Graph Representation of Code

Yulei Sui

University of Technology Sydney, Australia

SVF : Static Value-Flow Analysis Framework for Source Code

A scalable, precise and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.

SVF : Static Value-Flow Analysis Framework for Source Code

A scalable, precise and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

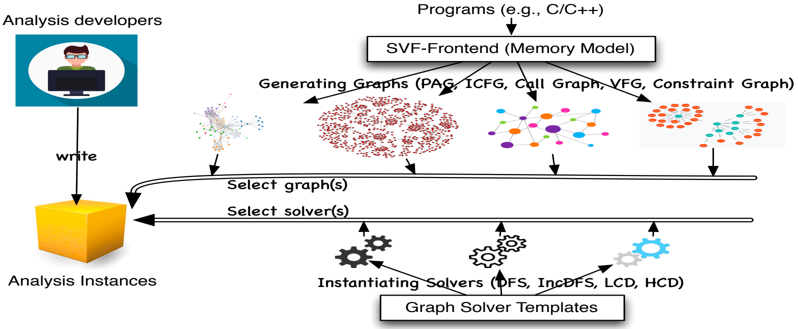
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point A flow to another program point B along some execution paths?
  - Can function F be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?

SVF : Static Value-Flow Analysis Framework for Source Code

A scalable, precise and on-demand interprocedural program dependence analysis framework for both sequential and multithreaded programs.

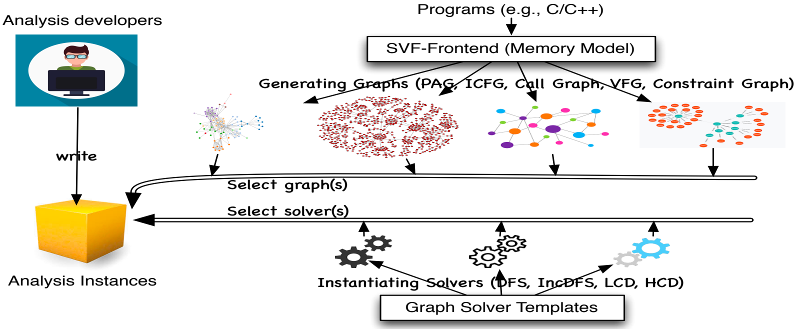
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 10.0.0) with over 100 KLOC C/C++ code and **530+ stars with 30+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point A flow to another program point B along some execution paths?
  - Can function F be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?
- Key features of SVF
  - **Sparse**: compute and maintain the data-flow facts where necessary
  - **Selective** : support mixed analyses for precision and efficiency trade-offs.
  - **On-demand** : reason about program parts based on user queries.

SVF: Design Principle



- Serving as an open-source foundation for building practical static source code analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.

SVF: Design Principle



- Serving as an open-source foundation for building practical static source code analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different analysis variants (flow-, context-, heap-, field-sensitive analysis) in a sparse and on-demand manner.
- Client applications:
  - Static bug detection (e.g., memory leaks, null dereferences, use-after-frees and data-races)
  - Accelerate dynamic analysis (e.g., Google's Sanitizers and AFL fuzzing)

Graph Representation of Code

- What is a graph representation of code?
  - Representing a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.

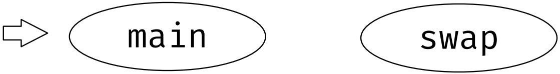
Graph Representation of Code

- What is a graph representation of code?
  - Representing a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.
- Why a graph representation?
  - Abstracting code from low-level complicated instructions
  - Applying general graph algorithms
  - Easy to maintain and extend

Call Graph

```
  define i32 @main() #0 {
1 entry:
2 %a1 = alloca i8, align 1
3 %b1 = alloca i8, align 1
4 %a = alloca i8*, align 8
5 %b = alloca i8*, align 8
6 store i8* %a1, i8** %a, align 8
7 store i8* %b1, i8** %b, align 8
8 call void @swap(i8** %a, i8** %b)
9 ret i32 0
  }
  define void @swap(i8** %p, i8** %q) #0
  {
10 entry:
11 %0 = load i8** %p, align 8
12 %1 = load i8** %q, align 8
13 store i8* %1, i8** %p, align 8
14 store i8* %0, i8** %q, align 8
15 ret void
  }
```

Program calling relations between methods



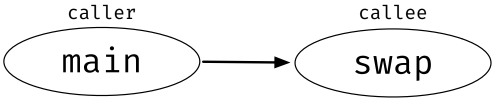
Call Graph

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

Call Graph

```
  define i32 @main() #0 {
1 entry:
2 %a1 = alloca i8, align 1
3 %b1 = alloca i8, align 1
4 %a = alloca i8*, align 8
5 %b = alloca i8*, align 8
6 store i8* %a1, i8** %a, align 8
7 store i8* %b1, i8** %b, align 8
8 call void @swap(i8** %a, i8** %b)
9 ret i32 0
  }
  define void @swap(i8** %p, i8** %q) #0
  {
10 entry:
11 %0 = load i8** %p, align 8
12 %1 = load i8** %q, align 8
13 store i8* %1, i8** %p, align 8
14 store i8* %0, i8** %q, align 8
15 ret void
  }
```

- each node represents a program method
- each edge represents a calling relation between two program methods



Call Graph

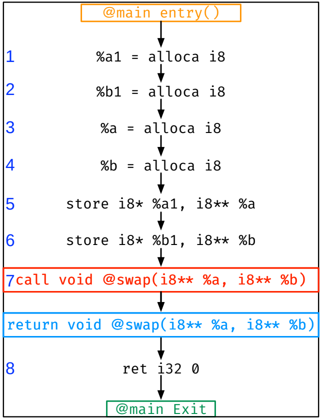
https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

Control Flow Graph

Program execution order **between two instructions.**

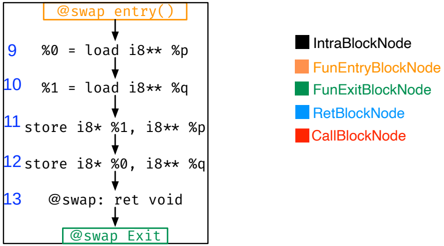
- Intra-procedural control-flow graph: control-flow graph within a program method.
- Inter-procedural control-flow graph: control-flow graph across program methods.

Intra-procedural Control Flow Graph



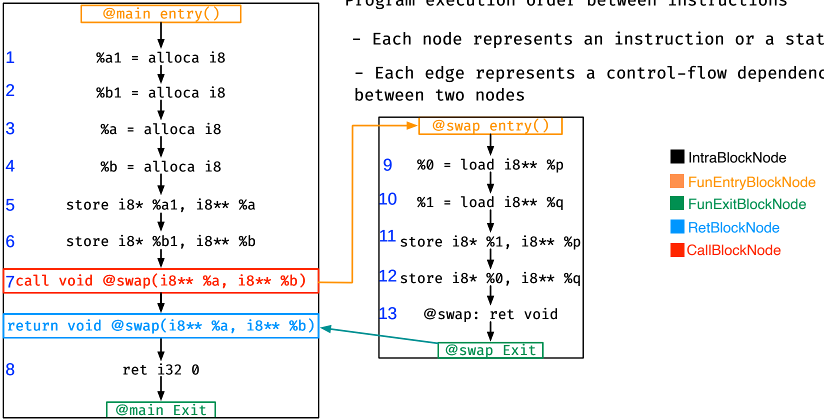
Program execution order between instructions

- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes



https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

Inter-procedural Control Flow Graph (ICFG)



Program execution order between instructions

- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes

https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

Constraint Graph (Program Assignment Graph)

- Constraint Graph represents the assignment **constraints between variables.**
- Constraint Graph and Program Assignment Graph (PAG) are essentially the same.
- The difference is that PAG can not be changed while you can add edges or nodes on the Constraint Graph to perform constraint solving.

Constraint Graph (Program Assignment Graph)

Program Assignment relation between two variables

- each node represent a pointer or an object
- each edge represents two nodes dependence or constraint relation

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

alloca instruction allocates typed integer
8 bytes of memory object as O_1 O_2 O_3 O_4

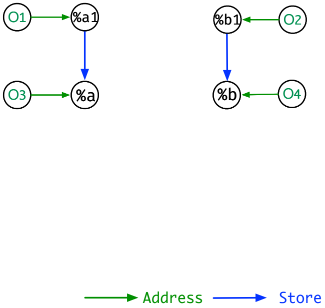
Address

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

Constraint Graph (Program Assignment Graph)

```
Program Assignment relation between two variables
   - each node represent a pointer or an object
   - each edge represents two nodes dependence or constraint relation
```

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




```
alloca instruction allocates typed integer
8 bytes of memory object as O1 O2 O3 O4
```

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

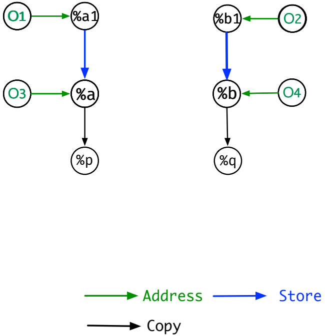
Constraint Graph (Program Assignment Graph)

Program Assignment relation between two variables

- each node represent a pointer or an object
- each edge represents two nodes dependence or constraint relation

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

```
define void @swap(i8** %p, i8** %q)
{
  ...
}
```

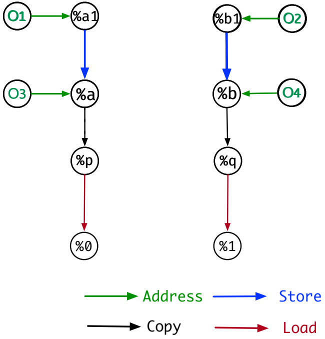


https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Constraint Graph (Program Assignment Graph)

```
Program Assignment relation between two variables
   - each node represent a pointer or an object
   - each edge represents two nodes dependence or constraint relation
```

```
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```

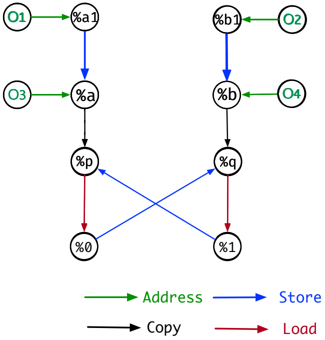


https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#5-pag

Constraint Graph (Program Assignment Graph)

```
Program Assignment relation between two variables
    - each node represent a pointer or an object
    - each edge represents two nodes dependence or constraint relation
```

```
define void @swap(i8** %p, i8** %q) #0
{
entry:
%0 = load i8** %p, align 8
%1 = load i8** %q, align 8
store i8* %1, i8** %p, align 8
store i8* %0, i8** %q, align 8
ret void
}
```



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

What's next?

- (1) Compile two C programs (`swap.c` and `example.c`) into their LLVM IR.
  - A guide can be found here `https://github.com/SVF-tools/SVF-Teaching/wiki/CodeGraph#2-llvm-ir-generation`
  - Understand the mapping from a C program to its corresponding LLVM IR.
- (2) Generate and visualize the graph representation of LLVM IR (`swap.ll` and `example.ll`).
  - `https://github.com/SVF-tools/SVF-Teaching/wiki/CodeGraph#3-run-and-debug-your-codegraph`
- (3) Write code to iterate nodes and edges of ICFG and PAG and print their contents.
  - `https://github.com/SVF-tools/SVF-Teaching/blob/main/CodeGraph/CodeGraph.cpp#L65-L82`
- (4) More about LLVM IR and SVF's graph representation
  - LLVM language manual `https://llvm.org/docs/LangRef.html`
  - SVF website `https://github.com/SVF-tools/SVF`