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 | Background

In this chapter, we introduce the motivation behind RL and provide a brief overview of the language. Then, we describe the language features tightly linked with fuzzing in greater detail, aiming to supply the reader enough of a background to be able to comprehend how our fuzzer generation technique works.

2.1. What is RL

RL (RuleBook) is a domain specific language to describe multi-agent interactions, most commonly games [1]. RLC is the compiler for RL. RL is created to solve a problem with the way games are implemented today that makes it difficult to apply automated analysis and machine learning techniques to games. Today, games are developed as a collection of event handlers. The developer specifies the actions players can take, as well as the effect of those actions on the game state. The actions are described in isolation, one has to analyze which actions can follow which others in order to simulate a game trace. In contrast, when we describe games to other humans in natural language, as RL does, we describe them as sequential scenarios where the players get to take actions as part of the scenario. We expect the actions to be taken at certain points in the scenario, with certain constraints. As an example, compare two natural language descriptions of a single hand of blackjack to illustrate the difference.

A description in terms of events, their results, and their constraints could be:

- Players can take additional cards.
- Players can pass.
- It is always a single player's turn.
- When a player takes an additional card, they get a card from the top of the deck.
- A player may not take an additional card unless it is their turn.
- A player may not take an additional card if they have passed before.

- When a player takes an additional card or passes, it becomes the next player's turn.
- Players start with 2 cards in their hands.
- If all players have passed once, the player with the hand closest to 21 wins.

On the other hand, a description as a sequential scenario could be:

- The game starts by dealing each player a hand of 2 cards.
- Until all players have passed once, each player that has not previously passed decides whether they want to take an additional card or pass.
- The players with the hand closest to 21 wins.

The description organized into event handlers makes games harder to comprehend and potentially harder to develop for humans. However, RL's main focus is not really on human readability. It tries to solve the problem that modern games implemented as event handlers are very difficult to perform automated analyses on. A sequential description makes game descriptions more amenable to automated analysis techniques. These techniques can range from fuzzing the game with random actions hoping to discover invalid game states early in development, to training ML models on the game.

2.2. Description of the RL language

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.

2.2.1. Actions

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 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 resumes 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.

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:
```

```
7           act pick_up_sticks(Int count)
8           remaining_sticks = remaining_sticks - count
9           current_player = 1 - current_player
10
11      winner = current_player
12
13  fun main() -> Int:
14      let game = nim(14)
15      game.pick_up_sticks(3)
16      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.

2.2.2. 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

```
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
```

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
```

To summarize, RL is a domain-specific language aimed at writing, reading and automatically analyzing game descriptions easier, as well as running machine learning methods on them. It has a Python-like syntax. RL allows describing games as sequential scenarios mainly through actions, which are stateful suspendable procedures. In addition, RL supports specifying preconditions for functions, actions and subactions, further facilitating the application of analysis techniques.

3 | State of the art

In this chapter, we aim to provide an overview of the widely adopted technologies for compilers and fuzzing.

3.1. Compiler technologies

3.1.1. LLVM

LLVM, short for Low Level Virtual Machine, is a compiler framework designed to perform lifelong program analysis and transformation for arbitrary software. [LLVM] Lifelong analysis and transformation includes link-time techniques for interprocedural analyses, install-time techniques for machine-dependent optimizations and runtime techniques for dynamic analyses. LLVM revolves around a common program representation independent from both the source language and the target architecture, called LLVM IR. LLVM IR is a RISC-like instruction set enhanced with some high level information to facilitate analyses. It includes source language independent type information, explicit control flow graphs and explicit dataflow representations using registers in Static Single Assignment form [ref here]. The design of this intermediate representation allows utilizing the LLVM framework with a wide range of source languages and target architectures, while being descriptive enough to allow powerful transformations, analyses and optimizations on the IR itself.

LLVM has a three tier architecture. First, the program in a source language is first compiled compiled into LLVM IR. This stage includes lexical analysis, syntax analysis, semantic analysis and high-level language dependent optimizations and transformations. The most popular example, clang, is an LLVM front-end that [ref here] compiles C, Objective C and C++ into LLVM IR. However, clang is far from being the only adopted frontend for LLVM. Researchers have developed LLVM frontends for to perform various tasks ranging from compiling languages like Rust [ref here] or Ruby[ref here] to creating a JIT compiler for Python [ref here].

Then, compiler language-independent optimizations, analyses and transformations are

performed on LLVM IR[ref here - llvm docs]. These are implemented as independent passes on the IR. Examples include analyses such as stack safety analysis and memory dependence analysis, as well as transformations such as dead code elimination, function inlining and duplicate global constant merging. Thanks to the modular nature of this optimization and analysis stage, researchers have implemented various custom passes for tasks such as polyhedral optimization[ref here] and race detection [ref here].

Lastly, the optimized LLVM IR is handed over to a target architecture dependent code generator. The code generator performs architecture-dependent optimizations, as well as tasks such as instruction selection and register allocation. Separating this stage form the rest of the compile architecture enables developers to support a new architecture by implementing a new code generator only.

3.1.2. MLIR

MLIR, short for Multi-Level IR Compiler Framework is a framework aiming to reduce software fragmentation in compiler development, improve compilation for heterogenous hardware, facilitate the development of compilers for domain specific languages and pave the way to connect existing compiler frameworks together[ref here]. MLIR is maintained as part of the LLVM project, and is tightly integrated with LLVM.

The creators of MLIR observed that even though the LLVM framework is useful for reusing compilation techniques and algorithms that do not depend on the source language and the target architecture, modern languages often resort to developing their own IR in order to solve domain-specific problems. Such as library-specific optimizations or optimizing machine learning pipelines. In addition, lowering from a decorated syntax tree for a modern language to LLVM IR is not always a straight-forward task and often necessitates one or more intermediate forms of IR, as illustrated in Figure 3.1.

Although it's certainly possible develop intermediate representations for compiler frontends, the developers end up solving many common problems such as location tracking, pass management and pattern based lowering. This manifests in a decreased quality in the infrastructure of these compilers, especially for smaller domain specific languages. MLIR solves this problem by introducing a standard for Static Single Assignment based intermediate representations. IR's developed with MLIR can make use of the MLIR's library of solutions to common problems, and integrate with other MLIR based IR's with great ease.

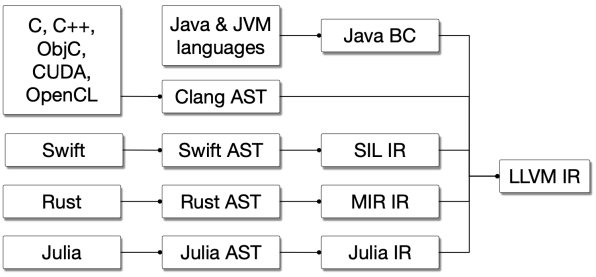

Figure 3.1: Compilation pipelines of different languages. Taken from [ref here].

Structure of MLIR

In MLIR, the unit of semantics is an "Operation", shortly referred to as Ops. Ops can represent various program concepts ranging from a single instruction or a function to a whole file or module. An Op has one or more operands, and more or more results. Both operands and results are SSA values. An Op's operands' definitions need to dominate the Op, and MLIR takes care of validating this constraint. In addition, MLIR allows nesting operations via Regions. An Op may have one more more attached Regions, which can contain one or more Blocks. Blocks are in turn sequential lists of operations. The blocks of a region form a control flow graph. The semantics of this control flow graph, as well as the semantics of the region itself depend on the type of Op they are attached to. An Op may also have one or more attributes, which represent compile-time information such as constants and names.

As opposed to LLVM IR's fixed set of instructions, MLIR has an extensible set of Ops. An MLIR based IR for a domain specific language is organized into one or more MLIR dialects. Dialects define types of Ops, as well as value types and attributes. The MLIR repository includes many dialects to model different types of IR [ref - MLIR docs]. For instance, there exists an LLVM dialect which models LLVM IR, a SPIR-V to represent graphics shaders and compute kernels, and a "linalg" dialect to represent linear algebra structures and operations.

During compilation, the IR does not need to consist solely of a single dialect at every instance. The IR in general is made of operations from a collection of dialects, and it's lowered incrementally by multiple passes into the target dialect. Most commonly, the target dialect is LLVM. But MLIR can support lowering into different dialects too. This extensibility has been utilized to create a wide range of domain specific compilers ranging from an Open Neural Network Exchange (ONNX) compiler [ref] to a High Level Synthesis (HLS) compiler[ref]. Most notably for this thesis, RLC itself is implemented as an MLIR dialect and a collection of passes to lower it to LLVM IR.

3.2. Fuzzing

Fuzzing, as introduced by Miller *et al.* in 1998[ref here], is an automated software testing technique where the program under test is invoked with randomly generated test cases hoping to trigger bugs. A fuzzer is a program that generates the test cases. Although Miller *et al.*'s idea was as simple as invoking UNIX utilities with random strings, fuzzers today are much more sophisticated. As an example, American Fuzzy Lop (AFL) [ref here] applies some lightweight instrumentation to the branch instructions of the compiled programs it tests. Then, it measures how many new CFG edges are traversed by each fuzz input at runtime. It generates new fuzz inputs by mutating the ones that result in high coverage. On the other hand, SlowFuzz [reference] tracks the resource usage of its test target to find inputs that trigger algorithmic complexity vulnerabilities. Today, fuzzing is an increasingly popular technique to find software vulnerabilities.

3.2.1. White-box vs. black-box fuzzers

Fuzzers can be categorized in terms of how much information they require about the program under test. Black-box fuzzers require no information about their target. They either generate inputs completely randomly or mutate an initial library of well-known inputs using a set of pre-defined rules. On the other hand, white-box fuzzers know everything there is to know about the program under test. One example, DART [ref], first feeds random inputs to the program under test, symbolically executes the discovered traces to gather a set of path constraints on them, then uses a solver to generate inputs that are guaranteed to discover new paths. In practice, most widely used fuzzers today fall into a category in between these two, gray-box fuzzing. They make use of partial information about the program under test. This information is generally obtained by instrumenting the program.

For the fuzzers described as part of our work, we use the term black-box for the fuzzers that make the smallest use of the information available about the fuzzed games. Similarly, we will use the term white-box to describe the fuzzers that use most of the available information, compared to the alternatives described in this thesis. In reality, all fuzzers implemented as part of this thesis can be described as grey-box fuzzers. Nevertheless, we find the terms black-box and white-box useful to highlight the difference in the fuzzers we implement.

3.2.2. libFuzzer - Fuzzing in LLVM

LibFuzzer [ref] is the fuzzing engine integrated with the LLVM project. Being integrated with the Clang compiler, libFuzzer is directly linked against the program under test, rather than running as a separate process that invokes the program under test. The link between libFuzzer and the program under test is formed by a function called the fuzz target. The fuzz target is a C function that should call the API of the program under test utilizing the fuzz input. It has the following signature:

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    DoSomethingInterestingWithMyAPI(Data, Size);
    return 0;
}
```

Since it is integrated with the LLVM framework, libFuzzer can make use of various LLVM sanitizers. For instance, information provided by Address Sanitizer (ASAN) [ref] enables libFuzzer to detect bugs that might otherwise happen silently, such as buffer overflows or memory leaks.

Futhermore, being a gray-box fuzzing engine, libFuzzer uses instrumentation on the program under test to track the basic blocks visited by each fuzz input. This instrumentation is provided by LLVM's SanitizerCoverage [ref]. The instrumentation being external to libFuzzer and included in LLVM is vital to the method we introduce in this thesis. Even though, we do not use the whole Clang pipeline, we can integrate SanitizerCoverage after lowering RL to LLVM IR. This enables us to seamlessly provide the coverage information to libFuzzer with little effort.

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

In this chapter, we describe the experiments we conducted to evaluate the performance of the fuzzers we generate. We measure the performance of the baseline black-box fuzzers, as well as the impact of our two major improvements to them: avoiding generating calls to unavailable subactions and analyzing subaction preconditions to reduce the number of generated illegal arguments. In addition, we describe another simple technique of automatically generating fuzzers for game descriptions that's completely independent from RL. We use this technique to generate black-box and white-box fuzzers, and compare their performance with the RLC counterparts to assess the overall performance of using RL to fuzz game descriptions.

We evaluate the fuzzers on three sample games. We introduce bugs in these games for the fuzzers to find. In addition, we construct the games in a way that allows us to parametrize the minimum number of legal actions to be taken on the game in order to find the bug. This allows us to investigate how the fuzzers' performance evolve as the bug complexity increases.

5.1. 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.

The OpenSpiel repository includes some example games. We describe two simple methods to generate white-box and black-box fuzz targets for these games. These games, modified minimally to introduce bugs the fuzzer can find, form the baseline of our evaluation.

5.1.1. Generating white-box 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 procedural 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 White-box fuzzer for OpenSpiel games

```
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 behavior. 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.

5.1.2. Generating black-box fuzzers for OpenSpiel games

The method to generate black-box fuzzers is very similar to its white-box alternative. The white-box fuzzers have access to perfect information about what actions can be taken on the game at any given state. They also know the complete action-space of the game. In contrast, the black-box fuzzers should be analogous to the simple black-box fuzzers we described in Section4.1.2. The white-box fuzzers always pick legal actions thanks to the function call `state.LegalActions(player)`. This returns the complete list of legal actions for the current game state, and the white-box fuzzer can pick among them.

Fortunately, OpenSpiel actions are represented by integers. Therefore, the black-box

fuzzers for OpenSpiel games can simply pick any integer as the action to be taken, as opposed to picking from a set of valid integers. In this way, we obtain fuzzers analogous to the black-box fuzzers generated by RLC.

5.2. Benchmarks

In this section, we describe how our benchmarks are constructed. We use three games from OpenSpiel samples as our benchmarks: `blackjack`, `tic_tac_toe` and `crazy_eights`. We modify these games lightly to introduce bugs in them for the fuzzers to find. In parallel, we implement the same three games in RL, introducing the same bugs.

Furthermore, we want to be able to observe how the fuzzer performances change as the bugs become harder to discover. In order to achieve this, we modify each game that allows us to control the minimum number of successive legal actions the fuzzer has to take before finding the bug via a numeric parameter. We call this parameter bug depth. Each game implements bug depth in a suitable form.

Blackjack

Blackjack is a simple card game played with a standard 52 card deck. Cards 2-9 are worth points equal to their rank, face cards are all worth 10 points and aces are worth the player's choice of 1 or 11 points. The goal is to get as close to 21 points without going above it. Players all start the game with 2 cards in their hand. Players take their turns in sequence. On a player's turn, they can choose to draw any number of cards as long as they do not surpass 21 points. When they draw a card that puts them above 21 points, they lose the game.

The benchmark is blackjack played with one player and a dealer. The dealer does not explicitly take actions on the game. When the player passes their turn, the dealer keeps drawing cards as long as they are more than 4 points short of the target score. The bug is triggered when either the player or the dealer reached the target score exactly.

We modify the game with a bug depth parameter. This parameter controls the number of suits in the deck, as well as the target score. The deck has $52 * \textit{bug depth}$ cards and the target score is $21 * \textit{bug depth}$. The dealer stops drawing at $21 * \textit{bug depth} - 4$ points. Since the point values of cards do not scale with bug depth, the number of cards a player has to draw before reaching the target score scales linearly with bug depth.

Tic Tac Toe

Tic tac toe is a game where players take turns marking unmarked spaces in a 3x3 grid. The first player to mark three spaces in a straight line, including the two diagonals, wins.

The benchmark is a sequence of tic tac toe games. The two players player as many tic tac toe games as bug depth, and the bug triggers when a player wins the last game by marking all three spaces of the secondary diagonal. Since the players need to reach the last game for the bug to trigger, they need to take a number of actions that scales linearly with bug depth.

Crazy Eights

Crazy eights is a precursor of UNO. It is played with a standard 52 card deck. At the start fo the game, each player is dealt 7 cards. Then, the top card of the deck is revealed. On their turn, the player can draw up to three cards, and play up to one card. They can only play cards that have a matching suit or a matching rank with the last played card, or an eight. When a player plays an eight, they pick the suit the next player has to match. A player needs to draw all three cards if they do not play a card. Once a player plays a card, they can not draw more cards than turn.

Crazy eights can include special cards that make players draw additional cards, skip their turn, or reverse the turn order. Our benchmark does not implement these special cards. The bug triggers when a player tries to draw from an empty deck. Similarly to blackjack, the bug depth controls how many cards there are in the deck. An increasing bug depth linearly increases the number of actions to be taken before reaching the bug.

5.2.1. Experiments

We measure the performance of fuzzers generated with six different methods the three games described above.

rlc-full is the fuzzer described in Chapter 4 including all mentioned improvements. This fuzzer is generated by invoking RLC with the –fuzzer flag.

rlc-no-fsm is another fuzzer generated by RLC. However, this one does not avoid generating calls to unavailable subactions. It is generated by calling RLC with –fuzzer-avoid-unavailable-subactions=0 in addition to –fuzzer.

rlc-no-precons is the version of `rlc-full` that does not analyze preconditions to generate reasonable subaction arguments. It is generated by passing RLC the –fuzzer-analyze-preconditions=0 flag in addition to –fuzzer.

rlc-blackbox is generated by turning off both improvements when calling rlc.

os-whitebox is the white-box fuzzer for OpenSpiel games described in Section 5.1.1.

os-blackbox is the black-box version of `os-whitebox` described in Section 5.1.2

We generate these six fuzzers for each benchmark for various bug depths. We measure the average time taken to find the bug, as well as the average number of fuzz inputs tested. For all experiments, we have to enforce some upper limit on the time we allow the fuzzers to search for the bug. This is necessary since the true measurement will be impractically large for the less advanced fuzzers. Which makes them very hard to measure and pointless to present here. We found 150 seconds to be a suitable time limit to run a single fuzzer instance.

5.3. Results

In this section, we present the results of our experiments and discuss our findings.

5.3.1. Blackjack

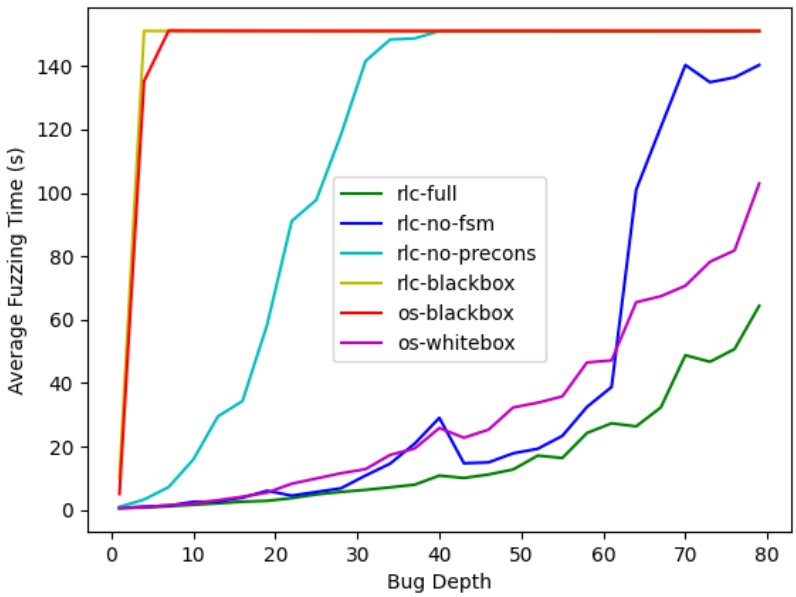

Figure 5.1: Average fuzzing times for blackjack.

For blackjack, we measured the average fuzzing times and the average number of tested fuzz inputs for bug depths between 1 and 80. We observed that `rlc-blackbox` fails to find the bug in less than 150 seconds starting at bug depth 3. Similarly, `os-blackbox`

hits the execution time threshold starting at bug depth 5. We expected the black-box fuzzers to perform worse than their white-box counterparts, and this finding matches our expectation. We observe a minor difference between the two black-box fuzzers.

Comparing `rlc-no-precons` with the black-box fuzzers, we observe that avoiding generating calls to unavailable subactions has a noticeable positive impact on the average fuzzing time. The RL implementation of blackjack has 6 different subactions, a maximum of 2 of those actions can be available at the same time. Picking the next action from among the available ones eliminates enough illegal action calls to justify the extra computation in this benchmark. `rlc-no-precons` can find the bug without exceeding the time threshold for bug depths under 41.

Similarly, we observe that analyzing the preconditions of the subactions in blackjack increases the performance dramatically. `rlc-no-fsm` outperforms `os-whitebox` for bug depths up to 61. Our method is able to analyze the preconditions for this game perfectly, the fuzzer never generates illegal arguments to a subaction call. However, it does pick unavailable subactions to call. Which explains why `os-whitebox` is the better fuzzer for higher bug depths.

`rlc-full` and `os-whitebox` are the two best fuzzers for higher bug depths. This matches our expectations, as they are the two fuzzers that make maximal use of the available information. We observe `os-whitebox` performing worse than `rlc-no-fsm` for bug depths smaller than 61. This can be attributed to the fact that when picking an action, `os-whitebox` enumerates all possible legal actions, pushes them into a list and then picks an action from that list. This additional computation does not seem to be amortized for short game traces. However, as the minimum game trace to find the bug gets longer, we can clearly observe the positive impact of this approach. On the other hand, `rlc-full` outperforms either for all bug depths. This is due to the fact that the method we use in RLC does not enumerate all the possible actions and all the possible legal inputs for them. Instead, it only determines the minimum and the maximum for the inputs. Which is an equally descriptive constraint for this game. Therefore, it picks the action inputs from precisely the same set with only a fraction of the overhead. The reduction of this overhead results in the superior performance of `rlc-full`.

Inspecting the average number of executions in Figure 5.1, we observe that the more the fuzzer knows about blackjack, the smaller number of fuzz inputs it needs to try to discover the bug. This supports our hypothesis. We also note that the difference between white-box fuzzers and their black-box counterparts in terms of the average number of tested fuzz inputs is much more dramatic than the difference in terms of fuzzing time. This can be

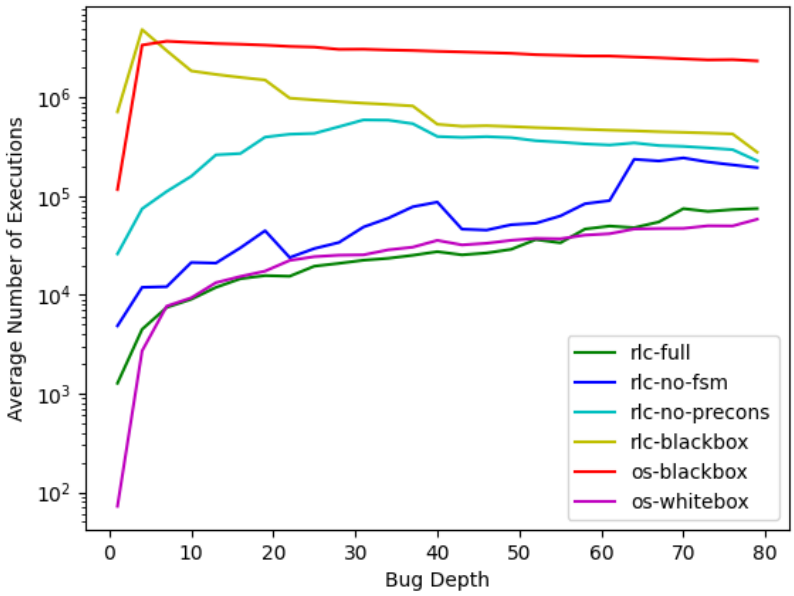

Figure 5.2: Average number of tested fuzz inputs for blackjack.

attributed to the fact that white-box fuzzers have a larger overhead on parsing the given fuzz input into subaction invocations. Hence, they require more time to process a single fuzz input. Lastly, we observe that the number of tested fuzz inputs have a decreasing trend for `rlc-blackbox`, `os-blackbox` and `rlc-no-precons`. This is due to the fact that these three fuzzers exceed the time threshold. The input counts after that point do not represent the number of inputs to find the bug, they represent the number of inputs tested within the threshold. Which has a decreasing trend because individual game traces get longer as bug depth increases.

5.3.2. Tic Tac Toe

Figure 5.4 shows that the fuzzers `rlc-no-precons`, `rlc-blackbox` and `os-blackbox` can not discover the bug within the time constraint even at bug depth 1. However, the RLC fuzzers that perform constraint analysis can find the bug for bug depths up to 31. This illustrates the dramatic impact of analyzing constraints, even if the analysis is limited in scope.

Interestingly, we observe that `rlc-no-fsm` outperforms `rlc-full` for this benchmark. This matches neither our expectation nor the result from the blackjack benchmark. The reason behind this asymmetry is that the RL implementation of tic tac toe only has a single subaction. Therefore, `rlc-no-fsm` is as good at choosing an available action as

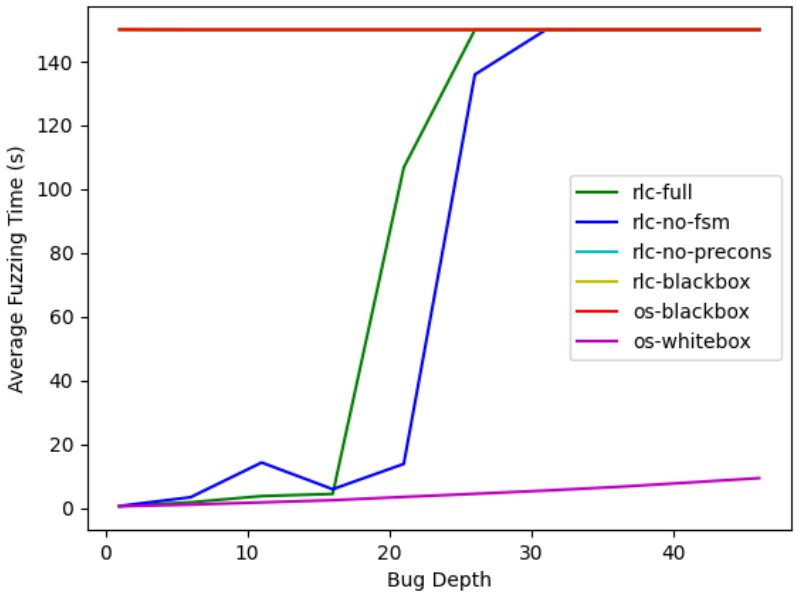

Figure 5.3: Average fuzzing times for tic tac toe.

`rlc-full`. `rlc-no-fsm` performs better because it pick the action with less computation overhead.

On the other hand, `os-whitebox` clearly performs tremendously better than any RLC fuzzer. To explain this, we need to inspect the precondition of the single subaction in the RL implementation.

```
act mark(Int x, Int y) {
    x < 3,
    x >= 0,
    y < 3,
    y >= 0,
    board.get(x, y) == 0
}
```

The technique we describe at Section 4.2.4 can handle the first four constraints but it can not handle the last constraint `board.get(x,y) == 0` since this constraint contains a function call with more than one unbound expression as parameters. It is able to deduce that both variables should be in the range $[0, 2]$, but discards the constraint about not marking already mark spaces. This causes the RLC fuzzers to generate and discard some number of invalid subaction calls. In contrast, `os-whitebox` can understand this last constraint as well and never generates invalid actions. Which explains why the complete whitebox fuzzer is able to significantly outperform the fuzzers generated by RLC.

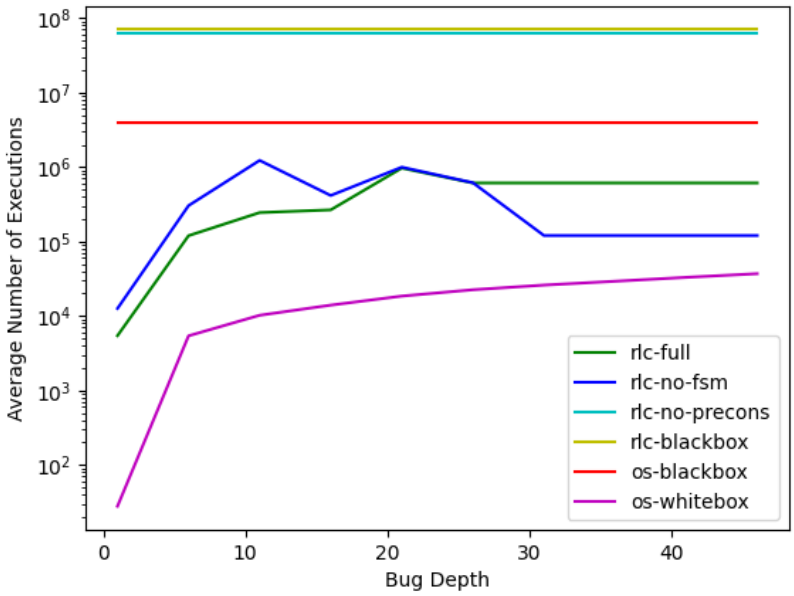

Figure 5.4: Average number of tested fuzz inputs for tic tac toe.

[TODO: write some discussion about the average number of tested fuzz inputs for tic tac toe here once you have the final graph.]

5.3.3. Crazy Eights

6 | Conclusion

Bibliography

[1] M. Fioravanti. Rulebook compiler. `https://github.com/rl-language/rlc`, 2023.

List of Figures

List of Tables

Acknowledgements

Here you may want to acknowledge someone.