Go into a toy security monitor of Serval

Zhilei Han, Xingyu Xie

CST, Tsinghua University

April 2020

Contents

Overview: Serval

The most lightweight operating system verification framework, which won the base paper and best artifact awards in SOSP'19.

- Language: Rosette
- A toy security monitor

Contents

Introduction

Rosette is a solver-aided programming system with two components

- A programming language that extends a subset of Racket
- A symbolic virtual machine

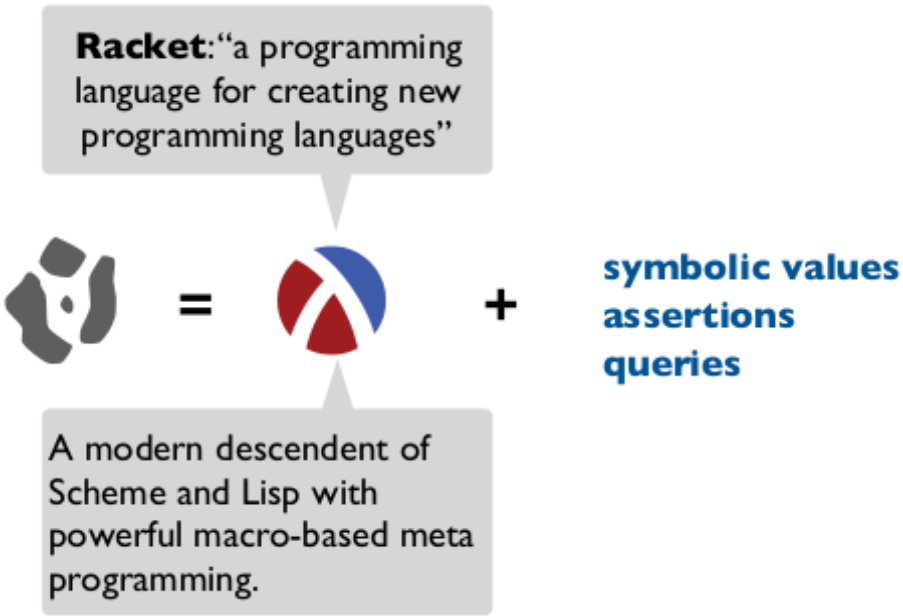


Figure: Rosette is an extension of Racket

Symbolic Value

A symbolic constant can be seen as a placeholder for a concrete constant of the same type.
Furthermore, they can be used just as concrete values in expression, and produce concrete or symbolic values.

```
> (define-symbolic b boolean?)
> (boolean? b)
#t
> (integer? b)
#f
> (not b)
(! b)
> (boolean? (not b))
#t
```

Verification

Rosette use a constraint solver like *Z3* to verify assertions:

```
(define (poly x)
    (+ (* x x x x) (* 6 x x x) (* 11 x x) (* 6 x)))
(define (factored x)
    (* x (+ x 1) (+ x 2) (+ x 2)))
(define (same p f x)
    (assert (= (p x) (f x))))
(define-symbolic i integer?)
(define cex (verify (same poly factored i)))
```

Solvable and unsolvable types

Rosette/safe: an extension of the Racket core.

- solvable type: safe as symbolic value
  - integer, real, boolean, bitvector
  - box, list, ...
- unsolvable type: unsafe, but could be composed as symbolic union
  - string, structure, ...

```
; An example of symbolic union
> (define-symbolic b boolean?)
> (define u (if b "c" 4))
> u
{[b c] [(! b) 4]}
```

Performance

Common Performance Issues:

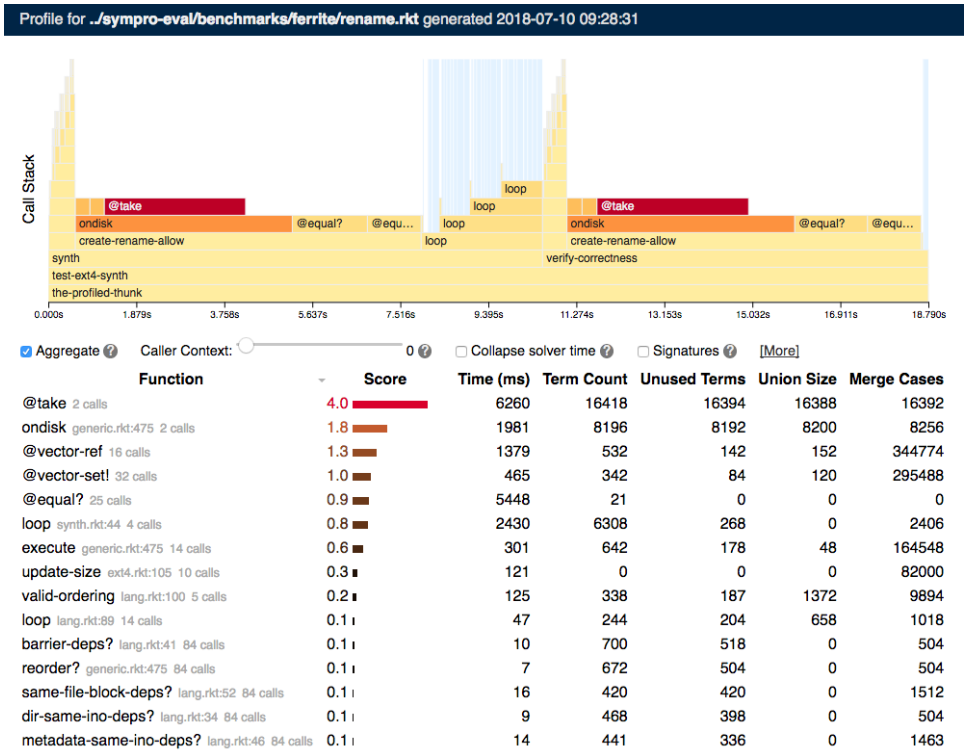
- finite current-bitwidth: perhaps unsound in infinite semantic
- algorithm mismatch: write efficient algorithm for symbol virtual machine
- irregular representation
- missed concretization

Performance

Common Performance Issues:

- finite current-bitwidth: perhaps unsound in infinite semantic
- algorithm mismatch: write efficient algorithm for symbol virtual machine
- irregular representation
- missed concretization

Profiling



Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
@take 2 calls	4.0	6260	16418	16394	16388	16392
ondisk generic.rkt:475 2 calls	1.8	1981	8196	8192	8200	8256
@vector-ref 16 calls	1.3	1379	532	142	152	344774
@vector-set! 32 calls	1.0	465	342	84	120	295488
@equal? 25 calls	0.9	5448	21	0	0	0
loop synth.rkt:44 4 calls	0.8	2430	6308	268	0	2406
execute generic.rkt:475 14 calls	0.6	301	642	178	48	164548
update-size ext4.rkt:105 10 calls	0.3	121	0	0	0	82000
valid-ordering lang.rkt:100 5 calls	0.2	125	338	187	1372	9894
loop lang.rkt:89 14 calls	0.1	47	244	204	658	1018
barrier-deps? lang.rkt:41 84 calls	0.1	10	700	518	0	504
reorder? generic.rkt:475 84 calls	0.1	7	672	504	0	504
same-file-block-deps? lang.rkt:52 84 calls	0.1	16	420	420	0	1512
dir-same-ino-deps? lang.rkt:34 84 calls	0.1	9	468	398	0	504
metadata-same-ino-deps? lang.rkt:46 84 calls	0.1	14	441	336	0	1463

Figure: Profiling of Rosette

James Bornholt and Emina Torlak. Finding Code That Explodes Under Symbolic Evaluation. OOPSLA'18.

Contents

Introduction


```
long sys_dict_get(void) {
    if (current_user < MAXUSER)
        return dictionary[current_user];
    return -1;
}
long sys_dict_set(long value) {
    if (current_user < MAXUSER) {
        dictionary[current_user] = value;
        return 0;
    }
    return -1;
}
long sys_change_user(long newuser) {
    current_user = newuser;
    return 0;
}
```

Implementation

Toy security monitor is not a toy.
A whole library, that an operating system needs, is extracted and used here.
Some initialization need to be done:

- mcall
- csr
- pmp
- tlb

Verifier

Serval ships a riscv verifier, where a CPU state consists of:

- The values of CSRs
- The values of 31 GPRs
- The Program Counter
- The Memory Region

This is a whole description of the underlying machine.

Verifier

To perform symbolic execution on riscv assembly , use an interpreter that defines the semantics of the code over the state, i.e a total function of type $I : State \times Instr \implies State$
For example, the *lui* instruction:

```
; LUI places the 20-bit U-immediate into bits
; 31{12 of register rd and places
; zero in the lowest 12 bits. The 32-bit result
; is sign-extended to 64 bits.
[(lui)
    (check-imm-size 20 imm20)
    (gpr-set! cpu rd
        (sign-extend (concat imm20 (bv 0 12))
        (bitvector (XLEN))))
    (cpu-next! cpu size)]
```

Verification

The verification of toy monitor is based on riscv assembly, which needs to be imported first:

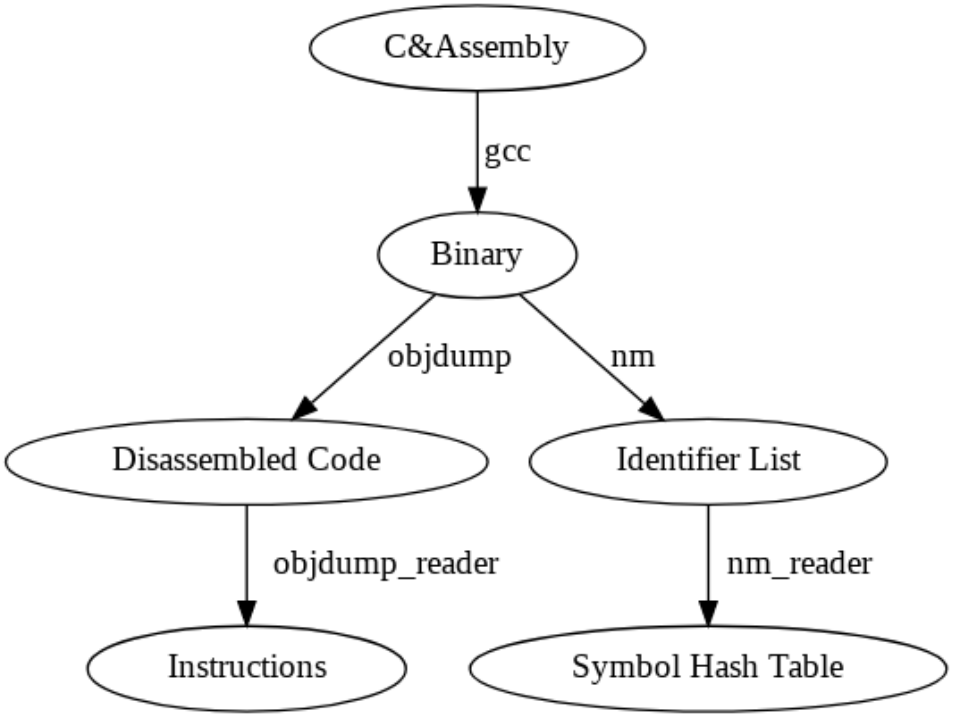


Figure: The verification process

Specification

Now we need to specify the functional behavior of the monitor. The only observable event of this system is the result of syscall.

```
(struct state (retval current-user dict)
  #:transparent
  #:mutable
  #:methods gen:equal+hash
  [(define (equal-proc s t equal?-recur)
   (state-equal? s t))
   (define (hash-proc s hash-recur) 1)
   (define (hash2-proc s hash2-recur) 2)])
```

Since all syscall of this toy system operates on two variables *current-user* and *dict*, it is sufficient to use them to model the system.

Refinement

Now, the crucial step is to prove the simulation relation between the implementation and abstract state, which needs a few more:

- Abstract Function $AF : S_{impl} \implies S_{spec}$
- Invariants: A unary predicate on implementation state

Then we prove by SMT solver

$$\forall \mathrm{c}((\mathrm{RI}(\mathrm{c}) \wedge \mathrm{AF}(\mathrm{c}) = \mathrm{s}) \rightarrow (\mathrm{RI}(\mathrm{f}_{\mathrm{impl}}(\mathrm{c})) \wedge \mathrm{AF}(\mathrm{f}_{\mathrm{impl}}(\mathrm{c})) = \mathrm{f}_{\mathrm{spec}}(\mathrm{s})))$$

Figure: The verification target

Safety Properties

How do we know our specification is correct? We need more constraints on it. In TM, it proves a non-interference properties.