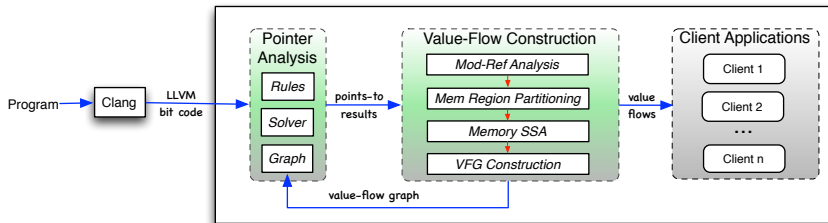


Interprocedural Value-Flow Graph

Yulei Sui

University of Technology Sydney, Australia

Static Value-Flow Analysis



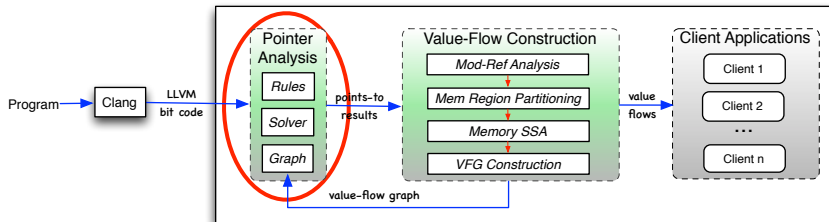
Static value-flow graph (VFG) resolves both the data and control dependence of a program. Three phrase to generate a VFG of a program: (1) **Pointer Analysis**, (2) **Interprocedural Memory SSA Form**, and (3) **Value-Flow Construction**.

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>

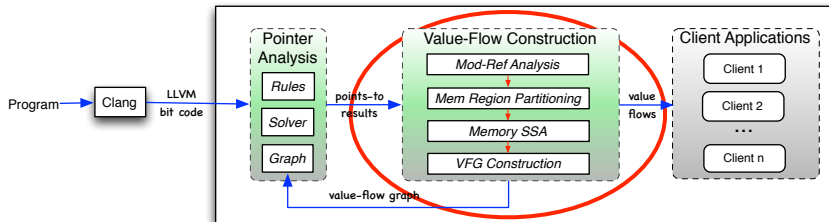
Static Value-Flow Analysis (Pointer Analysis)



- Support developing different analyses (flow-, context-, field-sensitivity²)
 - **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>

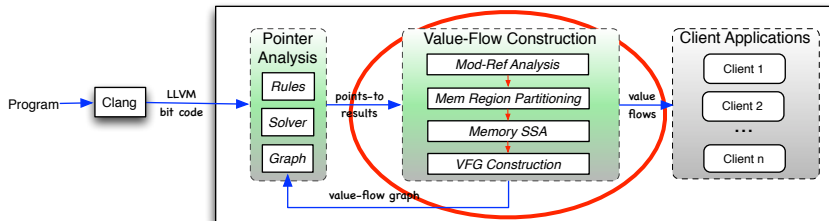
Static Value-Flow Analysis (Value-Flow Construction)



- Interprocedural memory SSA construction based on HSSA (CC '96³) 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:

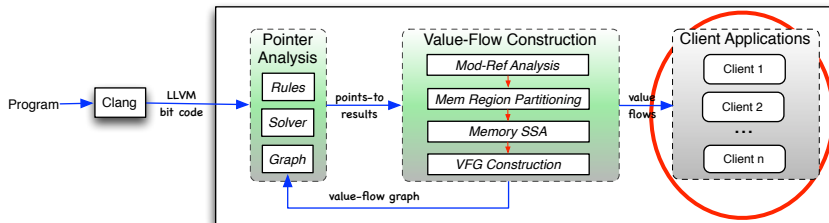
³F Chow, S Chan, SM Liu, R Lo, M Streich, *Effective representation of aliases and indirect memory operations in SSA form*, CC 1996

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 $p = \&q \rightarrow f$

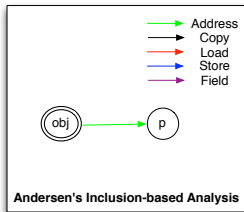
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



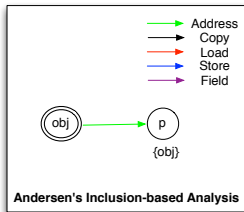
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



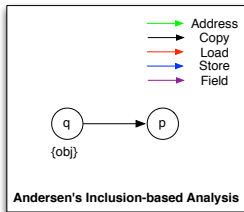
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



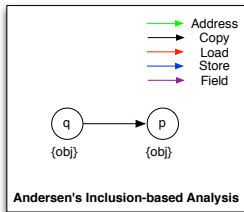
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



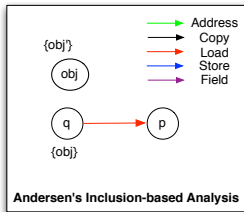
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



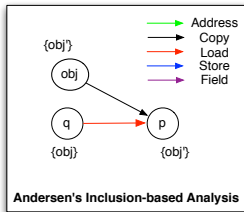
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



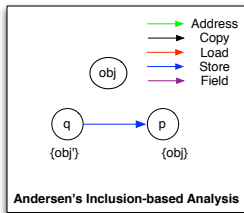
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



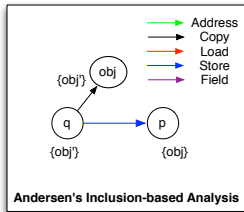
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



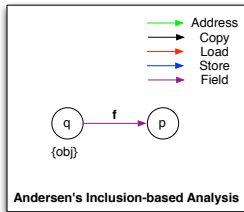
Andersen's Pointer Analysis

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	$p = \&q \rightarrow f$



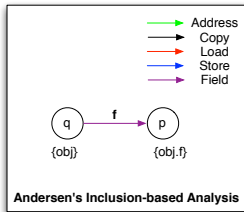
Andersen's Pointer Analysis

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

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 $p = \text{alloc}_{obj}$

Copy $p = q$

Load $p = *q$

Store $*p = q$

Field $p = q \text{ gep } f$

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	$p = \text{alloc}_{obj}$	$\{obj\} \subseteq \text{Pts}(p)$
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	$p = q \text{ gep } f$	$\forall o \in \text{Pts}(q), \{o.f\} \subseteq \text{Pts}(p)$

Andersen's Pointer Analysis

```
1 struct st{
2     char f1;
3     char f2;
4 };
5 typedef struct st ST;
6
7 int main(){
8     char a1; ST st;
9     char *a = &a1;
10    char *b = &(st.f2);
11    swap(&a,&b);
12 }
13 void swap(char **p, char **q){
14     char *t = *p;
15     *p = *q;
16     *q = t;
17 }
```

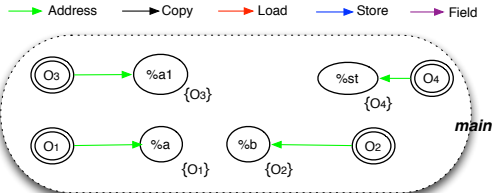


```
1 define i32 @main() {
2     entry:
3         %a = alloca i8*, align 8           // O1
4         %b = alloca i8*, align 8           // O2
5         %a1 = alloca i8, align 1           // O3
6         %st = alloca %struct.st, align 1    // O4
7         store i8* %a1, i8** %a, align 8
8         %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
9         store i8* %f2, i8** %b, align 8
10        call void @swap(i8** %a, i8** %b)
11        ret i32 0
12    }
13    define void @swap(i8** %p, i8** %q) {
14        entry:
15            %0 = load i8** %p, align 8
16            %1 = load i8** %q, align 8
17            store i8* %1, i8** %p, align 8
18            store i8* %0, i8** %q, align 8
19            ret void
20    }
```

Andersen's Pointer Analysis

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12  }  
13  define void @swap(i8** %p, i8** %q) {  
14    entry:  
15    %0 = load i8** %p, align 8  
16    %1 = load i8** %q, align 8  
17    store i8* %1, i8** %p, align 8  
18    store i8* %0, i8** %q, align 8  
19    ret void  
20  }
```

LLVM IR

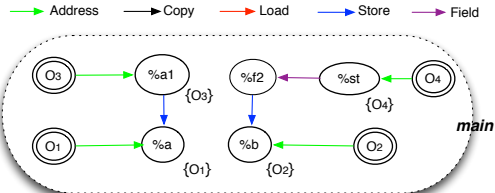


Constraint Graph

Andersen's Pointer Analysis

```
1  define i32 @main() {  
2  entry:  
3      %a = alloca i8*, align 8           // O1  
4      %b = alloca i8*, align 8           // O2  
5      %a1 = alloca i8, align 1           // O3  
6      %st = alloca %struct.st, align 1    // O4  
7      store i8* %a1, i8** %a, align 8  
8      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9      store i8* %f2, i8** %b, align 8  
10     call void @swap(i8** %a, i8** %b)  
11     ret i32 0  
12 }  
13 define void @swap(i8** %p, i8** %q) {  
14 entry:  
15     %0 = load i8** %p, align 8  
16     %1 = load i8** %q, align 8  
17     store i8* %1, i8** %p, align 8  
18     store i8* %0, i8** %q, align 8  
19     ret void  
20 }
```

LLVM IR

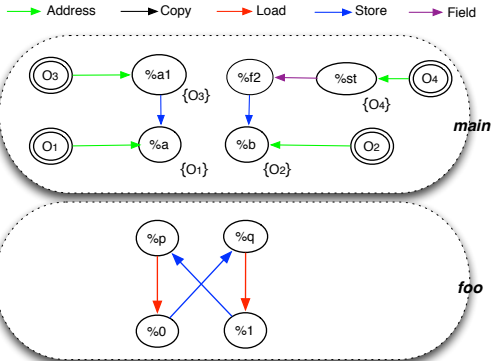


Constraint Graph

Andersen's Pointer Analysis

```
1  define i32 @main() {  
2  entry:  
3      %a = alloca i8*, align 8           // O1  
4      %b = alloca i8*, align 8           // O2  
5      %a1 = alloca i8, align 1           // O3  
6      %st = alloca %struct.st, align 1    // O4  
7      store i8* %a1, i8** %a, align 8  
8      %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9      store i8* %f2, i8** %b, align 8  
10     call void @swap(i8** %a, i8** %b)  
11     ret i32 0  
12 }  
13 define void @swap(i8** %p, i8** %q) {  
14 entry:  
15     %0 = load i8** %p, align 8  
16     %1 = load i8** %q, align 8  
17     store i8* %1, i8** %p, align 8  
18     store i8* %0, i8** %q, align 8  
19     ret void  
20 }
```

LLVM IR

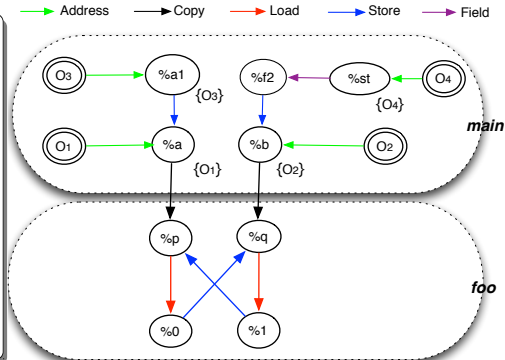


Constraint Graph

Andersen's Pointer Analysis

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12 }  
13 define void @swap(i8** %p, i8** %q) {  
14   entry:  
15     %0 = load i8** %p, align 8  
16     %1 = load i8** %q, align 8  
17     store i8* %1, i8** %p, align 8  
18     store i8* %0, i8** %q, align 8  
19     ret void  
20 }
```

LLVM IR

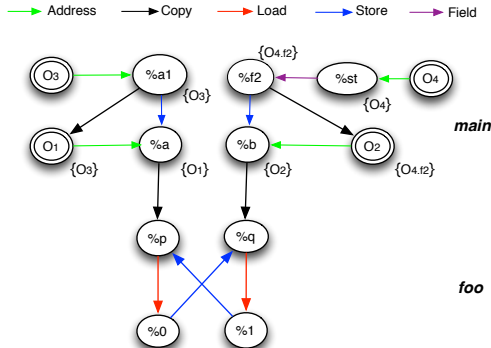


Constraint Graph

Andersen's Pointer Analysis

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12  }  
13  define void @swap(i8** %p, i8** %q) {  
14    entry:  
15    %0 = load i8** %p, align 8  
16    %1 = load i8** %q, align 8  
17    store i8* %1, i8** %p, align 8  
18    store i8* %0, i8** %q, align 8  
19    ret void  
20  }
```

LLVM IR

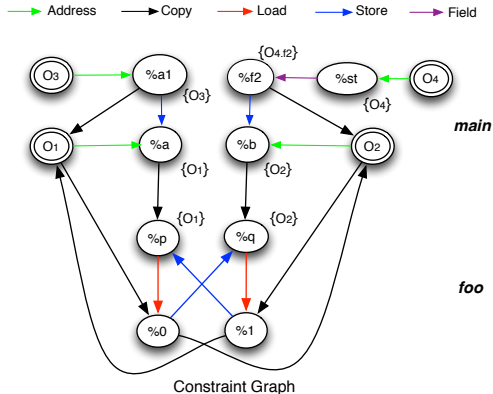


Constraint Graph

Andersen's Pointer Analysis

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12  }  
13  define void @swap(i8** %p, i8** %q) {  
14    entry:  
15    %0 = load i8** %p, align 8  
16    %1 = load i8** %q, align 8  
17    store i8* %1, i8** %p, align 8  
18    store i8* %0, i8** %q, align 8  
19    ret void  
20  }
```

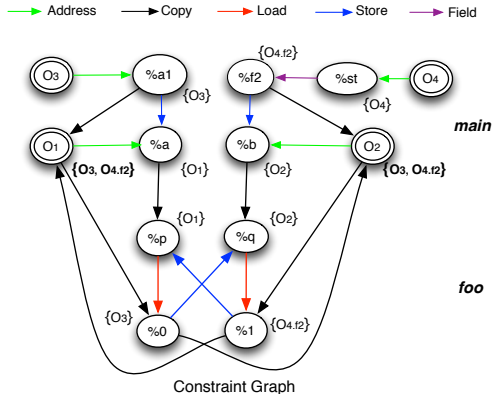
LLVM IR



Andersen's Pointer Analysis

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12  }  
13  define void @swap(i8** %p, i8** %q) {  
14    entry:  
15    %0 = load i8** %p, align 8  
16    %1 = load i8** %q, align 8  
17    store i8* %1, i8** %p, align 8  
18    store i8* %0, i8** %q, align 8  
19    ret void  
20  }
```

LLVM IR



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 $o \in \text{Pts}(q)$.
 - **Store:** $*p = q$ is annotated with a $o = \chi(o)$ for each variable $o \in \text{Pts}(p)$.
 - **Callsite:** $\text{foo}(\dots)$ 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:** $\text{foo}(\dots)$ 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

```
1 define i32 @main() {  
2   entry:  
3     %a = alloca i8*, align 8           // O1  
4     %b = alloca i8*, align 8           // O2  
5     %a1 = alloca i8, align 1           // O3  
6     %st = alloca %struct.st, align 1   // O4  
7     store i8* %a1, i8** %a, align 8  
8     %f2 = getelementptr ... %st, i32 0, i32 1  
9     store i8* %f2, i8** %b, align 8  
10    call void @swap(i8** %a, i8** %b)  
11    ret i32 0  
12  }  
13  define void @swap(i8** %p, i8** %q) {  
14    entry:  
15    %0 = load i8** %p, align 8  
16    %1 = load i8** %q, align 8  
17    store i8* %1, i8** %p, align 8  
18    store i8* %0, i8** %q, align 8  
19    ret void  
20  }
```

LLVM IR

Annotation

```
1  =====FUNCTION: main=====  
2  entry  
3    %a = alloca i8*, align 8           // O1  
4    %b = alloca i8*, align 8           // O2  
5    %a1 = alloca i8, align 1           // O3  
6    %st = alloca %struct.st, align 1   // O4  
7  
8    store i8* %a1, i8** %a, align 8  
9    MR1V_2 = STCHI(MR1V_1)  
10  
11    %f2 = getelementptr ... %st, i32 0, i32 1  
12    store i8* %f2, i8** %b, align 8  
13    MR2V_2 = STCHI(MR2V_1)  
14  
15    CALMU(MR1V_2)  
16    CALMU(MR2V_2)  
17    call void @swap(i8** %a, i8** %b)  
18    MR1V_3 = CALCHI(MR1V_2)  
19    MR2V_3 = CALCHI(MR2V_2)  
20  
21    ret i32 0  
22  =====FUNCTION: swap=====  
23  MR1V_1 = ENCHI(MR1V_0)  
24  MR2V_1 = ENCHI(MR2V_0)  
25  entry  
26  LDMU(MR1V_1)  
27  %0 = load i8*, i8** %p, align 8  
28  
29  LDMU(MR2V_1)  
30  %1 = load i8*, i8** %q, align 8  
31  
32  store i8* %1, i8** %p, align 8  
33  MR1V_2 = STCHI(MR1V_1)  
34  
35  store i8* %0, i8** %q, align 8  
36  MR2V_2 = STCHI(MR2V_1)  
37  
38  ret void  
39  RETMU(MR1V_2)  
40  RETMU(MR2V_2)
```

Annotated IR

Memory SSA

```
1  =====FUNCTION: main=====
2  entry
3  %a = alloca i8*, align 8      // O1
4  %b = alloca i8*, align 8      // O2
5  %a1 = alloca i8, align 1      // O3
6  %st = alloca %struct.st, align 1 // O4
7
8  store i8* %a1, i8** %a, align 8
9  MR1V_2 = STCHI(MR1V_1)
10
11  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
12  store i8* %f2, i8** %b, align 8
13  MR2V_2 = STCHI(MR2V_1)
14
15  CALMU(MR1V_2)
16  CALMU(MR2V_2)
17  call void @swap(i8** %a, i8** %b)
18  MR1V_3 = CALCHI(MR1V_2)
19  MR2V_3 = CALCHI(MR2V_2)
20
21  ret i32 0
22  =====FUNCTION: swap=====
23  MR1V_1 = ENCHI(MR1V_0)
24  MR2V_1 = ENCHI(MR2V_0)
25  entry
26  LDMU(MR1V_1)
27  %0 = load i8*, i8** %p, align 8
28
29  LDMU(MR2V_1)
30  %1 = load i8*, i8** %q, align 8
31
32  store i8* %1, i8** %p, align 8
33  MR1V_2 = STCHI(MR1V_1)
34
35  store i8* %0, i8** %q, align 8
36  MR2V_2 = STCHI(MR2V_1)
37
38  ret void
39  RETMU(MR1V_2)
40  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

```
1  =====FUNCTION: main=====
2  entry
3  %a = alloca i8*, align 8      // O1
4  %b = alloca i8*, align 8      // O2
5  %a1 = alloca i8, align 1      // O3
6  %st = alloca %struct.st, align 1 // O4
7
8  store i8* %a1, i8** %a, align 8
9  MR1V_2 = STCHI(MR1V_1)
10
11  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
12  store i8* %f2, i8** %b, align 8
13  MR2V_2 = STCHI(MR2V_1)
14
15  CALMU(MR1V_2)
16  CALMU(MR2V_2)
17  call void @swap(i8** %a, i8** %b)
18  MR1V_3 = CALCHI(MR1V_2)
19  MR2V_3 = CALCHI(MR2V_2)
20
21  ret i32 0
22  =====FUNCTION: swap=====
23  MR1V_1 = ENCHI(MR1V_0)
24  MR2V_1 = ENCHI(MR2V_0)
25  entry
26  LDMU(MR1V_1)
27  %0 = load i8*, i8** %p, align 8
28
29  LDMU(MR2V_1)
30  %1 = load i8*, i8** %q, align 8
31
32  store i8* %1, i8** %p, align 8
33  MR1V_2 = STCHI(MR1V_1)
34
35  store i8* %0, i8** %q, align 8
36  MR2V_2 = STCHI(MR2V_1)
37
38  ret void
39  RETMU(MR1V_2)
40  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

```
1  =====FUNCTION: main=====
2  entry
3  %a = alloca i8*, align 8           // O1
4  %b = alloca i8*, align 8           // O2
5  %a1 = alloca i8, align 1           // O3
6  %st = alloca %struct.st, align 1   // O4
7
8  store i8* %a1, i8** %a, align 8
9  MR1V_2 = STCHI(MR1V_1)
10
11  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
12  store i8* %f2, i8** %b, align 8
13  MR2V_2 = STCHI(MR2V_1)
14
15  CALMU(MR1V_2)
16  CALMU(MR2V_2)
17  call void @swap(i8** %a, i8** %b)
18  MR1V_3 = CALCHI(MR1V_2)
19  MR2V_3 = CALCHI(MR2V_2)
20
21  ret i32 0
22  =====FUNCTION: swap=====
23  MR1V_1 = ENCHI(MR1V_0)
24  MR2V_1 = ENCHI(MR2V_0)
25  entry
26  LDMU(MR1V_1)
27  %0 = load i8*, i8** %p, align 8
28
29  LDMU(MR2V_1)
30  %1 = load i8*, i8** %q, align 8
31
32  store i8* %1, i8** %p, align 8
33  MR1V_2 = STCHI(MR1V_1)
34
35  store i8* %0, i8** %q, align 8
36  MR2V_2 = STCHI(MR2V_1)
37
38  ret void
39  RETMU(MR1V_2)
40  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

```
1  =====FUNCTION: main=====
2  entry
3  %a = alloca i8*, align 8      // O1
4  %b = alloca i8*, align 8      // O2
5  %a1 = alloca i8, align 1      // O3
6  %st = alloca %struct.st, align 1 // O4
7
8  store i8* %a1, i8** %a, align 8
9  MR1V_2 = STCHI(MR1V_1)
10
11  %f2 = getelementptr inbounds %struct.st, %struct.st* %st, i32 0, i32 1
12  store i8* %f2, i8** %b, align 8
13  MR2V_2 = STCHI(MR2V_1)
14
15  CALMU(MR1V_2)
16  CALMU(MR2V_2)
17  call void @swap(i8** %a, i8** %b)
18  MR1V_3 = CALCHI(MR1V_2)
19  MR2V_3 = CALCHI(MR2V_2)
20
21  ret i32 0
22  =====FUNCTION: swap=====
23  MR1V_1 = ENCHI(MR1V_0)
24  MR2V_1 = ENCHI(MR2V_0)
25  entry
26  LDMU(MR1V_1)
27  %0 = load i8*, i8** %p, align 8
28
29  LDMU(MR2V_1)
30  %1 = load i8*, i8** %q, align 8
31
32  store i8* %1, i8** %p, align 8
33  MR1V_2 = STCHI(MR1V_1)
34
35  store i8* %0, i8** %q, align 8
36  MR2V_2 = STCHI(MR2V_1)
37
38  ret void
39  RETMU(MR1V_2)
40  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

```
1 define i32 @main() {
2   entry:
3     %a = alloca i8*, align 8           // O1
4     %b = alloca i8*, align 8           // O2
5     %a1 = alloca i8, align 1           // O3
6     %st = alloca %struct.st, align 1    // O4
7     store i8* %a1, i8** %a, align 8
8     %f2 = getelementptr ... %st, i32 0, i32 1
9     store i8* %f2, i8** %b, align 8
10    call void @swap(i8** %a, i8** %b)
11    ret i32 0
12 }
13 define void @swap(i8** %p, i8** %q) {
14   entry:
15     %0 = load i8** %p, align 8
16     %1 = load i8** %q, align 8
17     store i8* %1, i8** %p, align 8
18     store i8* %0, i8** %q, align 8
19     ret void
20 }
```

LLVM IR

Annotation

```
1  =====FUNCTION: main=====
2  entry
3    %a = alloca i8*, align 8           // O1
4    %b = alloca i8*, align 8           // O2
5    %a1 = alloca i8, align 1           // O3
6    %st = alloca %struct.st, align 1    // O4
7
8    store i8* %a1, i8** %a, align 8
9    MR1V_2 = STCHI(MR1V_1)
10
11    %f2 = getelementptr ... %st, i32 0, i32 1
12    store i8* %f2, i8** %b, align 8
13    MR2V_2 = STCHI(MR2V_1)
14
15    CALMU(MR1V_2)
16    CALMU(MR2V_2)
17    call void @swap(i8** %a, i8** %b)
18    MR1V_3 = CALCHI(MR1V_2)
19    MR2V_3 = CALCHI(MR2V_2)
20
21    ret i32 0
22  =====FUNCTION: swap=====
23  MR1V_1 = ENCHI(MR1V_0)
24  MR2V_1 = ENCHI(MR2V_0)
25  entry
26  LDMU(MR1V_1)
27    %0 = load i8*, i8** %p, align 8
28
29  LDMU(MR2V_1)
30    %1 = load i8*, i8** %q, align 8
31
32    store i8* %1, i8** %p, align 8
33    MR1V_2 = STCHI(MR1V_1)
34
35    store i8* %0, i8** %q, align 8
36    MR2V_2 = STCHI(MR2V_1)
37
38    ret void
39    RETMU(MR1V_2)
40    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 $\ell_{cs} : r = f(..., p, ...) [\mu(o')] [o = \chi(-)]$:
 - ACTUALPARAM (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)

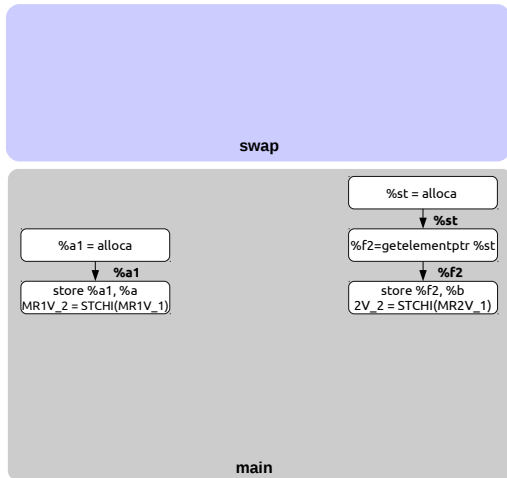
- **FunctionEntry/Exit VFNodes:** A variable defined at the entry or returned at the exit of a function $f(..., p, ...) \{ [o = \chi(-)] \dots [\mu(o')] \text{ return } q; \}$:
 - FORMALPARAM (parameter directly initialized): $p@l_f$.
 - FORMALRET (formal return): $q@l_{\underline{f}}$.
 - FORMALIN (parameter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

Rule	VFG Nodes	VFG Edges
COPY	$l : p = q$	$p@l \longleftrightarrow q@l'$
PHI	$l : v_3 = \phi(v_1, v_2)$	$v_3@l \longleftrightarrow v_1@l' \quad v_3@l \longleftrightarrow v_2@l''$
LOAD	$l : p = *q [\mu(o)]$	$p@l \longleftrightarrow o@l'$
STORE	$l : *p = q [o_2 = \chi(o_1)]$	$o_2@l \longleftrightarrow q@l' \quad o_2@l \longleftrightarrow o_1@l''$
CALL	$l_{cs} : r = f(..., p, ...) [\mu(o_1)] \quad [o_2 = \chi(-)]$ $l_f : f(..., q, ...) \{ [o_3 = \chi(-)] \dots [\mu(o_4)] \text{ return } x \}$	$q@l_f \longleftrightarrow p@l_1 \quad r@l_{cs} \longleftrightarrow x@l_2$ $o_3@l_f \longleftrightarrow o_1@l_3 \quad o_2@l_{cs} \longleftrightarrow o_4@l_4$

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



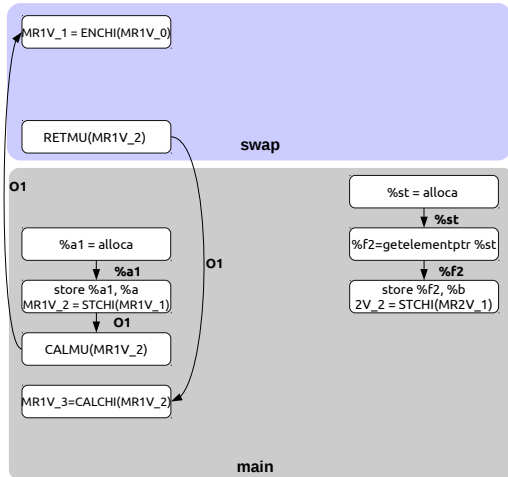
Value-Flow Graph

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

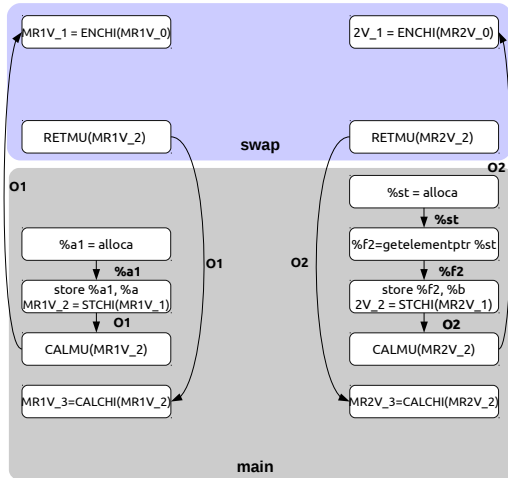
Interprocedural Value-Flow Graph (An Example)

```

=====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



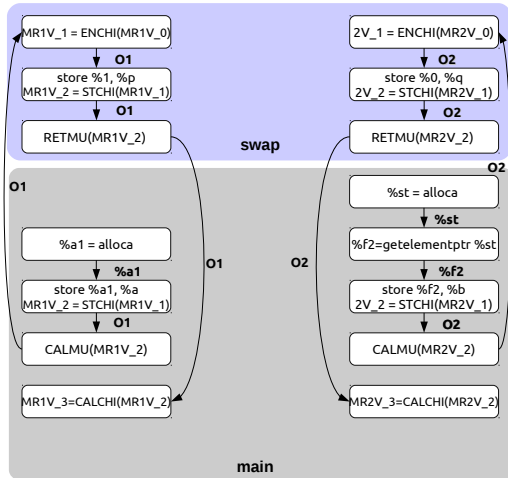
Value-Flow Graph

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=====
entry
  MR1V_1 = ENCHI(MR1V_0)
  MR2V_1 = ENCHI(MR2V_0)
  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

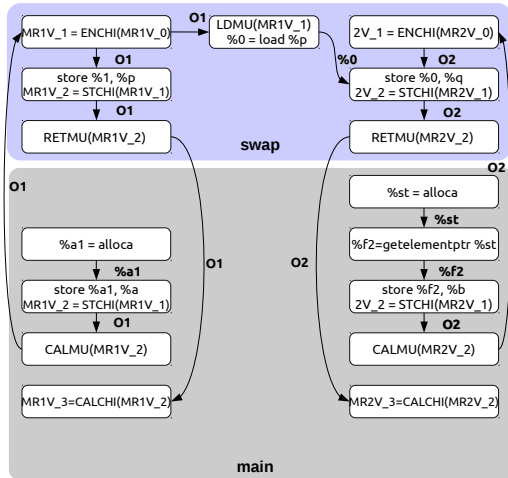
Interprocedural Value-Flow Graph (An Example)

```

=====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=====
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)

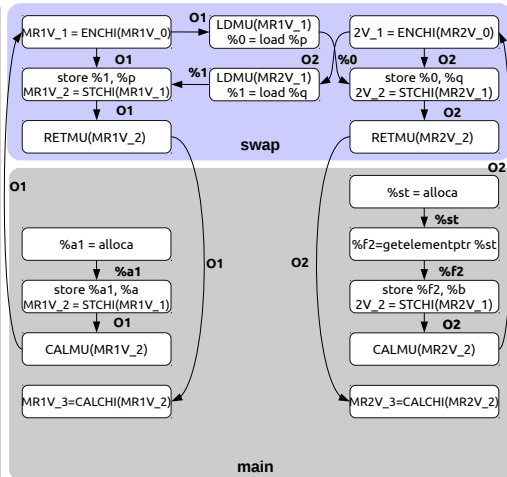
```

=====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=====
entry
MR1V_1 = ENCHI(MR1V_0)
MR2V_1 = ENCHI(MR2V_0)
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)

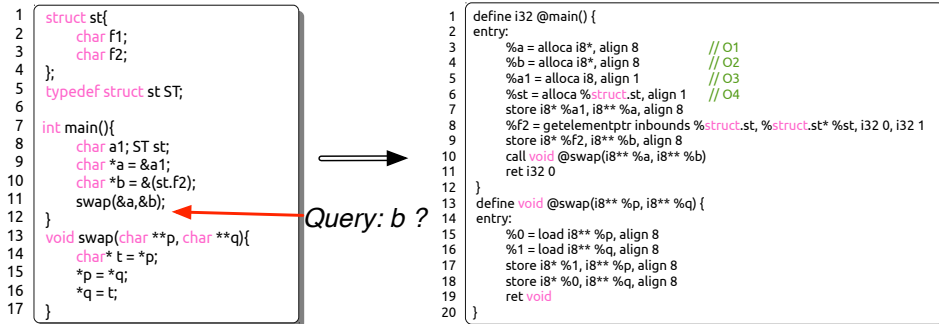
```

Annotated IR

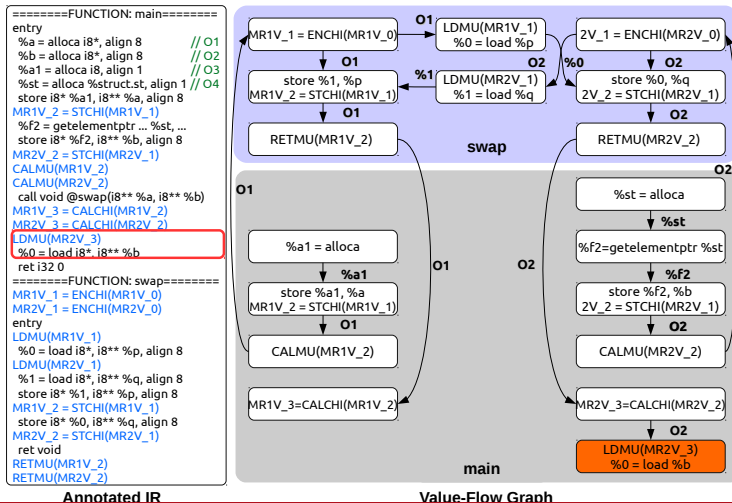


Value-Flow Graph

Demand-Driven Analysis via Value-Flows



Demand-Driven Analysis via Value-Flows



Annotated IR

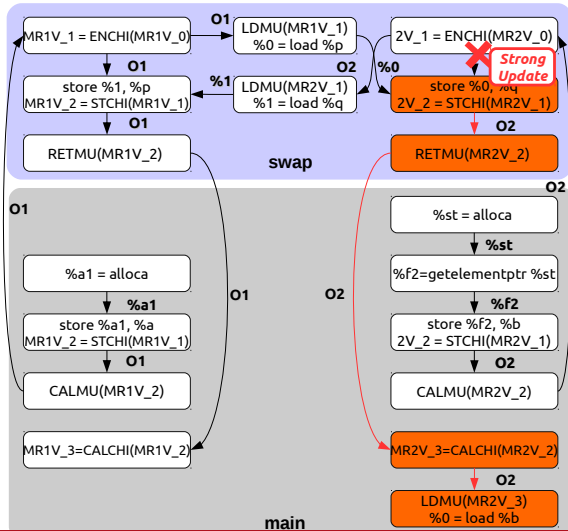
Value-Flow Graph

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)

```



Demand-Driven Analysis via Value-Flows

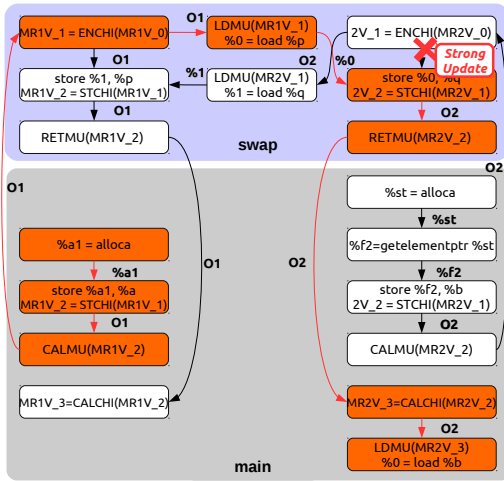
```

=====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