Exercises on code analysis techniques

Abstract Interpretation (value set analysis)

In the following we consider abstract interpretation on programs using the interval abstract domain.

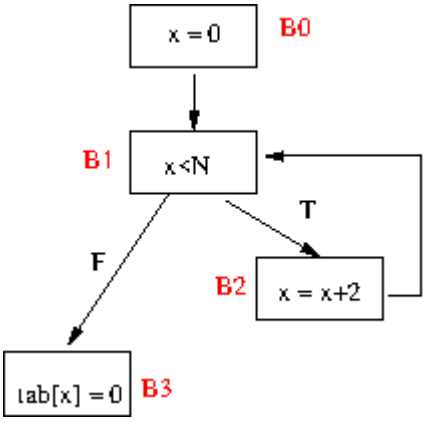
Exercise 1

We consider the following C code and its control-flow graph :

```
#define N 3

int x ;
int Tab[N] ;

x = 0 ;
while (x<N)
     x = x+2 ;
tab[x] = 0
```



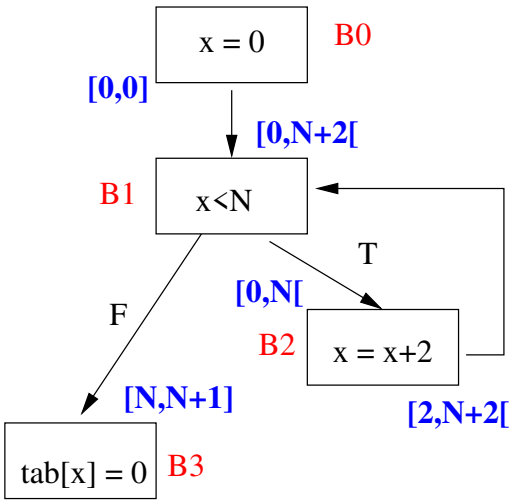
Q1. Compute the value sets at each entry/exit points of each basic blocks without using any acceleration technique (i.e., widening/narrowing).

Q2. Same as Q1, but using widening/narrowing operators.

Q3. Same as Q2 by replacing the constant 3 by the constants1000 and 1001.

Q4 . What can we conclude about potential program vunerabilities ?

Solution: the detail of the fix-point computations using intervals is available on Moodle, and the result is depicted on the Figure below. Hence, a buffer overflow will occur ...



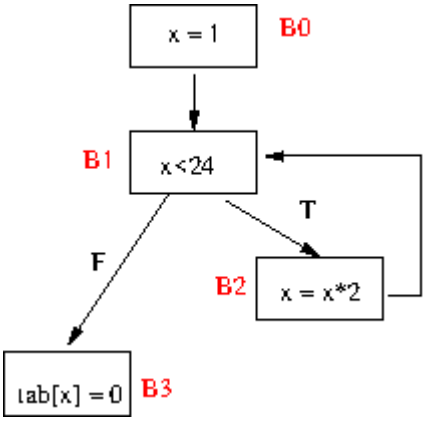
Exercise 2

We consider the following C code and its control-flow graph :

```
#define N 33

int x ;
int Tab[N] ;

x =  1;
while (x<N)
     x = x*2 ;
tab[x] = 0
```



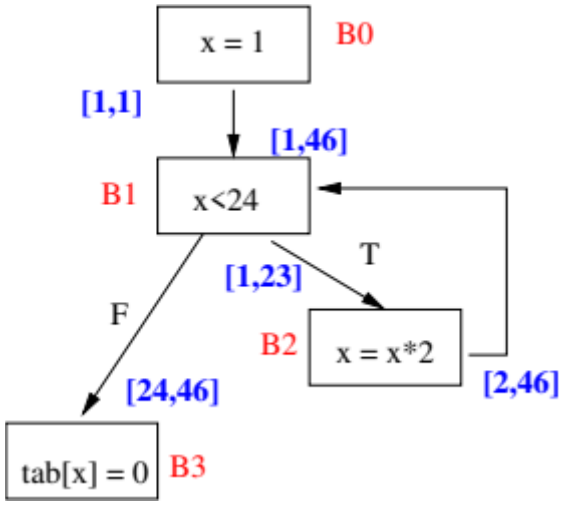
Q1. Compute the value sets at each entry/exit points of each basic blocks using acceleration techniques (i.e., widening/narrowing).

Q2 . What can we conclude about potential program vulnerabilities ?

Q3. How could we get more precise results with Frama-C ?

Solution:

Using fix-points computations with intervals (as in Exercise 1) we get the following result



The interval obtained when entering B3 is an over-approximation, the exact one would be [24,32].

Hence Frama-C would report a false posityive for a potential buffer overflow in B3. A way to get more precise results is to use "slevel" or "loop-unrolling" options in order to unroll the loop 5 times before applying the widening operator ...

Symbolic Execution

Exercise 3

We consider the following code, where variable x is a user input :

```
#define N ...
unsigned x, y z ;
int T[N] ;

read(x) ;
z = 2*x ;
if (z<x+20) {
    y = z -10
    if (y > 12)
        T[y] = 0 ;
    else
        T[x] = 0 ;
} else {
        T[z+ 3] = 0 ;
}
```

Q1. Give its sets of execution paths and corresponding path predicates

PC1: z0=2*x0 and z0<x0+20 and y0=z0-10 and y0>12
PC2: z0=2*x0 and z0<x0+20 and y0=z0-10 and y0<=12
PC3: z0=2*x0 and z0>=x0+20

Q2. Is there a valid input valuation for each of these path predicates ?

x0=30 satisfies PC1
x0=10 satisfies PC2
x0=21 satisfies PC3

Q3. How to extend theses path predicates in order to detect potential buffer overflows ?

We have to add extra constraints on each PC, s.t. a BoF occurs if one of them is satisfiable:

PC1: z0=2*x0 and z0<x0+20 and y0=z0-10 and y0>12 and (y<0 or y>=N)
PC2: z0=2*x0 and z0<x0+20 and y0=z0-10 and y0<12 and (x<0 or x>=N)
PC3: z0=2*x0 and z0>=x0+20 and ((z+3<0 or (z+3)>=N)

Exercise 4

We consider the following code example , where x is a **positive user input** :

```
#define N 3

int x ;
int Tab[N] ;

read (x) ;
while (x<N)
      x = x+2 ;
tab[x] = 0
```

Q1. Is a symbolic tool like PathCrawler able to find **all** the execution paths triggering the vulnerability ? Explain your answer, giving the set of path predicates to consider and their corresponding solutions (assuming no arithmetic overflows)

There are 3 execution paths allowing to reach the potentially vulnerable statement tab[x]=0:

- entering an initial value for x larger than N
  PC1 : x0>=N and (x0<0 or x0>=N), satisfiable for any x0>=N
  The buffer overflow is always triggered in this case (without arithmetic overflows).
  Note that if we consider arithmetic overflows the BoF is not trigerred for x0 in {UINT_MAX-1, UINT_MAX, UINT_MAX+1}
- unrolling the loop exactly once
  PC2 : x0<N and x1=x0+2 and x1>=N, satisfiable for x0=1 or x0=2, hence trigerring the BoF
- unrolling the loop twice
  x0<N and x1=x0+2 and and x1<N and x2=x1+2 and (x2>=N), satisfiable for x0=0, triggering the BoF

All of them could be found by a symbolic execution engine.

Q2 . Same question with N=1000

For N=1000, the number of execution paths becomes quite large (about 1000 ... !).
It is still feasible to find all of them using a symbolic execution engine, but in practice it would depend on the exploration strategy ...