POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Title of the thesis

Author:
Name Surname

Student ID:
XXXXXX

Advisor:
Prof. Name Surname

Academic Year:
20xx-xx

Dedicated to my family.

Abstract

Here goes the abstract.

Keywords: key, words, go, here

Abstract in lingua italiana

Qui va inserito l'abstract in italiano.

Parole chiave: qui, vanno, le, parole, chiave

Contents

1 | Introduction

2 | State of the art

3 | Background

3.1. Description of RL

RL is an imperative programming language and its structure is similar to other common imperative languages such as Python or C. An RL program is composed of functions, and the entry point is a special function with signature `fun main() -> Int`. Each function has a list of statements to be executed sequentially. These statements include what one might expect to find in an imperative language, such as variable declarations, assignments, if statements, while loops, and function calls. RL extends this familiar language structure with a couple key features.

3.1.1. Actions

(TODO: decide on a wording / voice to use for executing statements of a function body.) In addition to regular functions, RL features another function-like construct called actions. The main difference between a function and an action is in their control flow semantics. When a function is called, it runs until it executes a return statement. Then, all variables local to the function body are discarded and solely the return value is returned to the caller. In contrast, when an action is invoked it runs until it executes a special kind of statement which causes it to suspend. Then, it returns an action object that describes the current state of execution, including the values of local variables as well as where the execution has suspended. The caller can invoke methods of this returned object to resume the execution from its previous state. Action bodies do not contain explicit return statements, actions are meant to model processes, not values.

Two kinds of statements suspend an action's executions: Action statements and actions statements (TODO: the naming is very awkward here.) An action statement has the syntax `act action_name( param_type param_name, ...)`. When the action suspends on an action statement, it can be resumed by calling the method of the action object that shares a signature with the action statement. The action then resumes execution starting from the statement immediately succeeding the action statement. In addition, the action

statement's parameters are accessible by successive statements. The parameters assume the value of arguments passed to the function that resumed the execution.

An actions statement has the following syntax

```
actions:
    action_statement
    non_action_statement
    non_action_statement
    ...

    action_statement
    non_action_statement
    non_action_statement
    ...
```

An action suspended on an actions statement can be resumed on any of the action statements contained in it. When resumed, only statements until the next action statement are executed before proceeding to the actions statement's successor.

(TODO: Should I mention subaction statements here?)

Not all variables in an action's body are exposed to callers. Variables declared with the keyword `frm` are stored in the action's frame. They persist across suspensions and are reachable from outside the action. On the other hand, variables declared with the keyword `let` are temporary values. They are discarded when the action's execution suspends, just like local variables in regular functions.

As an example, consider the following program:

```
act nim(Int num_sticks) -> Nim:
    frm winner : Int
    frm current_player = 0
    frm remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count)
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player

fun main() -> Int:
    let game = nim(14)
    game.pick_up_sticks(3)
    game.pick_up_sticks(4)
```

```
17      game.pick_up_sticks(4)
18      game.pick_up_sticks(3)
19      if(!game.is_done()):
20          return 1
21      # This would result in a crash:
22      # game.pick_up_sticks(1)
23      if(game.winner != 0):
24          return 1
25      return 0
```

The main function invokes `nim(14)`. The action initializes the variables `current_player` and `remaining_sticks`, enters the loop and suspends at line 7. When it suspends, the action returns a `Nim` object, which holds the variables `current_player` and `remaining_sticks`, exposes the functions `pick_up_sticks(Int count)` and `is_done` and "remembers" that the action is suspended at the `pick_up_sticks` action statement. The main function then stores this object in the variable `game`, and calls the method it exposes multiple times. Each of these calls but the last one execute one iteration of the while loop before suspending on line 7 once again. During the last `pick_up_sticks` call at line 18, the action's execution exits the while loop and terminates, since there are no other action statements in the action's body. After this point, `game.is_done()` returns `True` and calling `pick_up_sticks` again will result in a crash since the action is not suspended on that statement. The action's frame variables are still accessible through the `Nim` object.

3.1.2. Preconditions

In RL, every function, action and subaction statement can have a list of preconditions attached to it. Any expression that evaluates to a boolean value can be a precondition, and preconditions can use the parameters of the function, action or action statement they are associated with. Unless optimizations are turned on while compiling, a function call or action instantiation with arguments that violate its target's preconditions results in a crash.

Being able to express preconditions is crucial for describing simulations. For instance, consider the nim example. The action

```
1 act nim(Int num_sticks) -> Nim:
2     frm winner : Int
3     frm current_player = 0
4     frm remaining_sticks = num_sticks
5
6     while remaining_sticks > 0:
7         act pick_up_sticks(Int count)
```

```
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player
```

allows players to pick up more sticks than there are in the game, or even a negative number of sticks. Moreover, the game is not restricted to start with a positive number of sticks in the first place. Enhancing the description with preconditions increases the description's readability, eases debugging and boosts the potential of automated analysis methods.

```
act nim(Int num_sticks) {num_sticks > 0} -> Nim:
    frm winner : Int
    frm current_player = 0
    frm remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count) {
            count > 0,
            count <= 4,
            count <= remaining_sticks
        }
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player
```

4 | Solution design

4.1. Conceptual design

As previously stated (TODO: make sure this is stated previously.), our goal is to implement a method of automatically generating efficient fuzzers for actions described in RL. In this section, we describe how we accomplish that goal. We start by describing a simple method to generate black-box fuzzers that uses almost none of the information available in an RL action description. Then, we build on this method incrementally, integrating parts of the available information one by one in order to improve performance.

4.1.1. The form of a fuzzer

Within the scope of this thesis, we only consider fuzzers integrated with LLVM's libFuzzer. (TODO: explain why?) LibFuzzer interfaces with the fuzzed action through a fuzzing entrypoint called the fuzz target. The fuzz target is a function that accepts an array of bytes and does something interesting with these bytes using the API under test (TODO: I took part of this from libFuzzer's page.). LibFuzzer invokes this function repeatedly with different fuzz inputs. It tracks which areas of the code are reached, and generates mutations on the corpus of input data in order to maximize the code coverage. Utilizing libFuzzer, we narrow the task of generating a fuzzer down to the task to generating a fuzz target. The fuzz target should use the fuzz input it receives to interact with the interface of the fuzzed action.

For the methods described in this section, we model the fuzz target as a function written in pseudo-code that has access to the fuzz input as well as the public methods of the fuzzed action. In reality, the fuzz target needs to be written in C and the action's interface is written in RL. We will describe how the two are connected in a later section.

Throughout this section, we will use the following action description to exemplify the methods we discuss:

```
1 act nim() -> Nim:
2     frm winner : Int
```

```
    frm current_player = 0
    frm remaining_sticks

    act decide_num_sticks(Int num_sticks) {num_sticks > 0}
    remaining_sticks = num_sticks

    while remaining_sticks > 0:
        act pick_up_sticks(Int count) {
            count > 0,
            count <= 4,
            count <= remaining_sticks
        }
        remaining_sticks = remaining_sticks - count
        current_player = 1 - current_player

    winner = current_player
```

The action is a slightly modified version of the Nim example from the previous section. In this version, the initial number of sticks is picked through an action. This change makes the examples more illustrative by introducing more than one action.

4.1.2. Generating black-box fuzz targets

As a baseline for our discussion, let us describe a simple method to generate black-box fuzz targets for RL actions. A black-box fuzz target knows about the subactions available in the action's interface, and their signatures. (TODO: think about renaming "subaction") The simplest black-box fuzz target uses some portion of the fuzz input to decide which subaction to call. Then, it generates arguments for each of the picked subaction's parameters. Finally, it calls the subaction with the generated arguments. This method interprets the fuzz input as an action call, and tests whether that action call results in problematic behavior.

A fuzz target for the Nim example would look like the following:

Algorithm 4.1 Black-box fuzz target for Nim

1: $game \leftarrow nim()$
2: $actionIndex \leftarrow pickIntegerValue(fuzzInput, 0, 1)$
3: **if** $actionIndex = 0$ **then**
4: $\quad arg0 \leftarrow pickValue(fuzzInput, INT\_MIN, INT\_MAX)$
5: $\quad game.decide\_num\_sticks(arg0)$
6: **end if**
7: **if** $actionIndex = 1$ **then**
8: $\quad arg0 \leftarrow pickValue(fuzzInput, INT\_MIN, INT\_MAX)$
9: $\quad game.pick\_up\_sticks(arg0)$
10: **end if**

Where `pickIntegerValue(byte[] fuzzInput, int min, int max)` is a function that consumes the next $log_2(max - min + 1)$ bits of `fuzzInput` to produce an integer between `min` and `max`. `INT_MIN`, `INT_MAX` represent the bounds of an integer value.

4.1.3. Performing a sequence of actions

One limitation of this simple black-box fuzzer is that it never executes more than one subaction of the fuzzed action. Therefore, it will not be able to discover bugs that occur only after multiple action calls. To amend this shortcoming, we can make the fuzz target repeat this process until it consumes every bit of the fuzz input. This method interprets the fuzz input as a sequence of action calls, instead of a single action call. In this way, the fuzzer will be able to capture the stateful behavior of RL actions.

Applied to the Nim example, this method would produce the following fuzz target.

Algorithm 4.2 Fuzz target performing multiple actions for Nim

```
1: game ← nim()
2: while fuzz input is long enough do
3:    actionIndex ← pickIntegerValue(fuzzInput, 0, 1)
4:    if actionIndex = 0 then
5:       arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
6:       game.decide_num_sticks(arg0)
7:    end if
8:    if actionIndex = 1 then
9:       arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
10:      game.pick_up_sticks(arg0)
11:   end if
12: end while
```

4.1.4. Avoiding expected crashes

Another critical shortcoming of this method is that invoking an arbitrary subaction with arbitrary parameters may result in an expected crash. For example, calling `pick_up_sticks` before calling `decide_num_sticks` is expected to cause a crash since the action will not have paused on the correct subaction. In addition, calling `decide_num_sticks(-1)` will also result in a crash since the precondition of the subaction is violated. If a call is expect to cause a crash, we describe the call as illegal.

When we make an illegal call, the program will crash and the fuzzer will report a bug. However, we are not interested in these crashes since they are a result of how the fuzz target interacts with the action, not a result of how the action is described. To solve this problem, we need some runtime mechanism to decide the legality of a call. We assume such a mechanism to be available for now and describe its implementation in a later section. (TODO: link that section here). Then, the fuzz target we generate should check the legality of all subaction calls it makes, and avoid making illegal calls. When it generates an illegal call, the fuzz target can simply continue to the next iteration of the main loop and generate a new call. This methods interprets the fuzz input as a sequence of legal action calls.

For the Nim example, we would obtain the following fuzz target:

Algorithm 4.3 Fuzz target performing multiple actions for Nim

```
1: game ← nim()
2: while fuzz input is long enough do
3:    actionIndex ← pickIntegerValue(fuzzInput, 0, 1)
4:    if actionIndex = 0 then
5:       arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
6:       if game.decide_num_sticks(arg0) is a legal call then
7:          game.decide_num_sticks(arg0)
8:       end if
9:    end if
10:   if actionIndex = 1 then
11:      arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
12:      if game.pick_up_sticks(arg0) is a legal call then
13:         game.pick_up_sticks(arg0)
14:      end if
15:   end if
16: end while
```

With these improvements, we have a functionally correct design. However, Even if we never make illegal calls, wasting fuzz input bits by generating illegal calls decreases the fuzzer's performance. (TODO: should I argue / demonstrate why?) We can reduce the number of illegal calls we generate, and therefore improve the efficiency of the fuzzers by making use of more information available in the action description.

4.1.5. Filtering out unavailable subactions

The first observation we make about the legality of subaction calls is that any subaction is always illegal if the action has not paused on the `ActionStatement` for that subaction, or an `ActionsStatement` containing that `ActionStatement`. With this observation, we can dynamically classify subactions as "available" or "unavailable". If we can guarantee we never generate a call to an unavailable subaction, we eliminate a large portion of generated illegal calls. To achieve that, we need a mechanism to dynamically decide whether a subaction is available. Assuming we can implement this mechanism, we can make sure to only pick available subactions. (TODO: link the section here when you write it.)

Here is an example for the Nim game:

Algorithm 4.4 Fuzz target performing multiple actions for Nim

```
1: game ← nim()
2: while fuzz input is long enough do
3:    availableSubactions ← []
4:    if decide_num_sticks is available then
5:       availableSubactions.push(0)
6:    end if
7:    if pick_up_sticks is available then
8:       availableSubactions.push(1)
9:    end if
10:   pickedIndex ← pickIntegerValue(fuzzInput, 0, len(availableSubactions))
11:   actionIndex ← availableSubactions[pickedIndex]
12:   if actionIndex = 0 then
13:      arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
14:      if game.decide_num_sticks(arg0) is a legal call then
15:         game.decide_num_sticks(arg0)
16:      end if
17:   end if
18:   if actionIndex = 1 then
19:      arg0 ← pickValue(fuzzInput, INT_MIN, INT_MAX)
20:      if game.pick_up_sticks(arg0) is a legal call then
21:         game.pick_up_sticks(arg0)
22:      end if
23:   end if
24: end while
```

4.1.6. Utilizing preconditions

Now that we have made sure we never pick unavailable subactions, the only remaining cause of generated illegal subaction calls is violated preconditions. Preconditions in RL can be arbitrarily complex. They can include any kind of expression RL supports, including calls to arbitrary functions. Therefore, given a set of conditions, finding a set of arguments that satisfy them is not an easy task. (TODO: Should I detail how hard this is?) As a result, we can not guarantee to never pick illegal arguments to a subaction call. Nevertheless, we can focus on particular forms of preconditions and extract some information from them to decrease the number of illegal arguments we generate.

As an example, consider the subaction arguments in the Nim action. We can replace

`pickValue(fuzzInput, INT_MIN, INT_MAX)` at line 13 with `pickValue(fuzzInput, 1, INT_MAX)`, since we know all negative choices are going to be discarded. Furthermore, when picking the argument of `pick_up_sticks`, we can `pickValue(fuzzInput, 1, min(4, game.remaining_sticks))`, so that we guarantee never picking an argument out of bounds.

The details of constraint types we consider in the scope of this thesis are described in a later section. (TODO: link that section here.)

4.2. Implementation

In this section, we explain how we implemented the method described in the previous section. We provide an architectural overview, then we detail how we address the challenges we have highlighted.

4.2.1. Architectural overview

As previously explained, a fuzzer built with libFuzzer needs to expose a fuzz target. The fuzz target is a C function that accepts the fuzz input and calls the fuzzed action's functions in some sequence, looking for unexpected behavior. The key challenge regarding generating fuzz targets is that they need to have access to both the fuzz input, which is a C byte array; and the action's subaction functions, which are available in RL. This requires part of the implementation to be in C or C++, and part of it to be integrated with RLC. The generated fuzz target needs to be customized to the number of subactions the action has, their signatures, and their preconditions.

One option is implement the body of the fuzz target in C or C++. In this case, we need to express the information related to the fuzzed action as either macros or templates. Then, we can extend RLC to emit a C header containing the relevant information. Finally, the C++ compiler can perform the necessary macro expansions and template instantiations to arrive at the customized fuzzer. We have explored this option extensively and generated fuzzers covering a subset of the features described in the previous section. However, we ultimately found it very difficult to read and maintain the fuzz target body written in terms of these macros. In addition, designing macro representations for RL concepts to be serializable into a C header without losing descriptive power proved to be challenging. Therefore, we ultimately decided against this option.

Instead, we extend RLC to emit a global function implementing the body of the fuzz target when it's passed the `-fuzzer` flag. This global function has a static signature,

independent of the characteristics of the action to be fuzzed. The fuzz target invoked by libFuzzer simply calls this function and then returns. Since RLC knows about all characteristics of the action to be fuzzed at compile time, it can customize this function accordingly, without the need of designing serializable representations for these characteristics.

On the other hand, it's difficult for RLC to emit the functions that interpret sections of the fuzz input as different data types. For instance, the `pickValue(fuzzInput, min, max)` function utilized in the previous section should parse the next $log_2(max - min + 1)$ bits of the fuzz input as an integer in the range $[min, max]$. A proper implementation of this function requires pointer arithmetic and bitwise operators. Fortunately, these functions do not depend on the characteristics of the action to be fuzzed. Hence, we implement these in C++ and the code emitted by RLC issues calls to them wherever necessary.

Furthermore, we extend RLC with a fuzzer standard library. This library contains functions, written in RL, that implement parts of the fuzz target body which don't depend on the fuzzed action's characteristics. For instance, the logic that initializes and maintains a vector of available subaction indices is implemented in RL source code. This increases the readability and maintainability of the RLC extension that emits the fuzz target. Since compiling RL source code is not a challenge for RLC, and emitting new operations that don't derive from a source code rapidly becomes very difficult to maintain.

To sum up, the fuzz target is generated in three distinct components that are then all linked together. The fuzz target called by libFuzzer and the utility functions to parse the fuzz input are written in C++ and compiled by a C++ compiler while building RLC. The parts of the actual fuzz target body that do not depend on the fuzzed action's characteristics are written in RL, and they are shipped with RLC as part of the standard library. The parts of the actual fuzz target that depend on the fuzzed action's characteristics are instead emitted by RLC at compile time if the `-fuzzer` option is specified. Linking these three components, as well as libFuzzer itself, produces the final fuzzer binary.

4.2.2. Deducing the availability of subactions

Available subactions are subactions which the action can resume from. A subaction is available if the execution of the action has last suspended on the `ActionStatement` corresponding to that subaction or an `ActionsStatement` containing it. In Section 4.1.5, we described how we can generate subaction calls only to available subactions. As a prerequisite, we assumed the availability of some mechanism to decide whether an action

is available. In this section, we explain the relevant implementation details of RLC and describe how we implemented this mechanism.

To start with, we should describe how subaction functions called at the wrong time cause crash. Recall that action calls return action objects that keep track of the state of the action. This state includes frame variables, as well as where the action has last suspended. The first field of the returned action object is always `resumeIndex`. `resumeIndex = 0` means the action has not started executing yet, and will start from the beginning of its body. When suspending on an `ActionStatement` or `ActionsStatement`, a `resumeIndex` corresponding to that statement is stored in the action object. The subaction functions all accept this action object as their first argument and they implicitly have a precondition checking whether the `resumeIndex` stored in it is the expected one. When a subaction function is called at the wrong time, this precondition fails, resulting in a crash.

Taking this into account, it's sufficient to check whether the stored `resumeIndex` is the one expected by the subaction function to dynamically decide its availability.

4.2.3. Deducing the legality of subaction calls

As alluded to in Section 4.1.4, we need a mechanism to dynamically decide whether a subaction call with a particular set of arguments is expected to result in a crash. In this section, we explain how this mechanism is implemented.

We make two extensions to RLC in order to be able to check the legality of subaction arguments. First, we introduce an additional IR node, called `CanOp`. `CanOp`s have a single operand and a single result, both of type `Function`. A `CanOp` returns a function that accepts the same arguments as its operand, and has a boolean return type. The function returns whether its operand's precondition evaluates to true with the arguments passed to it. Second, we introduce a new compiler pass. This pass walks through each function in the module, emits a global function that checks its precondition, then replaces the results of all `CanOp`s referring to the original function with the newly emitted precondition checker.

Having introduced `CanOp`, the emitted fuzz target can deduce whether a subaction call is legal simply by emitting a `CanOp` to the subaction function, and a call to the `CanOp`'s result with the picked arguments. This allows the emitted fuzzer to avoid running into expected crashes.

4.2.4. Precondition Analysis

The last major improvement to the fuzzers we generate is utilizing RL preconditions to avoid picking subaction arguments for which the subaction's precondition evaluates to false. Before we detail our implementation, we should discuss the difficulty of this problem in the general case. The precondition of an RL subaction consists of a set of boolean expressions implicitly in conjunction. These boolean expressions are not limited in the scopes and types of sub-expressions they can include. They may include function calls, and the function calls might be blocking, they may have side effects and they may be non-deterministic. Even if we could assume that preconditions consist only of pure, non-blocking, deterministic expressions, finding a set of arguments to satisfy their conjunctions is no easier than answering arbitrary SMT queries.

Given this perspective, our approach is to handle some subset of preconditions and extract the maximum information from those. We can not claim to always be able to generate legal arguments, but we can handle some of the most frequent types of constraints to greatly reduce the number of illegal arguments we generate. Our technique is a best-effort, not a definitive solution.

We restrict our precondition analysis to integer arguments to subaction functions, as well as struct arguments that are composed solely of integers and other such structs. For each integer argument, or integer field of a struct argument of a subaction call, we dynamically decide a minimum and a maximum value. Such that any value greater than the maximum or smaller than the minimum certainly invalidates the precondition. Then, we pick the argument to be any integer in this range using the $pickValue(fuzzInput, INT\_MIN, INT\_MAX)$ function implemented in C++ introduced previously. Although this is far from a perfect model of arbitrary constraints on integers, we have found it to be descriptive enough to significantly boost fuzzing performance.

Let us now describe how we decide the minimum and maximum values dynamically. For simplicity, we focus only on integer arguments here. We start by defining a classification on expressions that comprise the precondition in terms of the availability of their evaluation at runtime. We classify expressions into two categories:

Bound expressions are expressions such that we can decide the value they will evaluate to while evaluating the precondition before we evaluate the precondition. These include constants, local variables, global variables and combinations of these with deterministic operations, as well as deterministic function calls.

Unbound expressions are expressions such that it's not possible to know the value

they will evaluate to before evaluating the precondition. These include arguments of the subaction, nondeterministic function calls and any expression that them as sub-expressions.

Note that an argument of the subaction does not have to be an unbound value. In fact, since RLC implements action objects' member functions as global functions with the implicit first argument pointing to the object, the first argument of any subaction function call is always a bound expression, it evaluates to the action object. Having introduced this terminology, our goal is to find the minimum and maximum values the unbound arguments of a subaction function can take while satisfying the function's precondition.

We introduce a new compiler pass to emit the code that decides the minimum and maximum values dynamically. We start by normalizing the precondition to be a disjunction of conjunction of terms, where each term is a value produced by something other than an `AndOp` or an `OrOp`. In other words, we express the precondition in the form $(t_1 \wedge t_2 \wedge ...) \vee (t_3 \wedge t_4 \wedge ...) \vee ....$ Then, we emit a block of code for each unbound argument that computes its minimum and maximum values.

For each conjunction $(t_1 \wedge t_2 \wedge ...)$ in the normalized precondition, we classify its terms depending on how they relate to `arg`.

Conditions are terms that consist solely on bound expressions. A condition itself is bound and can be evaluated without invoking the subaction precondition.

Constraints are terms that include `arg` as a sub-expression, and no other unbound expressions. These terms impose constraints on `arg` that can be entirely expressed before the invocation of the precondition.

The rest of the terms depend on one or more unbound expressions other than `arg`. For this analysis, we assume they do not constrain `arg` in any capacity. This assumption does not always hold, and that might result in loose minimum and maximum bounds, but we simply can not decide the constraints imposed by these terms on `arg` before evaluating the precondition. Discarding these terms is as far as our best-effort approach can stretch.

At runtime, before evaluating the precondition, we can evaluate whether all conditions of a conjunction $(t_1 \wedge t_2 \wedge ...)$ hold. If all conditions hold, the constraints of this conjunctions are said to be active. Assuming we can decide the minimum and maximum values imposed on `arg` by a single constraint, we can compute the aggregate minimum and maximum values by intersecting the ranges imposed by the terms of each active conjunction, then computing the union of ranges imposed by all active conjunctions. We can achieve this by emitting an if statement of the following form for each conjunction:

Algorithm 4.5 Aggregating the minimum and maximum values imposed by individual constraints

```
1: aggregate_min ← ∞
2: for conjunction (cond_1 ∧ cond_2 ∧ constr_1 ∧ constr_2)innormalizedprecondition do
3:    if cond_1 ∧ cond_2 then
4:       current_min ← −∞
5:       if imposed_min(constr_1) > current_min then
6:          current_min ← imposed_min(constr_1)
7:       end if
8:       if imposed_min(constr_2) > current_min then
9:          current_min ← imposed_min(constr_2)
10:      end if
11:   end if
12:   if current_min < aggregate_min then
13:      aggregate_min ← current_min
14:   end if
15: end for
```

and similarly for the maximum value.

As for how how we decide the minimum and maximum values imposed by individual constraints, we adopt a similar best-effort approach where we handle some forms of constraints that can easily be handled. For more complex constraints, we simply assume they constrain the `arg` to be in the range $(-\infty, \infty)$.

We handle constraints that are binary operations where any one of the two operands is `arg` and the operation is one of $<$, $>$, $\leq$, $\geq$, $=$. In addition, we handle the constraints which are `CallOp`s with the callee being the result of a `CanOp`. In this case, we apply a recursive constraint analysis to the underlying callee, mapping the unbound expressions in the current analysis context to the new analysis context to decide how `arg` is constrained by the precondition of the callee. Recursively analyzing preconditions of `CallOp`s in the precondition is critical to the applicability of this technique because RLC has some passes that wrap an action function inside of another action function. We would not be able to utilize the preconditions of wrapped action functions without handling these constraints.

5 | Evaluation

5.1. Goals

5.2. Conditions

5.3. Baseline

In this section, we describe the baseline we use to evaluate the efficiency of fuzzers generated by RLC. As a baseline, we need a simpler method of automatically generating fuzzers for game descriptions. Any such method needs to establish an abstraction for how a game is described. We have chosen to use the abstraction of OpenSpiel, Google DeepMind's framework for applying reinforcement learning methods to games (TODO: I probably need to describe OpenSpiel in greater detail.).

The OpenSpiel repository includes some example games. We describe a simple method of generating fuzz targets for these games. These games, modified minimally to introduce bugs the fuzzer can find, form the baseline of our evaluation.

Generating fuzzers for OpenSpiel games

(TODO: I adapted this from the OpenSpiel paper, should figure out how to cite that.) Games in OpenSpiel are described as producedural extensive-form games. Where a game has:

- finite set of players. Including a special player representing chance.
- A finite set of all possible actions players can take in all game states.
- A finite set of histories. Each history is a sequence of actions that were taken from the start of the game.
- A finite set of terminal histories that represent a finished game.
- A utility for each player for each terminal history.

- A player assigned to take the next action for each non-terminal history. Including a special player representing simultaneous states, where players act simultaneously choosing a joint action.
- A set of states, where each state is a set of histories such that histories in the same state can not be distinguished by the acting player.

Complete game descriptions are objects that implement some interface methods exposing the elements described above, along with some utility methods to simplify moving from a history to its successors. In particular, OpenSpiel game descriptions implement the following methods that are useful for generating a fuzzer:

- `Game::NewInitialState`
- `State::isTerminal`
- `State::isChanceNode`
- `State::isSimultaneousNode`
- `State::LegalChanceOutcomes`
- `State::LegalActions`
- `State::ApplyAction`

Depending only on these functions, we can generate a fuzz target analogous to the ones we generate for RL descriptions as shown in Algorithm 5.1, where *pickOne* is a function that picks an element of the given set consuming the next $log_2(n)$ bits of the fuzz input. After generating the fuzz target, we plug the fuzz target into LLVM's libFuzzer to generate the complete fuzzer.

Algorithm 5.1 Fuzzing and OpenSpiel game

```
1: state ← game.NewInitialState()
2: while state.isTerminal() do
3:    if state.isChanceNode() then
4:       nextAction ← pickOne(state.LegalChanceOutcomes())
5:    else if state.isSimultaneousNode() then
6:       actions ← []
7:       for player ∈ players do
8:          actions.append(pickOne(state.LegalActions(player)))
9:       end for
10:      nextAction ← ApplyAction(actions)
11:   else
12:      nextAction ← pickOne(state.LegalActions())
13:   end if
14:   state.ApplyAction(nextAction)
15: end while
```

It should be noted that OpenSpiel games may have two kinds of chance nodes. Explicit stochastic chance nodes expose multiple legal actions, as well as a probability distribution over those actions. On the other hand, sampled stochastic chance nodes expose a single action with non-deterministic behaviour. This fuzz target is only suitable for games with no sampled stochastic chance nodes since the fuzz target has to be deterministic with respect to the fuzz input.

Tic tac toe

5.4. Results

6 | Conclusion

Bibliography

List of Figures

List of Tables

Acknowledgements

Here you may want to acknowledge someone.