constraints, some of them are related, and some of them are not. What KLEE does is they separate the query into little pieces. Each of the pieces contains a set of constraints that have no relation with other constraints. Chopping the query into smaller pieces can not only help the solver to obtain the result faster but also benefits the Cache optimization method as well.

Look up cache before throwing the query to the solver one the useful optimization at the solver side. KLEE maintain a `Map` in the engine. Using the constraint itself as the key and the counterexample that the solver generates as the value. Before asking solver for the counterexample, KLEE will first check if there is a solved one previously. The idea of this method is query `Map` is in constant time. Since the `Map` is maintained in the RAM, ideally, the look up speed should be way more faster than asking solver to generate a counterexample especially when the query is very complicate.

2.4 KQuery

KQuery is a textual representation of the constraint expressions and queries KLEE [2009]. Currently, KQuery is able to represent a quantifier-free formulas over bitvectors and arrays. It is designed to be easy read and write. Like many other programming languages require variables to be declared before using, bitvectors and arrays need to be declare before querying. A valid query should have the array declaration part and a query command part. Figure 2.4 shows the syntax of the

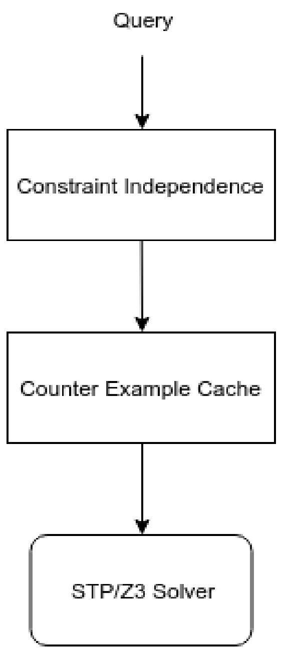


Figure 2.3: KLEE's Optimization Method

KQuery declaration. Figure 2.5 shows the syntax of a valid query command.

Combine both the declaration part and the query part can construct a valid query command in the format of KQuery. Snippet 2.4 is a valid query in KQuery format. Translating this query command back to plain English, it is "Having an array of size 10. The value at the index of 0 is Equal to 77". For this query command, the solver should return "`M000000000`". Note, since the constraint does not have any restriction of other values, the solver will put 0 on those indices by default.

```
array-declaration = "array" name "[" [ size ] "]" ":" domain "->" range "="
array-initializer
array-initializer = "symbolic" | "[" number-list "]"
number-list = number | number "," number-list
```

Figure 2.4: KQuery's Declaration Syntax

```
query-command = "(" "query" constraint-list query-expression [ eval-expr-list
[ eval-array-list ] ] ")"
query-expression = expression
constraint-list = "[" { expression } "]"
eval-expr-list = "[" { expression } "]"
eval-array-list = "[" { identifier } "]"
```

Figure 2.5: KQuery's Query Command Syntax

Snippet 2.4: KQuery

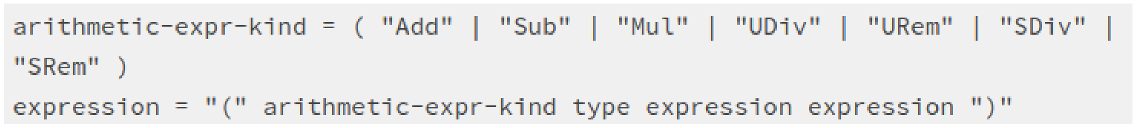
```
array foo[10]: w32 -> w8 = symbolic
(query [(Eq 77 (Read W8 0 foo))] false [] [foo])
```

Not limited to the comparisons, the expression also supports arithmetic operations, bitwise operations. Figure 2.6 shows the syntax of arithmetic operations. Figure 2.7 shows the syntax of bitwise operations. Figure 2.8 shows the syntax of the comparisons.

The meaning of the expression kinds in arithmetic operations are defined as below:

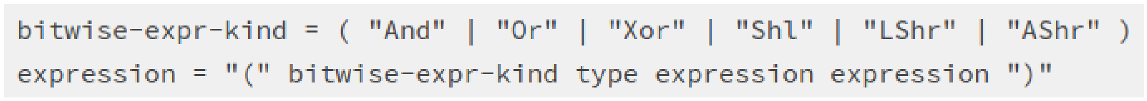
`Add`: add operation

`Sub`: subtract operation



```
arithmetic-expr-kind = ( "Add" | "Sub" | "Mul" | "UDiv" | "URem" | "SDiv" |
"SRem" )
expression = "(" arithmetic-expr-kind type expression expression ")"
```

Figure 2.6: KQuery's Arithmetic Operation Syntax



```
bitwise-expr-kind = ( "And" | "Or" | "Xor" | "Shl" | "LShr" | "AShr" )
expression = "(" bitwise-expr-kind type expression expression ")"
```

Figure 2.7: KQuery's Bitwise Operation Syntax

`Mul`: multiply operation

`UDiv`: truncated unsigned division. Undefined if divisor is 0.

`URem`: Unsigned remainder. Undefined if divisor is 0.

`SDiv`: Signed division. Undefined if divisor is 0.

`SRem`: Signed remainder. Undefined if divisor is 0. Sign of the remainder is the same as that of the dividend.

The meaning of the expression kinds in bitwise operations are defined as below:

`And`: and operation.

`Or`: or operation.

`Xor`: xor operation.

`Shl`: logical shift left. For example, `(Shl TYPE X Y)` means moves each bit of `X` to the left by `Y` positions. The `Y` right-most bits of `X` are replaced with zero, and the left-most bits discarded.

```
comparison-expr-kind = ( "Eq" | "Ne" | "Ult" | "Ule" | "Ugt" | "Uge" | "Slt" |
"Sle" | "Sgt" | "Sge" )
expression = "(" comparison-expr-kind [ type ] expression expression ")"
```

Figure 2.8: KQuery's Comparisons Syntax

`LShr`: logical shift right.

`AShr`: arithmetic shift right. Behaves as `LShr` except that left-most bit of `X` copy the initial left-most bit of `X`.

The meaning of the expression kinds of comparisons are defined as below:

`Eq`: equal.

`Ne`: not equal.

`Ult`: unsigned less than.

`Ule`: unsigned less or equal to.

`Ugt`: unsigned greater than.

`Uge`: unsigned greater or equal to.

`Slt`: signed less than.

`Sle`: signed less or equal to.

`Sgt`: signed greater than.

`Sge`: signed greater or equal to.

2.5 Related Work

The goal of this thesis is to address the the most significant issue of symbolic execution. A lot of people have done some fantasitc work to solve this problem before. I was inspired by one of the previous projects. In that project, the researchers modified KLEE's own data structure to serialize a query in order to send out to the remote solver Rakadjiev et al. [2015]. This method has fast query transfer speed. However, it limits the symbolic execution engine to be KLEE. It can't support multiple different symbolic execution tools working at the same time. To avoid this disadvantages, I decided to send out a query in KQuery format. Since KQuery is a textual representation of the queries and constraints, technically, all of the constraints can be represented in the format of KQuery. This method allows another engine to use the remote solver by sacrificing some of the query transfer time.

In my remote solver, I also adopt KLEE's optimization method – Constraint Independence and Caching. The main difference is that KLEE uses cache for the same program because KLEE is a synchronous engine. The solver is built-in in the engine. Once KLEE finishes the analysis task, the solver will be terminated and cache will be cleared. My solver is a remote solver, and it will not be terminated. Therefore, the cache will always exist. All of the queries can share the same cache map. I argue that this method can reduce the number of counterexample that generates by the solver.

Chapter 3

IMPLEMENTATION

In this chapter, I will introduce how I modified the KLEE to add an asynchronous mode to it as well as how do I wrapping the solver to use it as a standalone solver. Moreover, a new feature is added to KLEE. It is called “otherside testcase generation“.

3.1 Asynchronous Mode

As I mentioned in Chapter 2, KLEE is a synchronous symbolic execution tool. KLEE will wait for the result from the solver even though it is not necessary to do so. Figure 3.1 shows the control flow of KLEE. After terminating an `ExecutionState`, KLEE will ask solver to solve the constraints in that `ExecutionState` and wait for the result. Then checking if there is any `ExecutionState` waiting to be executed in the pool. However, KLEE will not use the result that it waits for. Previous research has pointed out that symbolic execution engine spends 90% of their time in solver constraints Rakadjiev et al. [2015]. Therefore, waiting for a result that it will not use anymore is wasting time. Alternatively, the solver can solve the constraints alone. KLEE is not helping the solver while it is waiting for the result. KLEE should keep executing other `ExecutionState` left in the pool. This improvement can significantly shorten the time for analyzing a program. In other words, the analysis task will finish much earlier than before. And the analyzing time relies on the engine's performance instead of solver's performance.

To transmit queries, I decided to send out the constraints in the KQuery format using HTTP `POST` method. The decision is under following consideration. KLEE has its special data structure to represent constraints. If a query needs to be sent out, it requires data serialization. Once the solver receives the query and deserializes it, it turns out to be KLEE's data structure. Under this setting, solver becomes "KLEE-ONLY" because the solver does not know about other engine's data structure and unable to handle those queries from other symbolic execution engines. Also, sending queries in KQuery format reduce the work for changing backend solver. All constraints solver has their own interface. Connecting symbolic execution engine's data structure to different solver's interface requires a solid understanding of data structure from the engine. However, if the solver only receives queries written in KQuery format, we just need to know how to connect KQuery to solver's interface if we need to replace the backend solver, even though the engine changes its own data structure in the future. Moreover, KQquery is a human readable representation. When we need to debug the solver, KQuery can make it easier. Receiving queries in KQuery has such advantages, yet nothing is

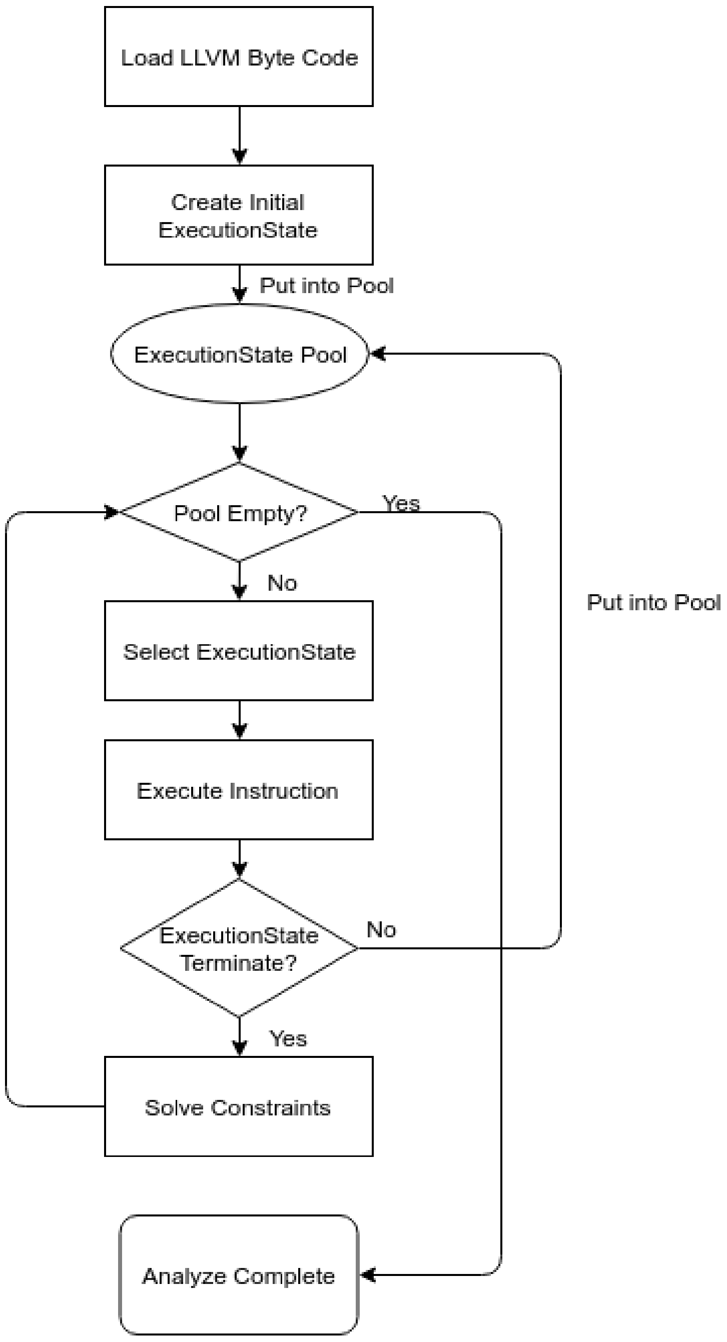


Figure 3.1: KLEE Original Control Flow

perfect. Presenting constraints in KQuery needs conversion time, and the process of conversion takes longer than the serialize the data structure of the symbolic engine. At the solver side, parsing the queries in KQuery into the data structure that can be used in solver's interface also takes longer than the data deserialization. After carefully considering the trade-off between speed and scalability, I decided to transfer the query command in KQuery format instead of serializing the original data structure of the symbolic execution tool.

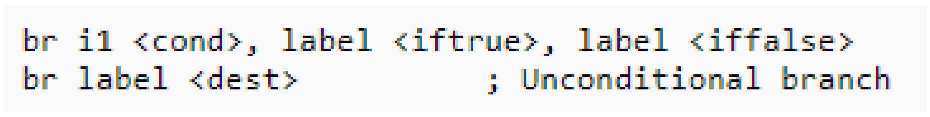
Once deciding how to transmit the queries to the remote solver, the rest of the implementation is straightforward. In KLEE's implementation of terminating an execution path, they first ask the solver to generate a testcase based on the constraints that the tool collected during the execution. Then delete the `ExecutionState`. In my implementation, I took out the constraints from the `ExecutionState`, then convert them into the format of KQuery and send it to the remote server via `HTTP POST` request.

In addition to providing KLEE an asynchronous mode, testcases generation for a concrete input of a program is also provided. Feeding in a concrete input, if the input went through some "forkable" statement, then generate the testcase to hit the "opposite" branch. For example, Snippet 3.1 is a program get the sign of an integer, feeding a concrete input "1", the program will return "positive". In this case, the "opposite" branches are the return of "negative" and "Zero." So feeding in a concrete input "1", the testcases generation for that concrete input will generate two testcases that contains a negative number and contains a 0 respectively in order to hit the other branch of the "forkable" statement.

Snippet 3.1: Get Sign

```
int get_sign(int x) {
    if (x == 0) return "Zero";
    if (x > 0) {
        return "Positive";
    }
    else {
        return "Negative";
    }
}
```

However, during the implementation, there is a problem need to be addressed, that is, once the input is concrete, KLEE will determine that there is no “forkable” in the target binary. Figure 3.2 indicate the syntax of LLVM `br` instruction. When a `br` instruction was hit, KLEE will check the `cond` of this instruction. The result is one the following, `True`, `False`, `Unknown`. `True` means the `cond` in this instruction is always true, for example, `if (1 == 1)`. `False` means the `cond` always be false, for example, `if (1 == 2)`. `Unknown` means the `cond` is can either be true or false. This situation happens only when there is a symbol involved in the `cond`. The term “forkable” means the component of an instruction is not concrete or there are two destinations for this instruction. `br`, `switch`, `alloc` and `load` are some potential forkable instructions.



```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>          ; Unconditional branch
```

Figure 3.2: LLVM Br Instruction Syntax

Without knowing which line of instructions is forkable, generating testcases for the opposite branch is impossible. Potential forkable instructions can appear many times in a normal program, but only a few of these instructions are involved with the target symbol. If the instruction is not "forkable," we have no idea if the instruction is involved with the symbol. More importantly, if an instruction cannot be forked, there is no opposite branch of that instruction. To solve this issue, I decided to use path log to guide the symbolic execution. The target binary will be run twice. The first time, a concrete input will be fed. During the execution, all of the potential forkable instructions will be logged. In my implementation, I will log four types of instruction I mention previously. For `br` instruction, instruction id and the result of `cond` will be logged. For `switch` instruction, instruction id and index of the successor will be logged. For `alloc` instruction, instruction id and allocation size will be logged. For `load` instruction, instruction id and the offset will be logged. The second run will run the program with the symbol. While exploring the path, the path log that collected during the first time can guide the execution. All the log information will be kept in a queue. If hitting the potential forkable instructions, matched the instruction id and the `cond` result/successor id/allocation size/offset with the path log in queue. If the instruction cannot be forked, the log and the execution should be the same. If it is forkable, use the log information to replace those corresponding components in the instruction. This step ensures the execution will choose the same path as the concrete input do. In `br` instruction, what will be added to constraints is based on the log. If the log is false, the negation of `cond` will be added. Otherwise, `cond` will be added. To produce the testcase that hit the other side, all we need to do is to connect the constraints (not including the newly added one) and negation of `cond` or `cond` regarding the log with the `AND` logic operator. Then we can ask solver to solve the constraints to produce a testcase that hit the opposite side. Figure 3.3 describes the workflow of this feature.

3.2 Remote Solver

The remote solver has two part. A server that handles the HTTP request it receives and a Solver solve the queries received by a solver. Figure 3.4 illustrates the architecture of the remote solver. Once a `String` arrives at the Server, it will first create a file with a unique name and write the `String` it receives to the file. Then append the absolute path of that file to another file called "Monitor" file. This file is responsible for communicating between the server and solver.

The solver is listening to "Monitor" file using `inotify`, which is a Linux kernel subsystem that acts to extend filesystems to notice changes to the filesystem and report those changes to applications contributors [2017]. There are two threads in the solver. The first one is the watching module, responsible for retrieving the new appended file and push it into a thread-safe queue waiting to be solved. The other thread is the solver itself. If the queue is not empty, poping a file path once a time. Then read the content of that file, parse the content. Thanks to KLEE's implementation, they have a parse that can parse a `String` in the format of KQuery and can be used in the STP/Z3 solver directly. After parsing, the solver will check the cache map to see if there is any hit. If so, get the result from the

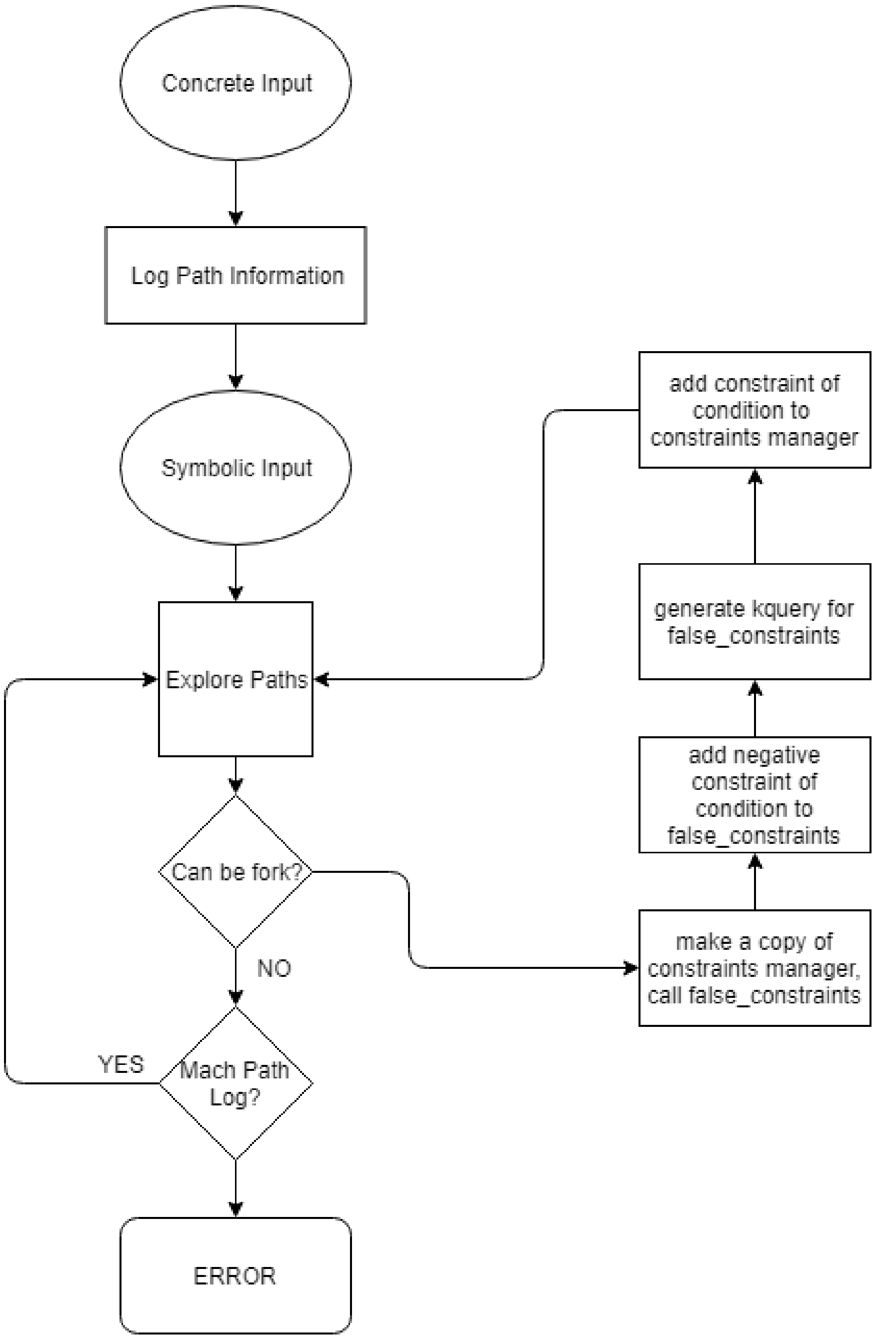

Concrete Input
Log Path Information
Symbolic Input
Explore Paths
Can be fork?
NO
YES
Mach Path Log?
ERROR
make a copy of constraints manager, call false_constraints
add negative constraint of condition to false_constraints
generate kquery for false_constraints
add constraint of condition to constraints manager

Figure 3.3: Workflow of Generating Testcase for "Opposite Brance"

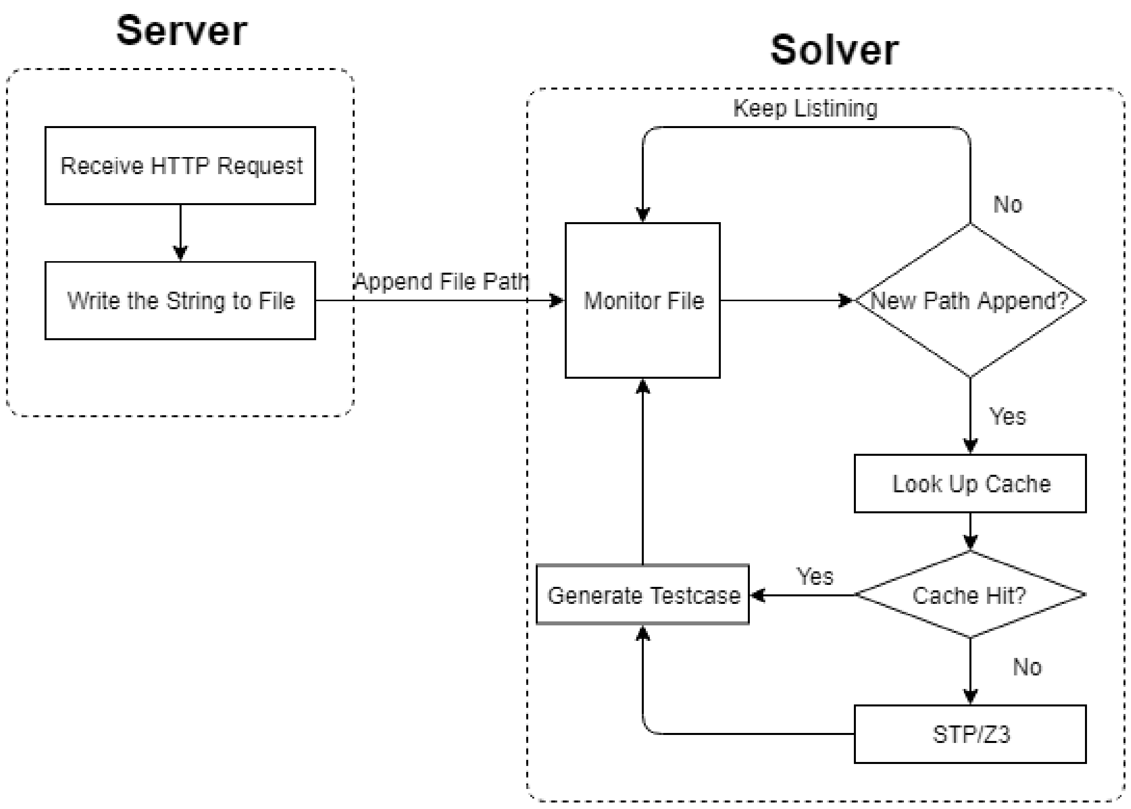


Figure 3.4: Remote Solver Architecture

cache map and save it as testcase. Otherwise, pass it to the solver and insert the `<constraints, result>` pair to the cache map for future use. Also, this solver will support the constraints independence optimization.

3.3 "Otherside" Testcase Generation

As we all know, symbolic execution is used to discover all the feasible paths of a binary, however, in some situation, we do not need to know all the paths but just some paths. To address this problem, I added a new feature to KLEE and it is called "Otherside" testcase generation.

The term "otherside" is refer to control flow direction. In the programming language level, a `if` statement may have many branches. However, in assembly level, all the "if" statements have just two branches – either "True" or "False". So, if a `br` instruction is "True", its otherside is "False". Otherside testcase generation aims to generate testcases that covered the otherside of the branches for a given input. Code snippet 3.2 shows a program that determine a number is a even number. For the given input "2", the otherside testcase generation will generate a testcase "1" since the this testcase will cover the otherside of the "if" branch of the program.

Snippet 3.2: Is Even

```
bool isEven(int n) {
    if (n % 2 == 0) {
        return True
    }
    else {
        return False
    }
}
```

To achieve this goal, we need a concrete input for the target binary as the given input. During the execution of the given input, we need to log the path that it travels through. Then, we reset the program and make the input as symbol and use the log we collected to guide the symbolic execution.

In KLEE, instructions that will cause a "fork" action is the log we need to collected – they are `br`, `switch` and `alloc`. And during the second run, we will use symbol as the input of the target binary. When it needs to fork, the path log that collected by the first run will be used to guide the execution. At the same time, the opposite condition will be concatenated to constraints using "AND" logic operator and send to the remote solver for testcase case generation purpose.

After two runs, we will have testcase that can hit all the opposite branches of the original concrete example.

Chapter 4

EXPERIMENTAL EVALUATION

In this chapter, the evaluation of the implementation will be provided. There are total three parts of the evaluation. First, exploration time for an application with the asynchronous mode comparing to synchronous mode. Second, the number of counterexamples generate by solver with the shared-cache solver. Third, with the help of the shared-cache solver, the time reduce for analysis the some applications using DFS (depth-fist-search) strategy.

4.1 Exploration Time

In this section, I will compare the exploration time for some application from GNU `COREUTILS`. KLEE will explore 20,000 paths for that application. If the application does not have 20,000 paths, then find all the feasible paths for the application. Each of the application will be run ten times. Table 4.1 shows the result of this experiment. This experiment is running on an Amazon Web Service's `t2.xlarge` machine, with four virtual CPUs and 16 gigabytes of RAM. The CPU's clock speed is up to 3.0 GHz.

Snippet 4.1 is the symbolic arguments using for KLEE in this experiment Cadar et al. [2008]. `posix-runtime` was enabled for making the command line arguments of the target binary symbolic.

Snippet 4.1: Symbolic Arguments

```
--posix-runtime
--libc=uclibc
--search=dfs
--sym-args 0 1 10
--sym-args 0 2 2
--sym-files 1 8
--sym-stdin 8
--sym-stdout
```

Application	Mode	Average	Max	Min	Reduce
test ([)	Async	233.36	337.32	173.26	21.92%
	Sync	298.88	394.55	253.24	
basename	Async	83.60	84.71	82.78	75.61%
	Sync	342.73	344.42	340.14	
chmod	Async	230.31	242.91	225.66	62.30%
	Sync	610.96	614.24	607.92	
arch	Async	1382.86	1392.90	1373.03	63.59%
	Sync	3797.66	3901.401	3753.34	
chgrp	Async	243.45	248.22	237.51	61.75%
	Sync	636.55	640.208	633.094	
ls	Async	1913.09	1943.21	1879.83	20.50%
	Sync	2406.32	2409.52	2403.307	

Table 4.1: Result of Comparing Async and Sync (unit: second)

In this experiment, it is evident that the asynchronous mode can improve the path exploration time significantly. The average time for `[` was reduced from 298.88 seconds to 233.36 seconds. It shortened the exploration by 21.92%. For `basename` and `chmod`, the average exploration time is shortened by 75.61% and 63.59% respectively. `chgrp` was improved 61.75%, and `ls` was enhanced 20.50%

In conclusion, original KLEE spend most of its time on I/O, writing the test-cases to hard drive, and waiting. With the help of the asynchronous mode, the exploration time can be shortened. The **Hypothesis 1** holds.

4.2 PoC of Share-Cache Solver

The prove of concept of the Share-Cache Solver runs in the following settings. The target binary is from GNU `Coreutils` package. The symbolic arguments remain the same as the previous experiment. Each of the application will be run twice. The solver will be terminated (cache clear) for every program. The purpose of the first run of each binary is "building cache map." The second run of each binary is checking if the solver use caches to generate testcases. The solver will not be terminated between the first and second run of the same binary. The result of this experiment is shown in Table 4.2.

As shown in the result, the second run will not generate any counterexamples because the first run has already built the cache. All the queries sent during the second run are exactly the same as the first run. Therefore, there is no cache missed during the second run. In other words, two runs of each program shared cache with each other. The experiment also indicates that solver the cache can

Application	CEx Generated (1st run)	CEx Generated (2nd run)
test ([)	517	0
arch	294	0
chgrp	190	0
chmod	285	0
chcon	189	0
base64	133	0
basename	300	0

Table 4.2: Number of Counter Examples Generated

help reduce the workload of the solver, and cache can be used in two different runs.

4.3 Share-cache Solver Analyzing Binary

In this section, I will use the asynchronous KLEE to analyze some binaries in GNU `Coreutils` package. First, I will run each binary for one time to collect the number of counterexamples generated as well as the solving and cache lookup time. The Share-cache solver will be terminated between each run. Then I will run those applications together. At this time, the Share-cache solver will not be terminated. The purpose of this experiment is to check if the cache can be used between different program. The target binaries are `[`, `arch`, `base64`, `basename`, `chcon`, `chgrp`, and `chmod`.

These binaries will be executed in the following order: 1. `[` 2. `arch` 3. `basename` 4. `base64` 5. `chcon` 6. `chgrp` 7. `chmod`

Table 4.3 shows the solving time, cache lookup time and number of counterexamples generated during single execution. Table 4.4 shows the result of analyzing the target binaries without terminating the shared-cache solver.

Application	Queries	Caching Time	Solving Time	Overall Time	Num. Cex
test ([)	1788	2.87	10.19	13.06	517
arch	20000	971.27	0.88	972.15	294
base64	20000	42.38	1.08	43.46	133
basename	14071	3.38	0.56	3.94	300
chcon	20000	11.42	1.00	12.42	189
chgrp	20000	14.74	1.61	16.35	190
chmod	20000	10.91	1.07	11.98	285
Total		1056.97	16.40	1073.36	1908

Table 4.3: Caching and Solving Time For Analyzing a Single Binary (unit: second)

Application	Queries	Caching Time	Solving Time	Overall Time	Num. Cex
test ([)	1788	2.88	10.23	13.11	518
arch	20000	3189.86	0.43	3190.29	75
basename	14071	6.71	0.23	6.94	98
base64	20000	38.77	0.77	39.54	71
chcon	20000	9.11	0.05	9.16	98
chgrp	20000	16.75	0.22	16.97	50
chmod	20000	9.21	0.05	9.26	86
Total		3273.28	11.98	3285.26	996

Table 4.4: Caching and Solving Time For Analyzing 7 Binaries (unit: second)

Without the cross-binary cache share solver, the total number of testcases generated by the solver is 1,908 for analyzing all these seven target binaries. With the cross-binary cache share solver, this number reduced from 1,908 to 996. That's a 47.83% reduction. With such reduction in the number of counterexamples generated by the solver, the solving time (time that spend in STP/Z3 solver) dropped from 16.40 seconds to 11.98 seconds. The cross-binary shared-cache solver spends 26.95% less time in STP/Z3.

Cache lookup time get hurts from the increase of cache map size because of the cache look up mechanism. The solver follows the following cache lookup policy. First, check if the expression exists in the cache map. If yes, then it is a direct cache hit. If not, iterate through the cache map, try all of the answers to see if any of these solutions fit the expression. If yes, then it is an indirect cache hit. Otherwise, it is a cache miss. The worst case of time complexity for the strategy of cache look up is $O(n)$ since the solver needs to iterate through the whole map to see if there is any answer satisfies the expression if there is no direct hit. Due to the increase of cache size, the cache map size will go up. As we can see, these binaries more or less have indirect hits, therefore the cache look up takes longer.

If the remote solver does not allow the indirect cache hit, the number of counterexample will increase a lot. Take the example of `arch`, the time for solving indirect hits using STP/Z3 takes longer than the current situation (first 7000 testcase spends around 1500 seconds in STP/Z3, about 21 second in cache lookup). So there is a trade off here. Allowing indirect hits will cause increase of the cache lookup time but decrease the number of counterexamples generated. Disable indirect cache hit will cause the number of counterexamples and the time for solving those indirect hits in STP/Z3 probably takes longer.

4.4 Cache Strategy Improvement

As the result shown above, current caching strategy is not that perfect. Therefore, a new strategy has been added to the remote constraints solver, that is let the solver race with the cache lookup.

In the original implementation, once the remote constraint solver receive the kquery, it will start lookup the cache in the cache pool. The original caching look up strategy has two steps. First check if there is a direct cache hit. Second, if there is no direct cache hittry all the result in cache pool to see if any result meets the constraints. Therefore, the worst situation for cache lookup is $O(n)$. With more and more cache gets into the cache pool, this time will increase.

Moreover, in reality, some cache lookup time will slower than solve the constraints directly. Therefore, in this improved cache lookup strategy, the remote solver have multithreads. One of them is responsible for cache lookup, and the other one is responsible for solve the query directly. And the solver will return the result whoever comes back first.

When testing the `gnu util` package, the average time form solving a query is 0.01 seconds, and the average time for solving query directly is 0.002 second. Under this strategy, the expensive “indirect cache“ strategy is abandoned since solving the query is much faster than finding a indirect cache hit.

Chapter 5

SUMMARY AND CONCLUSION

5.1 Summary

In this research, an asynchronous KLEE was implemented also a remote solver was built to work with the asynchronous mode. Compare to the original KLEE, the major difference of the asynchronous KLEE is that it has an asynchronous solver. Previously, when an `ExecutionState` was terminated, it will trigger a function called `ProcessTestcase`. In this function, KLEE will extract the constraints from the `ExecutionState`, parse it to a data structure that can use the interface that provided by STP/Z3 solver. Then KLEE will wait for solver's response. After receiving the result from solver, KLEE will wrapped up the answer and write the testcase to disk. What asynchronous KLEE does in `ProcessTestcase` is, it extracts the constraints from the `ExecutionState`, converted to a `String` in KQuery format, which is a textual representation of constraints. Instead of sitting tight and waiting for solver's response, the asynchronous KLEE sends out a string of query command using HTTP `POST` method to a remote solver. After sending the query out, the asynchronous KLEE immediately start to execute next `ExecutionState` in its `ExecutionState` pool.

The asynchronous KLEE optimize the procedure in producing testcase for a `ExecutionState`. In the original KLEE, it wastes a lot of time in generating testcases. Through the experiment in chapter 4, it has already been proved that the asynchronous KLEE can shorten the time for exploring paths significantly when comparing to the original version of KLEE.

A remote solver was also been wrapped up to serve the asynchronous KLEE. There are two parts of the remote solver, the HTTP server and an STP/Z3 solver. The HTTP solver is responsible for receiving query command `String` from asynchronous KLEE. Then, the server will create a file and write the query command it just receives to that file. Also, the absolute path of that file will be appended to a file called "monitor file." It performs like a queue because the solver is watching the change of this file using Linux kernel subsystem called `inotify`. Once a new path was appended, the solver will know a query command just came in and is waiting for the result.

The solver's responsibility is solving constraints and producing testcases. When there is a path appended to the "monitor file," the solver will read the contents of that file and parse the file to a data structure that can communicate with STP/Z3 solver. Here, some optimization will be performed before the query gets into the STP/Z3 solver. First of all, the query will be chopped into small pieces. Those pieces are independents, in another word, those small pieces is also a valid expression. This step was called "independence constraints." Second, the solver will check if the expressions have been solved previously. To accomplish this goal, the solver will save a cache map in RAM. The key of the cache map is the expression, and value is the answer. If the answer exists, the expression will not be passed into the STP/Z3 solver. Otherwise, the `expression, answer` pair will be inserted into the cache map after STP/Z3 solves the expression. Unlike KLEE's implementation, the solver keeps alive until the user terminates it. Therefore, sharing cache cross binary is possible. In KLEE's implementation, the solver instance will be terminated when execution completed and the cache will be deleted. If we use KLEE to analyze the same application, it still needs to solve the expression instead of pulling out the answer directly from the cache. This situation does not happen in the remote solver. Not only the same binary can share cache, but also another program since the remote solver does not know which program is being analyzed. All it knows are just expressions. Once the there is a same expression in the cache, it will pull out the answer from the cache instead of solving it. The experiment in Chapter 4 has proved that the solver share caches cross binaries.

5.2 Future Work

Although the asynchronous KLEE has some advantages on shortening the exploration time on the engine side as well as save some extra work on the solver side, it still has many places to work on in the future. Using cache in the solver is a perfect idea, but how to save the cache can be optimized a little bit more. In the current implementation, the cache map is saved in RAM. The access time of the RAM is fast. However, when the solver keeps working, the size of the cache map is growing. Ultimately, the machine will probably run out of memory. Also, if the solver died accidentally, such as having power outage issue, the cache will be unrecoverable. To address these problems, the cache should be stored in another place. For example, in a file or in a database. And this kind of action will sacrifice the access time.

Bibliography

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408795. URL `http://doi.acm.org/10.1145/2408776.2408795`.

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. URL `http://dl.acm.org/citation.cfm?id=1855741.1855756`.

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1066–1071, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985995. URL `http://doi.acm.org/10.1145/1985793.1985995`.

Wikipedia contributors. Inotify — wikipedia, the free encyclopedia, 2017. URL `https://en.wikipedia.org/w/index.php?title=Inotify&oldid=810255382`. [Online; accessed 31-March-2018].

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011. ISSN 0001-0782. doi: 10.1145/1995376.1995394. URL `http://doi.acm.org/10.1145/1995376.1995394`.

Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73367-6. URL `http://dl.acm.org/citation.cfm?id=1770351.1770421`.

James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL `http://doi.acm.org/10.1145/360248.360252`.

KLEE. The reference manual for the kquery language, 2009. URL `https://klee.github.io/docs/kquery/`. [Online; accessed 30-March-2018].

Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 53–68, New York, NY, USA, 2013. Springer-Verlag New York, Inc. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_3. URL `http://dx.doi.org/10.1007/978-3-642-39799-8_3`.

Emil Rakadjiev, Taku Shimosawa, Hiroshi Mine, and Satoshi Oshima. Parallel smt solving and concurrent symbolic execution. In *Proceedings of the 2015 IEEE Trustcom/BigDataSE/ISPA - Volume 03*, TRUSTCOM-BIGDATASE-ISPA '15, pages 17–26, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-7952-6. doi: 10.1109/Trustcom-BigDataSe-ISPA.2015.608. URL `http://dx.doi.org/10.1109/Trustcom-BigDataSe-ISPA.2015.608`.