KLEE-Reach: a KLEE version for Guiding Symbolic Execution with A-star
Developer Manual

Guilhem Ardouin
guilhem.ardouin@enseirb-matmeca.fr

April 12, 2024

Contents

LABORATOIRE
BORDELAIS
DE RECHERCHE
EN INFORMATIQUE
LaBRI

1 Introduction

KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure. Although KLEE was not initially designed to perform reachability analysis, it can be used to support new search heuristics for symbolic execution inspired by A-star [1]. These search heuristics have been specially designed to tackle the *line reachability problem* better than exiting strategies.

This document describes the development of KLEE-Reach, an updated version of KLEE that supports reachability analysis and the new A-star search heuristics. Section 2 describes how KLEE has been adapted to perform reachability analysis. The implementations of the search heuristics are detailled in section 3. Section 4 explains how distances are computed for the priority function used in A-star search heuristics.

2 Using KLEE for reachability analysis

KLEE is not a software that was initially developed to analyse reachability. However, it does have all the necessary tools. In this section, we will show how we have adapted KLEE's functionality to perform reachability analysis.

2.1 Symbolic execution for reachability analysis

KLEE's missing functionality for performing reachability analysis is the ability to define a target in the program that will be executed symbolically. We have therefore implemented a new instruction that does precisely that: `klee_reach()`. Implementing such an instruction will then enable a specific behaviour to be defined when the target is reached during symbolic execution, *i.e.* stopping the program and signalling to the user that the target has been reached.

It is already possible to ask KLEE to stop whether encountering a special instruction, which usually triggering an error. To do so, the user has to specify `--exit-on-error-type=<error_type>` in the command line. Even if reaching the target is not an error case, this native functionally could be useful to do reachability analysis. Therefore, to trigger a specific behaviour when KLEE reach the target, an error type will be associated when the instruction `klee_reach()` is executed.

2.2 Implementation of `klee_reach()` instruction

To implement the new instruction in KLEE, various source files need to be modified. The following section shows the changes that have been made.

Firstly, the prototype of the new `klee_reach()` function is indicated in the file `include/klee/klee.h`. Then, this function is implemented in

`runtime/Runtest/intrinsics.c`. This function only performs an `abort()` instruction, in order to stop the program when the target is reached. The new instruction is also listed in `modelledExternals` in `tools/klee/main.cpp`.

To specify a specific behavior, linked to an error type, it is necessary to declare and implement a handler, `handleReach` in `lib/Core/SpecialFunctionHandle.{h,cpp}`:

- in the `.h` : `HANDLER(handleReach)` has been added,
- in the `.cpp` : `handleReach` has been implemented and `addDNR("klee_reach", handleReach)` has been added.

The purpose of this handler is to execute a particular method to notify the `Executor` that the instruction `klee_reach()` has been executed. `handleReach()` simply calls a new method, `terminateStateOnReach()`, on the `Executor`.

`terminateStateOnReach()` alerts the user that the target has been reached and halts KLEE. This termination is associated with a new termination type: `Reach`. This termination type has been declared in `lib/Core/TerminationTypes.h`. Lastly, this new termination type is listed in the command line options list `ExitOnErrorType`.

3 Guiding symbolic execution with A-star

3.1 How search heuristics work in KLEE

In KLEE, a step of the symbolic execution is the selection of a state in the worklist. This state represents a branch in the symbolic execution tree and contains a range of information, including the LLVM instruction that will be executed if the state is selected. Whenever a state contains an instruction which is a branch concerning a symbolic variable, the state is forked, and each copy takes a direction in the symbolic execution tree.

A search heuristic is modelled by a `Searcher` in KLEE. There exists different searchers, all of which implement the abstract class `Searcher`. The differences lie in the type used to represent the worklist, and in the way states are selected. A `Searcher` maintains the current worklist `states` and directs symbolic execution.

Please note that a state in KLEE has a depth, which is the number of times KLEE branched for this state. In this document, the *depth* of a state refers to the depth in the symbolic tree. If n is an instruction and n' is the following instruction, thus $\text{depth}(n') = \text{depth}(n) + 1$.

3.2 Implementation of A-star search heuristic

Implementing a new search heuristic in KLEE simply involves the following steps:

- write a new `Searcher` as a child of the abstract class `Searcher`,

- add the "Searcher type" in `CoreSearchType` in `Searcher.h`,
- list the new search heuristic in the command line options `ExitOnErrorType` in `UserSearcher.cpp`.

In the implementation of `AStarSearcher`, the worklist is represented with a Fibonacci heap: we have used the one implemented in the Boost library[1]. This heap takes `HeapElement`, a pair of (`int`, `ExecutionState*`) where the integer is the priority of the element. To simplify use of this structure, a `FibonacciHeap` wrapper has been written in `include/klee/ADT`. This makes it easier to avoid using Boost and rewrite a Fibonacci heap according to Boost's abstract data type.

When a state in pushed into the heap, a `FibonacciHeap::handle_type` element is returned. This element is stored in a map `handlesMap`, as it will be useful to identify the heap element later, for updating or deleting purposes.

An element's priority is computed by the method `computePriority()`, according to the Equation 1 [1]:

$$PrioASTAR(n,t) = \text{depth}(n) + h_t(n.\texttt{instr}) \tag{1}$$

where n is a state in the worklist, depth(n) the depth of n and $h_t(n.\texttt{instr})$ is the estimated distance between the instruction $n.\texttt{instr}$ and the target t. The depth value is obtained with the instance of `StateInformation`, which is presented later, and the distance is returned by `distanceMap[n]`.

To collect additional information regarding the states in the worklist, we have introduced a new class `StateInformation`. Each A-star searcher has a map which associate a state with an instance of `StateInformation`.

For a state s, `StateInformation` stores:

- the line number of s current instruction,
- the depth of the state (as defined in section 3.1),
- an associative map between an instruction line number l and the number of execution of l for this state,
- a map storing for a LLVM line number l the *elementary depth* of the state, which is the depth at which l was executed for the first time.

To keep an instance of `StateInformation` for a state s up to date, the searcher updates s information with using a method `updateStateInformation()`: it checks which instruction is executed for s and increments all previous information according to this instruction. When a state is forked, the method `copyStateInformation()` is triggered, copying all information from the parent state to the child state.

[1] https://www.boost.org/doc/libs/1_84_0/boost/heap/fibonacci_heap.hpp

In the Equation 1, $h_t(n.\texttt{instr})$ is the estimated distance between the instruction $n.\texttt{instr}$ and the target t. The distance computation is carried out by a Python utility program, which is described in section 4. However, these distances are collected by KLEE and stored in an association map `distanceMap`.

`selectState()`, `empty()` and `printName()` are implemented as other searchers' methods. The `update()` method is much richer than the other methods and is presented in Listing 1.

```
void AStarSearcher::update(ExecutionState *current, const std::
    vector<ExecutionState *> &addedStates, const std::vector<
    ExecutionState *> &removedStates) {
  // collect distance map
  if (current == 0) {
    distanceMap = parseDistFile();
  }

  // insert states
  for (const auto state : addedStates) {
    copyStateInformation(current, state);
    updateStateInformation(state);

    HeapElement newState { computePriority(state), state };
    FibonacciHeap::handle_type handle = states.push(newState);
    handlesMap[state] = handle;
  }

  // update current
  if (current != 0) {
    updateStateInformation(current);
    HeapElement updatedState = { computePriority(current), current
    };
    states.update(handlesMap[current], updatedState);
  }

  // remove states
  // ...
}
```

Listing 1: `update()` method for the `AStarSearcher`

Firstly, the distances are collected when `update()` is called for the first time (when `current` equals 0). Then, for each new state to added, information from the current state is copied, and after the new state is updated. After all new states have been added, the current state is updated in the Fibonacci heap.

Please note that new states must be inserted *before* updating the current state, as information relating to the current state must not be updated until it has been copied for all child states.

For debugging purposes, a method `printWorklist()` has also been implemented to display the worklist at each iteration.

3.3 Another A-star search heuristic towards the unknown

In the paper [1], another more efficient method is presented: `AStar2Searcher`. The implementation of `AStar2Searcher` is very similar to the implementation of `AStarSearcher`. The only differences are the priority function used and the additional processing required to compute this function.

The priority function is presented in Equation 2:

$$PrioASTAR2(n,t) = g \cdot \lambda(\mu, g) + h_t(n.\texttt{instr}) \quad \text{where } \lambda(\mu, g) = \begin{cases} 0 & \text{if } \mu \leq g \\ \frac{\log_{10}(\mu-(g-1))}{10} & \text{otherwise} \end{cases} \tag{2}$$

where g is the elementary depth and μ the number of times n.`instr` has been executed for the state n. This data is stored in an instance of `StateInformation` for each state, as explained in section 3.2.

As the operations are the same as for `AStarSearcher`, `AStar2Searcher` inherits of `AStarSearcher`. Some additional processing is added, such as the collection of g or μ in `StateInformation`.

4 KReachDist, a Python program to compute distances

KLEE runs on LLVM bitcode files compiled directly from a high-level source file, such as C or C++. It then uses the LLVM file generated from this bytecode file to perform symbolic execution. While executing, KLEE's engine selects a state from its worklist and executes the corresponding LLVM instruction. For instance, selecting the same state until its termination is equivalent to following the program flow on this specific branch, and therefore executing LLVM instructions line by line. Thus, distances should be computed from the LLVM file that KLEE will use for symbolic execution. This file is generated when KLEE runs the program. This section focuses on how these distances are computed from such a LLVM file.

An ideal distance between two instructions is the minimum length of all executions from a start line to a destination line. Because this distance is not computable, we will make do with an underapproximation of the distance by only considering the control flow and ignoring variables values.

Distances are computed by a utility Python script in a `.dist` file. This file is required by KLEE-Reach to run A-star search heuristics. The choice to precompute distances independently of KLEE allows the user to generate the `.dist` file with the method of their choice.

4.1 Representing the LLVM file

The first step of the supplied Python script is to parse the LLVM file generated by KLEE from the source code. This analysis provides various information on the

program flow, such as jumps between branches, which are of particular interest for computing distances. Our goal is to represent this information in order to use it to determine the distance between two instructions in the execution of the program.

To illustrate our point, let's consider the function `foo()` of a program `foo_bar.c` in the Listing 2.

```
#define VALUE 144

int foo(int x) {
    int i = VALUE;
    while (i > x) {
        i--;
    }
    return i;
}
```

Listing 2: `foo()` function in `foo_bar.c`

The distance script will work on LLVM files. Therefore, we also introduce the function `foo()` as defined in `foo_bar.ll`, the LLVM compilation of `foo_bar.c`, in Listing 3.

```
define i32 @foo(i32 %0) #0 !dbg !10 {
    %2 = alloca i32, align 4
    %3 = alloca i32, align 4
    store i32 %0, i32* %2, align 4
    call void @llvm.dbg.declare(metadata i32* %2, metadata !14,
    metadata !DIExpression()), !dbg !15
    call void @llvm.dbg.declare(metadata i32* %3, metadata !16,
    metadata !DIExpression()), !dbg !17
    store i32 144, i32* %3, align 4, !dbg !17
    br label %4, !dbg !18

4:                                                ; preds = %8, %1
    %5 = load i32, i32* %3, align 4, !dbg !19
    %6 = load i32, i32* %2, align 4, !dbg !20
    %7 = icmp sgt i32 %5, %6, !dbg !21
    br i1 %7, label %8, label %11, !dbg !18

8:                                                ; preds = %4
    %9 = load i32, i32* %3, align 4, !dbg !22
    %10 = add nsw i32 %9, -1, !dbg !22
    store i32 %10, i32* %3, align 4, !dbg !22
    br label %4, !dbg !18, !llvm.loop !24

11:                                               ; preds = %4
    %12 = load i32, i32* %3, align 4, !dbg !27
    ret i32 %12, !dbg !28
}
```

Listing 3: `foo()` function in `foo_bar.ll`

All information regarding the function's flow appears in the Listing 3: here, following the code line by line leads to `br` instructions that indicate where to

resume the execution. Eventually, the last instruction, a `ret` instruction, is reached and indicates the end of the function.

For this work, a function is considered *defined* in LLVM if its definition, with an explicit set of instructions, is present in the LLVM file. A function which is only declared in the LLVM file, *i.e.* a function without a block of instructions, is not considered to be *defined*. These functions usually appear in `declare` statements. For instance, `@llvm.dbg.declare` is not defined in `foo_bar.bc` but declared. Keeping track of defined LLVM functions is important for the following steps, and we will come back to this later. For the moment, we only take advantage of the first pass through the source code to build a structure that tells us whether a function is a defined LLVM function or not.

A way of representing the program flow expressed in a LLVM file, which keeps in mind the program flow, is to use a specific type of graph: *control-flow graphs* (CFG). In this project, we choose to use them to represent the LLVM parsed file. Formally, a CFG is a directed graph where each node is called a *basic block*, which is a code sequence with no jump. Directed edges are used to represent jumps in the control flow, *i.e.* jumps from one basic block to another: this structure represents all possible paths for a program.

The CFG of the function `foo()`, defined in the Listing 3, and obtained by our Python program, is represented in Figure 1.

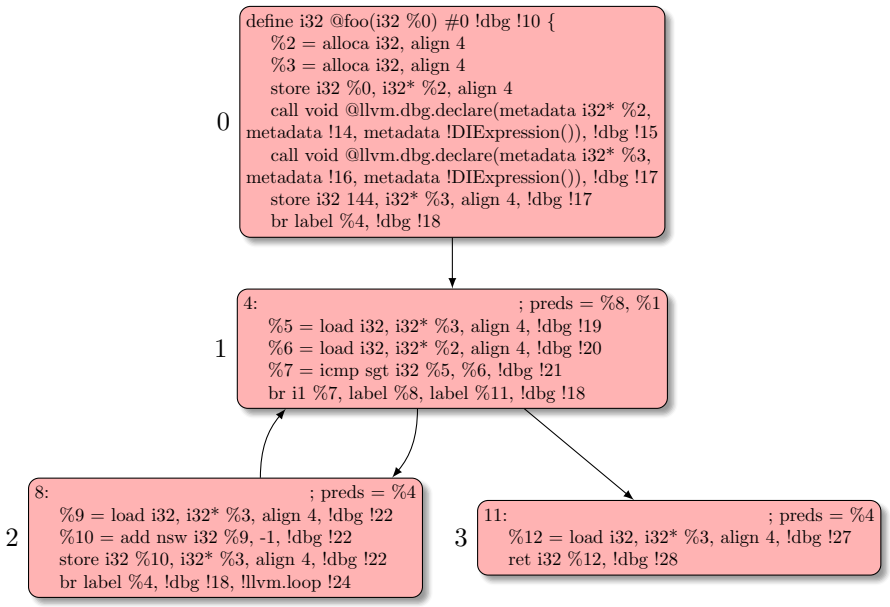

Figure 1: `foo()`'s control flow graph

When KLEE symbolically executes `foo_bar.bc` and runs `foo()`, the execution follows the CFG, entering the basic block 0 and exiting the basic block 3.

In KReachDist, a CFG is defined as a list of basic blocks where each basic block is identified by a unique identifier, starting at 0 and incremented for each new basic block. Some basic blocks contain a label: labels are used to identify the part of the code we jump on when we follow a `br` instruction. Because some `br` instructions refer to labels that are not yet seen in parsing, it is not possible to represent the edges of the CFG in a single pass. For instance, on line 4 of the Listing 3, the basic block corresponding to label `%4` has not yet been constructed. It is therefore not possible to guess what the identifier of this basic block will be. That is why we first construct a mapping table between the labels in the LLVM code and the identifiers of the corresponding basic blocks. This table is stored in the CFG structure and will be used in a second step to build the edges. Lastly, a CFG is identified by a unique identifier and name (both of which are useful depending on how we want to identify it).

We have defined a basic block as a straight-line code sequence where the last instruction is either a LLVM *terminator instruction*[2] or a call to a defined function in the LLVM file. The representation of the basic block includes a list of all LLVM instructions within it, where we represent a LLVM instruction by a structure containing an integer as the line number of the instruction and a string as the content of the instruction itself. The relationships between basic blocks, *i.e.* the edges of the CFG, are represented by a list of successors for each basic block. As explained earlier, this list is mainly established during a second pass[3], when we have all the information concerning the association between a label and a basic block. Only direct successors, which are obtained when a basic block is ended by a `call` to a defined LLVM function, are identified during the first pass. Figure 2 shows a basic block's direct successor.

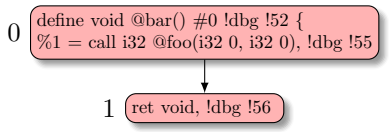

Figure 2: A direct successor of basic block 0

A list of predecessors is also maintained and used later for computing the transposed CFG. Lastly, some LLVM lines are ignored by KLEE but collected by our algorithm for debugging purpose. We call them *ignored instructions* and we store their number in a variable for each basic block. This allows us to define the *size* of a basic block: this is the number of LLVM instructions executed by KLEE. In other words, it is the *length* (the total number of stored instructions) of the basic block minus the number of ignored instructions.

To sum up, the first stage of KReachDist, carried out by the `parse()` function

[2]Terminator instructions are listed here: https://llvm.org/docs/LangRef.html#terminator-instructions

[3]It is not really a "pass" because we have all the CFGs and basic blocks and we just need to check all basic blocks' last line to do the job.

defined in `parse.py`, consists of collecting all the information and transforming it into basic blocks grouped by control flow graphs. A CFG represents therefore a function of the program.

4.2 Computing functions' summaries

The next step in computing distances is to calculate an intermediate distance value, called *summary*, for each defined function. We will then use this value to compute the distance between an instruction and the target in the code. A function *summary* is the shortest path between the function's entry point, in this case the first basic block of its CFG representation, and the function's nearest exit point. The distance will be computed based on the number of instructions executed by KLEE. Previously, in section 4.1, we defined this as the basic block size. However, some functions may call other functions, as a result, it is not enough to sum up the size of the basic blocks that form the path: we also need to take into account the summary of the called function. Indeed, counting the number of instructions in a function must consider all the instructions of a called function, as they have to be executed during the execution of the function.

For instance, when computing the summary of `bar()`, as defined in Figure 2, the following will be taken into consideration:

- KLEE ignores the first instruction, thus it is not considered in the distance;
- KLEE executes the last instruction of the basic block 0, thus the distance is incremented by 1;
- this instruction is a call to a defined LLVM function, thus the summary of `foo()` is added to the distance;
- the remaining instruction is the exit point of the function and is an instruction executed by KLEE, thus the distance is incremented by 1.

Let $S_{f()}$ be the summary of `f()`. Therefore, we have: $S_{bar()} = 2 + S_{foo()}$. This examples demonstrates the importance of the order of computation. To compute the summary of `bar()`, it is necessary to first compute the summary of `foo()`. Therefore, before computing any summary, a graph of dependency is established based on all CFGs. Figure 3 represents this graph for `foo()` and `bar()`.

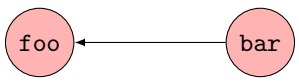

Figure 3: Dependency graph of `foo_bar.ll` CFGs

However, some function may have cross-dependencies. For instance, consider Figure 4 which represents the graph of dependencies for an extension of `foo_bar.c` presented in Listing 4.

```c
void ping(int v) {
    if (v > 0) {
        v--;
        pong(v);
    }
}

void pong(int v) {
    if (v > 0) {
        v--;
        ping(v);
    }
}
```

Listing 4: Addionnal functions in `foo_bar.c`

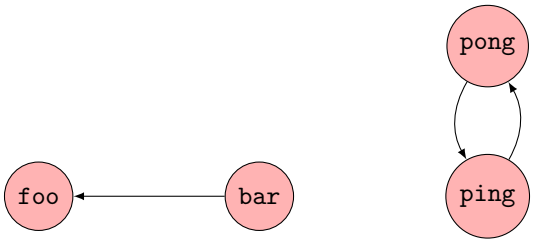

Figure 4: New dependency graph of `foo_bar.ll` CFGs

In this example, `ping()` and `pong()` call each other. There is therefore no precise order of computation for these two functions. To compute their summaries, consider that, for a function f_A calling a function f_B where the summary of f_B has not yet been calculated, the summary of f_B is considered to be infinite. This does not necessarily mean that the summary of f_A will be infinite, as there may be another path where f_B is not called (and where the distance is finite). By setting an arbitrary order for the calculation of the summaries of cross-dependent functions, it is possible to iterate over these calculations until a fixed point is reached, where summaries cease to change.

Using Tarjan's strongly connected components (SCC) algorithm automates the described procedure. This algorithm takes a directed graph as input and produces a list of strongly connected components in reverse topological order. In other words, no SCC will be identified before any of its successors. Applying this algorithm on the dependency graph results in the order in which the summaries should be calculated, *i.e.* by strongly connected component, in the reversed topological order. In addition, when a SCC is composed of more than one node, which means that there are cross-dependent functions, an arbitrary order between these functions is set and the summaries are computed until a fixed point is reached, as explained earlier. For our example, this algorithm produces the SCCs represented in Figure 5. The summary of `foo()` will be calculated before the summary `bar()` and lastly the summaries of `ping()` and `pong()` will

be computed as explained.

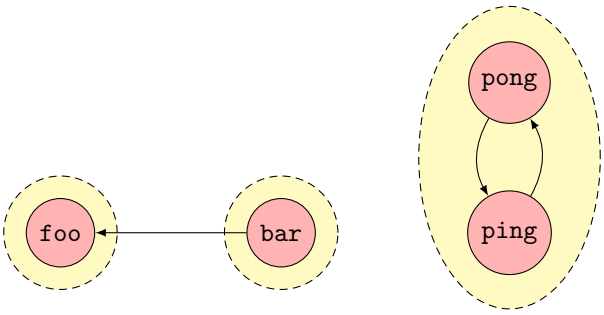

Figure 5: SCCs of the dependency graph defined in Figure 4

This being said, the overall algorithm used to determine the summary of a function f is Dijkstra's shortest path algorithm. Each element of the worklist is a pair $(\texttt{size}(id) + \texttt{call}(id), id)$ where id is the identifier of the concerned basic block, $\texttt{size}(id)$ returns the size of the basic block id and

$$\texttt{call}(id) = \begin{cases} S_g & \text{if } g \text{ is called at the end of BB } id \text{ and defined in LLVM} \\ 0 & \text{otherwise} \end{cases}.$$

As explained earlier, $S_g = \infty$ if the summary of g has not yet computed. Then, for each *new*[4] successor of the basic block id, a new pair is inserted in the worklist.

To summarise, the computation of functions summaries in the Python program is as follows the Algorithm 1.

4.3 Computing the `.dist` file

With all the information gathered in the previous two steps, it is now possible to compute the distances between all instructions and the target. The last part of KReachDist is to construct a `.dist` file where each line follows the format:

```
llvm_line:distance
```

where $(\texttt{llvm\_line}, \texttt{distance}) \in \mathbb{N} \times \mathbb{R}_+$. The following convention is used: if `llvm_line` is not found in the `.dist` file, then $h_{\texttt{llvm\_line}} = \infty$. If a distance is not found in the `.dist` file, the distance is considered to be infinite.

The first step in computing distances is to find the target in the LLVM file and match its position to a CFG c and a basic block id. If no target is found, the computation is complete and the `.dist` file is empty. With the target information, we will run Dijkstra's algorithm on the transposed CFG and from

[4] That has not yet be visited

Algorithm 1: Computation summaries of functions

Input : $CFGs$ `// list of all CFGs`
Output: S_{CFGs} `// set of CFGs' summaries`
$G \leftarrow$ `build_dependency_graph(`$CFGs$`);`
$SCCs \leftarrow$ `tarjan(`G`);`
$S_{CFGs} = \{\}$;
foreach $scc \in SCCs$ **do**
  $S_{scc} \leftarrow \{\}$;
  $S'_{scc} \leftarrow \{\}$;
  repeat
    for $n \in scc$ **do**
      $S_{scc} \leftarrow S_{scc} \cup S_{CFGs}$`.get_summary(`$n$`);`
      $S_{CFGs} \leftarrow S_{CFGs} \cup$ `dijkstra_summary(`n`);`
      $S'_{scc} \leftarrow S'_{scc} \cup S_{CFGs}$`.get_summary(`$n$`);`
    end
  until $S_{scc} = S'_{scc}$;
  $S_{CFGs} = S_{CFGs} \cup S_{scc}$;
end

the basic block containing the target. We want to trace the execution from the target, so we need to reverse the transitions between the basic blocks, hence the transposed CFG. We already dispose of this graph because we have constructed a list of predecessors for each basic block in section 4.1. Each member of Dijkstra's worklist follows the format:

$$(\texttt{distance}, (\texttt{cfg\_name}, \texttt{basic\_block\_id}, \texttt{has\_took\_ret}))$$

where `distance` $- 1$ is the distance between the first instruction of the basic block `basic_block_id` in the CFG `cfg_name` and the target. As with functions summaries, the distance between an instruction i and the target is interpreted as the number of executed instructions that separate i from the target. For each basic block id processed in the worklist, KReachDist computes the distance for this basic block and assigns the specific distance to each instruction. These distances are stored in an intermediate structure called `DistanceContainer`. To illustrate how the procedure works, let's look at the simplified[5] LLVM code of the function `foo_bar()` in the Listing 5. The transposed CFG of this function, in which we only represent the first and last instructions of each basic block, is represented in Figure 6.

```
1 define i32 @foo_bar(i32 %0, i32 %1) #0 !dbg !29 {
2   store i32 %0, i32* %3, align 4
3   %7 = load i32, i32* %3, align 4, !dbg !38
4   %8 = call i32 @foofoo(i32 %7), !dbg !39
5   store i32 %8, i32* %5, align 4, !dbg !37
```

[5]Certain instructions, which are not relevant to our example, are omitted.

```
  %9 = load i32, i32* %4, align 4, !dbg !42
  %10 = call i32 @foofoo(i32 %9), !dbg !43
  store i32 %10, i32* %6, align 4, !dbg !41
  %11 = load i32, i32* %5, align 4, !dbg !44
  %12 = load i32, i32* %6, align 4, !dbg !46
  %13 = icmp eq i32 %11, %12, !dbg !47
  br i1 %13, label %14, label %15, !dbg !48

14:                                               ; preds = %2
  call void @klee_reach() #4, !dbg !49
  unreachable, !dbg !49

15:                                               ; preds = %2
  ret i32 0, !dbg !51
}
```

Listing 5: `foo_bar()` function

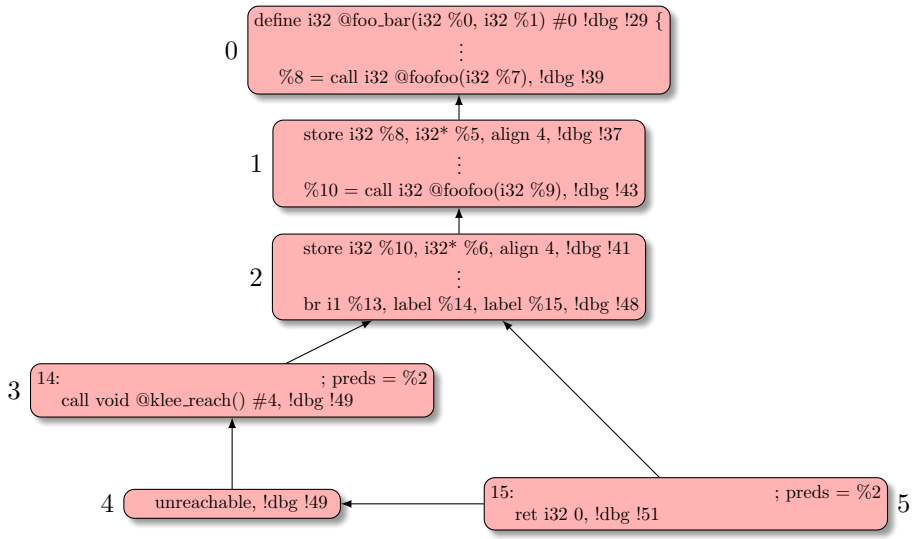

Figure 6: `foo_bar()`'s tranpose control flow graph

In the distance computation, the first element in the worklist would be $(1, (\texttt{foo\_bar}, 3, \texttt{False}))$. Note that the distance is 1, and not 2, because the label definition is an ignored instruction. There is only one predecessor to this basic block, therefore only $(6, (\texttt{foo\_bar}, 2, \texttt{False}))$ is inserted into the worklist. Considering the last inserted element will help us to explain how distances are computed in the Python structure. When this element is extracted from the worklist, we have the information that the first instruction in the basic block is at distance $6 - 1 = 5$ from the target. Because all adjacent instructions are at distance 1 from each other, we will obtained, for this basic block, the distances in Listing 6 in the `.dist` file.

```
8:5
9:4
10:3
11:2
12:1
```

Listing 6: Extract of `.dist` file for the basic block 2 in `foo_bar`

After processing this basic block, the next basic block to be treated would be 1. In this case, in addition to the size of the basic block, the summary of `foofoo()` must be taken into account when calculating the distance. Indeed, each instruction preceding the call to `foofoo()` must *at least* execute each instruction in the shortest path to exit this function before resuming its way to the target. Unlike the Dijkstra algorithm used in section 4.2, it is possible to insert elements referring to other CFGs. Because the basic block 1 ends with a call to `foofoo()`, there are different paths leading to it. Obviously, all predecessors of this basic block led to it, in this case the basic block 0, but so do all the exit points of the CFG `foofoo`. It is thus possible, from some of the instructions in `foofoo()`'s CFG, to reach the target using this call. We already take into account the cost of a call to `foofoo()`, thanks to its summary, but we also need to compute the distances for each reachable instruction in `foofoo()`. It is therefore necessary to represent the different links between CFGs to enable Dijkstra's algorithm to process elements in different CFGs. To do so, we will compute two graphs called G_{call} and G_{ret}. The former represents all associations between a basic block ending with a call to a function f and the first basic block of the CFG representing f. The latter represents all associations between a return statement in a CFG c and all basic blocks which are successors of those having a call to the function representing c. We have represented the edges of these graphs on the representation of the CFGs of `foo_bar()` and `foofoo()` in Figure 7. For instance, fb_0 and fb_1 end with a call to `foofoo()`. Thus, ff_4, which contains a return instruction, is associated with the successors of fb_0 and fb_1, *i.e.* fb_1 and fb_2, in G_{ret}.

However, we can't just follow any G_{ret} or G_{call} edge whenever we want. For example, starting from fb_0 leads to fb_1, then we can take the G_{call} edge to ff_0 and, after arriving at ff_4, take the G_{ret} edge leading to fb_2 instead of fb_2. This path is shorter than the expected path but is also forbiden. Starting from fb_0, it is mandatory to pass through `foofoo` twice before reaching fb_3, otherwise the flow of the program will be broken. To prevent this from happening, we introduce a rule to ensure that G_{call} and G_{ret} edges are well-matched [2]. The rule is as follows: when an ${}^tG_{ret}$ edge is taken, it is forbidden to take *any* ${}^tG_{call}$ edge. Figure 8 shows the transposed version of Figure 7. By doing so, we avoid the observed situation happening and we still obtain the correct distance using `foofoo` summary.

Adding these additional paths to Dijkstra's worklist is done simply by looking for them at the end of an element processing. The Boolean `has_took_ret` is there to ensure that we can't take a G_{call} when `has_took_ret` is set to true.

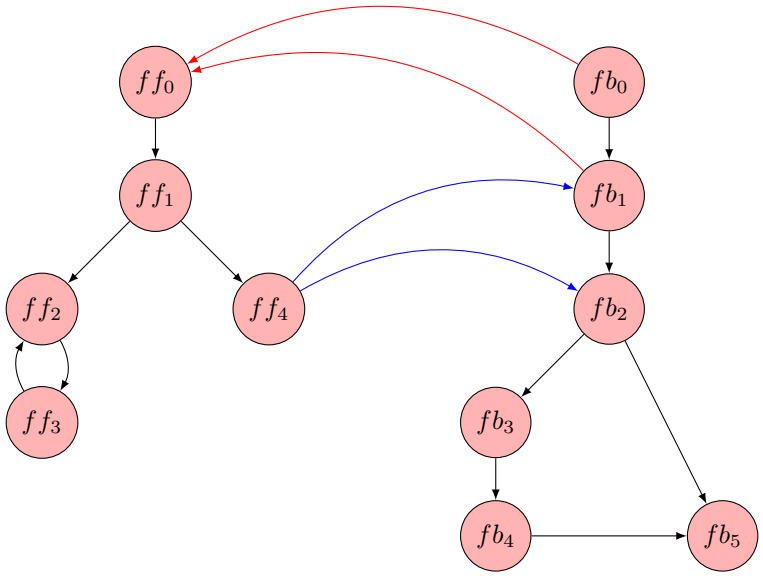

Figure 7: Representation of G_{call} and G_{ret} edges on `foofoo` (ff) and `foobar` (fb) representations

When Dijkstra's algorithm is complete, the container structure executes a method that writes the distances to the `.dist` file and the Python program is finished. To summarise the procedure, the last step of KReachDist follows Algorithm 2.

Algorithm 2: Computation of distances

Input : $CFGs$, S_{CFGs}
Output: `.dist` file
$G_{ret} \leftarrow$ `compute_gret`$(CFGs)$;
$G_{call} \leftarrow$ `compute_gcall`$(CFGs)$;
${}^tG_{ret} \leftarrow$ `transpose`(G_{ret});
${}^tG_{call} \leftarrow$ `transpose`(G_{call});
$t \leftarrow$ `find_target`$(CFGs)$;
$dist \leftarrow$ `dijkstra_distances`$(t, CFGs, S_{CFGs}, {}^tG_{ret}, {}^tG_{call})$;

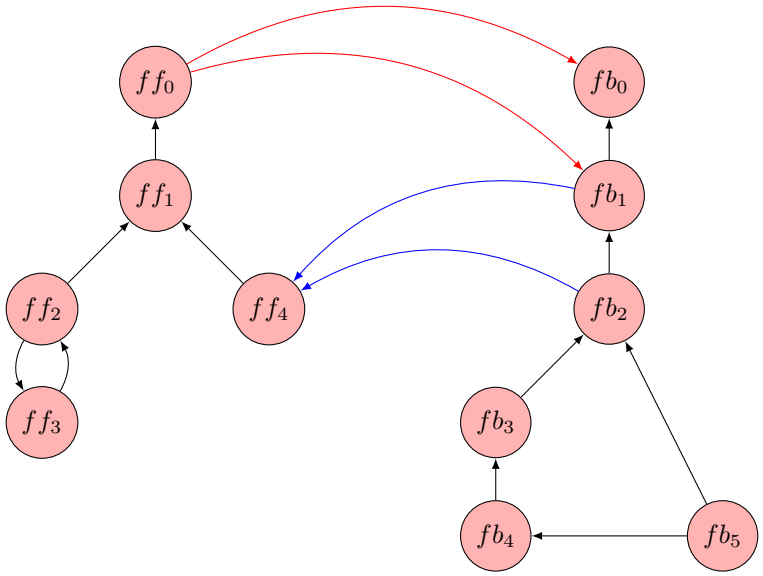

Figure 8: Representation of ${}^{t}G_{call}$ and ${}^{t}G_{ret}$ edges on `foofoo` (ff) and `foobar` (fb) transposed representations

References

[1] Theo De Castro Pinto, Antoine Rollet, Grégoire Sutre, and Ireneusz Tobor. Guiding symbolic execution with a-star. In Carla Ferreira and Tim A. C. Willemse, editors, *Software Engineering and Formal Methods*, pages 47–65, Cham, 2023. Springer Nature Switzerland.

[2] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings*, 2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings, pages 12–22, 2011. 20th International Symposium on Software Testing and Analysis, ISSTA 2011 ; Conference date: 17-07-2011 Through 21-07-2011.