# LLVM, SVF IR and Control-Flow Reachability

## (Week 2)
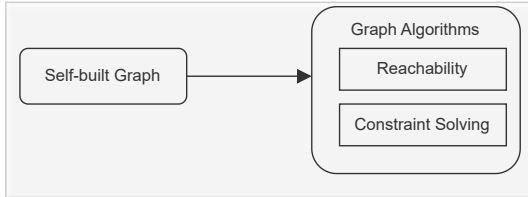
Yulei Sui

School of Computer Science and Engineering
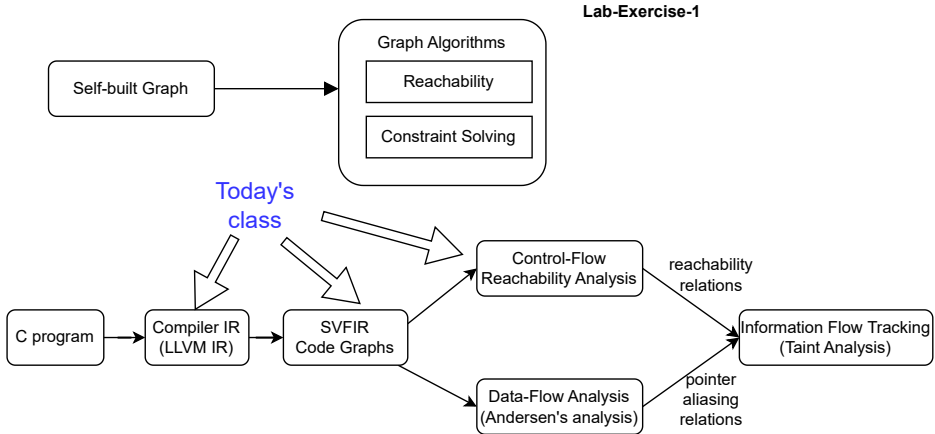
University of New South Wales, Australia
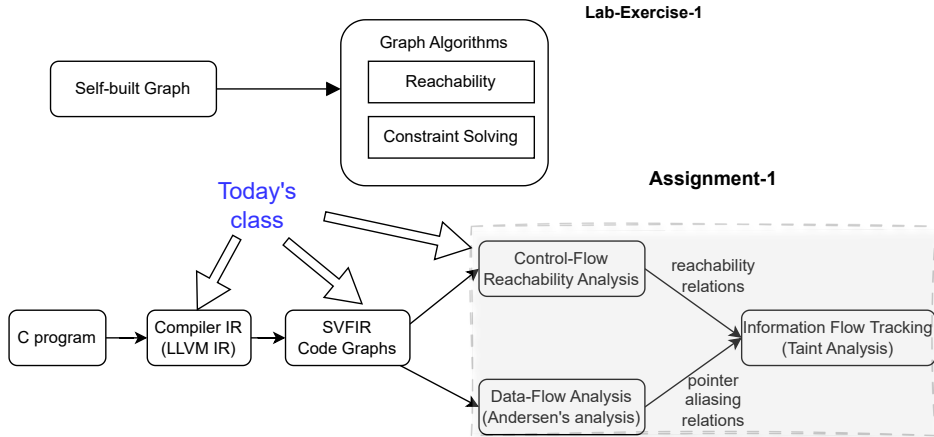
# Where We Are Now and Today's Class

# Today's Class

**Lab-Exercise-1**

Graph Algorithms

Reachability

Constraint Solving

Self-built Graph

Today's class

C program

Compiler IR (LLVM IR)

SVFIR Code Graphs

Control-Flow Reachability Analysis

reachability relations

Data-Flow Analysis (Andersen's analysis)

pointer aliasing relations

Information Flow Tracking (Taint Analysis)

# Today's Class



Lab-Exercise-1

Self-built Graph → Graph Algorithms
- Reachability
- Constraint Solving

Assignment-1

Today's class

C program → Compiler IR (LLVM IR) → SVFIR Code Graphs → Control-Flow Reachability Analysis → reachability relations → Information Flow Tracking (Taint Analysis)

Data-Flow Analysis (Andersen's analysis) → pointer aliasing relations → Information Flow Tracking (Taint Analysis)
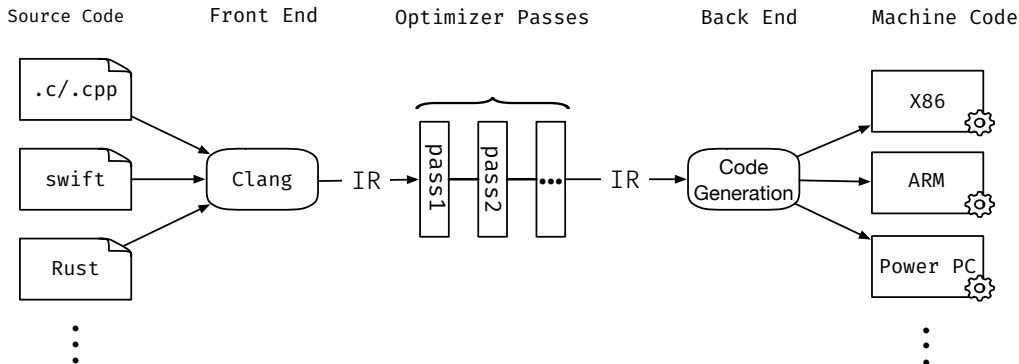
# What is LLVM ?

**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/`

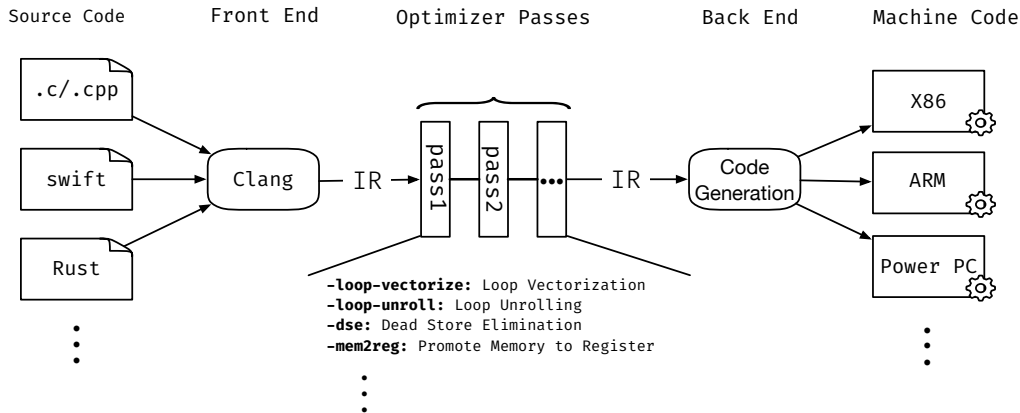# Why Learn LLVM or Learn Compilers in General?

- 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



Source Code     Front End     Optimizer Passes     Back End     Machine Code

.c/.cpp

swift → Clang → IR → pass1 pass2 ••• → IR → Code Generation → X86 / ARM / Power PC

Rust

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

# LLVM's Architecture



Source Code     Front End     Optimizer Passes     Back End     Machine Code

```
.c/.cpp
swift
Rust
```

Clang → IR → pass1 pass2 ••• → IR → Code Generation

X86
ARM
Power PC

**–loop-vectorize:** Loop Vectorization
**–loop-unroll:** Loop Unrolling
**–dse:** Dead Store Elimination
**–mem2reg:** Promote Memory to Register

\*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
  - LLVM IR is higher level than assembly (e.g., call, invoke, switch, aggregate types, etc.).
  - 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)
- **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)

- **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.

```
x = 1;
x = x + 2;
```
Translate to SSA form $\Rightarrow$
```
x1 = 1;
x2 = x1 + 2;
```

# 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)

- **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.

# 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)

- **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 ⟹

```
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**
  - Three addresses and one operator. Each instruction typically 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

- Get your correct version of `clang`, by typing `source env.sh`
  - `clang -v` // make sure clang/llvm version 16.0
  - `source env.sh`
- Compile `swap.c` to a human readable IR `swap.ll` (default opt level `-O0`).
  - `clang -c -S -emit-llvm swap.c -o swap.ll`
- Keep the variable names.
  - `clang -c -S -fno-discard-value-names -emit-llvm swap.c -o swap.ll`
- Remove `optnone` attribute to perform certain analyses or transformations
  - `clang -c -S -Xclang -disable-O0-optnone -fno-discard-value-names -emit-llvm swap.c -o swap.ll`
- Keep source code debug information (optional).
  - `clang -g -c -S -fno-discard-value-names -Xclang -disable-O0-optnone -emit-llvm swap.c -o swap.ll`
- Convert the LLVM IR to more compact static single assignment form.
  - `opt -p=mem2reg -S swap.ll -o swap.ll`

# An Example <u>without</u> Source Code Debug Info

```c
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 -p=mem2reg ...

⟹

```llvm
define void @swap(ptr %p, ptr %q) #0 !9 {
entry:
  %0 = load ptr, ptr %q, 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, !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
}
```

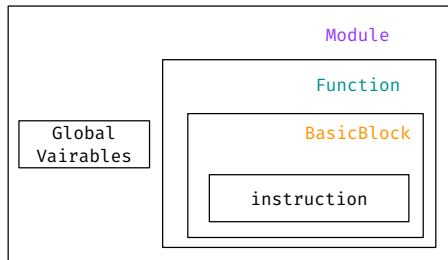swap.c                                                                    swap.ll

14

# An Example <u>with</u> Source Code Debug Info (`compile.sh`)

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

# Mapping C code to LLVM IR

```c
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

```llvm
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
}
```
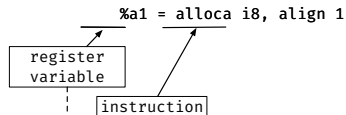
Function

BasicBlock

Instruction

swap.ll

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

# 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)

```
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
}
```

```
%a1 = alloca i8, align 1
```

register
variable

identifiers:
    **[% / @]** [-a-zA-Z$._][-a-zA-Z$._0-9]

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

# LLVM Intermediate Representation (IR)

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

```llvm
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
}
```

```
%a1 = alloca i8, align 1
```

register variable

instruction

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)

```
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
}
```
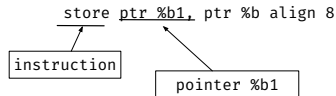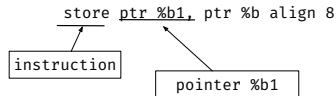
%a1 = alloca i8, align 1

register variable

instruction

8 bit-integer

memory alignment
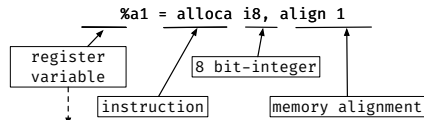
identifiers:
   [% / @] [-a-zA-Z$._][-a-zA-Z$._0-9]

   - % 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

- align 8   specifies that the allocated memory should be aligned on an 8-byte boundary. For example, its address could be 0x0008, 0x0010, 0x0018, 0x0020, and so on, but not 0x0001, 0x0002, or any other value that isn't a multiple of 8.

# LLVM Intermediate Representation (IR)

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

```llvm
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
}
```
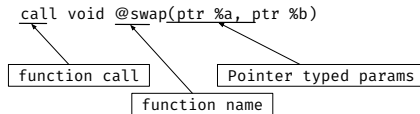
```llvm
%a = alloca ptr, align 8
```

allocate 8-bit integer pointer for a

# LLVM Intermediate Representation (IR)

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

```llvm
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 ptr %b1, ptr %b align 8

instruction

pointer %b1

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

# LLVM Intermediate Representation (IR)

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

```llvm
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 ptr %b1, ptr %b align 8

instruction

pointer %b1

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

# LLVM Intermediate Representation (IR)

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

```llvm
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
call void @swap(ptr %a, ptr %b)
```

| function call |   | function name |   | Pointer typed params |

call instruction will be used to
build control flow.

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

COMP6131 Software Security Analysis 2025

# 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`
- Writing a compiler with LLVM
  `https://www.youtube.com/watch?v=vrRXIQDCCEk`
- LLVM Tutorial IEEE SecDev 2020 `https://cs.rochester.edu/u/ejohns48/secdev19/secdev20-llvm-tutorial-version4_copy.pdf`

# SVF and Graph Representations of Code

## (Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

# SVF : <u>S</u>tatic <u>V</u>alue-<u>F</u>low Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural static analysis and verification 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 keynote in EuroLLVM 2016, and awarded an ICSE 2018 Distinguished Paper, an SAS Best Paper 2019 and an OOPSLA 2020 Distinguished Paper, **Google Aspire Award** 2023 and **Amazon Research Award** 2025.

# SVF : <u>S</u>tatic <u>V</u>alue-<u>F</u>low Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural static analysis and verification 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 keynote in EuroLLVM 2016, and awarded an ICSE 2018 Distinguished Paper, an SAS Best Paper 2019 and an OOPSLA 2020 Distinguished Paper, **Google Aspire Award** 2023 and **Amazon Research Award** 2025.

- 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 $F'$?
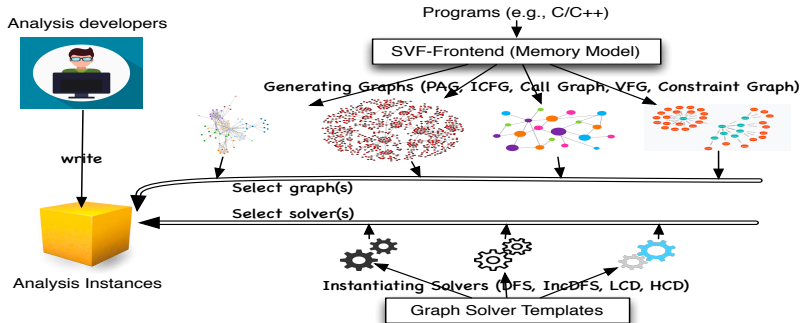  - Is there an unsafe memory access that may trigger a bug or security risk?

# SVF : <u>S</u>tatic <u>V</u>alue-<u>F</u>low Analysis Framework for Source Code

**A scalable, precise and on-demand** interprocedural static analysis and verification 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 keynote in EuroLLVM 2016, and awarded an ICSE 2018 Distinguished Paper, an SAS Best Paper 2019 and an OOPSLA 2020 Distinguished Paper, **Google Aspire Award** 2023 and **Amazon Research Award** 2025.

- 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



Programs (e.g., C/C++)

SVF-Frontend (Memory Model)

Generating Graphs (PAG, ICFG, Call Graph, VFG, Constraint Graph)

Analysis developers

write

Select graph(s)

Select solver(s)

Analysis Instances

Instantiating Solvers (DFS, IncDFS, LCD, HCD)

Graph Solver Templates

- 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)
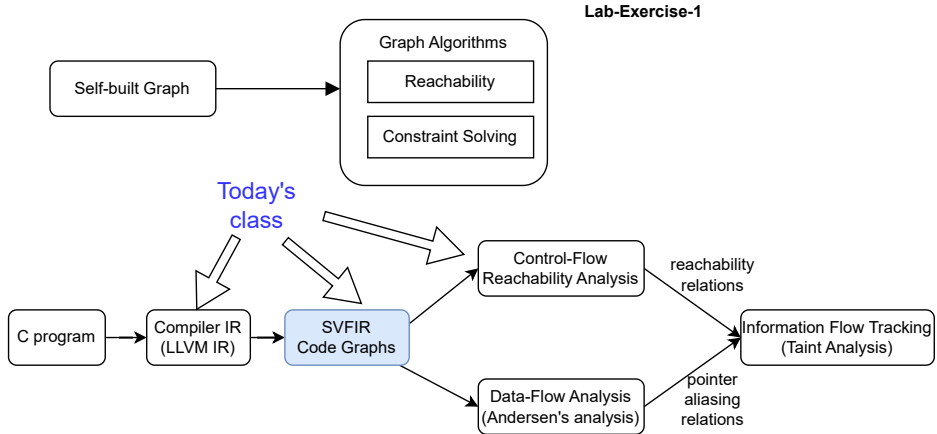
29

# Today's Class

**Lab-Exercise-1**

# SVFIR and Why?

- **SVFIR = 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 any modern language for fast **static analysis** prototyping but **NOT** for code generation or optimizations.
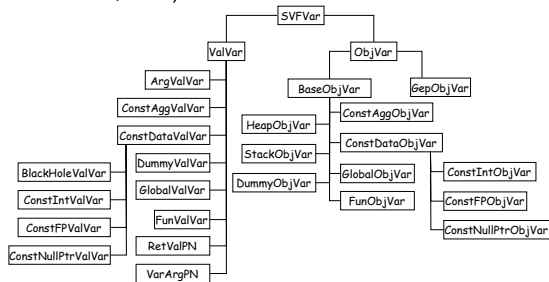
# SVFIR and Why?

- **SVFIR = 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 any modern language for fast **static analysis** prototyping but **NOT** for code generation or optimizations.
- **SVFValue** = **SVFVar** (program values) + **nodes** of code graphs (ICFG, CallGraph, VFG, etc.)
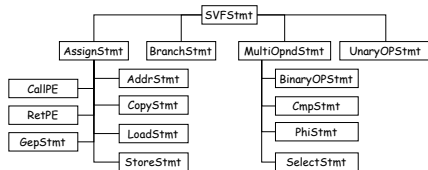


```
https://github.com/SVF-tools/SVF/
tree/master/svf/include/SVFIR
```

# 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` ⟸ `LLVMModuleSet::getSVFValue(llvm_value)`
  - `SVFValue` ⟸ `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` ⟸ `SVFIR::getGNode(SVFIR::getValueNode(svf_value));`
  - `ObjVar` ⟸ `SVFIR::getGNode(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 ⇐ LLVMModuleSet::getSVFValue(llvm_value)
  - SVFValue ⇐ 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 ⇐ SVFIR::getGNode(SVFIR::getValueNode(svf_value));
  - ObjVar ⇐ SVFIR::getGNode(SVFIR::getObjectNode(svf_value));
- **1:N** mapping from an `ICFGNode` to one or more `SVFStmts`.
  - SVFStmts ⇐ ICFGNode::getSVFStmts();
- **Always use the `toString` method** in SVF's data structures to understand their meanings and the mappings.
  - SVFVar::toString()
  - ICFGNode::toString()
  - ICFGEdge::toString()

# SVF Program Variables (SVFVar)

- An `SVFVar` represent either a top-level variable (`ValVar`, $\mathbb{P}$) or a memory object variable (`ObjVar`, $\mathbb{O}$)
- 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) |
| `BaseObjVar` | $o \in (\mathbb{S} \cup \mathbb{G} \cup \mathbb{H})$ | 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 | $\ell \in \mathbb{L}$ | 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 | $\%ptr = alloca_o$ | $p = alloc$ | $\mathbb{P} \times \mathbb{O}$ |
| ConsStmt | $\%ptr = constantData$ | $p = c$ | $\mathbb{P} \times \mathbb{C}$ |
| 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 | $\%p = getelementptr \%q, \%i$ | $p = \&(q \to i)$ or $p = \&q[i]$ | $\mathbb{P} \times \mathbb{P} \times \mathbb{P}$ |
| PhiStmt | $\%p = phi [\ell_1, \%q_1], [\ell_2, \%q_2]$ | $p = phi(\ell_1 : q_1, \ell_2 : q_2)$ | $\mathbb{P} \times (\mathbb{L} \times \mathbb{P})^2$ |
| 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 | $r = \otimes\ p, q$ | $r = p \otimes q$ | $\mathbb{P} \times \mathbb{P} \times \mathbb{P}$ |
| CallPE | $\%r = call\ f(\dots \%q_i \dots)$ $f(\dots \%p_i \dots)\{\ \dots\ ret\ \%z\}$ $\%p_i = \%q_i \quad (1 < i < n)$ | $r = f(\dots, q_i, \dots)$ $f(\dots, p_i, \dots)\{\dots\ return\ z\}$ $p_i = q_i \quad (1 < i < n)$ | $(\mathbb{P} \times \mathbb{P})^n$ |
| 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`.

# 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`.

# 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 **call p**arameter **e**dge passing from an actual parameter at a callsite to a formal parameter of a callee function.
- RetPE represents the **ret**urn **p**arameter **e**dge passing from a function return to a callsite return variable.

# Graph Representations of Code

- What are graph representations of code (code graphs)?
  - Put `SVFVars` and `SVFStmts` on one or more graph representations. For example:
    - **Call Graph**: each node represents a function and each edge represents a calling relation.
    - **ICFG** (Interprocedural Control-Flow Graph): where each node represents a statement and each edge represents a program's control-flow (i.e., execution order)
    - **PAG** (Program Assignment Graph): where each represents a variable (`SVFVar`) and each edge represents a program statement (`SVFStmt`).
    - **Constraint Graph**: A subgraph of PAG where each node is either a variable of pointer type or a memory object.

# Graph Representations of Code

- What are graph representations of code (code graphs)?
  - Put `SVFVars` and `SVFStmts` on one or more graph representations. For example:
    - **Call Graph**: each node represents a function and each edge represents a calling relation.
    - **ICFG** (Interprocedural Control-Flow Graph): where each node represents a statement and each edge represents a program's control-flow (i.e., execution order)
    - **PAG** (Program Assignment Graph): where each represents a variable (`SVFVar`) and each edge represents a program statement (`SVFStmt`).
    - **Constraint Graph**: A subgraph of PAG where each node is either a variable of pointer type or a memory object.
- Why graph representations?
  - Abstracting code from low-level complicated instructions
  - Applying general graph algorithms
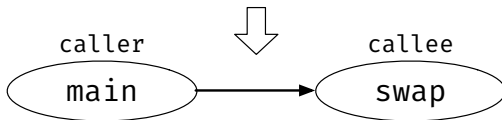  - Easy to maintain and extend

# 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
}
```

```
       caller                    callee
     ( main )                   ( swap )

              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.

```llvm
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
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
}
```

caller          callee

```
main  ------->  swap
```

Call Graph

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

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

# Control Flow Graph

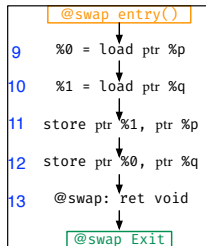Program execution order **between two LLVM instructions (SVFStmts)**.

- 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
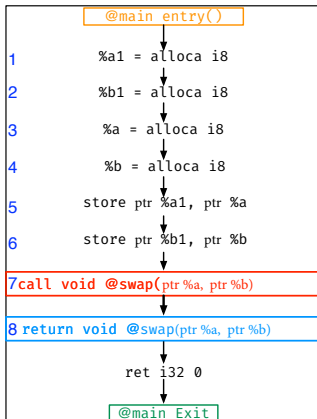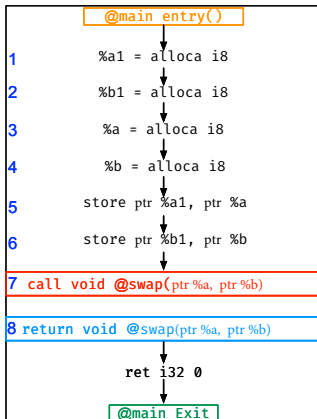


Program execution order between instructions

- Each node represents an instruction or a statement

- Each edge represents a control-flow dependence between two nodes

■ IntraICFGNode
■ FunEntryICFGNode
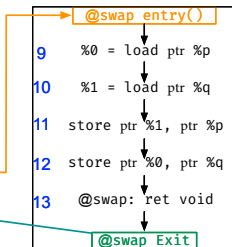■ FunExitICFGNode
■ RetICFGNode
■ CallICFGNode

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

■ IntraICFGNode
■ FunEntryICFGNode
■ FunExitICFGNode
■ RetICFGNode
■ CallICFGNode

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

# SVFIR Example (SVFVar + SVFStmt = PAG)

**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 }
```

# SVFIR Example (SVFVar + SVFStmt = PAG)

```
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 }
```

42

# SVFIR Example (SVFVar + SVFStmt = PAG)

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 |
| BaseObjVar ID: 5 | foo (object) |
| RetPN ID: 6 | ret of foo |
| ValVar ID: 15 | main |
| BaseObjVar ID: 16 | main (object) |
| RetPN ID: 17 | ret of main |

# SVFIR Example (SVFVar + SVFStmt = PAG)

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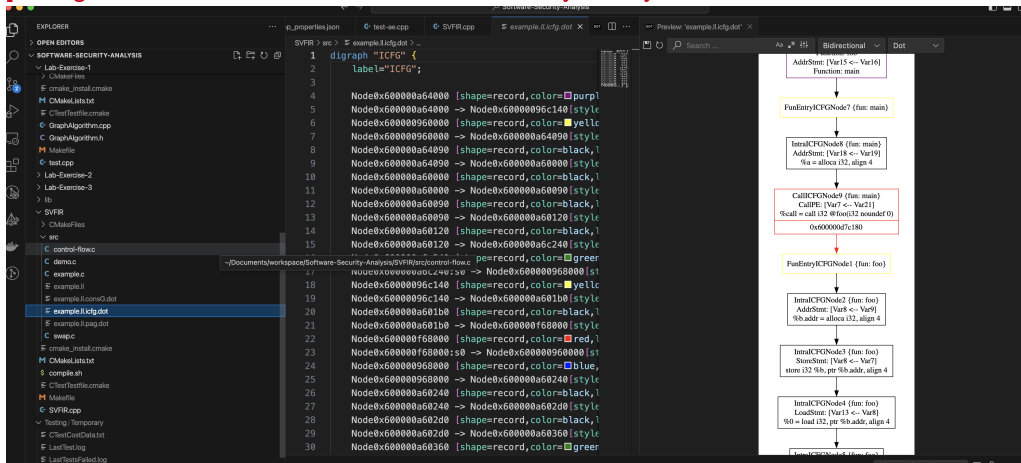
```
1  define i32 @foo(i32 %b) {
2  entry:
3    %b.addr = alloca i32
4    store i32 %b, ptr %b.addr,
5    %0 = load i32, ptr %b.addr,
6    ret i32 %0
7  }
8  define i32 @main() {
9    %a = alloca i32
10   %call = call i32 @foo(i32 0)
11   store i32 %call, i32* %a
12   ret i32 0
13 }
```

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 |
| BaseObjVar ID: 5 | foo (object) |
| RetPN ID: 6 | ret of foo |
| ValVar ID: 15 | main |
| BaseObjVar ID: 16 | main (object) |
| RetPN ID: 17 | 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 |
| BaseObjVar ID: 9 | %b.addr = alloca i32 |
| ValVar ID: 10 | i32 1 { constant data } |
| BaseObjVar 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 |
| BaseObjVar ID: 19 | %a = alloca i32 |
| ValVar ID: 20 | %call = call i32 @foo(i32 0) |
| ValVar ID: 21 | i32 0 { constant data } |
| BaseObjVar ID: 22 | i32 0 { constant data } |
| ValVar ID: 23 | store i32 %call, ptr %a |
| ValVar ID: 24 | ret i32 0 |

# SVFIR Example (ICFG and SVFStmt)

```
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`

# SVFIR and Code Graphs in DOT Fomat

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

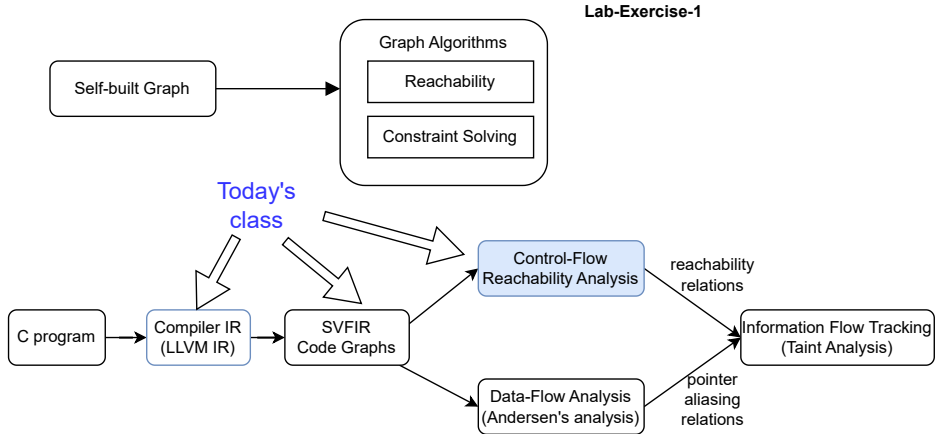# Control-Flow and Reachability Analysis

## (Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

# Today's Class

# What are Control-Flow and Data-Flow?

- **Control-flow 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 SVFVars' dependence

# Why Learning Control-Flow and Data-Flow?

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

# Why Learning Control-Flow and Data-Flow?

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.

# Why Learning Control-Flow and Data-Flow?

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.
  - . . .

# Why Learning Control-Flow and Data-Flow?

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.

# Why Learning Control-Flow and Data-Flow?

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 variables $v1$ and $v2$ are aliases (their points-to sets intersect, indicating they may reference the same memory location), then if $v1$ is tainted by user inputs, $v2$ may also be tainted.
  - ...

# Control-Flow Reachability

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-flow reachability
  - control-flow traversal without matching calls and returns.
  - fast but imprecise

# Control-Flow Reachability

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-flow reachability
  - control-flow traversal without matching calls and returns.
  - fast but imprecise
- Context-sensitive control-flow reachability
  - 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 Reachability

```c
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 Reachability - An Example

```llvm
   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
   if.then:          ; preds = %entry
 9 %call1 = call i32 @bar(i32 %call)
10 call void @sink(i32 %call1)
11 br label %if.end
12
   if.else:          ; preds = %entry
13 %call2 = call i32 @bar(i32 %call)
14 call void @sink(i32 %call2)
15 br label %if.end
16
   if.end:          ; preds = %if.else, %if.then
17 ret i32 0
18 }
```

# Control-Flow Reachability - An Example

```
   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
   if.then:          ; preds = %entry
 9 %call1 = call i32 @bar(i32 %call)
10 call void @sink(i32 %call1)
11 br label %if.end
12
   if.else:          ; preds = %entry
13 %call2 = call i32 @bar(i32 %call)
14 call void @sink(i32 %call2)
15 br label %if.end
16
   if.end:           ; preds = %if.else, %if.then
17 ret i32 0
18 }
```

```
 6 %call = call i32 (...) @source()
 7 %cmp = icmp sgt i32 %call, 0
 8 br i1 %cmp, label %if.then, label %if.else
```

**call1**
call1 i32 @bar(i32 %call)

**call2**
call2 i32 @bar(i32 %call)
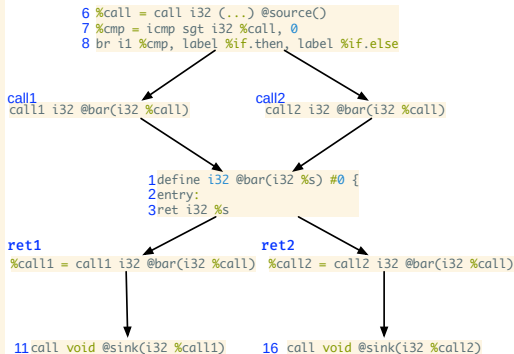
```
 1 define i32 @bar(i32 %s) #0 {
 2 entry:
 3 ret i32 %s
```

**ret1**
%call1 = call1 i32 @bar(i32 %call)

**ret2**
%call2 = call2 i32 @bar(i32 %call)

```
11 call void @sink(i32 %call1)
```

```
16 call void @sink(i32 %call2)
```

ICFG

# Control-Flow Reachability - An Example
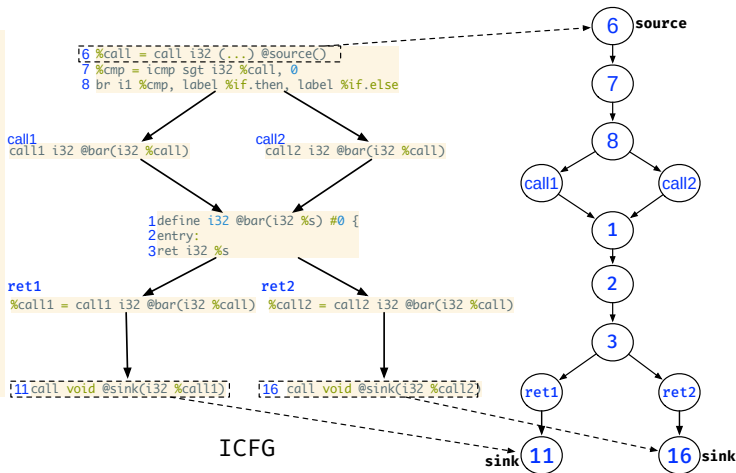


ICFG

# Context-Insensitive Control-Flow Reachability

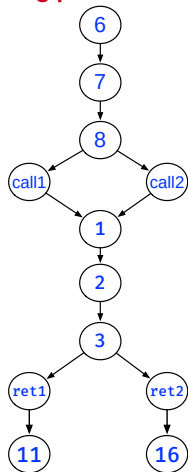## Obtaining a path from source to sink on ICFG
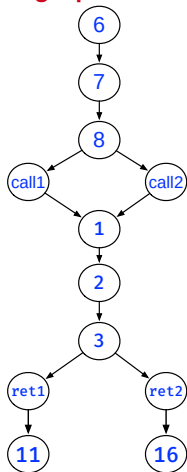
```
Basic DFS on ICFG: source → sink
```

```
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();
```

# Context-Insensitive Control-Flow Reachability

## Obtaining paths from node 6 to node 11 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 11

Path 1:
    6→7→8→call1→1→2→3→ret1→11
Path 2:
    6→7→8→call2→1→2→3→ret1→11
```

# Context-Insensitive Control-Flow Reachability

**Feasible paths from node 6 to node 11**
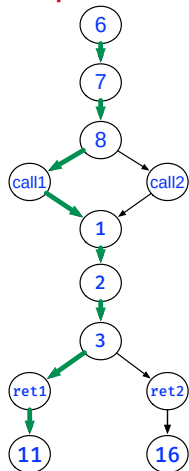


```
Basic DFS on ICFG: source → sink
```

```
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();
```

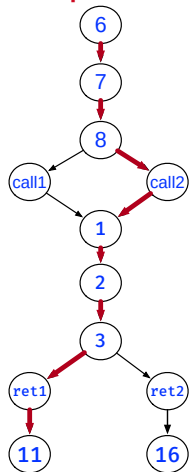ICFG paths: node 6 → node 11

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 Reachability

**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
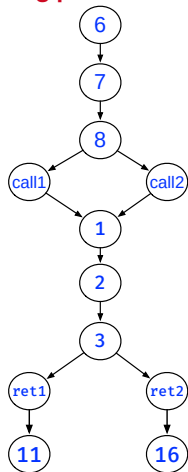
Path 1:
    6→7→8→**call1**→1→2→3→**ret1**→11
Path 2:
    6→7→8→**call2**→1→2→3→**ret1**→11
          **spurious path**

# Context-Insensitive Control-Flow Reachability

**Obtaining paths from node 6 to node 16 on 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
```
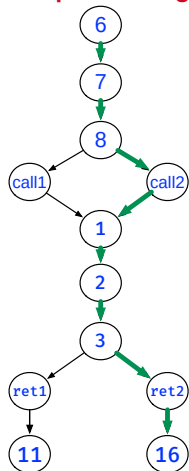
# Context-Insensitive Control-Flow Reachability

**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();
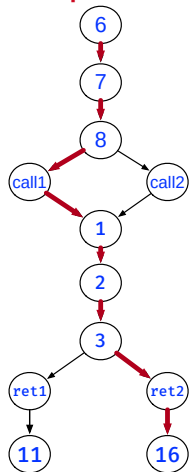```

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 Reachability

**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-Flow Reachability

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 Reachability (Algorithm)

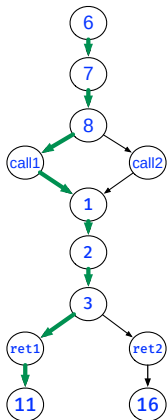**Algorithm 1: 1** Context sensitive control-flow reachability

**Input :** curNode : ICFGNode   snk : ICFGNode   path : vector⟨ICFGNode⟩   callstack : vector⟨SVFInstruction⟩
        visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, snk);
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.isIntraCFGEdge() then
11   |  |  dfs(edge.dst, snk);
12   |  else if edge.isCallCFGEdge() then
13   |  |  callstack.push_back(edge.getCallSite());
14   |  |  dfs(edge.dst, snk);
15   |  |  callstack.pop_back();
16   |  else if edge.isRetCFGEdge() 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();
```

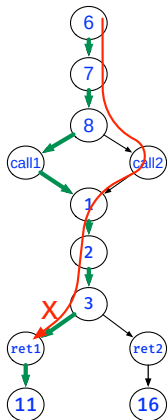# Context-Sensitive Control-Flow Reachability (Example)

**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,snk)
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.isIntraCFGEdge() then
11           dfs(edge.dst,snk);
12       else if edge.isCallCFGEdge() then
13           callstack.push_back(edge.getCallSite());
14           dfs(edge.dst,snk);
15           callstack.pop_back();
16       else if edge.isRetCFGEdge() 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-Flow Reachability (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, snk)
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.isIntraCFGEdge() then
11           dfs(edge.dst, snk);
12       else if edge.isCallCFGEdge() then
13           callstack.push_back(edge.getCallSite());
14           dfs(edge.dst, snk);
15           callstack.pop_back();
16       else if edge.isRetCFGEdge() 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?

- Debug and work with the code under the `SVFIR` and `CodeGraph` folders, including the understanding of `ICFG`, `PAG` and `ConstraintGraph`.
- Understand control-flow reachability in the today's slides
- If you finished `Quiz-1` and `Lab-Exercise-1`, you could have a look at the spec of `Assignment-1` and start implementing the `readSrcSnkFromFile` and `reachability` methods or all other methods (if you want) in `Assignment-1`
  - `Assignment-1`'s specification: `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-1`