Concolic Execution for Fuzzing in Julia

Final project for 6.858 (Spring 2018)

Valentin Churavy

JuliaLab@CSAIL

16. May 2018

Table of Contents

Concolic Execution

Welcome to Lab 3

Concolic Execution

Welcome to Lab 3

... but in Julia!

Concolic Execution

Welcome to Lab 3

... but in Julia!

General strategy

- Generate symbolic trace of the program for a given input

Based on [4, 3]

Concolic Execution

Welcome to Lab 3

... but in Julia!

General strategy

- Generate symbolic trace of the program for a given input
- Feed symbolic trace to Z3 to generate inputs that cause exploration of unvisited branches

Based on [4, 3]

Concolic Execution

Welcome to Lab 3

... but in Julia!

General strategy

- Generate symbolic trace of the program for a given input
- Feed symbolic trace to Z3 to generate inputs that cause exploration of unvisited branches
- Iterate until all branches are visited, record which branches throw exceptions

Based on [4, 3]

Julia in brief

Julia [1] is a dynamic high-performance language, it looks like Python, runs like Fortran, and talks like Lisp. It has interesting language design properties, like a rich type system, multiple dispatch, macros, and staged programming. It uses a compiler based on LLVM[5].

Julia in brief

Julia [1] is a dynamic high-performance language, it looks like Python, runs like Fortran, and talks like Lisp. It has interesting language design properties, like a rich type system, multiple dispatch, macros, and staged programming. It uses a compiler based on LLVM[5].

This is not a Julia tutorial!

I am going to talk about a couple of advanced concepts and use a development version of Jula with experimental tooling!
If you want to learn more about Julia, ask me later!

Julia in brief

Julia [1] is a dynamic high-performance language, it looks like Python, runs like Fortran, and talks like Lisp. It has interesting language design properties, like a rich type system, multiple dispatch, macros, and staged programming. It uses a compiler based on LLVM[5].

This is not a Julia tutorial!

I am going to talk about a couple of advanced concepts and use a development version of Jula with experimental tooling!
If you want to learn more about Julia, ask me later!

AD in 5s

Example time!

Implementation

- Goal: Be able to deal with arbitrary user code

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic
- Issue: The trace is linearized and no longer contains control-flow

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic
- Issue: The trace is linearized and no longer contains control-flow
- Solution: Rewrite and rejit user-provided functions and instert calls to the assert function, which records conditionals that are used as part of the control-flow.

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic
- Issue: The trace is linearized and no longer contains control-flow
- Solution: Rewrite and rejit user-provided functions and instert calls to the assert function, which records conditionals that are used as part of the control-flow.
- Filter concolic trace to get a trace that only depends on the tainted arguments

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic
- Issue: The trace is linearized and no longer contains control-flow
- Solution: Rewrite and rejit user-provided functions and instert calls to the assert function, which records conditionals that are used as part of the control-flow.
- Filter concolic trace to get a trace that only depends on the tainted arguments
- Cut trace at branch and negate branch condition

Implementation

- Goal: Be able to deal with arbitrary user code
- Trace an invocation of a function, e.g. record recursivly, which calls occur and which input + output arguments they have.
- Mark/Taint input arguments as symbolic
- Issue: The trace is linearized and no longer contains control-flow
- Solution: Rewrite and rejit user-provided functions and instert calls to the assert function, which records conditionals that are used as part of the control-flow.
- Filter concolic trace to get a trace that only depends on the tainted arguments
- Cut trace at branch and negate branch condition
- Translate symbolic trace to SMT2/Z3, if satisfiable use the input value to continue exploration

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler
- Usage for this project

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler
- Usage for this project
  - Trace generation through pre-/post-hooks on every function call

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler
- Usage for this project
  - Trace generation through pre-/post-hooks on every function call
  - Rewrite IR to inject assert statements

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler
- Usage for this project
  - Trace generation through pre-/post-hooks on every function call
  - Rewrite IR to inject assert statements
  - Taint propagation through the metadata system of Cassette

Solving the type-constraint problem with Cassette

One of the challenges was to be able to taint arbitrary user-code, including user code that has type-constraints.

Cassette

- Cassette.jl [6] developed here at MIT provides a framework to extend Julia with non-standard execution models, like AD or symbolic execution, through contextual dispatch
- User-code is executed within a *context* in which the context author can provide alternative *primitives* and can dynamically inject passes into the compiler
- Usage for this project
  - Trace generation through pre-/post-hooks on every function call
  - Rewrite IR to inject assert statements
  - Taint propagation through the metadata system of Cassette
  - Context specific primitives of operations (like addition) to propagate taint or to create tain (rand)

Extensible design: rand

One problem in concolic execution is how to deal with random variables, concolic execution depends on being able to *deterministically* execute programs. *rand* is implemented as a primitive that generates taint and the return value of a seen execution of *rand* can be controlled and during fuzzing Z3 will generate values for *rand* much in the same way it does for other inputs.
This could be extended to other foreign-calls like reading from the filesystem.

Examples

- Proof some properties
- Fuzzing!
- Loops, structs, random variables

Future-work: Features

- Reasoning about the type-domain

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays
- Taint analysis with non-integer inputs

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays
- Taint analysis with non-integer inputs
  - Strings

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays
- Taint analysis with non-integer inputs
  - Strings
  - Structs

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays
- Taint analysis with non-integer inputs
  - Strings
  - Structs
  - Floating-point values

Future-work: Features

- Reasoning about the type-domain
- Taint analysis through Arrays
- Taint analysis with non-integer inputs
  - Strings
  - Structs
  - Floating-point values
- Arithmetic on Bool

Future-work: Tooling

- Using the Klee query language

Future-work: Tooling

- Using the Klee query language
- Maybe deeper integration with Klee[2]

Future-work: Tooling

- Using the Klee query language
- Maybe deeper integration with Klee[2]
- Only intercept intrinsics and built-ins instead of higher level methods

Future-work: Tooling

- Using the Klee query language
- Maybe deeper integration with Klee[2]
- Only intercept intrinsics and built-ins instead of higher level methods
- Better analysis of function to taint usage of global variables and foreigncalls

Where to find this work

`https://github.com/vchuravy/ConcolicFuzzer.jl`

Where to find this work

`https://github.com/vchuravy/ConcolicFuzzer.jl`

If you want to learn more about Julia!

1. Write me an e-mail or go to `https://julialang.org`
2. Find me in 32-G785

Bibliography I


Jeff Bezanson et al. 'Julia: A Fast Dynamic Language for Technical Computing'. In: (Sept. 2012). arXiv: 1209.5145 [cs.PL].


Cristian Cadar, Daniel Dunbar and Dawson Engler. 'KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs'. In: (). URL: http://hci.stanford.edu/cstr/reports/2008-03.pdf.


Cristian Cadar et al. 'EXE: Automatically Generating Inputs of Death'. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335.


Patrice Godefroid, Nils Klarlund and Koushik Sen. 'DART: Directed Automated Random Testing'. In: *SIGPLAN Not.* 40.6 (June 2005), pp. 213–223.

Bibliography II


Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.


Jarrett Revels. *Cassette.jl*. 2018. URL: https://github.com/jrevels/Cassette.jl.