Data-Flow and Pointer Aliasing

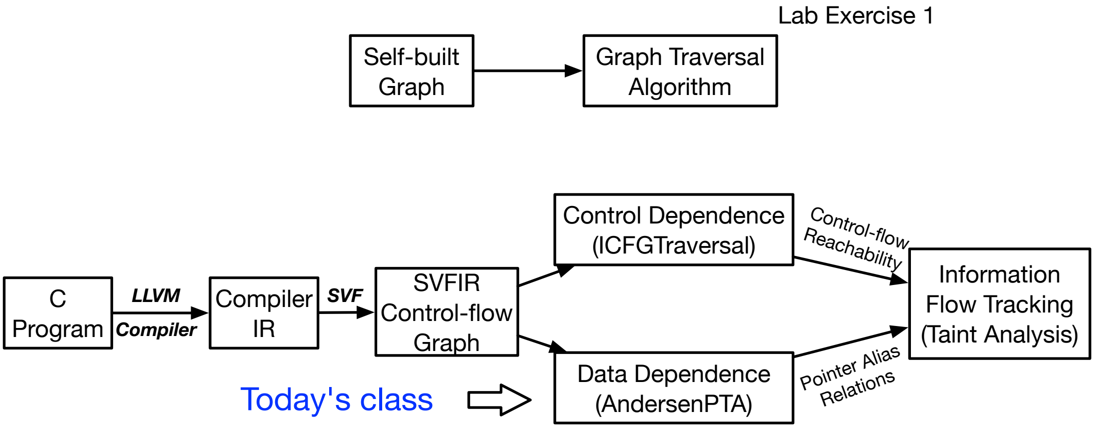
(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's class



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-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-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-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

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

Pointer analysis

- Points-to Analysis: aims to statically determine the possible runtime values of a pointer at compile-time.
  - Compute the *points-to set* (**a set of address-taken variables**) of each *pointer* (**top-level variable)**
  - For example, `p = &a; q = p;`
  - The resulting points-to sets of `p` and `q` are: `pts(p) = pts(q) =` $\{a\}$
- Alias Analysis: determine 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., `pts(p)` $\cap$ `pts(q) is not empty`) then `p` and `q` are aliases. The derereferences of `p` and `q` may refer to the same memory location.

Pointer analysis

Why shall we learn pointer analysis?

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

Pointer analysis

Why shall we learn pointer analysis?

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

Pointer analysis

Why shall we learn pointer analysis?

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

Pointer analysis

Why shall we learn pointer analysis?

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

Pointer analysis

Why shall we learn pointer analysis?

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

Precision Dimensions

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

Flow-insensitive analysis:

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

Context-insensitive analysis:

- Merges all calling contexts when analysing a program method

Path-insensitive analysis:

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

Flow-sensitive analysis:

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

Context-sensitive analysis:

- Distinguishes between different calling contexts of a program method

Path-sensitive analysis:

- Computes a solution per (abstract) program path.

Precision Dimensions

Levels of Abstractions

Assume x is a tainted value

p = x

p = y

flow-sensitivity

at which

program point

p is tainted?

```
foo(x)      foo(y)
  call ↘   ↙ call
     foo(p){

     }
```

context-sensitivity

under which

calling context

p is tainted?

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

path-sensitivity

along which

program path

p is tainted?

Andersen's Pointer analysis

Flow-, context-, and path-insensitive analysis

In this subject, we will practice **Andersen's analysis**[1], a **flow-insensitive, context-insensitive and path-insensitive pointer analysis** through analyzing the **Constraint Graph** of a program.

- One of the most popular and widely used pointer analyses
- Constraint solving, i.e., inclusion-based constraint solving between program variables (`ConstraintNode` in SVF)

Andersen's Pointer Analysis

An inclusion-based analysis operating on top of the constraint graph of a program. SVF transforms each LLVM instruction into a constraint edge connecting two nodes

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

Andersen's Pointer Analysis

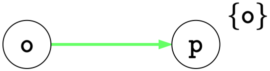
An inclusion-based analysis operating on top of the constraint graph of a program. SVF transforms each LLVM instruction into a constraint edge connecting two nodes

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

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

Andersen's Pointer Analysis

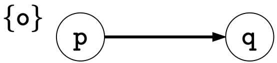
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	%ptr = $\texttt{alloca}_o$	$p = \texttt{alloc}$	$pts(p) = pts(p) \cup \{o\}$
CopyStmt	%p = bitcast %q	$p = q$	$pts(q) = pts(q) \cup pts(p)$
LoadStmt	%p = load %q	$p = *q$	$\forall o \in pts(p) : pts(q) = pts(o) \cup pts(q)$
StoreStmt	store %p, %q	$*p = q$	$\forall o \in pts(p) : pts(o) = pts(q) \cup pts(o)$



pts(p) = {o}

Andersen's Pointer Analysis

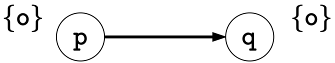
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	%ptr = $\mathtt{alloca}_o$	$p = \mathtt{alloc}$	$pts(p) = pts(p) \cup \{o\}$
CopyStmt	%p = bitcast %q	$p = q$	$pts(q) = pts(q) \cup pts(p)$
LoadStmt	%p = load %q	$p = *q$	$\forall o \in pts(p) : pts(q) = pts(o) \cup pts(q)$
StoreStmt	store %p, %q	$*p = q$	$\forall o \in pts(p) : pts(o) = pts(q) \cup pts(o)$



pts(p) = {o}

Andersen's Pointer Analysis

SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	`%ptr = alloca`$_o$	$p = \texttt{alloc}$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
CopyStmt	`%p = bitcast %q`	$p = q$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
LoadStmt	`%p = load %q`	$p = *q$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(o) \cup \texttt{pts}(q)$
StoreStmt	`store %p, %q`	$*p = q$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(o) = \texttt{pts}(q) \cup \texttt{pts}(o)$

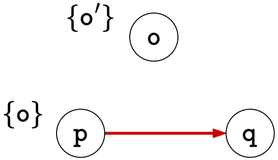


pts(p) = {o}

pts(q) = {o}

Andersen's Pointer Analysis

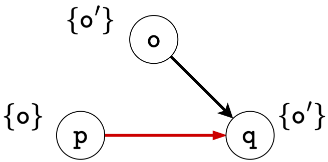
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	%ptr = alloca$_o$	$p = \texttt{alloc}$	$\texttt{pts}(p) = \texttt{pts}(p) \cup \{o\}$
CopyStmt	%p = bitcast %q	$p = q$	$\texttt{pts}(q) = \texttt{pts}(q) \cup \texttt{pts}(p)$
LoadStmt	%p = load %q	$p = *q$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(q) = \texttt{pts}(o) \cup \texttt{pts}(q)$
StoreStmt	store %p, %q	$*p = q$	$\forall o \in \texttt{pts}(p) : \texttt{pts}(o) = \texttt{pts}(q) \cup \texttt{pts}(o)$



pts(p) = {o}
pts(o) = {o′}

Andersen's Pointer Analysis

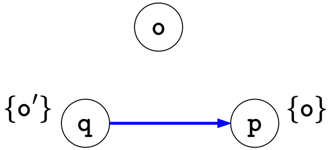
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	%ptr = alloca$_o$	$p = alloc$	$pts(p) = pts(p) \cup \{o\}$
CopyStmt	%p = bitcast %q	$p = q$	$pts(q) = pts(q) \cup pts(p)$
LoadStmt	%p = load %q	$p = *q$	$\forall o \in pts(p) : pts(q) = pts(o) \cup pts(q)$
StoreStmt	store %p, %q	$*p = q$	$\forall o \in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



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

Andersen's Pointer Analysis

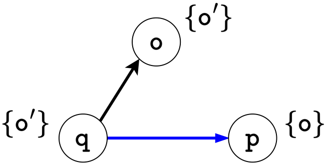
SVF IR	C code	LLVM IR	Constraint rules
AddrStmt	%ptr = alloca_o	$p = \text{alloc}$	$pts(p) = pts(p) \cup \{o\}$
CopyStmt	%p = bitcast %q	$p = q$	$pts(q) = pts(q) \cup pts(p)$
LoadStmt	%p = load %q	$p = *q$	$\forall o \in pts(p) : pts(q) = pts(o) \cup pts(q)$
StoreStmt	store %p, %q	$*p = q$	$\forall o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



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

Andersen's Pointer Analysis

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



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

Compiling a C Program to Its LLVM IR

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

swap.c

Compile

⇨

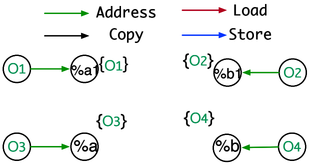
```llvm
define void @swap(ptr %p, ptr %q) #0 {
entry:
  %0 = load ptr, ptr %p, align 8
  %1 = load ptr, ptr %q, align 8
  store ptr %1, ptr %p, align 8
  store ptr %0, ptr %q, align 8
  ret void
}

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

swap.ll

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

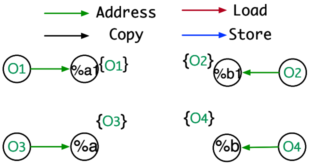
Construct a Constraint Graph from LLVM IR

```
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 a Constraint Graph from LLVM IR

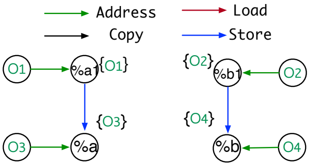



```
define i32 @main() #0 {
entry:
    %a1 = alloca i8, align 1          // O1
    %a = alloca ptr, align 8          // O2
    %b1 = alloca i8, align 1          // O3
    %b = alloca ptr, align 8          // O4
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr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

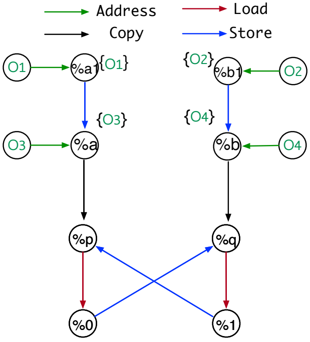
```
define i32 @main() #0 {
entry:
   %a1 = alloca i8, align 1                  // O1
   %a = alloca ptr, align 8                  // O2
   %b1 = alloca i8, align 1                  // O3
   %b = alloca ptr, align 8                  // O4
   store ptr %a1, ptr %a, align 8
   store ptr %b1, ptr %b, align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



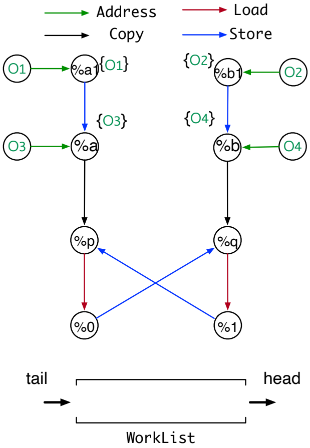
G = < V,E > // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes

foreach $o \xrightarrow{Address} p$ **do** // Address rule
    pts(p) = {o}
    pushIntoWorklist(p)
while WorkList ≠ ∅ do
  p <- popFromWorklist()
  foreach o ∈ pts(p) **do**
    foreach $q \xrightarrow{Store} p$ **do** // Store rule
      if $q \xrightarrow{Copy} o \notin E$ **then**
        $E \leftarrow E \cup \{ q \xrightarrow{Copy} o \}$ // Add copy edge
        pushIntoWorklist(q)
    foreach $p \xrightarrow{Load} r$ **do** // Load rule
      if $o \xrightarrow{Copy} r \notin E$ **then**
        $E \leftarrow E \cup \{ o \xrightarrow{Copy} r \}$ // Add copy edge
        pushIntoWorklist(o)
  foreach $p \xrightarrow{Copy} x \in E$ **do** // Copy rule
    pts(x) <- pts(x) ∪ pts(p)
    if pts(x) changed **then**
      pushIntoWorklist(x)

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

Construct a 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
}
```



G = < V,E > // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes

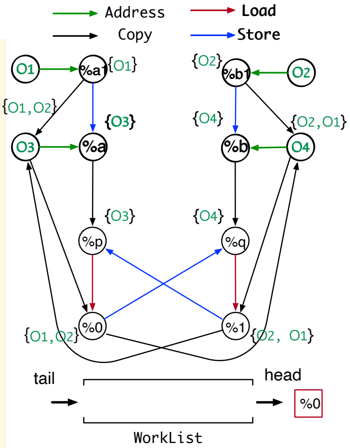
foreach o $\xrightarrow{\text{Address}}$ p **do** // Address rule
  pts(p) = {o}
  pushIntoWorklist(p)
while WorkList ≠ ∅ do
  p <- popFromWorklist()
  foreach o ∈ pts(p) **do**
    foreach q $\xrightarrow{\text{Store}}$ p **do** // Store rule
      if q $\xrightarrow{\text{Copy}}$ o ∉ E **then**
        E <- E ∪ { q $\xrightarrow{\text{Copy}}$ o } // Add copy edge
        pushIntoWorklist(q)
    foreach p $\xrightarrow{\text{Load}}$ r **do** // Load rule
      if o $\xrightarrow{\text{Copy}}$ r ∉ E **then**
        E <- E ∪ { o $\xrightarrow{\text{Copy}}$ r } // Add copy edge
        pushIntoWorklist(o)
  foreach p $\xrightarrow{\text{Copy}}$ x ∈ E **do** // Copy rule
    pts(x) <- pts(x) ∪ pts(p)
    if pts(x) changed **then**
      pushIntoWorklist(x)

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

Andersen's Pointer Analysis

Algorithm

```
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
}
```




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

Andersen's Pointer Analysis

Constraint solving Algorithm

- 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

SVF :: AndersenBase

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

SVF :: AndersenPTA (derived from SVF :: AndersenBase)

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

SVF :: ConstraintGraph

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

SVF :: ConstraintNode

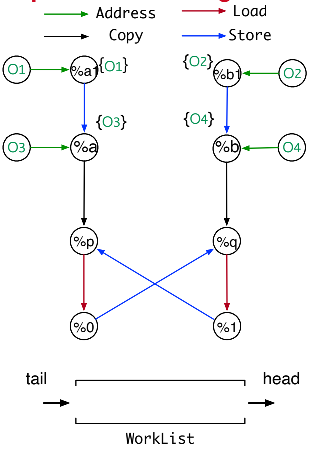
```
::getStoreInEdge()                    // get incoming store edges of the node
::getStoreOutEdge()                   //get outgoing store edges of the node
::getDirectOutEdge()                  // get outgoing copy edges of the node
::getDirectInEdge()                   // get incoming copy edges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

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

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
}
```

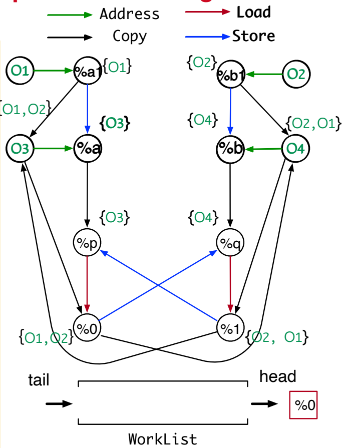



```
G = < V,E >  // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes
foreach  o --Address--> p  do                        // Address rule
      pts(p) = {o}
      pushIntoWorklist(p)
while WorkList ≠ ∅ do
   p <- popFromWorklist()
   foreach o ∈ pts(p)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

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
}
```



G = < V,E > // Constraint Graph
V: a set of nodes in graph
E: a set of edges in graph
WorkList: a vector of nodes

foreach o $\xrightarrow{Address}$ p **do** // Address rule
    pts(p) = {o}
    pushIntoWorklist(p)
while WorkList ≠ ∅ do
  p <- popFromWorklist()
  foreach o ∈ pts(p) **do**
    foreach q $\xrightarrow{Store}$ p **do** // Store rule
      if q $\xrightarrow{Copy}$ o ∉ E **then**
        E <- E ∪ { q $\xrightarrow{Copy}$ o } // Add copy edge
        pushIntoWorklist(q)
    foreach p $\xrightarrow{Load}$ r **do** // Load rule
      if o $\xrightarrow{Copy}$ r ∉ E **then**
        E <- E ∪ { o $\xrightarrow{Copy}$ r } // Add copy edge
        pushIntoWorklist(o)
  foreach p $\xrightarrow{Copy}$ x ∈ E **do** // Copy rule
    pts(x) <- pts(x) ∪ pts(p)
    if pts(x) changed **then**
      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

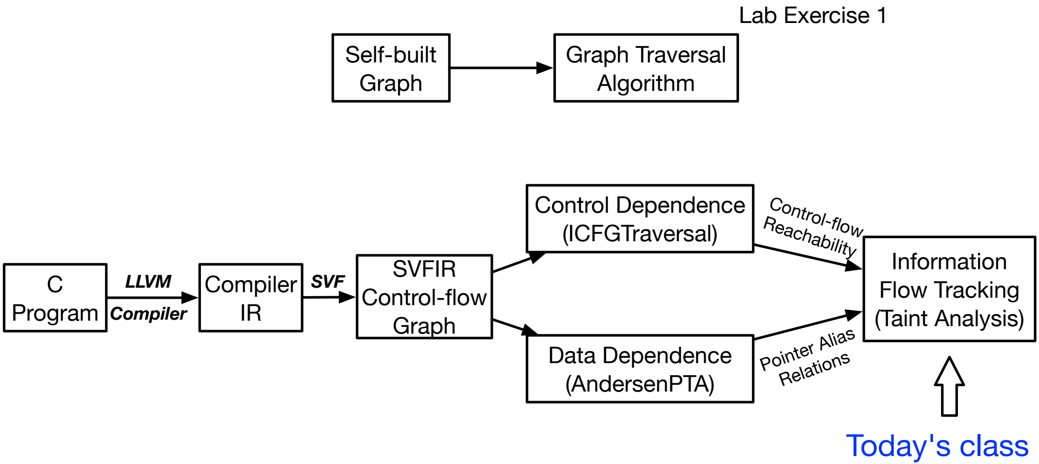
(Week 3)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's Class



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** $\mathbf{v_{src}}$@$\mathbf{s_{src}}$ 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`.

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 (**Assignment 2**), and
  - (2) $\mathbf{v_{src}}$ is aliased with $\mathbf{v_{snk}}$, i.e., $pts(v_{src}) \cap pts(v_{snk}) \neq \varnothing$ (**Assignment 3**)

Taint Analysis

Example 1

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

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

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

Do we have a tainted flow from source to sink?

Taint Analysis

Example 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

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

Do we have a tainted flow from source to sink?

Taint Analysis

Example 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

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

How many tainted flows from source to sink?

Taint Analysis

Example 4

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

How many tainted flows from source to sink?

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

Configuring Sources and Sinks for Taint Analysis

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

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

Configuring Sources and Sinks for Taint Analysis

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

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

Configuring Sources and Sinks for Taint Analysis

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

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