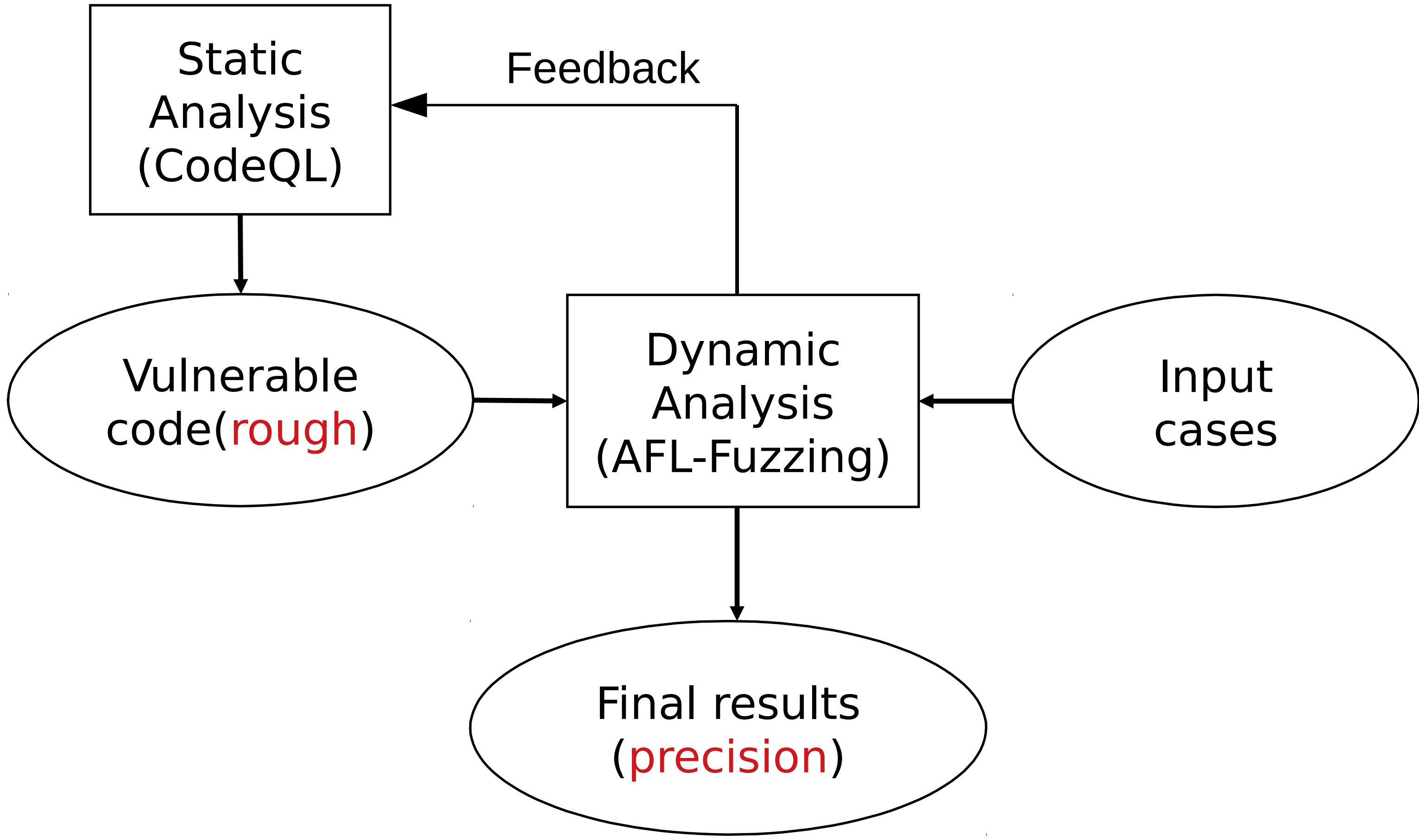
Fuzzing Aided Static Analyzer

Name: Bin Xie
Advisor : Professor Yue

Design overview

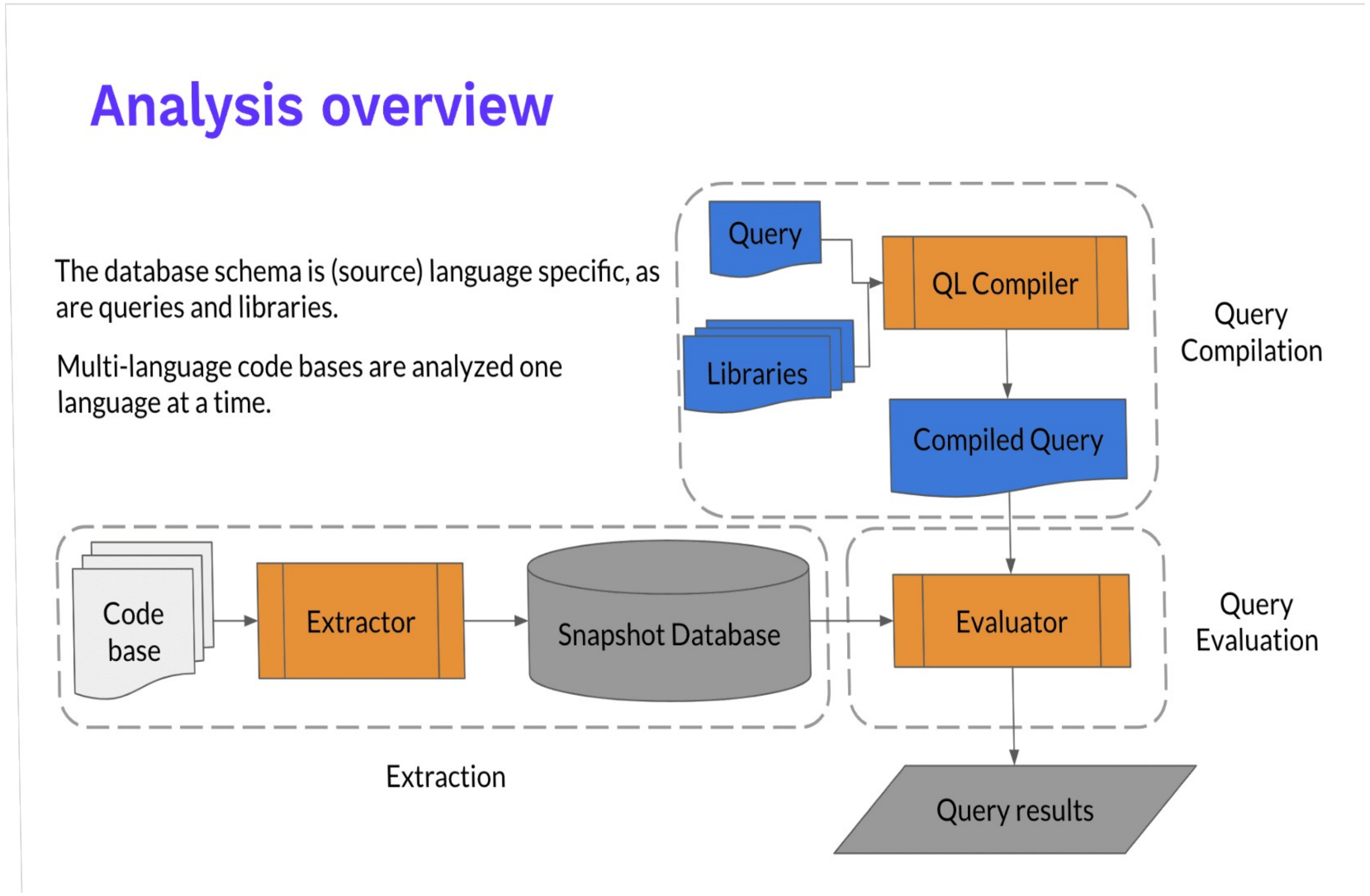
- 1. Firstly, the static analysis tool, CodeQL, will use several special patterns and codebase to produce locations of vulnerable code. The precision is low.
- 2. Secondly, AFL will receive several input cases and implement to improve the precision.
- 3. Finally, the results of vulnerabilities will be output.



Know and find limitations of CodeQL

CodeQL overview

- CodeQL is the state-of-the-art static analysis tool.
- It uses source code to build a database.
- Query several special attributes from database to find matched attributes.
- Map matched attributes to source codes.



Limitations of CodeQL

- There are a lot of false positive in most static analysis tools, as well as CodeQL, meanwhile some false negative.
    - Point-to Analysis
    - Standard library function
    - Aliasing
    - etc

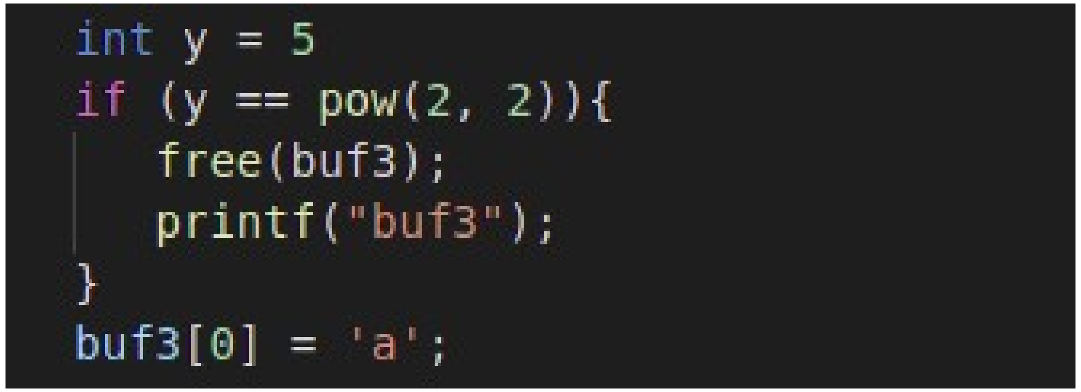
Two examples

- Use-after-free defect

- Taint tracking: the code is vulnerable if tainted data flows from a network integer source(ntoh, ntohll, or ntohs) to a sink in the length argument of a memcpy call.

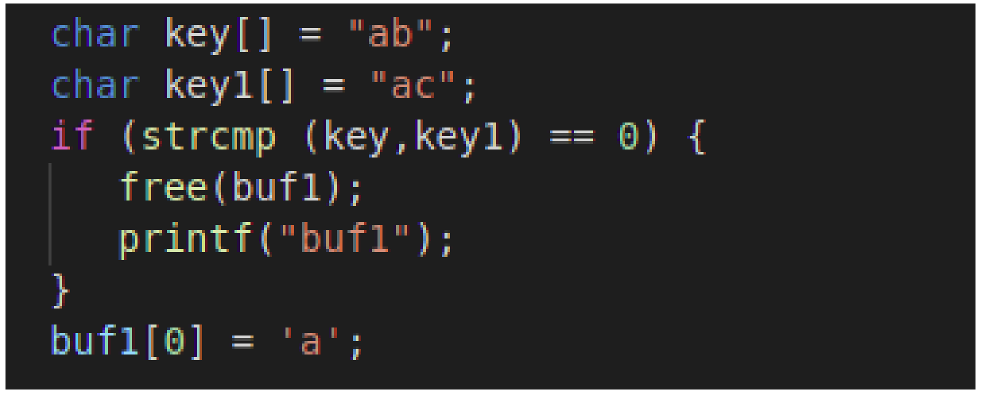
False positive: Standard library function

- Although the if conditions that contain standard library functions are always false, CodeQL reports use-after-free defects, which are false positive



```
int y = 5
if (y == pow(2, 2)){
    free(buf3);
    printf("buf3");
}
buf3[0] = 'a';
```

The expression uses pow() in math.h

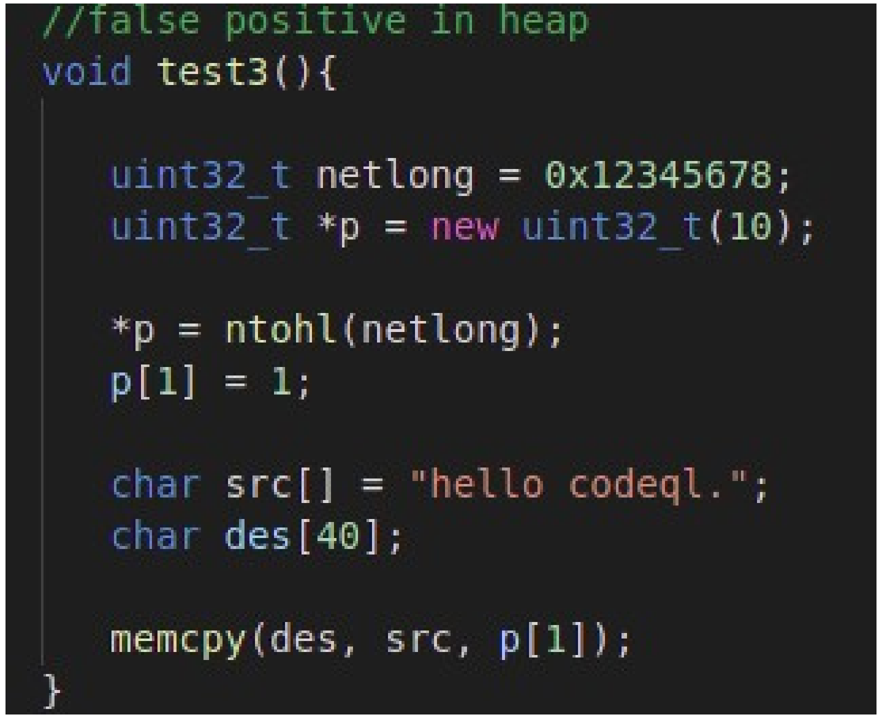


```
char key[] = "ab";
char key1[] = "ac";
if (strcmp (key,key1) == 0) {
    free(buf1);
    printf("buf1");
}
buf1[0] = 'a';
```

The expression uses strcmp() in string.h

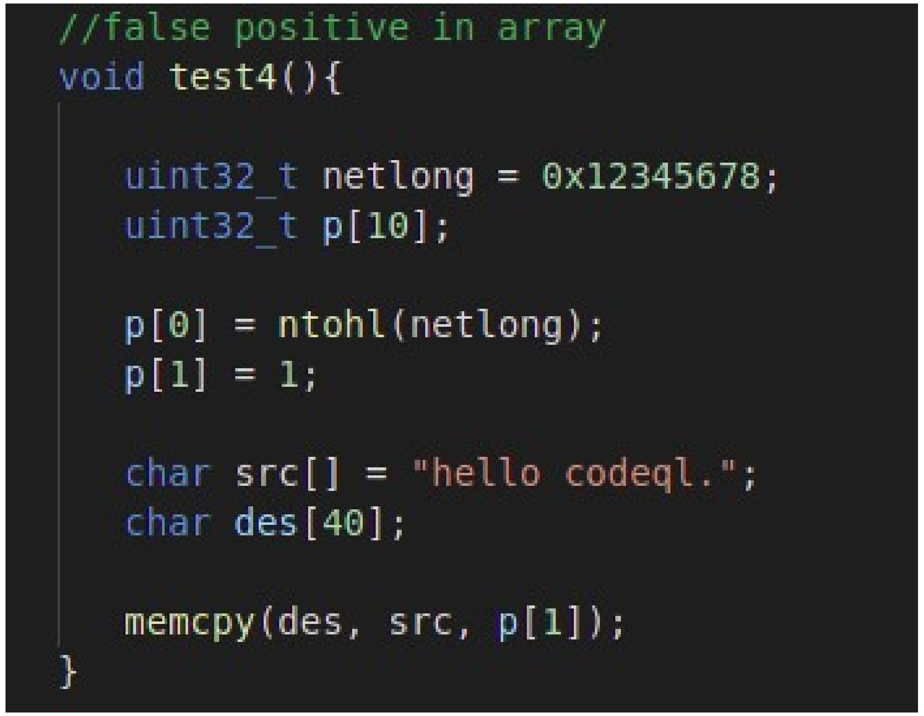
False positive: Point-to Analysis

- If there is a taint pointer to point an address in a segment of memory, all addresses in the segment will be taint.



```
//false positive in heap
void test3(){

    uint32_t netlong = 0x12345678;
    uint32_t *p = new uint32_t(10);

    *p = ntohl(netlong);
    p[1] = 1;

    char src[] = "hello codeql.";
    char des[40];

    memcpy(des, src, p[1]);
}
```

False positive in heaps

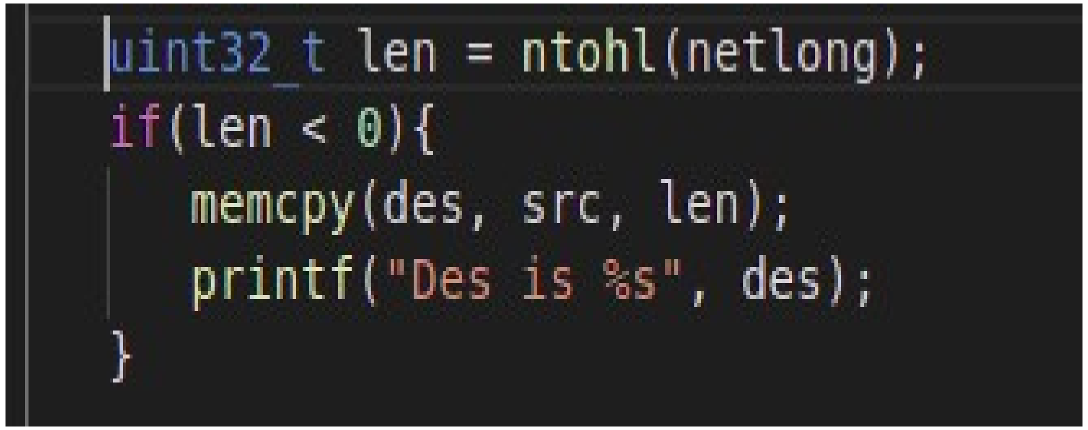


```
//false positive in array
void test4(){

    uint32_t netlong = 0x12345678;
    uint32_t p[10];

    p[0] = ntohl(netlong);
    p[1] = 1;

    char src[] = "hello codeql.";
    char des[40];

    memcpy(des, src, p[1]);
}
```

False positive in arrays

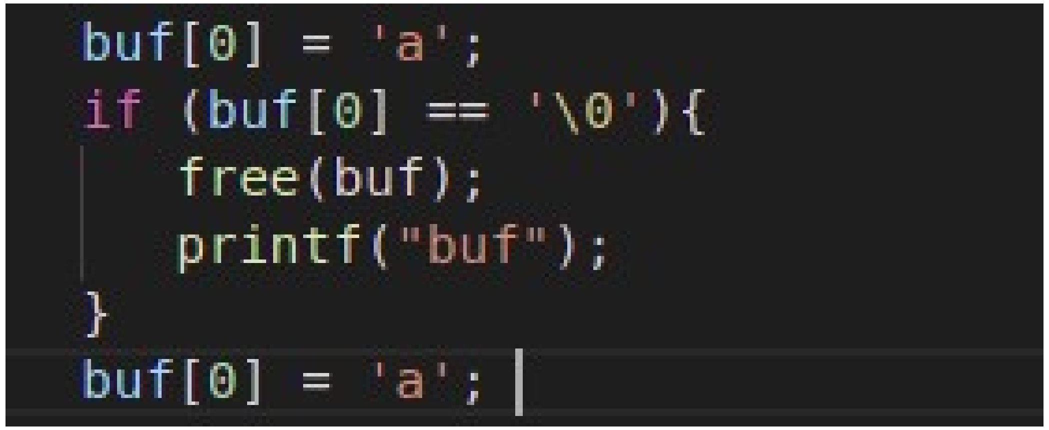
False positive: Condition expression contains taint values

- Although the if conditions that contain taint values are always false, CodeQL reports defects, which are false positive.



```
uint32_t len = ntohl(netlong);
if(len < 0){
    memcpy(des, src, len);
    printf("Des is %s", des);
}
```

“len” is never less than 0

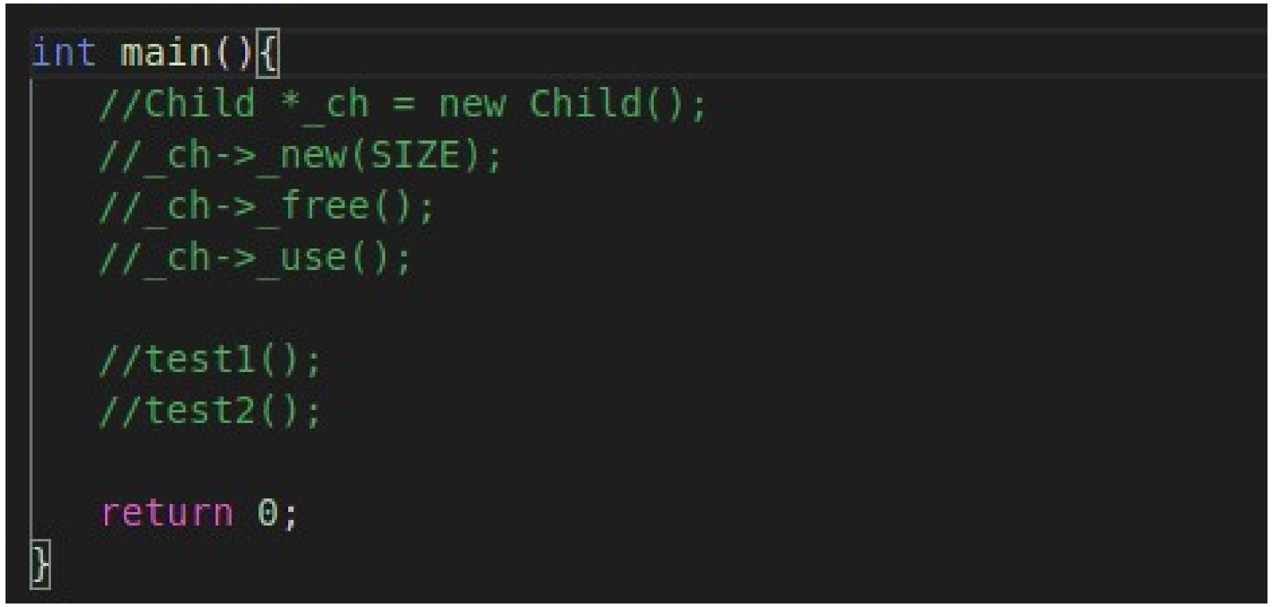


```
buf[0] = 'a';
if (buf[0] == '\0'){
    free(buf);
    printf("buf");
}
buf[0] = 'a';
```

“buf[0]” is never equal to ‘\0’

False positive: Never executed

- Although some codes don’t execute, CodeQL would report a defect if these codes have a specified pattern.



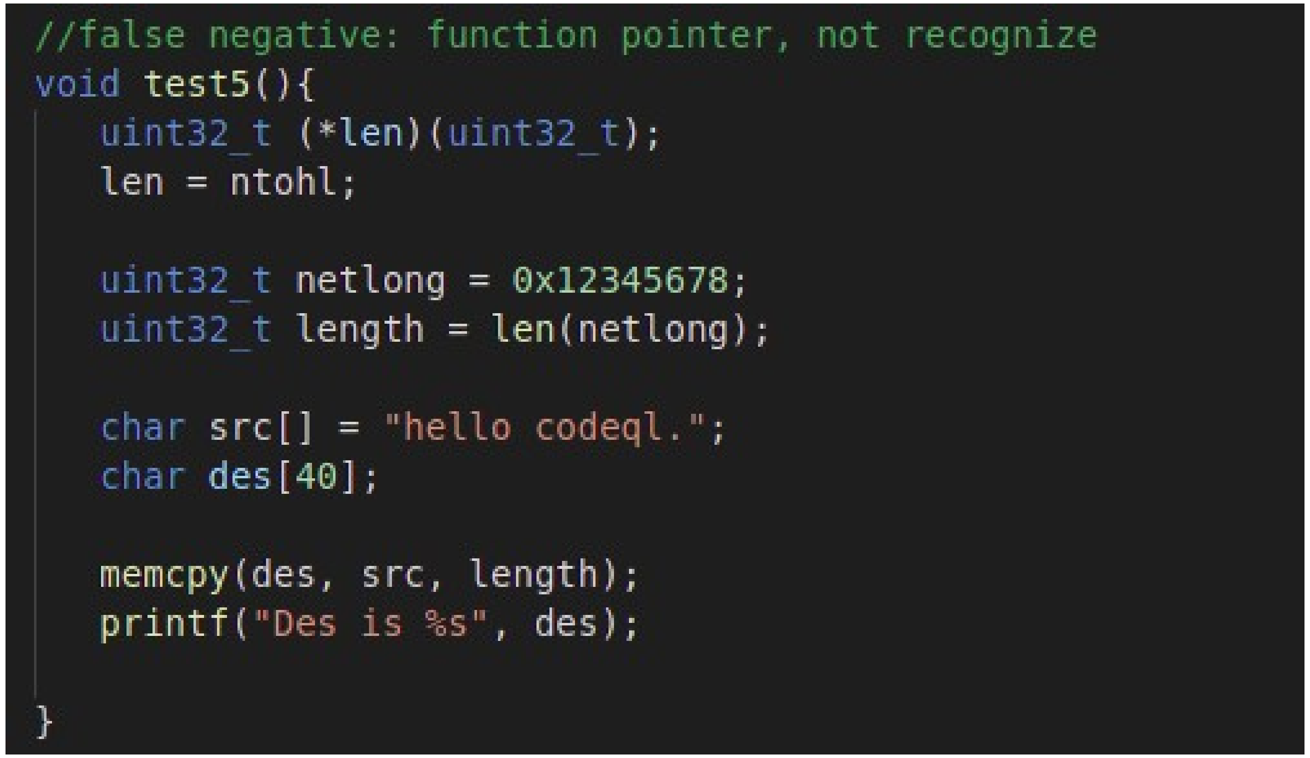
```
int main(){
    //Child *_ch = new Child();
    //_ch->_new(SIZE);
    //_ch->_free();
    //_ch->_use();

    //test1();
    //test2();

    return 0;
}
```

Main() do nothing.

test1() have a use-after-free error.
Although test1() never execute,
CodeQL also reports this defect.

False negative: Function pointer

- If there is a function pointer that points a taint function, CodeQL will never report this defect.



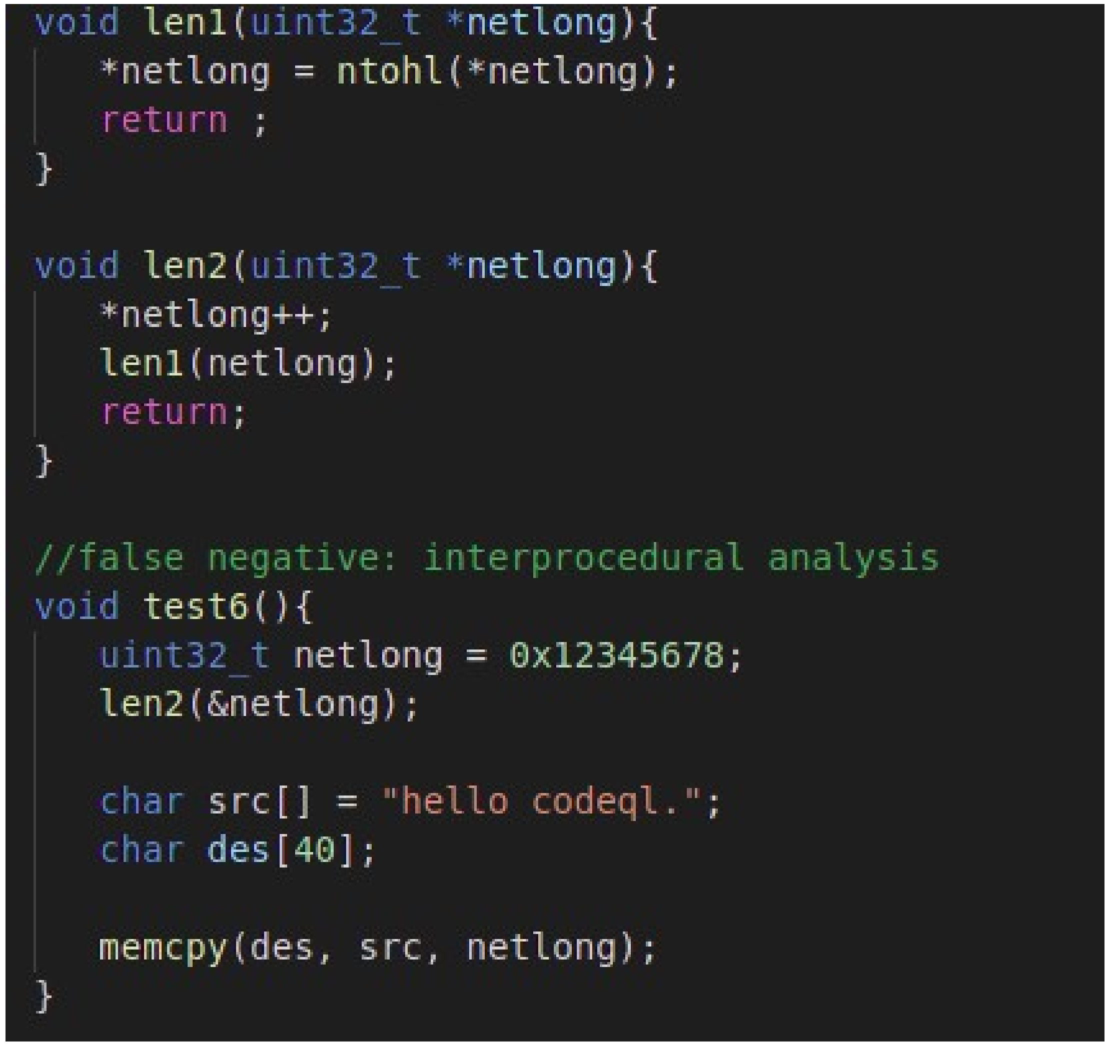
```
//false negative: function pointer, not recognize
void test5(){
    uint32_t (*len)(uint32_t);
    len = ntohl;

    uint32_t netlong = 0x12345678;
    uint32_t length = len(netlong);

    char src[] = "hello codeql.";
    char des[40];

    memcpy(des, src, length);
    printf("Des is %s", des);


}
```

Function pointer

test1() have a use-after-free error.
Although test1() never execute,
CodeQL also reports this defect.

False negative: Interprocedural analysis

- If there are more than two functions that operate a taint value by pointer, CodeQL will never report this defect.



```
void len1(uint32_t *netlong){
    *netlong = ntohl(*netlong);
    return ;
}

void len2(uint32_t *netlong){
    *netlong++;
    len1(netlong);
    return;
}

//false negative: interprocedural analysis
void test6(){
    uint32_t netlong = 0x12345678;
    len2(&netlong);

    char src[] = "hello codeql.";
    char des[40];

    memcpy(des, src, netlong);
}
```

Pointer operation among interprocedural analysis

Aided by Fuzzing

- Fuzzing is a dynamic tool, and has a high throughput(execute thousands time per second.
  - False positive
    - Standard library function
    - Point-to analysis
    - Condition expression contains taint values
    - Never executed
  - False negative
    - Function pointer
    - Interprocedural analysis

Aided by Fuzzing

- Find more limitations
- Do instrumentations
- Combination between Fuzzing and CodeQL
- Reduce false positive and false negative