LLVM Compiler and Its Intermediate Representation

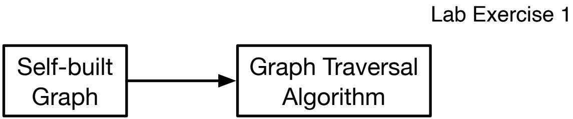
(Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Where We Are Now and Today’s Class

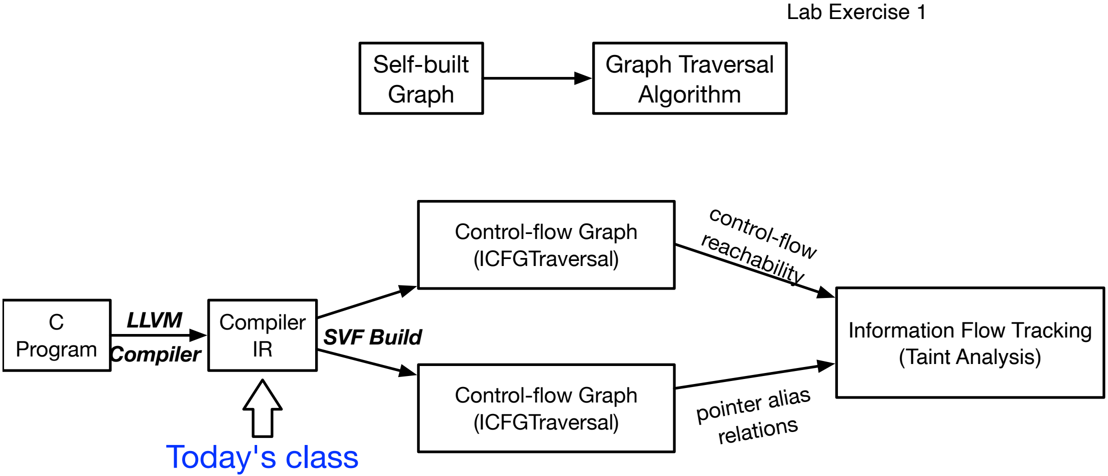


Lab Exercise 1

Self-built Graph

Graph Traversal Algorithm

Where We Are Now and Today’s Class


Lab Exercise 1
Self-built Graph
Graph Traversal Algorithm
C Program
LLVM Compiler
Compiler IR
SVF Build
Control-flow Graph (ICFGTraversal)
Control-flow Graph (ICFGTraversal)
control-flow reachability
pointer alias relations
Information Flow Tracking (Taint Analysis)
Today's class

What is LLVM ?

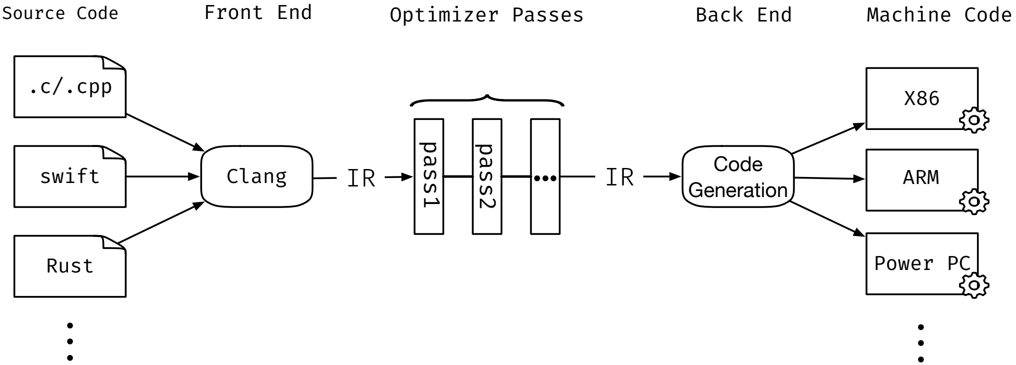
LLVM compiler infrastructure is a collection of compiler and tool-chain technologies.

- 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?

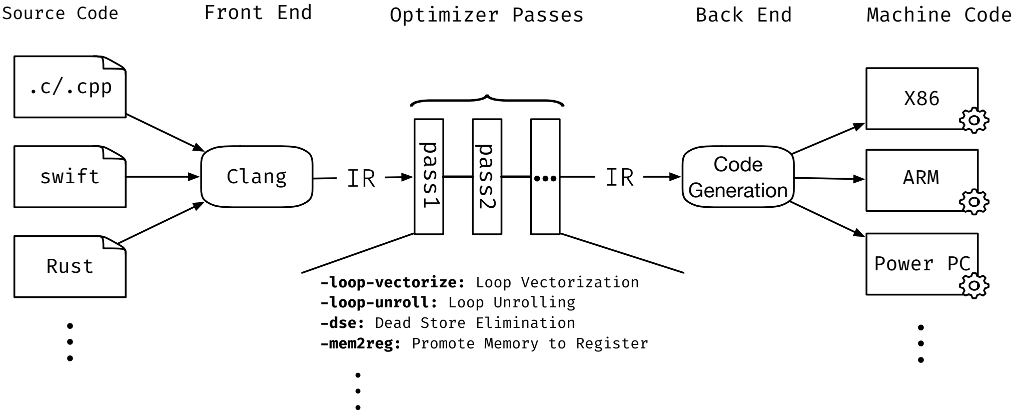
- An essential part of the standard curriculum in computer science.
- One of the most complex systems required for building virtually all other software.
- A perfect base framework to build your own tools for code analysis and verification
- Sharpen your software design and implementation skills.
- Widely 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



-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 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
- **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`)

- **Language independent.** Not machine code, but one step just above assembly
- **Clear lexical scope**, such as modules, functions, basic blocks, and instructions
- **3-address code style** in **static single assignment (SSA)** form
  - Variables are strongly typed
  - Global variable (symbol starting with '@')
  - Stack/register variable (symbol starting with '%')
  - Three addresses and one operator.
    - 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'.

Compiling a C Program to Its LLVM IR

Clang/LLVM compiler options

- 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-O0-optnone -emit-llvm swap.c -o swap.ll`
- Keep the variable names.
  - `clang -c -S -fno-discard-value-names -Xclang -disable-O0-optnone -emit-llvm swap.c -o swap.ll`
- Convert the LLVM IR to more compact SSA form for later static analysis.
  - `opt -S -p=mem2reg swap.ll -o swap.ll`

Compiling a C Program to Its LLVM IR

An example

```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
}
```

swap.ll

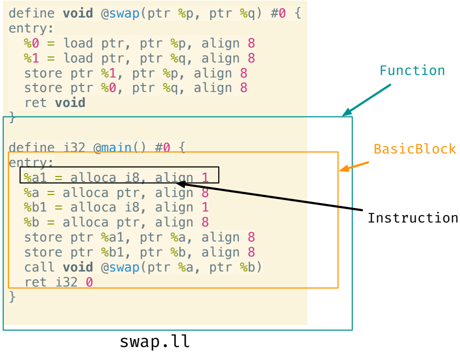
C code to LLVM IR

An example

```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 ⇨

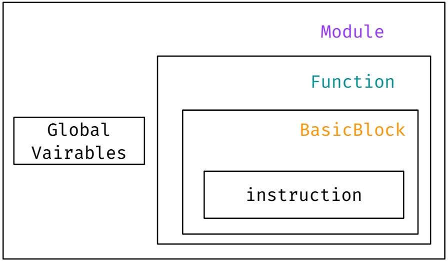


```llvm
define void @swap(ptr %p, ptr %q) #0 {
entry:
  %0 = load ptr, ptr %p, align 8
  %1 = load ptr, ptr %q, align 8
  store ptr %1, ptr %p, align 8
  store ptr %0, ptr %q, align 8
  ret void
}

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
}
```

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.

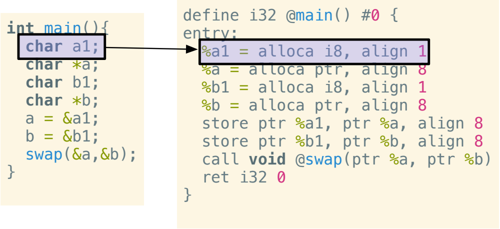
Function contains BasicBlocks and Arguments, which correspond to functions.

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

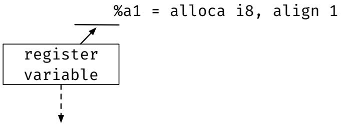
LLVM Intermediate Representation (IR)

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
}
```



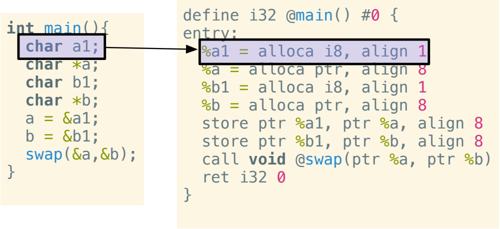
```
%a1 = alloca i8, align 1
```

register variable

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

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

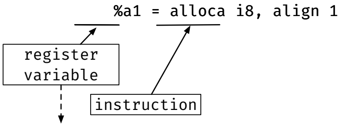
LLVM Intermediate Representation (IR)

LLVM Instructions



```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:

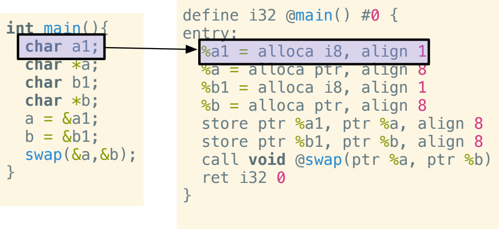
[% / @] [-a-zA-Z$._][-a-zA-Z$._0-9]

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

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

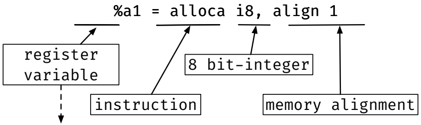
LLVM Intermediate Representation (IR)

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
}
```



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

identifiers:

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

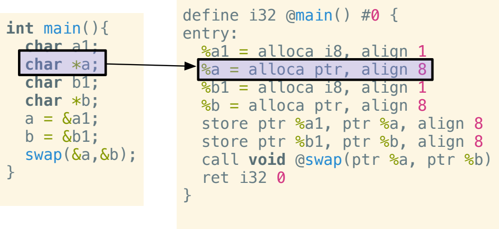
- % is for local variable
- @ is for global
- temporary variables are numbered

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`

↓

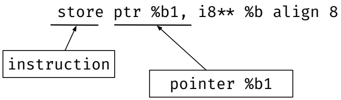
`allocate 8-bit integer pointer for a`

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 ptr %b1, i8** %b align 8

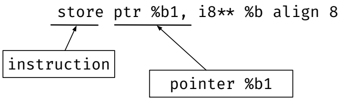
store the pointer %b1 to the memory location that %b points to

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 ptr %b1, i8** %b align 8
```

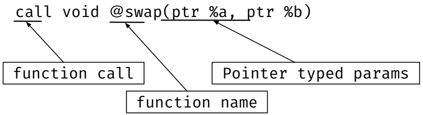
store the pointer %b1 to the memory location that %b points to

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 void @swap(ptr %a, ptr %b)

function call

function name

Pointer typed params

call instruction will be used to
build control flow.

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

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 (the latest version 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.

SVF : Static Value-Flow Analysis Framework for Source Code

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

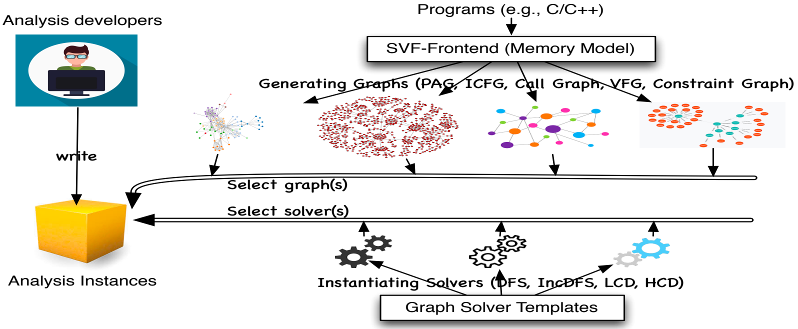
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point A flow to another program point B along some execution paths?
  - Can function F be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?

SVF : Static Value-Flow Analysis Framework for Source Code

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

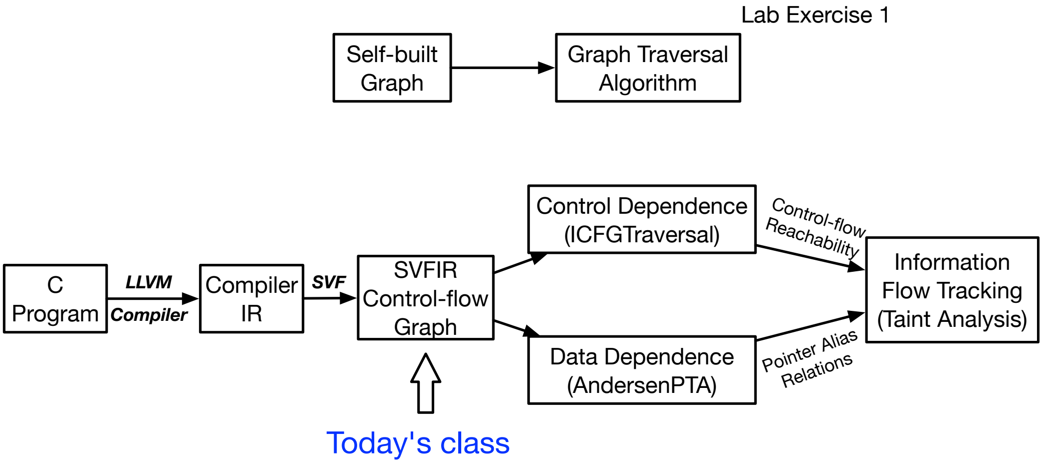
- The SVF project
  - **Publicly available** since early 2015 and actively maintained: `http://svf-tools.github.io/SVF`.
  - Implemented on top of LLVM compiler (the latest version 12.0.0) with over 100 KLOC C/C++ code and **700+ stars with 40+ contributors** and over 1K commits on Github.
  - Invited for a **plenary talk in EuroLLVM 2016**, and awarded an **ICSE 2018 Distinguished Paper**, an **SAS Best Paper 2019** and an **OOPSLA 2020 Distinguished Paper**.
- Value-Flow Analysis: resolves **both control and data dependence**.
  - Does the information generated at program point A flow to another program point B along some execution paths?
  - Can function F be called either directly or indirectly from some other function F'?
  - Is there an unsafe memory access that may trigger a bug or security risk?
- Key features of SVF
  - **Sparse**: compute and maintain the data-flow facts where necessary
  - **Selective** : support mixed analyses for precision and efficiency trade-offs.
  - **On-demand** : reason about program parts based on user queries.

SVF: Design Principle


Programs (e.g., C/C++)
Analysis developers
SVF-Frontend (Memory Model)
Generating Graphs (PAG, ICFG, Call Graph, VFG, Constraint Graph)
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)

Today’s class

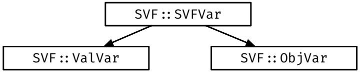

Lab Exercise 1
Self-built Graph
Graph Traversal Algorithm
C Program
LLVM
Compiler
Compiler IR
SVF
SVFIR Control-flow Graph
Control Dependence (ICFGTraversal)
Data Dependence (AndersenPTA)
Control-flow Reachability
Pointer Alias Relations
Information Flow Tracking (Taint Analysis)
Today's class

SVF IR and Why?

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

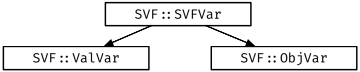
SVF IR and Why?

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

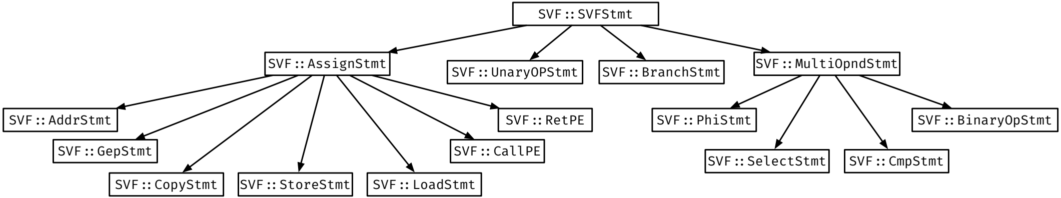


SVF IR and Why?

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



- SVFStmt: program statements



SVF Program Variables (SVFVar)

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

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

SVF Program Statements (SVFStmt)

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

SVFStmt	LLVM-Like form	C-Like form	Operand types
AddrStmt	$\%ptr = alloca_o$	$p = alloc$	$\mathbb{P} \times \mathbb{O}$
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	$\%p = phi\ [\ l_1, \%q_1\ ], [\ l_2, \%q_2\ ]$	$p = phi(l_1 : q_1,\ l_2 : q_2)$	$\mathbb{P} \times (\mathbb{L} \to \mathbb{P}^2)$
BranchStmt	$br\ i1\ \%p, label\ \%l_1, label\ \%l_2$	$if\ (p)\ l_1$ else l_2	$\mathbb{P} \times \mathbb{L}^2$
UnaryOPStmt	$p = \neg q$	$p = \neg q$	$\mathbb{P} \times \mathbb{P}$
BinaryOPStmt/CmpStmt	$r = \otimes\ p, q$	$r = p \otimes q$	$\mathbb{P} \times \mathbb{P} \times \mathbb{P}$
	$\%r = call\ f(\ldots\%q_i\ldots)$ $f(\ldots\%p_i\ldots)\{\ \ldots\ ret\ \%z\}$	$r = f(\ldots, q_i, \ldots)$ $f(\ldots, p_i, \ldots)\{\ldots\ return\ z\}$	
CallPE	$\%p_i = \%q_i \quad (1 < i < n)$	$p_i = q_i \quad (1 < i < n)$	$(\mathbb{P} \times \mathbb{P})^n$
RetPE	$\%r = \%z$	$r = z$	$\mathbb{P} \times \mathbb{P}$

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

SVF Program Statements (SVFStmt)

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

Graph Representation of Code

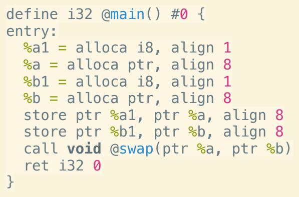
- What is a graph representation of code (code graph)?
  - Put the LLVM IR or SVF IR on a graph representation.
  - Represent a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.

Graph Representation of Code

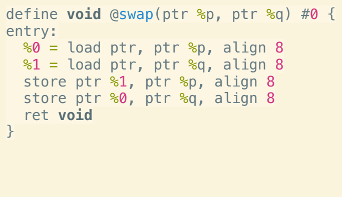
- What is a graph representation of code (code graph)?
    - Put the LLVM IR or SVF IR on a graph representation.
    - Represent a program's control-flow (i.e., execution order) and/or data-flow (variable definition and use relations) using nodes and edges of a graph.
- Why a graph representation?
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

Call Graph

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



```
define i32 @main() #0 {
entry:
  %a1 = alloca i8, align 1
  %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
}
```

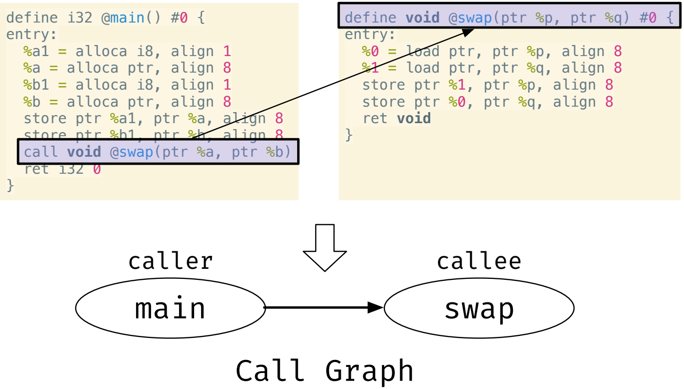
caller
main
callee
swap

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.



```
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

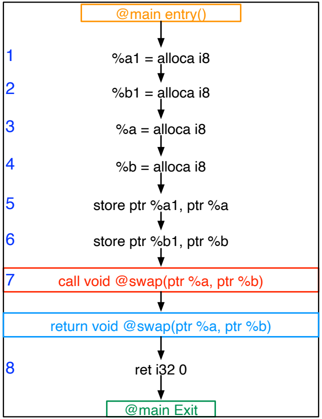
1. Each node represents a program method
2. Each edge represents a calling relation between two program methods

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



@main entry()

1 %a1 = alloca i8

2 %b1 = alloca i8

3 %a = alloca i8

4 %b = alloca i8

5 store ptr %a1, ptr %a

6 store ptr %b1, ptr %b

7 call void @swap(ptr %a, ptr %b)

return void @swap(ptr %a, ptr %b)

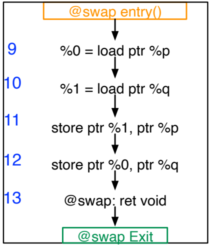
8 ret i32 0

@main Exit

Program execution order between instructions

- Each node represents an instruction or a statement

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



@swap entry()

9 %0 = load ptr %p

10 %1 = load ptr %q

11 store ptr %1, ptr %p

12 store ptr %0, ptr %q

13 @swap: ret void

@swap Exit

■ IntraBlockNode

■ FunEntryBlockNode

■ FunExitBlockNode

■ RetBlockNode

■ CallBlockNode

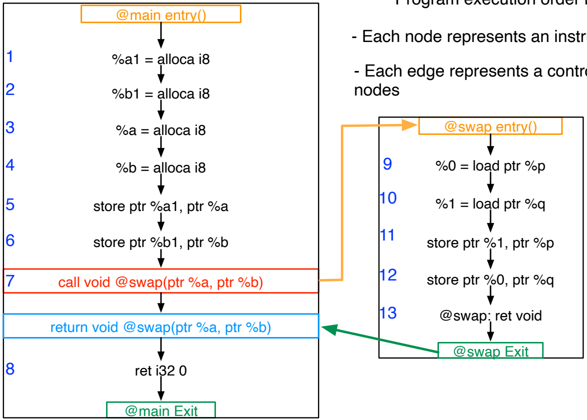
https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph

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



■ IntraBlockNode
🟧 FunEntryBlockNode
🟩 FunExitBlockNode
🟦 RetBlockNode
🟥 CallBlockNode

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

SVF IR Example

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

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

SVF IR Example

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

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

```llvm
define i32 @foo(i32 %b) {
entry:
  %b.addr = alloca i32
  store i32 %b, ptr %b.addr,
  %0 = load i32, ptr %b.addr,
  ret i32 %0
}
define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

SVF IR Example

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

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

```
define i32 @foo(i32 %b) {
entry:
  %b.addr = alloca i32
  store i32 %b, ptr %b.addr,
  %0 = load i32, ptr %b.addr,
  ret i32 %0
}
define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

Variables introduced by SVF
(created internally)

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

SVF IR Example

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

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

```llvm
define i32 @foo(i32 %b) {
entry:
  %b.addr = alloca i32
  store i32 %b, ptr %b.addr,
  %0 = load i32, ptr %b.addr,
  ret i32 %0
}
define i32 @main() {
  %a = alloca i32
  %call = call i32 @foo(i32 0)
  store i32 %call, i32* %a
  ret i32 0
}
```

Variables introduced by SVF
(created internally)

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

Variables introduced by LLVM
(created by LLVM Values)

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

ICFG and SVFStmt Example[2]

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

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

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

What's next?

- (1) Compile two C programs (`SVFIR/src/example.c` and `SVFIR/src/swap.c`) into their LLVM IR.
  - A guide can be found at `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR`
  - Understand the mapping from a C program to its corresponding LLVM IR.
- (2) Generate and visualize the graph representation of LLVM IR (`example.ll` `swap.ll`).
  - `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR#3-run-and-debug-your-svfir`
- (3) Write code to iterate SVFVars and also the nodes and edges of ICFG and print their contents.
  - `https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/SVFIR.cpp#L74-L98`
- (4) More about LLVM IR and SVF's graph representation
  - LLVM language manual `https://llvm.org/docs/LangRef.html`
  - SVF website `https://github.com/SVF-tools/SVF`

Control-Flow and Reachability Analysis

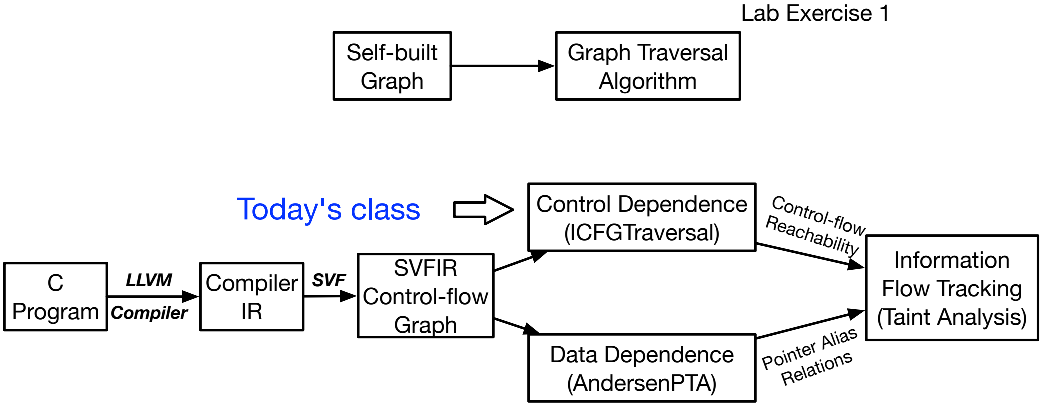
(Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today’s Class


Lab Exercise 1
Self-built Graph
Graph Traversal Algorithm
Today's class
Control Dependence (ICFGTraversal)
Control-flow Reachability
Information Flow Tracking (Taint Analysis)
C Program
LLVM
Compiler
Compiler IR
SVF
SVFIR Control-flow Graph
Data Dependence (AndersenPTA)
Pointer Alias Relations

Control- and Data-Dependence

What are control- and data-dependence?

- **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-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.

Control- and Data-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.

Control- and Data-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.

Control- and Data-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.
  - . . .

Control- and Data-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.

Control- and Data-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 v1 and v2 are aliases (e.g., representing the same memory location), if v1 is tainted by user inputs, then v2 is also tainted.
  - . . .

Control-Dependence

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-Dependence

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.

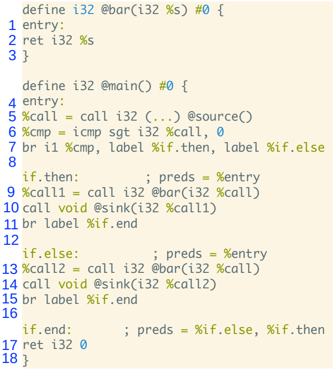
Control-Dependence

```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);
    }
}
```

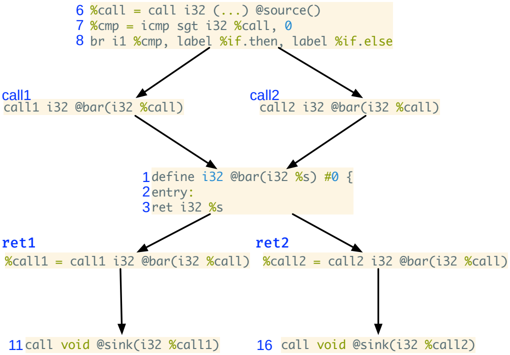
Control-Dependence

```
define i32 @bar(i32 %s) #0 {
entry:
ret i32 %s
}

define i32 @main() #0 {
entry:
%call = call i32 (...) @source()
%cmp = icmp sgt i32 %call, 0
br i1 %cmp, label %if.then, label %if.else

if.then:          ; preds = %entry
%call1 = call i32 @bar(i32 %call)
call void @sink(i32 %call1)
br label %if.end

if.else:           ; preds = %entry
%call2 = call i32 @bar(i32 %call)
call void @sink(i32 %call2)
br label %if.end

if.end:       ; preds = %if.else, %if.then
ret i32 0
}
```

Control-Dependence



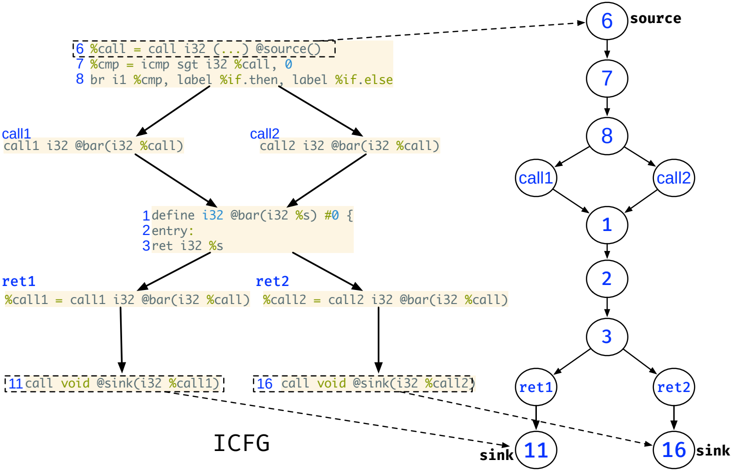
```
   define i32 @bar(i32 %s) #0 {
1  entry:
2  ret i32 %s
3  }

   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

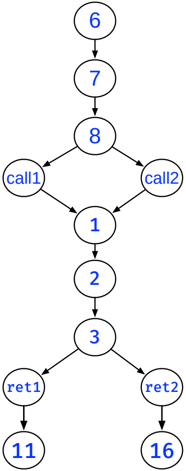
Control-Dependence

```
   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
6
source
7
8
call1
call2
1
2
3
ret1
ret2
sink
11
16
sink

Context-Insensitive Control-Dependence

Obtaining a path from source to sink on ICFG

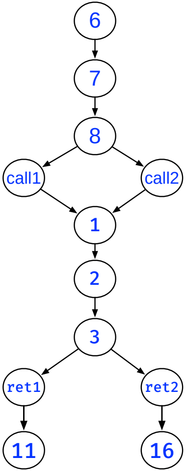


Basic DFS on ICFG: source → sink

```
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-Dependence

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();
```

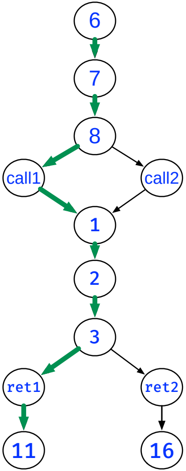
ICFG paths: node 6 → node 11

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-Dependence

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**

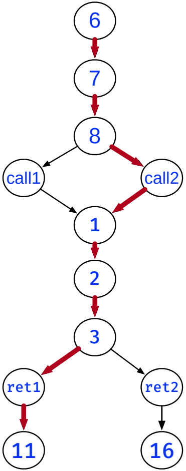
6→7→8→**call1**→1→2→3→**ret1**→11

Path 2:

6→7→8→**call2**→1→2→3→**ret1**→11

Context-Insensitive Control-Dependence

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

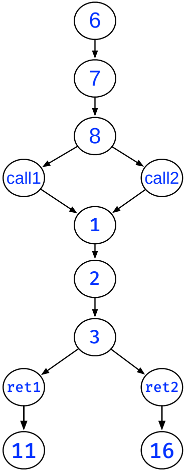
Path 1:
6→7→8→**call1**→1→2→3→**ret1**→11

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

spurious path

Context-Insensitive Control-Dependence

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();
```

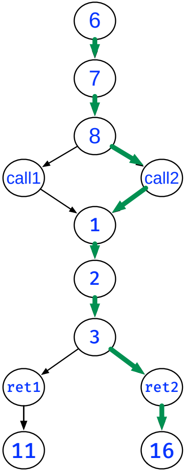
ICFG paths: node 6 → node 16

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-Dependence

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**

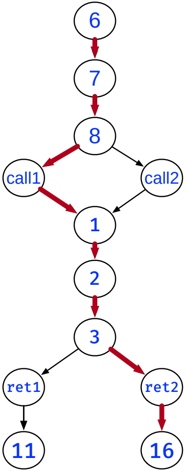
6→7→8→**call2**→1→2→3→**ret2**→16

Path 4:

6→7→8→**call1**→1→2→3→**ret2**→16

Context-Insensitive Control-Dependence

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

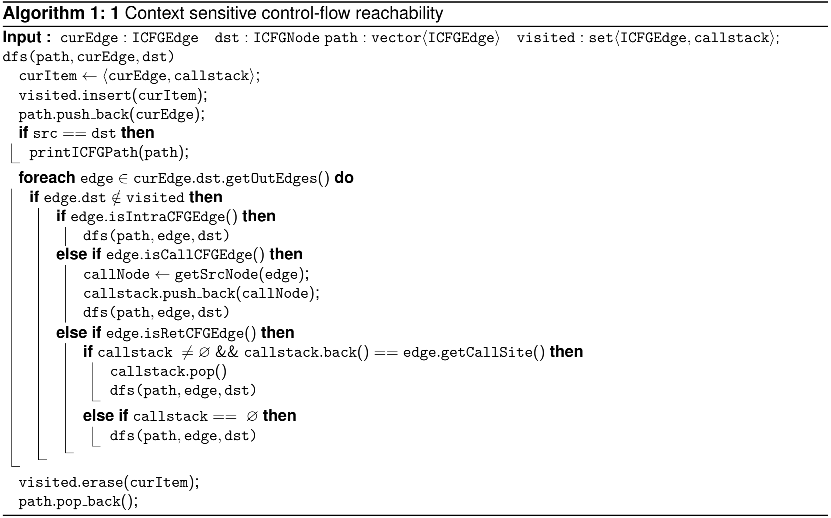
spurious path

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-Dependence (Algorithm)

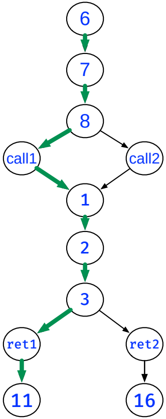


Algorithm 1: 1 Context sensitive control-flow reachability

```
Input : curEdge : ICFGEdge   dst : ICFGNode path : vector$\langle$ICFGEdge$\rangle$   visited : set$\langle$ICFGEdge, callstack$\rangle$;
dfs(path, curEdge, dst)
  curItem $\leftarrow$ $\langle$curEdge, callstack$\rangle$;
  visited.insert(curItem);
  path.push_back(curEdge);
  if src == dst then
    printICFGPath(path);
  foreach edge $\in$ curEdge.dst.getOutEdges() do
    if edge.dst $\notin$ visited then
      if edge.isIntraCFGEdge() then
        dfs(path, edge, dst)
      else if edge.isCallCFGEdge() then
        callNode $\leftarrow$ getSrcNode(edge);
        callstack.push_back(callNode);
        dfs(path, edge, dst)
      else if edge.isRetCFGEdge() then
        if callstack $\neq \varnothing$ && callstack.back() == edge.getCallSite() then
          callstack.pop()
          dfs(path, edge, dst)
        else if callstack == $\varnothing$ then
          dfs(path, edge, dst)
  visited.erase(curItem);
  path.pop_back();
```

Context-Sensitive Control-Dependence (Example)

call1 matches with ret1

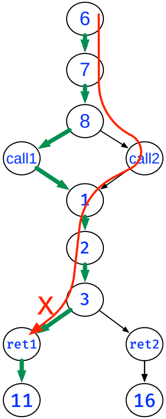


Algorithm 2: 1 Context sensitive control-flow reachability

```
Input : curEdge : ICFGEdge   dst : ICFGNode path : vector⟨ICFGEdge⟩   visited : set⟨ICFGEdge, callstack⟩;
dfs(curEdge, dst)
  curItem ← ⟨curEdge, callstack⟩;
  visited.insert(curItem);
  path.push_back(curEdge);
  if src == dst then
    printICFGPath(path);
  foreach edge ∈ curEdge.dst.getOutEdges() do
    if edge.dst ∉ visited then
      if edge.isIntraCFGEdge() then
        dfs(path, edge, dst)
      else if edge.isCallCFGEdge() then
        callNode ← getSrcNode(edge);
        callstack.push_back(callNode);
        dfs(path, edge, dst)
      else if edge.isRetCFGEdge() then
        if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
          callstack.pop()
          dfs(path, edge, dst)
        else if callstack == ∅ then
          dfs(path, edge, dst)
  visited.erase(curItem);
  path.pop_back();
```

Context-Sensitive Control-Dependence (Example)

call2 does not match with ret1



Algorithm 3: 1 Context sensitive control-flow reachability

```
Input : curEdge : ICFGEdge   dst : ICFGNode path : vector⟨ICFGEdge⟩   visited : set⟨ICFGEdge, callstack⟩;
dfs(curEdge, dst)
  curItem ← ⟨curEdge, callstack⟩;
  visited.insert(curItem);
  path.push_back(curEdge);
  if src == dst then
    printICFGPath(path);
  foreach edge ∈ curEdge.dst.getOutEdges() do
    if edge.dst ∉ visited then
      if edge.isIntraCFGEdge() then
        dfs(path, edge, dst)
      else if edge.isCallCFGEdge() then
        callNode ← getSrcNode(edge);
        callstack.push_back(callNode);
        dfs(path, edge, dst)
      else if edge.isRetCFGEdge() then
        if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
          callstack.pop()
          dfs(path, edge, dst)
        else if callstack == ∅ then
          dfs(path, edge, dst)
  visited.erase(curItem);
  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`