

# LLVM, SVF IR and Control-Flow Reachability

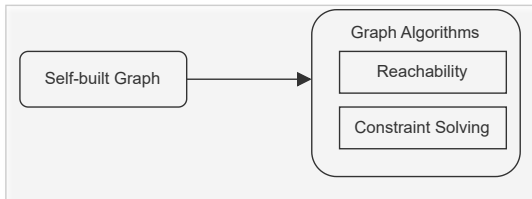
## (Week 2)

Yulei Sui

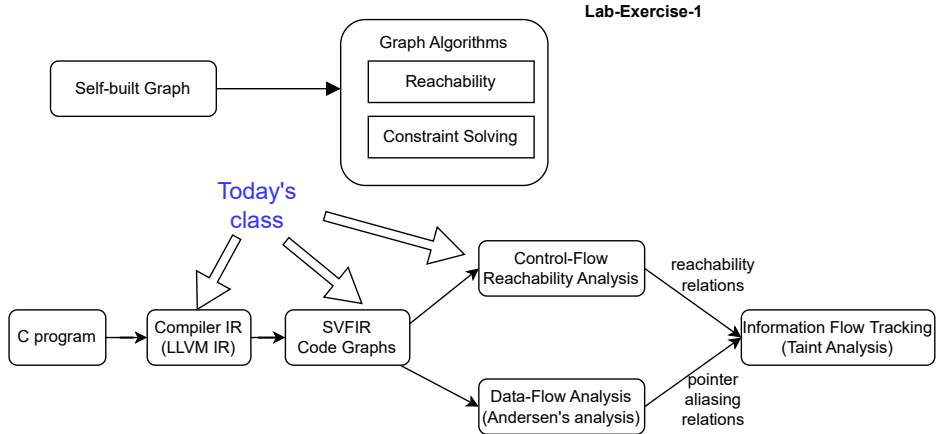
School of Computer Science and Engineering  
University of New South Wales, Australia

# Where We Are Now and Today's Class

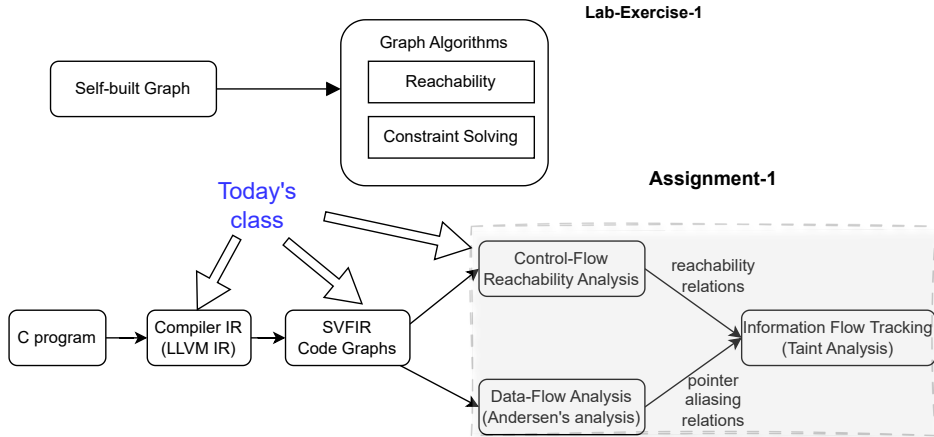
## Lab-Exercise-1



# Today's Class



# Today's Class



# What is LLVM ?

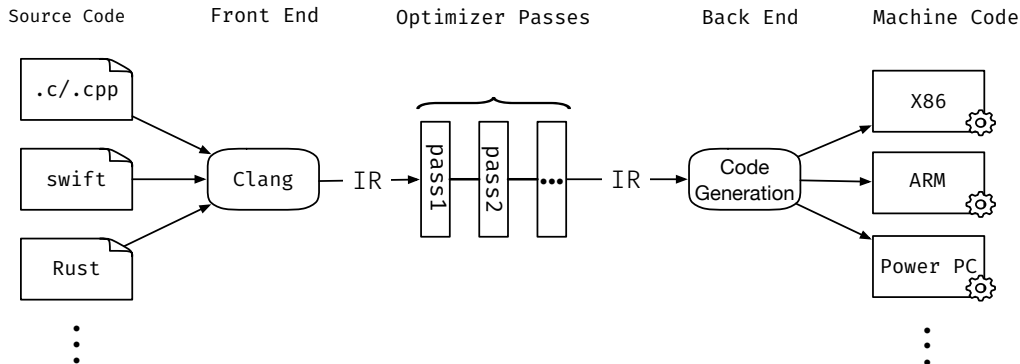
**LLVM compiler infrastructure is a collection of compiler and tool-chain technologies.**

- Originally started in 2000 from UIUC. An **open-source project** and supported and contributed by a range of high-tech companies such as Apple, Google, Intel, ARM.
- Modern compiler infrastructure can be used to develop a **front-end for any programming language** and a **back-end for any instruction set architecture**.
- A set of **reusable software modules** to quickly design your own compiler or software tool chains.
- **Language-independent intermediate representation (IR)** used for a variety of purposes, such as compiler optimizations, static analysis and bug detection.
- **More information on LLVM's website:** <https://llvm.org/>

# Why Learn LLVM or Learn Compilers in General?

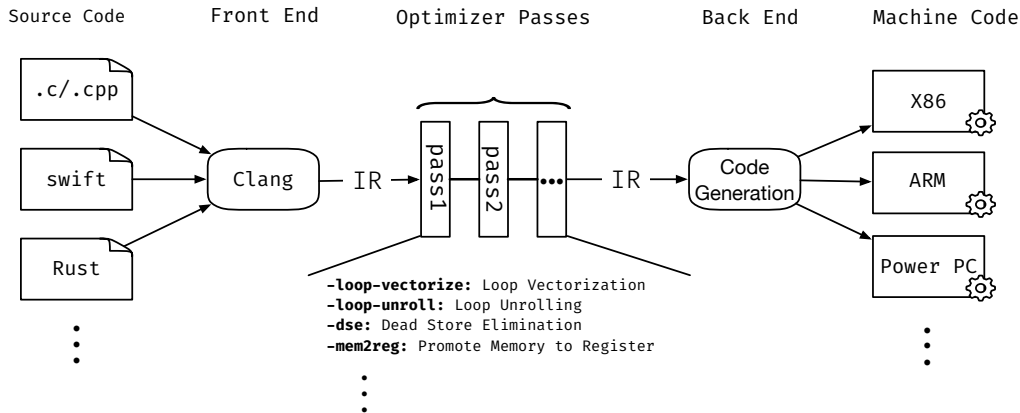
- An essential part of the standard curriculum in computer science.
- One of the most complex systems required for building virtually all other software.
- A perfect base framework to build your own tools for code analysis and verification
- Sharpen your software design and implementation skills.
- Widely used by many major software companies. In-demand skills and competitive salaries in job market.

# LLVM's Architecture



\*IR: Human-readable LLVM IR (.ll files) or dense 'bitcode' binary representation (.bc files)

# LLVM's Architecture



\*IR: Human-readable LLVM IR (.ll files) or dense 'bitcode' binary representation (.bc files)



# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **Language independent.** Not machine code, but one step just above assembly
- **Clear lexical scope**, such as modules, functions, basic blocks, and instructions
  - The IR proper is a higher level than assembly (cf. call, invoke, switch, aggregate types, etc.). MachineInstr is the closest analogue to assembly and the machine code layer to encoded object files.
  - LLVM IR is a good place to start to understand how high level languages get lowered.

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **3-address code style** in **static single assignment (SSA)** form

# LLVM Intermediate Representation (IR)

LLVM IR is LLVM's code representation which is generated by its front-end `clang` when compiling a program (<https://llvm.org/docs/LangRef.html>)

- **3-address code style** in **static single assignment (SSA)** form
  - Variables are strongly typed
  - Global variable (symbol starting with '@'), stack/register var (starting with '%')
  - Each global and stack variable is assigned exactly once.
    - A variable lowered to LLVM IR is renamed if it has multiple definitions, so that each one has a unique name, turning them into a set of static single assignments.
    - The SSA form makes it easier to perform data-flow analysis and optimizations such as constant propagation, dead code elimination, and register allocation because the compiler can track the values of variables more precisely.

```
1 x = 1;
2 x = x + 2;
3 if (x > 3) {
4     x = x * 2;
5 }
```

Translate to SSA form  $\Rightarrow$

```
1 x1 = 1;
2 x2 = x1 + 2;
3 if (x2 > 3) {
4     x3 = x2 * 2;
5 }
6 x4 = phi(x2, x3);
```

# LLVM Intermediate Representation (IR)

**LLVM IR is LLVM's code representation which is generated by its front-end clang when compiling a program** (<https://llvm.org/docs/LangRef.html>)

- **3-address code style in static single assignment (SSA) form**
  - Three addresses and one operator. Each instruction has at most three operands, aligning with the principles of SSA form, where each variable is assigned exactly once and allows for efficient analyses and optimizations.
    - For example, 'a = b op c', where 'a', 'b', 'c' are either programmer defined variables (e.g., heap, global or stack), constants or compiler-generated temporary names. 'op' stands for an operation which is applied on 'a' and 'b'.

// source code

```
a = b + c * d;
```

Translate to SSA form  $\Rightarrow$

// introducing a temporal variable t.

```
t = c * d;
```

```
a = b + t;
```

# Compiling a C Program to Its LLVM IR

## Clang/LLVM compiler options (compile.sh)

- Compile a C program 'swap.c' to a human readable IR 'swap.ll'.
  - `clang -c -S -emit-llvm swap.c -o swap.ll`
- Compilation without optimisation.
  - `clang -c -S -Xclang -disable-00-optnone -emit-llvm swap.c -o swap.ll`
- Keep the variable names.
  - `clang -c -S -fno-discard-value-names -Xclang -disable-00-optnone -emit-llvm swap.c -o swap.ll`
- Keep source code debug information (optional).
  - `clang -g -c -S -fno-discard-value-names -Xclang -disable-00-optnone -emit-llvm swap.c -o swap.ll`
- Convert the LLVM IR to more compact static single assignment form.
  - `opt -S -p=mem2reg swap.ll -o swap.ll`

# LLVM IR - An Example without Source Code Debug Info

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}

int main(){
    char a1;
    char b1;
    char *a;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

clang ...

+

opt -mem2reg ...

⇒

```
define void @swap(ptr %p, ptr %q) #0 !9 {
entry:
    %1 = load ptr, ptr %q, align 8
    store ptr %1, ptr %p, align 8
    store ptr %0, ptr %q, align 8
    ret void, !24
}

define i32 @main() #0 !25 {
entry:
    %a1 = alloca i8, align 1
    %b1 = alloca i8, align 1
    %a = alloca ptr, align 8
    %b = alloca ptr, align 8
    store ptr %a1, ptr %a, align 8
    store ptr %b1, ptr %b, align 8
    call void @swap(ptr %a, ptr %b)
    ret i32 0
}
```

swap.c

swap.ll

<https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/src/swap.c>

# LLVM IR - An Example with Source Code Debug Info

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;    clang -g ...
}
+
int main(){
    char a1;    opt -mem2reg ...
    char b1;    or
    char *a;    SVFIR/compile.sh
    char *b;
    a = &a1;    ⇒
    b = &b1;
    swap(&a,&b);
}
```

swap.c

```
define void @swap(ptr %p, ptr %q) #0 !dbg !9 {
entry:
    call void @llvm.dbg.value(metadata ptr %p, metadata !16, metadata !DIExpression(), !dbg!17)
    call void @llvm.dbg.value(metadata ptr %q, metadata !18, metadata !DIExpression(), !dbg!17)
    %0 = load ptr, ptr %p, align 8, !dbg!19
    call void @llvm.dbg.value(metadata ptr %0, metadata !20, metadata !DIExpression(), !dbg!17)
    %1 = load ptr, ptr %q, align 8, !dbg!21
    store ptr %1, ptr %p, align 8, !dbg!22
    store ptr %0, ptr %q, align 8, !dbg!23
    ret void, !dbg !24
}
define i32 @main() #0 !dbg !25 {
entry:
    %a1 = alloca i8, align 1
    %b1 = alloca i8, align 1
    %a = alloca ptr, align 8
    %b = alloca ptr, align 8
    call void @llvm.dbg.declare(metadata ptr %a1, metadata !29, metadata !DIExpression(), !dbg!30)
    call void @llvm.dbg.declare(metadata ptr %b1, metadata !31, metadata !DIExpression(), !dbg!32)
    call void @llvm.dbg.declare(metadata ptr %a, metadata !33, metadata !DIExpression(), !dbg!34)
    call void @llvm.dbg.declare(metadata ptr %b, metadata !35, metadata !DIExpression(), !dbg!36)
    store ptr %a1, ptr %a, align 8, !dbg!37
    store ptr %b1, ptr %b, align 8, !dbg!38
    call void @swap(ptr noundef %a, ptr noundef %b), !dbg !39
    ret i32 0, !dbg !40
}
```

swap.ll



# Compiling a C Program to Its LLVM IR

## An example

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}

int main(){
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a, &b);
}
```

swap.c

Compile



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

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

swap.ll

<https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/src/swap.c>

# C code to LLVM IR

## An example

```
void swap(char **p, char **q){
    char* t = *p;
    *p = *q;
    *q = t;
}

int main(){
    char a1;
    char *a;
    char b1;
    char *b;
    a = &a1;
    b = &b1;
    swap(&a,&b);
}
```

swap.c

Compile



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

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

Function

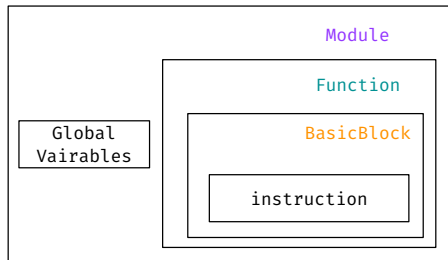
BasicBlock

Instruction

swap.ll

# LLVM Intermediate Representation (IR)

## Structure Organization



LLVM-IR Scopes

**Module** contains **Functions** and **Global Variables**

- Whole module is the unit of translation, analysis and optimization.

**Function** contains **BasicBlocks** and **Arguments**, which correspond to functions.

**BasicBlock** contains list of instructions.

- Each block is contiguous chunk of instructions

Instruction is opcode + vector of operands in '3-address' style

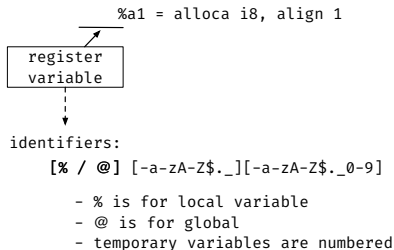
- All operands have types
- Instruction result is typed

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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

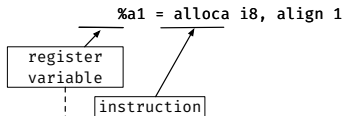


# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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



identifiers:

`[% / @] [-a-zA-Z$. _][-a-zA-Z$. _0-9]`

- % is for local variable
- @ is for global
- temporary variables are numbered

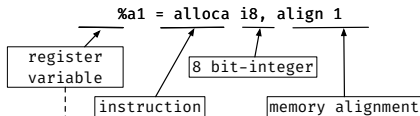
`alloca`: instruction allocates `i8` (sizeof) bytes of memory on runtime stack

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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



identifiers:

`[% / @] [-a-zA-Z$. _][-a-zA-Z$. _0-9]`

- % is for local variable
- @ is for global
- temporary variables are numbered

`alloca`: instruction allocates `i8` (`sizeof`) bytes of memory on runtime stack

`align`: indicates the memory operation should be aligned to 1 byte

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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

%a = alloca ptr, align 8



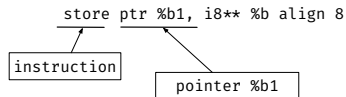
allocate 8-bit integer pointer for a

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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



store the pointer %b1 to the memory location that %b points to

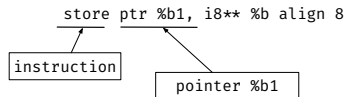


# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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



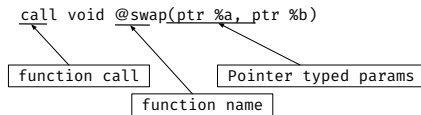
store the pointer %b1 to the memory location that %b points to

# LLVM Intermediate Representation (IR)

## LLVM Instructions

```
int main(){  
    char a1;  
    char *a;  
    char b1;  
    char *b;  
    a = &a1;  
    b = &b1;  
    swap(&a,&b);  
}
```

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



call instruction will be used to build control flow.

# Compiler IR Playground <https://godbolt.org/z/sra58dhbG>

The screenshot shows the Compiler Explorer interface. On the left, the C source code is displayed in a file named 'C'. The code defines a `swap` function and a `main` function that calls it. The right pane shows the LLVM IR output for the same code, generated using `x86_64 clang 9.0.1` with the flags `-c -S -fno-discard-value-names -Xclang -disable-O0-optnone -emit-llvm`. The IR includes metadata for debugging and declares variables `%p`, `%q`, and `%t` as pointers to 8-byte aligned memory.

```
1 void swap(char **p, char **q){
2     char* t = *p;
3     *p = *q;
4     *q = t;
5 }
6 int main(){
7     char a1;
8     char b1;
9     char *a;
10    char *b;
11    a = &a1;
12    b = &b1;
13    swap(&a,&b);
14 }
```

```
1 define dso_local void @swap(i8** %p, i8** %q) #0 !dbg !7 {
2     entry:
3     %p.addr = alloca i8**, align 8
4     %q.addr = alloca i8**, align 8
5     %t = alloca i8*, align 8
6     store i8** %p, i8*** %p.addr, align 8
7     call void @llvm.dbg.declare(metadata i8*** %p.addr, metadata !14, metadata !DIExpress:
8     store i8** %q, i8*** %q.addr, align 8
9     call void @llvm.dbg.declare(metadata i8*** %q.addr, metadata !16, metadata !DIExpress:
10    call void @llvm.dbg.declare(metadata i8** %t, metadata !18, metadata !DIExpression()),
11    %0 = load i8**, i8*** %p.addr, align 8, !dbg !20
12    %1 = load i8*, i8** %0, align 8, !dbg !21
13    store i8* %1, i8** %t, align 8, !dbg !19
14    %2 = load i8**, i8*** %q.addr, align 8, !dbg !22
15    %3 = load i8*, i8** %2, align 8, !dbg !23
16    %4 = load i8**, i8*** %p.addr, align 8, !dbg !24
17    store i8* %3, i8** %4, align 8, !dbg !25
18    %5 = load i8*, i8** %t, align 8, !dbg !26
19    %6 = load i8**, i8*** %q.addr, align 8, !dbg !27
20    store i8* %5, i8** %6, align 8, !dbg !28
21    ret void, !dbg !29
22 }
23
```

# LLVM Documentations

- LLVM Language Reference Manual <https://llvm.org/docs/LangRef.html>
- LLVM Programmer's Manual  
<https://llvm.org/docs/ProgrammersManual.html>
- Writing an LLVM Pass <http://llvm.org/docs/WritingAnLLVMPass.html>
- Tutorials for Clang/LLVM  
<https://freecompilercamp.org/clang-llvm-landing>
- LLVM Tutorial IEEE SecDev 2020 [https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4\\_copy.pdf](https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4_copy.pdf)

# SVFIR and Graph Representation of Code

## (Week 2)

Yulei Sui

School of Computer Science and Engineering  
University of New South Wales, 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 (version 16.0.0) with over 200 KLOC C/C++ code and **1300+ stars with 80+ contributors** and over 4K 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 (version 16.0.0) with over 200 KLOC C/C++ code and **1300+ stars with 80+ contributors** and over 4K 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?

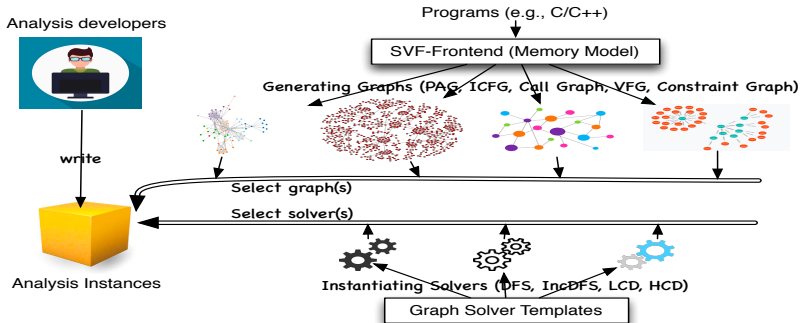
# SVF : Static Value-Flow Analysis Framework for Source Code

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 (version 16.0.0) with over 200 KLOC C/C++ code and **1300+ stars with 80+ contributors** and over 4K 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.

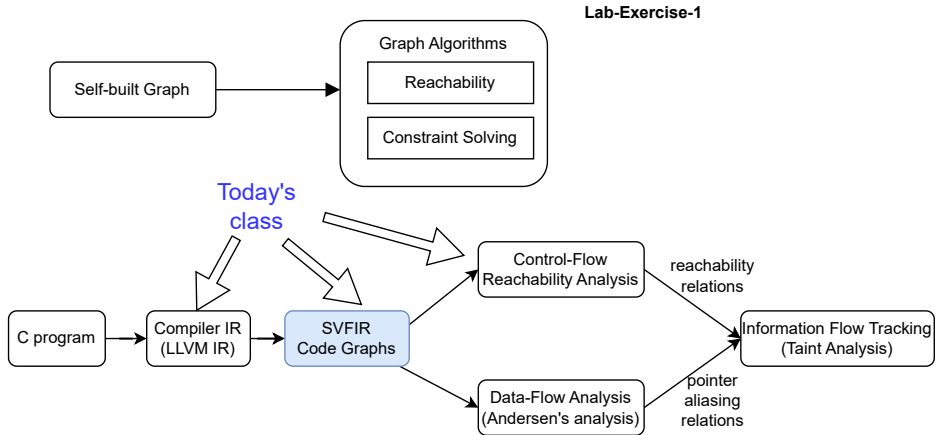


# SVF: Design Principle



- 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)

# Today's Class

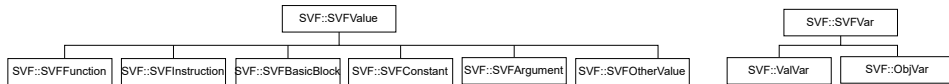


## SVFIR and Why?

- **SVFIR = SVFValue + SVFVar + SVFStmt + Code Graphs**
- A **lightweight wrapper and abstraction** with fewer types of LLVM instructions/vars. Complicated ones are broken down into basic SVFStmts.
- SVFIR aims to accommodate (translate from) any modern language for fast **static analysis** prototyping but **NOT** for code generation or optimizations.

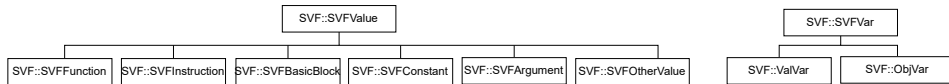
# SVFIR and Why?

- **SVFIR = SVFValue + SVFVar + SVFStmt + Code Graphs**
- A **lightweight wrapper and abstraction** with fewer types of LLVM instructions/vars. Complicated ones are broken down into basic SVFStmts.
- SVFIR aims to accommodate (translate from) any modern language for fast **static analysis** prototyping but **NOT** for code generation or optimizations.
- SVFValue and SVFVar: program values and variables

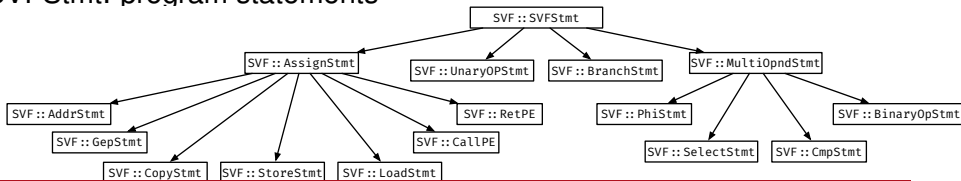


# SVFIR and Why?

- **SVFIR = SVFValue + SVFVar + SVFStmt + Code Graphs**
- A **lightweight wrapper and abstraction** with fewer types of LLVM instructions/vars. Complicated ones are broken down into basic SVFStmts.
- SVFIR aims to accommodate (translate from) any modern language for fast **static analysis** prototyping but **NOT** for code generation or optimizations.
- SVFValue and SVFVar: program values and variables



- SVFStmt: program statements



# Mapping between SVF-IR and LLVM-IR

- **1:1 mapping from an LLVM::Value to one of SVFValue's six subclasses.**
  - `LLVM::Value  $\Leftarrow$  LLVMModuleSet::getSVFValue(llvm_value)`
  - `SVFValue  $\Leftarrow$  LLVMModuleSet::getLLVMValue(svf_value)`

# Mapping between SVF-IR and LLVM-IR

- **1:1 mapping from an LLVM::Value to one of SVFValue's six subclasses.**
  - `LLVM::Value  $\Leftarrow$  LLVMModuleSet::getSVFValue(llvm_value)`
  - `SVFValue  $\Leftarrow$  LLVMModuleSet::getLLVMValue(svf_value)`
- Retrieve an SVFVar (either ValVar or ObjVar) from an SVFValue.
  - **1:1 mapping from each SVFValue to an ValVar. Only memory-allocation-related SVFValue can map to ObjVar.**
  - `ValVar  $\Leftarrow$  SVFIR::getNode(SVFIR::getValueNode(svf_value));`
  - `ObjVar  $\Leftarrow$  SVFIR::getNode(SVFIR::getObjectNode(svf_value));`

# Mapping between SVF-IR and LLVM-IR

- **1:1 mapping from an LLVM::Value to one of SVFValue's six subclasses.**
  - `LLVM::Value  $\Leftarrow$  LLVMModuleSet::getSVFValue(llvm_value)`
  - `SVFValue  $\Leftarrow$  LLVMModuleSet::getLLVMValue(svf_value)`
- Retrieve an SVFVar (either ValVar or ObjVar) from an SVFValue.
  - **1:1 mapping from each SVFValue to an ValVar. Only memory-allocation-related SVFValue can map to ObjVar.**
  - `ValVar  $\Leftarrow$  SVFIR::getNode(SVFIR::getValueNode(svf_value));`
  - `ObjVar  $\Leftarrow$  SVFIR::getNode(SVFIR::getObjectNode(svf_value));`
- **1:1 mapping from an SVFInstruction to an ICFGNode (containing one or more SVFStmts).**
  - `ICFGNode  $\Leftarrow$  ICFG::getICFGNode(svf_inst);`
  - `SVFStmts  $\Leftarrow$  ICFGNode::getSVFStmts();`
- Always use the toString method in SVF's data structures to understand their meanings and the mappings.
  - `SVFVar::toString()`
  - `ICFGNode::toString()`    `ICFGEde::toString()`



# 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 (constant data, stack, heap, global) (function objects are considered as global objects)
FIObjVar	$\mathbf{o} \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$	A single (base) memory object
GepObjVar	$\mathbf{o}_i \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H}) \times \mathbb{P}$	$i$ -th subfield/element of an (aggregate) object
ConstantData	$\mathbb{C}$	Constant data (e.g., numbers and strings)
Program Statement	$\mathbf{l} \in \mathbb{L}$	Statements labels

# SVF Program Statements (SVFStmt)

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

SVFStmt	LLVM-Like form	C-Like form	Operand types
AddrStmt	<code>%ptr = alloca<sub>o</sub></code>	<code>p = alloc</code>	$\mathbb{P} \times \mathbb{O}$
ConstStmt	<code>%ptr = constantData</code>	<code>p = c</code>	$\mathbb{P} \times \mathbb{C}$
CopyStmt	<code>%p = bitcast %q</code>	<code>p = q</code>	$\mathbb{P} \times \mathbb{P}$
LoadStmt	<code>%p = load %q</code>	<code>p = *q</code>	$\mathbb{P} \times \mathbb{P}$
StoreStmt	<code>store %p, %q</code>	<code>*p = q</code>	$\mathbb{P} \times \mathbb{P}$
GepStmt	<code>%p = getelementptr %q, %i</code>	<code>p = &amp;(q → i) or p = &amp;q[i]</code>	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
PhiStmt	<code>%p = phi [ l<sub>1</sub>, %q<sub>1</sub> ], [ l<sub>2</sub>, %q<sub>2</sub> ]</code>	<code>p = phi(l<sub>1</sub> : q<sub>1</sub>, l<sub>2</sub> : q<sub>2</sub>)</code>	$\mathbb{P} \times (\mathbb{L} \times \mathbb{P})^2$
BranchStmt	<code>br i1 %p, label %l<sub>1</sub>, label %l<sub>2</sub></code>	<code>if (p) l<sub>1</sub> else l<sub>2</sub></code>	$\mathbb{P} \times \mathbb{L}^2$
UnaryOPStmt	<code>p = ¬q</code>	<code>p = ¬q</code>	$\mathbb{P} \times \mathbb{P}$
BinaryOPStmt/CmpStmt	<code>r = ⊗ p, q</code>	<code>r = p ⊗ q</code>	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
CallPE	<code>%r = call f(...%q<sub>i</sub>...)</code>	<code>r = f(..., q<sub>i</sub>, ...)</code>	$(\mathbb{P} \times \mathbb{P})^n$
	<code>f(...%p<sub>i</sub>...){ ... ret %z }</code>	<code>f(..., p<sub>i</sub>, ...){... return z }</code>	
RetPE	<code>%r = %z</code>	<code>r = z</code>	$\mathbb{P} \times \mathbb{P}$

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

# 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,  $*p = 3$ ;  $\Rightarrow$   $t = 3$ ;  $*p = t$ ;
  - $t$  is a temporal ValVar. For any constant (e.g., 3) SVF will create a ValVar and an ObjVar.
- 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 Representations 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 Representations 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  
  %a = alloca ptr, align 8  
  %b1 = alloca i8, align 1  
  %b = alloca ptr, align 8  
  store ptr %a1, ptr %a, align 8  
  store ptr %b1, ptr %b, align 8  
  call void @swap(ptr %a, ptr %b)  
  ret i32 0  
}
```

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



## Call Graph

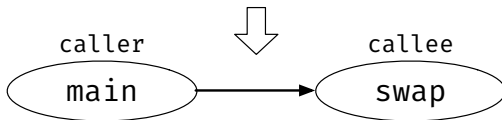
<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#3-call-graph>

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

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



Call Graph

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

1. Each node represents a program method
2. Each edge represents a calling relation between two program methods

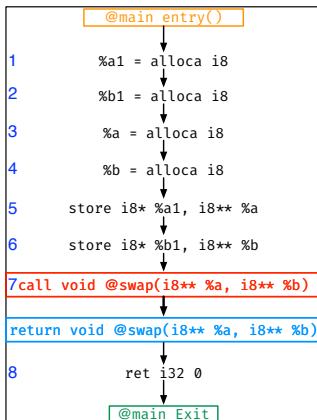
# Control Flow Graph

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.

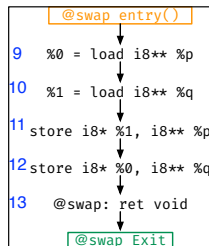


# Intra-procedural Control Flow Graph



Program execution order between instructions

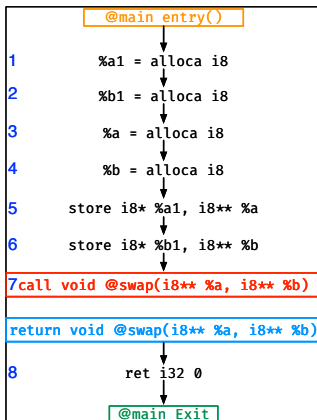
- Each node represents an instruction or a statement
- Each edge represents a control-flow dependence between two nodes



- IntraICFGNode
- FunEntryICFGNode
- FunExitICFGNode
- RetICFGNode
- CallICFGNode

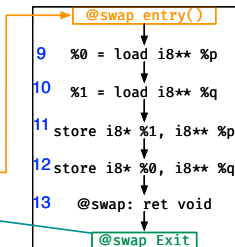
<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph>

# 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



- IntraICFGNode
- FunEntryICFGNode
- FunExitICFGNode
- RetICFGNode
- CallICFGNode

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

# SVF IR Example (SVFVar and SVFStmt)

`clang -S -c -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm example.c -o example.ll`

```
1 int foo(int b){  
2     return b;  
3 }  
4 int main(){  
5     int a = foo(0);  
6 }
```

# SVF IR Example (SVFVar and SVFStmt)

`clang -S -c -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm example.c -o example.ll`

```
1 int foo(int b){
2     return b;
3 }
4 int main(){
5     int a = foo(0);
6 }

1 define i32 @foo(i32 %b) {
2 entry:
3     %b.addr = alloca i32
4     store i32 %b, ptr %b.addr,
5     %0 = load i32, ptr %b.addr,
6     ret i32 %0
7 }
8 define i32 @main() {
9     %a = alloca i32
10    %call = call i32 @foo(i32 0)
11    store i32 %call, i32* %a
12    ret i32 0
13 }
```

# SVF IR Example (SVFVar and SVFStmt)

`clang -S -c -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm example.c -o example.ll`

```
1 int foo(int b){
2     return b;
3 }
4 int main(){
5     int a = foo(0);
6 }

1 define i32 @foo(i32 %b) {
2 entry:
3     %b.addr = alloca i32
4     store i32 %b, ptr %b.addr,
5     %0 = load i32, ptr %b.addr,
6     ret i32 %0
7 }
8 define i32 @main() {
9     %a = alloca i32
10    %call = call i32 @foo(i32 0)
11    store i32 %call, i32* %a
12    ret i32 0
13 }
```

Variables introduced by SVF  
(created internally)

SVFVar	Meaning
DummyValVar ID: 0	nullptr
DummyValVar ID: 1	reserved
DummyObjVar ID: 2	reserved
DummyObjVar ID: 3	reserved
ValVar ID: 4	foo
FIObjVar ID: 5	foo (object)
RetPN ID: 6	ret of foo
ValVar ID: 15	main
FIObjVar ID: 16	main (object)
RetPN ID: 17	ret of main
ValVar ID: 21	0
FIObjVar ID: 22	0 (object)

# SVF IR Example (SVFVar and SVFStmt)

`clang -S -c -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm example.c -o example.ll`

```
1 int foo(int b){
2     return b;
3 }
4 int main(){
5     int a = foo(0);
6 }

1 define i32 @foo(i32 %b) {
2 entry:
3     %b.addr = alloca i32
4     store i32 %b, ptr %b.addr,
5     %0 = load i32, ptr %b.addr,
6     ret i32 %0
7 }
8 define i32 @main() {
9     %a = alloca i32
10    %call = call i32 @foo(i32 0)
11    store i32 %call, i32* %a
12    ret i32 0
13 }
```

Variables introduced by SVF  
(created internally)

SVFVar	Meaning
DummyValVar ID: 0	nullptr
DummyValVar ID: 1	reserved
DummyObjVar ID: 2	reserved
DummyObjVar ID: 3	reserved
ValVar ID: 4	foo
FIObjVar ID: 5	foo (object)
RetPN ID: 6	ret of foo
ValVar ID: 15	main
FIObjVar ID: 16	main (object)
RetPN ID: 17	ret of main
ValVar ID: 21	0
FIObjVar ID: 22	0 (object)

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: 10	i32 1 { constant data }
FIObjVar ID: 11	i32 1 { constant data }
ValVar ID: 12	store i32 %b, ptr %b.addr
ValVar ID: 13	%0 = load i32, ptr %b.addr
ValVar ID: 14	ret i32 %0
ValVar ID: 18	%a = alloca i32
FIObjVar ID: 19	%a = alloca i32
ValVar ID: 20	%call = call i32 @foo(i32 0)
ValVar ID: 23	store i32 %call, ptr %a
ValVar ID: 24	ret i32 0

# ICFG and SVFStmt Example

```
1 define i32 @foo(i32 %b) {  
2 entry:  
3   %b.addr = alloca i32  
4   store i32 %b, ptr %b.addr,  
5   %0 = load i32, ptr %b.addr,  
6   ret i32 %0  
7 }  
8  
9 define i32 @main() {  
10  %a = alloca i32  
11  %call = call i32 @foo(i32 0)  
12  store i32 %call, i32* %a  
13  ret i32 0  
14 }
```

ICFGNode	SVFStmt	LLVM Value
GlobalICFGNode0	CopyStmt: [Var1 ← Var0]	ptr null (constant data)
	AddrStmt: [Var21 ← Var22]	i32 0 (constant data)
	AddrStmt: [Var10 ← Var11]	i32 1 (constant data)
	AddrStmt: [Var4 ← Var5]	foo
	AddrStmt: [Var15 ← Var16]	main
FunEntryICFGNode1	fun: foo	
IntraICFGNode2	AddrStmt: [Var8 ← Var9]	%b.addr = alloca i32
IntraICFGNode3	StoreStmt [Var8 ← Var7]	store i32 %b, ptr %b.addr
IntraICFGNode4	LoadStmt: [Var13 ← Var8]	%0 = load i32, ptr %b.addr
IntraICFGNode5	fun:foo	ret i32 %0
FunExitICFGNode6	PhiStmt: [Var6 ← ([Var13, ICFGNode5],)]	ret i32 %0
FunEntryICFGNode7	fun: main	
IntraICFGNode8	AddrStmt: [Var18 ← Var19]	%a = alloca i32
CallICFGNode9	CallPE: [Var7 ← Var21]	%call = call i32 @foo(i32 0)
RetICFGNode10	RetPE: [Var20 ← Var6]	%call = call i32 @foo(i32 0)
IntraICFGNode11	StoreStmt: [Var18 ← Var20]	store i32 %call, ptr %a
IntraICFGNode12	fun: main	ret i32 0
FunExitICFGNode13	PhiStmt: [Var17 ← ([Var21, ICFGNode12],)]	ret i32 0

<https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR>

# Control-Flow and Reachability Analysis

(Week 2)

Yulei Sui

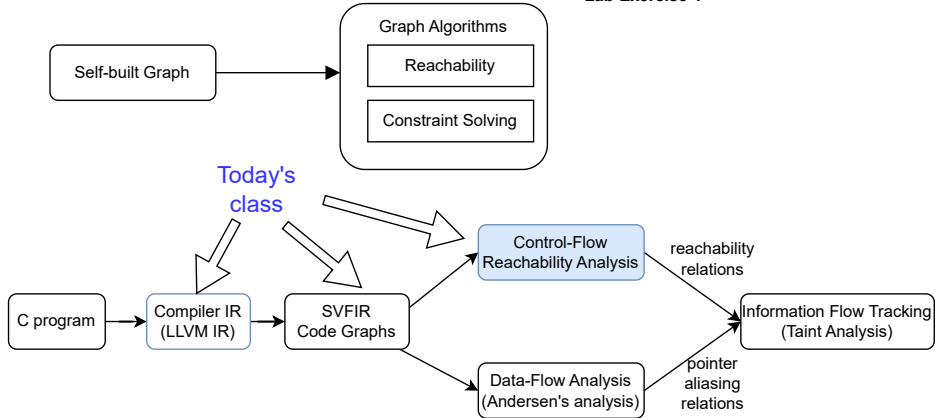
School of Computer Science and Engineering

University of New South Wales, Australia



# Today's Class

## Lab-Exercise-1



# Control-Flow and Data-Flow

## What are control-flow and data-flow?

- **Control-control or control-dependence**
  - Execution order between two program statements/instructions.
  - Can program point B be reached from point A in the control-flow graph of a program?
  - Obtained through traversing the ICFG of a program
- **Data-data or data-dependence**
  - Definition-use relation between two program variables.
  - Will the definition of a variable X be used and passed to another variable Y?
  - Obtained through analyzing the SVFIR of a program
  - Combining SVFIR with ICFG to conduct symbolic execution (mimic the runtime path-based execution) of a program.

# Program Dependence

## Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

# Program Dependence

## Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.

# Program Dependence

## Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

# Program Dependence

## Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

- **Applications of data-dependence**

- Pointer alias analysis: statically determine possible runtime values of a pointer to detect memory errors, such as null pointer dereferences and use-after-frees.

# Program Dependence

## Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

- **Applications of data-dependence**

- Pointer alias analysis: statically determine possible runtime values of a pointer to detect memory errors, such as null pointer dereferences and use-after-frees.
- Taint analysis: if two program variables  $v_1$  and  $v_2$  are aliases (e.g., representing the same memory location), if  $v_1$  is tainted by user inputs, then  $v_2$  is also tainted.
- ...

# Control-Flow

We say that a program statement (ICFG node) `snk` is control-flow dependent on `src` if `src` can reach `snk` on the ICFG.

- Context-insensitive control-dependence
  - control-flow traversal without matching calls and returns.
  - fast but imprecise



# Control-Flow

We say that a program statement (ICFG node) `snk` is control-flow dependent on `src` if `src` can reach `snk` on the ICFG.

- Context-insensitive control-dependence
  - control-flow traversal without matching calls and returns.
  - fast but imprecise
- Context-sensitive control-dependence
  - control-flow traversal by matching calls and returns.
  - precise but maintains an extra abstract call stack (storing a sequence of callsite ID information) to mimic the runtime call stack.

# Control-Flow

```
int bar(int s){
    return s;
}
int main(){
    int a = source();
    if (a > 0){
        int p = bar(a);
        sink(p);
    }else{
        int q = bar(a);
        sink(q);
    }
}
```

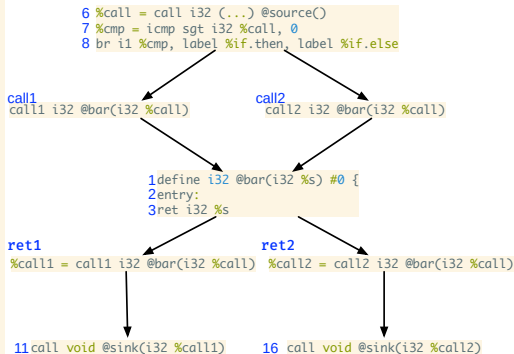
<https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/src/control-flow.c>

# Control-Flow

```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:                ; preds = %entry  
9 %call1 = call i32 @bar(i32 %call)  
10 call void @sink(i32 %call1)  
11 br label %if.end  
12  
13 if.else:                ; preds = %entry  
13 %call2 = call i32 @bar(i32 %call)  
14 call void @sink(i32 %call2)  
15 br label %if.end  
16  
17 if.end:                ; preds = %if.else, %if.then  
17 ret i32 0  
18}
```

# Control-Flow

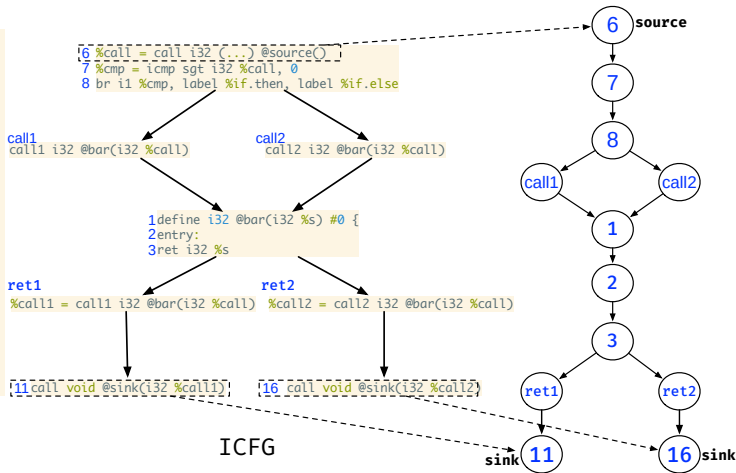
```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:                ; preds = %entry  
9 %call1 = call i32 @bar(i32 %call)  
10 call void @sink(i32 %call1)  
11 br label %if.end  
12  
13 if.else:                ; preds = %entry  
13 %call2 = call i32 @bar(i32 %call)  
14 call void @sink(i32 %call2)  
15 br label %if.end  
16  
17 if.end:                 ; preds = %if.else, %if.then  
17 ret i32 0  
18}
```



ICFG

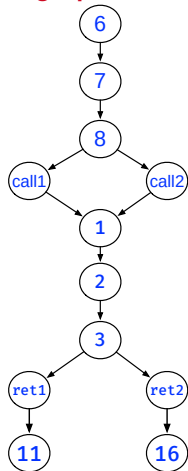
# Control-Flow

```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:                ; preds = %entry  
9 %call1 = call i32 @bar(i32 %call)  
10 call void @sink(i32 %call1)  
11 br label %if.end  
12  
13 if.else:                ; preds = %entry  
13 %call2 = call i32 @bar(i32 %call)  
14 call void @sink(i32 %call2)  
15 br label %if.end  
16  
17 if.end:                 ; preds = %if.else, %if.then  
17 ret i32 0  
18}
```



# Context-Insensitive Control-Flow

## Obtaining a path from source to sink on ICFG



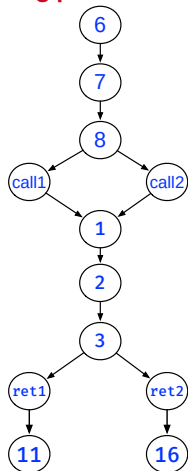
Basic DFS on ICFG: source  $\rightarrow$  sink

```
visited: set<NodeID>  
path: vector<NodeID>
```

```
DFS(visited, path, src, dst)  
  visited.insert(src);  
  path.push_back(src);  
  if src == dst then  
    Print path;  
  foreach edge e  $\in$  outEdges(src) do  
    if (e.dst  $\notin$  visited)  
      DFS(visited, path, e.dst, dst);  
  visited.erase(src);  
  path.pop_back();
```

# Context-Insensitive Control-Flow

## Obtaining paths from node 6 to node 11 on the ICFG



Basic DFS on ICFG: source  $\rightarrow$  sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e  $\in$  outEdges(src) do
    if (e.dst  $\notin$  visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6  $\rightarrow$  node 11

Path 1:

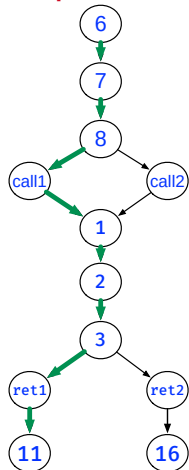
6  $\rightarrow$  7  $\rightarrow$  8  $\rightarrow$  **call1**  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  **ret1**  $\rightarrow$  11

Path 2:

6  $\rightarrow$  7  $\rightarrow$  8  $\rightarrow$  **call2**  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  **ret1**  $\rightarrow$  11

# Context-Insensitive Control-Flow

Feasible paths from node 6 to node 11



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 11

Path 1: **feasible path**

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret1** → 11

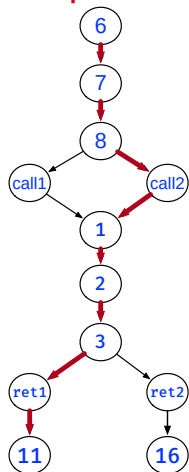
Path 2:

6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret1** → 11



# Context-Insensitive Control-Flow

Infeasible path from node 6 to node 11



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 11

Path 1:

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret1** → 11

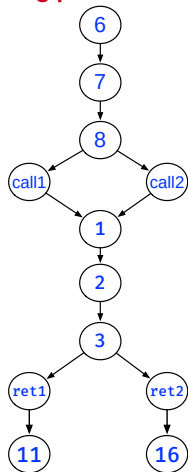
Path 2:

6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret1** → 11

**spurious path**

# Context-Insensitive Control-Flow

## Obtaining paths from node 6 to node 16 on ICFG



Basic DFS on ICFG: source  $\rightarrow$  sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
    visited.insert(src);
    path.push_back(src);
    if src == dst then
        Print path;
    foreach edge e  $\in$  outEdges(src) do
        if (e.dst  $\notin$  visited)
            DFS(visited, path, e.dst, dst);
    visited.erase(src);
    path.pop_back();
```

ICFG paths: node 6  $\rightarrow$  node 16

Path 3:

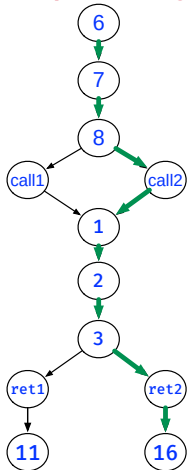
6  $\rightarrow$  7  $\rightarrow$  8  $\rightarrow$  **call2**  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  **ret2**  $\rightarrow$  16

Path 4:

6  $\rightarrow$  7  $\rightarrow$  8  $\rightarrow$  **call1**  $\rightarrow$  1  $\rightarrow$  2  $\rightarrow$  3  $\rightarrow$  **ret2**  $\rightarrow$  16

# Context-Insensitive Control-Flow

Feasible paths using from node 6 to node 16 on the ICFG



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 16

Path 3: **feasible path**

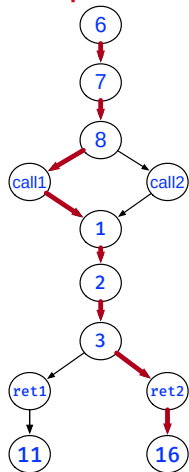
6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret2** → 16

Path 4:

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret2** → 16

# Context-Insensitive Control-Flow

Infeasible paths using from node 6 to node 16 on the ICFG



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 16

Path 3:

6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret2** → 16

Path 4:

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret2** → 16

**spurious path**

# Context-Sensitive Control-Dependence

An extension of the context-insensitive algorithm by matching calls and returns.

- Get only feasible interprocedural paths and exclude infeasible ones
- Requires an extra callstack to store and mimic the runtime calling relations.

# Context-Sensitive Control-Flow (Algorithm)

---

**Algorithm 1: 1** Context sensitive control-flow reachability

---

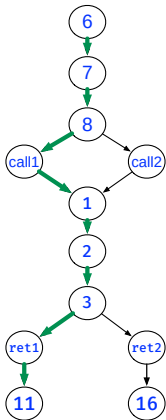
**Input :** curNode : ICFGNode    snk : ICFGNode    path : vector(ICFGNode)    callstack : vector(SVFInstruction)  
visited : set(ICFGNode, callstack);

```
1 dfs(curNode, dst)
2   pair = <curNode, callstack>;
3   if pair ∈ visited then
4   |   return;
5   visited.insert(pair);
6   path.push_back(curNode);
7   if src == snk then
8   |   collectICFGPath(path);
9   foreach edge ∈ curNode.getOutEdges() do
10  |   if edge.isIntraCFGEde() then
11  |   |   dfs(edge.dst, snk);
12  |   else if edge.isCallCFGEde() then
13  |   |   callstack.push_back(edge.getCallSite());
14  |   |   dfs(edge.dst, snk);
15  |   |   callstack.pop_back();
16  |   else if edge.isRetCFGEde() then
17  |   |   if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18  |   |   |   callstack.pop_back();
19  |   |   |   dfs(edge.dst, snk);
20  |   |   |   callstack.push_back(edge.getCallSite());
21  |   |   else if callstack == ∅ then
22  |   |   |   dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

---

# Context-Sensitive Control-Dependence (Example)

call1 matches with ret1

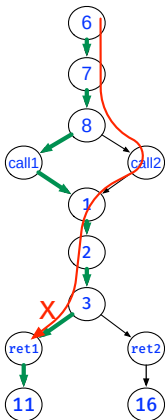


## Algorithm 2: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector<ICFGNode>  
        callstack : vector<SVFInstruction>  visited : set<ICFGNode, callstack>;  
1 dfs(curNode, dst)  
2   pair = (curNode, callstack);  
3   if pair ∈ visited then  
4     return;  
5   visited.insert(pair);  
6   path.push_back(curNode);  
7   if src == snk then  
8     collectICFGPath(path);  
9   foreach edge ∈ curNode.getOutEdges() do  
10    if edge.isIntraCFGEde() then  
11      dfs(edge.dst, snk);  
12    else if edge.isCallCFGEde() then  
13      callstack.push_back(edge.getCallSite());  
14      dfs(edge.dst, snk);  
15      callstack.pop_back();  
16    else if edge.isRetCFGEde() then  
17      if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then  
18        callstack.pop_back();  
19        dfs(edge.dst, snk);  
20        callstack.push_back(edge.getCallSite());  
21      else if callstack == ∅ then  
22        dfs(edge.dst, snk);  
23   visited.erase(pair);  
24   path.pop_back();
```

# Context-Sensitive Control-Dependence (Example)

call2 does not match with ret1



## Algorithm 3: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector<ICFGNode>
        callstack : vector<SVFInstruction>  visited : set<ICFGNode, callstack>;
1 dfs(curNode, dst)
2   pair = (curNode, callstack);
3   if pair ∈ visited then
4     return;
5   visited.insert(pair);
6   path.push_back(curNode);
7   if src == snk then
8     collectICFGPath(path);
9   foreach edge ∈ curNode.getOutEdges() do
10    if edge.isIntraCFGEde() then
11      dfs(edge.dst, snk);
12    else if edge.isCallCFGEde() then
13      callstack.push_back(edge.getCallSite());
14      dfs(edge.dst, snk);
15      callstack.pop_back();
16    else if edge.isRetCFGEde() then
17      if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18        callstack.pop_back();
19        dfs(edge.dst, snk);
20        callstack.push_back(edge.getCallSite());
21      else if callstack == ∅ then
22        dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```



# What's next?

- Understand control-flow reachability in this slides
- Debug and work with the code under the SVFIR and CodeGraph folders
- If you finished Quiz-1 and Lab-Exercise-1, you could have a look at the spec of Assignment-1. Once the data flow is taught in Week 3, you could start coding Assignment-1
  - Assignment-1's specification: <https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-1>