CS357 - Assignment 3

Due: November 16, 2015, 23:59

In this exercise you will use STOKE to optimize x86_64 programs and verify formally that they are correct (that is, equivalent to some existing implementation).

Instructions

1. **Part I: Using STOKE to optimize straight-line code**

   STOKE uses Markov Chain Monte Carlo sampling to optimize x86_64 programs. It can operate in either *optimization* or *synthesis* mode. In optimization mode, the random search starts with a given program, and STOKE tries to find a better one by making random changes, guided by a cost function. In this first part of the assignment, we will only use synthesis mode, where STOKE starts with the *empty* program and tries to learn a fast program, again using the same random search strategy.

   We will use STOKE to optimize the following function (written in C):

   ```
   uint32_t f(uint32_t x) {
     int res = 0;
     for (; x > 0; x >>= 1) {
       res += x & 0x1ull;
     }
     return res;
   }
   ```

   (a) Explain what the function `f` computes.

   (b) Now, let's use STOKE to find a faster implementation. Download the binary release of STOKE available at http://web.stanford.edu/class/cs357/handouts/stoke-distrib.zip. STOKE requires relatively new hardware (at least a Sandybridge or Haswell CPU). You are welcome to try it on your own computer, but it may not work depending on your CPU. STOKE will definitely work on corn34.stanford.edu (where all other tools such as `g++` are also available). Note that not all nodes of the corn cluster have new enough CPU's, but we know that corn34 (and probably others) works.

   Verify that you can run STOKE by invoking `stoke search --help` (which prints the command line arguments for the STOKE search).

   (c) Before STOKE can optimize the program, we need to compile it and extract the x86 code for `f`. We have prepared all needed files (including the code above) here: http://web.stanford.edu/class/cs357/handouts/stoke-inputs.zip.

   Compile the code in `main.cc`:

   ```
   g++ -std=c++11 -fno-inline -O3 main.cc
   ```

(d) Next, we need to extract the x86 code for `f` from the binary, which STOKE can do for us:

```
stoke extract -i ./a.out -o bins
```

This creates a folder `bins` with all functions in the binary. Inspect this folder and find the code for `f`.

(e) During the search, STOKE uses testcases to evaluate the programs, so we need to generate testcases for our function `f`. For x86 code, a testcase essentially specifies the value of all CPU registers (or at least the ones that are live at function entry), as well as all memory contents (though in our case, `f` does not read or write any memory).

There are many ways we could get testcases, but since `main.cc` includes a main function that runs `f` on test inputs, we can use that. STOKE uses PIN (a binary instrumentation framework) to record the CPU and memory state just before `f` is executed. Obtain 1024 testcases by running:

```
stoke testcase --bin ./a.out --args 10000000 --functions bins \
    -o testcases.tc --fxn _Z1fj --max_testcases 1024
```

You can look at the testcases by opening the file `testcases.tc`.

(f) Now we are ready to perform a search with STOKE. We provide a configuration file as `search.conf`. In particular, we choose the following options:

i. `--init zero`. We start the search with the empty program; i.e., we are using STOKE in synthesis mode.

ii. `--def_in "{ %edi }"` and `--live_out "{ %eax }"`. The register `edi` holds the input, and the result is stored in `eax`.

iii. `--testcases testcases.tc`. We use the tescases we generated earlier.

iv. `--training_set "{ 0 ... 7 }"` and `--test_set "{ 8 ... 1023 }"`. During the search, we will only use the first 8 testcases and use the remaining testcases at the end to verify the program we find (not formally verify, but verify that it works on all testcases).

v. `--cost "correctness + latency"`. During the search, we measure how close the program we have found so far matches the output of the target (i.e., the function `f`) on the output register `eax` (the term `correctness`), as well as what the latency of all instructions used by the rewrite is (the term `latency`). This measures both how correct our program is, as well as how fast it is.

vi. `--correctness "correctness == 0"`. We want to minimize the latency (to get a fast program), but we also want the program we find to be fully correct. Thus, at the end of the search we require that `correctness == 0`.

With these options set up, we are ready to run the STOKE search:

```
stoke search --config search.conf
```

The search outputs various statistics during its run, and then finally (if it is successful), saves the program found in `result.s`. Inspect the program, and explain why the program is correct. If you are not happy with the program STOKE found, you can re-run the search (hint: there is a very short program that is much faster than the original implementation). Make sure you include the program found in your submission.

If you are not familiar with all x86 instructions, the Intel x86 manual (specifically, volume 2) contains a description of all instructions.

(g) Let's see how fast the new program is, compared to the original one. First, let's time the original implementation:

```
time ./a.out 100000000
```

Now, let's replace the implementation for `f` with our synthesized program. STOKE can do this for us:

```
stoke replace -i ./a.out --rewrite result.s
```

Now run the timing command again, and report the speedup.

(h) So far we have only tested our new program, but not formally verified it. The original program contains a loop, but it turns out the loop can execute at most N times. What is N?

With this knowledge, we can use STOKE's bounded validator to formally prove correctness. The bounded validator takes a command-line argument `--bound` to indicate how many times a loop should be unrolled (very similar to what you did in assignments 1 and 2). Formally prove correctness (don't forget to plug in the correct bound):

```
stoke debug verify --strategy bounded --bound N --target bins/_Z1fj.s \
  --rewrite result.s --def_in "{ %edi }" --live_out "{ %eax }"
```

If you're program does not verify, then STOKE may have found a program that works on all testcases, but not on all possible inputs. Try running the STOKE search again.

2. **Part II: Formal verification of programs with loops**

   In the first part one of our programs had a loop, but we got away with just unrolling the loop because the loop executes at most a constant number of times. Of course, in general this will not be true, which is why techniques like the Data-Driven Equivalence Checking (DDEC) you have seen in class are necessary.

   In this part, we will prove two implementations of a wide-character string copy routine (wcpcpy, see its manpage) equivalent using DDEC. Here is the first program:

```
.wcpcpy:
  movq %rdi, %rax
  movl (%rsi), %edx
  testl %edx, %edx
  je .L_4005f0
.L_4005df:
  addq $0x4, %rax
  addq $0x4, %rsi
  movl %edx, -0x4(%rax)
  movl (%rsi), %edx
  testl %edx, %edx
  jne .L_4005df
.L_4005f0:
  movl $0x0, (%rax)
  retq
```

   And here is the second one:

```
.wcpcpy:
  movq %rdi, %rax
  movl (%rsi), %edx
  testl %edx, %edx
  je .L_4005f0
  xorq %r8, %r8
.L_4005df:
  addq $0x4, %rax
  addq $0x2, %r8
  movl %edx, -0x4(%rax)
  movl (%rsi, %r8, 2), %edx
  testl %edx, %edx
  jne .L_4005df
.L_4005f0:
  movl $0x0, (%rax)
  retq
```

   (a) First, draw the control flow graph for both programs.

   (b) Identify all possible cutpoints that could be used to prove the two programs equivalent (you can give them as line numbers, where line number n indicates a cutpoint between line $n-1$ and line n). Remember that cutpoints need to be executed the same number of times. Furthermore, we need to find an invariant that allows us to prove the overall equivalence of the two programs. Here, we will learn complex invariants about registers, and simply require that the memory contents at cutpoints are equivalent. Note that you do not have to give any invariants at this point, just the cutpoints as pairs of line numbers.

(c) By convention, we will only use cutpoints that involve the end of a basic block. We can now run the program on some test inputs, and collect the values for all registers at the cutpoints. If we do this for the two programs on 8 testcases, we might get data like shown in the table on the last page (and is also shown in this file: http://web.stanford.edu/class/cs357/handouts/matrix.txt).

Use MATLAB to determine the null space of this data when interpreted as a matrix. Note that we are interested only in integer solutions, so we want to compute the null space over $\mathbb{Z}$ rather than $\mathbb{R}$ (which is what MATLAB does normally). MATLAB cannot do the computation over integers directly, but it provides a way to give a rational basis. Multiplying that with the least common multiple of the denominators gives the integer solution we are looking for.

Note that Octave does not support this rational basis for the null space, so make sure you use MATLAB. If you don't have MATLAB installed, you can use it on corn:

```
ssh -Y yoursunetid@corn.stanford.edu
module load matlab
matlab
```

(d) Now that you have the null space, interpret the result as invariants that appear to hold according to the data. The invariants will relate the values of registers with each other, possibly across the two programs. Register names with a prime correspond to the second program, those without primes correspond to the first program.

(e) The invariants we found by looking at the data are not all true invariants. That is, some of them hold on all 8 supplied tests, but do not hold in general. For each invariant, state whether it is a true invariant or not. If it is not, give a description of an input where the invariant would not hold. You don't have to give the input necessarily, but just describe what the input needs to look like. For example, you might say "Any input where the first character is the letter "a" violates this invariant".

(f) After throwing out all bad invariants, we are left with a set of invariants that are sufficient to prove that the two programs are, in fact, equivalent on all possible inputs.

Hints

Note that some of the subproblems in this problem set require a correct solution for the previous step. If you get stuck or just want to confirm that your solutions so far are correct, feel free to email sheule@cs.stanford.edu. I can tell you if your partial solution is correct, and give hints as to what is wrong if necessary.

Submission Process

Electronically submit the solution to this assignment by emailing it to cs357-aut1516-staff@lists.stanford.edu. Make sure you include the necessary steps you took, including the relevant output of STOKE.

rax	eax	rdx	edx	rsi	esi	rdi	edi	rax'	eax'	rdx'	edx'	rsi'	esi'	rdi'	edi'	r8'	r8d'	1
175051885	175051885	807604391	807604391	175051864	175051864	175051881	175051881	175051885	175051885	807604391	807604391	175051860	175051860	175051881	175051881	2	2	1
175051889	175051889	3637451659	3637451659	175051868	175051868	175051881	175051881	175051889	175051889	3637451659	3637451659	175051860	175051860	175051881	175051881	4	4	1
1165504690	1165504690	2467993491	2467993491	1165504629	1165504629	1165504686	1165504686	1165504690	1165504690	2467993491	2467993491	1165504625	1165504625	1165504686	1165504686	2	2	1
1165504694	1165504694	3474620852	3474620852	1165504633	1165504633	1165504686	1165504686	1165504694	1165504694	3474620852	3474620852	1165504625	1165504625	1165504686	1165504686	4	4	1
172755665	172755665	284463817	284463817	172755619	172755619	172755661	172755661	172755665	172755665	284463817	284463817	172755615	172755615	172755661	172755661	2	2	1
172755669	172755669	1705539540	1705539540	172755623	172755623	172755661	172755661	172755669	172755669	1705539540	1705539540	172755615	172755615	172755661	172755661	4	4	1
213189768	213189768	789181076	789181076	213189721	213189721	213189764	213189764	213189768	213189768	789181076	789181076	213189717	213189717	213189764	213189764	2	2	1
213189772	213189772	2347504353	2347504353	213189725	213189725	213189764	213189764	213189772	213189772	2347504353	2347504353	213189717	213189717	213189764	213189764	4	4	1