Week 1 Introduction to Software Analysis

https://github.com/SVF-tools/SVF-Teaching

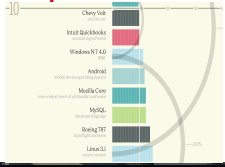
Yulei Sui

University of Technology Sydney, Australia

June 2021

Modern System Software

Extremely large and complex but error-prone



More Complex!

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

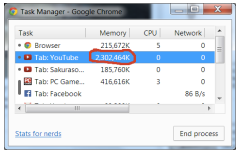
Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

- 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

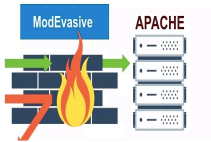
massive leaks over 2GB on a single browser tab

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

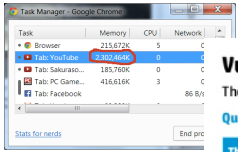
11 civilians died

uninitialized variables

password leakage via *tar* on Solaris OS

Modern System Software

Extremely large and complex but error-prone

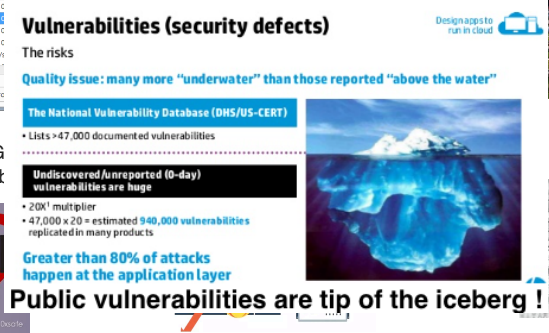


memory leaks

massive leaks over 2G on a single browser tab

buffer overflow

66% websites affected



null pointer

denial of service affecting millions of servers worldwide

data race

11 civilians died

uninitialized variables

password leakage via *tar* on Solaris OS

Memory Leak

- A dynamically allocated object is not freed along some execution path of a program
- A major concern of long running server applications due to gradual loss of available memory.

```
1  /* CVE-2012-0817 allows remote attackers to cause a denial of service through adversarial connection requests.*/
2  /* Samba --libads/ldap.c:ads_leave_realm */.
3
4  host = memAlloc(hostname);
5   ...
6  if (...)   {...; return ADS_ERROR_SYSTEM(ENOENT);} // The programmer forgot to release host on error.
7
```

```
1  /* A memory leak in Php-5.5.11 */
2  for (...) {
3      char* buf = readBuffer();
4      if (condition)
5          printf (buf);
6      else
7          continue;   // buf is leaked in else branch
8      freeBuf(buf);
9  }
```

Buffer Overflow

- Attempt to put more data in a buffer than it can hold.
- Program crashes, undefined behavior or zero-day exploit[1].

```
1  /* A simplified example from "Young and Mchugh, IEEE S&P 1987", exploited by attackers to bypass verfication*/
2
3    void verfiyPassword(){
4      char buff[15]; int pass = 0;
5       printf ("\n Enter the password : \n");
6      gets(buff);
7
8      if (strcmp(buff, "thegeekstuff")){     // return non-zero if the two strings do not match
9           printf ("\n Wrong Password \n");
10     }
11     else{                                  // return zero if two strings matched or a buffer overrun
12          printf ("\n Correct Password \n");
13          pass = 1;
14     }
15     if (pass)
16          printf ("\n Root privileges given to the user \n");
17   }
18
```

[1] Heartbleed, a well-known vulnerability in OpenSSL is also caused by buffer overflow (It took more than 2 years to discover and fix it since first patch, and over 500,000 websites were affected). Vulnerability is exploited when more data can be read than should be allowed.

Uninitialized Variable

- Stack variables in C and C++ are not initialized by default.
- Undefined behavior or denial of service via memory corruption

```
1  /* An uninitialized variable vulnerability simplified from gnuplot (CVE-2017-9670) */
2
3  void load(){
4          switch ( ctl ) {
5                  case -1:
6                          xN = 0; yN = 0;
7                          break;
8                  case 0:
9                          xN = i; yN = -i;
10                         break;
11                 case 1:
12                         xN = i + NEXT_SZ; yN = i - NEXT_SZ;
13                         break;
14                 default:
15                         xN = -1; xN = -1; // xN is accidentally set twice while yN is uninitialized
16                         break;
17         }
18                 plot(xN, yN);
19 }
20
21
```

Use-After-Free

- Attempt to access memory after it has been freed.
- Program crashes, undefined behavior or zero-day exploit.

```
1  /* CVE-2015-6125 and CVE-2018-12377 with similar heap use after free patterns*/
2
3  char* msg = memAlloc(...);
4  ...
5  if (err) {
6          abrt = 1;
7           ....
8          free(msg); // the memory is released when an error occurs at server
9  }
10 ...
11 if (abrt) {
12          ...
13          logError("operation aborted before commit", msg); // try to access released heap variable,
14                                                            // causing either crash or writing confidential data
15 }
```

Data Race

- A data race occurs when two threads access the same memory concurrently and at least one of the accesses is for writing.
- Program crashes, undefined behavior and zero-day exploit.

```
1  typedef std::map<std::string, u32_int> map_t;
2
3  void *balance_Inquire(void *p) {
4    map_t& m = *(map_t*)p;
5    m["client"] = amount;     // map m is written in thread t
6    return 0;
7  }
8
9  int main() {
10   map_t m;
11   pthread_t t;
12   pthread_create(&t, 0, threadfunc, &m);
13    printf ("client=%d\n", m["client"]);          // map m is read in thread main
14   pthread_join(t, 0);
15 }
16
```

Code Review by Developers

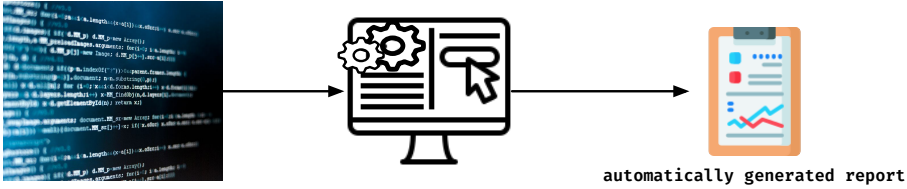
However ...

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

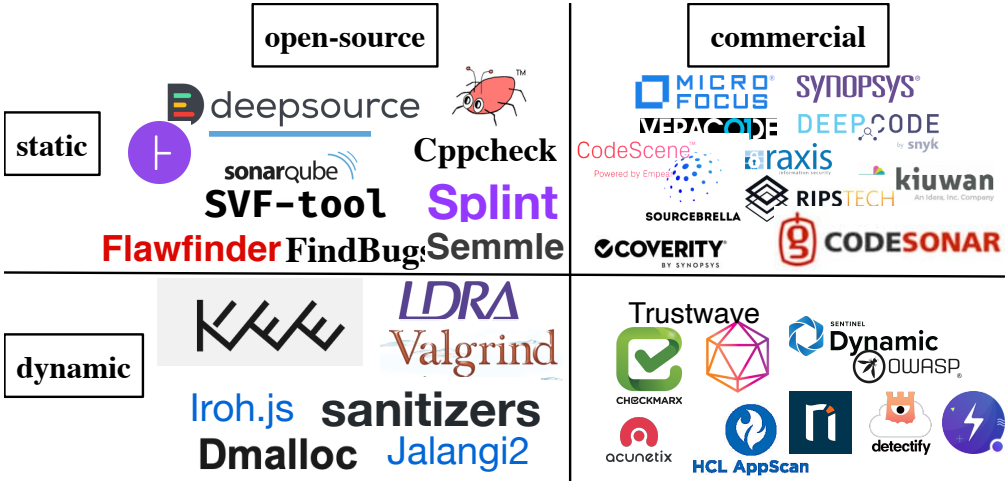
Static Analysis

- *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.

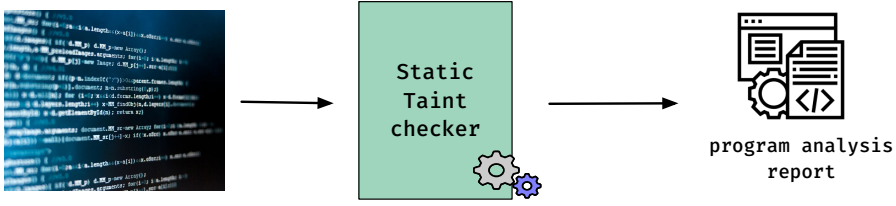
Existing Program Analysis Tools



The Project of This Subject

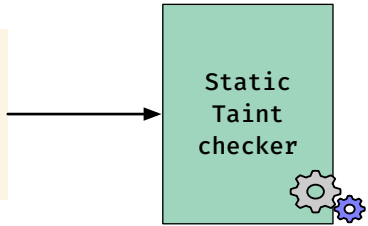
Goal of this subject: develop your own software analysis tool in 12 weeks.

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



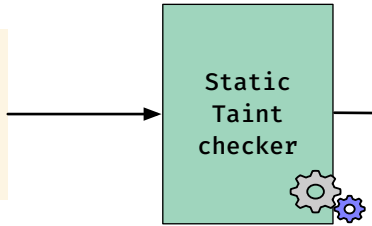
The Project of This Subject

```
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

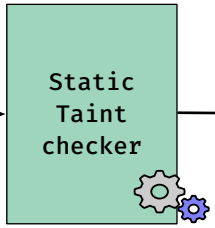
```
1 void exeCmd()
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7    char* cmd = home;

8    execl(cmd, "ls", "-l", NULL);
9 }
```




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

8    execl(cmd, "ls", "-l", NULL);
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    execl(cmd, "ls", "-l", NULL);
9 }
```

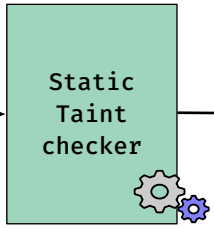



```
1 void exeCmd()                    source
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7    char* cmd = home;

8    execl(cmd, "ls", "-l", NULL);
9 }                                 sink
```

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    execl(cmd, "ls", "-l", NULL);
9 }
```



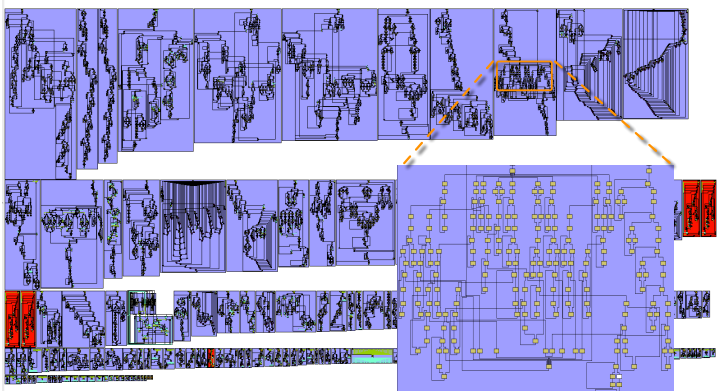

```
1 void exeCmd()                    source
2 {
3     char *home = getenv("APPHOME");
4     if (home == NULL) {
5         return;
6     }
7    char* cmd = home;

8    execl(cmd, "ls", "-l", NULL);
9 }                                 sink
```

Control dependence

Data dependence

How about real-world large programs?

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

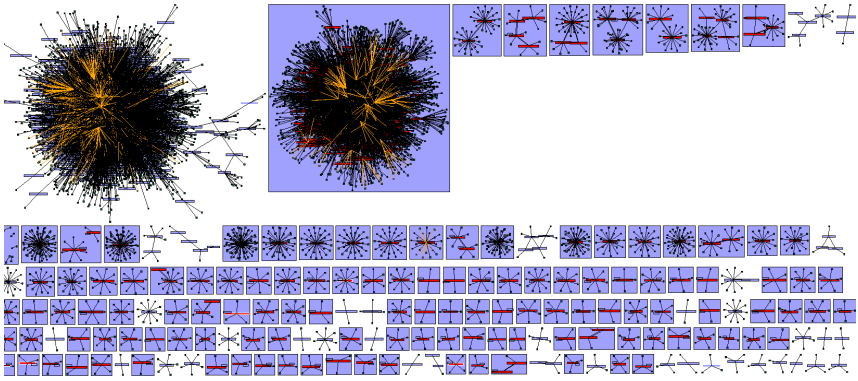


#functions: 194

#pointers: 20773 #loads/stores: 8657 Costly to reason about flow of values on CFGs!

How about real-world large programs?

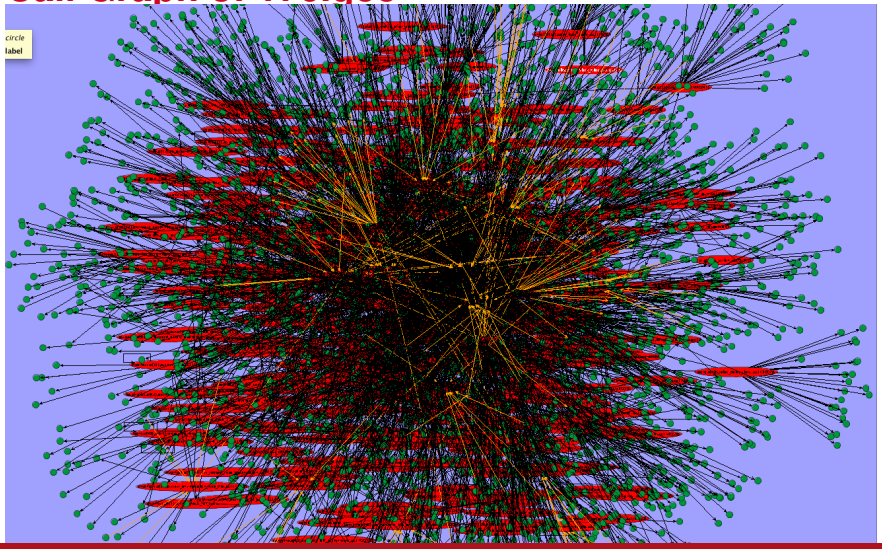
Call Graph of 176.gcc (230.5K lines of code)



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

Costly to reason about flow of values on CFGs!

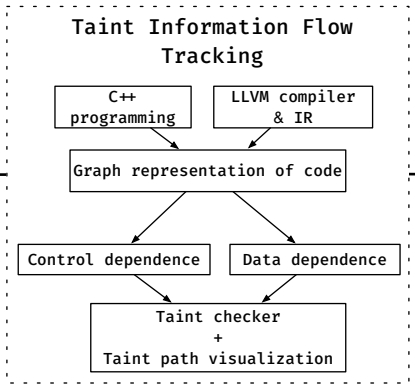
Call Graph of 176.gcc



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
8    execl(cmd, "ls", "-l", NULL);
9 }
```




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

source

sink

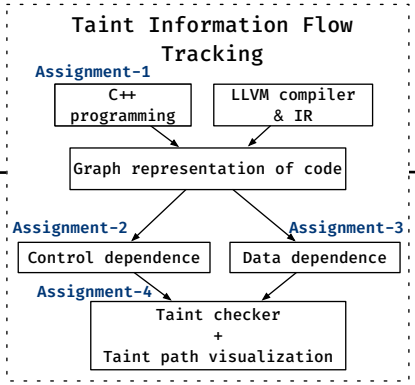
Control dependence

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    execl(cmd, "ls", "-l", NULL);
9 }
```




```
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

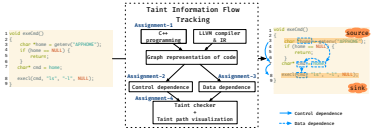
Control dependence

Data dependence

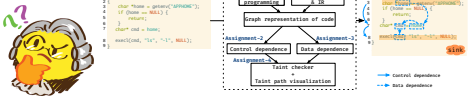
The final prototype tool will be a taint checker and its taint path visualization.

The Project of This Subject

The project sounds complicated?



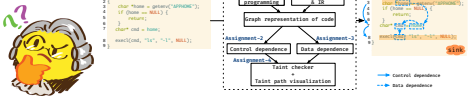
The Project of This Subject



The project sounds complicated?

- Do I need to implement it from scratch?

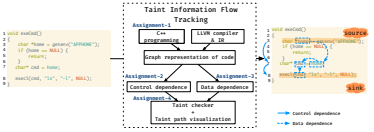
The Project of This Subject



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?

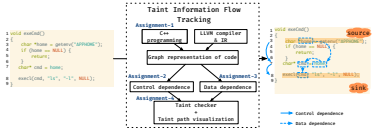
The Project of This Subject



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`
  - 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)