

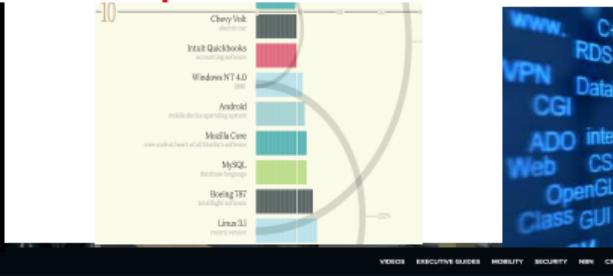
# Introduction to Software Analysis

Yulei Sui

University of Technology Sydney, Australia

# Modern System Software

Extremely large and complex but error-prone



More  
Complex!

Microsoft: 70 percent of all security bugs are  
memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.



Memory Leaks



Buffer Overflows



Null Pointers



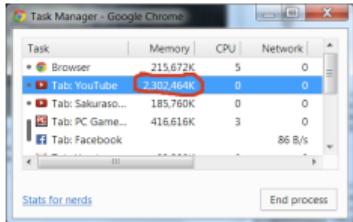
Use-After-Frees

Data-races

More  
Buggy!

# Modern System Software

Extremely large and complex but error-prone



**memory leaks**

massive leaks over 2GB  
on a single browser tab



**buffer overflow**

66% websites affected



**use-after-free**

exploit price up to \$100k  
per bug in Chrome



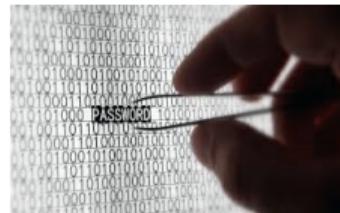
**null pointer**

denial of service affecting  
millions of servers worldwide



**data race**

11 civilians died

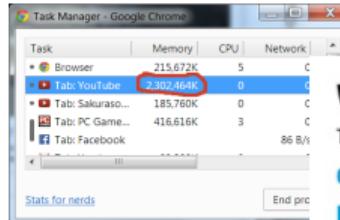


**uninitialized variables**

password leakage via tar on  
Solaris OS

# Modern System Software

Extremely large and complex but error-prone



memory leaks

massive leaks over 2G  
on a single browser tab



buffer overflow

66% websites affected



## Vulnerabilities (security defects)

The risks

Quality issue: many more “underwater” than those reported “above the water”

### The National Vulnerability Database (DHS/US-CERT)

- Lists >47,000 documented vulnerabilities

### Undiscovered/unreported (0-day) vulnerabilities are huge

- 20X<sup>1</sup> multiplier
- 47,000 x 20 = estimated 940,000 vulnerabilities replicated in many products

Greater than 80% of attacks  
happen at the application layer



Public vulnerabilities are tip of the iceberg !

Design apps to  
run in cloud



data race

11 civilians died

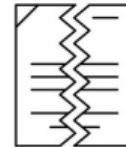


uninitialized variables

password leakage via tar on  
Solaris OS

# Code Review by Developers

However ...



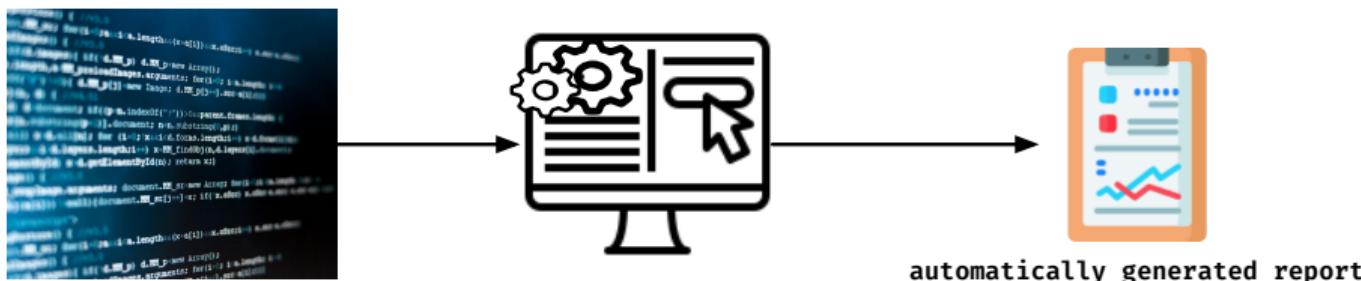
incomplete debug report

A large project (e.g., consists of millions of lines of code) is almost impossible to be manually checked by human :

- intractable due to potentially unbounded number of paths that must be analyze
- undecidable in the presence of dynamically allocated memory and recursive data structures

# What is Software/Program Analysis

- Software Analysis a.k.a Program analysis is the process of automatically analyzing the **behavior of computer programs** such as correctness, robustness, safety and security.
- Program analysis is to develop algorithms and tools which can **analyze other programs**



# What is Software/Program Analysis

- Software Analysis a.k.a Program analysis is the process of automatically analyzing the **behavior of computer programs** such as correctness, robustness, safety and security.
- Program analysis is to develop algorithms and tools which can **analyze other programs**
- Applications of program analysis
  - **Compiler optimizations:** transforming the source code to minimize a program's execution time, memory footprint, storage size, and power consumption
  - **Bug finding:** Identify the program or system that cause failure or produce an unexpected result
  - **Security vulnerability assessment:** Protect private users' data in databases
  - **Automatic Parallel Computation:** Guarantee the safe execution in different iterations on parallel calculations

# Static Analysis vs. Dynamic Analysis

## Static Analysis

- *Analyze a program without actually executing it – inspection of its source code by examining all possible program paths*
  - + Pin-point bugs at source code level.
  - + Catch bugs earlier during software development.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

# Static Analysis vs. Dynamic Analysis

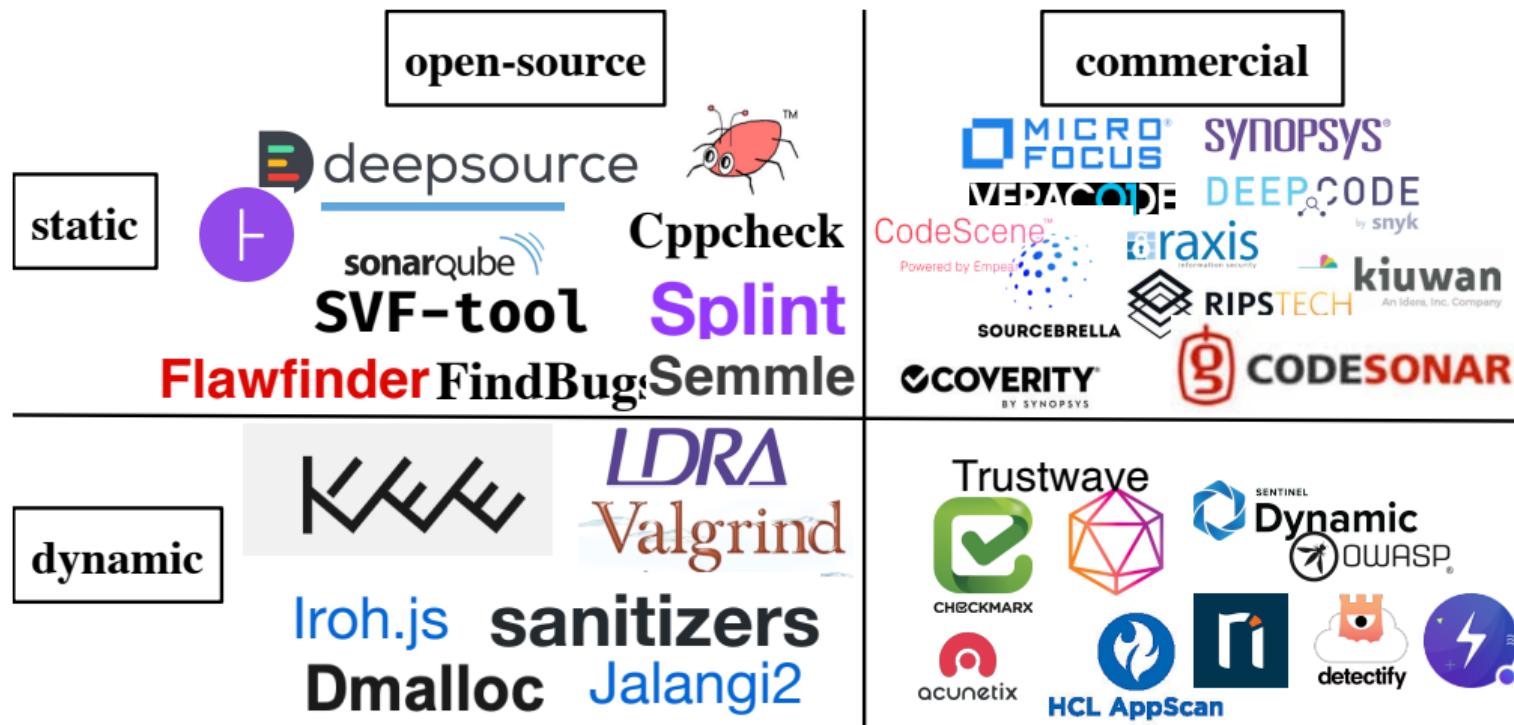
## Static Analysis

- *Analyze a program without actually executing it – inspection of its source code by examining all possible program paths*
  - + Pin-point bugs at source code level.
  - + Catch bugs earlier during software development.
  - - False alarms due to over-approximation.
  - - Precise analysis has scalability issue for analyzing large size programs.

## Dynamic Analysis

- *Analyze a program at runtime – inspection of its running program by examining some executable paths depending on specific test inputs*
  - + Identify bugs at runtime (catch it when you observe it).
  - + Zero or very low false alarm rates.
  - - Runtime overhead due to code instrumentation.
  - - May miss bugs (false negative) due to under-approximation.

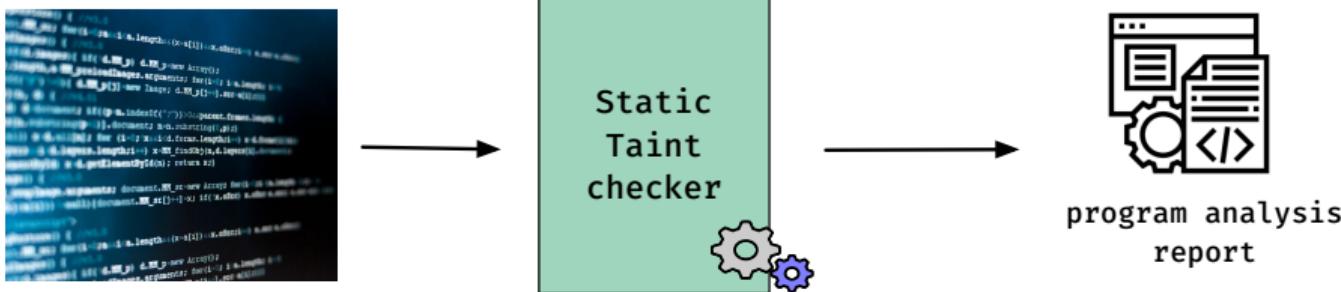
# Existing Program Analysis Tools



# The Project of This Subject

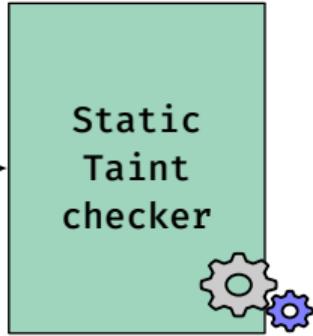
**Goal of this subject:** develop your own software analysis tool in 12 weeks.

**More concretely:** develop an automated static analysis checker using C++ to analyze tainted information flow of a C program.



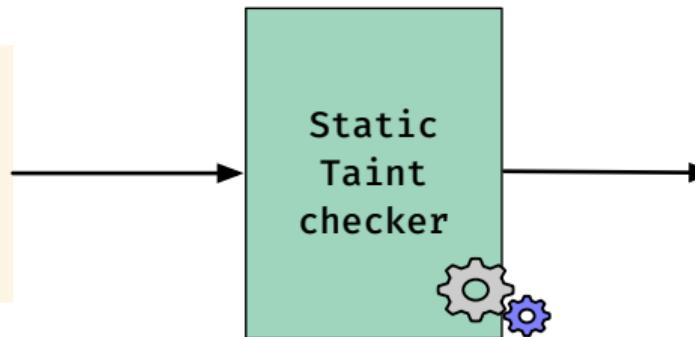
# The Project of This Subject

```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char cmd[100] = "test";
8     cmd += home;
9     execl(cmd, "ls", "-l", NULL);
10 }
```



# The Project of This Subject

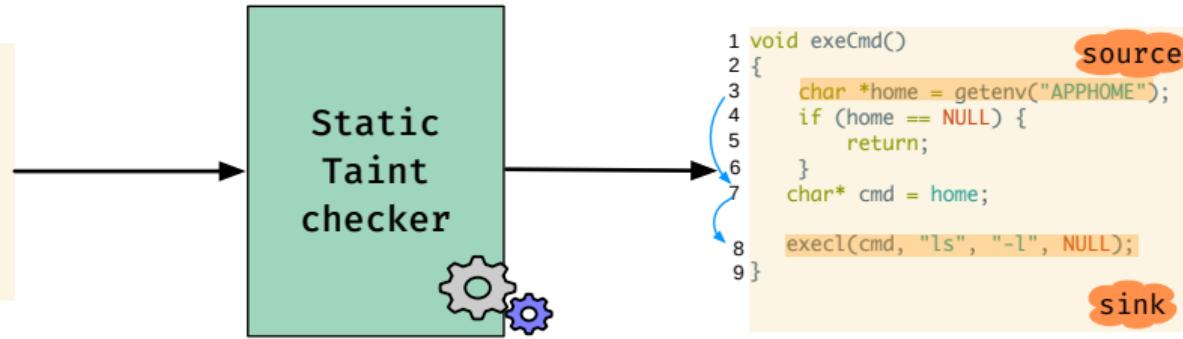
```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
```



```
1 void exeCmd() source
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9 } sink
```

# The Project of This Subject

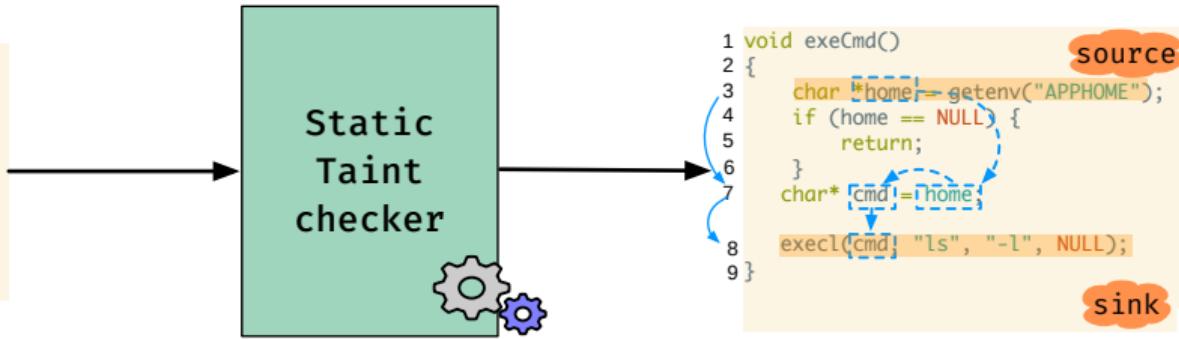
```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
```



→ Control dependence

# The Project of This Subject

```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
```

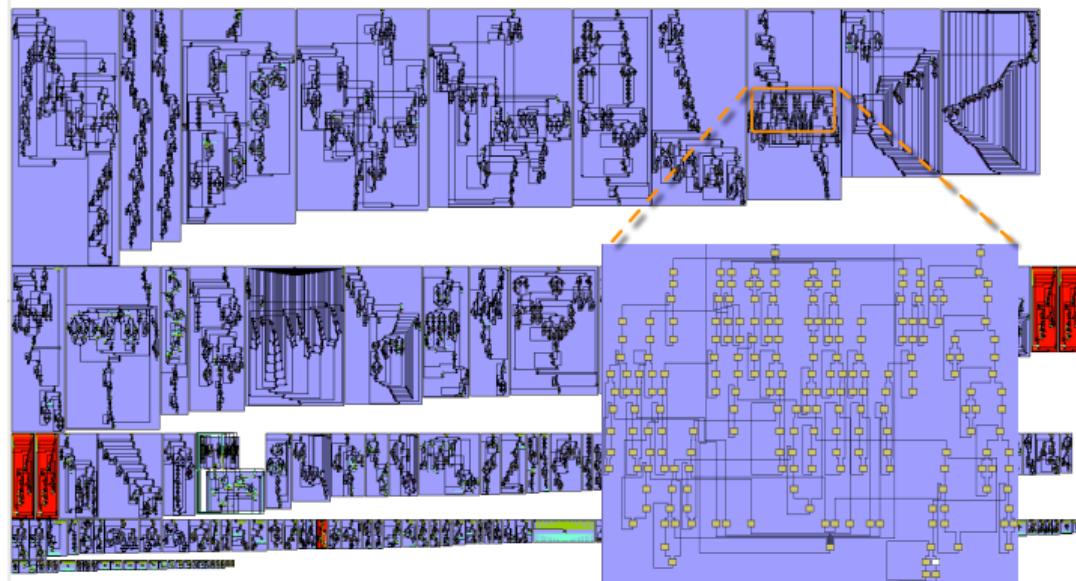


→ Control dependence

→ Data dependence

# How about real-world large programs?

Whole-Program CFG of 300.twolf (20.5K lines of code)



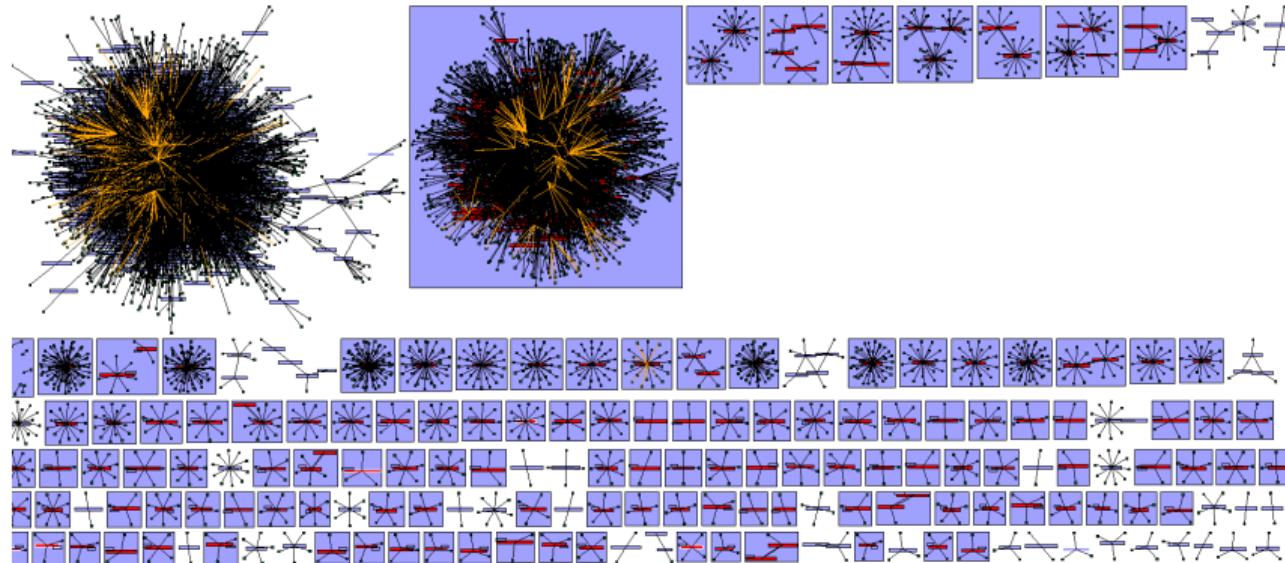
#functions: 194

#pointers: 20773  
on CFGs!

#loads/stores: 8657 Costly to reason about flow of values

# How about real-world large programs?

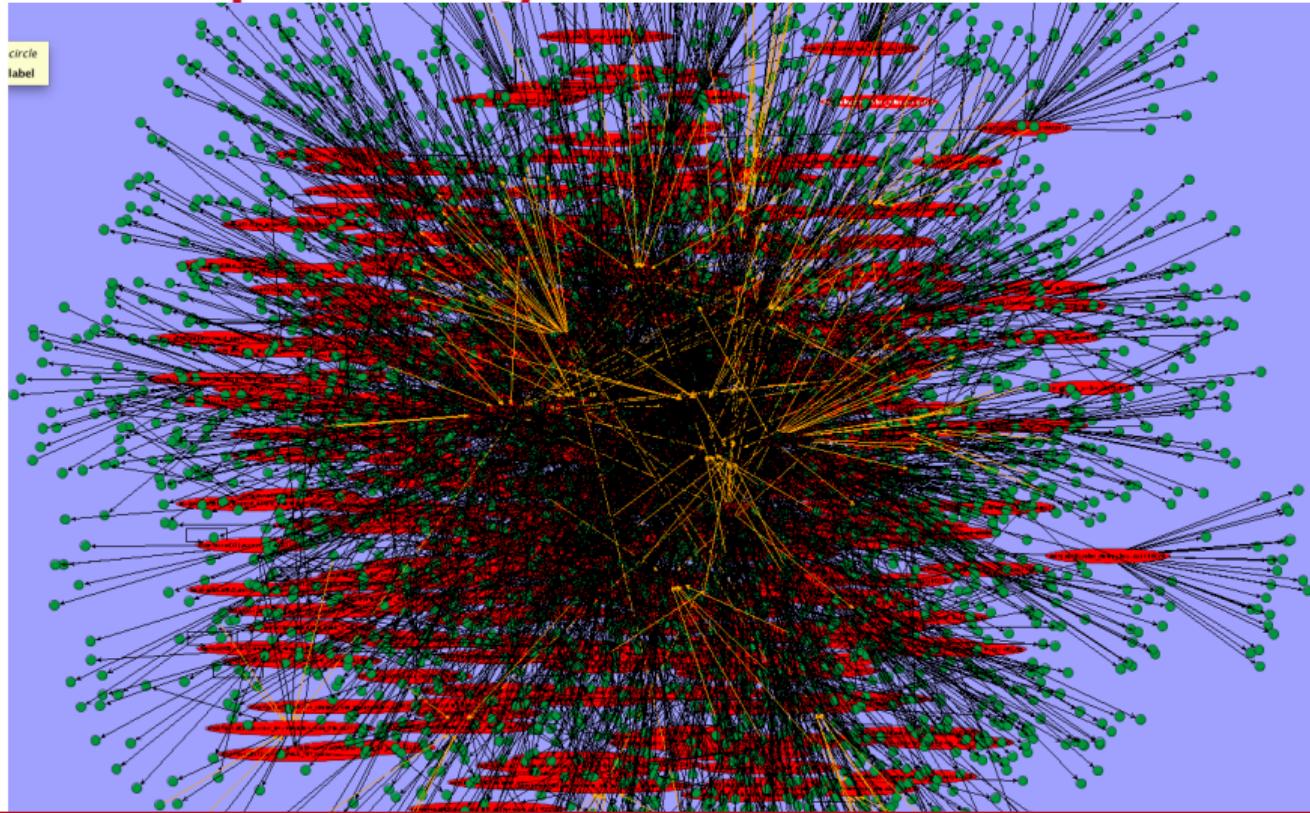
Call Graph of 176.gcc (230.5K lines of code)



#functions: 2256   #pointers: 134380   #loads/stores: 51543

Costly to reason about flow of values on CFGs!

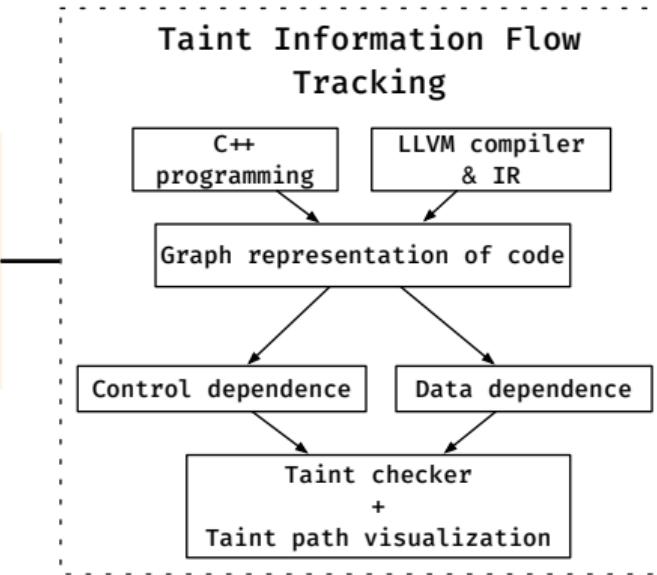
# Call Graph of 176.gcc



# The Project of This Subject

What are the building blocks to write a taint checker?

```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
```



```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
```

source

sink

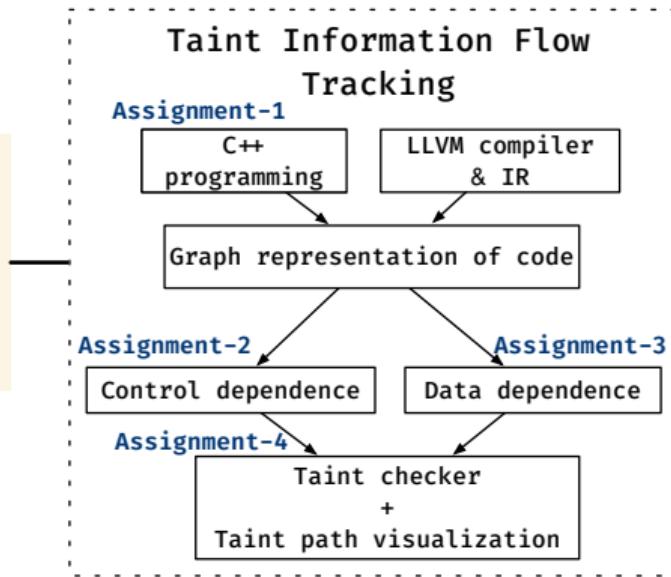
→ Control dependence

→ Data dependence

# The Project of This Subject

What are the building blocks to write a taint checker?

```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8
9     execl(cmd, "ls", "-l", NULL);
}
```



```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7     char* cmd = home;
8     execl(cmd, "ls", "-l", NULL);
9 }
```

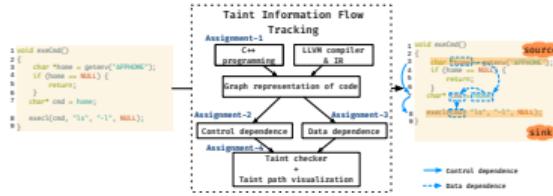
**source**      **sink**

This diagram shows the execution flow of the `exeCmd()` function. The variable `home` is highlighted in blue and labeled as a **source**. The variable `cmd` is also highlighted in blue and labeled as a **sink**. Blue arrows indicate the flow of tainted data from the source to the sink, representing both control and data dependences.

The final prototype tool will be a taint checker and its taint path visualization.

# The Project of This Subject

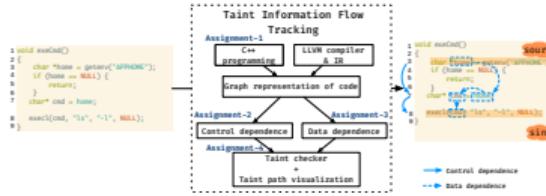
The project sounds complicated?



# The Project of This Subject

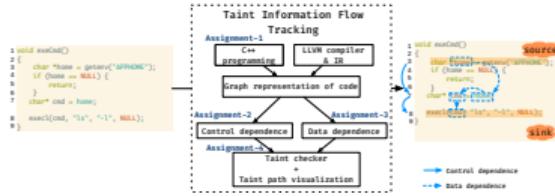
The project sounds complicated?

- Do I need to implement it from scratch?



# The Project of This Subject

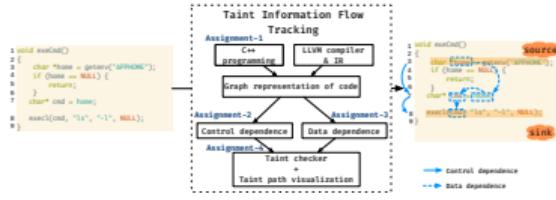
The project sounds complicated?



- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (<https://github.com/SVF-tools/SVF>)
- How many lines of code do I need to write?

# The Project of This Subject

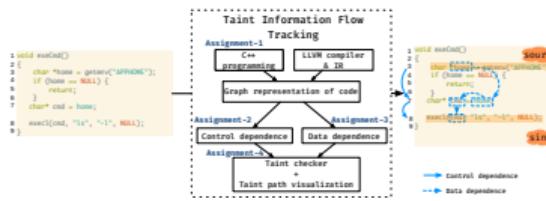
The project sounds complicated?



- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (<https://github.com/SVF-tools/SVF>)
- How many lines of code do I need to write?
  - **500-800 lines** of core code **in total** for all the four assignments
- Really? What are the challenges then?

# The Project of This Subject

The project sounds complicated?



- Do I need to implement it from scratch?
  - **No**, you will implement a lightweight tool based on the open-source framework SVF (<https://github.com/SVF-tools/SVF>)
- How many lines of code do I need to write?
  - **500-800 lines** of core code **in total** for all the four assignments
- Really? What are the challenges then?
  - Good C++ programming and debugging skills
  - Understanding basic principles of compilers and graph representation of code
  - Understanding static code analysis and apply it in practice
  - **Please do attend each class** to make sure you can catch up!

# What's next?

- (1) Self-enrol groups on canvas.
  - Though you will join a group, it is used to discuss assignment tasks and solve programming issues. You will still need to submit your code implementation individually for each assignment.
- (2) Configure Programming Environment  
[https://github.com/SVF-tools/SVF-Teaching/wiki/  
Installation-of-Docker,-VSCode-and-its-extensions](https://github.com/SVF-tools/SVF-Teaching/wiki/Installation-of-Docker,-VSCode-and-its-extensions)
  - Write and run your program in a docker container (virtual machine) on top of any operating system.
- (3) Write a hello world C++ program.
- (4) Revisit and practice C++ programming (More about C++ programming will be coming next week)