Concolic Execution in Julia

Final project report for 6.858 (2018)

Valentin Churavy
JuliaLab@CSAIL
vchuravy@csail.mit.edu

1 A short introduction to Julia

The Julia Language [1] is a dynamic programming language that aims to square the circle of provide high-performance whilst also being high-level. The language is build around a JIT compiler based on LLVM [5]. While the language semantics are dynamic, extensive type-inference let's Julia compile and optimize functions that are proven to be type instable. Julia's unit of compilation are method and each method is specialised on a given set of argument types. While most of the compiler is written in C/C++, the language implementation and runtime is written in Julia itself. And bootstrapped on a very small set of intrinsics and built-in methods. Julia has a rich type system, metaprogramming support, uses multiple-dispatch, and has a large set of foreign-function interfaces.

2 Goals

I set out to implement a concolic execution engine for Julia, with a focus on proving application specific invariances, I followed the general ideas set out in the third problem set and papers [4, 3]. My original goals as layed out in the proposal were:

1. Given a particular method to analyze, recursively build a graph of methods that are called (terminating the recursion at intrinsic functions)
2. Collect set of asserts (asserts can either be assumptions or invariants to be proven)
3. Implement symbolic execution scheme for primitive data-types and intrinsic functions
4. Extend symbolic execution scheme for user-defined immutable and mutable datatypes
5. Propagate set of assumptions through the call graph, instead of attempting to prove general correctness
6. Add concolic execution for all cases that we can't handle symbolically

3 Implementation

Julia has several different intermediate representation on which one could build a symbolic execution system. The important ones are the untyped AST, the typed AST and LLVM IR. The latter would provide an interesting opportunity to integrate with KLEE [2]. The untyped/typed AST is currently under heavy development and quite a complex target. After initial experimentation I decided against building a symbolic execution system for Julia and narrowed the focus of this project. The approach I finally decided to go with is a taint-based concolic first approach, that suits itself well for guided fuzzying.

For both tainting and trace generation I use an extension to the Julia language that provides contextual dispatch called Cassette [7]. Cassette allows for non-standard execution of Julia code and the dynamic insertion of code-transformation passes to compile arbitrary user code. The motivation for Cassette originally originated in the context of automatic-differentiation [6]. As an example take the following code-snippet:

```
# D is a pair of the concrete value and the derivative
struct D <: Number
    f::Tuple{Float64, Float64}
end
import Base: +, /, convert, promote_rule
+(x::D, y::D) = D(x.f .+ y.f)
/(x::D, y::D) = D((x.f[1]/y.f[1], (y.f[1]*x.f[2] - x.f[1]*y.f[2])/y.f[1]^2))
convert(::Type{D}, x::Real) = D((x,zero(x)))
promote_rule(::Type{D}, ::Type{<:Number}) = D

function Babylonian(x; N = 10)
    t = (1+x)/2
    for i = 2:N
        t=(t + x/t)/2
    end
    t
end
x=pi; Babylonian(D((x,1))), (sqrt(x), 0.5/sqrt(x))
# (D((1.7724538509055159, 0.28209479177387814)),
#    (1.7724538509055159, 0.28209479177387814))
```

Using this minimal implementation of forward-mode auto-differentation we can calculate the derivative of any generic user-defined function (as long as they only use plus and division). In this examle we use the babylonian method for the calculation of the square-root. This approach works spendidly, but trouble arises if they user provides a function that is type-constrained, e.g.

```
Babylonian(x::Float64, N = 10)
```

, then we no longer thread our Dual number into the user-provided function. A similar problem would arise in the context of symbolic-execution, while we easily could define a symbolic type that records all operations done on it, every type annotation would create a blackbox. Contextual-dispatch allows us to sidestep this issue, by defining a non-standard execution in which we rewrite type-annotated functions to be permissive for our Dual type. In actuality Cassette defines a metadata propagation system that is used in situation like this. Another important feature is that contextual-dispatch occurs within a context and does not modify the code running outside the context, code transformations are not visible to the user.

3.1 Concolic execution – Generation of traces and tainting

Initally we mark input/taint arguments – in Cassette parlance, they are boxes with associated metadata – and propagate them through the function. Each operation that takes a tainted argument has to mark it's output as tainted (see section 4 for the limitations of the current prototype). In parallel we record a trace of the current execution in form of nested callsites (callsites are function names, with arguments and outputs) This execution occurs over a set of concrete values, but the trace are concolic, e.g. if a value was tainted only the symbolic variable name is recorded, otherwise the concrete value is recorded.

The recorded trace can then be flattened (removal of all non-leaf nodes, this action is corresponding to inlining) and filtered (removal off all nodes that have not at least one symbolic component), these two proceddurs yield a linearized control-flow stream. Streams have the property that they still contain all operations that use tainted values, e.g. values that are under an attacker control. It is possible to add additional sources of taint for an example see section 3.3.

3.2 Assertions

Since streams and traces follow concrete executions of a function, they no longer contain information about the control flow. ConcolicFuzzer.jl therefore provides the *assert* function to manually insert user-defined invariances. Since manually annotating every if-condition would be a nuisance, it uses a Cassette pass to insert an assertion before every branch. Branches are represented as the *goto_if_not*

statement in the Julia AST. This is implemented in the *InsertAssertsPass*, and allows us to capture the conditional of the branch and its value in our trace.

3.3 Other sources of taint: rand

Another source of control-flow variation and a case in which the input based tracing is too limited, in contrast to full symbolic execution, is the usage of *rand* to generate random values within a traced function.

rand is defined as a primitive function in the context of ConcolicFuzzer.jl. The primitive taints the return value and checks within the trace local context if a value is prescribed.

The latter feature allows for deterministics execution of traces and it provides an avenue for the fuzzer to drive execution, the values coming from *rand* are represented as free variables SMT2 and the fuzzer uses Z3 to generate values that will cause control-flow divergence within the trace. This approach is extensible for other sources of non-determentistic values (that are not controlled through arguments), like calls to the operating system, the runtime system, or arbitrary libraries that are called through a FFI.

3.4 Using symbolic traces to generate input values

The current implmentation only supports a limited number of integer programs, due to the fact that the translation of traces/streams to SMT2 is rudimentary. In the linearized stream assertions occur sequentially. The main fuzzer loop starts at a depth of 0 with an initial guess at the parameters. The execution either fails (in which case we found a bug) or succeeds and the program obtains a trace, with n branches. Starting at the first branch we cut the subsequent trace and negate the branch, then we convert the stream to SMT2 and let Z3 provide us a set of input arguments that satisfy the condition. If it is unsatisfiable we have shown that we can't reach that branch in the opposite direction and we remove it from the candidate list. If it was satisfiable we found a new candidate that we add to workqueue. We continue to iterate until the workqueue is empty and we have explored all branches in the program that are controllable by the input arguments.

4 Limitations

ConcolicFuzzer.jl is currently limited to integer programs and the two main bottleneck is that all primitive/leaf functions need to be enumerated and they need to be translated to repsective SMT2 expressions. Further work needs to be gone to use SMT2 BitVectors and add support for floating-point value. The original goal was to only declare intrinsics (about 100) and built-in functions (about 24) as primitives, but for expedience sake a limited set of integer operations was defined as primitives.

Additionally functions that return tuples and where callsites use desugaring to extract the values are unsupported.

5 Conclusion

The code is available on Github at `https://github.com/vchuravy/ConcolicFuzzer.jl` under the MIT license. Please check the examples in the test-suite at `https://github.com/vchuravy/ConcolicFuzzer.jl/tree/master/test`.

To run the code please check the README, it contains the versions that of Julia and Cassette that this code, was developed against.

References

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

[2] 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`.

[3] 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.

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

[5] 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–.

[6] Jarrett Revels. *Capstan.jl*. 2018. URL: `https://github.com/JuliaDiff/Capstan.jl`.

[7] Jarrett Revels. *Cassette.jl*. 2018. URL: `https://github.com/jrevels/Cassette.jl`.