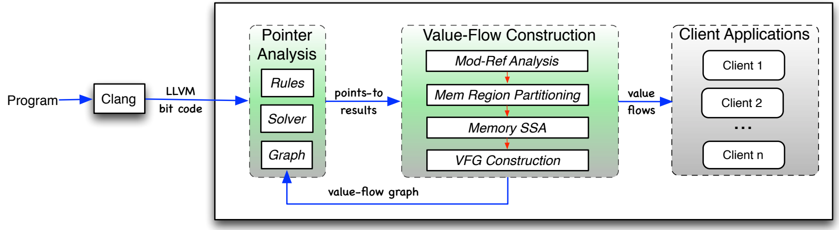
Interprocedural Value-Flow Graph

Yulei Sui

University of Technology Sydney, Australia

Static Value-Flow Analysis



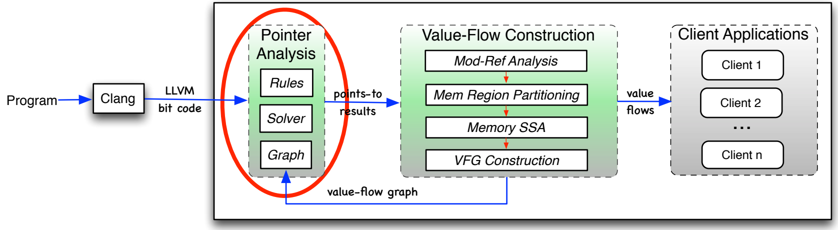
Static value-flow graph (VFG) resolves both the data and control dependence of a program. Three phrase to generate a VFG of a program: (1) **Pointer Analysis**, (2) **Interprocedural Memory SSA Form**, and (3) **Value-Flow Construction**.

SVF's command line to generate a VFG of a program (e.g., `swap.c`)[1]

- `clang -S -c -fno-discard-value-names -emit-llvm swap.c -o swap.ll`
- `wpa -ander -svfg -dump-vfg swap.ll`

[1] https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#12-value-flow-graph

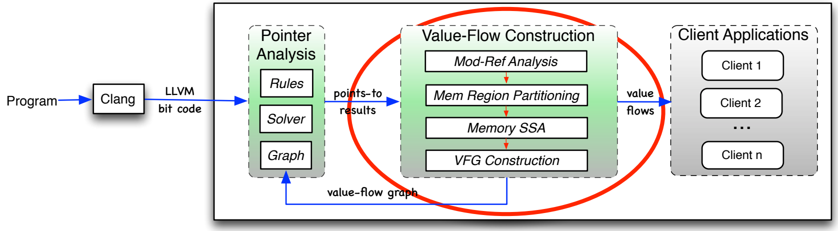
Static Value-Flow Analysis (Pointer Analysis)



- Support developing different analyses (flow-, context-, field-sensitivity[2])
  - **Graph** is a higher-level abstraction extracted from the LLVM IR indicating ***where*** pointer analysis should be performed.
  - **Rules** defines ***how*** to derive the points-to information from each statement,
  - **Solver** determines in ***what*** order to resolve all the constraints.

[2] More details can be found at `https://github.com/SVF-tools/SVF/wiki/Write-a-flow--and-field---insensitive-pointer-analysis`

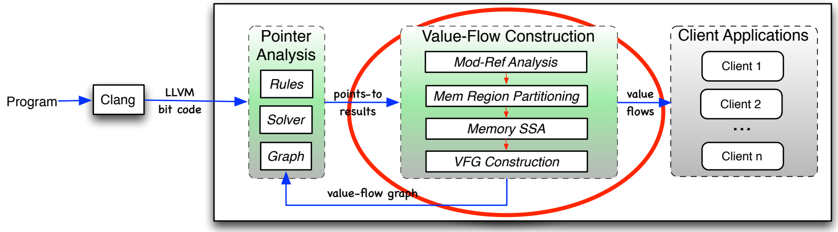
Static Value-Flow Analysis (Value-Flow Construction)



- Interprocedural memory SSA construction based on HSSA (CC '96[3]) and widely used in Open64.
  - **Side-Effect Annotation** at loads/stores and callsites
  - **Placing Memory SSA** ϕ for memory objects.
  - **SSA Renaming** for objects with different versions:

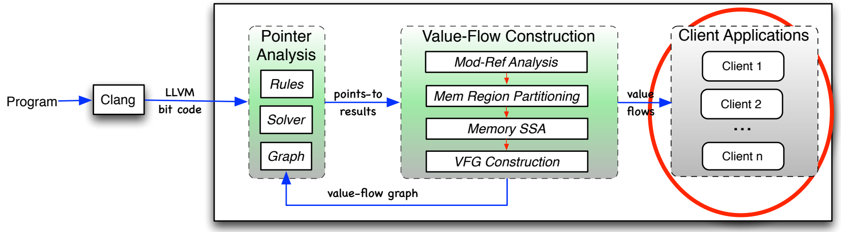
[3] F Chow, S Chan, SM Liu, R Lo, M Streich, *Effective representation of aliases and indirect memory operations in SSA form*, CC 1996

Static Value-Flow Analysis (Value-Flow Construction)



- Value-Flow Construction:
  - **Direct Value-Flows**: def-use of top-level pointers
  - **Indirect Value-Flows**: def-use of address-taken variables based on memory SSA

Static Value-Flow Analysis (Clients)



- Detecting memory errors
  - **Memory leaks**
  - **Use-after-frees**
  - **Null pointers**
  - **. . .**
- Code embedding:
  - **Code summarization**
  - **Method name prediction**
  - **. . .**

Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive). SVF transforms LLVM instructions into a Constraint Graph (`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

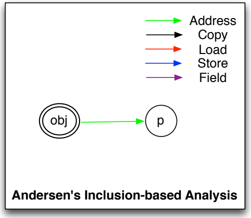
Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	p = &q $\rightarrow$ f

Andersen’s Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	p = &q $\rightarrow$ f

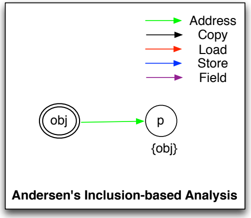


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	$p = \&q \rightarrow f$

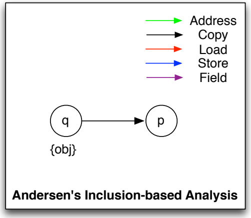


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	$p = \&obj$
Copy	$p = q$
Load	$p = *q$
Store	$*p = q$
Field	$p = \&q \rightarrow f$

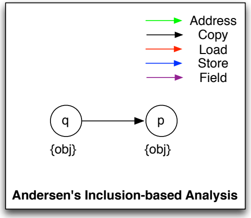


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	$p = \&obj$
Copy	$p = q$
Load	$p = {*q}$
Store	${*p} = q$
Field	$p = \&q \rightarrow f$

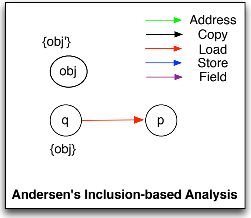


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	$p = \&q \rightarrow f$



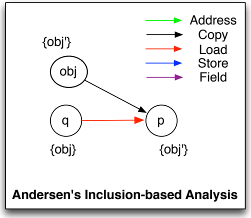
Andersen's Inclusion-based Analysis

Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	$p = \&obj$
Copy	$p = q$
Load	$p = *q$
Store	$*p = q$
Field	$p = \&q \rightarrow f$



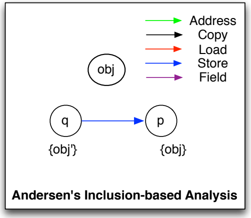
Andersen's Inclusion-based Analysis

Andersen’s Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive). SVF transforms LLVM instructions into a Constraint Graph (`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	p = &q $\rightarrow$ f

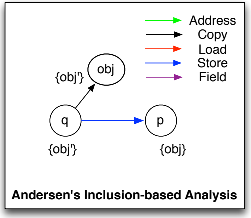


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	$p = \&q \rightarrow f$

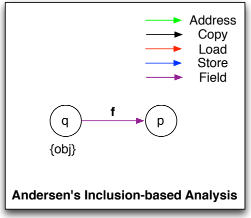


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	p = &q $\rightarrow$ f

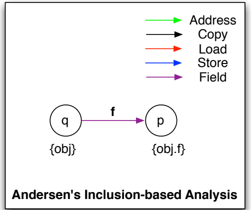


Andersen's Pointer Analysis

A popular inclusion-based pointer analysis (flow-insensitive and field-sensitive).
SVF transforms LLVM instructions into a Constraint Graph
(`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = &obj
Copy	p = q
Load	p = *q
Store	*p = q
Field	$p = \&q \rightarrow f$



Andersen’s Pointer Analysis

SVF transforms LLVM instructions into a graph representation Constraint Graph (`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = alloc_{obj}
Copy	p = q
Load	p = *q
Store	*p = q
Field	p = q *gep* f

Andersen's Pointer Analysis

SVF transforms LLVM instructions into a graph representation Constraint Graph (`https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ConsG.h`)

- Node:
  - A pointer: (LLVM Value in pointer type)
  - An object: (heap, stack, global, function)
- Edge: A Constraint between two nodes

Address	p = alloc_{obj}	$\{obj\} \subseteq \text{Pts}(p)$
Copy	p = q	$\text{Pts}(q) \subseteq \text{Pts}(p)$
Load	p = *q	$\forall o \in \text{Pts}(q), \text{Pts}(o) \subseteq \text{Pts}(p)$
Store	*p = q	$\forall o \in \text{Pts}(p), \text{Pts}(q) \subseteq \text{Pts}(o)$
Field	p = q *gep* f	$\forall o \in \text{Pts}(q), \{o.f\} \subseteq \text{Pts}(p)$

Andersen’s Pointer Analysis

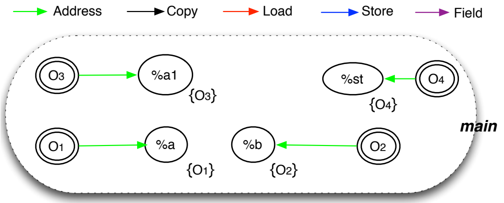
```c
struct st{
        char f1;
        char f2;
};
typedef struct st ST;

int main(){
        char a1; ST st;
        char *a = &a1;
        char *b = &(st.f2);
        swap(&a,&b);
}
void swap(char **p, char **q){
        char* t = *p;
        *p = *q;
        *q = t;
}
```

```llvm
define i32 @main() {
entry:
        %a = alloca i8*, align 8                    // O1
        %b = alloca i8*, align 8                    // O2
        %a1 = alloca i8, align 1                    // O3
        %st = alloca %struct.st, align 1            // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

Andersen's Pointer Analysis

```
define i32 @main() {
entry:
      %a = alloca i8*, align 8              // O1
      %b = alloca i8*, align 8              // O2
      %a1 = alloca i8, align 1              // O3
      %st = alloca %struct.st, align 1      // O4
      store i8* %a1, i8** %a, align 8
      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
      store i8* %f2, i8** %b, align 8
      call void @swap(i8** %a, i8** %b)
      ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR

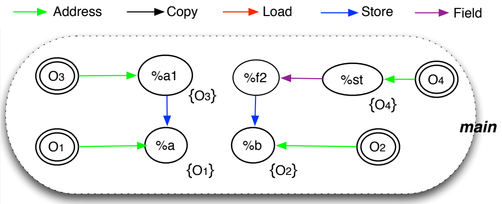


Constraint Graph

Andersen’s Pointer Analysis

```
define i32 @main() {
entry:
      %a = alloca i8*, align 8                  // O1
      %b = alloca i8*, align 8                  // O2
      %a1 = alloca i8, align 1                  // O3
      %st = alloca %struct.st, align 1          // O4
      store i8* %a1, i8** %a, align 8
      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
      store i8* %f2, i8** %b, align 8
      call void @swap(i8** %a, i8** %b)
      ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR

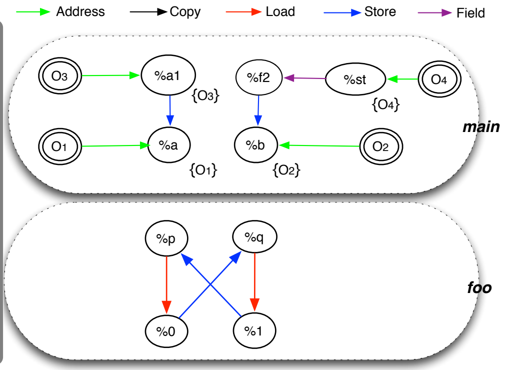


Constraint Graph

Andersen’s Pointer Analysis

```
define i32 @main() {
entry:
      %a = alloca i8*, align 8                 // O1
      %b = alloca i8*, align 8                 // O2
      %a1 = alloca i8, align 1                 // O3
      %st = alloca %struct.st, align 1         // O4
      store i8* %a1, i8** %a, align 8
      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
      store i8* %f2, i8** %b, align 8
      call void @swap(i8** %a, i8** %b)
      ret i32 0
}
 define void @swap(i8** %p, i8** %q) {
 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
}
```

LLVM IR

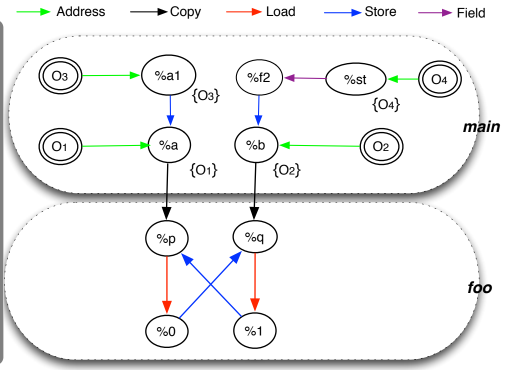


Constraint Graph

Andersen's Pointer Analysis

```
define i32 @main() {
entry:
      %a = alloca i8*, align 8              // O1
      %b = alloca i8*, align 8              // O2
      %a1 = alloca i8, align 1              // O3
      %st = alloca %struct.st, align 1      // O4
      store i8* %a1, i8** %a, align 8
      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
      store i8* %f2, i8** %b, align 8
      call void @swap(i8** %a, i8** %b)
      ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR

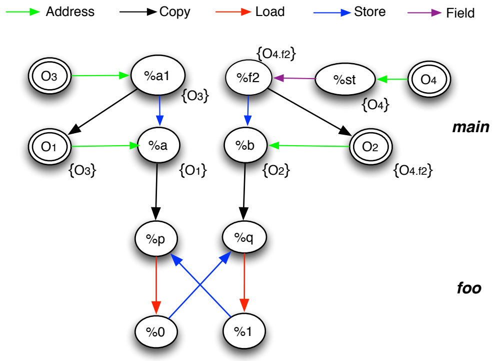


Constraint Graph

Andersen’s Pointer Analysis

```
define i32 @main() {
entry:
    %a = alloca i8*, align 8             // O1
    %b = alloca i8*, align 8             // O2
    %a1 = alloca i8, align 1             // O3
    %st = alloca %struct.st, align 1     // O4
    store i8* %a1, i8** %a, align 8
    %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
    store i8* %f2, i8** %b, align 8
    call void @swap(i8** %a, i8** %b)
    ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR

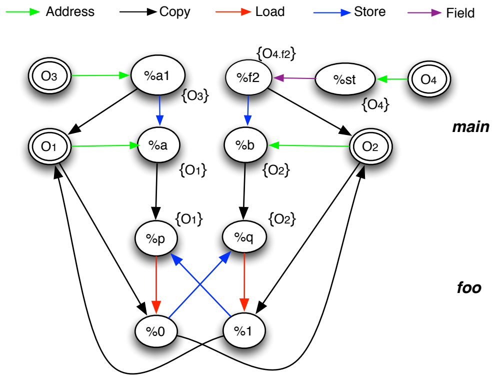


Constraint Graph

Andersen’s Pointer Analysis

```
define i32 @main() {
entry:
        %a = alloca i8*, align 8            // O1
        %b = alloca i8*, align 8            // O2
        %a1 = alloca i8, align 1            // O3
        %st = alloca %struct.st, align 1    // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR

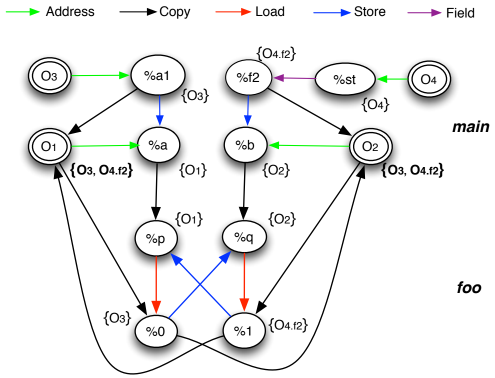


Constraint Graph

Andersen's Pointer Analysis

```
define i32 @main() {
entry:
       %a = alloca i8*, align 8                    // O1
       %b = alloca i8*, align 8                    // O2
       %a1 = alloca i8, align 1                    // O3
       %st = alloca %struct.st, align 1            // O4
       store i8* %a1, i8** %a, align 8
       %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
       store i8* %f2, i8** %b, align 8
       call void @swap(i8** %a, i8** %b)
       ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

LLVM IR



Constraint Graph

Constraint solving techniques: Wave-Deep Propagation, HCD, LCD. More details can be found here

Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.

Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.

- Side-effect annotation.
  - **Load**: $p = *q$ is annotated with a $\mu(o)$ for each variable $o \in \mathrm{Pts}(q)$.
  - **Store**: $*p = q$ is annotated with a $o = \chi(o)$ for each variable $o \in \mathrm{Pts}(p)$.
  - **Callsite**: *foo*(...) is annotated with $\mu(o)/\chi(o)$ if o is referred or modified inside caller *foo*.
  - **Function entry/exit**: $\chi(o)/\mu(o)$ is annotated at the entry of a function (e.g., foo) if o is referred or modified in *foo*.

Memory SSA

Memory SSA is constructed per procedure given the global points-to information after region partitioning.

- Side-effect annotation.
  - **Load**: $p = *q$ is annotated with a $\mu(o)$ for each variable $o \in \text{Pts}(q)$.
  - **Store**: $*p = q$ is annotated with a $o = \chi(o)$ for each variable $o \in \text{Pts}(p)$.
  - **Callsite**: $foo(...)$ is annotated with $\mu(o)/\chi(o)$ if o is referred or modified inside caller *foo*.
  - **Function entry/exit**: $\chi(o)/\mu(o)$ is annotated at the entry of a function (e.g., foo) if o is referred or modified in *foo*.
- Memory SSA construction
  - **Placing Memory SSA** ϕ for memory objects.
  - **Renaming** objects with different versions:
    - $\mu(o)$ is treated as a use of o.
    - $o = \chi(o)$ is treated as both a def and a use of o.

Memory SSA

```
define i32 @main() {
entry:
        %a = alloca i8*, align 8                // O1
        %b = alloca i8*, align 8                // O2
        %a1 = alloca i8, align 1                // O3
        %st = alloca %struct.st, align 1        // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr ... %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
 }
 define void @swap(i8** %p, i8** %q) {
 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
}
```

LLVM IR

Annotation

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8            // O1
 %b = alloca i8*, align 8            // O2
 %a1 = alloca i8, align 1            // O3
 %st = alloca %struct.st, align 1    // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr ... %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

Memory SSA

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8            // O1
 %b = alloca i8*, align 8            // O2
 %a1 = alloca i8, align 1            // O3
 %st = alloca %struct.st, align 1    // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR of the example code

Pre-computed Points-to:

pt(%a) = pt(%p) = {O1}
pt(%b) = pt(%q) = {O2}

Memory Region:

MR2: O2
MR1: O1

Annotated CHIs at stores

Memory SSA

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8           // O1
 %b = alloca i8*, align 8           // O2
 %a1 = alloca i8, align 1           // O3
 %st = alloca %struct.st, align 1  // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR of the example code

Pre-computed Points-to:

pt(%a) = pt(%p) = {O1}
pt(%b) = pt(%q) = {O2}

Memory Region:

MR2: O2
MR1: O1

Annotated MUs at loads

Memory SSA

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8           // O1
 %b = alloca i8*, align 8           // O2
 %a1 = alloca i8, align 1           // O3
 %st = alloca %struct.st, align 1   // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR of the example code

Pre-computed Points-to:

pt(%a) = pt(%p) = {O1}
pt(%b) = pt(%q) = {O2}

Memory Region:

MR2: O2
MR1: O1

Annotated MUs/CHIs at callsite

Memory SSA

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8          // O1
 %b = alloca i8*, align 8          // O2
 %a1 = alloca i8, align 1          // O3
 %st = alloca %struct.st, align 1  // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR of the example code

Pre-computed Points-to:

pt(%a) = pt(%p) = {O1}
pt(%b) = pt(%q) = {O2}

Memory Region:

MR2: O2
MR1: O1

Annotated MUs/CHIs at Function entry/exit

Memory SSA

```
define i32 @main() {
entry:
        %a = alloca i8*, align 8                 // O1
        %b = alloca i8*, align 8                 // O2
        %a1 = alloca i8, align 1                 // O3
        %st = alloca %struct.st, align 1         // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr ... %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
 }
 define void @swap(i8** %p, i8** %q) {
 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
}
```

LLVM IR

Annotation

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8              // O1
 %b = alloca i8*, align 8              // O2
 %a1 = alloca i8, align 1              // O3
 %st = alloca %struct.st, align 1      // O4

 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)

 %f2 = getelementptr ... %st, i32 0, i32 1
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)

CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)

 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8

LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8

 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)

 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)

 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

Interprocedural Value-Flow Construction (Nodes and Edges)

Given annotated μ and χ functions, its VFG is constructed by connecting the definition of each SSA variable with its uses. Each node in the VFG represents one of the following:

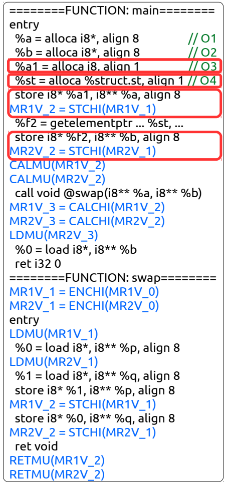
- **Statement VFGNodes**: A definition of a variable at a non-call statement ℓ:
  - COPY ($\ell : p = q$): $p@\ell$;
  - PHI ($\ell : v_3 = \phi(v_2, v_1)$): $v_3@\ell$;
  - GEP ($\ell : p = \&q \rightarrow f$): $p@\ell$.
  - LOAD ($\ell : p = *q\ [\mu(o)]$): $p@\ell$;
  - STORE ($\ell : *p = q\ [o_2 = \chi(o_1)]$): $o_2@\ell$.
- **CallSite VFGNodes**: A variable passing or defined at a callsite $\ell_{cs} : r = f(..., p, ...)\ [\mu(o')]\ [o = \chi(\_)]$:
  - ACTUALPARM (callsite actual parameter): $p@\ell_{cs}$.
  - ACTUALRET (value directly returned): $r@\ell_{cs}$;
  - ACTUALIN (value indirectly passed into callee f) $o'@\ell_{cs}$.
  - ACTUALOUT (value indirectly returned from callee f): $o@\ell_{cs}$.

Interprocedural Value-Flow Construction (Nodes and Edges)

- **FunctionEntry/Exit VFNodes**: A variable defined at the entry or returned at the exit of a function $f(..., p, ...)\{[o = \chi(\_)] \; ... \; [\mu(o')] \; \textit{return } q; \}$:
  - FORMALPARM (parameter directly initialized): $p@\ell_f$.
  - FORMALRET (formal return): $q@\ell_{\underline{f}}$.
  - FORMALIN (parameter indirectly initialized): $o@\ell_f$.
  - FORMALOUT (value indirectly modified in callee): $o'@\ell_{\underline{f}}$.

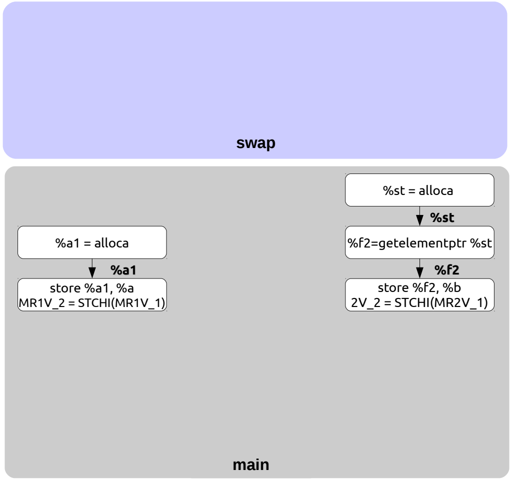
Rule	VFG Nodes	VFG Edges
COPY	$\ell : p = q$	$p@\ell \hookleftarrow q@\ell'$
PHI	$\ell : v_3 = \phi(v_1, v_2)$	$v_3@\ell \hookleftarrow v_1@\ell'$ $\quad v_3@\ell \hookleftarrow v_2@\ell''$
LOAD	$\ell : p = *q \; [\mu(o)]$	$p@\ell \hookleftarrow o@\ell'$
STORE	$\ell : *p = q \; [o_2 = \chi(o_1)]$	$o_2@\ell \hookleftarrow q@\ell'$ $\quad o_2@\ell \hookleftarrow o_1@\ell''$
CALL	$\ell_{cs} : r = f(..., p, ...) \quad [\mu(o_1)] \quad [o_2 = \chi(\_)]$ $\ell_f : f(..., q, ...)\{[o_3 = \chi(\_)] \quad ... \quad [\mu(o_4)] \; \mathrm{return} \; x\}$	$q@\ell_f \hookleftarrow p@\ell_1$ $\quad r@\ell_{cs} \hookleftarrow x@\ell_2$ $o_3@\ell_f \hookleftarrow o_1@\ell_3$ $\quad o_2@\ell_{cs} \hookleftarrow o_4@\ell_4$

Interprocedural Value-Flow Graph (An Example)



```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8            // O1
  %b = alloca i8*, align 8            // O2
  %a1 = alloca i8, align 1            // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

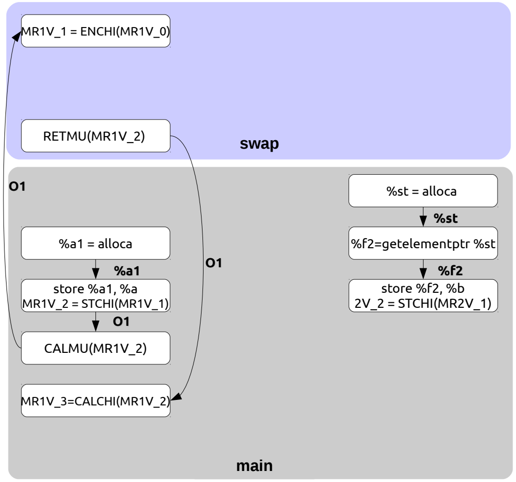


Value-Flow Graph

Interprocedural Value-Flow Graph (An Example)

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8           // O1
  %b = alloca i8*, align 8           // O2
  %a1 = alloca i8, align 1           // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

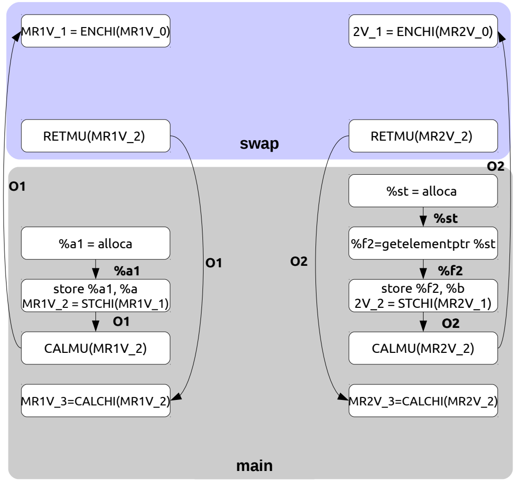


Value-Flow Graph

Interprocedural Value-Flow Graph (An Example)

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8          // O1
  %b = alloca i8*, align 8          // O2
  %a1 = alloca i8, align 1          // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

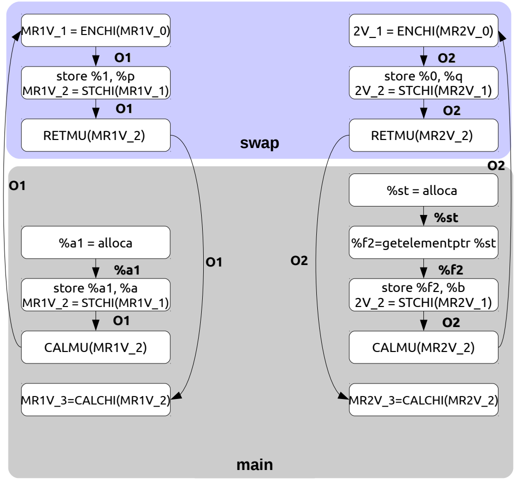


Value-Flow Graph

Interprocedural Value-Flow Graph (An Example)

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8              // O1
  %b = alloca i8*, align 8              // O2
  %a1 = alloca i8, align 1              // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

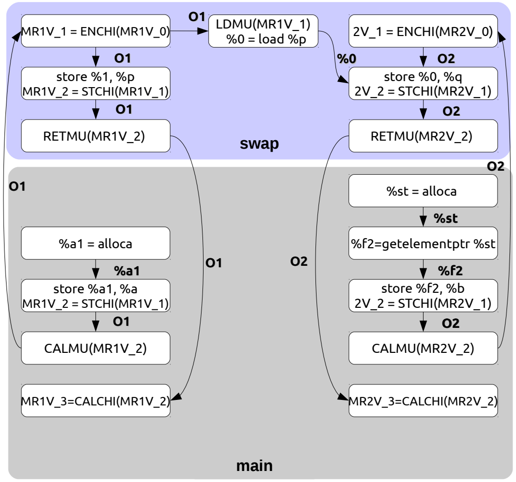


Value-Flow Graph

Interprocedural Value-Flow Graph (An Example)

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8               // O1
  %b = alloca i8*, align 8               // O2
  %a1 = alloca i8, align 1               // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

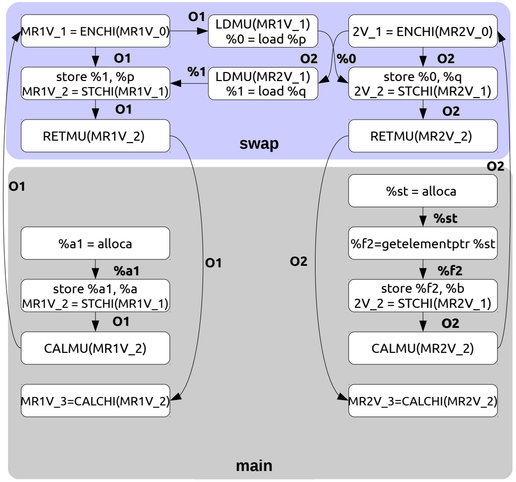


Value-Flow Graph

Interprocedural Value-Flow Graph (An Example)

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8              // O1
  %b = alloca i8*, align 8              // O2
  %a1 = alloca i8, align 1              // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR



Value-Flow Graph

15

Demand-Driven Analysis via Value-Flows

```c
struct st{
        char f1;
        char f2;
};
typedef struct st ST;

int main(){
        char a1; ST st;
        char *a = &a1;
        char *b = &(st.f2);
        swap(&a,&b);
}
void swap(char **p, char **q){
        char* t = *p;
        *p = *q;
        *q = t;
}
```

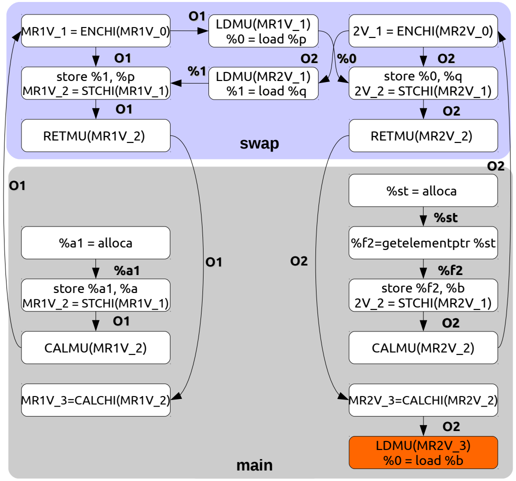
Query: b ?

```llvm
define i32 @main() {
entry:
        %a = alloca i8*, align 8                // O1
        %b = alloca i8*, align 8                // O2
        %a1 = alloca i8, align 1                // O3
        %st = alloca %struct.st, align 1        // O4
        store i8* %a1, i8** %a, align 8
        %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
        store i8* %f2, i8** %b, align 8
        call void @swap(i8** %a, i8** %b)
        ret i32 0
}
define void @swap(i8** %p, i8** %q) {
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
}
```

Demand-Driven Analysis via Value-Flows

```
========FUNCTION: main========
entry
 %a = alloca i8*, align 8            // O1
 %b = alloca i8*, align 8            // O2
 %a1 = alloca i8, align 1            // O3
 %st = alloca %struct.st, align 1 // O4
 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
 %f2 = getelementptr ... %st, ...
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
 %0 = load i8*, i8** %b
 ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8
 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR

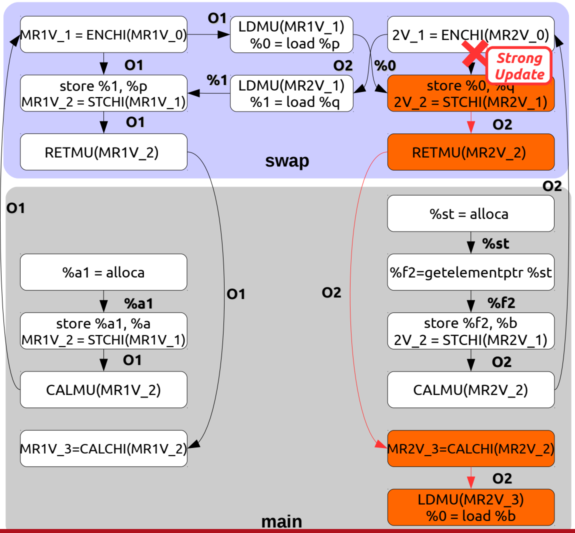


Value-Flow Graph

Demand-Driven Analysis via Value-Flows

```
========FUNCTION: main========
entry
  %a = alloca i8*, align 8              // O1
  %b = alloca i8*, align 8              // O2
  %a1 = alloca i8, align 1              // O3
  %st = alloca %struct.st, align 1 // O4
  store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
  %f2 = getelementptr ... %st, ...
  store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
  call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
  %0 = load i8*, i8** %b
  ret i32 0
========FUNCTION: swap========
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
  %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
  %1 = load i8*, i8** %q, align 8
  store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
  store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
  ret void
RETMU(MR1V_2)
```

Annotated IR

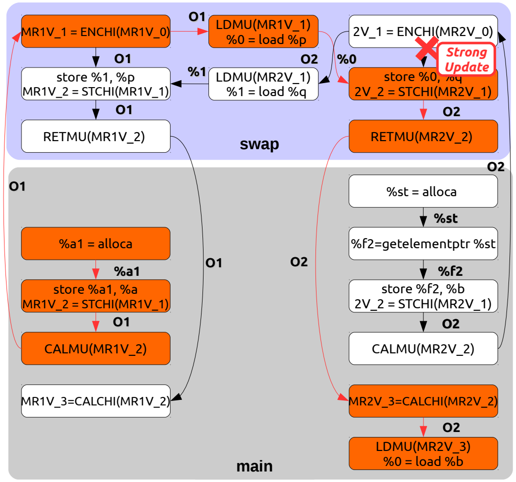


Value-Flow Graph

Demand-Driven Analysis via Value-Flows

```
=======FUNCTION: main=======
entry
 %a = alloca i8*, align 8            // O1
 %b = alloca i8*, align 8            // O2
 %a1 = alloca i8, align 1            // O3
 %st = alloca %struct.st, align 1 // O4
 store i8* %a1, i8** %a, align 8
MR1V_2 = STCHI(MR1V_1)
 %f2 = getelementptr ... %st, ...
 store i8* %f2, i8** %b, align 8
MR2V_2 = STCHI(MR2V_1)
CALMU(MR1V_2)
CALMU(MR2V_2)
 call void @swap(i8** %a, i8** %b)
MR1V_3 = CALCHI(MR1V_2)
MR2V_3 = CALCHI(MR2V_2)
LDMU(MR2V_3)
 %0 = load i8*, i8** %b
 ret i32 0
=======FUNCTION: swap=======
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
entry
LDMU(MR1V_1)
 %0 = load i8*, i8** %p, align 8
LDMU(MR2V_1)
 %1 = load i8*, i8** %q, align 8
 store i8* %1, i8** %p, align 8
MR1V_2 = STCHI(MR1V_1)
 store i8* %0, i8** %q, align 8
MR2V_2 = STCHI(MR2V_1)
 ret void
RETMU(MR1V_2)
RETMU(MR2V_2)
```

Annotated IR



Value-Flow Graph