Near Complete Formal Semantics of X86-64

Abstract

ToDo

1. Introduction

2. Challenges

2.1. Using Strata Results

Following are the challenges in using `Strata` [1] (or `Stoke`) formula as is.

- `Stoke` uses C+-functions which define the semantics of instructions. For example, following is the function to define the semantics of add instruction. The functions are generic in the sense that they can be used to obtain obtain the concrete semantics of any instruction like `add %rax, %rbx`

```
S1.   void add(SymBitVector dest, SymBitVector a,
           SymBitVector b) {
S2.       set(d, a+b);
S3.   }
```

  The untested assumption here is the generic formula will behave identically for all the variants. We have tested all the formula for each instruction variant.
- `Strata` gives the concrete semantics for a concrete instructions. For other variants it generalize from the concrete semantics. Assumption is the generalization is correct. Test all the generalization.
- While porting to `K` rule, we generalize the from a concrete semantics that strata provides. Is this generalization faithful? For instruction like `xchg, xadd, cmpxchg`, the formula is different for different operands. So the general `K` rule we obtain from `xchgl a, b` may not represent the semantics for `xchgl a, a`. Fortunately there exists different instruction variants if the their semantics might be different and accordingly we might have different `K` rules. For example, *xchgl_r*32*_eax* and *xchgl_r*32*_r*32. But even for *xchgl_r*32_32 semantics could be different for cases $r1 \,! = r2$ and $r1 \ == \ r2$. Idea: Once lifted as `K` rule, test the instruction for all variants.

  Lets consider `xaddb SRC, DEST`, as per manual the semantics is as follows:

```
S1.   Temp = Src + Dest
S2.   Src = Dest
S3.   Dest = Temp
```

  The point to note here is that the register updates follow an order. `Strata` uses `xaddb %rax, %rbx`, to obtain the semantics and it happened that the ordering is maintained and hence strata can generalize the semantics of `xaddb R1, R1`. But even if the ordering is not maintained the semantics is going to be the same for the case $R1! = R2$, but the generalization for the `R1 == R1` case will mess up. We cannot trust the above generalization by strata. We need to test the `K` rule for all possible operands.

3. Implementation

3.1. Porting Formulas for stratified instructions to `K` Rules

For the purpose of getting `K` rules, we could have directly converted the `Strata` formulas for an instruction to `K` rule assuming that the `Strata`'s symbolic execution over the stratified instruction sequence is correct.

Given that fact the `K`'s symbolic execution engine is more trusted as that has been used extensively in language-agnostic manner to perform symbolic execution, we decided to use `K`'s symbolic executor. Also in order to check if `Strata`'s symbolic execution engine is correct, we did an equivalence check on the outputs of both the symbolic executions.

1. Implementing the base instructions semantics in `K` and testing them.
2. Symbolic execution of the stratified instruction sequences.
3. Dealing with scratch pad registers.
4. Equivalence check between `Strata` formula ad the output of 2.

   All the checks are *unsat*, expect one where the check fail to due a bug in the simplification rules in `Strata`, which states the following lemma related to two single precision floating point numbers `A` and `B`, which is not correct for `NaNs`. However this bug is fixed in the latest version of `Stoke`.

   `add_single(A, B)` $\equiv$ `A if B == 0`
5. Simplification of formulas: Simplification generates simple `K` rule (sometimes simpler than the corresponding `Strata` formula). Also it is much easier to write the simplification rules in `K`.show the example for concat(A[1:2], concate(B[2:3], X)) ≡ concate(A[1:3], X)
6. One of the issue with `Strata` formulas is they could be too complex to comprehend at times, which is mainly because 1. `Strata` tried to define the semantics of an instruction using other simpler instructions, 2. The simplification rules in `Strata` or the ones we define in `K` are not sophisticated enough to simplify the complex formulas. An example of one such simplification opportunity is: $(0_{32} \cdot \texttt{\%rax[32:0]}) \oplus \texttt{\%rax} \equiv \texttt{\%rax[63:32]} \cdot 0_{32}$

   In order to simplify those, we borrowed the hand written formula (provided they ate simpler) from `Stoke` or manually write the simpler formulas and check equivalence with the stratified formula. If they match on all register state, we employ that in our `K` semantics.

3.2. Supporting un-stratified instructions & Porting their formulas to `K` Rules

3.2.1. Supporting un-stratified instructions

Instruction support status

3.3. Porting to `K` Rules

`Strata` could output the internal AST, used to model a register state formula, in different formats. Supported backend are SmtLib and Prefix notation. We have added another backend to generate `K` rule. We need some way to validate the backend.

Validate the Backend The `K` rules generated using the backend are matched (syntactically) against the ones we already obtained via symbolic execution on stratified instructions. Other than validaing the backend, this has an added benefit that in order to get the exact match, we need to port all the simplification rules from `K` to strata code, which in turn will later help in generating simplified `K` rules for non-stratified instructions.

Main challenges in getting an exact match are:

- `Strata` rules uses *extract* to extract portion of a bit-vector. The high and low indices of *extract* are obtained considering LSB at index 0, whereas `K` uses *extractMInt* for the same purpose, but uses MSB at index zero.
- `Strata` uses flags as `Bool`, whereas they are treated as `Bitvector` in our semantics. We modifed strata so as to treat flag registers as 1 bit bitvectors.

References

[1] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 237–250, New York, NY, USA, 2016. ACM.

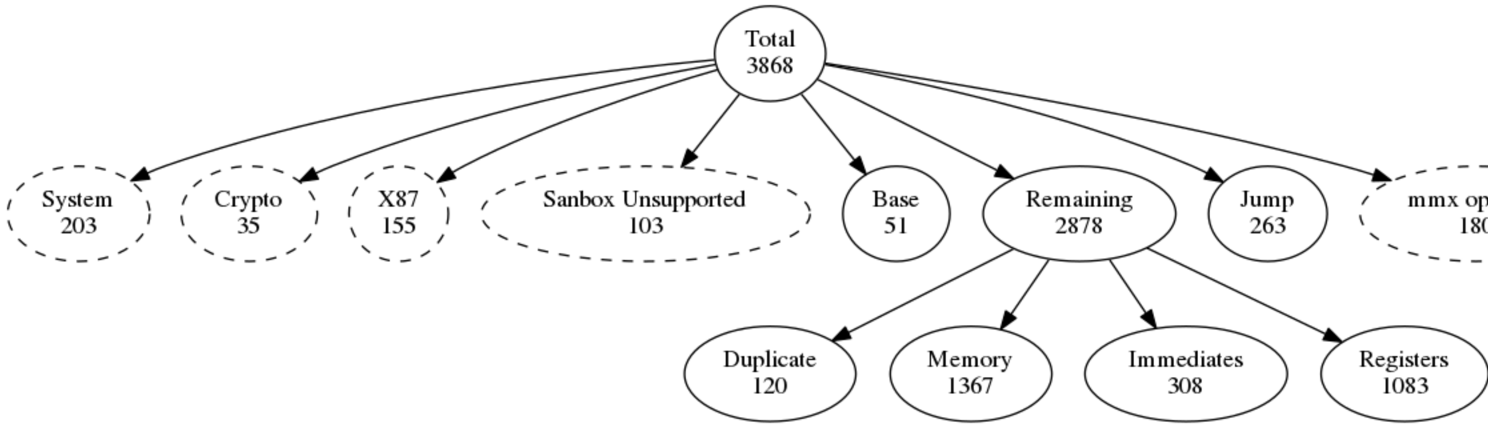

Figure 1: Instruction classification