

Data-Flow and Taint Analysis

(Week 3)

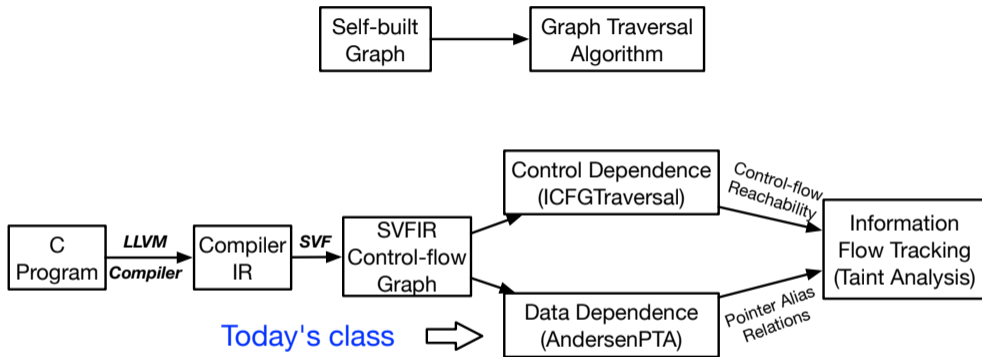
Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's class

Lab Exercise 1



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: $\text{pts}(p) = \text{pts}(q) = \{a\}$

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: $\text{pts}(p) = \text{pts}(q) = \{a\}$
- **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., $\text{pts}(p) \cap \text{pts}(q) \neq \emptyset$) then p and q are aliases. The dereferences 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

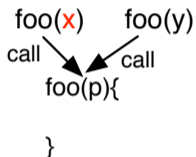
Assume x is a tainted value

$p = x$

$p = y$

flow-sensitivity

at which
program point
 p is tainted?



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 Copenhagen).

[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

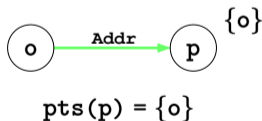
The analysis operating upon a program's constraint graph which is a subgraph of PAG (program assignment graph).

- 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
$p \xleftarrow{\text{Addr}} o$	<code>%p = alloca o</code>	<code>p = &o</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
$q \xleftarrow{\text{Copy}} p$	<code>%q = bitcast %p</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
$q \xleftarrow{\text{Load}} p$	<code>%q = load %p</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{add edge } q \xleftarrow{\text{Copy}} o$
$p \xleftarrow{\text{Store}} q$	<code>store %q, %p</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{add edge } o \xleftarrow{\text{Copy}} q$
$q \xleftarrow{\text{Gep, fld}} p$	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$

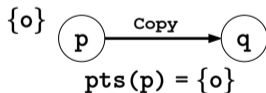
Field-Sensitive Andersen's Pointer Analysis

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



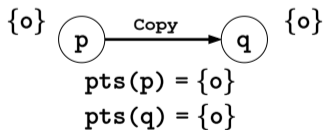
Field-Sensitive Andersen's Pointer Analysis

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



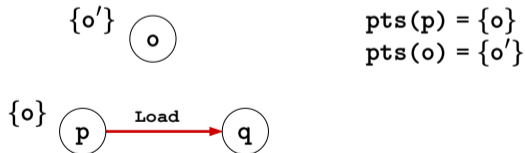
Field-Sensitive Andersen's Pointer Analysis

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



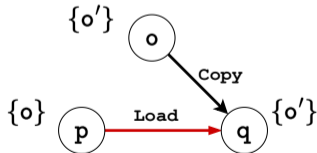
Field-Sensitive Andersen's Pointer Analysis

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



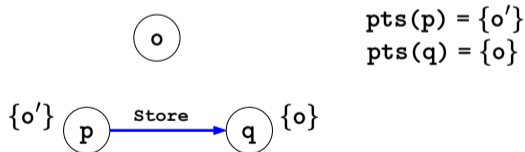
Field-Sensitive Andersen's Pointer Analysis

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



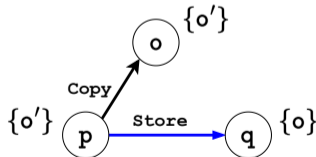
Field-Sensitive Andersen's Pointer Analysis

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



Field-Sensitive Andersen's Pointer Analysis

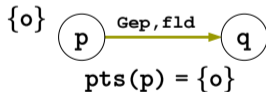
Constraint Edge	LLVM IR	C code	Constraint rules
$p \xleftarrow{\text{Addr}} o$	<code>%p = alloca_o</code>	<code>p = &o</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
$q \xleftarrow{\text{Copy}} p$	<code>%q = bitcast %p</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
$q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



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

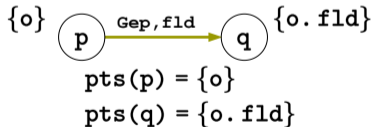
Field-Sensitive Andersen's Pointer Analysis

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



Field-Sensitive Andersen's Pointer Analysis

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



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

```
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



```
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
}
```

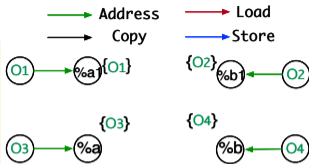
swap.ll

*<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
```

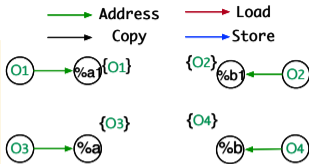
```
  store ptr %a1, ptr %a, align 8  
  store ptr %b1, ptr %b, align 8  
  call void @swap(ptr %a, ptr %b)  
  ret i32 0
```

```
}
```

```
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
```

```
}
```

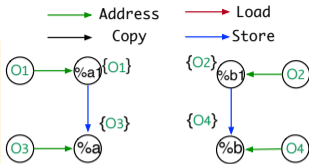


*<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag>

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  
}
```



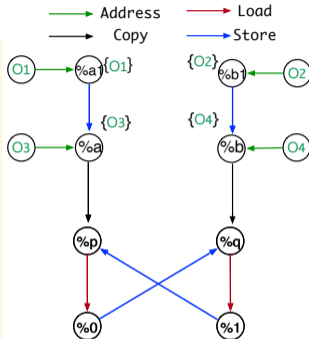
<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag>

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: 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

```

1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3   pts(p) = o;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10        E := E  $\cup$  {  $q \xrightarrow{\text{Copy}} o$  }; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14          E := E  $\cup$  {  $o \xrightarrow{\text{Copy}} r$  }; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          pushIntoWorklist(x);
    
```

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

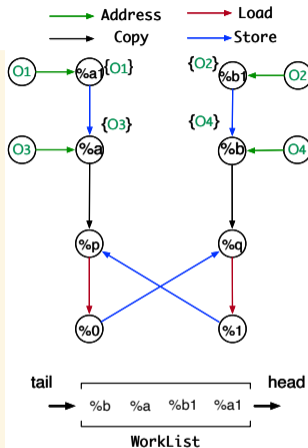
Andersen's Pointer Analysis

```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 $\text{pushIntoWorklist}(p)$;

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 $\text{pushIntoWorklist}(q)$;

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 $\text{pushIntoWorklist}(o)$;

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 $\text{pushIntoWorklist}(x)$;

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 $\text{pushIntoWorklist}(x)$;

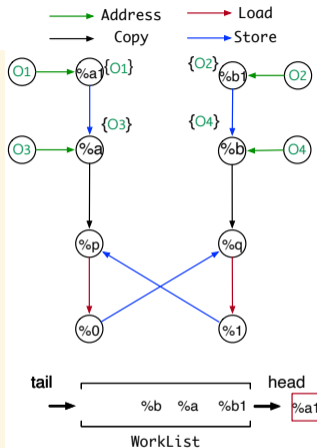
Andersen's Pointer Analysis

```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

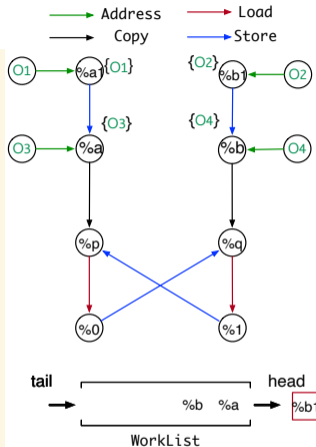
24 pushIntoWorklist(x);

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 $WorkList :=$ an empty vector of nodes;

2 **foreach** $o \xrightarrow{Address} p$ **do** // Address rule

3 $pts(p) := pts(p) \cup \{o\}$;

4 $pushIntoWorklist(p)$;

5 **while** $WorkList \neq \emptyset$ **do**

6 $p := popFromWorklist()$;

7 **foreach** $o \in pts(p)$ **do**

8 **foreach** $q \xrightarrow{Store} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{Copy} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge

11 $pushIntoWorklist(q)$;

12 **foreach** $r \xrightarrow{Load} p \in E$ **do** // Load rule

13 **if** $o \xrightarrow{Copy} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge

15 $pushIntoWorklist(o)$;

16 **foreach** $x \xrightarrow{Copy} p \in E$ **do** // Copy rule

17 $pts(x) := pts(x) \cup pts(p)$;

18 **if** $pts(x)$ changed **then**

19 $pushIntoWorklist(x)$;

20 **foreach** $p \xrightarrow{Gep.fld} x \in E$ **do** // Gep rule

21 **foreach** $o \in pts(p)$ **do**

22 $pts(x) := pts(x) \cup \{o.fld\}$;

23 **if** $pts(x)$ changed **then**

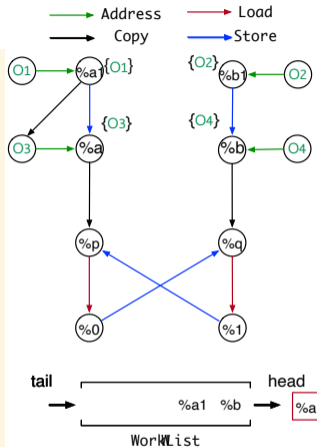
24 $pushIntoWorklist(x)$;

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorklist(x);

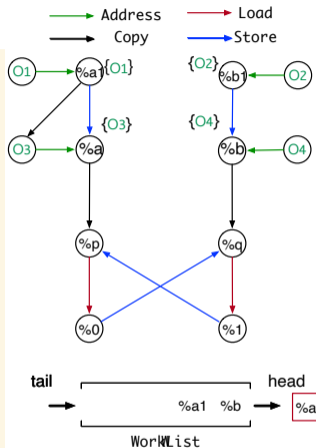
Andersen's Pointer Analysis

```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorklist(x);

25

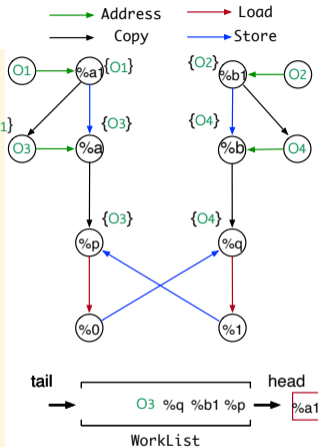
Andersen's Pointer Analysis

```

define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1 // O1
  %a = alloca ptr, align 8 // O1
  %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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 $WorkList :=$ an empty vector of nodes;

2 **foreach** $o \xrightarrow{Address} p$ **do** // Address rule

3 $pts(p) := pts(p) \cup \{o\}$;

4 $pushIntoWorklist(p)$;

5 **while** $WorkList \neq \emptyset$ **do**

6 $p := popFromWorklist()$;

7 **foreach** $o \in pts(p)$ **do**

8 **foreach** $q \xrightarrow{Store} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{Copy} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge

11 $pushIntoWorklist(q)$;

12 **foreach** $p \xrightarrow{Load} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{Copy} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge

15 $pushIntoWorklist(o)$;

16 **foreach** $p \xrightarrow{Copy} x \in E$ **do** // Copy rule

17 $pts(x) := pts(x) \cup pts(p)$;

18 **if** $pts(x)$ changed **then**

19 $pushIntoWorklist(x)$;

20 **foreach** $p \xrightarrow{Gep.fld} x \in E$ **do** // Gep rule

21 **foreach** $o \in pts(p)$ **do**

22 $pts(x) := pts(x) \cup \{o.fld\}$;

23 **if** $pts(x)$ changed **then**

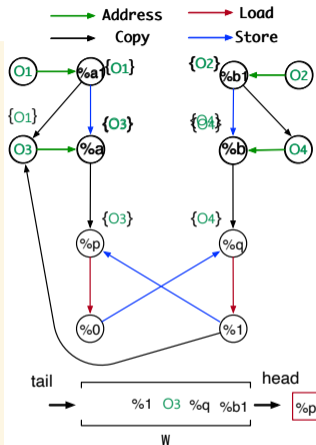
24 $pushIntoWorklist(x)$;

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

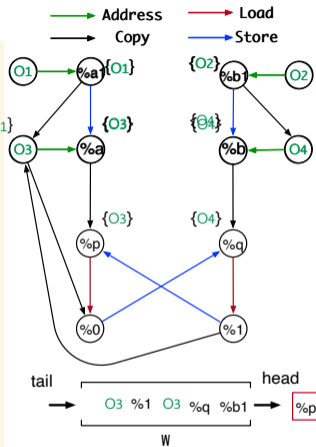
24 pushIntoWorklist(x);

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

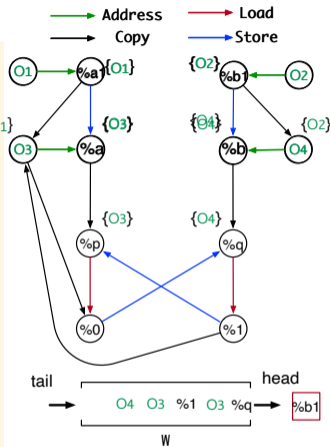
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3   pts(p) := pts(p)  $\cup$  {o};
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10        E := E  $\cup$  { $q \xrightarrow{\text{Copy}} o$ }; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14          E := E  $\cup$  { $o \xrightarrow{\text{Copy}} r$ }; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

1 WorkList := an empty vector of nodes;
2 foreach o  $\xrightarrow{\text{Address}}$  p do // Address rule
3   pts(p) := pts(p)  $\cup$  {o};
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach o  $\in$  pts(p) do
8     foreach q  $\xrightarrow{\text{Store}}$  p  $\in E$  do // Store rule
9       if q  $\xrightarrow{\text{Copy}}$  o  $\notin E$  then
10        E := E  $\cup$  {q  $\xrightarrow{\text{Copy}}$  o}; // Add copy edge
11        pushIntoWorklist(q);
12      foreach r  $\xrightarrow{\text{Load}}$  p  $\in E$  do // Load rule
13        if o  $\xrightarrow{\text{Copy}}$  r  $\notin E$  then
14          E := E  $\cup$  {o  $\xrightarrow{\text{Copy}}$  r}; // Add copy edge
15          pushIntoWorklist(o);
16      foreach p  $\xrightarrow{\text{Copy}}$  x  $\in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach p  $\xrightarrow{\text{Gep.fld}}$  x  $\in E$  do // Gep rule
21        foreach o  $\in$  pts(p) do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          pushIntoWorklist(x);
    
```

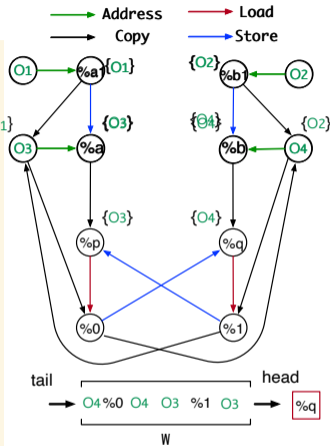
Andersen's Pointer Analysis

```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

1 WorkList := an empty vector of nodes;
2 foreach o  $\xrightarrow{\text{Address}}$  p do // Address rule
3   pts(p) := pts(p)  $\cup$  {o};
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach o  $\in$  pts(p) do
8     foreach q  $\xrightarrow{\text{Store}}$  p  $\in E$  do // Store rule
9       if q  $\xrightarrow{\text{Copy}}$  o  $\notin E$  then
10        E := E  $\cup$  {q  $\xrightarrow{\text{Copy}}$  o}; // Add copy edge
11        pushIntoWorklist(q);
12      foreach r  $\xrightarrow{\text{Load}}$  p  $\in E$  do // Load rule
13        if o  $\xrightarrow{\text{Copy}}$  r  $\notin E$  then
14          E := E  $\cup$  {o  $\xrightarrow{\text{Copy}}$  r}; // Add copy edge
15          pushIntoWorklist(o);
16  foreach p  $\xrightarrow{\text{Copy}}$  x  $\in E$  do // Copy rule
17    pts(x) := pts(x)  $\cup$  pts(p);
18    if pts(x) changed then
19      pushIntoWorklist(x);
20  foreach p  $\xrightarrow{\text{Gep.fld}}$  x  $\in E$  do // Gep rule
21    foreach o  $\in$  pts(p) do
22      pts(x) := pts(x)  $\cup$  {o.fld};
23    if pts(x) changed then
24      pushIntoWorklist(x);

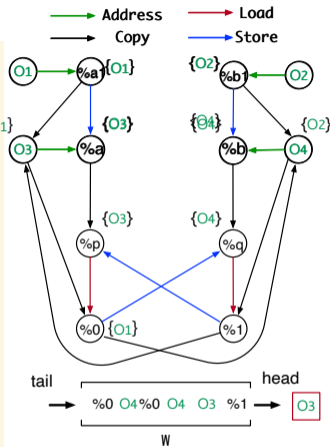
```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

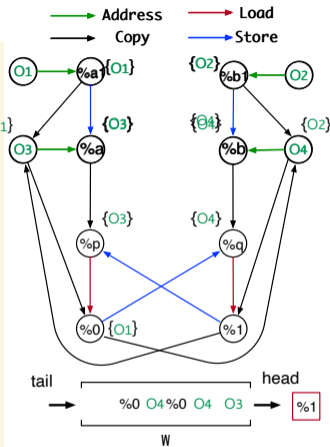
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) := \text{pts}(p) \cup \{o\}$ ;
4    $\text{pushIntoWorklist}(p)$ ;
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11         $\text{pushIntoWorklist}(q)$ ;
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15           $\text{pushIntoWorklist}(o)$ ;
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19           $\text{pushIntoWorklist}(x)$ ;
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24           $\text{pushIntoWorklist}(x)$ ;
    
```

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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

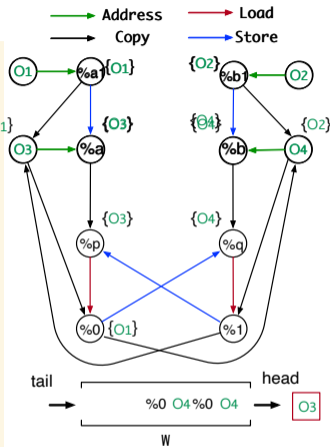
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) := \text{pts}(p) \cup \{o\}$ ;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

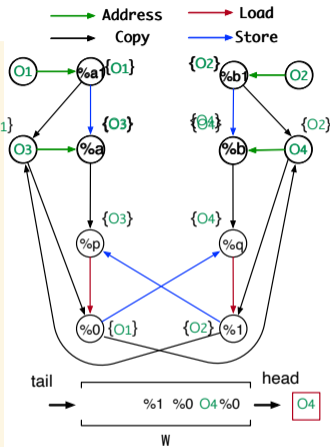
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) := \text{pts}(p) \cup \{o\}$ ;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

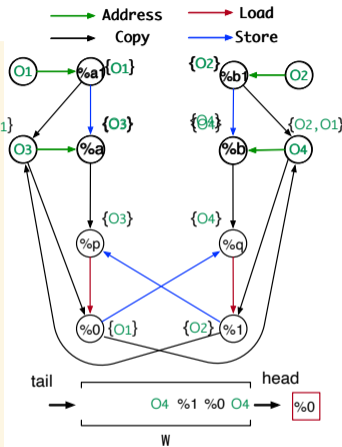
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3   pts(p) := pts(p)  $\cup$  {o};
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10        E := E  $\cup$  { $q \xrightarrow{\text{Copy}} o$ }; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14          E := E  $\cup$  { $o \xrightarrow{\text{Copy}} r$ }; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

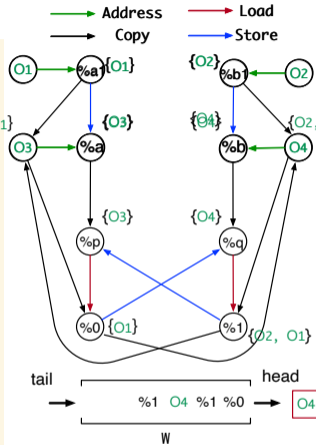
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) := \text{pts}(p) \cup \{o\}$ ;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

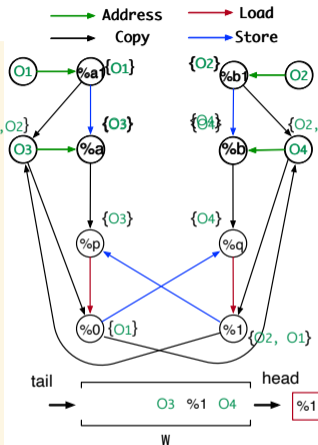
1 WorkList := an empty vector of nodes;
2 foreach o  $\xrightarrow{\text{Address}}$  p do // Address rule
3   pts(p) := pts(p)  $\cup$  {o};
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach o  $\in$  pts(p) do
8     foreach q  $\xrightarrow{\text{Store}}$  p  $\in E$  do // Store rule
9       if q  $\xrightarrow{\text{Copy}}$  o  $\notin E$  then
10        E := E  $\cup$  {o  $\xrightarrow{\text{Copy}}$  q}; // Add copy edge
11        pushIntoWorklist(q);
12      foreach r  $\xrightarrow{\text{Load}}$  p  $\in E$  do // Load rule
13        if o  $\xrightarrow{\text{Copy}}$  r  $\notin E$  then
14          E := E  $\cup$  {o  $\xrightarrow{\text{Copy}}$  r}; // Add copy edge
15          pushIntoWorklist(o);
16      foreach p  $\xrightarrow{\text{Copy}}$  x  $\in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach p  $\xrightarrow{\text{Gep.fld}}$  x  $\in E$  do // Gep rule
21        foreach o  $\in$  pts(p) do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```

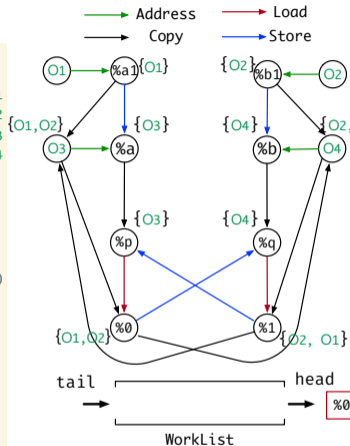
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) := \text{pts}(p) \cup \{o\}$ ;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $r \xrightarrow{\text{Load}} p \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24          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 : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) := \text{pts}(p) \cup \{o\}$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

21 **foreach** $o \in \text{pts}(p)$ **do**

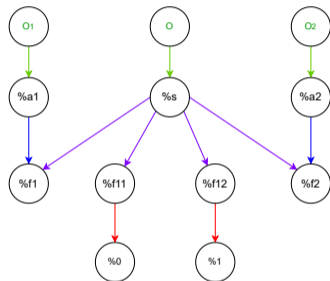
22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

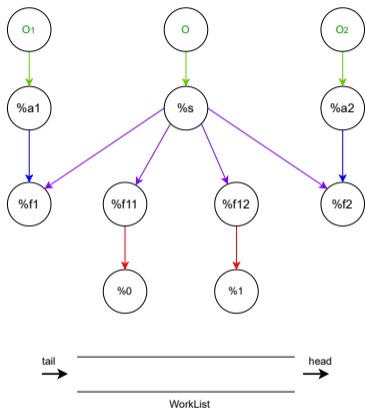
24 pushIntoWorklist(x);

Field-Sensitive Andersen's Pointer Analysis

```
1 struct S{
2   int* f1;
3   int* f2;
4 };
5 int main(){
6   struct S s;
7   int a1, a2;
8   s.f1 = &a1;
9   s.f2 = &a2;
10  int* p = s.f1;
11  int* q = s.f2;
12 }
13
14 define i32 @main() #0 {
15 entry:
16   %s = alloca @struct.S, align 8
17   %a1 = alloca i32, align 4
18   %a2 = alloca i32, align 4
19   %f1 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 0
20   store ptr %a1, ptr %f1, align 8
21   %f2 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 1
22   store ptr %a2, ptr %f2, align 8
23   %f11 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 0
24   %0 = load ptr, ptr %f11, align 8
25   %f22 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 1
26   %1 = load ptr, ptr %f22, align 8
27   ret i32 0
28 }
```



Field-Sensitive Andersen's Pointer Analysis



Algorithm 20: 1 Andersen's Pointer Analysis

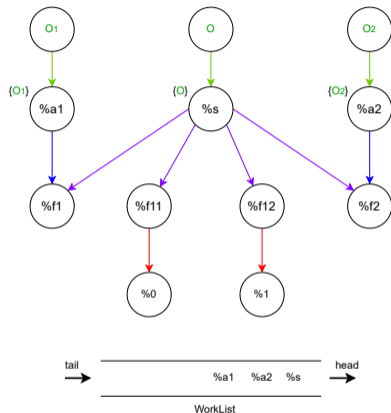
Input : $G = \langle V, E \rangle$: Constraint Graph

V : a set of nodes in graph

E : a set of edges in graph

```
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) = o$ ;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep\_fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$ ;
23        if  $\text{pts}(x)$  changed then
24          pushIntoWorklist(x);
```

Field-Sensitive Andersen's Pointer Analysis



Algorithm 21: 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

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) = o$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep_fld}} x \in E$ **do** // Gep rule

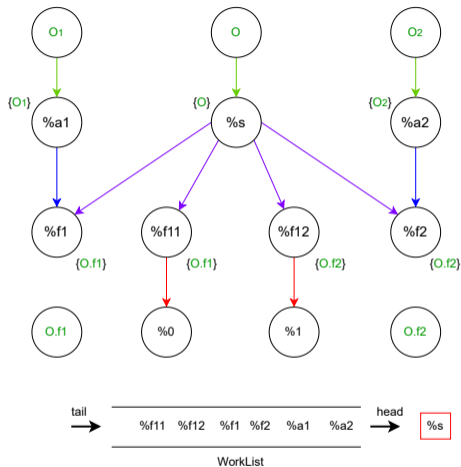
21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorklist(x);

Field-Sensitive Andersen's Pointer Analysis



Algorithm 22: 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

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) = o$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

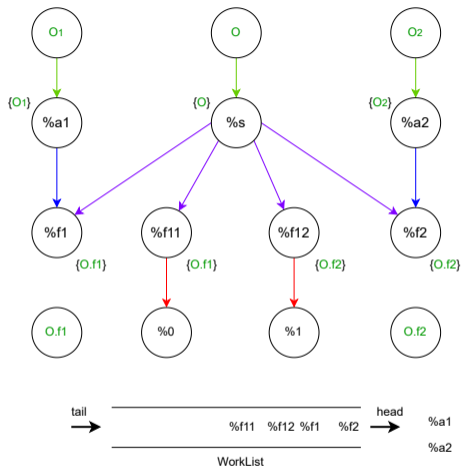
21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.fld\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorklist(x);

Field-Sensitive Andersen's Pointer Analysis



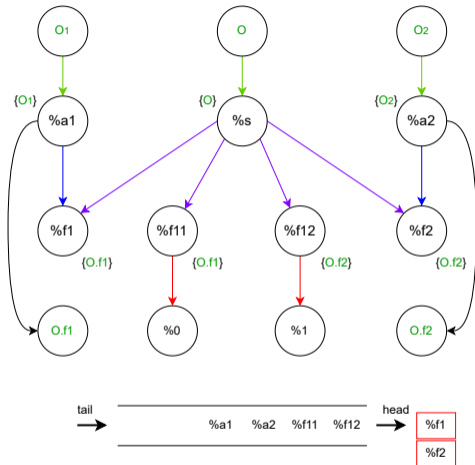
Algorithm 23: 1 Andersen's Pointer Analysis

```

Input :  $G = \langle V, E \rangle$ : Constraint Graph
          $V$ : a set of nodes in graph
          $E$ : a set of edges in graph
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) = o$ ;
4    $\text{pushIntoWorklist}(p)$ ;
5 while  $\text{WorkList} \neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11         $\text{pushIntoWorklist}(q)$ ;
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15           $\text{pushIntoWorklist}(o)$ ;
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19           $\text{pushIntoWorklist}(x)$ ;
20      foreach  $p \xrightarrow{\text{Gep\_fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.fld\}$ ;
23        if  $\text{pts}(x)$  changed then
24           $\text{pushIntoWorklist}(x)$ ;

```

Field-Sensitive Andersen's Pointer Analysis



Algorithm 24: 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

1 WorkList := an empty vector of nodes;

2 **foreach** $o \xrightarrow{\text{Address}} p$ **do** // Address rule

3 $\text{pts}(p) = o$;

4 pushIntoWorklist(p);

5 **while** WorkList $\neq \emptyset$ **do**

6 $p := \text{popFromWorklist}()$;

7 **foreach** $o \in \text{pts}(p)$ **do**

8 **foreach** $q \xrightarrow{\text{Store}} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$; // Add copy edge

11 pushIntoWorklist(q);

12 **foreach** $p \xrightarrow{\text{Load}} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{\text{Copy}} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$; // Add copy edge

15 pushIntoWorklist(o);

16 **foreach** $p \xrightarrow{\text{Copy}} x \in E$ **do** // Copy rule

17 $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$;

18 **if** $\text{pts}(x)$ changed **then**

19 pushIntoWorklist(x);

20 **foreach** $p \xrightarrow{\text{Gep.fld}} x \in E$ **do** // Gep rule

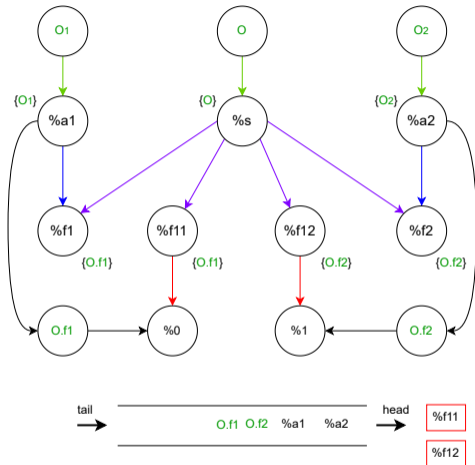
21 **foreach** $o \in \text{pts}(p)$ **do**

22 $\text{pts}(x) := \text{pts}(x) \cup \{o.\text{fld}\}$;

23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorklist(x);

Field-Sensitive Andersen's Pointer Analysis



Algorithm 25: 1 Andersen's Pointer Analysis

Input : $G = \langle V, E \rangle$: Constraint Graph

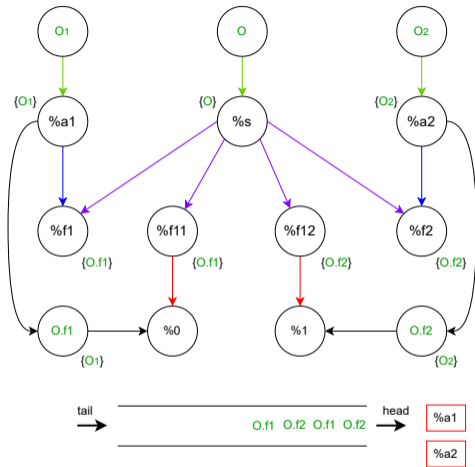
V : a set of nodes in graph

E : a set of edges in graph

```

1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3   pts(p) = o;
4   pushIntoWorklist(p);
5 while WorkList  $\neq \emptyset$  do
6   p := popFromWorklist();
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10        E := E  $\cup$  { $q \xrightarrow{\text{Copy}} o$ }; // Add copy edge
11        pushIntoWorklist(q);
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14          E := E  $\cup$  { $o \xrightarrow{\text{Copy}} r$ }; // Add copy edge
15          pushIntoWorklist(o);
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17        pts(x) := pts(x)  $\cup$  pts(p);
18        if pts(x) changed then
19          pushIntoWorklist(x);
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22          pts(x) := pts(x)  $\cup$  {o.fld};
23        if pts(x) changed then
24          pushIntoWorklist(x);
  
```

Field-Sensitive Andersen's Pointer Analysis



Algorithm 26: 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

1 $WorkList :=$ an empty vector of nodes;

2 **foreach** $o \xrightarrow{Address} p$ **do** // Address rule

3 $pts(p) = o$;

4 $pushIntoWorklist(p)$;

5 **while** $WorkList \neq \emptyset$ **do**

6 $p := popFromWorklist()$;

7 **foreach** $o \in pts(p)$ **do**

8 **foreach** $q \xrightarrow{Store} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{Copy} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge

11 $pushIntoWorklist(q)$;

12 **foreach** $p \xrightarrow{Load} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{Copy} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge

15 $pushIntoWorklist(o)$;

16 **foreach** $p \xrightarrow{Copy} x \in E$ **do** // Copy rule

17 $pts(x) := pts(x) \cup pts(p)$;

18 **if** $pts(x)$ **changed** **then**

19 $pushIntoWorklist(x)$;

20 **foreach** $p \xrightarrow{Gep.fld} x \in E$ **do** // Gep rule

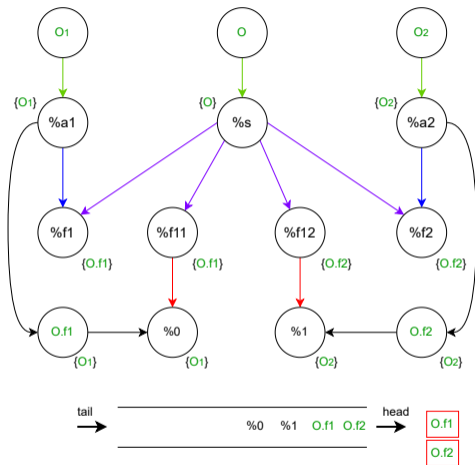
21 **foreach** $o \in pts(p)$ **do**

22 $pts(x) := pts(x) \cup \{o.fld\}$;

23 **if** $pts(x)$ **changed** **then**

24 $pushIntoWorklist(x)$;

Field-Sensitive Andersen's Pointer Analysis



Algorithm 27: 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

1 $WorkList :=$ an empty vector of nodes;

2 **foreach** $o \xrightarrow{Address} p$ **do** // Address rule

3 $pts(p) = o$;

4 $pushIntoWorklist(p)$;

5 **while** $WorkList \neq \emptyset$ **do**

6 $p := popFromWorklist()$;

7 **foreach** $o \in pts(p)$ **do**

8 **foreach** $q \xrightarrow{Store} p \in E$ **do** // Store rule

9 **if** $q \xrightarrow{Copy} o \notin E$ **then**

10 $E := E \cup \{q \xrightarrow{Copy} o\}$; // Add copy edge

11 $pushIntoWorklist(q)$;

12 **foreach** $p \xrightarrow{Load} r \in E$ **do** // Load rule

13 **if** $o \xrightarrow{Copy} r \notin E$ **then**

14 $E := E \cup \{o \xrightarrow{Copy} r\}$; // Add copy edge

15 $pushIntoWorklist(o)$;

16 **foreach** $p \xrightarrow{Copy} x \in E$ **do** // Copy rule

17 $pts(x) := pts(x) \cup pts(p)$;

18 **if** $pts(x)$ **changed** **then**

19 $pushIntoWorklist(x)$;

20 **foreach** $p \xrightarrow{Gep.fld} x \in E$ **do** // Gep rule

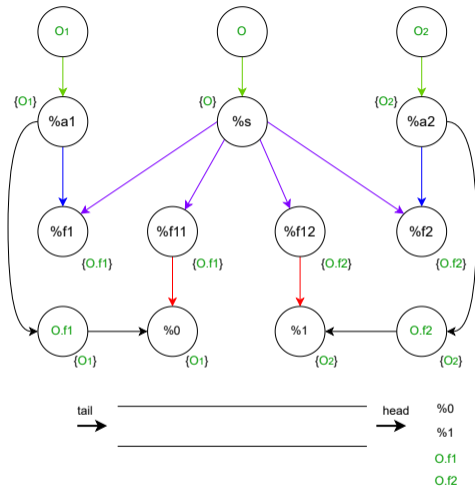
21 **foreach** $o \in pts(p)$ **do**

22 $pts(x) := pts(x) \cup \{o.fld\}$;

23 **if** $pts(x)$ **changed** **then**

24 $pushIntoWorklist(x)$;

Field-Sensitive Andersen's Pointer Analysis



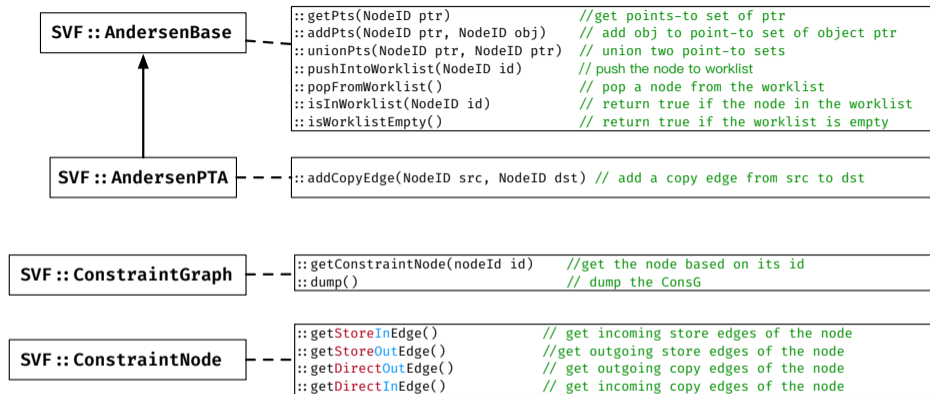
Algorithm 28: 1 Andersen's Pointer Analysis

```

Input :  $G = \langle V, E \rangle$ : Constraint Graph
          $V$ : a set of nodes in graph
          $E$ : a set of edges in graph
1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p$  do // Address rule
3    $\text{pts}(p) = o$ ;
4    $\text{pushIntoWorklist}(p)$ ;
5 while  $\text{WorkList} \neq \emptyset$  do
6    $p := \text{popFromWorklist}()$ ;
7   foreach  $o \in \text{pts}(p)$  do
8     foreach  $q \xrightarrow{\text{Store}} p \in E$  do // Store rule
9       if  $q \xrightarrow{\text{Copy}} o \notin E$  then
10         $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$ ; // Add copy edge
11         $\text{pushIntoWorklist}(q)$ ;
12      foreach  $p \xrightarrow{\text{Load}} r \in E$  do // Load rule
13        if  $o \xrightarrow{\text{Copy}} r \notin E$  then
14           $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$ ; // Add copy edge
15           $\text{pushIntoWorklist}(o)$ ;
16      foreach  $p \xrightarrow{\text{Copy}} x \in E$  do // Copy rule
17         $\text{pts}(x) := \text{pts}(x) \cup \text{pts}(p)$ ;
18        if  $\text{pts}(x)$  changed then
19           $\text{pushIntoWorklist}(x)$ ;
20      foreach  $p \xrightarrow{\text{Gep.fld}} x \in E$  do // Gep rule
21        foreach  $o \in \text{pts}(p)$  do
22           $\text{pts}(x) := \text{pts}(x) \cup \{o.fld\}$ ;
23        if  $\text{pts}(x)$  changed then
24           $\text{pushIntoWorklist}(x)$ ;

```

APIs for Implementing Andersen's analysis



<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-constrainededge>