SVFIR and 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 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ 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.

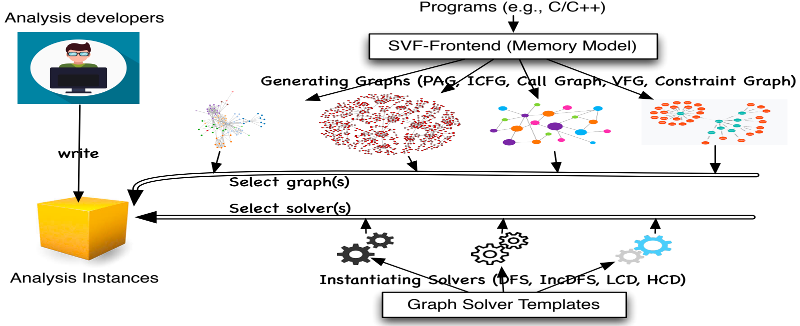
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ contributors** and over 1K commits on Github.
  - Invited 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 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ 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



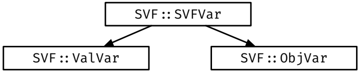
- Serving as an open-source foundation for building practical static source code analysis
  - Bridge the gap between research and engineering
  - Minimize the efforts of implementing sophisticated analysis (extendable, reusable, and robust via layers of abstractions)
  - Support developing different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

SVF IR and Why?

- SVFIR is a much simplified representation of LLVM IR (or SSA-based programming languages) for static analysis purposes.
- Lightweight in terms of fewer types of program variables and statements.

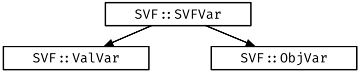
SVF IR and Why?

- SVFIR is a much simplified representation of LLVM IR (or SSA-based programming languages) for static analysis purposes.
- Lightweight in terms of fewer types of program variables and statements.
- SVFVar: program variables

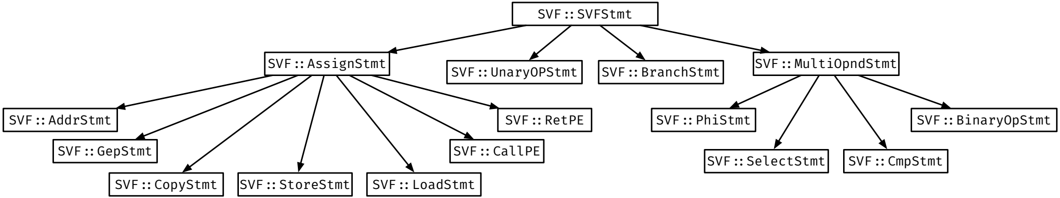


SVF IR and Why?

- SVFIR is a much simplified representation of LLVM IR (or SSA-based programming languages) for static analysis purposes.
- Lightweight in terms of fewer types of program variables and statements.
- SVFVar: program variables



- SVFStmt: program statements



SVF Program Variables (SVFVar)

- An SVFVar represent either a top-level variable ($\mathbb{P}$) or a memory object variable ($\mathbb{O}$)
- Each SVFVar has a unique identifier (ID)
- SVFVar ID 0-4 are reserved

Program Variables	Domain	Meanings
SVFVar	$\mathbb{V} = \mathbb{P} \cup \mathbb{O}$	Program Variables
ValVar	$\mathbb{P}$	Top-level variables (scalars and pointers)
ObjVar	$\mathbb{O} = \mathbb{S} \cup \mathbb{G} \cup \mathbb{H} \cup \mathbb{C}$	Memory Objects (stack, global[1], heap and constant data)
FIObjVar	$o \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$	A single (base) memory object
GepObjVar	$o_i \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}) \times \mathbb{P}$	*i*-th subfield/element of an (aggregate) object
ConstantData	$\mathbb{C}$	Constant data (e.g., numbers and strings)
Program Statement	$l \in \mathbb{L}$	Statements labels

[1] Function objects are considered as global objects

SVF Program Statements (SVFStmt)

An SVFStmt is one of the following program statements representing the relations between SVFVars.

SVFStmt	LLVM-Like form	C-Like form	Operand types
AddrStmt	%ptr = alloca or constantData	$p = \texttt{alloc}$ or $p = c$	$\mathbb{P} \times (\mathbb{O} \cup \mathbb{C})$
CopyStmt	%p = bitcast %q	$p = q$	$\mathbb{P} \times \mathbb{P}$
LoadStmt	%p = load %q	$p = *q$	$\mathbb{P} \times \mathbb{P}$
StoreStmt	store %p, %q	$*p = q$	$\mathbb{P} \times \mathbb{P}$
GepStmt	%p = getelementptr %q, %i	$p = \&(q \rightarrow i)$ or $p = \&q[i]$	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
PhiStmt	%p = phi [l_1, %q_1], [l_2, %q_2]	$p = \texttt{phi}(l_1 : q_1,\ l_2 : q_2)$	$\mathbb{P} \times (\mathbb{L} \rightarrow \mathbb{P}^2)$
BranchStmt	br i1 %p, label %l_1, label %l_2	if (p) l_1 else l_2	$\mathbb{P} \times \mathbb{L}^2$
UnaryOPStmt	$p = \neg q$	$p = \neg q$	$\mathbb{P} \times \mathbb{P}$
BinaryOPStmt/CmpStmt	$r = \otimes\ p, q$	$r = p \otimes q$	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
	%r = call f(...%q_i...) f(...%p_i...){ ... ret %z}	$r = f(\ldots, q_i, \ldots)$ $f(\ldots, p_i, \ldots)\{\ldots$ return $z\}$	
CallPE	%p_i = %q_i $\quad (1 < i < n)$	$p_i = q_i \quad (1 < i < n)$	$(\mathbb{P} \times \mathbb{P})^n$
RetPE	%r = %z	$r = z$	$\mathbb{P} \times \mathbb{P}$

$\otimes \in \{+, -, *, /, \%, <<, >>, <, >, \&, \&\&, <=, >=, \equiv, \sim, |, \wedge\}$

SVF Program Statements (SVFStmt)

- SVFStmt follows the LLVM's SSA form for top-level variables
  - Top-level variables ($\mathbb{P}$) can only be defined once
  - Memory objects (i.e., $\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}$ excluding constant data) can only be modified/read through top-level pointers at `StoreStmt` and `LoadStmt`.
  - For example, `p = &a; *p = r;` The value of `a` can only be modified/read via dereferencing `p`.
- A ConstantData ($\mathbb{C}$) object needs first to be assigned to a temp top-level variable and can only be read through that top-level variable in any `SVFStmt`.
  - For example, `*p = 3;` $\Rightarrow$ `t = 3; *p = t;`
- `CallPE` represents the parameter passing from an actual parameter at a callsite to a formal parameter of a callee function.
- `RetPE` represents the parameter passing from a function return to a callsite return variable.

Graph Representation of Code

- What is a graph representation of code (code graph)?
  - Put the LLVM IR or SVF IR on a graph representation.
  - Represent 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

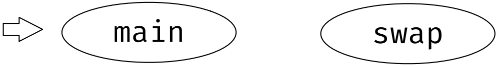
- What is a graph representation of code (code graph)?
  - Put the LLVM IR or SVF IR on a graph representation.
  - Represent a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.
- Why a graph representation?
  - Abstracting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

Call Graph

- Program calling relations between methods
- Whether a method A can call method B directly or transitively.

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

Program calling relations between methods



Call Graph

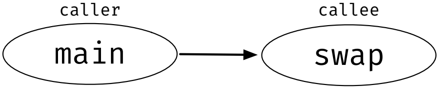
https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

Call Graph

- Program calling relations between methods
- Whether a method A can call method B directly or transitively.

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

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



Call Graph

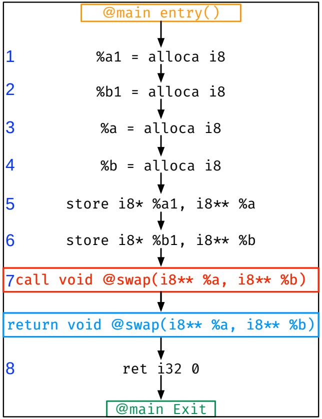
https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph

Control Flow Graph

Program execution order **between two LLVM instructions (SVFStmts)**.

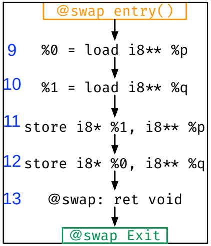
- Intra-procedural control-flow graph: control-flow within a program method.
- Inter-procedural control-flow graph: control-flow across program methods.

Intra-procedural Control Flow Graph



Program execution order between instructions

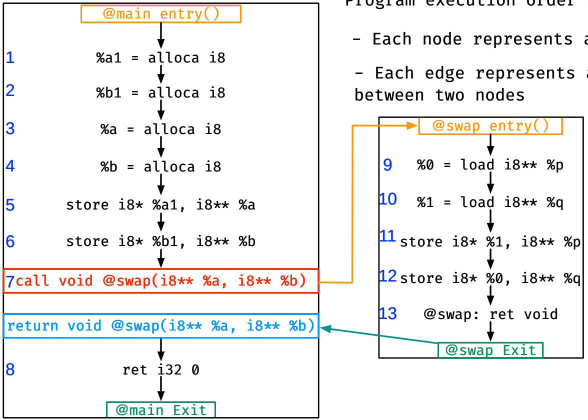
- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes



IntraICFGNode
FunEntryICFGNode
FunExitICFGNode
RetICFGNode
CallICFGNode

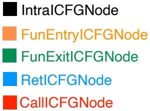
https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

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

SVF IR Example[2]

```
int foo(b){
    return b;
}
int main(){
    int a = foo(0);
}
```

[2] https://github.com/SVF-tools/Teaching-Software-Verification/wiki/svfir

SVF IR Example[2]

```c
int foo(b){
    return b;
}
int main(){
    int a = foo(0);
}
```

```llvm
define i32 @foo(i32 %b) {
  %b.addr = alloca i32
  store i32 %b, i32* %b.addr
  %0 = load i32, i32* %b.addr
  ret i32 %0
}

define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

[2] https://github.com/SVF-tools/Teaching-Software-Verification/wiki/svfir

SVF IR Example[2]

```
int foo(b){
    return b;
}
int main(){
    int a = foo(0);
}
```

```
define i32 @foo(i32 %b) {
  %b.addr = alloca i32
  store i32 %b, i32* %b.addr
  %0 = load i32, i32* %b.addr
  ret i32 %0
}

define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

Variables introduced by SVF (created internally)

SVFVar	Meaning
DummyValVar ID: 0	reserved
DummyValVar ID: 1	reserved
DummyObjVar ID: 2	reserved
DummyObjVar ID: 3	reserved
ValVar ID: 4	foo
FIObjVar ID: 5	foo
RetPN ID: 6	ret of foo
ValVar ID: 13	main
FIObjVar ID: 14	main
RetPN ID: 15	ret of main

[2] https://github.com/SVF-tools/Teaching-Software-Verification/wiki/svfir

SVF IR Example[2]

```c
int foo(b){
    return b;
}
int main(){
    int a = foo(0);
}
```

```llvm
define i32 @foo(i32 %b) {
  %b.addr = alloca i32
  store i32 %b, i32* %b.addr
  %0 = load i32, i32* %b.addr
  ret i32 %0
}

define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

Variables introduced by SVF
(created internally)

SVFVar	Meaning
DummyValVar ID: 0	reserved
DummyValVar ID: 1	reserved
DummyObjVar ID: 2	reserved
DummyObjVar ID: 3	reserved
ValVar ID: 4	foo
FIObjVar ID: 5	foo
RetPN ID: 6	ret of foo
ValVar ID: 13	main
FIObjVar ID: 14	main
RetPN ID: 15	ret of main

Variables introduced by LLVM
(created by LLVM Values)

SVFVar	LLVM Value
ValVar ID: 7	`i32 %b { 0th arg foo }`
ValVar ID: 8	`%b.addr = alloca i32`
FIObjVar ID: 9	`%b.addr = alloca i32`
ValVar ID: 11	`%0 = load i32, i32* %b.addr`
ValVar ID: 16	`%a = alloca i32`
FIObjVar ID: 17	`%a = alloca i32`
ValVar ID: 18	`%call = call i32 @foo(i32 0)`
ValVar ID: 19	`i32 0 { constant data }`
FIObjVar ID: 20	`i32 0 { constant data }`
ValVar ID: 21	`store i32 %call, i32* %a`
ValVar ID: 22	`ret i32 0`

[2] https://github.com/SVF-tools/Teaching-Software-Verification/wiki/svfir

ICFG and SVFStmt Example[3]

```
define i32 @foo(i32 %b) {
  %b.addr = alloca i32
  store i32 %b, i32* %b.addr
  %0 = load i32, i32* %b.addr
  ret i32 %0
}

define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

ICFGNode	SVFStmt	LLVM Value
GlobalICFGNode0	CopyStmt: Var1 ← Var0	`i8* null (constant data)`
	AddrStmt: Var19 ← Var20	`i32 0 (constant data)`
	AddrStmt: Var4 ← Var5	`foo`
	AddrStmt: Var13 ← Var14	`main`
FunEntryICFGNode1	fun: foo	
IntraICFGNode2	AddrStmt: Var8 ← Var9	`%b.addr = alloca i32`
IntraICFGNode3	StoreStmt Var8 ← Var7	`store i32 %b, i32* %b.addr`
IntraICFGNode4	LoadStmt: Var11 ← Var8	`%0 = load i32, i32* %b.addr`
IntraICFGNode5	fun:foo	`ret i32 %0`
FunExitICFGNode6	PhiStmt: [Var6 ← ([Var11, ICFGNode5],)]	`ret i32 %0`
FunEntryICFGNode7	fun: main	
IntraICFGNode8	AddrStmt: [Var16 ← Var17]	`%a = alloca i32`
CallICFGNode9	CallPE: [Var7 ← Var19]	`%call = call i32 @foo(i32 0)`
RetICFGNode10	RetPE: [Var18 ← Var6]	`%call = call i32 @foo(i32 0)`
IntraICFGNode11	StoreStmt: [Var16 ← Var18]	`store i32 %call, i32* %a`
IntraICFGNode12	fun: main	`ret i32 0`
FunExitICFGNode13	PhiStmt: [Var15 ← ([Var19, ICFGNode12],)]	`ret i32 0`

[3] https://github.com/SVF-tools/Teaching-Software-Verification/wiki/svfir

What's next?

- (1) Compile two C programs (`example.c` and `swap.c`) into their LLVM IR.
  - A guide can be found at `https://github.com/SVF-tools/Teaching-Software-Verification/wiki/SVFIR#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 (`example.ll` `swap.ll`).
  - `https://github.com/SVF-tools/Teaching-Software-Verification/wiki/SVFIR#3-run-and-debug-your-svfir`
- (3) Write code to iterate SVFVars and also the nodes and edges of ICFG and print their contents.
  - `https://github.com/SVF-tools/Teaching-Software-Verification/blob/main/SVFIR/SVFIR.cpp#L67-L93`
- (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`