Coursework 03
Detection of Software Vulnerabilities: Static Analysis (Part I)
Model Solution

1. Satisfiability Modulo Theories

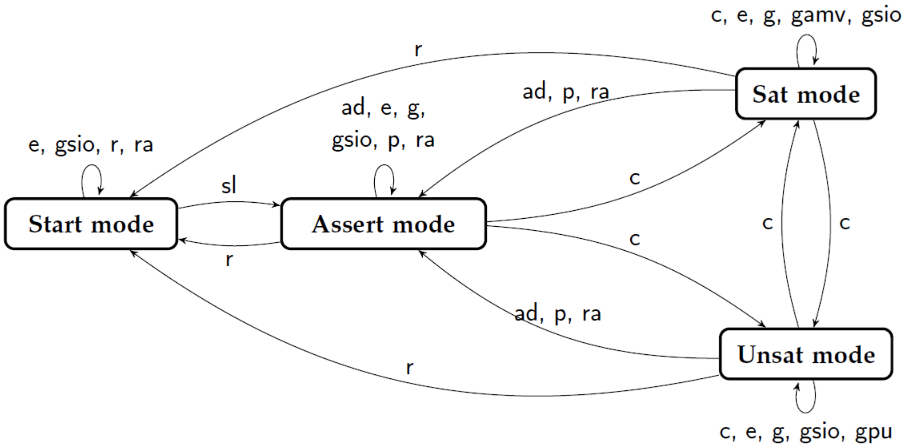
a. Syntax:

```
F ::= F con F | ¬F | A
con ::= ∧ | ∨ | ⊕ | ⇒ | ⇔
A ::= T rel T | Id | true | false
rel ::= < | ⩽ | > | ⩾ | = | 6=
T ::= T op T | ∼ T | ite(F, T,T) | Const | Id |
Extract(T, i, j)| SignExt(T, k)| ZeroExt(T, k)
op ::= + | − | ∗ | / | rem | << | >> | & | | | ⊕ | @
```

Here, F denotes Boolean-valued expressions with atoms A, and T denotes terms built over integers, reals, and bit-vectors. The logical connectives con consist of conjunction (∧), disjunction (∨), exclusive-or (⊕), implication (⇒), and equivalence (⇔). The bit-level operators are and (&), or (|), exclusive-or (⊕), complement (∼), rightshift (>>), and left-shift (<<). Extract (T, i, j) denotes bitvector extraction from bits i down to j to yield a new bitvector of size i−j+1, while @ denotes the concatenation of the given bit-vectors. SignExt (T, k) extends a bitvector of size w to the signed equivalent bit-vector of size w + k, while ZeroExt (T, k) extends the bit-vector with zeros to the unsigned equivalent bit-vector of size w+k. The conditional expression ite(f, t1, t2) takes a Boolean formula f and, depending on its value, selects either the second or the third argument. The interpretation of the relational operators (i.e., <, ≤, >, ≥), the nonlinear arithmetic operators ∗, /, the remainder (rem), and the right-shift operator (>>) depends on whether their arguments are unsigned or signed bit-vectors, integers, or real numbers. The arithmetic operators induce checks to ensure that the arithmetic operations do not overflow and/or underflow.

b. Semantics:

The expected interaction mode with a compliant solver is that of a read-eval-print loop: the user or client application issues a command in the format of the Command Language to the SMT solver via the solver's standard textual input channel. The solver then responds over two textual output channels, one for standard output and one for diagnostic output, and waits for another command. A non-interactive mode is also allowed where the solver reads commands from a script stored in a file. The following graph describes the Abstract view of transitions between solver execution modes.




```
Command name abbreviations:
      ad = assert, declare-*, define-*
      c = check-sat*
      e = echo
      g = get-assertions
      gamv = get-assignment, get-model, get-value
      gsio = get-info, get-option, set-info, set-option
      gpu = get-proof, get-unsat-*
      p = pop, push
      r = reset
      ra = reset-assertions
      sl = set-logic
```

At a high-level, a compliant solver can be understood as being at all times in one of four execution modes: a start mode, an assert mode, and two query modes, sat and un-sat. The solver starts in start mode, moves to assert mode once logic is set, and then moves to one of the two query modes after executing a check command. Any command other than reset that modifies the assertion stack brings the solver back from a query mode to the assert mode. The reset command takes the solver back to start mode. The transition system in the figure illustrates in some detail which commands can trigger, which mode transitions. The set of labels for each transition describes the commands that may cause it. Except for exit, if a command does not appear on any transitions originating from a mode, it is not permitted in that mode. The exit command, which causes the solver to quit, can be issued in any mode.

Floating-Point:

Constructor:
(fp (_ BitVec 1) (_ BitVec eb) (_ BitVec i) (_ FloatingPoint eb sb))
where eb and sb are numerals greater than 1 and i = sb - 1.

Arithmetic: eb and sb are numerals greater than 1

- Absolute value
  (fp.abs (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Negation (no rounding needed)
  (fp.neg (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Addition
  (fp.add RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Subtraction
  (fp.sub RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Multiplication
  (fp.mul RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Division
  (fp.div RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Fused multiplication and addition; (x * y) + z
  (fp.fma RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Square root
  (fp.sqrt RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Remainder: x - y * n, where n in Z is nearest to x/y
  (fp.rem (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Rounding to integral
  (fp.roundToIntegral RoundingMode (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

- Minimum and maximum
  (fp.min (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))
  (fp.max (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) (_ FloatingPoint eb sb))

Positive and negative infinities and zeroes:

((_ +oo eb sb) (_ FloatingPoint eb sb))
((_ -oo eb sb) (_ FloatingPoint eb sb))

Semantically, for each eb and sb, there is exactly one +infinity value and exactly one -infinity value in the set denoted by (_ FloatingPoint eb sb), in agreement with the IEEE 754-2008 standard. However, +/-infinity can have two representations in this theory. E.g., +infinity for sort (_ FloatingPoint 2 3) is represented equivalently by (_ +oo 2 3) and (fp #b0 #b11 #b00).

((_ +zero eb sb) (_ FloatingPoint eb sb))
((_ -zero eb sb) (_ FloatingPoint eb sb))

The +zero and -zero symbols are abbreviations for the corresponding fp literals. E.g., (_ +zero 2 4) abbreviates (fp #b0 #b00 #b000) (_ -zero 3 2) abbreviates (fp #b1 #b000 #b0)

NaNs

((_ NaN eb sb) (_ FloatingPoint eb sb))
For each eb and sb, there is precisely one NaN in the set denoted by (_ FloatingPoint eb sb), in agreement with Level 2 of IEEE 754-2008 (floating-point data). There is no distinction in this theory between a ``quiet'' and a ``signaling'' NaN. NaN has several representations, e.g.,(_ NaN eb sb) and any term of the form (fp t #b1..1 s) where s is a binary containing at least a 1 and t is either #b0 or #b1.

Comparison operators:
Note that all comparisons evaluate to false if either argument is NaN
(fp.leq (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool :chainable)
(fp.lt (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool :chainable)
(fp.geq (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool :chainable)
(fp.gt (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool :chainable)

IEEE 754-2008 equality (as opposed to SMT-LIB =)
(fp.eq (_ FloatingPoint eb sb) (_ FloatingPoint eb sb) Bool :chainable)

Five rounding modes:
RNE = roundNearestTiesToEven.
RNA = roundNearestTiesToAway.
RTP = roundTowardPositive.
RTN = roundTowardNegative.
RTZ = roundTowardZero.

Used mostly for conversion from and to other types/sorts.
Example:

((_ to_fp eb sb) RoundingMode (_ BitVec m) (_ FloatingPoint eb sb))

Let b in [[(_ BitVec m)]] and let n be the signed integer represented by b (in 2's complement format).
[[(_ to_fp eb sb)]](r, b) = +/-infinity if n is too large/too small to be represented as a finite number of [[(_ FloatingPoint eb sb)]];
[[(_ to_fp eb sb)]](r, x) = y otherwise, where y is the finite number such that [[fp.to_real]](y) is closest to n according to rounding mode r.

```
0 10000010 11001001000011111100111
^     ^               ^
|     |               |
|     |               +--- significand = 0.7853975...
|     |
|     +------------------- exponent = 2 (130 - 128)
|
+------------------------- sign = 0 (positive)

value= -1^(sign) * 2^(exponent) * (significand)
value= -1^0 * 2^2 * 0.7853975...
value= 3.14159...
```

2) (Solving SMT equations) Check the satisfiability of these propositional equations. You must use the theory of bit-vector (QF_BF) implemented in the Z3 SMT solver. 2

http://smtlib.cs.uiowa.edu/logics-all.shtml
https://rise4fun.com/z3/tutorial

Using the websites above to understand the functions of QF_BV. Also, to know how to use the basic Z3 commands. For example, we want to check a ∧ ¬a is always false. So the correct commands to use here are:

```
(declare-const a (_ BitVec 1))
(assert (not (= (bvand a (bvnot a)) #b0)))
(check-sat)

$ z3 f4
unsat
```

The first line where we declare our variable "a" (as a bit vector of length 1 bit).

In the second line, we use the assert command (assert <boolean>), where we replace <boolean> with the boolean statement we want to verify. The form we put our statement in is ((a ∧ ¬a) ≠ #b0) where the variable a in a bit vector and #b0 is zero in bit vector length 1 (if length 2 it will be #b00 and so on). We make the comparison to zero because we want the results of this command to be boolean to be used as input for assert command. Since variable a is a bit vector, we need to use bitvector commands such as bvand and bvnot to check the satisfiability of our original statement "a ∧ ¬a". The result, as expected, is unsat.

a. (¬a ∨ ¬b) ∧ (¬b ∨ c) ∧ b

```
(declare-const a (_ BitVec 1))
(declare-const b (_ BitVec 1))
(declare-const c (_ BitVec 1))
(assert (not (= (bvand (bvand (bvor (bvnot a) (bvnot b)) (bvor
(bvnot b) c)) b) #b0)))
(check-sat)
(get-model)
```

Output:

```
$z3 f1
sat
(model
  (define-fun b () (_ BitVec 1)
```

```
    #b1)
  (define-fun c () (_ BitVec 1)
    #b1)
  (define-fun a () (_ BitVec 1)
    #b0)
)
```

b. (((d ∧ c) ∨ (p ∧ ¬ ((c ∧ ¬ (d))))) ≡ ((c ∧ d) ∨ (p ∧ c) ∨ (p ∧ ¬ (d))))

```
(declare-const d (_ BitVec 1))
(declare-const c (_ BitVec 1))
(declare-const p (_ BitVec 1))
(assert (not (= (bvxnor (bvor  (bvand d c) (bvand p (bvand c
(bvnot d)))) (bvor (bvor (bvand c d) (bvand p c)) (bvand p (bvnot
d)))) #b0)))
(check-sat)
(get-model)
```

Output:

```
$ z3 f2
sat
(model
  (define-fun d () (_ BitVec 1)
    #b1)
  (define-fun p () (_ BitVec 1)
    #b1)
  (define-fun c () (_ BitVec 1)
    #b1)
)
```

c. (((a) ∧ ((b) → ¬ (a)) ≡ ¬ (b))

(((a) ∧ (¬(b) ∨ ¬ (a)) ≡ ¬ (b))
(bvxnor (bvand a (bvor (bvnot b) (bvnot a))) (bvnot b))

```
(declare-const a (_ BitVec 1))
(declare-const b (_ BitVec 1))
(assert (not (= (bvxnor (bvand a (bvor (bvnot b) (bvnot a)))
(bvnot b)) #b0)))
(check-sat)
```

```
(get-model)
```

Output:

```
$ z3 f3
sat
(model
  (define-fun b () (_ BitVec 1)
    #b1)
  (define-fun a () (_ BitVec 1)
    #b1)
)
```

3. Soundness and Completeness

Advantages:

First, it is automatic; It does not rely on complicated interaction with the user for incremental property proving. If a property does not hold, the model checker generates a counterexample trace automatically. Second, the systems being checked are assumed to be finite; typical examples of finite systems for which model checking has successfully been applied are digital sequential circuits and communication protocols. Finally, temporal logic is used for specifying the system properties. Thus, model checking can be summarized as an algorithmic technique for checking temporal properties of finite systems.

Disadvantages：

1. For an infinite state program model, the model detection problem itself is an undecidable problem due to the infinite state.
2. For a finite state software system, state-space explosion is a fundamental and challenging problem to be solved.
3. The loss of high-level information during the translation prevents potential optimizations from pruning the state space to be explored.
4. The size of the encoding increases with the size of the arrays used in the program.

Techniques based on induction can be used to make BMC complete and soundness.
BMC algorithm is sound in the following sense: If the algorithm reports "reachable", then indeed, a bad state is reachable. If the algorithm reports "unreachable up to n steps", then there is no path of length ≤ n that reaches a bad state. In order to make BMC complete, it should report unreachable if and only if there are no reachable bad states.

Completeness:
CT (completeness threshold) represents a value of depth k such that if there are no counterexamples in length CT or less.

[Completeness and Complexity of Bounded Model Checking](#)
[Edmund Clarke, Daniel Kroening, Joël Ouaknine, Ofer Strichman.](#)

4) (C/C++ API of Z3) Write a program using the C/C++ API of Z3 to verify the following C programs to check for buffer overflow and memory leak. Note that you must build two sets of formulas C and P and then check for the satisfiability of the resulting equation C ∧ ¬ P.

https://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.225.8231
http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
gcc test_capi.c -lz3

a. Buffer overflow.

```
int main() {
      int a[2], i, x, *p;
      p=a;
      if (x==0)
            a[i]=0;
```

```
        else
              a[i+1]=1;
        assert(*(p+2)==1);
  }
```

C code:

```
       Z3_context ctx = mk_context();
       Z3_solver s = mk_solver(ctx);
       Z3_sort array_sort, bool_sort, pointer_sort,bv1_sort,bv32_sort,bv64_sort;
       Z3_ast c[10], p[10],C,P,TRUE,thm;

       Z3_ast a0, a1, a2, a3, a4;
       Z3_ast x0;
       Z3_ast i0, i1, i2, i3;
       Z3_ast g0;
       Z3_ast p0,p1,p2,p3;
       Z3_ast guard_exec;
       Z3_ast addr_a;
       Z3_ast zero, one, two;
       Z3_ast addr_a_start, mem_zero,mem_8;

       printf("\nProgram1: Buffer overflow\n");
       bv1_sort = Z3_mk_bv_sort(ctx,1);
       bv32_sort = Z3_mk_bv_sort(ctx,32);
       bv64_sort = Z3_mk_bv_sort(ctx,64);
       array_sort = Z3_mk_array_sort(ctx, bv1_sort, bv32_sort);
       bool_sort = Z3_mk_bool_sort(ctx);

       //pointer_sort as tuple
       printf("\n making pointer type\n");
       Z3_symbol mk_tuple_name;
       Z3_func_decl mk_pointer_sort_decl;
       Z3_symbol proj_names[2];
       Z3_sort proj_sorts[2];
       Z3_func_decl proj_decls[2];
       Z3_func_decl get_object, get_offset;

       /* Create pair (tuple) type */
       mk_tuple_name = Z3_mk_string_symbol(ctx, "struct_pointer_type_struct");
       proj_names[0] = Z3_mk_string_symbol(ctx, "pointer_object");
       proj_names[1] = Z3_mk_string_symbol(ctx, "pointer_offset");
       proj_sorts[0] = bv64_sort;
       proj_sorts[1] = bv64_sort;
       /* Z3_mk_tuple_sort will set mk_tuple_decl and proj_decls */
       pointer_sort = Z3_mk_tuple_sort(ctx, mk_tuple_name, 2, proj_names,
proj_sorts,
                    &mk_pointer_sort_decl, proj_decls);
       get_object = proj_decls[0]; /* function that extracts the first element of
a tuple. object*/
       get_offset = proj_decls[1]; /* function that extracts the second element of
a tuple. offset */

       printf("struct_pointer_type_struct: ");
       display_sort(ctx, stdout, pointer_sort);

       printf("\n initialising memory values and &a[0]\n");
       addr_a_start = Z3_mk_numeral(ctx,"2",bv64_sort);
       mem_zero = Z3_mk_numeral(ctx,"0",bv64_sort);
       mem_8 = Z3_mk_numeral(ctx,"8",bv64_sort);

       addr_a = mk_binary_app(ctx, mk_pointer_sort_decl, addr_a_start ,mem_zero);

       printf("\n making constants\n");
       zero = Z3_mk_numeral(ctx, "0", bv32_sort);
       one = Z3_mk_numeral(ctx, "1", bv32_sort);
       two = Z3_mk_numeral(ctx, "2", bv32_sort);
```

```
        TRUE=Z3_mk_true(ctx);
        printf("\n defining variables\n");
        i0 = mk_var(ctx, "i0", bv32_sort);
        i1 = mk_var(ctx, "i1", bv32_sort);
        i2 = mk_var(ctx, "i2", bv32_sort);
        i3 = mk_var(ctx, "i3", bv32_sort);

        guard_exec = mk_var(ctx,"guard_exec",bool_sort);

        g0 = mk_var(ctx, "g1", bool_sort);

        x0 = mk_var(ctx, "x0", bv32_sort);

        a0 = mk_var(ctx, "a0", array_sort);
        a1 = mk_var(ctx, "a1", array_sort);
        a2 = mk_var(ctx, "a2", array_sort);
        a3 = mk_var(ctx, "a3", array_sort);
        a4 = mk_var(ctx, "a4", array_sort);

        p0 = mk_var(ctx, "p0", pointer_sort);
        p1 = mk_var(ctx, "p1", pointer_sort);
        p2 = mk_var(ctx, "p2", pointer_sort);
        p3 = mk_var(ctx, "p3", pointer_sort);
        //p4 = mk_var(ctx, "a4", array_sort);

        printf("\n Constructing C\n");
        printf("  p1 := store(p0, 0, &a[0])\n");
        //     p1 := store(p0, 0, &a[0])
        printf("  ∧p2 := store(p1, 1, 0)₩n");
        //        ∧ p2 := store(p1, 1, 0)
        c[5] = Z3_mk_eq(ctx, p2, addr_a);

        printf("  ∧ g2 := (x2 == 0)₩n");
        //        ∧ g2 := (x2 == 0)
        c[0] = Z3_mk_eq(ctx, g0, Z3_mk_eq(ctx, x0, zero));

        printf("  ∧ a1 := store(a0, i0, 0)₩n");
        //        ∧ a1 := store(a0, i0, 0)
        c[1] = Z3_mk_eq(ctx, a1, Z3_mk_store(ctx, a0, Z3_mk_extract(ctx,0,0,i0),
zero));

        printf("  ∧ a2 := a0₩n");
        //        ∧ a2 := a0
        c[2] = Z3_mk_eq(ctx, a2, a0);

        printf("  ∧ a3 := store(a2, 1+ i0, 1)₩n");
        //        ∧ a3 := store(a2, 1+ i0, 1)
        c[3] = Z3_mk_eq(ctx, a3, Z3_mk_store(ctx, a2,
Z3_mk_extract(ctx,0,0,Z3_mk_bvadd(ctx, i0, one)), one));

        printf("  ∧a4 := ite(g1, a1, a3)₩n");
        //        ∧ a4 := ite(g1, a1, a3)
        c[4] = Z3_mk_eq(ctx, a4, Z3_mk_ite(ctx, g0, a1, a3));

        printf("  ∧p3 := store(p2, 1, select(p2 , 1)+2)₩n");
        //        ∧ p3 := store(p2, 1, select(p2 , 1)+2)
        c[6] = Z3_mk_eq(ctx, p3, mk_binary_app(ctx, mk_pointer_sort_decl,
                        mk_unary_app(ctx, get_object, addr_a) ,mem_8));
```

```
	printf("\nConstruct C\n");
	C = Z3_mk_and(ctx, 7, c);

	printf("\nassert C\n");
	for(int i=0;i<6;i++)
	{
		Z3_solver_assert(ctx,s,c[i]);
	}

	printf("C := [%s]\n", Z3_ast_to_string(ctx, C));


	printf("\nConstructing P\n");
	printf("₩n  i0 ≥ 0₩n");
	//	i0 ≥ 0
	p[0] = Z3_mk_bvuge(ctx, i0, zero);

	printf("  ∧ i0 < 2₩n");
	//  ∧ i0 < 2
	p[1] = Z3_mk_bvult(ctx, i0, two);

	printf("  ∧ 1+ i0 ≥ 0₩n");
	//	∧ 1+ i0 ≥ 0
	p[2] = Z3_mk_bvuge(ctx, Z3_mk_bvadd(ctx, i0, one), zero);

	printf("  ∧ 1+ i0 < 2₩n");
	//  ∧ 1+ i0 < 2
	p[3] = Z3_mk_bvult(ctx, Z3_mk_bvadd(ctx, i0, one), two);

	printf("  ∧ select(p3 , 0) == &a[0]₩n");
	//	∧ select(p3 , 0) == &a[0]
	p[4] = Z3_mk_eq(ctx, mk_unary_app(ctx, get_object, p3) , mk_unary_app(ctx, 
get_object, addr_a));
	printf("  ∧ select(select(p3 , 0),select(p3 , 1)) == 1₩n");
	//	∧ select(select(p3 , 0),select(p3 , 1)) == 1
	p[5] = 
Z3_mk_eq(ctx,Z3_mk_select(ctx,a3,Z3_mk_extract(ctx,0,0,Z3_mk_bvadd(ctx, i0, 
two))), one);

	printf("\nConstruct ¬P\n");
	for(int i=0;i<6;i++)
	{
		p[i] = Z3_mk_not(ctx, 
Z3_mk_implies(ctx,/*Z3_mk_eq(ctx,one,one)*/TRUE,Z3_mk_implies(ctx,guard_exec,p[i]
)));
	}

	printf("\nConstruct ¬P\n");
	P = Z3_mk_or(ctx, 6, p);

	printf("\nassert P\n");
	Z3_solver_assert(ctx, s, P);

	printf("P := [%s]\n", Z3_ast_to_string(ctx, P));

	Z3_ast cp[] = {C,P};
	printf("solver := [%s]\n", Z3_solver_to_string(ctx, s));
	check(ctx, s,  Z3_L_TRUE);
```

```
        printf("\nEND\n");
        del_solver(ctx, s);
        Z3_del_context(ctx);
}
```

SMT Formula:

$$C \coloneqq (p1 \coloneqq store(p0,0,a) \wedge p2 \coloneqq store(p1,1,0) \wedge g1 \coloneqq (x1 == 0) \wedge a1 :\\ = store(a0,i0,0)\\ \wedge a2 \coloneqq a0 \wedge a3 \coloneqq store(a2,1+i0,1) \wedge a4 \coloneqq ite(g1,a1,a3)\\ \wedge p3 \coloneqq store(p2,1,select(p2,1)+2))$$

$$P \coloneqq i0 \geq 0 \wedge i0 < 2 \wedge 1+i0 \geq 0 \wedge 1+i0 < 2\\ \wedge select(p3,0) == a[0] \wedge select(select(p3,0),select(p3,1)) == 1$$

Output:

```
(declare-datatypes ((struct_pointer_type_struct 0))
(((struct_pointer_type_struct (pointer_object (_ BitVec 64))
(pointer_offset (_ BitVec 64))))))
(declare-fun x0 () (_ BitVec 32))
(declare-fun g1 () Bool)
(declare-fun i0 () (_ BitVec 32))
(declare-fun a0 () (Array (_ BitVec 1) (_ BitVec 32)))
(declare-fun a1 () (Array (_ BitVec 1) (_ BitVec 32)))
(declare-fun a2 () (Array (_ BitVec 1) (_ BitVec 32)))
(declare-fun a3 () (Array (_ BitVec 1) (_ BitVec 32)))
(declare-fun a4 () (Array (_ BitVec 1) (_ BitVec 32)))
(declare-fun p2 () struct_pointer_type_struct)
(declare-fun guard_exec () Bool)
(declare-fun p3 () struct_pointer_type_struct)
(assert (= g1 (= x0 #x00000000)))
(assert (= a1 (store a0 ((_ extract 0 0) i0) #x00000000)))
(assert (= a2 a0))
(assert (= a3 (store a2 ((_ extract 0 0) (bvadd i0 #x00000001))
#x00000001)))
(assert (= a4 (ite g1 a1 a3)))
(assert (= p2 (struct_pointer_type_struct #x0000000000000002
#x0000000000000000)))
(assert (let ((a!1 (not (=> true (=> guard_exec (bvuge i0
#x00000000)))))
      (a!2 (not (=> true (=> guard_exec (bvult i0 #x00000002)))))
      (a!3 (=> true (=> guard_exec (bvuge (bvadd i0 #x00000001)
#x00000000))))
      (a!4 (=> true (=> guard_exec (bvult (bvadd i0 #x00000001)
#x00000002))))
```

```
      (a!5 (=> guard_exec
               (= (pointer_object p3)
                  (pointer_object (struct_pointer_type_struct
                                    #x0000000000000002
                                    #x0000000000000000)))))
      (a!6 (= (select a3 ((_ extract 0 0) (bvadd i0 #x00000002)))
#x00000001)))
   (or a!1
       a!2
       (not a!3)
       (not a!4)
       (not (=> true a!5))
       (not (=> true (=> guard_exec a!6))))))
]
sat
```

b. Memory leak.

```
#include <stdlib.h>
int main() {
      char *p = malloc(5);
      char *q = malloc(5);
      p=q;
      free(p);
      p = malloc(5);
      free(p);
      return 0;
}
```

C code:

```
Z3_context ctx = mk_context();
      Z3_solver s = mk_solver(ctx);
      Z3_sort array_sort, bool_sort, pointer_sort,bv32_sort,bv64_sort;
      Z3_ast c[30],p[10],C,P;

      Z3_ast p0, p1, p2, q0;

      Z3_ast alloc0,alloc1,alloc2,alloc3,alloc4,alloc5,alloc6;
      Z3_ast dealloc0,dealloc1,dealloc2,dealloc3,dealloc4,dealloc5,dealloc6;
      Z3_ast loc0,loc1,loc2;
      Z3_ast id0,id1,id2, s0,s1,s2;
      Z3_ast malloc_r0,malloc_r1,malloc_r2;

      Z3_ast mem_null, mem_2,mem_3,mem_4,size5,TRUE,FALSE,zero,one,two,three;

      printf("\nProgram 2: Memory leak.\n");
      bool_sort = Z3_mk_bool_sort(ctx);
      bv32_sort = Z3_mk_bv_sort(ctx,32);
      bv64_sort = Z3_mk_bv_sort(ctx,64);
      array_sort = Z3_mk_array_sort(ctx, bv64_sort, bool_sort);
```

```
	//pointer_sort as tuple
	printf("\n making pointer type\n");
	Z3_symbol mk_tuple_name;
	Z3_func_decl mk_pointer_sort_decl;
	Z3_symbol proj_names[2];
	Z3_sort proj_sorts[2];
	Z3_func_decl proj_decls[2];
	Z3_func_decl get_object, get_offset;


	/* Create pointer_sort type */
	mk_tuple_name = Z3_mk_string_symbol(ctx, "struct_pointer_type_struct");
	proj_names[0] = Z3_mk_string_symbol(ctx, "pointer_object");
	proj_names[1] = Z3_mk_string_symbol(ctx, "pointer_offset");
	proj_sorts[0] = bv64_sort;
	proj_sorts[1] = bv64_sort;
	/* Z3_mk_tuple_sort will set mk_pointer_sort_decl and proj_decls */
	pointer_sort = Z3_mk_tuple_sort(ctx, mk_tuple_name, 2, proj_names,
proj_sorts,
	                  &mk_pointer_sort_decl, proj_decls);
	get_object = proj_decls[0]; /* function that extracts the first element of
a tuple. object */
	get_offset = proj_decls[1]; /* function that extracts the second element of
a tuple. offset */

	printf("struct_pointer_type_struct: ");
	display_sort(ctx, stdout, pointer_sort);

	printf("\n making constants\n");
	mem_null = Z3_mk_numeral(ctx, "0", bv64_sort);
	mem_2 = Z3_mk_numeral(ctx, "2", bv64_sort);
	mem_3 = Z3_mk_numeral(ctx, "3", bv64_sort);
	mem_4 = Z3_mk_numeral(ctx, "4", bv64_sort);
	size5 = Z3_mk_numeral(ctx, "5", bv64_sort);
	TRUE=Z3_mk_true(ctx);
	FALSE = Z3_mk_false(ctx);
	zero = Z3_mk_numeral(ctx, "0", bv64_sort);
	one = Z3_mk_numeral(ctx, "1", bv64_sort);
	two = Z3_mk_numeral(ctx, "2", bv64_sort);
	three = Z3_mk_numeral(ctx, "3", bv64_sort);

	printf("\n defining variables\n");
	alloc0 = mk_var(ctx, "alloc0", array_sort);
	alloc1 = mk_var(ctx, "alloc1", array_sort);
	alloc2 = mk_var(ctx, "alloc2", array_sort);
	alloc3 = mk_var(ctx, "alloc3", array_sort);
	alloc4 = mk_var(ctx, "alloc4", array_sort);
	alloc5 = mk_var(ctx, "alloc5", array_sort);
	alloc6 = mk_var(ctx, "alloc6", array_sort);

	dealloc0 = mk_var(ctx, "dealloc0", array_sort);
	dealloc1 = mk_var(ctx, "dealloc1", array_sort);
	dealloc2 = mk_var(ctx, "dealloc2", array_sort);
	dealloc3 = mk_var(ctx, "dealloc3", array_sort);
	dealloc4 = mk_var(ctx, "dealloc4", array_sort);
	dealloc5 = mk_var(ctx, "dealloc5", array_sort);
	dealloc6 = mk_var(ctx, "dealloc6", array_sort);
```

```
	id0 = mk_var(ctx,"id0",bv64_sort);
	id1 = mk_var(ctx,"id1",bv64_sort);
	id2 = mk_var(ctx,"id2",bv64_sort);

	loc0 = mk_var(ctx,"loc0",bv64_sort);
	loc1 = mk_var(ctx,"loc1",bv64_sort);
	loc2 = mk_var(ctx,"loc2",bv64_sort);

	s0 = mk_var(ctx,"s0",bv64_sort);
	s1 = mk_var(ctx,"s1",bv64_sort);
	s2 = mk_var(ctx,"s2",bv64_sort);

	p0 = mk_var(ctx, "p0", pointer_sort);
	p1 = mk_var(ctx, "p1", pointer_sort);
	p2 = mk_var(ctx, "p2", pointer_sort);

	q0 = mk_var(ctx, "q0", pointer_sort);

	malloc_r0 = mk_var(ctx, "malloc_r0", pointer_sort);
	malloc_r1 = mk_var(ctx, "malloc_r1", pointer_sort);
	malloc_r2 = mk_var(ctx, "malloc_r2", pointer_sort);

	printf("\n Constructing C\n");
	//	//p1 = malloc(5)
	//	do1.ρ=1
	c[0] = Z3_mk_eq(ctx,id0,one);
	c[1] = Z3_mk_eq(ctx,loc0,mem_2);
	//	∧ do1.s=5
	c[2] = Z3_mk_eq(ctx,s0,size5);
	//	∧ do1.υ=true
	c[3] = Z3_mk_eq(ctx,alloc1,Z3_mk_store(ctx,alloc0,loc0,TRUE));
	c[4] = Z3_mk_eq(ctx,dealloc1,Z3_mk_store(ctx,dealloc0,loc0,FALSE));
	//	∧ p1=do1
	c[5] = Z3_mk_eq(ctx, malloc_r0, mk_binary_app(ctx, mk_pointer_sort_decl,
loc0 ,zero));
	c[6] = Z3_mk_eq(ctx,p0,mk_binary_app(ctx, mk_pointer_sort_decl,
mk_unary_app(ctx, get_object, malloc_r0) ,zero));
	//	//q1 = malloc(5)
	//	∧ do2.ρ=2
	c[7] = Z3_mk_eq(ctx,id1,one);
	c[8] = Z3_mk_eq(ctx,loc1,mem_3);
	//	∧ do2.s=5
	c[9] = Z3_mk_eq(ctx,s1,size5);
	//	∧ do2.υ=true
	c[10] = Z3_mk_eq(ctx,alloc2,Z3_mk_store(ctx,alloc1,loc1,TRUE));
	c[11] = Z3_mk_eq(ctx,dealloc2,Z3_mk_store(ctx,dealloc1,loc1,FALSE));
	//	∧ q1=do2
	c[12] = Z3_mk_eq(ctx, malloc_r1, mk_binary_app(ctx, mk_pointer_sort_decl,
loc1 ,zero));
	c[13] = Z3_mk_eq(ctx,q0,mk_binary_app(ctx, mk_pointer_sort_decl,
mk_unary_app(ctx, get_object, malloc_r1) ,zero));
	//	//p2=q1
	//	∧ p2=do2
```

```
	c[14] = Z3_mk_eq(ctx,p1,mk_binary_app(ctx, mk_pointer_sort_decl,
mk_unary_app(ctx, get_object, q0) ,zero));
	//	//free (p2)
	//	∧ do2.υ=false
	c[15] = Z3_mk_eq(ctx,alloc3,Z3_mk_store(ctx,alloc2,mk_unary_app(ctx,
get_object, p1),FALSE));
	c[16] = Z3_mk_eq(ctx,dealloc3,Z3_mk_store(ctx,dealloc2,mk_unary_app(ctx,
get_object, p1),TRUE));
	//	//p3 = malloc(5)
	//	∧ do3.ρ=3
	c[17] = Z3_mk_eq(ctx,id2,three);
	c[18] = Z3_mk_eq(ctx,loc2,mem_4);
	//	∧ do3.s=5
	c[19] = Z3_mk_eq(ctx,s2,size5);
	//	∧ do3.υ=true
	c[20] = Z3_mk_eq(ctx,alloc4,Z3_mk_store(ctx,alloc3,loc2,TRUE));
	c[21] = Z3_mk_eq(ctx,dealloc4,Z3_mk_store(ctx,dealloc3,loc2,FALSE));
	//	∧ p3=do3
	c[22] = Z3_mk_eq(ctx, malloc_r2, mk_binary_app(ctx, mk_pointer_sort_decl,
loc2 ,zero));
	c[23] = Z3_mk_eq(ctx,p2,mk_binary_app(ctx, mk_pointer_sort_decl,
mk_unary_app(ctx, get_object, malloc_r2) ,zero));
	//	//free(p3)
	//	∧ do3.υ=false
	c[24] = Z3_mk_eq(ctx,alloc5,Z3_mk_store(ctx,alloc4,mk_unary_app(ctx,
get_object, p2),FALSE));
	c[25] = Z3_mk_eq(ctx,dealloc5,Z3_mk_store(ctx,dealloc4,mk_unary_app(ctx,
get_object, p2),TRUE));

	//c[26] =  Z3_mk_eq(ctx,alloc6,Z3_mk_store(ctx,alloc5,loc0,FALSE));
	//c[27] = Z3_mk_eq(ctx,dealloc6,Z3_mk_store(ctx,dealloc5,loc0,TRUE));

	printf("\n asserting C\n");
	for(int i=0;i<26;i++)
	{
		Z3_solver_assert(ctx,s,c[i]);
	}

	//printf("C := [%s]\n", Z3_ast_to_string(ctx, C));
	printf("\n Constructing P\n");
	p[0] = Z3_mk_not(ctx,Z3_mk_select(ctx,dealloc5,loc0));
	p[1] = Z3_mk_not(ctx,Z3_mk_select(ctx,dealloc5,loc1));
	p[2] = Z3_mk_not(ctx,Z3_mk_select(ctx,dealloc5,loc2));

	P = Z3_mk_or(ctx,3,p);
	printf("\n asserting P\n");

	Z3_solver_assert(ctx,s,P);

	printf("%s\n", Z3_solver_to_string(ctx, s));
	printf("-----------------------------------------------\n");
	check(ctx, s,  Z3_L_TRUE);

	printf("\nEND\n");
	del_solver(ctx, s);
```

```
    Z3_del_context(ctx);
```

SMT Formula:

$$C \coloneqq do1.\rho = 1 \wedge do1.s = 5 \wedge do1.v = true \wedge p = do1 \wedge do2.\rho = 2 \wedge do2.s = 5 \wedge do2.v = true \wedge q = do2 \wedge p = do2 \wedge do2.v = false \wedge do3.\rho = 3 \wedge do3.s = 5 \wedge do3.v = true \wedge p = do3 \wedge do3.v = false$$

$$P \coloneqq (do1.v \wedge \neg do2.v \neg do3.v)$$

Output:

```
(declare-datatypes ((struct_pointer_type_struct 0)) (((struct_pointer_type_struct
(pointer_object (_ BitVec 64)) (pointer_offset (_ BitVec 64))))))
(declare-fun id0 () (_ BitVec 64))
(declare-fun loc0 () (_ BitVec 64))
(declare-fun s0 () (_ BitVec 64))
(declare-fun alloc0 () (Array (_ BitVec 64) Bool))
(declare-fun alloc1 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc0 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc1 () (Array (_ BitVec 64) Bool))
(declare-fun malloc_r0 () struct_pointer_type_struct)
(declare-fun p0 () struct_pointer_type_struct)
(declare-fun id1 () (_ BitVec 64))
(declare-fun loc1 () (_ BitVec 64))
(declare-fun s1 () (_ BitVec 64))
(declare-fun alloc2 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc2 () (Array (_ BitVec 64) Bool))
(declare-fun malloc_r1 () struct_pointer_type_struct)
(declare-fun q0 () struct_pointer_type_struct)
(declare-fun p1 () struct_pointer_type_struct)
(declare-fun alloc3 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc3 () (Array (_ BitVec 64) Bool))
(declare-fun id2 () (_ BitVec 64))
(declare-fun loc2 () (_ BitVec 64))
(declare-fun s2 () (_ BitVec 64))
(declare-fun alloc4 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc4 () (Array (_ BitVec 64) Bool))
(declare-fun malloc_r2 () struct_pointer_type_struct)
(declare-fun p2 () struct_pointer_type_struct)
(declare-fun alloc5 () (Array (_ BitVec 64) Bool))
(declare-fun dealloc5 () (Array (_ BitVec 64) Bool))
(assert (= id0 #x0000000000000001))
(assert (= loc0 #x0000000000000002))
(assert (= s0 #x0000000000000005))
(assert (= alloc1 (store alloc0 loc0 true)))
(assert (= dealloc1 (store dealloc0 loc0 false)))
(assert (= malloc_r0 (struct_pointer_type_struct loc0 #x0000000000000000)))
(assert (= p0
   (struct_pointer_type_struct (pointer_object malloc_r0) #x0000000000000000)))
(assert (= id1 #x0000000000000001))
(assert (= loc1 #x0000000000000003))
(assert (= s1 #x0000000000000005))
(assert (= alloc2 (store alloc1 loc1 true)))
(assert (= dealloc2 (store dealloc1 loc1 false)))
(assert (= malloc_r1 (struct_pointer_type_struct loc1 #x0000000000000000)))
(assert (= q0
   (struct_pointer_type_struct (pointer_object malloc_r1) #x0000000000000000)))
(assert (= p1 (struct_pointer_type_struct (pointer_object q0)
#x0000000000000000)))
(assert (= alloc3 (store alloc2 (pointer_object p1) false)))
```

```
(assert (= dealloc3 (store dealloc2 (pointer_object p1) true)))
(assert (= id2 #x0000000000000003))
(assert (= loc2 #x0000000000000004))
(assert (= s2 #x0000000000000005))
(assert (= alloc4 (store alloc3 loc2 true)))
(assert (= dealloc4 (store dealloc3 loc2 false)))
(assert (= malloc_r2 (struct_pointer_type_struct loc2 #x0000000000000000)))
(assert (= p2
   (struct_pointer_type_struct (pointer_object malloc_r2) #x0000000000000000)))
(assert (= alloc5 (store alloc4 (pointer_object p2) false)))
(assert (= dealloc5 (store dealloc4 (pointer_object p2) true)))
(assert (or (not (=> true (select dealloc5 loc0)))
    (not (=> true (select dealloc5 loc1)))
    (not (=> true (select dealloc5 loc2)))))

--------------------------------------------
sat
dealloc0 -> ((as const (Array (_ BitVec 64) Bool)) false)
dealloc5 -> (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                          #x0000000000000003
                          true)
                   #x0000000000000004
                   true)
alloc0 -> ((as const (Array (_ BitVec 64) Bool)) false)
alloc5 -> (store (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                               #x0000000000000002
                               true)
                        #x0000000000000003
                        false)
                 #x0000000000000004
                 false)
p2 -> (struct_pointer_type_struct #x0000000000000004 #x0000000000000000)
malloc_r2 -> (struct_pointer_type_struct #x0000000000000004 #x0000000000000000)
dealloc4 -> (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                          #x0000000000000003
                          true)
                   #x0000000000000004
                   false)
alloc4 -> (store (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                               #x0000000000000002
                               true)
                        #x0000000000000003
                        false)
                 #x0000000000000004
                 true)
s2 -> #x0000000000000005
loc2 -> #x0000000000000004
id2 -> #x0000000000000003
dealloc3 -> (store ((as const (Array (_ BitVec 64) Bool)) false)
#x0000000000000003 true)
alloc3 -> (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                        #x0000000000000002
                        true)
                 #x0000000000000003
                 false)
p1 -> (struct_pointer_type_struct #x0000000000000003 #x0000000000000000)
q0 -> (struct_pointer_type_struct #x0000000000000003 #x0000000000000000)
malloc_r1 -> (struct_pointer_type_struct #x0000000000000003 #x0000000000000000)
dealloc2 -> ((as const (Array (_ BitVec 64) Bool)) false)
alloc2 -> (store (store ((as const (Array (_ BitVec 64) Bool)) false)
                        #x0000000000000002
                        true)
                 #x0000000000000003
                 true)
s1 -> #x0000000000000005
loc1 -> #x0000000000000003
id1 -> #x0000000000000001
```

```
p0 -> (struct_pointer_type_struct #x0000000000000002 #x0000000000000000)
malloc_r0 -> (struct_pointer_type_struct #x0000000000000002 #x0000000000000000)
dealloc1 -> ((as const (Array (_ BitVec 64) Bool)) false)
alloc1 -> (store ((as const (Array (_ BitVec 64) Bool)) false) #x0000000000000002
true)
s0 -> #x0000000000000005
loc0 -> #x0000000000000002
id0 -> #x0000000000000001
```

5) (Aligned Memory Model)

Derive the equations C and P from the following program C.You must enforce the C alignment rules when writing the resulting formula C ∧ ¬ P.

```
#include <stdint.h>
struct foo {
      uint16_t bar[2];
      uint8_t baz;
};
int main() {
      struct foo qux;
      qux.bar[0] = 10;
      qux.bar[1] = 20;
      qux.baz = 'C';
      struct foo *quux = &qux;
      quux++;
      quux->baz = 'D';
      return 0;
}
```

Assumptions:
1. Assuming the code will be run on an architecture in which each memory block is 4 bytes (32 bits). The memory alignment in this question will be based on fitting in such blocks.
2. Since a structure's size is NOT necessarily the sum of the size of its members, we assume the machine-dependent boundary alignment is done as shown in Figure 1.
3. Assuming the machine does 8-bit extraction. We're using 8-bit extraction because it represents byte by byte access in most machine architectures.

To fit in blocks of 4 bytes, struct foo will have paddings like this:

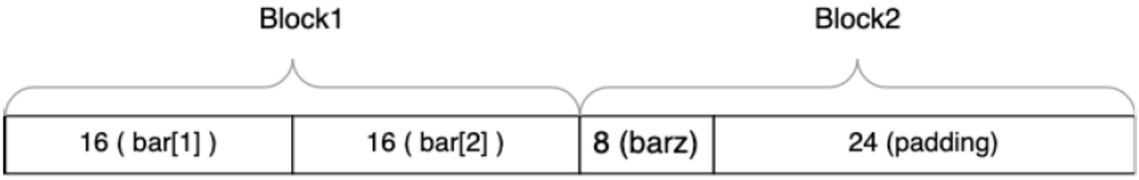


Figure 1

The legal and illegal extraction boundaries are defined in a code fragment. These boundaries will be used as guards in our P formulae.

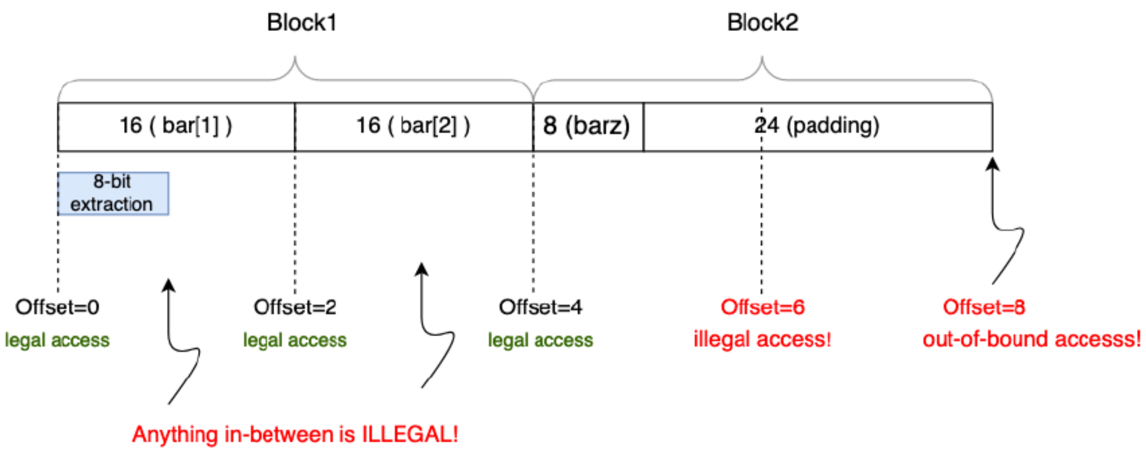


Figure 2: legal extraction boundaries to be used as guards based on 16-bit extraction

Hence the set of legal extraction boundaries is {0, 2, 4} based on a unit of 8-bit extraction, which corresponds to bar[0], bar[1] and barz in struct foo. Anything else is illegal, e.g., offset 6 will be considered as access to an undefined memory region. Anything after that is illegal, e.g., offset 8 will be considered as out-of-bound access.

```
int main() {
    // assuming we have  8 bit extraction
    struct foo qux;       // Assignment qux0 = 0, pointing to the beginning of struct
    qux.bar[0] = 10;      // offset 0, Assignment: qux1 = qux0 + 0
    // Added guard here: ASSERT( (qux1 == 0) \/ (qux1 == 2) \/ (qux1 = 4))
    qux.bar[1] = 20;      // offset 2, Assignment: qux2 = qux0 + 2
    // Added guard here: ASSERT( (qux2 == 0) \/ (qux2 == 2) \/ (qux2 = 4))
    qux.barz = 'C';       // offset 4, Assignment: qux3 = qux0+4
    // Added guard here: ASSERT( (qux3 == 0) \/ (qux3 == 2) \/ (qux3 = 4))
    struct foo *quux = &qux; // Assignment quux0 = qux0
    quux++;                  // offset 8, Assignment: quux1 = quux0 + 8
    // Added guard here: ASSERT( (quux1 == 0) \/ (quux1 == 2) \/ (quux1 = 4))
    quux->baz = 'D';         // accessing (quux+64) is out of bound!
    return 0;
}
```

Figure 3: Annotations represent SSA. Each assignment corresponds to a formula in C, and each ASSERT corresponds to a property we would like to check in P.

Each assignment (underlined by cyan) corresponds to a formula in C i.e., the essential assignment in SSA. Each ASSERT (underlined by blue) corresponds to a formula in P, i.e., the safety property we want to check.

The C and P formulae are shown here:

$$
\begin{aligned}
C = [\quad & \\
& (qux0=0) \land (qux1=qux0+0) \\
& \land (qux2=qux0+2) \land (qux3=qux0+4) \\
& \land (quux0=qux0) \land (quux1=quux0+8) \\
]\quad &
\end{aligned}
$$

$$
\begin{aligned}
P = [\quad & \\
& (qux1==0 \lor qux1==2 \lor qux1==4) \\
& \land (qux2==0 \lor qux2==2 \lor qux2==4) \\
& \land (qux3==0 \lor qux3==2 \lor qux3==4) \\
& \underline{\land (quux1==0 \lor quux1==2 \lor quux1==4)} \\
]\quad &
\end{aligned}
$$

Figure 4: C and P formulae

Z3 verification result and script are

```
z3_example> z3 q5_solution.smt2
sat
(
  (define-fun p4 () Bool
    false)
  (define-fun p3 () Bool
    true)
  (define-fun p2 () Bool
    true)
  (define-fun p1 () Bool
    true)
  (define-fun quux1 () Int
    8)
  (define-fun quux0 () Int
    0)
  (define-fun qux3 () Int
    4)
  (define-fun qux2 () Int
    2)
  (define-fun qux1 () Int
    0)
  (define-fun qux0 () Int
    0)
)
```

Figure 5: Z3 results showing C /\ ~P is satisfiable because the last line in P is violated, i.e. false

```
; variables as int type
(declare-const qux0 Int)
(declare-const qux1 Int)
(declare-const qux2 Int)
(declare-const qux3 Int)
(declare-const quux0 Int)
(declare-const quux1 Int)
; tmp variables to hold value of each formula in P
(declare-const p1 Bool)
(declare-const p2 Bool)
(declare-const p3 Bool)
(declare-const p4 Bool)

; Build C formulae
; qux0 = 0
(assert (= qux0 0))
; qux1 = qux0 + 0
(assert (= qux1 (+ qux0 0)))
; qux2 = qux0 + 2
(assert (= qux2 (+ qux0 2)))
; qux3 = qux0 + 4
(assert (= qux3 (+ qux0 4)))
; quux0 = qux0
(assert (= quux0 qux0))
; quux1 = quux0 + 8
(assert (= quux1 (+ quux0 8)))

; Build P formulae
; qux1==0 \/ qux1==2 \/ qux1==4
(assert (= p1 (or (= qux1 0) (or (= qux1 2) (= qux1 4)))))
; qux2==0 \/ qux2==2 \/ qux2==4
(assert (= p2 (or (= qux2 0) (or (= qux2 2) (= qux2 4)))))
; qux3==0 \/ qux3==2 \/ qux3==4
(assert (= p3 (or (= qux3 0) (or (= qux3 2) (= qux3 4)))))
; quux1==0 \/ qux1==2 \/ qux1==4
(assert (= p4 (or (= quux1 0) (or (= quux1 2) (= quux1 4)))))
; ~P
(assert (not (and p1 (and p2 (and p3 p4)))))

(check-sat)
(get-model) ; if satisfiable, print out the model
```

Figure 6: Z3 script to implement C /\ ~P in Q5