

Data-Flow and Pointer Aliasing

(Week 3)

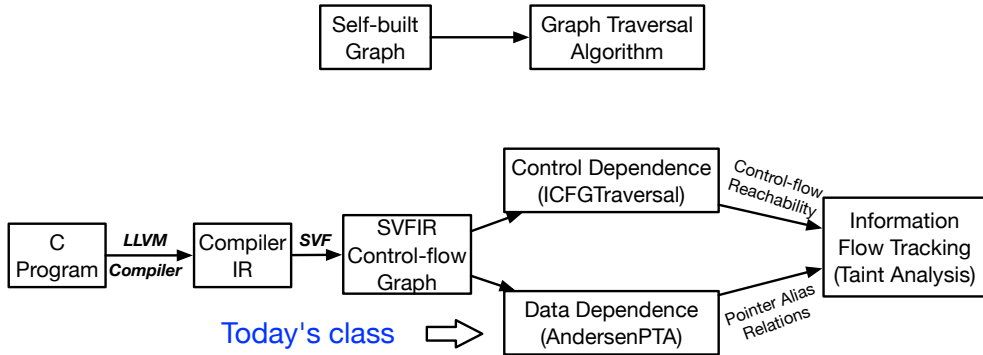
Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's class

Lab Exercise 1



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 i8* **%a1**, i8** %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 i8* **%a1**, i8** %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 i8* %a1, i8** %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 i8* %a1, i8** %a, align 8`
 - Instruction-2: `%c = load i8** %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

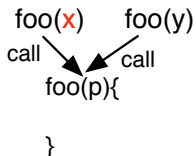
Assume **x** is a tainted value

$p = \mathbf{x}$

$p = y$

flow-sensitivity

at which
program point
 p is tainted?



context-sensitivity

under which
calling context
 p is tainted?

if(cond)

$p = \mathbf{x}$

else

$p = y$

path-sensitivity

along which
program path
 p is tainted?

Andersen's Pointer Analysis

Flow-, context-, and path-insensitive 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 called **inclusion-based points-to analysis**, the most popular and widely used pointer analysis.
- Constraint solving, i.e., inclusion-based constraint solving between SVFVars in the form of ConstraintNodes.
- The analysis requires iterative constraint solving by (1) propagating points-to sets among constraint graph nodes, and (2) adding new edges until a fixed point is reached, i.e., no new edges are added. (**Lab-Exercise-1**)

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

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

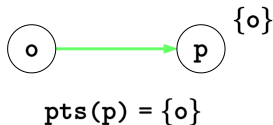
The analysis operating upon a program's constraint graph which is a subgraph of PAG (program assignment graph).

- 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

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$

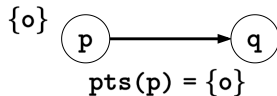
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



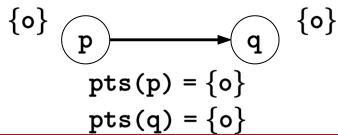
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



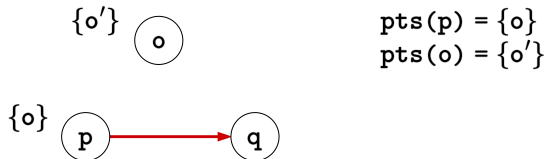
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



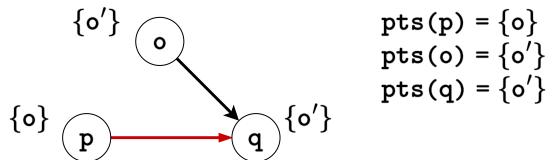
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



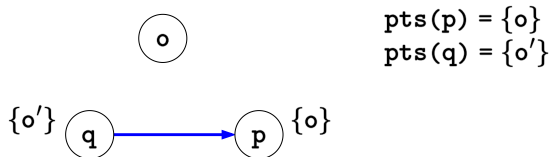
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



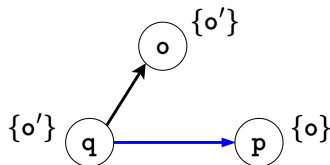
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<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

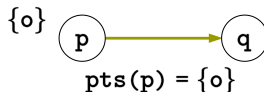
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



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

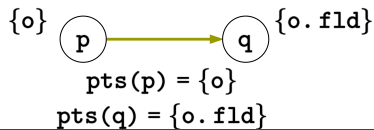
Field-Sensitive Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<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

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	<code>%ptr = alloca_o</code>	<code>p = alloc</code>	$\text{pts}(p) = \text{pts}(p) \cup \{o\}$
CopyStmt	<code>%p = bitcast %q</code>	<code>q = p</code>	$\text{pts}(q) = \text{pts}(q) \cup \text{pts}(p)$
LoadStmt	<code>%p = load %q</code>	<code>q = *p</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(o) \cup \text{pts}(q)$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\forall o \in \text{pts}(p) : \text{pts}(o) = \text{pts}(q) \cup \text{pts}(o)$
GepStmt	<code>%q = gep %p, fld</code>	<code>q = &p → fld</code>	$\forall o \in \text{pts}(p) : \text{pts}(q) = \text{pts}(q) \cup \{o.\text{fld}\}$



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
```

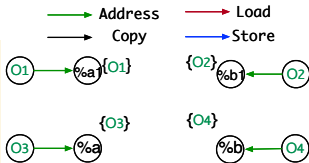
```
  store ptr %a1, ptr %a, align 8  
  store ptr %b1, ptr %b, align 8  
  call void @swap(ptr %a, ptr %b)  
  ret i32 0
```

```
}
```

```
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
```

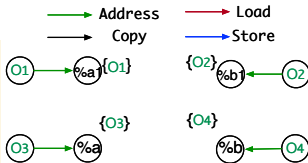
```
  store ptr %a1, ptr %a, align 8  
  store ptr %b1, ptr %b, align 8  
  call void @swap(ptr %a, ptr %b)  
  ret i32 0
```

```
}
```

```
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
```

```
}
```

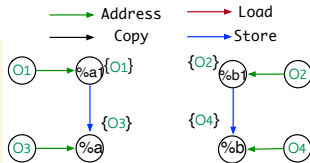


*<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag>

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  
}
```

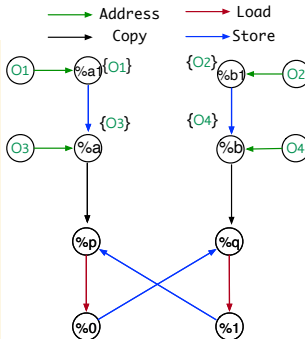


<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag>

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: Anderson'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  $o \xrightarrow{\text{Address}} p \in E$  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  $p \xrightarrow{\text{Gep}} x \in E$  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);
```

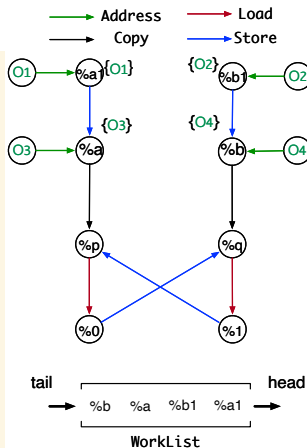
<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag>

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 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

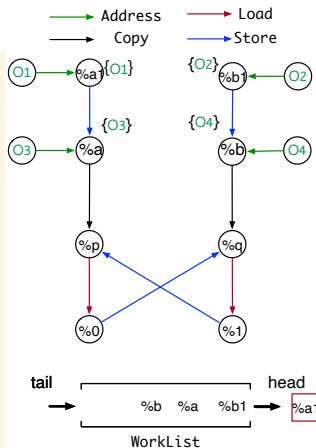
23 **if** $\text{pts}(x)$ changed **then**

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 3: 1 Anderson'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** $p \xrightarrow{Copy} x \in E$ **do** // Copy rule

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

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

19 $pushIntoWorkList(x)$;

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

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

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

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

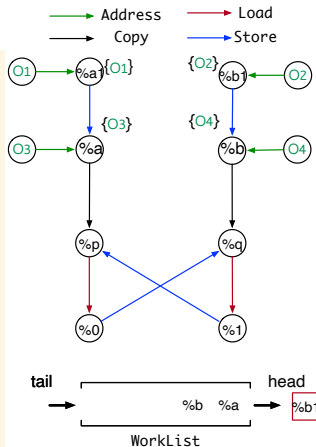
24 $pushIntoWorkList(x)$;

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 4: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

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

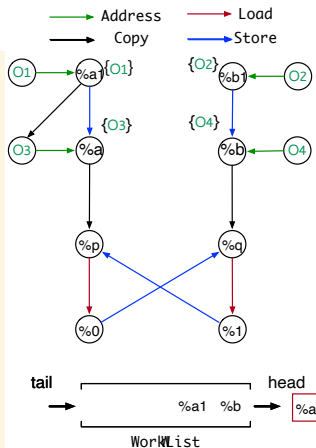
24 $pushIntoWorkList(x)$;

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 Anderson'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** $p \xrightarrow{Gep} x \in E$ **do** // Gep rule

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

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

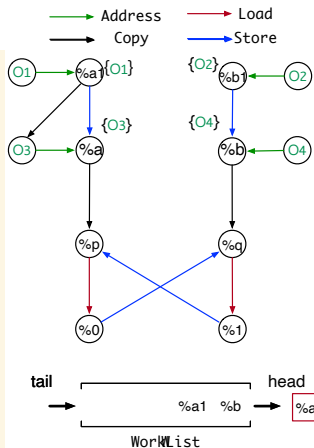
23 **if** $pts(x)$ changed **then**

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

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** $o \xrightarrow{\text{Load}} r \in E$ **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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

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

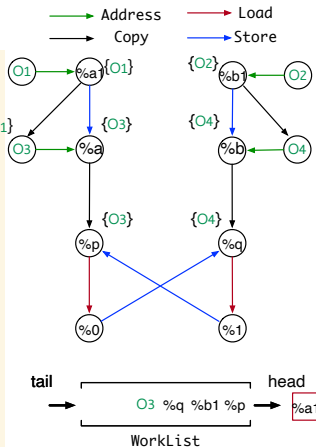
24 $pushIntoWorklist(x);$

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 Anderson'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** $p \xrightarrow{Copy} x \in E$ **do** // Copy rule

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

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

19 $pushIntoWorkList(x)$;

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

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

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

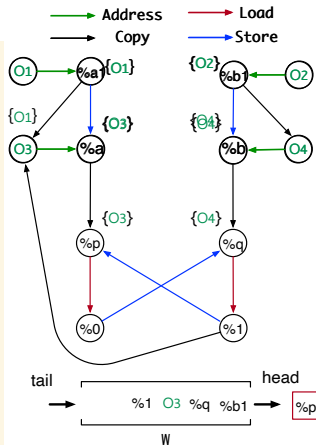
23 **if** $pts(x)$ changed **then**

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

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

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

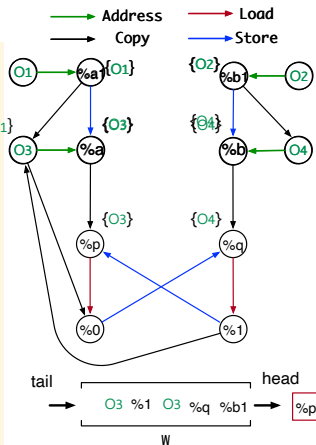
23 **if** $\text{pts}(x)$ changed **then**

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

4 pushIntoWorkList(p);

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

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

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

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

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

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

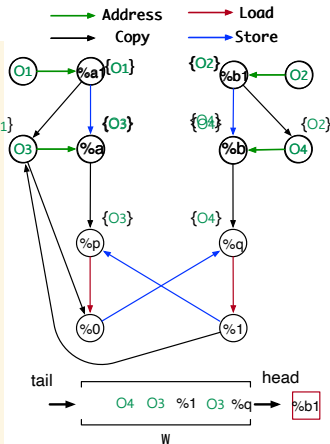
23 **if** $\text{pts}(x)$ changed **then**

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 10: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

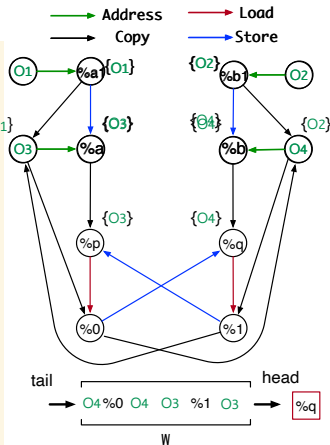
23 **if** $\text{pts}(x)$ changed **then**

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 11: 1 Anderson'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** // Store rule

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

9 **if** $q \xrightarrow{\text{Copy}} o \notin E$ **then** // Add copy edge

10 $E := E \cup \{q \xrightarrow{\text{Copy}} o\}$;

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** // Add copy edge

14 $E := E \cup \{o \xrightarrow{\text{Copy}} r\}$;

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

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

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

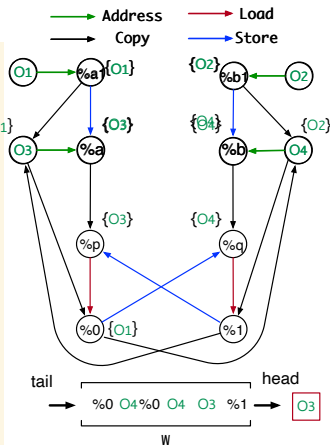
23 **if** $\text{pts}(x)$ changed **then**

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 12: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

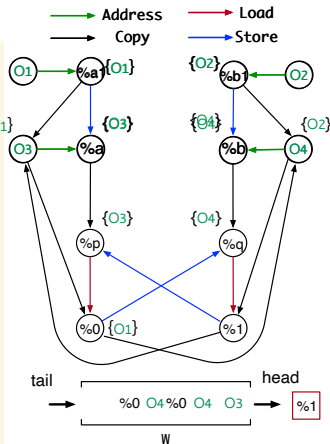
23 **if** $\text{pts}(x)$ changed **then**

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 13: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

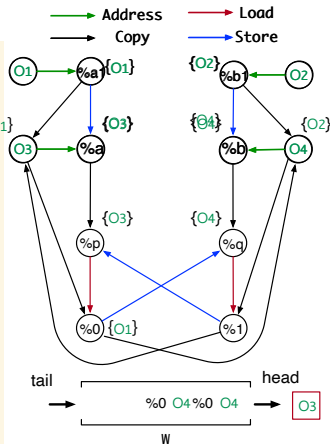
23 **if** $\text{pts}(x)$ changed **then**

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 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

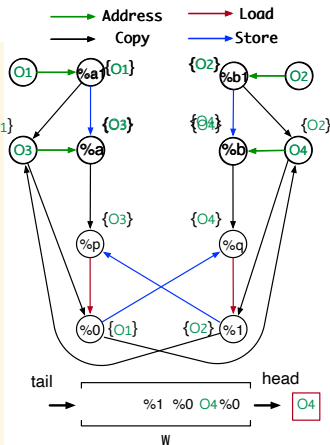
23 **if** $\text{pts}(x)$ changed **then**

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 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

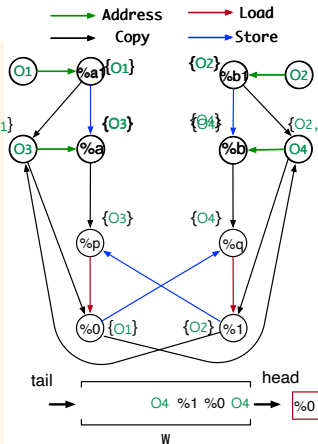
23 **if** $\text{pts}(x)$ changed **then**

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 16: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

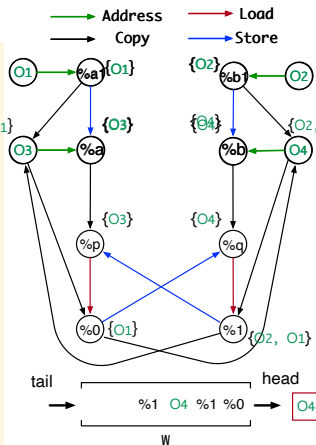
23 **if** $\text{pts}(x)$ changed **then**

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 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

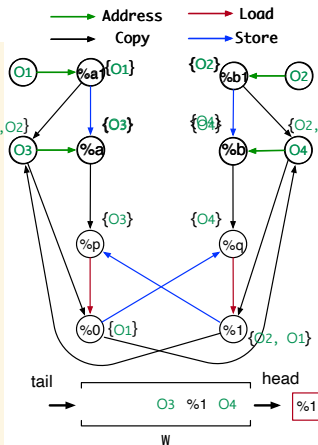
23 **if** $\text{pts}(x)$ changed **then**

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 18: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

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

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

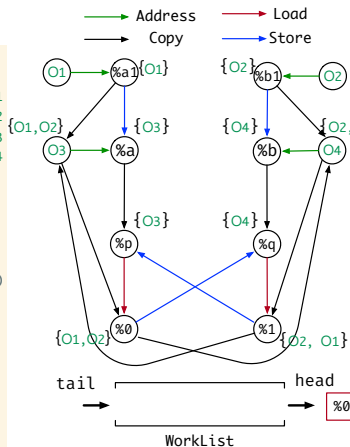
23 **if** $\text{pts}(x)$ changed **then**

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 Anderson'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** // Store rule

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

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

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

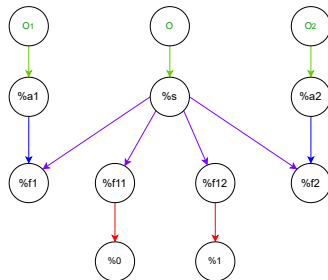
23 **if** $\text{pts}(x)$ changed **then**

24 pushIntoWorkList(x);

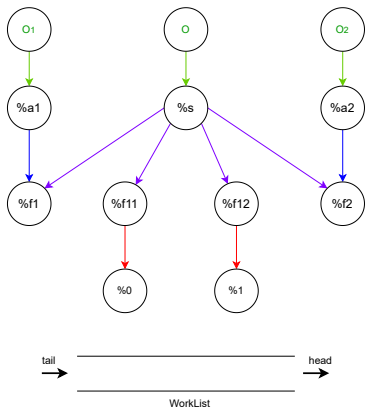
Field-Sensitive Andersen's Pointer Analysis

```
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 }

1 define i32 @main() #0 {
2 entry:
3     %s = alloca @struct.S, align 8
4     %a1 = alloca i32, align 4
5     %a2 = alloca i32, align 4
6     %f1 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 0
7     store ptr %a1, ptr %f1, align 8
8     %f2 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 1
9     store ptr %a2, ptr %f2, align 8
10    %f11 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 0
11    %0 = load ptr, ptr %f11, align 8
12    %f22 = getelementptr inbounds @struct.S, ptr %s, i32 0, i32 1
13    %1 = load ptr, ptr %f22, align 8
14    ret i32 0
15 }
```



Field-Sensitive Andersen's Pointer Analysis



Algorithm 20: 1 Anderson's Pointer Analysis

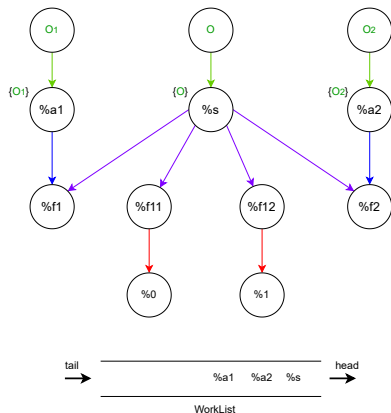
Input : $G = \langle V, E \rangle$: Constraint Graph
 V : a set of nodes in graph
 E : a set of edges in graph

```

1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p \in E$  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  $p \xrightarrow{\text{Gep}} x \in E$  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 21: 1 Anderson'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** $o \xrightarrow{\text{Address}} p \in E$ **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** $q \xrightarrow{\text{Store}} p \in E$ **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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

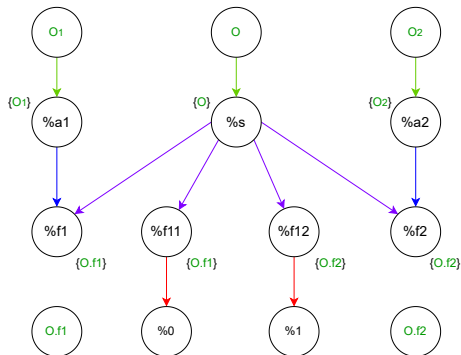
21 **foreach** $o \in pts(p)$ **do**

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

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

24 $pushIntoWorklist(x)$;

Field-Sensitive Andersen's Pointer Analysis



Algorithm 22: 1 Anderson'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** $WorkList \neq \emptyset$

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** $p \xrightarrow{\text{Load}} r \in E$ **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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

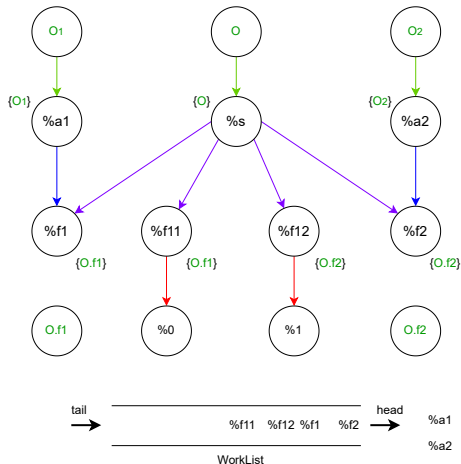
21 **foreach** $o \in pts(p)$ **do**

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

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

24 $pushIntoWorklist(x)$;

Field-Sensitive Andersen's Pointer Analysis



Algorithm 23: 1 Anderson's Pointer Analysis

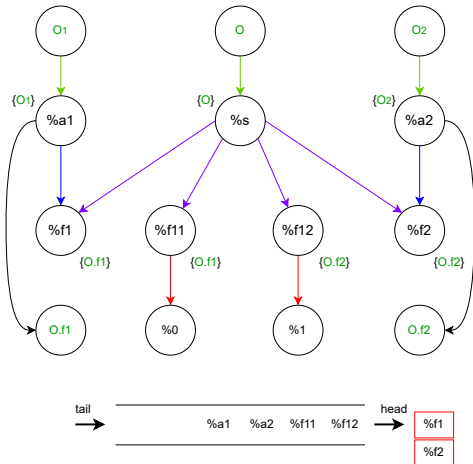
Input : $G = \langle V, E \rangle$: Constraint Graph
 V : a set of nodes in graph
 E : a set of edges in graph

```

1 WorkList := an empty vector of nodes;
2 foreach  $o \xrightarrow{\text{Address}} p \in E$  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  $p \xrightarrow{\text{Gep}} x \in E$  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 24: 1 Anderson'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 $\neq \emptyset$

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

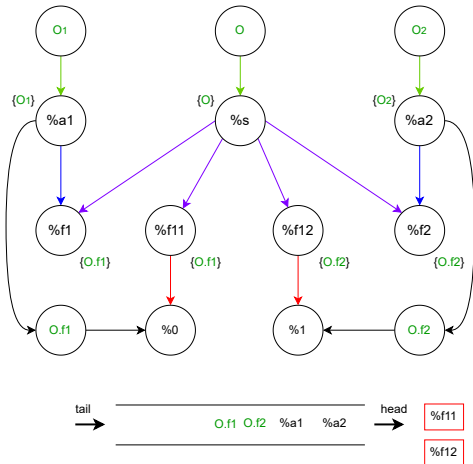
21 **foreach** $o \in \text{pts}(p)$ **do**

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 Anderson'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** $p \xrightarrow{Gep} x \in E$ **do** // Gep rule

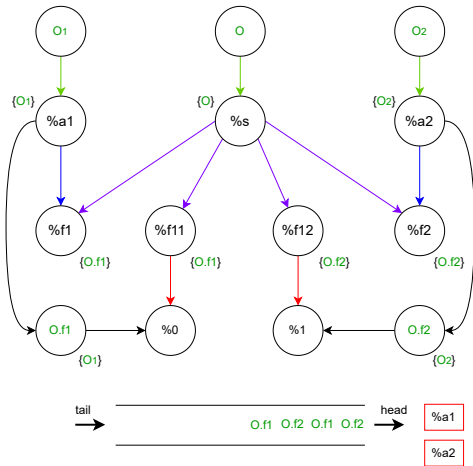
21 **foreach** $o \in pts(p)$ **do**

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

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

24 $pushIntoWorklist(x)$;

Field-Sensitive Andersen's Pointer Analysis



Algorithm 26: 1 Anderson'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** $o \in pts(p)$ **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** $p \xrightarrow{\text{Gep}} x \in E$ **do** // Gep rule

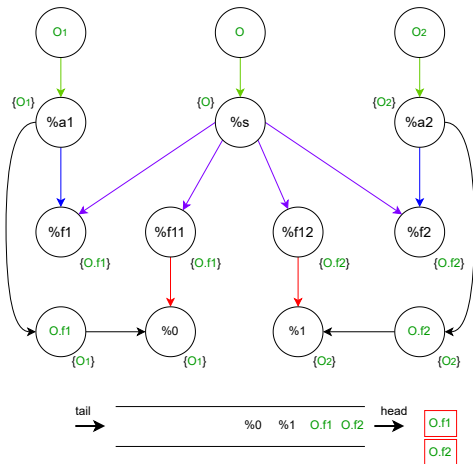
21 **foreach** $o \in pts(p)$ **do**

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 Anderson'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 $p \xrightarrow{\text{Gep}} x \in E$ do // Gep rule

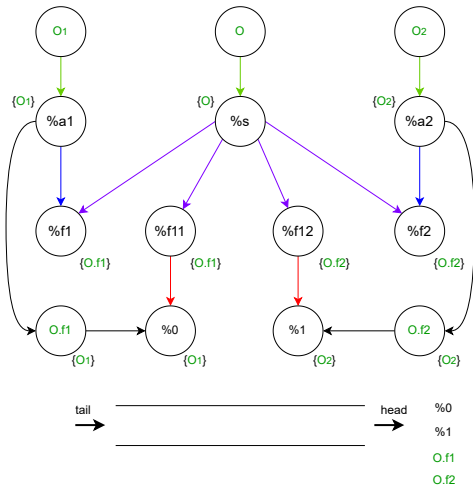
21 foreach $o \in \text{pts}(p)$ do

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

23 if pts(x) changed then

24 pushIntoWorklist(x);

Field-Sensitive Andersen's Pointer Analysis



Algorithm 28: 1 Anderson'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** $o \xrightarrow{Address} p \in E$ **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** $p \xrightarrow{Gep} x \in E$ **do** // Gep rule

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

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

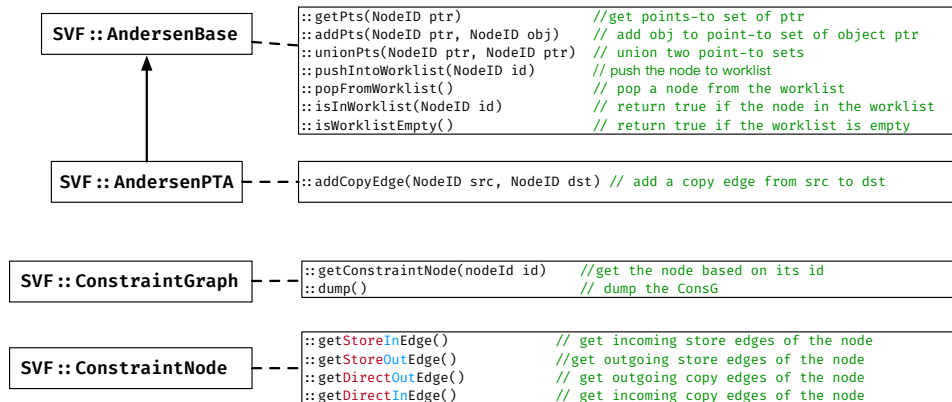
23 **if** $pts(x)$ *changed* **then**

24 $pushIntoWorklist(x)$;

Constraint solving Algorithm for Andersen's Analysis

- A worklist holds a list of constraint graph nodes for processing
- 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.
- The constraint solving stops when no points-to set of a pointer is changed.

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>

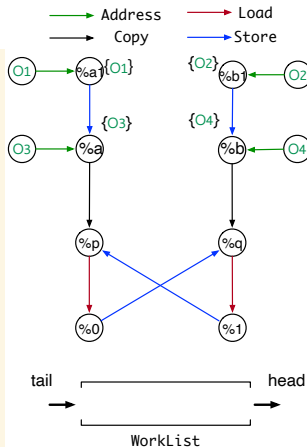
<https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API#constraintgraph-constraintnode-and-constrainededge>

Andersen's Pointer Analysis

Constraint graph before the while loop worklist solving

```
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 29: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **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**

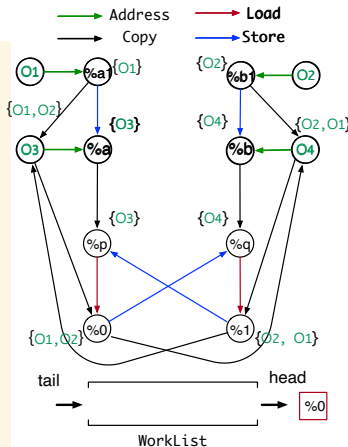
24 pushIntoWorkList(x);

Andersen's Pointer Analysis

Constraint graph after the while loop worklist solving

```
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 30: 1 Anderson'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** $p \xrightarrow{\text{Gep}} x \in E$ **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);

What's next?

- (1) Understand data-dependence in today's slides
- (2) Finish the quiz for Assignment-3
- (3) Implement Andersen's Pointer Analysis, i.e., coding task in Assignment 3
 - Refer to 'Assignment-3.pdf' on Canvas to know more about Assignment 3.

Information Flow Tracking

(Week 3)

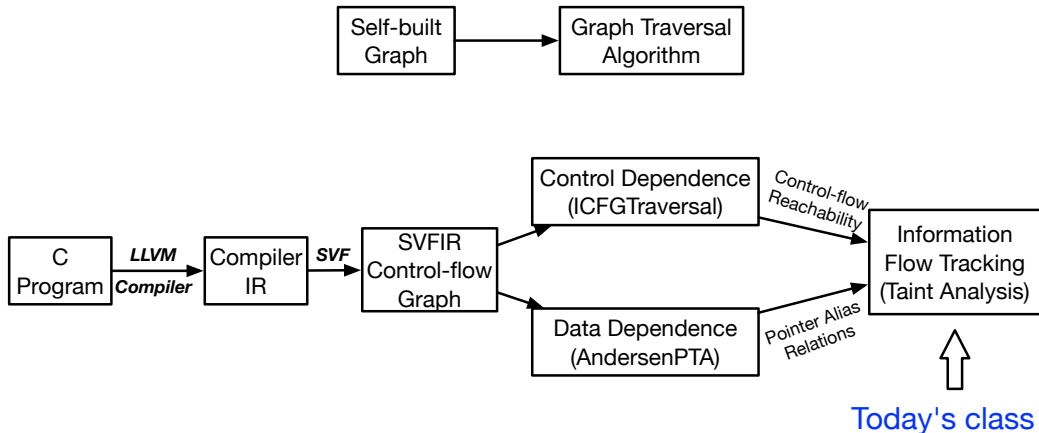
Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's Class

Lab Exercise 1



What is 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 subject**)
 - Dynamic taint analysis: taint tracking during runtime.

What is 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 subject**)
 - 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)
 - ...

How to Perform Static Taint Analysis?

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

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

How to Perform Static Taint Analysis?

Let us use what we have learned about control- and data-dependence 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 PAGNodes. 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:
 - (1) \mathbf{s}_{src} reaches \mathbf{s}_{snk} on the ICFG (**reachability**), and
 - (2) \mathbf{v}_{src} is aliased with \mathbf{v}_{snk} , i.e., $pts(v_{src}) \cap pts(v_{snk}) \neq \emptyset$ (**solveWorklist**)

Taint Analysis

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?

Taint Analysis

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 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?

Taint Analysis

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 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?

Taint Analysis

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 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?

Taint Analysis

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

Configuring Sources and Sinks for Taint Analysis

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

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