# Assignment 6
# Having a (Bit) Blast

15-414: Bug Catching: Automated Program Verification

Due Thursday, March 31, 2022
75 pts

This assignment is due on the above date and it must be submitted electronically on Gradescope. Please carefully read the policies on collaboration and credit on the course web pages at `http://www.cs.cmu.edu/~15414/s22/assignments.html`.

## What To Hand In

You should hand in the following files on Gradescope:

- Submit the file `asst6.zip` to Assignment 5 (Code). You can generate this file by running `make handin`. This will include your solutions `taocp.py`, `prng.py`, and `bmc.py`.
- Submit a PDF containing your answers to the written questions to Assignment 6 (Written). You may use the file `asst6.tex` as a template and submit `asst6.pdf`.

## Using LaTeX

We prefer the answer to your written questions to be typeset in LaTeX, but as long as you hand in a readable PDF with your solutions it is not a requirement. We package the assignment source `asst6.tex` with handout to get you started on this.

## Useful Resources

The notes and live-coding examples for lectures 16 and 17 provide the primary background needed to complete these problems. If you are unsure about how to implement your approach using the Z3 API, please review Section 3 of lecture 17 before posting a question.

You may also find sections 2, 4, and 5 of the tutorial written by Nikolaj Bjorner, the lead developer of Z3, helpful for additional background and examples:

https://theory.stanford.edu/ nikolaj/programmingz3.html

## 1 Identical Bits (25 pts)

The following problem was posed by Donald Knuth in *The Art of Computer Programming, Volume 4 Fascicle 1: Bitwise Tricks and Techniques*. Recall from Lecture 16 that $\oplus$ denotes bitwise exclusive-or.

> J.H. Quick noted that $((x + 2) \oplus 3) - 2 = ((x - 2) \oplus 3) + 2$ for all $x$. Find all constants $a$ and $b$ such that $((x + a) \oplus b) - a = ((x - a) \oplus b) + a$ is an identity.
>
> (Problem 3.15, Page 53 [*M26*])

*Task* 1 (15 pts). Assuming that $x, a$ and $b$ are bit vectors of width 1 ($\ell = 1$), solve this problem by bit-blasting the equation to propostitional logic, and solving for $a$ and $b$. You may solve the propositional formula however you like, including with a truth table or using a solver such as Z3, but be sure to show the bit-blasted propositional formula, and explain how you derived it.

*Hint: You may find the following propositional identity useful on the bit-blasted formula:*

$$\neg(x \oplus y) \leftrightarrow (\neg x) \oplus y \leftrightarrow x \oplus (\neg y)$$

*Task* 2 (10 pts). Now assume that $x, a$ and $b$ are bit vectors of arbitrary width. Complete the function `solve_identity` in `taocp.py`.

```python
def solve_identity(width: int) -> int:

  a, b, x = BitVecs('a b x', width)

  '''
    Replace the formula `identity` below with your
    encoding of the identity from Problem 1
  '''
  identity = ...

  s = Solver()
  s.add(identity)

  solutions = []

  '''
    Now implement code to enumerate all solutions to your encoding,
    adding each one to `solutions`
  '''

  return len(solutions)
```

Run your solution with $\ell = 1, 2, 3, 4, \ldots$ (note that larger values of $\ell$ may take long to compute), and conjecture a general solution to Knuth's original problem: what are the constants $a$ and $b$ that make the equation and identity?

*Hint: if you don't feel that the number of solutions for a range of $\ell$ provides enough information to form a conjecture, try printing the solutions out for a few values and looking for a pattern.*

## 2 Cracking LCG (20 pts)

Pseudorandom numbers are important in many computing applications, especially so in those related to security and cryptography: encryption keys, random nonces in networking protocols, and other such quantities must be extremely difficult for an attacker to guess a-priori. For this reason, secure implementations make use of pseudorandom number generators that are expressly designed with this goal in mind.

For more casual applications that do not rely on randomness for security, a popular method for generating pseudorandom numbers is the *linear congruential generator* (LCG), with a representative implementation in C shown below.

```c
int rseed = 0;

void srand(int x) { rseed = x; }

inline int rand()
{
  rseed = rseed * 214013 + 2531011;
  return (rseed >> 16) & 0x7FFFh;
}
```

In this problem, we will see why it is not advisable to use simple (but cheap!) generators like LCG in secure applications.

*Task* 3 (15 points). Complete the function `recover_previous` in `prng.py`, a sketch of which is shown in Figure 1. The purpose of this function is to take a sequence of observed outputs from `rand`, which have been modulo-reduced by a given base, check whether the previous (unobserved) output of `rand` is uniquely determined, and if so, return it.

For example, suppose that eight calls to `rand() % 100` (i.e. mod-reduced base 100) produced:

$$0, 78, 23, 98, 13, 35, 45, 25, 61$$

Then `recover_previous([78, 23, 98, 13, 35, 45, 25, 61], 100)` should return `0`, if `0` is the only possible previous value for the given sequence.

- You should assume that the modular reduction is unsigned; this operation is exposed by the Z3 API's `URem(n, base)` function.
- You should assume that shift operations are signed; these are overloaded by the Z3 API, and can be accessed using the normal Python shift notation: << and >>.
- Addition, multiplication, and other bitwise operators are likewise overloaded.

Check your implementation by testing it on the sequence given above: it should output `0` given the last seven outputs. Then, determine how many observations are required to uniquely determine the solution to this sequence.

*Task* 4 (5 points). Use your solution to Task 3 to determine the largest number of consecutive zeros that `rand() % 10` can output. It may help to modify `recover_previous` to write to standard output to complete this task.

```
def recover_previous(observed: List[int], modulo: int) -> Optional[int]:

  prev_output = BitVec('previous', 32)

  s = Solver()

  #TODO: Add appropriate constraints to `s`

  if s.check() == sat:
    prev_sol = s.model().evaluate(prev_output, model_completion=True).
        as_signed_long()

    '''
      TODO: Determine whether `prev_sol` is the unique solution,
      and return the appropriate value (None in the case of not-unique)
    '''
  else:
    prev_sol = None

  return prev_sol
```

Figure 1: Template for Task 3

## 3 State Invariants (30 points)

In lecture, we studied an approach to bounded model checking aimed at verifying Hoare triples $\{P\}\,\alpha\,\{Q\}$, i.e., that whenever $P$ is true in the initial state, $Q$ must be true in any state that $\alpha$ terminates in.

In many practical settings, bounded model checking is used to check *state invariants*, which are properties of the program state that are intended to remain true at all times while the program executes. In other words, given a formula $R$, the model checker attempts to verify that $R$ is true in each intermediate state that $\alpha$ enters.

The strongest postcondition generator that we wrote in lecture can be modified to implement a bounded model checking procedure that handles state invariants. The approach is conceptually similar to unwinding assertions, in that we can view it largely in terms of a transformation on the program being checked. Namely, for any statement that directly changes the state, we insert a conditional check after it of $R$; if $R$ no longer holds, then we assign 1 to a special variable, `err`.

$$x := e \qquad \text{becomes} \qquad x := e; ((?\neg R; \mathtt{err} := 1) \cup (?R; \mathtt{err} := \mathtt{err}))$$

*Task* 5 (10 points). Modify the `post` function in `bmc.py` to simulate having rewritten the program as described above. Because the only statements that can directly change state are assignments, your modifications will be limited to the case for `Asgn`.

```
  if isinstance(alpha, Asgn):
    left, right = alpha.left, alpha.right
    ...
    if R is not None:
      # Add code here to "simulate" having rewritten the program as
      # being followed by a check on `R`, as described in the handout
    ...
```

*Task* 6 (20 points). Implement the `check_invariant` function in `bmc.py`, which takes a program `alpha`, a precondition `P`, a state invariant `R`, and a maximum unwinding depth, and returns `True` exactly when the state invariant `R` is true in all states of `alpha` starting in `P`, up to the given depth.

```
def check_invariant(alpha: Prog, P: BoolRef, R: BoolRef, max_depth=10) -> bool:

  '''
    TODO: implement this procedure so that it returns True
          if and only if all states that alpha enters
          up to the given execution depth satisfy R
  '''

  return False
```

- Your implementation will make use of your solution for Task 5 to ensure that whenever an intermediate state fails to satisfy `R`, then `err` is set to 1.
- Make sure that your solution also reflects the fact that `err` is initialized to `0` whenever the initial state satisfies `R`.
- You can use Z3 to check the satisfiability of the strongest postcondition generated by `post`, conjoined with a condition on `err` that you must derive; the result of this check should allow you to determine whether `alpha` satisfies the state invariant.
- Be sure to test your implementation on simple examples. For example, $(x := x + 1)*$ with the state invariant $x < 5$, at different settings of `max_depth`.