# CS5218 Assignment1 – Taint Analysis Design

Ma Yuan E0674520

February 26, 2021

## 1 Task 1: Designing Taint Analysis

According to the description of Taint Checking, the data flow analysis can be defined as follow:

1. For each program basic block, logging the assignment have been made and yet not been overwritten when program execution reaches the basic block along some path.
2. When program analysis reach a fixed point, the tainted variable is observed in the least upper bound, we say that the program is possible to be exposed to a potentially dangerous tainted variable.
3. This data flow analysis can be performed by a *Reach Definition analysis*.

Therefore base on the data flow equations for Reach Definition, we can modify it to perform taint variables analysis as such:

- $L = \mathcal{P}(Var_* \times Lab_*)$ (Example powerset lattice map:$(\mathcal{P}(Var_* \times Lab_*)^A, \sqsubseteq)$ Figure 1)
- Analysis Type: May, Forward
- Data Flow Equations:

$$TaintRD_{exit}(\ell) \ = \ (TaintRD_{entry}(\ell) \setminus kill_{TaintRD}(B^\ell)) \cup gen_{TaintRD}(B^\ell), \ where B^\ell \in blocks(S_F) \tag{1}$$

$$TaintRD_{entry}(\ell) \ = \ \begin{cases} \{(x, ?) \mid x \in FV(S_F)\} \ \ if \ell = init(S_F) \\ \cup \{TaintRD_{exit}(\ell') | (\ell', \ell) \in flow(S_F)\} \ \ otherwise) \end{cases} \tag{2}$$

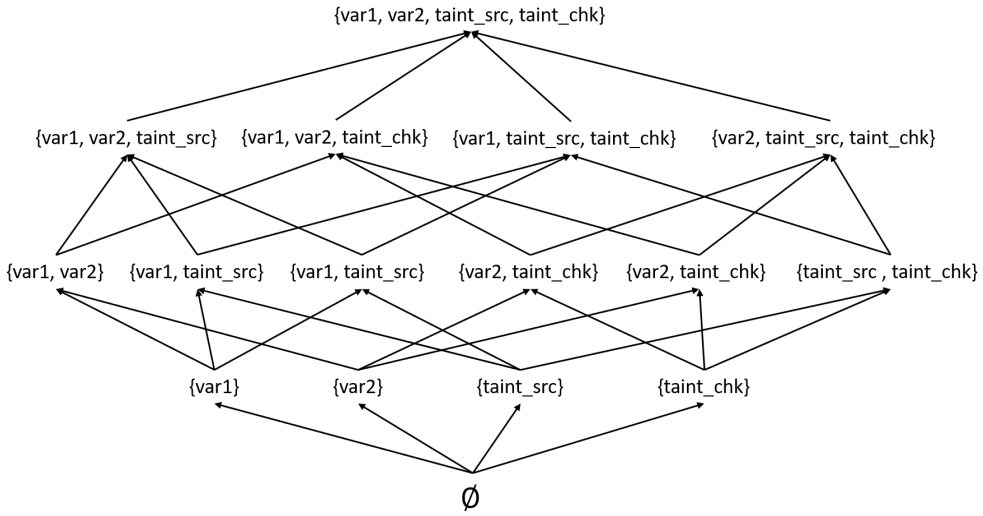


Figure 1: Example of powerset lattice map for taint variables analysis.

where $kill_{TaintRD}(B^\ell)$ and $gen_{TaintRD}(B^\ell)$ are formulated as:

$$kill_{TaintRD}([z := a]^\ell) \;=\; \begin{cases} \{(z,?)\} \cup \{(z,\ell)\}, \;\; if\ B^{\ell'}\ is\ an\ assignment\ to\ z\ in\ S_F \\ \emptyset \;\; otherwise) \end{cases} \tag{3}$$

$$gen_{TaintRD}([z := a]^\ell) \;=\; \{(z,\ell)\} \tag{4}$$

- $\sqcup : \cup$
- $\perp : \emptyset$
- $\iota : \{(x,?) \mid x \in FV(S_F)\}$
- $E : \{init(S_*)\}$
- $F : flow(S_*)$
- Transfer function:

$$\mathcal{F} = \{f : L \to L \mid \exists l_k, l_g : f(l) = (l \;\setminus\; l_k) \cup l_g\} \tag{5}$$

$$f_\ell \;=\; (l \;\setminus\; kill(B^\ell) \;\cup\; gen(B^\ell) \;\; where\ B^\ell \;\in\; blocks(S_*)) \tag{6}$$

In order to find the Least fixed point of F, here the Chaotic Iteration for Reaching Definition algorithm in the lecture is used to perform analysis. Base on the properties of the Chaotic Iteration, the analysis program always terminates:

**Algorithm 1:** Chaotic Iteration for Reaching Definition

**input :** $F(RD) = (F_1(RD), ..., F_A(RD))$
**output:** $RD = (RD_1, ..., RD_A)$
Initialization: $RD_1 := \emptyset, ..., RD_A := \emptyset$;
**while** $RD_j \neq F_j(RD_1, ..., RD_A)$ *for some* $j$ **do**
  $RD_j := F_j(RD_1, ..., RD_A)$
**end**

# 2 Task 2: Simple Taint Analysis Implementation

## 2.1 TaintChk.cpp implementation

From the first example, I observed example1.ll(Appendix: Figure 2) has following 4 key instructions [1]:

- *alloca*: The 'alloca' instruction allocates memory on the current stack frame of the procedure that is live until the current function returns to its caller. (Initialization)
- *store*: The 'store' instruction is used to write to memory.(specifies a value to store and an address to store it into)
- *load*: The 'load' instruction is used to read from memory.(specifies the memory address to load from)
- *icmp*: The 'icmp' instruction perform comparison.

According to the definition of the instructions [1], we can implement the Taint checking algorithm as following:

**Algorithm 2:** Simple algorithm for Taint Variable Analyzer

$Init\ lfp_{map} = \{\$string\ block\_name : \$struct < string, set < string >> \ blk_{map}\};$
$Init\ \$set < string > \ Last\_load\_var\_\{var\}\};$
$Init\ \$set < string > \ Alloca\_var\_\{var\}\};$
**for all** $block \in test.ll$ **do**
  $blk_{map} = \{\$string\ var_{store\_Var} : \$set < string > \ var_{load\_Var}\};$
  **for all** $Instr \in Block$ **do**
    **if** $Instr =$ "$AllocaInst$" **then**
      $Alloca\_var \leftarrow Instr.getName()$
      **if** $allocate_vars =$ "$source$" **then**
        $lfp_{map}[block.name].insert(\text{"}source\text{"})$
      **end if**
    **end if**
    **if** $Instr =$ "$store$" **then**
      $Last\_load\_var \leftarrow Instr.getOperand(0).name$
    **end if**
    **if** $Instr =$ "$store$" **then**
      $blk_{map}[Instr.getOperand(1).name].remove(items)$
      $blk_{map}[Instr.getOperand(1).name] \leftarrow \ Last\_load\_var$
      $clear(Last\_load\_var\_var)$
    **end if**
    **if** $Instr =$ "$icmp$" **then**
      $clear(Last\_load\_var\_var)$
    **end if**
  **end for**
  $lfp_{map}[block.name].insert(blk_{map})$
**end for**
**output:** $lfp_{map}$

# 3 Task 3: Loop Handling Implementation

As mentioned in section1, the Chaotic Iteration algorithm is implemented to perform taint variable analysis. In order to terminates the while loop, we need to check if $RD = F(RD_1, ..., RD_A)$. To implement this, we just need to create a loop to explore the blocks map from algorithm1

follow the pre-defined sequence and for each iteration we maintain the taint variables in block and continue to update it, until the predecessor block contains the same information as current analyzed block(if program has a fixed point).

**Algorithm 3:** Chaotic Iteration for Reaching Definition

**input :** $lfp_{map}$ *from Algorithm*1
**input :** $pre-defined\ blockOrder$
**output:** $result\_lfp_{map}$
Initialization: $lfp_indicator$; $result\_lfp_{map}$
  **while** true **do**
    $lfp_{indicator} := 0$
    **for all** $block \in blockOrder$ **do**
      $lfp_{map}[block].insert(result\_lfp_{map}[block.predecessor])$
      $prev\_lfp_{map} = lfp_{map}.copy()$
      **for all** $blk_{map} \in lfp_{map}[block]$ **do**
        $< string > var_{store\_Var} = blk_{map}.store\_Var$
        $set < string > var_{load\_Var} = blk_{map}.load\_Var$
        **if** $result\_lfp_{map}[block][var_{load\_Var}] != result\_lfp_{map}[block].end()[var_{load\_Var}]$
        **then**
          $result\_lfp_{map}[block].update(result\_lfp_{map}[block])$
        **end if**
      **end for**
      **if** $prev\_lfp_{map}[block] == lfp_{map}[block]$ **then**
        *break*
      **end if**
    **end for**
  **end while**
  **end**

# References

[1] Llvm language reference manual. https://releases.llvm.org/2.6/docs/LangRef.html.

example1.ll

```
; ModuleID = 'example1.c'
source_filename = "example1.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %sink = alloca i32, align 4
  %source = alloca i32, align 4
  store i32 0, i32* %1
  %2 = load i32* %source, align 4
  store i32 %2, i32* %b, align 4
  %3 = load i32* %a, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5                                       ; preds = %0
  br label %8

; <label>:6                                       ; preds = %0
  %7 = load i32* %b, align 4
  store i32 %7, i32* %c, align 4
  br label %8

; <label>:8                                       ; preds = %6, %5
  %9 = load i32* %c, align 4
  store i32 %9, i32* %sink, align 4
  %10 = load i32* %1
  ret i32 %10
}

attributes #0 = { nounwind uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false"
"no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false"
"no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.9.1-19ubuntu1 (tags/RELEASE_391/rc2)"}
```

Figure 2: example1.ll.

# Appendix