Optimal Taint: Update and Next Steps

José Cambronero
jose.cambronero@cs.nyu.edu

February 24, 2016

1 Updates

1.1 Random program generation

The randomly generated programs now adhere to the core syntax fully (the issue with scopes was resolved).

I had to make some tweaks to the random program generation to make sure things worked out for the benchmarking. Namely, random programs generated now have minimum numbers of commands executed, to avoid having random programs with tons of code but not much done (e.g. a single if statement at the top-level that has an empty branch, and takes that branch).

I also limited the execution time of random programs generated, so they need to complete execution in a maximum of 2 seconds (and 10 seconds for compilation). Microbenchmarks have to run within 10 seconds for Google Caliper not to throw an exception under the default configuration. I occasionally still have issues when running the instrumented code (since it can be significantly slower than the none instrumented). I'm trying to figure out how to modify the configuration, but documentation is pretty bad.

Both of these are enforced by generating a program, testing these conditions, and if it fails, regenerate with a new random seed.

1.2 Benchmarking

Execution time benchmarking is now setup. You can check out a simple visualization of dummy results of some sample experiments I ran on my laptop, you can access it here (setup as an R Shiny web app). I thought this might be a good way for us to share results in a more interactive way.

In general, the results are broken down along various factors

- Number of variables declared
- Minimum number of non-skip commands in the code
- Type of instrumentation (none, naive)

You can see ratios of execution times and instruction counts in the bytecode. I also included some sample (not necessarily the ones corresponding to timings here) files, so you can see what the random code being generated looks like.

Given comments in Section 2.1, all these measrements should be taken with a grain of salt. If you'd like to see different graphs, let me know.

2 TODOs

2.1 Tracking

I had a conversation with Jon, and he mentioned that Phosphor currently incurs an overhead on **all** variables, not just those that are explicitly instrumented. While this is fine for our naive strategy, since we instrument all variables (although we do pay an additional penalty on the loop variables, which is not necessarily intended), it certainly doesn't work for the optimal strategy.

2.2 Benchmarking

I need to setup memory benchmarking. I spent the better part of two days trying to set up the memory measurements in Caliper, but there seem to be documented issues there (and unresolved).

Jon suggested using MXBean, which is what they used for the Phosphor paper (although not super precise). I have not yet tried this.

2.3 Core Syntax to Z3

We currently have a Java implementation for a random-code generator that adheres to the core-syntax. The next couple of steps (in my mind are): parsing random code, unwinding loops, transformation to SSA, trace (and thus branching condition) collection, and translation to z3. I provide some info on each below. Note that each section is listed in the order in which the tasks have to be performed (with the results of one step feeding into the next).

2.3.1 Parse

We should parse the random code generated, since we'll need to traverse the AST for various of the tasks that remain. We could arguably do without this, and perhaps generate all necessary info during the initial code generation, but I think this will eventually become too complicated and seems to mix completely disparate tasks.

2.3.2 Loop unwinding

I read a bit into bounded model checking ([2], [3]), since it seemed to provide some good background for our task (translating source code to first-order logic and solving with SMT). The way they handled loops was to unroll the while loops into a sequence of k if-statements. In their case, the k was a user provided input in the analysis. In our case, all our while loops are of the form

```
int loop_var = expr;
while (loop_var < n) { ...; loop_var++;}
```

So we are guaranteed termination. If we decide to place a floor of 0 on the values of loop variables, then we can also have a loop-specific k that guarantees our loop-unwinding to be an overapproximation of the number of iterations.

In our AST, we must replace a while-loop node with a sequence of if-statements, recursively translating the body of each if-statement (as these might in turn have loops).

2.3.3 Static Single Assigment

In order to appropriately express the source code in first order logic, each assignment should be unique, so we should translate statements like

```
x = 1;
y = x;
x = x + 1;
```

into

```
x_1 = 1;
y_1 = x_1;
x_2 = x_1 + 1;
```

along with changing appropriate references. In branching situations, where there might be two possible values of a variable x, we can unify at the point in which the two branches join once again by using a conditional statement of the form $\phi(x', x'')$, which returns the appropriate x based on the condition associated.

So our AST should now have a) modified variable names (and references to them), b) inserted new join nodes of the form $\phi(x, y)$, which should have three fields: *condition*, *variable name if true*, *variable name if false.*

2.3.4 Branching conditions collection along traces

We need to collect the relevant branching conditions along traces, and partition these sequences of branching conditions based on the results of a query. This seems straightforward enough.

Separately we collect all variable declarations/definitions as well.

2.3.5 Translation to z3

We translate (in the following order):

- variable declarations, using `(declare-const ....)`, representing the initial environment
- variable definitions, using `(define-fun () Int ...)` . In cases with joins (ϕ), we can use z3's conditional statements `(ite (cond) true_val false_val)`.

- branch conditions along a given trace, using `(define-fun () Bool ..)`
- ϕ^P and ϕ^N by defining new functions that are the disjunction of the appropriate branch condition sequences.
- interpolant computation, using `(compute-interpolant ...)`

2.4 An Example: From Code to Interpolant

Source Code Our example doesn't include while loops, but these would be unwound prior to this translation step. We use `*` to represent the values assigned to a variable by the initial environment σ_0 (which in our random programs are just random integers within a particular range).

```
// intialization of variables (initial environment r_0)
int x = *;
int y = *;
int z = *;

if (z > 0) {
  x = 10;
} else {
  // skip
}

x = x + 3;

if (x > 5) {
  y = x
} else {
  // skip
}
```

Static Single Assignment We need to modify our code into SSA in order to effectively represent the different possible values in the theorem prover.

```
int x_1 = *;
int y_1 = *;
int z-1 = *;

if (z_1 > 0) {
 x_2 = 10;
} else {
  // skip
}

x_3 = ϕ(x_2, x_1); // ϕ condition = z_1 > 0

x_4 = x_3 + 3;

if (x_4 > 5) {
  y = x
} else {
  //skip
}
```

The ϕ here is not to be confused with that relating to branching conditions, but rather is used to join different possible values for a given variable.

Branching conditions per trace We can now collect the branching conditions associated with each trace. Since all of our while-loops are guaranteed to terminate, all our traces are finite.

$$\begin{aligned}
b_{1,1} &= (z_1 > 0) \wedge (x_4 > 5) \\
b_{1,2} &= (z_1 > 0) \wedge \neg(x_4 > 5) \\
b_{2,1} &= \neg(z_1 > 0) \wedge (x_4 > 5) \\
b_{2,2} &= \neg(z_1 > 0) \wedge \neg(x_4 > 5)
\end{aligned}$$

Constructing ϕ^P and ϕ^B Given the query $x \rightsquigarrow y$ we can take the b in the prior section and partition based on whether the query is true or not.

This yields

$$\begin{aligned}
\phi^P &= b_{1,1} \vee b_{2,1} \\
\phi^N &= b_{1,2} \vee b_{2,2}
\end{aligned}$$

Converting to iZ3 Given the assignment nodes in SSA, along with the branching conditions and $\phi^{P|N}$, we are now ready to translate to z3 (and use iZ3 specifically).

```
; initial values
(declare-const x_1 Int)
(declare-const z_1 Int)
(declare-const y_1 Int)

; declare all SSA
; phi conditions translate into ite (if-then-else)
(define-fun x_2 () Int 10)
(define-fun x_3 () Int
  (ite (> z_1 0) x_2 x_1)
)
(define-fun x_4 () Int
  (+ x_3 3)
)

; branch conditions of each trace encoded as functions
(define-fun b_1_1 () Bool
  (and
    (> z_1 0)
    (> x_4 5)
  )
)
(define-fun b_1_2 () Bool
  (and
    (> z_1 0)
    (not (> x_4 5))
  )
)
(define-fun b_2_1 () Bool
  (and
    (not (> z_1 0))
    (> x_4 5)
  )
)
(define-fun b_2_2 () Bool
  (and
    (not (> z_1 0))
    (not (> x_4 5))
  )
)

; define phi^P and phi^N, our interpolant pairs
(define-fun phi_P () Bool
  (or
    b_1_1
    b_2_1
  )
)
(define-fun phi_N () Bool
  (or
    b_1_2
    b_2_2
  )
)
(compute-interpolant phi_P phi_N)
```

Running this in iz3 (linked here with code above), results in the interpolant

```
unsat
(let ((a!1 (or (<= z_1 0) (<= (ite (<= z_1 0) x_1 10) 2)))
      (a!2 (or (not (<= z_1 0)) (<= (ite (<= z_1 0) x_1 10) 2))))
  (not (and a!1 a!2)))
```

3 Questions

- iZ3 returns an interpolant, as shown before. The next step is to identify instrumentation points based on this. My immediate intuition is that I can try to relate the interpolant to the (SSA and unwound) source code by: identifying the variables used in the interpolant, and identifying the branching conditions that use those variables as instrumentation points.

  I'm not immediately clear on whether this is the right approach (I **think** it is sound, but perhaps too much of an overapproximation?), but I plan to think/work through some examples to confirm/refute. I have some readings that I think will help me with this.

- Thoughts on putting a floor of zero on loop-variables (as per argument in Section 2.3.2?
- I'm going to look for other tools that might make sense to use, given the Phosphor details mentioned in Section 2.1, but if you know if any off hand that are worth looking into, let me know.

References

[1] Mike Barnett and K Rustan M Leino. Weakest-precondition of unstructured programs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 82–87. ACM, 2005.

[2] Chia Yuan Cho, Vijay D'Silva, and Dong Song. Blitz: Compositional bounded model checking for real-world programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 136–146. IEEE, 2013.

[3] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.