DCC888 - Static Analysis of Programs

Name: ____ **ID**: ________________

Name: ____ **ID**: ________________

By turning this exam in, I give my word that I have done it with my team only, on the understanding that we are allowed to consult any material publicly available, except material disclosed by other colleagues outside our team who are taking this course, or who have taken it in the past.

Goal: The goal of this exercise is to understand the principles of analyses and code transformations that make programs "isochronous". An isochronous program is a program that performs always the same operations, regardless of the input that it receives [2]. Isochronicity follows from two properties:
 * *Operation Invariance*: regardless of the input, the program executes always the same instructions in the same order.
 * *Data Invariance*: regardless of the input, the program accesses always the same addresses in the same order.
Below we show four examples of programs to illustrate these notions:

$\boxed{\text{!O \& !D}}$

```
int oFdF(int *a, int *b) {
  for (int i = 0; i < 2; i++) {
    if (a[i] != b[i]) {
      return 0;
    }
  }
  return 1;
}
```

$\boxed{\text{!O \& D}}$

```
int oFdT(int *a, int *b) {
  int r = 1;
  for (int i = 0; i < 2; i++) {
    if (a[i] != b[i]) {
      r = 0;
    }
  }
  return r;
}
```

$\boxed{\text{O \& !D}}$

```
int oTdF(int *a, int *b, int *t) {
  r0 = t[a[0]] != t[b[0]];
  r1 = t[b[1]] != t[b[1]];
  r2 = r0 I r1;
  r3 = r2 ? 0 : 1;
  return r3;
}
```

$\boxed{\text{O \& D}}$

```
int oTdT(int *a, int *b) {
  r0 = a[0] != b[0];
  r1 = a[1] != b[1];
  r2 = r0 I r1;
  r3 = r2 ? 0 : 1;
  return r3;
}
```

Function oFdF is neither operation invariant nor data invariant. Function oFdT is not operation invariant, but it is data invariant. Function oTdF is operation invariant, but it is not data invariant. Finally, function oTdT is both operation and data invariant.

Q1 [2 Points] If a function is data variant, then it is possible to discover secret information about it (albeit that is still difficult). For instance, consider function oTdF above. An adversary can time the execution of the program, and whenever time changes, she might know that a miss happened in the D-Cache (the Data Cache). In this way, she might know that some a[i] or some b[i] contains a value that causes the address t[a[i]] or t[b[i]] to be absent in the cache. To know more about it, check the work of Page [1] (*reading the paper is optional for the exam*). If the function is not operation invariant; however, it is much easier to infer information from it! In this question, you must explain how a brute-force timing attack can discover the value of array b in function oFdF, provided that array a is known, in linear time, instead of in exponential time.

We shall model the kind of time-based side-channels that we are mentioning in this exam through a toy assembly-like language. The semantics of this language is available as a Prolog program at:
https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/previousExams/Lif.pl
The language contains 11 different types of instructions. The operational semantics of these instructions is given in Figure 1. The predicate evalInsts(P, I, N, M, S, I', N', M', S', E) receives five inputs (in blue), and produces five outputs (in red). The inputs are an instruction P, the program input I, which is a list of constants, the index of the next block of memory that can be allocated N, a list M that represents the memory and a list S of pairs (Name, Value), where Name is some program variable, and Value is a constant that is associated with that variable. Once the predicate is evaluated, it produces, as already mentioned, five outputs. These outputs are updated copies of the list of inputs, the index of the next memory block to be allocated, the memory and the stack, plus an "effect" E. The effect is either the atom "noEffect", or an address in the memory that has been accessed by the instruction. In this case, we write d(@), where @ is the address.

```
evalInst(add(Dst, Src0, Src1), I, N, M, S, I, N, M, SS, noEffect) :-
  lookup(S, Src0, Value0),
  lookup(S, Src1, Value1),
  ValueDst is Value0 + Value1,
  SS = [(Dst, ValueDst)|S].

evalInst(alloc(Address, Size), I, N, M, S, I, NN, M, SS, noEffect) :-
  SS = [(Address, N)|S],
  NN is N + Size.

evalInst(input(Dst), [Constant|I], N, M, S, I, N, M, SS, noEffect) :-
  SS = [(Dst, Constant)|S].

evalInst(const(Dst, Constant), I, N, M, S, I, N, M, SS, noEffect) :-
  SS = [(Dst, Constant)|S].

evalInst(leq(Dst, Src0, Src1), I, N, M, S, I, N, M, SS, noEffect) :-
  lookup(S, Src0, Value0),
  lookup(S, Src1, Value1),
  Value0 =< Value1,
  SS = [(Dst, 1)|S].

evalInst(leq(Dst, Src0, Src1), I, N, M, S, I, N, M, SS, noEffect) :-
  lookup(S, Src0, Value0),
  lookup(S, Src1, Value1),
  Value0 > Value1,
  SS = [(Dst, 0)|S].

evalInst(load(Address, Dst), I, N, M, S, I, N, M, SS, d(ValueAddress)) :-
  lookup(S, Address, ValueAddress),
  get(M, ValueAddress, Value),
  SS = [(Dst, Value)|S].

evalInst(move(Dst, Src), I, N, M, S, I, N, M, SS, noEffect) :-
  lookup(S, Src, Value),
  SS = [(Dst, Value)|S].

evalInst(phi(Dst, Params), I, N, M, S, I, N, M, SS, noEffect) :-
  lookupFirst(S, Params, Value),
  SS = [(Dst, Value)|S].

evalInst(store(Address, Src), I, N, M, S, I, N, MM, S, d(ValueAddress)) :-
  lookup(S, Address, ValueAddress),
  lookup(S, Src, Value),
  set(M, ValueAddress, Value, MM).
```

Figure 1: The Semantics of individual instructions.

The rules above are written in Prolog. However, we could also write them in a more mathematical style. For instance, we have below the transcription of the rule that evaluates addition, in the small-step semantics seen in class. This rule, for add, reads the values of variables s_0 and s_1 on the stack; then it adds up these values, obtaining a new value v_D. It then erases the previous value of variable d on the stack, if any, and updates it with the new value v_D.

$$\frac{S[s_0] = v_0 \qquad S[s_1] = v_1 \qquad v_D = v_0 + v_1 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, v_D)}{\text{add}(d, s_0, s_1), I, N, M, S \rightarrow I, N, M, S', \text{noEffect}}$$

Q2 [1 Points] Write, in plain English, a description of the rule that evaluates store instructions. The rule, in the small-step style, is given below.

$$\frac{S[a] = v_a \qquad S[s] = v \qquad M' = M[v_a \mapsto v]}{store(a, s), I, N, M, S \rightarrow I, N, M', S, d(v_a)}$$

Q3 [2 Points] Now, consider the rule that implements phi-functions. The rule, in the small-step style, is given below. You need to explain in English why it works. For that, you will need to understand (in the Prolog implementation) how lookupFirst works, or rather you will have to guess how you can select the right parameter to copy into the destination of the phi-function. Figure 2 shows the implementation of lookupFirst. When producing your answer to this question, try to think about how a phi-function works. For instance, based on which criteria does the phi-function selects the right argument to copy to the destination register?

$$\frac{lookupFirst(S, [s_1, s_2, \ldots, s_n]) = v_D \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, v_D)}{phi(d, [s_1, s_2, \ldots, s_n]), I, N, M, S \rightarrow I, N, M, S', noEffect}$$

```
lookupFirst([(Var, Value)|_], VarList, Value) :-
  member(Var, VarList).
lookupFirst([(OldVar, _)|S], VarList, Value) :-
  \+ member(OldVar, VarList),
  lookupFirst(S, VarList, Value).
```

Figure 2: The Prolog implementation of lookupFirst. The predicate member(E, L) is true if E is an element within the list L. The notation \+ means negation, i.e., it is true whenever the predicate that follows cannot be true.

Q4 [3 Points] Write the rules left in Figure 1 in the small-step style. There are six instructions left: mov, load, leq, const, input and alloc. You might use, in your notation, standard mathematical operations, like set union (U). You can also find mathematical equivalents for the auxiliary functions that are used in Figure 1, e.g., lookup and get. Try to be clear about your notation. If convenient, add in plain English a description of the rule, so that it will be easier to check if your notation is sound.

The rules in Figure 1 are only part of the semantics of our toy language. Those rules, alone, cannot be used to build an interpreter for the language. The other rules necessary to build this interpreter are given in Figure 3. Figure 3 defines a predicate, evalProg(P, I, PC, N, M, S, E, N', M', S', E'). This predicate has seven inputs: a program P, a list of input values I, the address of the next instruction to be fetched PC, the index of the next block of memory that can be allocated N, the memory M, the stack S and the list of events E observed thus far. It produces four outputs: a new index for the next memory block that can be allocated, a new version of the memory M', a new version of the stack S', and a new list of observable events E'. Notice that whereas evalInst evaluates instructions that bear no influence on the program's control flow, evalProg evaluates instructions that change the program's control flow. The control flow is controlled by a program counter (the PC). Control flow might change in four ways: either the program terminates (when the instruction at PC is halt); or we execute a conditional branch; or we execute an unconditional branch, or we simply advance the PC after executing one of the other instructions. Notice that the evaluation of any instruction adds a new effect on the list of observable events E. This effect is the instruction that was just executed. Therefore, we have two kinds of effects: addresses of memory (imagine that these addresses exist in the Data Cache) and addresses of instructions (imagine that these addresses exist in the Instruction Cache).

Q5 [1 Point] Which style of operational semantics is encoded in Figure 3: big-step or small-step?

```
evalProg(P, _, PC, N, M, S, E, N, M, S, [i(PC)|E]) :-
  get(P, PC, halt).
evalProg(P, I, PC, N, M, S, E, NN, MM, SS, [i(PC)|EE]) :-
  get(P, PC, bnz(Predicate, _)),
  lookup(S, Predicate, 0),
  PCC is PC + 1,
  evalProg(P, I, PCC, N, M, S, E, NN, MM, SS, EE).
evalProg(P, I, PC, N, M, S, E, NN, MM, SS, [i(PC)|EE]) :-
  get(P, PC, bnz(Predicate, Label)),
  lookup(S, Predicate, Value),
  Value \= 0,
  evalProg(P, I, Label, N, M, S, E, NN, MM, SS, EE).
evalProg(P, I, PC, N, M, S, E, NN, MM, SS, [i(PC)|EE]) :-
  get(P, PC, jmp(L)),
  evalProg(P, I, L, N, M, S, E, NN, MM, SS, EE).
evalProg(P, I, PC, N, M, S, E, NNN, MMM, SSS, [i(PC)|EE]) :-
  get(P, PC, Inst),
  Inst \= halt, Inst \= bnz(_, _), Inst \= jmp(_),
  evalInst(Inst, I, N, M, S, II, NN, MM, SS, noEffect),
  NewPC is PC + 1,
  evalProg(P, II, NewPC, NN, MM, SS, E, NNN, MMM, SSS, EE).
evalProg(P, I, PC, N, M, S, E, NNN, MMM, SSS, [i(PC),d(A)|EE]) :-
  get(P, PC, Inst),
  Inst \= halt, Inst \= bnz(_, _), Inst \= jmp(_),
  evalInst(Inst, I, N, M, S, I, N, MM, SS, d(A)),
  NewPC is PC + 1,
  evalProg(P, I, NewPC, N, MM, SS, E, NNN, MMM, SSS, EE).
```

Figure 3: Evaluation of the whole program.

If you want to test the semantic rules, you can invoke the predicate evalProg with some input. As an example, Figure 4 shows the output produced for two different programs. One assigns variable "a" using a constant, the other reads the input and assigns variable "a" the value just read.

```
?- evalProg([const(a, 1), halt], [], 0, 0, [0, 0, 0, 0, 0, 0, 0, 0], [], [], N, M, S, E).
N = 0,
M = [0, 0, 0, 0, 0, 0, 0, 0],
S = [ (a, 1)],
E = [i(0), i(1)]

?- evalProg([input(a), halt], [1], 0, 0, [0, 0, 0, 0, 0, 0, 0, 0], [], [], N, M, S, E).
N = 0,
M = [0, 0, 0, 0, 0, 0, 0, 0],
S = [ (a, 1)],
E = [i(0), i(1)] ;
```

Figure 4: Evaluation of two different programs.

Q6 [1 Point] What would be the result of evaluating the program below with an empty list of input values?

[const(a, 1), const(b, 2), add(c, a, b), alloc(x, 1), store(x, c), halt]

You must inform in your answer only the list of events E that will be produced for this evaluation.

Q7 [2 Points] A program is said to be operation invariant if it always runs the same sequence of instructions, regardless of the input that it receives. A program that does not read inputs is trivially operation invariant. However, even programs that read inputs can be operation invariant. Below we have an example of a program that is operation invariant, even though it reads two values from the list of inputs:

[input(a), input(b), add(c, a, b), alloc(x, 1), store(x, c), halt]

Regardless of the input, this program always produces the following trace of observable events:

E = [i(0), i(1), i(2), i(3), i(4), d(0), i(5)]

Write a program that is not operation invariant, but is data invariant, and show how two different inputs produce two different lists of observable events. Important: Your program must either read or write the memory M. You must show the list of inputs **I**, and the corresponding traces E.

Obs.: you might want to take a look into Q11 before solving this question, for the program that you will produce here will bear consequence on your solution to that question.

Q8 [2 Points] A program is said to be data invariant if it always accesses the same sequence of addresses in the memory M, regardless of the input that it receives. As in Q7, a program that does not read inputs is trivially data invariant. However, even programs that read inputs can be data invariant (similarly to operation invariance in Q7). The program that we had seen before, in Q7, is data invariant.

Write a program that is not data invariant, and show how two different inputs produce two different lists of observable events. Notice that you can either have this program accessing memory a different number of times, or accessing different addresses in the memory array. You must show two inputs, and the corresponding traces of observable events for each of these inputs.

Q9 [2 Points] Define a new instruction "or", and add it to your language. You will have to update the predicate evalInst to accommodate the new instruction. Figure 5 shows part of this predicate, with the signature of the or instruction.

```
evalInst(or(Dst, Src0, Src1), I, N, M, S, I, N, M, SS, noEffect) :- ??
```

Figure 5: Part of the evalInst predicate that evaluates the or instruction. You will have to complete this predicate. Notice that you might have to implement multiple predicates to ensure that your evaluation of the or instruction is deterministic.

The small step evaluation of the or instruction is given by the rules below. They might guide you when implementing your predicate, but you are free to decide how to implement the semantics. Notice that when implementing the rules in Prolog, you might prefer to use four predicates, instead of three (the full truth-table of logical OR), so that the evaluation of the new instruction will always have a single answer.

$$\frac{S[s_0] = 0 \qquad S[s_1] = 0 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, 0)}{or(d, s_0, s_1), \mathbf{I}, N, M, S \rightarrow \mathbf{I}, N, M, S', noEffect}$$

$$\frac{S[s_0] = v \qquad v \mathrel{!=} 0 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, 1)}{or(d, s_0, s_1), \mathbf{I}, N, M, S \rightarrow \mathbf{I}, N, M, S', noEffect}$$

$$\frac{S[s_1] = v \qquad v \mathrel{!=} 0 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, 1)}{or(d, s_0, s_1), \mathbf{I}, N, M, S \rightarrow \mathbf{I}, N, M, S', noEffect}$$

Q10 [2 Points] Define a new instruction "ctsel", and add it to your language. You will have to update the predicate evalInst to accommodate the new instruction. Figure 6 shows part of this predicate, with the signature of the ctsel instruction.

```
evalInst(ctsel(Dst, _, Src1, Cond), I, N, M, S, I, N, M, SS, noEffect) :- ??
```

Figure 6: Part of the evalInst predicate that evaluates the ctsel instruction. You will have to complete this predicate. Notice that you might have to define multiple predicates to evaluate the instruction.

The small step evaluation of the ctsel instruction is given by the rules below. This new instruction works a bit like the ternary operator of C, e.g., Dst = Cond ? Src1 : Src0.

$$\frac{S[c] = 0 \qquad S[s_0] = v_0 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, v_0)}{\text{ctsel}(d, s_0, \_, c), I, N, M, S \rightarrow I, N, M, S', \text{noEffect}}$$

$$\frac{S[c] = v \qquad v \mathrel{!=} 0 \qquad S[s_1] = v_1 \qquad S' = (S \setminus \{(d, ?)\}) \cup (d, v_1)}{\text{ctsel}(d, \_, s_1, c), I, N, M, S \rightarrow I, N, M, S', \text{noEffect}}$$

Q11 [2 Points] Use the two new instructions that you have designed, namely, or and ctsel, plus the instructions that are already part of our toy language, to rewrite the program that you have proposed in Q7, so that this program becomes operation invariant.

Let P7 be the program that you have written in Q7, and let P11 be the program that you have derived from P7 to solve this question. The following properties must hold:

1. P11 is operation invariant.
2. P11 is data invariant (notice that P7 is data invariant, as per the specification).
3. If: evalProg(P7, I, 0, 0, [0, 0, 0, 0, 0, 0, 0, 0], [], [], N, M, S7, E7) is true;
   then evalProg(P11, I, 0, 0, [0, 0, 0, 0, 0, 0, 0, 0], [], [], N, M, S11, E11) is also true.

Notice that Property 3 implies that the evaluation of P7 and P11 with the same empty inputs (PC = 0, N = 0, M = [0…0], S = [], E = []) will always produce the same index N, and the same memory array M. The stack might not be the same. Also, the trace of observable events might differ (E7 != E11).

Notice that this kind of transformation can be mechanized, in order to be performed automatically by a compiler. For more details, we refer the interested student to the work of Soares and Pereira [2].

References

[1] Theoretical use of cache memory as a cryptanalytic side-channel. D. Page, IACR, 2002

[2] Memory-Safe Elimination of Side Channels. Luigi Soares and Fernando Magno Quintão Pereira. CGO 2021

Software

You can try online the implementation of an analysis and code transformation for the LLVM compiler that makes programs isochronous. The implementation is available at: `http://cuda.dcc.ufmg.br/lif/`