A Lightweight Symbolic Execution Framework for Ruby-on-Rails

Gowtham Kaki *

305 N University St., Purdue University, USA

gkaki@cs.purdue.edu

1. Introduction

Ruby-on-Rails, a web programming framework written in Ruby, has gained wide adoption among Web 2.0 applications, such as Twitter, Airbnb, Hulu, Groupon, and SoundCloud. One of the factors that led to the success of Rails framework is its singular focus on facilitating the expression of complex web functionality via succinct code. For instance, Selfstarter [8], a popular Rails application that implements end-to-end functionality of a crowdfunding site, including the integration with Amazon Payments, was implemented in less than 600 lines of Ruby. While Rails's philosophy of prioritizing "convention over configuration" [7] contributes to its expressivity, the main facilitator, however, is the meta-programming support offered by Ruby that enables rich functionality, such as reflection, run-time code generation, singleton objects, and also the much-maligned *eval* [5] function.

Indeed, meta-programming features are available in many mainstream languages [6, 10]. However, their usage in mainstream programming is not as prevalent as it is in Ruby. For example, the idiomatic way to write a `Rectangle` class in Ruby is to rely on the inbuilt `attr_accessor` meta-function to introduce fields named `length` and `breadth`, and define their setters and getters:

```
class Rectangle
  attr_accessor :length, :breadth
end
```

Moreover, unlike the static languages, where meta-functions are "evaluated away" at the compile time, Ruby is unapologetically dynamic (no just-in-time compilation; all code is interpreted at run-time), and makes no distinction between object-level code and meta-level code. The dynamic nature of Ruby complicates the task of building formal analyses for the language, which perhaps explains why there are relatively few proposals for static analysis of Ruby code, despite the language not being inherently safer than other dynamic languages.

In this paper, we propose MAGLEV, a lightweight symbolic execution framework for Ruby that also performs partial evaluation. MAGLEV accepts a Ruby program, complete with meta-functions and their applications, as its input, emits an equivalent Ruby program[1], where all applications of meta-level functions are evaluated away. The language of the generated program is a small subset of Ruby that is amenable to static analysis. MAGLEV thus simplifies the task of building static analyses for Ruby by preempting the need to reason about meta features in an analysis-specific way.

* Research Advisor: Suresh Jagannathan, Category: Graduate.

[1] We do not yet have a formal proof for this claim.

```
class Micropost < ActiveRecord::Base
  belongs_to :user
  validates :content, presence: true,
            length: { maximum: 140 }
  validates :user, presence: true
end
```

Figure 1. Micropost class (model) of Microblog Rails Application

MAGLEV is lightweight in the sense that it is a language library (a Ruby "gem") rather than a language implementation. MAGLEV was designed to coexist with an unmodified Rails library, and rely on a concrete Ruby interpreter to perform symbolic execution. We believe that this choice is appropriate for a fast-evolving language like Ruby, whose semantics are only vaguely defined.

Chaudhuri and Foster [1] have previously proposed a symbolic execution for Ruby-on-Rails tailor-made for their security analysis. Their approach relies on a custom-built symbolic interpreter that generates verification conditions. To the best of our knowledge, MAGLEV is the first symbolic execution engine for Ruby-on-Rails that works off an off-the-shelf interpreter, and generates Ruby code that can be consumed by other analyses.

2. Motivating Example

Fig. 2 shows the implementation of `Micropost` class taken from a Microblogging application developed using Rails [4]. The class extends Rails's `ActiveRecord::Base` class which implements Rails's Object-Relational Mapping (ORM) system. Almost the entire functionality of `Micropost` class was implemented using meta-functions offered by `ActiveRecord::Base`. For example, instead of implementing a custom logic to save a post to the database, `Micropost` class relies on the generic `save` method provided by `Base` class of `ActiveRecord`, but customizes it by specifying additional constraints via the meta-function `validates`. The result is a `save` method that performs additional checks (for eg., content length $\leq$ 140 characters) before saving the post to the database. However, the customized `save` method never really exists; it's logic is generated on-the-fly when the `save` method is called on a concrete `Micropost` object. This makes it difficult for humans to understand the program, and for the analyses to verify it.

MAGLEV runs the `save` method of `Micropost` on a symbolic input, and generates its new definition as shown in Fig. 2 (simplified for clarity). Observe that the generated definition has no meta-function applications, is self-contained, and easier to understand and analyze.

```
def save(post)
  v0 := post.id
  v1 := post.content
  v2 := post.user_id
  SQL (begin transaction)
  v6 := (v1.length)<=140
  if (v6==true) then
    v7 := SQL (SELECT  "users".* FROM "users"
                WHERE "users"."id" = v2)
    v9 := v7.map do |v8|
      {id => v8.id, name => v8.name}
    end
    v10 = v9.first
    v11 = v10==nil
    if (v11==true) then
      SQL (rollback transaction)
    else
      v12 := SQL (INSERT INTO "microposts"
          VALUES (v1, v0, v2))
      SQL (commit transaction)
    end
  else
    SQL (rollback transaction)
  end
end
```

Figure 2. Micropost class (model) of Microblog Rails Application

3. Key Ideas Underlying MagLev

MagLev is based on some fundamental observations about Ruby in general, and Ruby-on-Rails in particular. First, unlike in multi-stage programming, Ruby has no stratification of the code that consumes application's inputs and the code that consumes the application itself. Under this context, symbolic execution serves as an effective partial evaluation technique. Second, Ruby adopts *duck typing*, meaning that it does not distinguish between two objects that respond to same messages (method calls), even when they are constructed from different classes. This applies even to the objects of core types, such as integers and strings. Consequently, a Ruby function expecting a concrete integer input can instead be run on a symbolic integer as long as the later behaves like the former[2]. To exploit this observation, MagLev framework defines symbolic counterparts of all core classes that respond to the same methods as core classes. However, instead of a concrete result, the symbolic methods return new symbolic values, which may then flow into other (symbolic or concrete) methods. MagLev assigns a unique symbolic AST node to each symbolic value, and traces (i.e., logs to trace) the relationship between new symbolic AST nodes and existing nodes. Third, Ruby's reflection features allow control over class and method bindings at the run-time (Even those of core types[3]). Coupled with the fact that "all data are objects" and "all control are method applications" in Ruby, the semantics of any application can be controlled from the application space itself.

In MagLev, symbolic methods that are expected to return either `true` or `false` do so by initially making an ambivalent choice, and later backtracking to explore the other choice. For this purpose, we implement an `amb` function similar to McCarthy's `amb` [2], except that our `amb` relies on Unix processes rather than continuations to checkpoint and restore the state [9]. Unix processes are needed because restoring a stored continuation does not rollback side-effects on heap. Whenever we make an ambivalent choice in method `f` of a symbolic value `v`, we write an `if` expression to the trace (eg., `if (v.f() == true) then ... end`). The symbolic execution continues in a new (child) process, tracing under the `then` branch, while the parent process waits. Once the child process returns, the parent process resumes execution, assuming `v.f()` as `false`, and traces under the `else` branch. While this approach does not generalize to arbitrary loops, such loops are very rare in Ruby. The idiomatic way to write a loop in Ruby is to use a higher-order combinator, such as `map` and `each` on arrays, and `times` on integer (for eg., `n.times f()` will execute `f()` `n` times). Since higher-order combinators are methods, they too can be traced and made to return a symbolic value, just like other methods.

The result of symbolic execution is a trace that contains a slice of the original program that is data-dependent on its inputs. Any interactions with the environment (eg., database I/O) are also traced, given that such interactions happen via a symbolic adapter.

4. Implementation and Experiments

MagLev is implemented as a Ruby library that works on top of an off-the-shelf Ruby interpreter (We used MRI [3] for development). From an engineering perspective, leveraging an off-the-shelf Ruby interpreter and relying on Unix processes for heap checkpointing and restoration enabled an implementation comprising roughly only 1300 lines of Ruby code, packaged as a Ruby gem[4].

We used MagLev to compute summaries of various operations supported by applications similar to the one described in § 2. Although the complexity being exponential in terms on number of branches, MagLev was able to compute method summaries within 2s.

It is possible that the Ruby code generated by MagLev runs faster than the original code that involves calls to meta-functions. We have not tested this hypothesis, and we plan to explore this direction in the near future.

References

[1] A. Chaudhuri and J. S. Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 585–594, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. . URL `http://doi.acm.org/10.1145/1866307.1866373`.

[2] J. McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238, New York, NY, USA, 1961. ACM. . URL `http://doi.acm.org/10.1145/1460690.1460715`.

[3] MRI. Matz's Ruby Interpreter, 2016. URL `https://github.com/ruby/ruby`. Accessed: 2016-3-11.

[4] Rails Tutorial. A Sample Microblog Application, 2016. URL `https://github.com/railstutorial/sample_app_rails_4`. Accessed: 2016-3-21.

[5] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL `http://dl.acm.org/citation.cfm?id=2032497.2032503`.

[6] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. *Commun.*

[2] This is only true as long as the control does not escape the boundary of Ruby and C, the language in which certain core functionality is implemented.

[3] `a+b` is treated as `a.+(b)` in Ruby. This is a problem if `a` is concrete and `b` is symbolic. To circumvent this problem, We override (at run-time) the definition of concrete integer addition method so as to treat `a+b` as `b+a`.

[4] Source code available at `https://github.com/gowthamk/conflict_analysis`

ACM, 55(6):121–130, June 2012. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/2184319.2184345.

[7] Ruby on Rails. Convention over Configuration, 2016. URL http://rubyonrails.org/. Accessed: 2016-3-20.

[8] SelfStarter. A starting point for building an ad-hoc crowdfunding site., 2016. URL https://github.com/lockitron/selfstarter. Accessed: 2016-3-20.

[9] StackOverflow. Checkpoint and Restore the Heap in Ruby, 2016. URL http://stackoverflow.com/a/35346622/2016967. Accessed: 2016-3-9.

[10] W. Taha. *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*, chapter A Gentle Introduction to Multi-stage Programming, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25935-0. . URL http://dx.doi.org/10.1007/978-3-540-25935-0_3.