Data-Flow and Taint Analysis

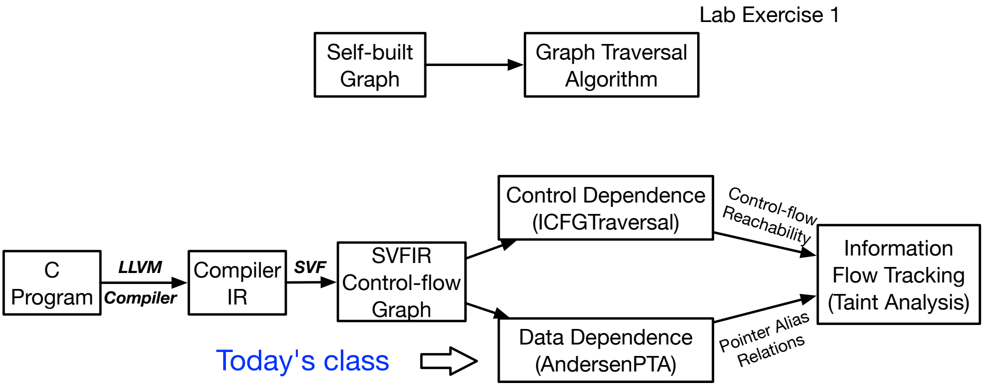
(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today’s class



Revisiting Andersen’s Analysis

Data-Flow and Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.

Data-Flow and Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with “%”) and **global** variables (symbols starting with “@”) are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available from LLVM’s SSA form**.
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store ptr %a1, ptr %a, align 8`

Data-Flow and Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available from LLVM's SSA form.**
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store ptr %a1, ptr %a, align 8`
- **Address-taken variables** (abstract objects), accessed indirectly at load or store instructions via top-level variables (`ObjPN` in SVF)
  - A **stack object** created at an LLVM's 'alloca' instruction or a **heap object** created via (e.g., 'malloc' callsite) or a **global object**.

Data-Flow and Data-Dependence

Definition-use relations between variables. Two types of variables on LLVM IR:

- **Top-level variables**, whose addresses are not taken (`ValPN` in SVF)
  - Including stack virtual **registers** (symbols starting with "%") and **global** variables (symbols starting with "@") are explicit, i.e., directly accessed.
  - **Def-use for top-level variables are directly available from LLVM's SSA form.**
  - For example, def-use for %a1 from Instruction-1 to Instruction-2.
    - Instruction-1: `%a1 = alloca i8, align 1;`
    - Instruction-2: `store ptr %a1, ptr %a, align 8`
- **Address-taken variables** (abstract objects), accessed indirectly at load or store instructions via top-level variables (`ObjPN` in SVF)
  - A **stack object** created at an LLVM's 'alloca' instruction or a **heap object** created via (e.g., 'malloc' callsite) or a **global object**.
  - **Def-use for address-taken variables are computed via pointer analysis**.
  - For example, there is a def-use for object `o` from Instruction-1 to Instruction-2 if pointers %a and %b both point to `o`.
    - Instruction-1: `store ptr %a1, ptr %a, align 8`
    - Instruction-2: `%c = load ptr %b, align 8`

Pointer Analysis (Revisit Andersen's Analysis in Lab-Exercise-1)

A typical data-flow analysis

- **Points-to Analysis**: aims to statically determine the possible runtime values of a pointer at compile-time.
  - Compute the *points-to set* (**a set of address-taken variables**) of each *pointer* (**top-level variable**)
  - For example, `p = &a; q = p;`
  - The resulting points-to sets of `p` and `q` are: `pts(p) = pts(q) = {a}`

Pointer Analysis (Revisit Andersen's Analysis in Lab-Exercise-1)

A typical data-flow analysis

- **Points-to Analysis**: aims to statically determine the possible runtime values of a pointer at compile-time.
  - Compute the *points-to set* (**a set of address-taken variables**) of each *pointer* (**top-level variable**)
  - For example, `p = &a; q = p;`
  - The resulting points-to sets of p and q are: $\mathtt{pts(p)} = \mathtt{pts(q)} = \{\mathtt{a}\}$
- **Alias Analysis**: determines whether two pointer dereferences refer to the same memory location.
  - If the points-to sets of two pointers p and q have overlapping elements (i.e., $\mathtt{pts(p)} \cap \mathtt{pts(q)} \neq \emptyset$) then p and q are aliases. The derereferences of p and q may refer to the same memory location.

Pointer Analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).

Pointer Analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses

Pointer Analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both always refer to `a`.

Pointer Analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both always refer to `a`.
- Compiler optimizations and bug detection
  - Constant propagation
    - `*p = 1; x = *q;`
      `x` is a constant value and equals 1, if `p` and `q` are must-aliases (always point to the same memory location w.r.t every execution path).
    - `*p = 1; *q = r; x = *p;`
      `x` is a constant value and equals 1, if `p` and `q` do not alias with each other.

Pointer Analysis

Why shall we learn pointer analysis?

- Essential for building data-dependence relations between variables (memory objects).
- Understanding aliases through different memory accesses
  - `p = &a; q = p; *p = x; y = *q;`
    `y` has the same value as `x` since `*p` and `*q` both always refer to `a`.
- Compiler optimizations and bug detection
  - Constant propagation
    - `*p = 1; x = *q;`
      `x` is a constant value and equals 1, if `p` and `q` are must-aliases (always point to the same memory location w.r.t every execution path).
    - `*p = 1; *q = r; x = *p;`
      `x` is a constant value and equals 1, if `p` and `q` do not alias with each other.
  - Taint analysis
    - `*p = taintedInput; x = *q;`
      `x` is tainted if `p` and `q` are aliases.

Precision Dimensions

Can be generally classified into the following precision dimensions at different levels of abstractions.

Flow-insensitive analysis:

- Ignores program execution order
- A single solution across whole program

Context-insensitive analysis:

- Merges all calling contexts when analysing a program method

Path-insensitive analysis:

- Merges all incoming path information at the join points of the control-flow graph

Flow-sensitive analysis:

- Respects the program execution order
- Separate solution at each program point

Context-sensitive analysis:

- Distinguishes between different calling contexts of a program method

Path-sensitive analysis:

- Computes a solution per (abstract) program path.

Precision Dimensions

Levels of Abstractions

Assume x is a tainted value

p = x

p = y

flow-sensitivity

at which

program point

p is tainted?

```
foo(x)      foo(y)
   call \   / call
     foo(p){

     }
```

context-sensitivity

under which

calling context

p is tainted?

```
if(cond)
    p = x
else
    p = y
```

path-sensitivity

along which

program path

p is tainted?

Andersen’s Pointer Analysis

A **flow-insensitive, context-insensitive and path-insensitive points-to analysis** to determine points-to set of a pointer by analyzing the **Constraint Graph** or **Program Assignment Graph (PAG)** of a program.

Andersen's Pointer Analysis

A **flow-insensitive, context-insensitive and path-insensitive points-to analysis** to determine points-to set of a pointer by analyzing the **Constraint Graph** or **Program Assignment Graph (PAG)** of a program.

- Also known as **inclusion-based points-to analysis**, the most popular and widely used pointer analysis.
- Solving constraint edges between `ConstraintNodes` (`SVFVars`, which are either pointer types or objects).
- The analysis requires iterative solving of the `ConstraintGraph` by (1) propagating points-to sets among graph nodes, and (2) adding new edges until a fixed point is reached, i.e., no new edges are added and no points-to sets change. (**Lab-Exercise-1**)

Andersen, L. O. (1994). Program analysis and specialization for the C programming language (Doctoral dissertation, University of Cophenhagen).

The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code, PLDI 2007

Field-Sensitive Andersen's Pointer Analysis

The analysis operating upon a program's constraint graph which is a subgraph of PAG (program assignment graph).

- `ConstraintNode` represents
  - A pointer (`ValVar`): (top-level variable) or
  - An object (`ObjVar`): (address-taken objects, i.e., heap/stack/global/function objs)
- `ConstraintEdge` represents a constraint between two nodes

Field-Sensitive Andersen's Pointer Analysis

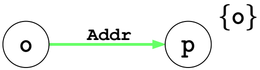
The analysis operating upon a program's constraint graph which is a subgraph of PAG (program assignment graph).

- `ConstraintNode` represents
  - A pointer (`ValVar`): (top-level variable) or
  - An object (`ObjVar`): (address-taken objects, i.e., heap/stack/global/function objs)
- `ConstraintEdge` represents a constraint between two nodes

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	`%p = alloca`$_o$	$p = \&o$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	`%q = bitcast %p`	$q = p$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
$q \xleftarrow{\texttt{Load}} p$	`%q = load %p`	$q = *p$	$\forall o \in \texttt{pts}(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	`store %q, %p`	$*p = q$	$\forall o \in \texttt{pts}(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	`%q = gep %p, fld`	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(q) \cup \{o.\texttt{fld}\}$

Field-Sensitive Andersen's Pointer Analysis

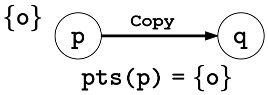
Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	$\%p = \texttt{alloca}_o$	$p = \&o$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	$\%q = \texttt{bitcast}\ \%p$	$q = p$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
$q \xleftarrow{\texttt{Load}} p$	$\%q = \texttt{load}\ \%p$	$q = *p$	$\forall o \in \texttt{pts}(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	$\texttt{store}\ \%q, \%p$	$*p = q$	$\forall o \in \texttt{pts}(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	$\%q = \texttt{gep}\ \%p, \texttt{fld}$	$q = \&p \to \texttt{fld}$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(q) \cup \{o.\texttt{fld}\}$



$\texttt{pts(p)} = \{o\}$

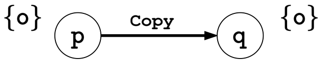
Field-Sensitive Andersen's Pointer Analysis

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	`%p = alloca`$_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	`%q = bitcast %p`	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	`%q = load %p`	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	`store %q, %p`	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	`%q = gep %p, fld`	$q = \&p \rightarrow fld$	$\forall o \in pts(p) : pts(q) = pts(q) \cup \{o.fld\}$



Field-Sensitive Andersen's Pointer Analysis

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	$\%p = \texttt{alloca}_o$	$p = \&o$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	$\%q = \texttt{bitcast}\ \%p$	$q = p$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
$q \xleftarrow{\texttt{Load}} p$	$\%q = \texttt{load}\ \%p$	$q = *p$	$\forall o \in \texttt{pts}(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	$\texttt{store}\ \%q, \%p$	$*p = q$	$\forall o \in \texttt{pts}(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	$\%q = \texttt{gep}\ \%p, \texttt{fld}$	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(q) \cup \{o.\texttt{fld}\}$

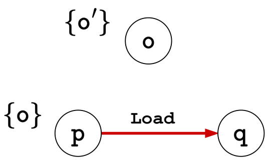


pts(p) = {o}

pts(q) = {o}

Field-Sensitive Andersen's Pointer Analysis

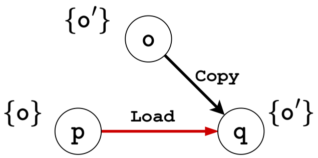
Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	%p = alloca$_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	%q = bitcast %p	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	%q = load %p	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	store %q, %p	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	%q = gep %p, fld	$q = \&p \rightarrow fld$	$\forall o \in pts(p) : pts(q) = pts(q) \cup \{o.fld\}$



pts(p) = {o}
pts(o) = {o′}

Field-Sensitive Andersen's Pointer Analysis

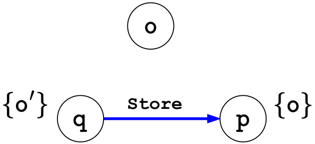
Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	`%p = alloca`$_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	`%q = bitcast %p`	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	`%q = load %p`	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	`store %q, %p`	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	`%q = gep %p, fld`	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in pts(p)$: $pts(q) = pts(q) \cup \{o.\texttt{fld}\}$



$pts(p) = \{o\}$
$pts(o) = \{o'\}$
$pts(q) = \{o'\}$

Field-Sensitive Andersen's Pointer Analysis

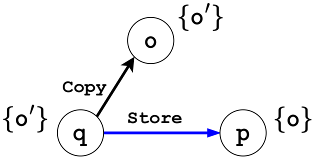
Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	`%p = alloca`$_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	`%q = bitcast %p`	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	`%q = load %p`	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	`store %q, %p`	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	`%q = gep %p, fld`	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in pts(p) : pts(q) = pts(q) \cup \{o.\texttt{fld}\}$



$pts(q) = \{o'\}$
$pts(p) = \{o\}$

Field-Sensitive Andersen's Pointer Analysis

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	%p = alloca$_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	%q = bitcast %p	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	%q = load %p	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	store %q, %p	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	%q = gep %p, fld	$q = \&p \rightarrow fld$	$\forall o \in pts(p) : pts(q) = pts(q) \cup \{o.fld\}$



$pts(q) = \{o'\}$
$pts(p) = \{o\}$
$pts(o) = \{o'\}$

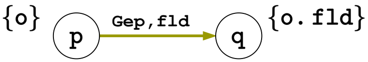
Field-Sensitive Andersen's Pointer Analysis

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	$\%p = \texttt{alloca}_o$	$p = \&o$	$pts(p) = pts(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	$\%q = \texttt{bitcast}\ \%p$	$q = p$	$pts(q) = pts(q) \cup pts(p)$
$q \xleftarrow{\texttt{Load}} p$	$\%q = \texttt{load}\ \%p$	$q = *p$	$\forall o \in pts(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	$\texttt{store}\ \%q, \%p$	$*p = q$	$\forall o \in pts(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	$\%q = \texttt{gep}\ \%p, \texttt{fld}$	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in pts(p) : pts(q) = pts(q) \cup \{o.\texttt{fld}\}$

pts(p) = {o}

Field-Sensitive Andersen's Pointer Analysis

Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\texttt{Addr}} o$	`%p = alloca`$_o$	$p = \&o$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
$q \xleftarrow{\texttt{Copy}} p$	`%q = bitcast %p`	$q = p$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
$q \xleftarrow{\texttt{Load}} p$	`%q = load %p`	$q = *p$	$\forall o \in \texttt{pts}(p)$: add edge $q \xleftarrow{\texttt{Copy}} o$
$p \xleftarrow{\texttt{Store}} q$	`store %q, %p`	$*p = q$	$\forall o \in \texttt{pts}(p)$: add edge $o \xleftarrow{\texttt{Copy}} q$
$q \xleftarrow{\texttt{Gep,fld}} p$	`%q = gep %p, fld`	$q = \&p \rightarrow \texttt{fld}$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(q) \cup \{o.\texttt{fld}\}$



pts(p) = {o}

pts(q) = {o. fld}

Constraint Solving Algorithm for Andersen's Analysis

- A `worklist` holds a list of constraint graph nodes for iterative processing
  - Initialize the points-to set of the destination node of each `address` edge. Initialize the `worklist` with nodes that have incoming address `edges`.
  - Pop a node `p` from the `worklist`.
  - Handle each incoming `store` edge and each outgoing `load` edge of node `p` by adding `copy` edges.
  - Handle each outgoing `copy` edge of `p` by propagating points-to information.
  - A node is pushed into the `worklist` if (1) its points-to set changes or (2) it is a source node of a new copy edge added to the graph.
- Any new `copy` edge added needs to be resolved and performs points-to propagation. New points-to sets discovered may trigger introducing new `copy` edges via `load` and `store` edges. The constraint solving should converge to a fixed point, where no new edges are added, and no points-to sets change.

Compiling a C Program to Its LLVM IR

```c
void swap(char **p, char **q){
  char* t = *p;
  *p = *q;
  *q = t;
}
int main(){
  char a1;
  char *a;
  char b1;
  char *b;
  a = &a1;
  b = &b1;
  swap(&a,&b);
}
```

swap.c

Compile ⇒

```llvm
define void @swap(ptr %p, ptr %q) #0 {
entry:
  %0 = load ptr, ptr %p, align 8
  %1 = load ptr, ptr %q, align 8
  store ptr %1, ptr %p, align 8
  store ptr %0, ptr %q, align 8
  ret void
}

define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %a = alloca ptr, align 8
  %b1 = alloca i8, align 1
  %b = alloca ptr, align 8
  store ptr %a1, ptr %a, align 8
  store ptr %b1, ptr %b, align 8
  call void @swap(ptr %a, ptr %b)
  ret i32 0
}
```

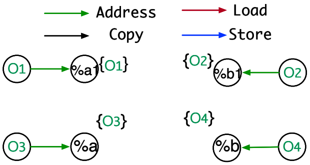
swap.ll

*https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR#2-llvm-ir-generation

Construct the Constraint Graph from LLVM IR

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```

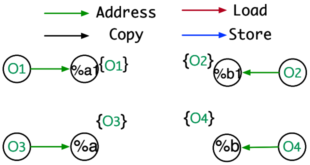


https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Construct the Constraint Graph from LLVM IR

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1        // O1
    %a = alloca ptr, align 8        // O2
    %b1 = alloca i8, align 1        // O3
    %b = alloca ptr, align 8        // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```

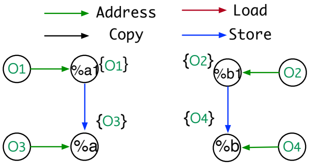


*https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Construct the Constraint Graph from LLVM IR

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```

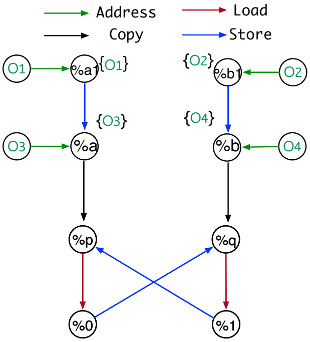


https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Construct the Constraint Graph from LLVM IR

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 1: 1 Andersen's Pointer Analysis

Input : $G = \langle V, E \rangle$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

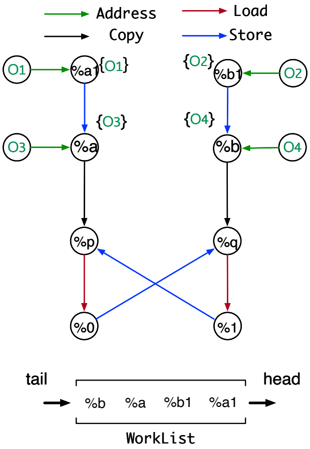
```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                        // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do               // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};           // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};           // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                    // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                 // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Andersen’s Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 2: 1 Andersen’s Pointer Analysis

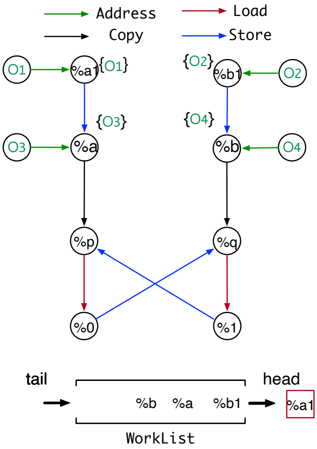
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                 // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};             // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                  // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};             // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                      // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                   // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



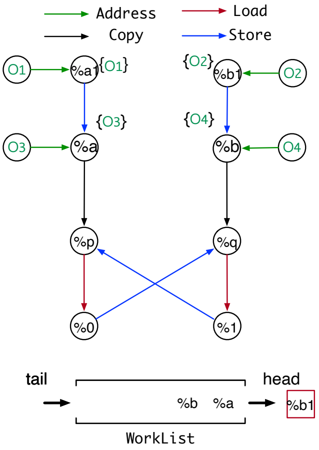
Algorithm 3: 1 Andersen's Pointer Analysis

```
Input : G =< V, E >: Constraint Graph
        V: a set of nodes in graph
        E: a set of edges in graph
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                 // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};             // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                  // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};             // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                      // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep.fld--> x ∈ E do                   // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
   %a1 = alloca i8, align 1          // O1
   %a = alloca ptr, align 8          // O2
   %b1 = alloca i8, align 1          // O3
   %b = alloca ptr, align 8          // O4
   store ptr %a1, ptr %a, align 8
   store ptr %b1, ptr %b, align 8
   call void @swap(ptr %a, ptr %b)
   ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
   %0 = load ptr, ptr %p, align 8
   %1 = load ptr, ptr %q, align 8
   store ptr %1, ptr %p, align 8
   store ptr %0, ptr %q, align 8
   ret void
}
```



Algorithm 4: 1 Andersen's Pointer Analysis

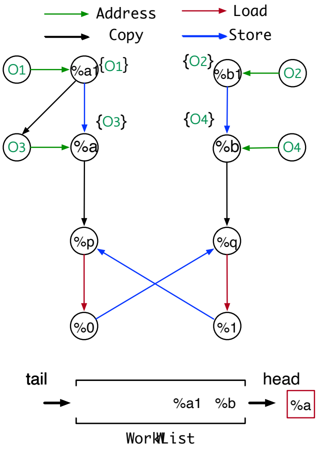
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
   %a1 = alloca i8, align 1          // O1
   %a = alloca ptr, align 8          // O2
   %b1 = alloca i8, align 1          // O3
   %b = alloca ptr, align 8          // O4
   store ptr %a1, ptr %a, align 8
   store ptr %b1, ptr %b, align 8
   call void @swap(ptr %a, ptr %b)
   ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
   %0 = load ptr, ptr %p, align 8
   %1 = load ptr, ptr %q, align 8
   store ptr %1, ptr %p, align 8
   store ptr %0, ptr %q, align 8
   ret void
}
```



Algorithm 5: 1 Andersen's Pointer Analysis

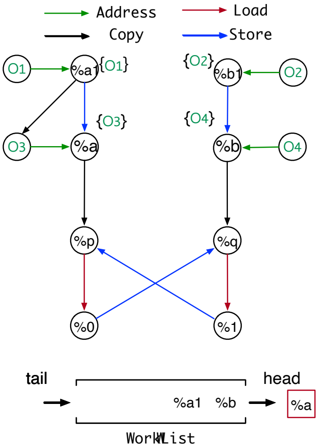
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList ≠ ∅ **do**
  p := popFromWorklist();
  foreach o ∈ pts(p) **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    pts(x) := pts(x) ∪ pts(p);
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach o ∈ pts(p) **do**
      pts(x) := pts(x) ∪ {o.fld};
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
   %a1 = alloca i8, align 1              // O1
   %a = alloca ptr, align 8              // O2
   %b1 = alloca i8, align 1              // O3
   %b = alloca ptr, align 8              // O4
   store ptr %a1, ptr %a, align 8
   store ptr %b1, ptr %b, align 8
   call void @swap(ptr %a, ptr %b)
   ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
   %0 = load ptr, ptr %p, align 8
   %1 = load ptr, ptr %q, align 8
   store ptr %1, ptr %p, align 8
   store ptr %0, ptr %q, align 8
   ret void
}
```



Algorithm 6: 1 Andersen's Pointer Analysis

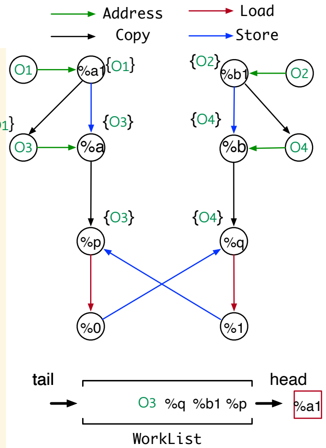
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
   %a1 = alloca i8, align 1          // O1
   %a = alloca ptr, align 8          // O2
   %b1 = alloca i8, align 1          // O3
   %b = alloca ptr, align 8          // O4
   store ptr %a1, ptr %a, align 8
   store ptr %b1, ptr %b, align 8
   call void @swap(ptr %a, ptr %b)
   ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
   %0 = load ptr, ptr %p, align 8
   %1 = load ptr, ptr %q, align 8
   store ptr %1, ptr %p, align 8
   store ptr %0, ptr %q, align 8
   ret void
}
```



Algorithm 7: 1 Andersen's Pointer Analysis

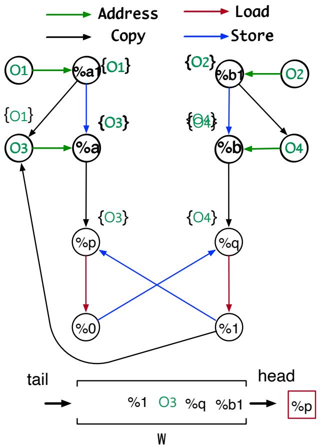
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep.fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 8: 1 Andersen's Pointer Analysis

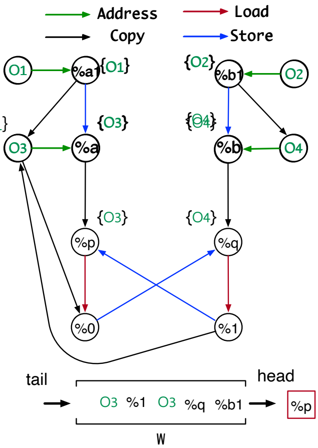
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 9: 1 Andersen's Pointer Analysis

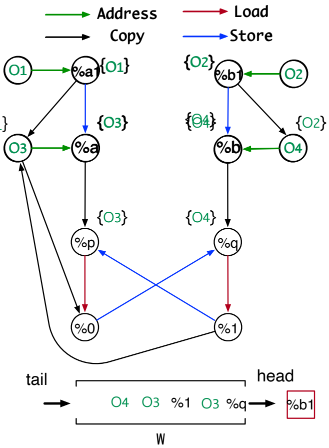
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                      // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do              // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};          // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do               // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};          // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                   // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 10: 1 Andersen's Pointer Analysis

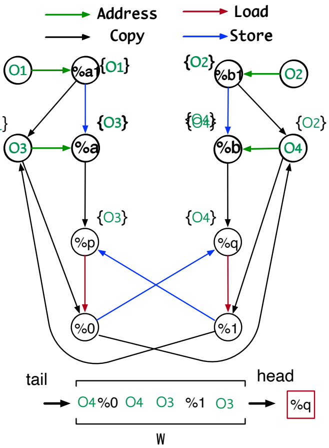
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList ≠ ∅ **do**
  p := popFromWorklist();
  foreach o ∈ pts(p) **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    pts(x) := pts(x) ∪ pts(p);
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach o ∈ pts(p) **do**
      pts(x) := pts(x) ∪ {o.fld};
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen’s Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 11: 1 Andersen’s Pointer Analysis

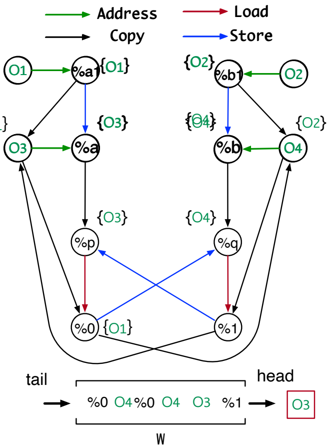
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do               // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};           // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};           // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                    // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                 // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 12: 1 Andersen's Pointer Analysis

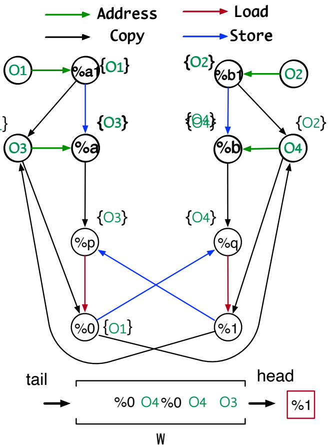
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{Address} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList $\neq \varnothing$ **do**
  p := popFromWorklist();
  foreach $o \in pts(p)$ **do**
    foreach $q \xrightarrow{Store} p \in E$ **do** // Store rule
      if $q \xrightarrow{Copy} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{Load} r \in E$ **do** // Load rule
      if $o \xrightarrow{Copy} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{Copy} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{Gep,fld} x \in E$ **do** // Gep rule
    foreach $o \in pts(p)$ **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen’s Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



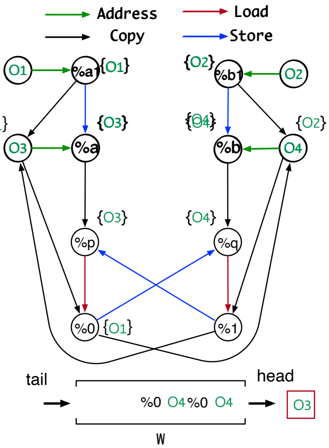
Algorithm 13: 1 Andersen’s Pointer Analysis

```
Input : G =< V, E >: Constraint Graph
        V: a set of nodes in graph
        E: a set of edges in graph
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1        // O1
    %a = alloca ptr, align 8        // O2
    %b1 = alloca i8, align 1        // O3
    %b = alloca ptr, align 8        // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 14: 1 Andersen's Pointer Analysis

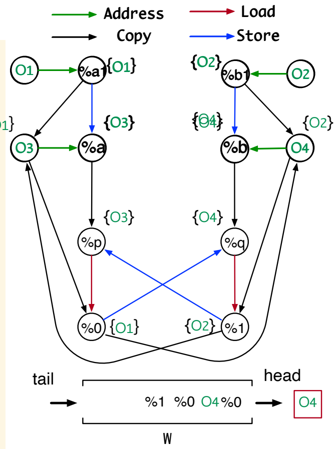
Input : $G = <V, E>$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{Address} p$ **do** // Address rule
  pts(p) := pts(p) $\cup$ {o};
  pushIntoWorklist(p);
while WorkList $\neq \emptyset$ **do**
  p := popFromWorklist();
  foreach $o \in$ pts(p) **do**
    foreach $q \xrightarrow{Store} p \in E$ **do** // Store rule
      if $q \xrightarrow{Copy} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{Load} r \in E$ **do** // Load rule
      if $o \xrightarrow{Copy} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{Copy} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{Gep,fld} x \in E$ **do** // Gep rule
    foreach $o \in$ pts(p) **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 15: 1 Andersen's Pointer Analysis

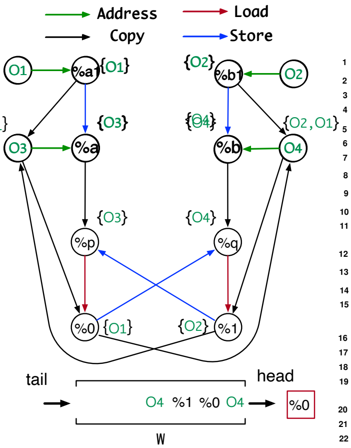
Input : $G = < V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList $\neq \varnothing$ **do**
  p := popFromWorklist();
  foreach $o \in$ pts(p) **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    pts(x) := pts(x) ∪ pts(p);
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach $o \in$ pts(p) **do**
      pts(x) := pts(x) ∪ {o.fld};
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1        // O1
    %a = alloca ptr, align 8        // O2
    %b1 = alloca i8, align 1        // O3
    %b = alloca ptr, align 8        // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 16: 1 Andersen's Pointer Analysis

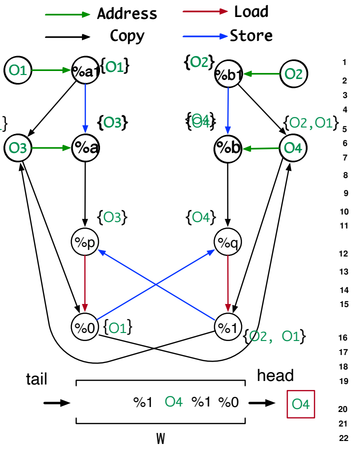
Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList $\neq \varnothing$ **do**
  p := popFromWorklist();
  foreach $o \in pts(p)$ **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach $o \in pts(p)$ **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1        // O1
    %a = alloca ptr, align 8        // O2
    %b1 = alloca i8, align 1        // O3
    %b = alloca ptr, align 8        // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



Algorithm 17: 1 Andersen's Pointer Analysis

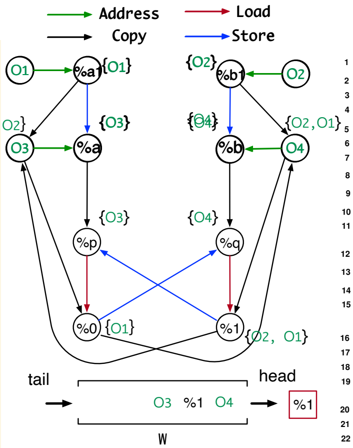
Input : $G = < V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList ≠ ∅ **do**
  p := popFromWorklist();
  foreach o ∈ pts(p) **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach o ∈ pts(p) **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}

define void @swap(ptr %p, ptr %q) #0 {
entry:
    %0 = load ptr, ptr %p, align 8
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void
}
```



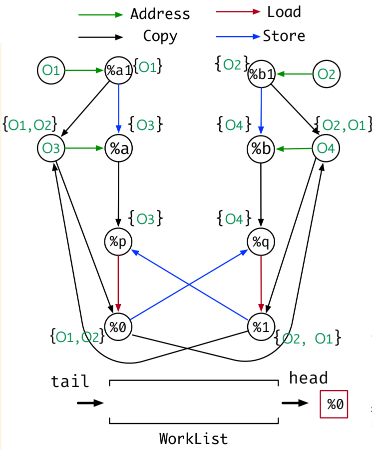
Algorithm 18: 1 Andersen's Pointer Analysis

Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) := pts(p) ∪ {o};
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Andersen's Pointer Analysis

```
define i32 @main() #0 {
entry:
%a1 = alloca i8, align 1        // O1
%b1 = alloca i8, align 1        // O2
%a = alloca i8*, align 8        // O3
%b = alloca i8*, align 8        // O4
store i8* %a1, i8** %a, align 8
store i8* %b1, i8** %b, align 8
call void @swap(i8** %a, i8** %b)
ret i32 0
}
define void @swap(i8** %p, i8** %q)
#0 {
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```



Algorithm 19: 1 Andersen's Pointer Analysis

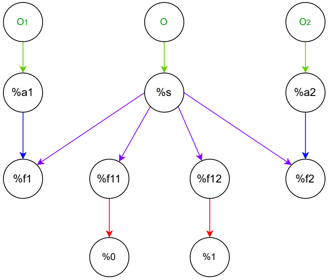
Input : $G = < V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) := pts(p) ∪ {o};
  pushIntoWorklist(p);
while WorkList ≠ ∅ **do**
  p := popFromWorklist();
  foreach o ∈ pts(p) **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    pts(x) := pts(x) ∪ pts(p);
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach o ∈ pts(p) **do**
      pts(x) := pts(x) ∪ {o.fld};
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

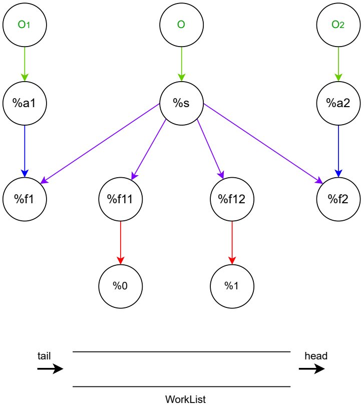
Field-Sensitive Andersen’s Pointer Analysis

```c
struct S{
  int* f1;
  int* f2;
};
int main(){
   struct S s;
   int a1, a2;
   s.f1 = &a1;
   s.f2 = &a2;
   int* p = s.f1;
   int* q = s.f2;
}
```

```llvm
define i32 @main() #0 {
entry:
  %s = alloca %struct.S, align 8
  %a1 = alloca i32, align 4
  %a2 = alloca i32, align 4
  %f1 = getelementptr inbounds %struct.S, ptr %s, i32 0, i32 0
  store ptr %a1, ptr %f1, align 8
  %f2 = getelementptr inbounds %struct.S, ptr %s, i32 0, i32 1
  store ptr %a2, ptr %f2, align 8
  %f11 = getelementptr inbounds %struct.S, ptr %s, i32 0, i32 0
  %0 = load ptr, ptr %f11, align 8
  %f22 = getelementptr inbounds %struct.S, ptr %s, i32 0, i32 1
  %1 = load ptr, ptr %f22, align 8
  ret i32 0
}
```



Field-Sensitive Andersen's Pointer Analysis

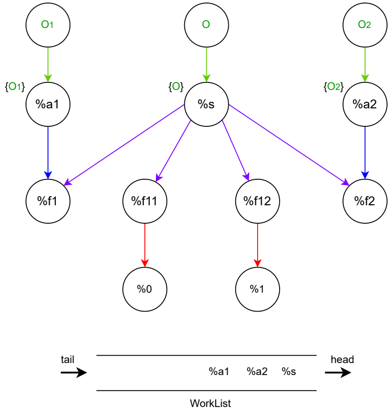


Algorithm 20: 1 Andersen's Pointer Analysis

Input : $G = < V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis

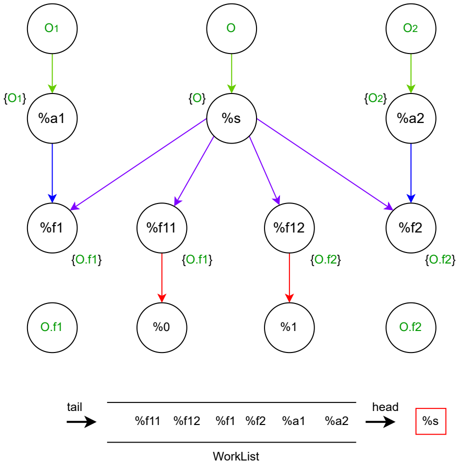


Algorithm 21: 1 Andersen's Pointer Analysis

Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis

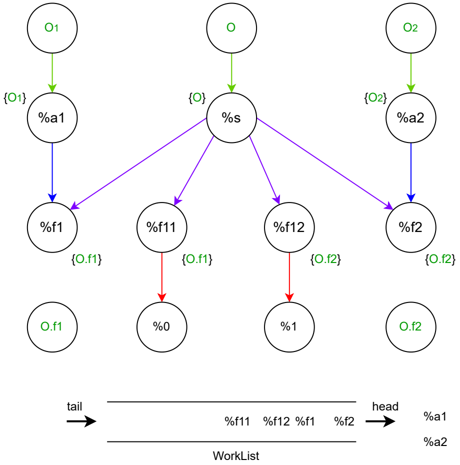


Algorithm 22: 1 Andersen's Pointer Analysis

Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

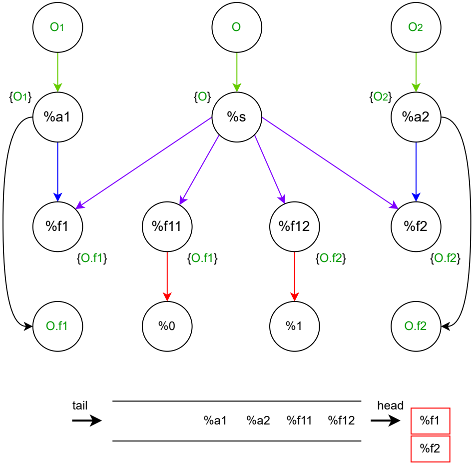
Field-Sensitive Andersen's Pointer Analysis



Algorithm 23: 1 Andersen's Pointer Analysis

```
Input : G =< V,E >: Constraint Graph
        V: a set of nodes in graph
        E: a set of edges in graph
WorkList := an empty vector of nodes;
foreach o --Address--> p do                         // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do               // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};           // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};           // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                    // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                 // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis

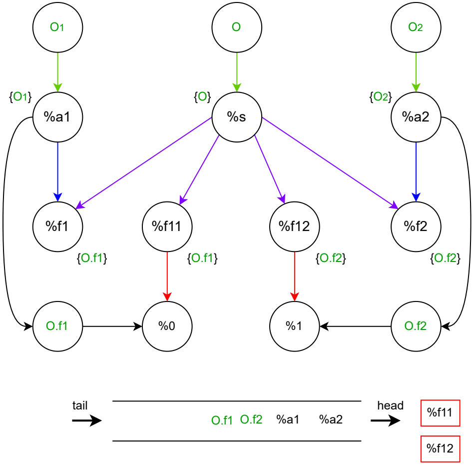


Algorithm 24: 1 Andersen's Pointer Analysis

Input : $G = <V, E>$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                         // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do               // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};           // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};           // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                    // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                 // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis

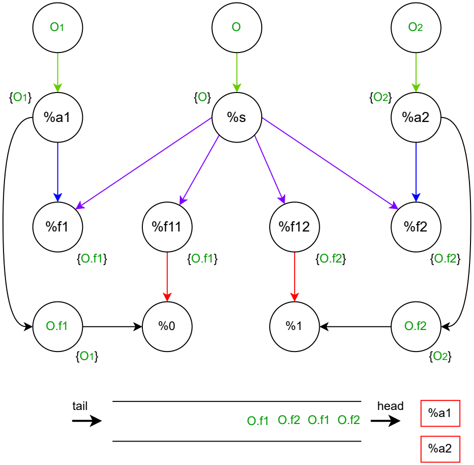


Algorithm 25: 1 Andersen's Pointer Analysis

Input : $G =< V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

```
WorkList := an empty vector of nodes;
foreach o --Address--> p do                      // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do            // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};        // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do             // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};        // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                 // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do              // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis

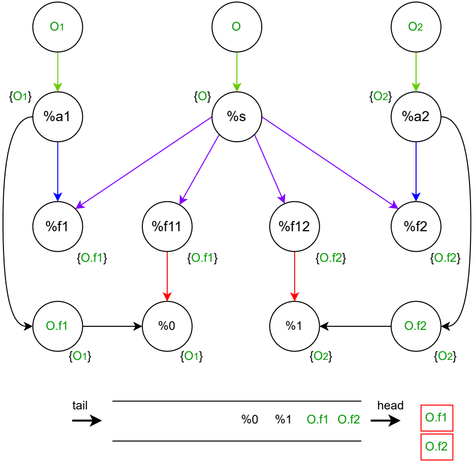


Algorithm 26: 1 Andersen's Pointer Analysis

Input : $G = < V, E >$: Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) = o;
  pushIntoWorklist(p);
while WorkList $\neq \varnothing$ **do**
  p := popFromWorklist();
  foreach $o \in pts(p)$ **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach $o \in pts(p)$ **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

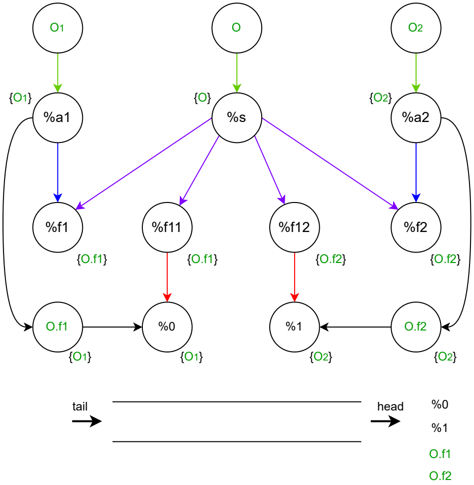
Field-Sensitive Andersen's Pointer Analysis



Algorithm 27: 1 Andersen's Pointer Analysis

```
Input : G =< V,E >: Constraint Graph
        V: a set of nodes in graph
        E: a set of edges in graph
WorkList := an empty vector of nodes;
foreach o --Address--> p do                          // Address rule
    pts(p) = o;
    pushIntoWorklist(p);
while WorkList ≠ ∅ do
    p := popFromWorklist();
    foreach o ∈ pts(p) do
        foreach q --Store--> p ∈ E do                // Store rule
            if q --Copy--> o ∉ E then
                E := E ∪ {q --Copy--> o};            // Add copy edge
                pushIntoWorklist(q);
        foreach p --Load--> r ∈ E do                 // Load rule
            if o --Copy--> r ∉ E then
                E := E ∪ {o --Copy--> r};            // Add copy edge
                pushIntoWorklist(o);
    foreach p --Copy--> x ∈ E do                     // Copy rule
        pts(x) := pts(x) ∪ pts(p);
        if pts(x) changed then
            pushIntoWorklist(x);
    foreach p --Gep,fld--> x ∈ E do                  // Gep rule
        foreach o ∈ pts(p) do
            pts(x) := pts(x) ∪ {o.fld};
        if pts(x) changed then
            pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis



Algorithm 28: 1 Andersen's Pointer Analysis

Input : $G = < V, E >$: Constraint Graph
 V: a set of nodes in graph
 E: a set of edges in graph

WorkList := an empty vector of nodes;
foreach $o \xrightarrow{\text{Address}} p$ **do** // Address rule
  pts(p) = o;
  pushIntoWorklist(p);
while WorkList $\neq \varnothing$ **do**
  p := popFromWorklist();
  foreach $o \in pts(p)$ **do**
    foreach $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule
      if $q \xrightarrow{\text{Copy}} o \notin E$ **then**
        $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge
        pushIntoWorklist(q);
    foreach $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule
      if $o \xrightarrow{\text{Copy}} r \notin E$ **then**
        $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge
        pushIntoWorklist(o);
  foreach $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule
    $pts(x) := pts(x) \cup pts(p)$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);
  foreach $p \xrightarrow{\text{Gep,fld}} x \in E$ **do** // Gep rule
    foreach $o \in pts(p)$ **do**
      $pts(x) := pts(x) \cup \{o.fld\}$;
    if *pts(x) changed* **then**
      pushIntoWorklist(x);

APIs for Implementing Andersen’s analysis

SVF::AndersenBase

```
::getPts(NodeID ptr)                   //get points-to set of ptr
::addPts(NodeID ptr, NodeID obj)       // add obj to point-to set of object ptr
::unionPts(NodeID ptr, NodeID ptr)     // union two point-to sets
::pushIntoWorklist(NodeID id)          // push the node to worklist
::popFromWorklist()                    // pop a node from the worklist
::isInWorklist(NodeID id)              // return true if the node in the worklist
::isWorklistEmpty()                    // return true if the worklist is empty
```

SVF::AndersenPTA

```
::addCopyEdge(NodeID src, NodeID dst) // add a copy edge from src to dst
```

SVF::ConstraintGraph

```
::getConstraintNode(nodeId id)     //get the node based on its id
::dump()                           // dump the ConsG
```

SVF::ConstraintNode

```
::getStoreInEdge()            // get incoming store edges of the node
::getStoreOutEdge()           //get outgoing store edges of the node
::getDirectOutEdge()          // get outgoing copy edges of the node
::getDirectInEdge()           // get incoming copy edges of the node
```

https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API#worklist-operations
https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API#points-to-set-operations
https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API#alias-relations
https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API#constraintgraph-constraintnode-and-constraintedge

Information Flow Tracking

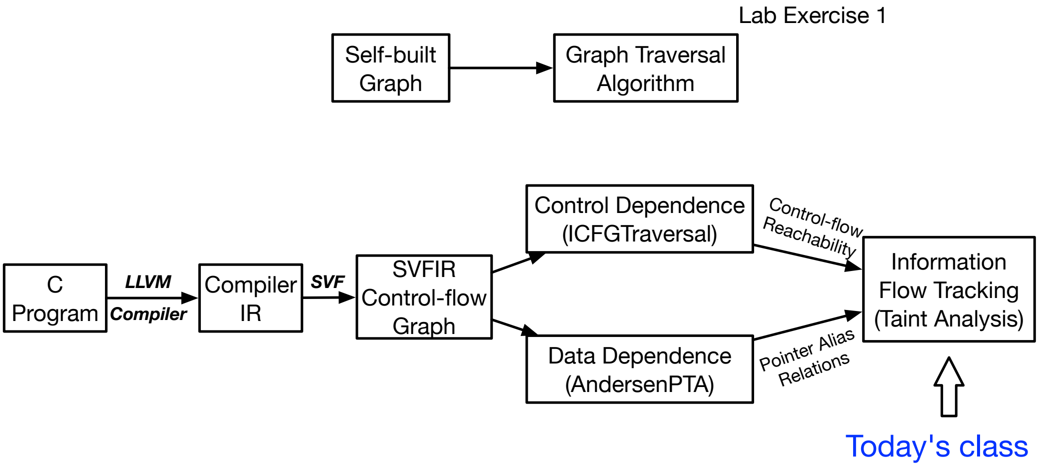
(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's Class



Taint Analysis

- Taint analysis aims to reason about the control and data dependence from a source (statement/node) to a sink (statement/node).
- Taint analysis can also be seen as information flow tracking analysis.
  - Static taint analysis: taint tracking at compile time (**this course**)
  - Dynamic taint analysis: taint tracking during runtime.

Taint Analysis

- Taint analysis aims to reason about the control and data dependence from a source (statement/node) to a sink (statement/node).
- Taint analysis can also be seen as information flow tracking analysis.
  - Static taint analysis: taint tracking at compile time (**this course**)
  - Dynamic taint analysis: taint tracking during runtime.

Why learn Taint Analysis?

- Detect information leakage
  - sensitive data stored in a heap object and manipulated by pointers can be passed around and stored to an unchecked memory (untrusted third-party APIs)
- Detect code vulnerability
  - There is a vulnerability if an unchecked tainted **source** (e.g., return value from an untrusted third party function) flows into one of the following **sinks**, where the tainted variable being used as
    - a parameter passed to a sensitive function or
    - a bound access (array index) or
    - a termination condition (loop condition)
    - . . .

Tainted Information Flows

Let us use what we have learned about control-flow and data-flow to develop an information flow checker to validate tainted flows from a source to a sink.

- A **source** $\mathbf{v_{src}}$@$\mathbf{s_{src}}$ is a tuple consisting of a variable $\mathbf{v_{src}}$ and a statement $\mathbf{s_{src}}$ where $\mathbf{v_{src}}$ is defined.
- A **sink** $\mathbf{v_{snk}}$@$\mathbf{s_{snk}}$ is also a tuple consisting of a variable $\mathbf{v_{snk}}$ and a statement $\mathbf{s_{snk}}$ where $\mathbf{v_{snk}}$ is used.
- In SVF, variables $\mathbf{v_{src}}$ and $\mathbf{v_{snk}}$ are `SVFVars`. Statements $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are `ICFGNodes`.

Tainted Information Flows

Let us use what we have learned about control-flow and data-flow to develop an information flow checker to validate tainted flows from a source to a sink.

- A **source** $\mathbf{v_{src}}@\mathbf{s_{src}}$ is a tuple consisting of a variable $\mathbf{v_{src}}$ and a statement $\mathbf{s_{src}}$ where $\mathbf{v_{src}}$ is defined.
- A **sink** $\mathbf{v_{snk}}@\mathbf{s_{snk}}$ is also a tuple consisting of a variable $\mathbf{v_{snk}}$ and a statement $\mathbf{s_{snk}}$ where $\mathbf{v_{snk}}$ is used.
- In SVF, variables $\mathbf{v_{src}}$ and $\mathbf{v_{snk}}$ are `SVFVars`. Statements $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are `ICFGNodes`.
- Given a **tainted** source $\mathbf{v_{src}}@\mathbf{s_{src}}$, we say that a sink $\mathbf{v_{snk}}@\mathbf{s_{snk}}$ is also **tainted** if both of the following two conditions satisfy:
  - $\mathbf{s_{src}}$ reaches $\mathbf{s_{snk}}$ on the ICFG (**`reachability` in Assignment-1**),
  - $\mathbf{v_{src}}$ and $\mathbf{v_{snk}}$ are aliases, (i.e., $pts(v_{src}) \cap pts(v_{snk})) \neq \emptyset$ (**`solveWorklist` in Assignment-1**)

Taint Analysis Example

Example 1

```c
int main(){
    char* secretToken = tgetstr();   // source
    char* a = secretToken;
    char* b = a;
    broadcast(b);                    // sink
}
```

What is the tainted flow?

Taint Analysis Example

Example 1

```
1 int main(){
2     char* secretToken = tgetstr();    // source
3     char* a = secretToken;
4     char* b = a;
5     broadcast(b);                     // sink
6 }
```

What is the tainted flow?

- Line 2 reaches Line 5 along the ICFG (control-dependence holds) `secretToken` and `b` are aliases (data-dependence holds)
- Both control-dependence and data-dependence hold. Therefore, `secretToken`@Line 2 flows to `b`@Line 5.

Taint Analysis Example

Example 2

```
int main(){
    char* secretToken = tgetstr(...);    // source
    char* a = secretToken;
    char* b = a;
    char* publicToken = "hello";
    broadcast(publicToken);          // sink
}
```

Do we have a tainted flow from source to sink?

Taint Analysis Example

Example 2

```
1  int main(){
2      char* secretToken = tgetstr(...);   // source
3      char* a = secretToken;
4      char* b = a;
5      char* publicToken = "hello";
6      broadcast(publicToken);         // sink
7  }
```

Do we have a tainted flow from source to sink?

- Line 2 reaches Line 6 along the ICFG (control-dependence holds),
- `secretToken` and `publicToken` are not aliases (data-dependence does not hold),
- `secretToken`@Line 2 does not flow to `publicToken`@Line 6.

Taint Analysis Example

Example 3

```
char* foo(char* token){ return token; }
int main(){
    if(condition){
        char*  secretToken = tgetstr(...);     // source
        char* b = foo(secretToken);
    }
    else{
        char* publicToken = "hello";
        char* a = foo(publicToken);
        broadcast(a);                         // sink
    }
}
```

Do we have a tainted flow from source to sink?

Taint Analysis Example

Example 3

```
1   char* foo(char* token){ return token; }
2   int main(){
3       if(condition){
4           char*  secretToken = tgetstr(...);    // source
5           char* b = foo(secretToken);
6       }
7       else{
8           char* publicToken = "hello";
9           char* a = foo(publicToken);
10          broadcast(a);                       // sink
11      }
12  }
```

Do we have a tainted flow from source to sink?

- `secretToken` and `a` are aliases due to callee `foo` (data-dependence holds),
- Line 4 does not reach Line 10 on ICFG (control-dependence does not hold),
- `secretToken`@Line 4 does not flow to `a`@Line 10.

Taint Analysis Example

Example 4

```c
int main(){
    char* secretToken = tgetstr(...);                  // source
    while(loopCondition){
        if(BranchCondition){
            char* a = secretToken;
            broadcast(a);                              // sink
        }
        else{
            char* b = "hello";
        }
    }
}
```

How many tainted flows from source to sink?

Taint Analysis Example

Example 4

```
1  int main(){
2      char* secretToken = tgetstr(...);                    // source
3      while(loopCondition){
4          if(BranchCondition){
5              char* a = secretToken;
6              broadcast(a);                                 // sink
7          }
8          else{
9              char* b = "hello";
10         }
11     }
12 }
```

How many tainted flows from source to sink?

- (At least) two paths from Line 2 to Line 6 on ICFG (control-dependence holds),
- `secretToken` and `a` are aliases (data-dependence holds),
- `secretToken`@Line 2 has two tainted paths flowing to `a`@Line 6.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}}$@$\mathbf{s_{src}}$ and a sink $\mathbf{v_{snk}}$@$\mathbf{s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}@s_{src}}$ and a sink $\mathbf{v_{snk}@s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.
- $\mathbf{v_{src}}$ is a return value from the call statement $\mathbf{s_{src}}$.
- $\mathbf{v_{snk}}$ is a parameter being passed to a call statement $\mathbf{s_{snk}}$.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}}@\mathbf{s_{src}}$ and a sink $\mathbf{v_{snk}}@\mathbf{s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.
- $\mathbf{v_{src}}$ is a return value from the call statement $\mathbf{s_{src}}$.
- $\mathbf{v_{snk}}$ is a parameter being passed to a call statement $\mathbf{s_{snk}}$.
- We can identify $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ according to different APIs, so as to configure sources and sinks.
- In Example 1, variable `secretToken` is $\mathbf{v_{src}}$ and `b` is $\mathbf{v_{snk}}$. The call statement `tgetstr(..)` represents $\mathbf{s_{src}}$ and `broadcast(..)` are used for $\mathbf{s_{snk}}$.
- In `Assignment-1`, you will need to implement `readSrcSnkFromFile` to identify sources and sinks configured by `SrcSnk.txt`.

What's next?

- (1) Understand data-flow and points-to analysis in today's slides
- (2) Finish the implementation of the four methods `readSrcSnkFromFile`, `reachability`, `solveWorklist`, `aliasCheck` in Assignment-1
- (3) Submit `Assignment-1.cpp` by 23:59 Tuesday, Week 4.