Finding Non-Deterministic Constraint Gadgets for Fuzzing

Jared B. Carlson
Principal Researcher
Research Deparment
Veracode
Burlington MA 01803
Email: jcarlson@veracode.com

This proposal is a work-in-progress around our research into finding Non-Deterministic Constraint Logic (NCL) gadgets in code, specifically within LLVM IR. NCL has been used in complexity analysis (see Hearn and DeMaine [2]), especially within the realm of games and puzzles to prove complexity properties. However NCL also appears to be an interesting tool to understand complexity of control flow and underlying logical structure. We will quickly present some tools and findings to find NCL gadgets and present them in diagrams analogous to circuit theory. Well also (quickly) mention how we believe these gadgets can be used to understand code complexity and opportunities to simplifications and analysis.

Note to the reviewer that this is work in progress.

Nomenclature

NCL Non-Deterministic Constraint Logic.
Red Edge Weight 1 edge
Blue Edge Weight 2 edge
Node Generally these nodes should, unless otherwise stated, be considered to have an inflow constraint of at least two.

1 Introduction

Non-Deterministic Constraint Logic (NCL) has been succesfully used to explore the complexity of games and puzzles.

The purpose of this research is to investigate the logical complexity of code and intermediate representations without evaluating the predicates or similar conditions. By treating each logical operation, an *AND* and *OR* as primitive examples, as an NCL gadget we can research the complexity and possible reductions and similarities without evaluating the predicate conditions.

The control flow can be modelled generally with clause, door and related gadgets. We can follow the ideas in [1], where game complexity is evaluated using some of these gadgets to model paths the character might take in various games (Pac-Man, Super-Mario Bros, etc.) and their decisions entail.

The benefits of this work are the ability to investigate logical complexity without performing any evaluations, rather assuming that any valid configuration is equally so and therefore possible.

2 Forming NCL Gadgets

If we start from a simple fibonacci implementation (in C for example, see Appendix A for the bulk of the IR), the LLVM IR has a relatively simple control flow, as seen in figure 1.

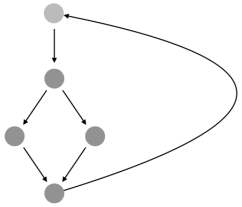

Fig. 1. Simple representation of the control flow in a fibonacci program

This control graph can be modeled using *DOOR* and *PATH* traversal gadgets. In this case the recursion can be thought of as a path that leads to a door that can lead back to the same door. The path, can be thought of a penrose stair, as seen in the classical Penrose image, see figure 2.

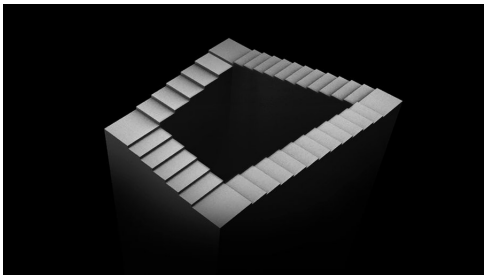

Fig. 2. Classical Penrose Stair

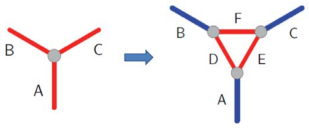

Fig. 3. Two equivalent constructions of a CHOICE gadget

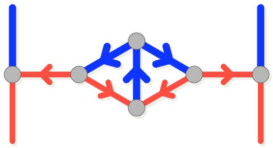

Fig. 4. A red/blue conversion gadget, allowing us to connect arbitrary edges

The basics of logical branching and their dependencies then rests on our ability to use primitive *AND*, *OR*, and *CHOICE* gadgets. The *CHOICE* gadgets uses an input edge that if active only one of two the possible other edges is an output (as the node is constrained to have at least a minimum inflow of 2), following [2] where the interested reader can find an excellent exposition of NCL and it's applications to puzzles and games.

To join these gadgets we'll need to a converter from red to blue (generally representing a change in edge weight - in this case, red to blue or blue to red). This is done via a gadget as seen in figure 4.

It's important to note that the actual construction of gadgets is somewhat arbitrary, similar to an implementation of an algorithm. If we disregard efficiency, then as we've seen in figure 3 while one *CHOICE* gadget might have more edges and nodes than the other, they are equivalent. This provides a basis for evaluating similar implementations. We don't have to be constrained by a particular implementation, we can hope to recognize similar implementations of a *CHOICE* gadget for instance. Some common gadget conversions can be seen in 1.

2.1 Constraints

An astute reader, or one already familiar with NCL might be realize that NCL formulated with a minimum node inflow (or in parlance, an "at least" condition), that a *NOT* condition is not possible in this formulation. To have a *NOT*, an upper bound must be present.

To solve this we use *CLAUSE* gadgets for each path, namely if we have a choice between two states, $\neg x$ and x then we have a clause gadget for each. A *CHOICE* gadget is relatively simple and we can extend to any number of choices, allowing us to range from a simple binary choice, i.e. an *if-else* or a one-to-many branch point, i.e. a *switch* statement.

Table 1. Some common conversions to NCL Gadgets

Code	IR	Gadget
if-else	OR	OR
and	AND	AND
switch	one-to-many branch	CHOICE
for-loop	cycle	PATH

2.2 IR to Gadgets

Basic Gagdets of *AND* and *OR* are necessary to model disjoint or conjoined domains. We also need a few conversion gadgets to join edges. A blue edge (weight 2) and red edge (weight 1) can be joined in the following gadget.

As we look at basic blocks and branching in IR, we group blocks into clauses, that can be joined or disjoint as needed. Each choice can be modelled as a generic *DOOR* gadget, with *PATH*s leading from *DOOR* to *DOOR*. This allows us to lift IR to NCL gadgets with increasing specificty as we analyze the topology of the control flow. For example, a door that has a prerequisite can be modelled as a door with a *LATCH* gadget. It's worth pointing out that at least at a very high level, the idea of pointer-as-implementation or a private interface is similar to a *DOOR* and *LATCH*, i.e., some aspects of an interface might be "protected".

2.3 IR to Higher Level Structures

When we combine these gadgets we can uncover interesting structures. For example, we can find similar structures to the aforementioned Fibonacci implementation as other recursive algorithms, as seen in figure 5.

3 Complexity Analysis

Complexity analysis can be done via NCL, generally proving PSPACE complexity. However, what is more interesting to us is exploring the configuration space of gadgets around branch points. As NCL explores the legality of edges flippoing orientations of these edges.

4 Implementation

Our current implementation [1]attempts to automate finding NCL gadgets by querying the basic block topology typ-

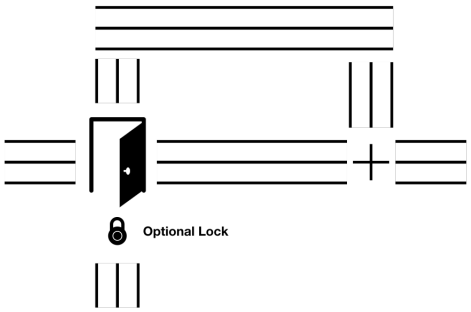

Fig. 5. The PATH and DOOR representation of recursive calling

ically used in CFG's. We add predicate operations (such as AND, OR, etc) but do not evaluate them. This is done to find domain boundaries for the basic blocks and their relational dependencies, i.e. to construct inflow or outflow edges as best we can.

5 Fuzzing

Identifying gadgets in code provides a means of understanding both what to fuzz as well as how to fuzz them. Gadgets provide means to looking at where code complexity lies, as well as how to expose it. For instance, high complexity code is likely more error prone than other areas in the program and both defensive and offensive minded practitioners will want to isolate these areas for in-depth research.

The static analysis we've presented, albeit mostly at a theoretical level for the most part, could be realtively easily coupled to dynamic analysis (logging when and where a gadget has been reached and/or exercised). This utlimately provides our ultimate goal.

6 Conclusions

This paper lays out the theoretical approach to find NCL gadgets in code, specifically we are investigating finding gadgets in LLVM IR. Our current approach is intepretive but one of the advantages of IR that we hope to take advantage of is transforming the IR (or perhaps lifting it) to construct relevant gadgets.

We hope that non-derterminisitc gadets might eb an intermediate step in caculating cmoplexity in given implementations as well as discovering deeper structures.

Acknowledgements

The author would like to thank the Research Group at Veracode, with special thanks to Andrew Reiter, `reiter@vercode.com` for his encouragement and discussions of NCL and possible applications.

[1] Please note that our implementation is a work-in-progress. We're happy to present what we have and will likely post code to GitHub but the bulk of the proposed talk is to discuss the theoretical interests.

References

[1] Viglietta, G. "Gaming is a hard job, but someone has to do it!". *Journal*.

[2] Hearn, R., and Demaine, E., 2009. *Games, Puzzles, and Computation*. A K Peters/CRC Press, Boca Raton, FL.

Appendix A: Fibonacci IR

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %n = alloca i32, align 4
  %first = alloca i32, align 4
  %second = alloca i32, align 4
  %next = alloca i32, align 4
  %c = alloca i32, align 4
  ...
  br label %for.cond

for.cond:
; preds = %for.inc, %entry
  %1 = load i32, i32* %c, align 4
  %2 = load i32, i32* %n, align 4
  %cmp = icmp slt i32 %1, %2
  br i1 %cmp, label %for.body, label %for.end

for.body:
; preds = %for.cond
  %3 = load i32, i32* %c, align 4
  %cmp3 = icmp sle i32 %3, 1
  br i1 %cmp3, label %if.then, label %if.else

if.then:
; preds = %for.body
  %4 = load i32, i32* %c, align 4
  store i32 %4, i32* %next, align 4
  br label %if.end

if.else:
; preds = %for.body
  %5 = load i32, i32* %first, align 4
  %6 = load i32, i32* %second, align 4
  %add = add nsw i32 %5, %6
  store i32 %add, i32* %next, align 4
  %7 = load i32, i32* %second, align 4
  store i32 %7, i32* %first, align 4
  %8 = load i32, i32* %next, align 4
  store i32 %8, i32* %second, align 4
  br label %if.end

if.end:
; preds = %if.else, %if.then
  %9 = load i32, i32* %next, align 4
```

Appendix B: Head of Second Appendix

Subsection head in appendix

The equation counter is not reset in an appendix and the numbers will follow one continual sequence from the beginning of the article to the very end as shown in the following example.

$$a = b + c. \tag{1}$$