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_r32_eax* and *xchgl_r32_r32*. But even for *xchgl_r32_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. Modeling X86-64 Instruction Semantics

In this work we supported formal semantics of the input/output behavior of 2929 out of 3868 of the x86-64 Haswell ISA instruction variants. The un-supported ones (# 939) falls in the following categories: **Systems** (304 variant), **Cryptography** (304 variant), **X87**, (155 variant), **MMX** (180 variant) & **Jumps** (263 variants).

In order to get semantics of individual instructions, we build on top of project `Strata` [1] which automatically synthesized formal semantics of the input/output behavior for 1796 Haswell ISA X86-64 instructions. The key to their results is stratified synthesis, where they use a set of instructions whose semantics are known to synthesize the semantics of additional instructions whose semantics are unknown. Using the technique, they come up with the semantics of 692 register and $\sim$ 120 immediate instructions. The rest $\sim$ 984 instructions are the immediate and memory variants obtained by generalizing the 692 register instructions.

Following are some of the immediate challenges that we need to overcome.

- **CH.1** For immediate variants that do not have a corresponding register-only instruction, `Strata` learns a separate formula for every possible value of the constant provided the constant value is of width 8 bits. Also in some cases, they learned a formula only for some of the possible constants. In order to have a more intuitive semantics of those instructions, we a generic formula for those immediate instructions.
- **CH.2** `Strata` does not model the *%af* flag. This is not a fundamental challenge, but a choice as this isn't a commonly used flag. Supporting this flag fall within the scope of our work.
- **CH.3** There are instructions which conditionally sets some cpu flags to *undef*. For example, the shift left instruction `salq %cl, %rbx` sets flag *%of* to *undef* state if the count mask > 1. Again there are instructions like `blsr %eax, %ebx` which un-conditionally puts flags like *%pf* & *%af* into *undef* state.
  `Strata` while doing the *initial search* does not test the flags which *may* (for conditional *undef*s) or *must* (for unconditional *undef*s) be taking undefined values. We intend to model the semantics of these flags with the same correctness guarantee as the other registers which are defined and

hence model by Strata.

- **CH.4** How reliable is the generalization of register instructions to memory or immediate variants? Strata states that the claim for the generalization is based on random testing.
- **CH.5** Finally how to support the unsupported or *unstratified* ones. The paper [1] mentions that adding some primitive instructions (like saturated add) as the base instruction might help stratified more instructions. We would like to explore similar directions.

Following is a key observation concerning stratification which help us handle the most of the above mentioned challenges.

Observation In order to get the semantics of a target instruction I, Strata uses Stoke along with set TS of 6580 test cases to synthesize an instruction sequence which agrees with I on TS (which means the output behavior of the instruction sequence matches with the real hardware execution on input TS). After having that *initial search*, they keep on searching additional sequences (which they call secondary searches each agreeing with I on TS) in a hope of getting one which would prove non-equivalent to existing ones and thereby gaining more confidence on the search and probably a better test-suite (as TS might get augmented with a counter example from equivalence checker in the event of non-equivalence). One unfavorable possibility for Strata is when all subsequent secondary search results proves equivalent to the one obtained from initial search, in which case it means that secondary searches fail to add any "confidence" to the initial search result and the final outcome of stratification is having the same correctness guarantee as that provided by the initial search result, which is "correctness over TS". But in those unfavorable case, the secondary searches might have provided "better" choices to pick the final formula from. A better choice of formula do not contain uninterpreted functions or non-linear arithmetics and are simple.

In the paper[1], it is mentioned that there are only 50 cases, where they found a (valid) counterexample. That means, there are $762 = (692 + 120 - 50)$ cases, where the initial search is sufficient enough to be accepted, as all the later secondary searches results are equivalent to the one obtained from the initial search. In other words, in most of the cases, the correctness guarantee of stratification is same as that of the initial search result.

For the unstratified instructions, we would need a *semantics generator* to provide us with an initial candidate of the instruction semantics. Once we have that semantics, we could test it against hardware on the same augmented test-suite (containing $6630 = 6580 + 50$) that Stoke uses and if the candidate matches then we can claim to have the same correctness guarantee as above.

Now the missing piece, the *semantic generator*, can be projects like Stoke, which have manually written instruction semantics (in terms of logical formulas), or can be manually written. We understand that this is not as efficient as Stoke, which is fully automatic in getting these formulas, but our contribution is 1. To deliver in cases where Stoke cannot 2. To cover as many instruction semantics as possible. Moreover, writing the semantics manually might alleviate the need of secondary search as a means to provide "better" formula as we can control the complexity and choice of operations to include in the formula. Also carefully written manual formula tend to need less number of conflicting searches than the onces generated by random search engines like Stoke.

Handling CH.1

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.
   ```
   sub_single(A, B) ≡ 0 if A == B
   ```
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 drawback of the Strata formulas is they could be non-intuitive and complex at times when the simplification rules are not adequate enough to simplify their complexity to more intuitive formulas. Appendix A provides such an example. Towards the goal of having intuitive formulas, we borrowed the hand written formula (provided they are simpler) from Stoke or manually write those and check equivalence with the stratified formula. If they match on all register state and/or memory, 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.

A. An Example of Strata Formula

Following is the Strata formula for an instruction `vpxor %ymm3, %ymm2, %ymm1`,

```

(let ((a!1 (bvxor ((_ extract 255 192) ymm3)
                  ((_ extract 255 192) ymm2)
                  (bvor ((_ extract 255 192) ymm3)
                        (bvxor ((_ extract 255 192) ymm3)
                               ((_ extract 255 192) ymm2)))))
      (a!3 (bvxor ((_ extract 191 128) ymm3)
                  ((_ extract 191 128) ymm2)
                  (bvor ((_ extract 191 128) ymm3)
                        (bvxor ((_ extract 191 128) ymm3)
                               ((_ extract 191 128) ymm2)))))
      (a!5 (bvxor ((_ extract 127 64) ymm3)
                  ((_ extract 127 64) ymm2)
                  (bvor ((_ extract 127 64) ymm3)
                        (bvxor ((_ extract 127 64) ymm3)
                               ((_ extract 127 64) ymm2)))))
      (a!7 (bvxor ((_ extract 63 0) ymm3)
                  ((_ extract 63 0) ymm2)
                  (bvor ((_ extract 63 0) ymm3)
                        (bvxor ((_ extract 63 0) ymm3) ((_ extract 63 0) ymm2))))))
(let ((a!2 (bvxor ((_ extract 255 192) ymm3)
                  ((_ extract 255 192) ymm2)
                  (bvor ((_ extract 255 192) ymm3)
                        (bvxor ((_ extract 255 192) ymm3)
                               ((_ extract 255 192) ymm2)))))
      (a!4 (bvxor ((_ extract 191 128) ymm3)
                  ((_ extract 191 128) ymm2)
                  (bvor ((_ extract 191 128) ymm3)
                        (bvxor ((_ extract 191 128) ymm3)
                               ((_ extract 191 128) ymm2)))))
      (a!6 (bvxor ((_ extract 127 64) ymm3)
                  ((_ extract 127 64) ymm2)
                  (bvor ((_ extract 127 64) ymm3)
                        (bvxor ((_ extract 127 64) ymm3)
                               ((_ extract 127 64) ymm2)))))
      (a!8 (bvxor ((_ extract 63 0) ymm3)
                  ((_ extract 63 0) ymm2)
                  (bvor ((_ extract 63 0) ymm3)
                        (bvxor ((_ extract 63 0) ymm3) ((_ extract 63 0) ymm2)))))
      (a!9 (bvxor ((_ extract 255 192) ymm3)
                  (bvor ((_ extract 255 192) ymm3)
                        (bvxor ((_ extract 255 192) ymm3)
                               ((_ extract 255 192) ymm2))))
                  (bvor a!1
                        ((_ extract 255 192) ymm2)
                        ((_ extract 255 192) ymm3)))
      (a!4 (bvxor ((_ extract 191 128) ymm3)
                  ((_ extract 191 128) ymm2)
                  (bvor ((_ extract 191 128) ymm3)
                        (bvxor ((_ extract 191 128) ymm3)
                               ((_ extract 191 128) ymm2)))
                  (bvor a!3
                        ((_ extract 191 128) ymm2)
                        ((_ extract 191 128) ymm3)))
      (a!6 (bvxor ((_ extract 127 64) ymm3)
                  ((_ extract 127 64) ymm2)
                  (bvor ((_ extract 127 64) ymm3)
                        (bvxor ((_ extract 127 64) ymm3)
                               ((_ extract 127 64) ymm2)))
                  (bvor ((_ extract 127 64) ymm2) ((_ extract 127 64) ymm3) a!5)))
      (a!8 (bvxor ((_ extract 63 0) ymm3)
                  ((_ extract 63 0) ymm2)
                  (bvor ((_ extract 63 0) ymm3)
                        (bvxor ((_ extract 63 0) ymm3) ((_ extract 63 0) ymm2)))
                  (bvor ((_ extract 63 0) ymm2) ((_ extract 63 0) ymm3) a!7))))
  (concat a!2 a!4 a!6 a!8)))
```

where as following is the formula obtained from Stoke (hand-written) and Z3 took 88.70 secs to prove that they are equivalent.

```
%ymm1    : (bvxor %ymm2 %ymm3)
```

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.

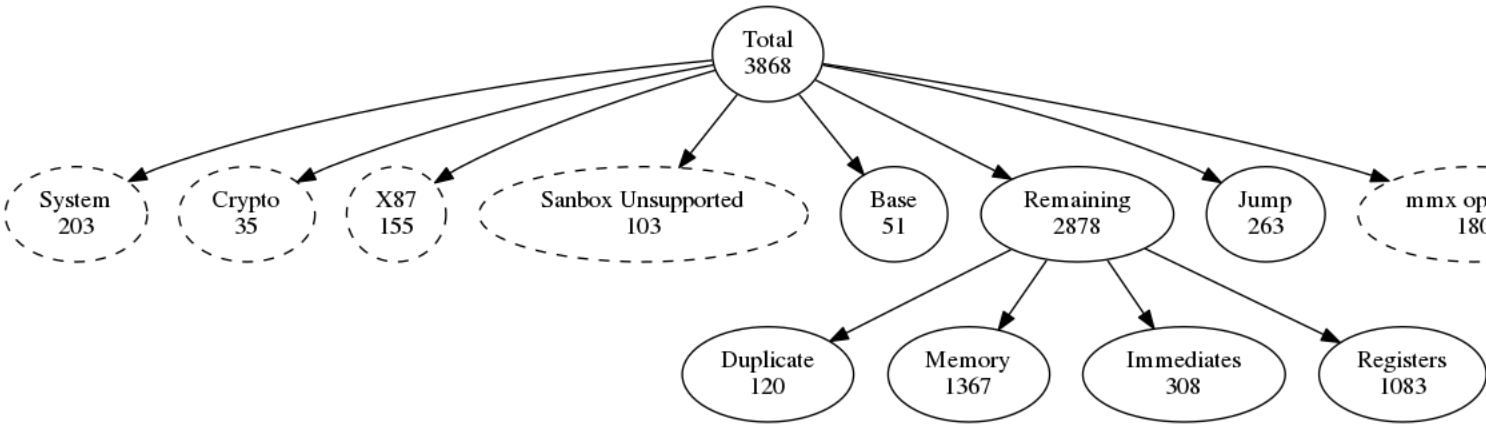

Figure 1: Instruction classification