从源代码漏洞挖掘谈有价值研究

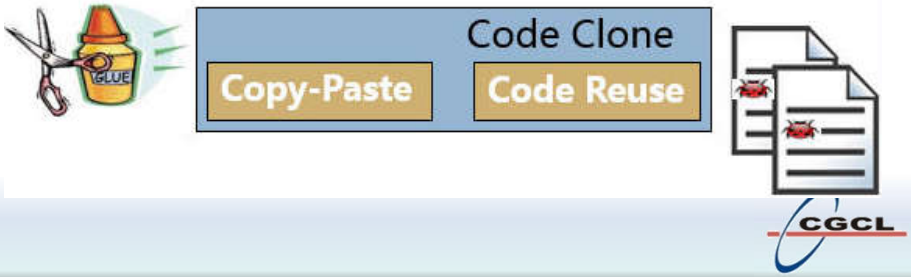
邹德清 教授

华中科技大学 网络空间安全学院

源代码漏洞挖掘

- 源代码含有丰富的语义信息，源代码漏洞挖掘能够与软件开发生命周期有效集成，便于更早发现并修补漏洞
- 软件的开源化趋势成为主流
  - 软件开发人员进行代码分享和代码托管

  - 安全缺陷随着开源软件的使用和扩散传播



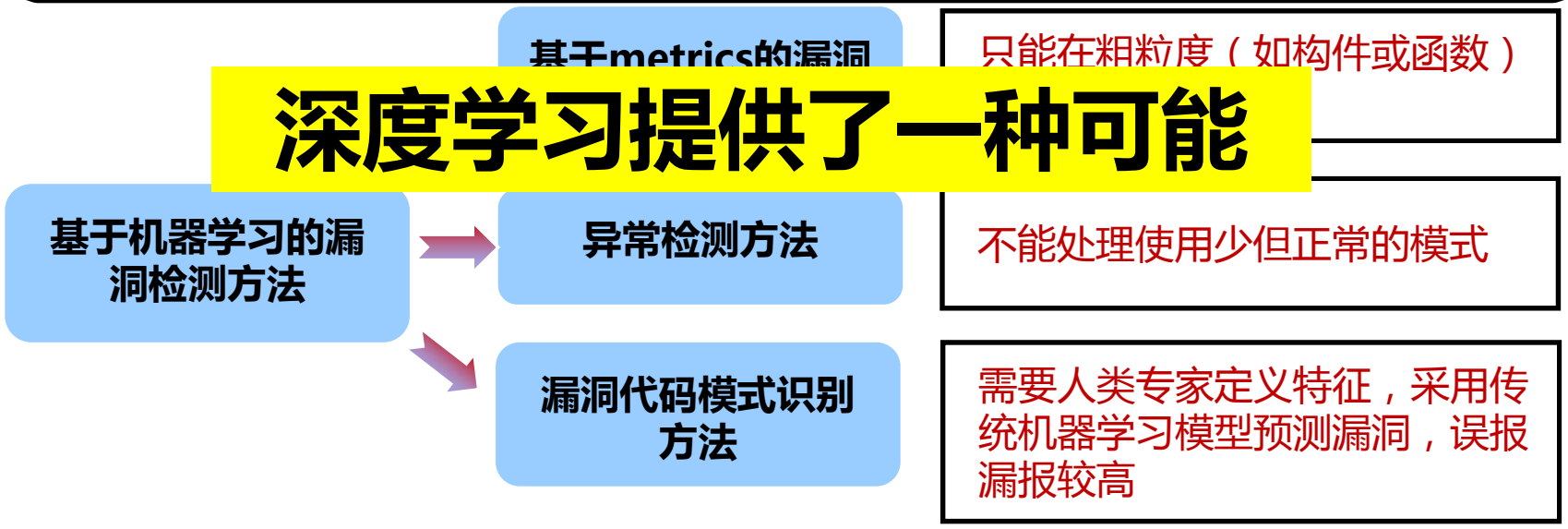
在源代码漏洞挖掘上已开展的工作

基于深度学习的漏洞检测

研究现状

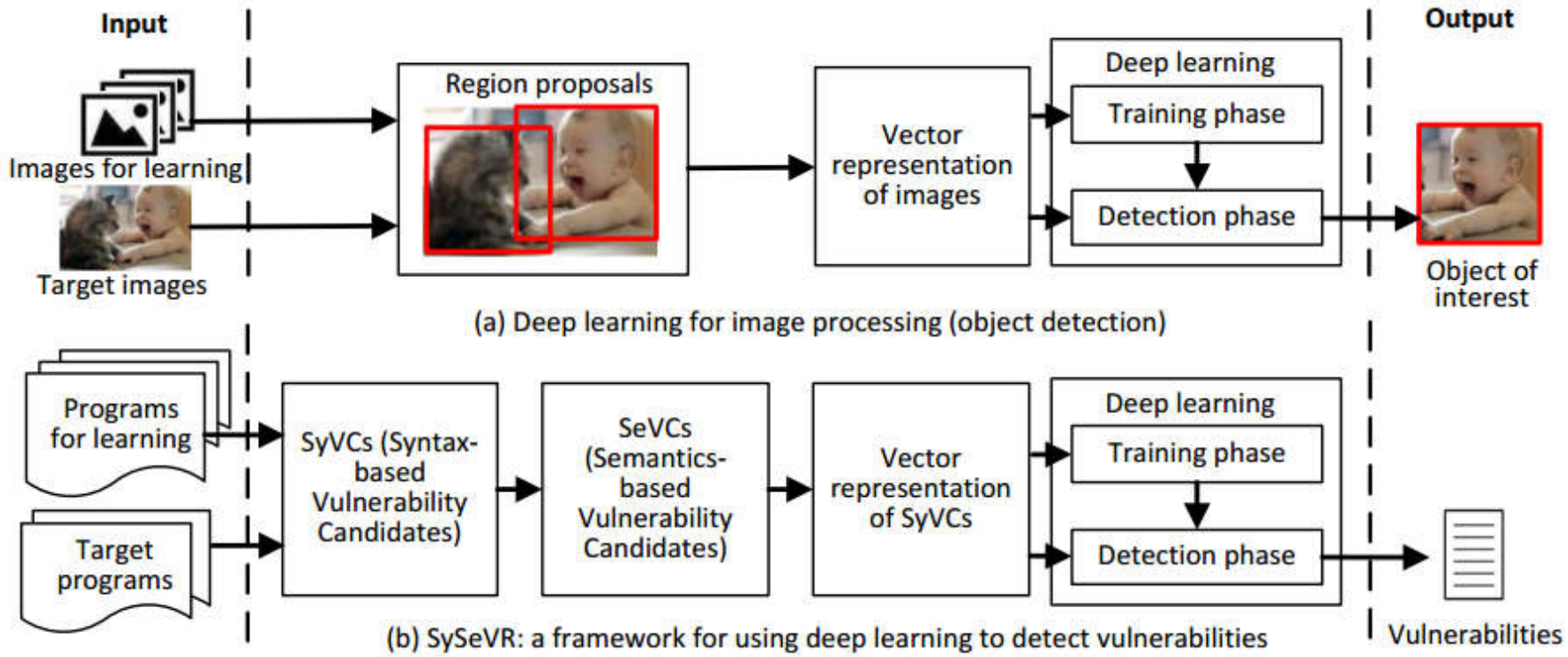
- 目前存在很多开源工具、商业产品及学术界研究，静态检测各种类型的漏洞

如何在不需人类专家定义特征的前提下，以较低的误报和漏报，自动检测目标程序是否含有漏洞？



1. SySeVR框架

- 借鉴计算机视觉领域中的目标检测过程，将深度学习引入源代码漏洞检测



定义SyVC和SeVC

- SyVC (Syntax-based Vulnerability Candidates)

  SyVC是一个代码元素，由代码中一个或多个token组成；该代码元素满足已知漏洞的某个基本语法特性，可能是漏洞，也可能不是漏洞

  基本语法特性：可通过现有的漏洞规则进行初步归类得到；仅是漏洞检测的出发点

  库/API函数调用 **数组使用** **指针使用** **算术表达式**

- SeVC (Semantics-based Vulnerability Candidates) 或code gadget

  SeVC是SyVC的扩展，由与SyVC语义相关（如控制依赖和数据依赖）的语句组成

SyVC和code gadget举例

```
1  void printLine(const char * line)
2  {
3      if(line != NULL)
4          printf("%s\n", line);
5  }
6
7  void func()
8  {
9      char *data;
10     char dataBuffer[100];
11     char source[100];
12     ...
13     memset(dataBuffer, 'A', 99);
14     dataBuffer[99] = '\0';
15     ...
16     while(1)
17     {
18         data = dataBuffer - 8;
19         break;
20     }
21     ...
22     memset(source, 'C', 99);
23     source[99] = '\0';
24     memmove(data, source,
       100*sizeof(char));
25     data[99] = '\0';
26     printLine(data);
27 }
```

源代码

```
1  void printLine(const char * line)
2  {
3      if(line != NULL)
4          printf("%s\n", line);
5  }
6
7  void func()
8  {
9      char *data;
10     char dataBuffer[100];
11     char source[100];
12     ...
13     memset(dataBuffer, 'A', 99);
14     dataBuffer[99] = '\0';
15     ...
16     while(1)
17     {
18         data = dataBuffer - 8;
19         break;
20     }
21     ...
22     memset(source, 'C', 99);
23     source[99] = '\0';
24     memmove(data, source,
       100*sizeof(char));
25     data[99] = '\0';
26     printLine(data);
27 }
```

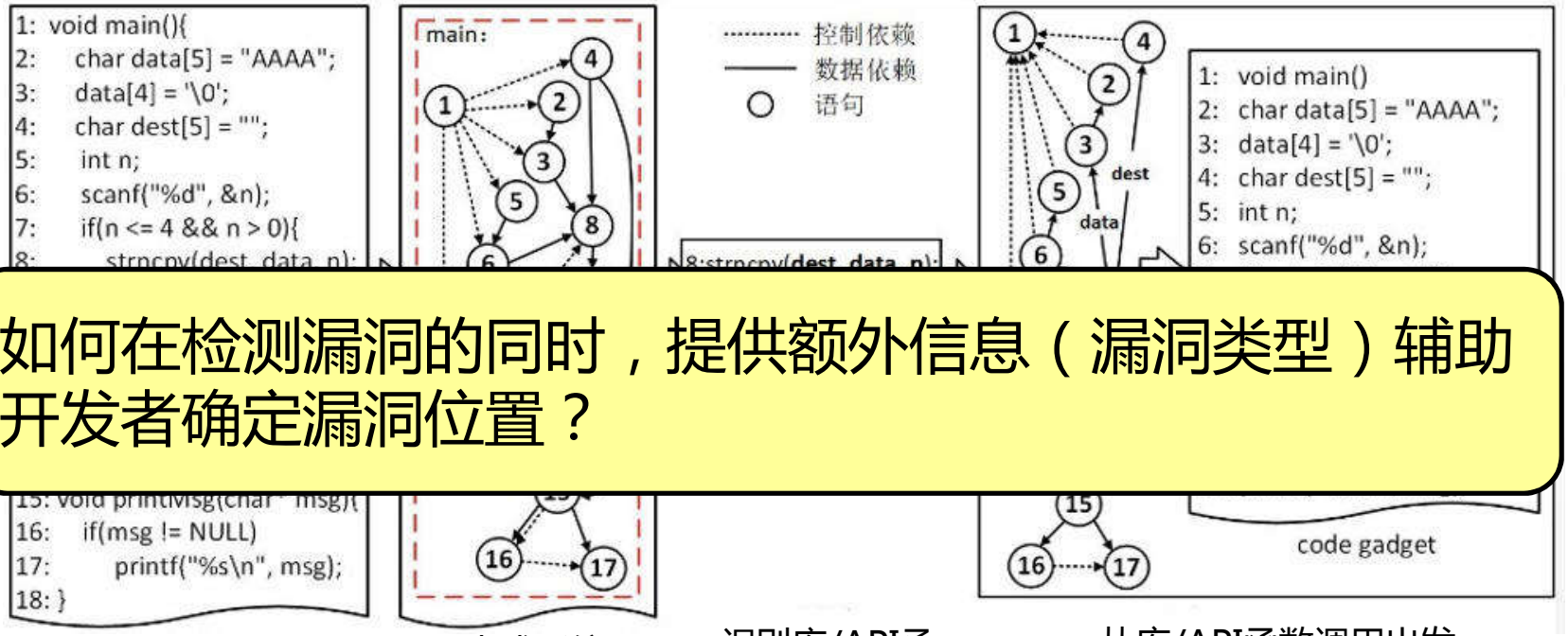
SyVC（红框标出）

```
7   void func()
9   char *data ;
10  char dataBuffer[100];
11  char source[100];
13  memset(dataBuffer, 'A', 99);
14  dataBuffer[99] = '\0';
16  while (1)
18  data = dataBuffer - 8;
22  memset(source, 'C', 99);
23  source[99] = '\0';
24  memmove (data, source,
    100*sizeof(char));
25  data[99] = '\0';
26  printLine(data);
1   void printLine(const char * line)
3   if(line != NULL)
4   printf("%s\n", line);
```

第25行SyVC “data”对应的code gadget

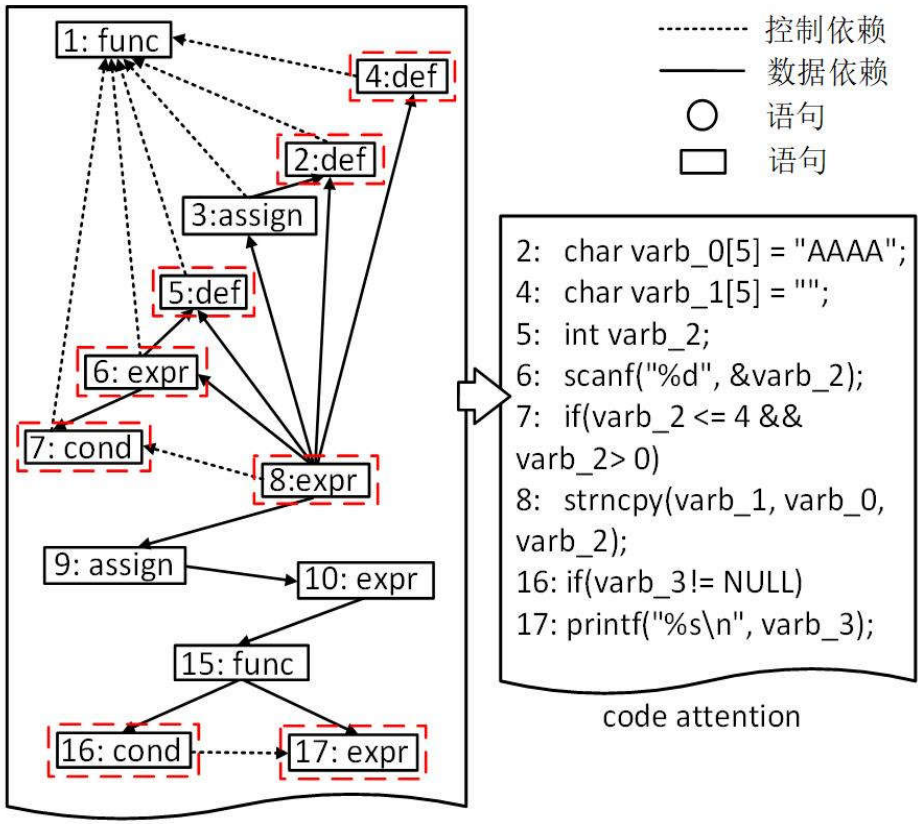
2.多类漏洞检测系统μVulDeePecker

- 问题描述
  - Code gadget可用于检测一段代码是否包含漏洞，但无法提供具体的漏洞信息，如漏洞类型等



如何在检测漏洞的同时，提供额外信息（漏洞类型）辅助开发者确定漏洞位置？

引入Code Attention



在code gadget中定位出匹配漏洞语法规则的语句

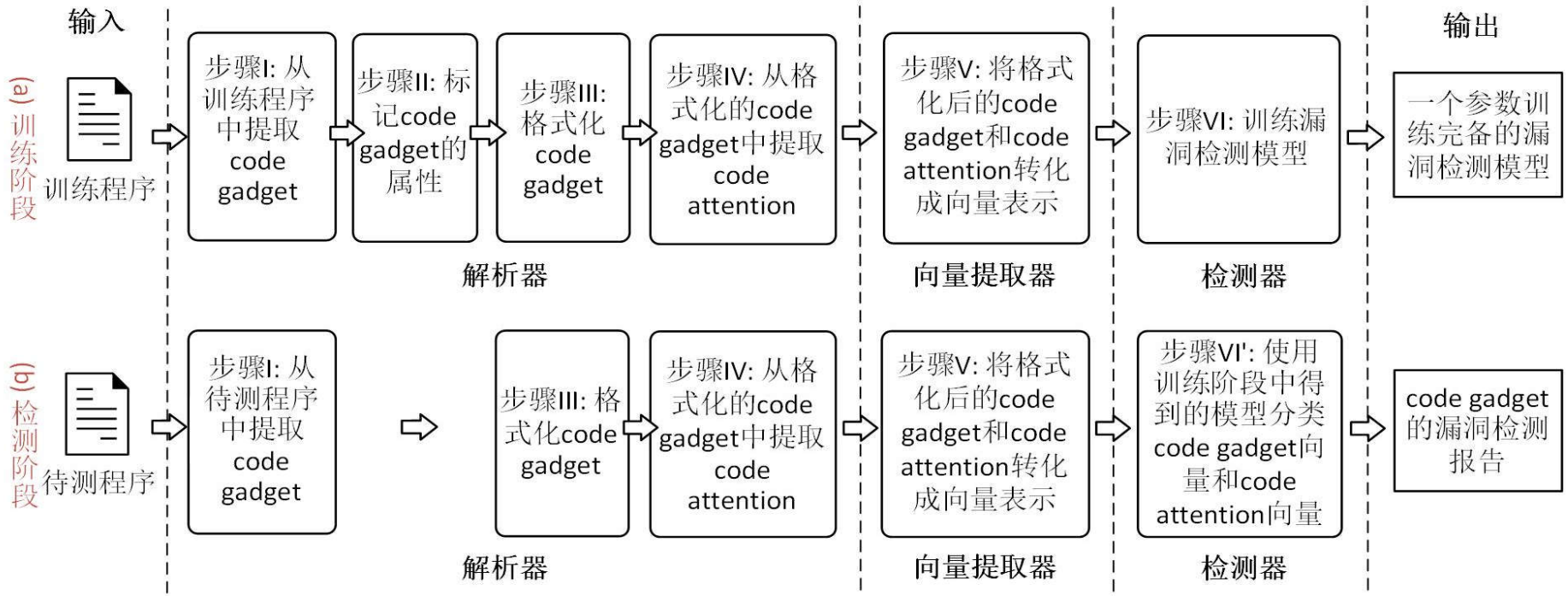
- 符合漏洞语法特征规则的代码语句集合
- 漏洞语法特征规则覆盖不安全库/API函数调用的数据源检查，操作条件检查以及调用合法性检查，包括：

  a.不安全的库/API函数调用参数的定义语句

  b.控制语句

  c.不安全的库/API函数调用语句
- 捕获“局部”信息，可辅助模型识别漏洞具体类型

μVulDeePecker结构



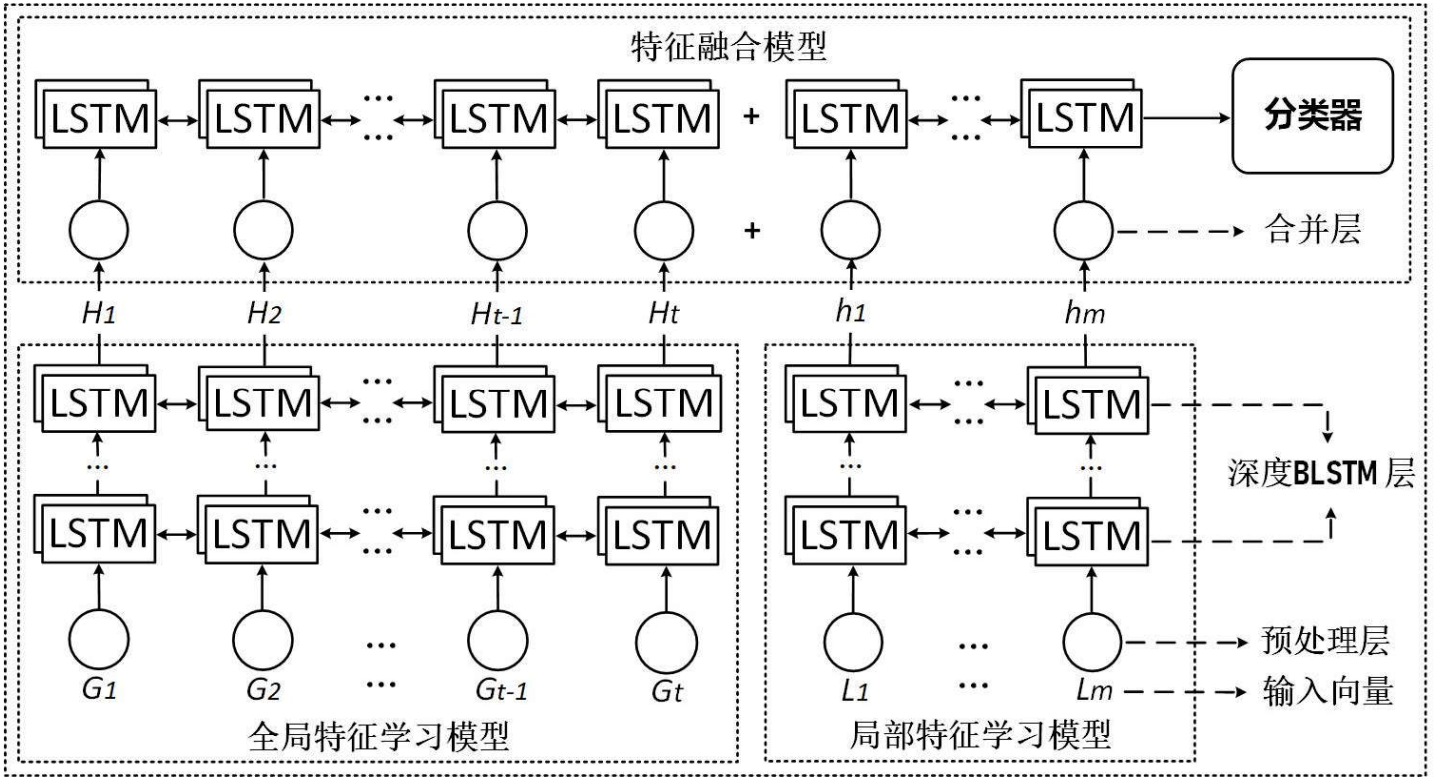
μVulDeePecker（multiclass Vulnerability Deep Pecker）系统一共包含三个模块：解析器、向量提取器和检测器，以及两个使用阶段：训练阶段和检测阶段

核心步骤

Step IV: 提取 code attention

- 利用词法分析技术分析code gadget语句属性
- 根据语句属性和代码文本匹配漏洞语法规则，提取code attention

Step VI: 训练模型



实验结论

结论一：现有基于深度学习的漏洞检测方法不适合直接用于多类漏洞检测，μVulDeePecker系统是可行的

结论二：μVulDeePecker系统对多类漏洞检测十分有效，而且可在真实软件中查找出新漏洞，具备一定的实用价值

结论三：控制依赖能够有效增强μVulDeePecker系统检测多种类型漏洞的能力

结论四：μVulDeePecker可配合其他基于深度学习的检测系统使用以提高检测多类漏洞的效果

3.利用中间代码进行细粒度漏洞检测

- 问题描述
  - 目前基于深度学习的源代码漏洞检测方法存在以下问题：
    - 漏洞定位精度不够：通常在函数级或切片级检测漏洞，无法精确定位漏洞
      - 例如，47.8%的切片含有至少20行语句，函数则含有更多行语句
    - 漏洞检测能力不够：利用源代码无法准确识别宏、全局变量等的定义和使用关系，无法完全暴露出控制流及其变量的定义使用关系

源代码漏洞候选代码举例

- 源代码无法识别如条件运算符的控制流；
- 无法识别不同文件（甚至相同文件）中的宏定义或全局变量使用

```
1  #define N 100
2  char *data;
3  void printLine()
4  {
5     if(data != "")
6        printf("%s\n", data);
7  }
8  int main()
9  {
10    char dataBuffer[N];
11    char source[N];
12    int m=99;
13    ...
14    memset(dataBuffer, 'A', N<m?N:99);
15    dataBuffer[99] = '\0';
16    ...
17    while(dataBuffer != "")
18    {
19       data = dataBuffer - 8;
20       break;
21    }
22    ...
23    memset(source, 'C', 99);
24    source[99] = '\0';
25    memmove(data, source, N*sizeof(char));
26    data[99] = '\0';
27    printLine();
28    return 0;
29 }
```

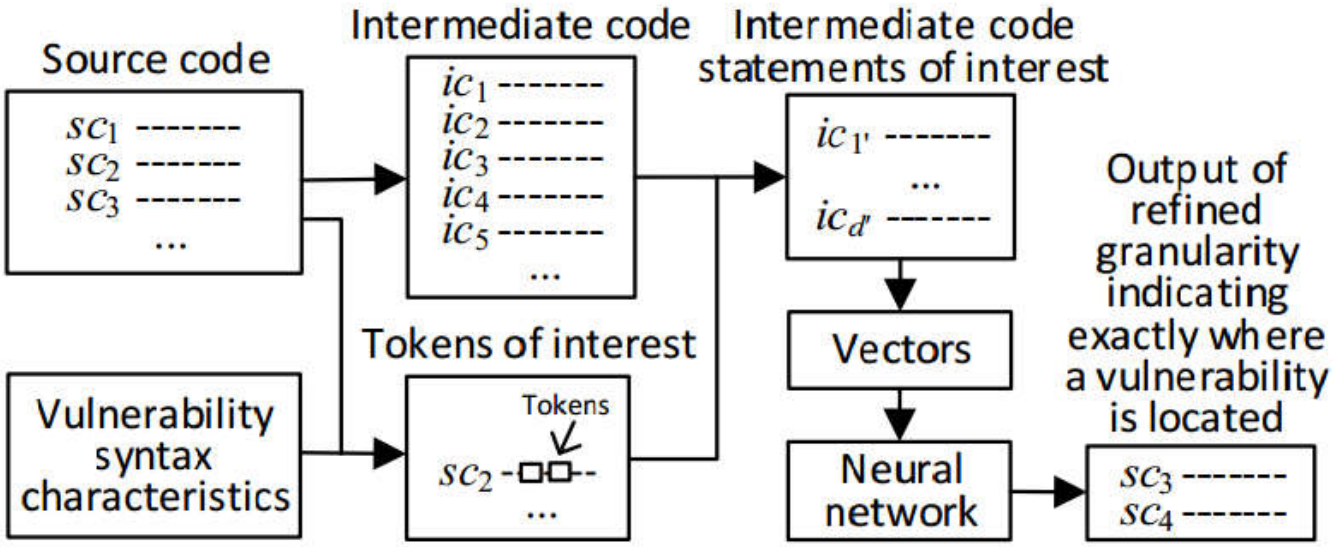
Source program

```
2    char *data;
8    int main()
10   char dataBuffer[N];
11   char source[N];
12   int m=99;
14   memset(dataBuffer, 'A', N<m?N:99);
15   dataBuffer[99] = '\0';
17   while(dataBuffer != "")
19   data = dataBuffer - 8;
23   memset(source, 'C', 99);
24   source[99] = '\0';
25   memmove(data, source, N*sizeof(char));
26   data[99] = '\0';
27   printLine();
3    void printLine()
5    if(data != "")
6    printf("%s\n", data);
```

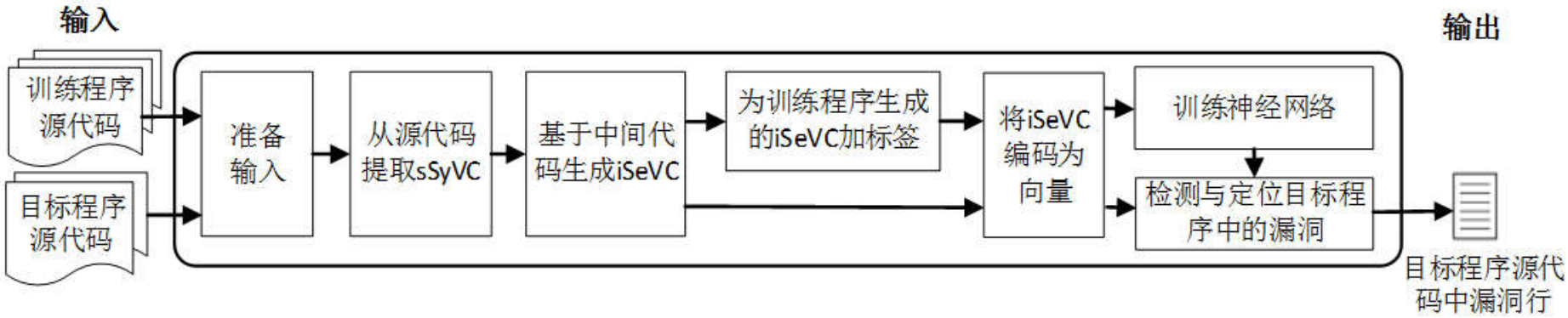
A source code-based vulnerability candidate related to “*data*” (Line 2)

主要思路

- 利用中间代码捕获比源码本身更多的语义信息
- 提出改进的深度神经网络模型，将输入粒度提炼为更细的输出粒度
  - 例如，输入32行代码，输出2行漏洞代码



VulDeeLocator系统结构



注：sSyVC为从源码中基于语法提取的与漏洞可能有关的token；
iSeVC为从中间代码中提取的与sSyVC语义相关的语句

并发漏洞检测

Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, Hai Jin: A Heuristic Framework to Detect Concurrency Vulnerabilities. ACSAC 2018: 529-541

背景

- 趋势：多线程和多核
  - 摩尔定律的终结，走向多核时代
  - 多线程是利用多核提供的算力的最主要的方式
  - 由并发漏洞造成的影响越来越大
    - Dirty COW (**C**opy **O**n **W**rite)

检测并发错误的挑战

- 与一般程序错误相比
  - 更难触发（程序执行过程中有几率发生）
    - 除了程序输入外，还需要线程调度
      - 某些特定的线程才会触发并发错误
      - 线程之间的调度所形成的搜索空间一般而言很大
  - 更难重现
    - 即使触发了并发错误，线程调度也很难重现
      - 很难获取触发了并发错误的线程调度
      - 即使获取了触发错误的线程调度、一般而言也需要重量级的工具去重现
  - 更难寻找错误的根源
    - 所有有并发关系的线程都需要查看

研究现状

并发错误检测	主要关注在数据竞争、原子违例、顺序违例，和并发漏洞没有什么关联。即使消除了上述三种典型的并发错误，并发漏洞仍然会出现
并发漏洞检测	1. 基于静态的并发漏洞检测：静态扫描程序，很难扩展到其他类型的并发漏洞； 2. 基于逻辑的方法：符号化执行，可扩展性差
模糊测试	模糊测试是检测程序错误的强力工具，然而其检测并发错误或漏洞还不够，目前的模糊测试只在于提高程序运行时的代码覆盖率，不擅长提高程序调度的覆盖率，一个并发程序往往有很多的线程调度需要覆盖

启发式并发漏洞检测框架

1. 在模糊测试中不断调整线程调度(e.g., AFL)
   - 探索尽可能多的线程调度来检测并发错误

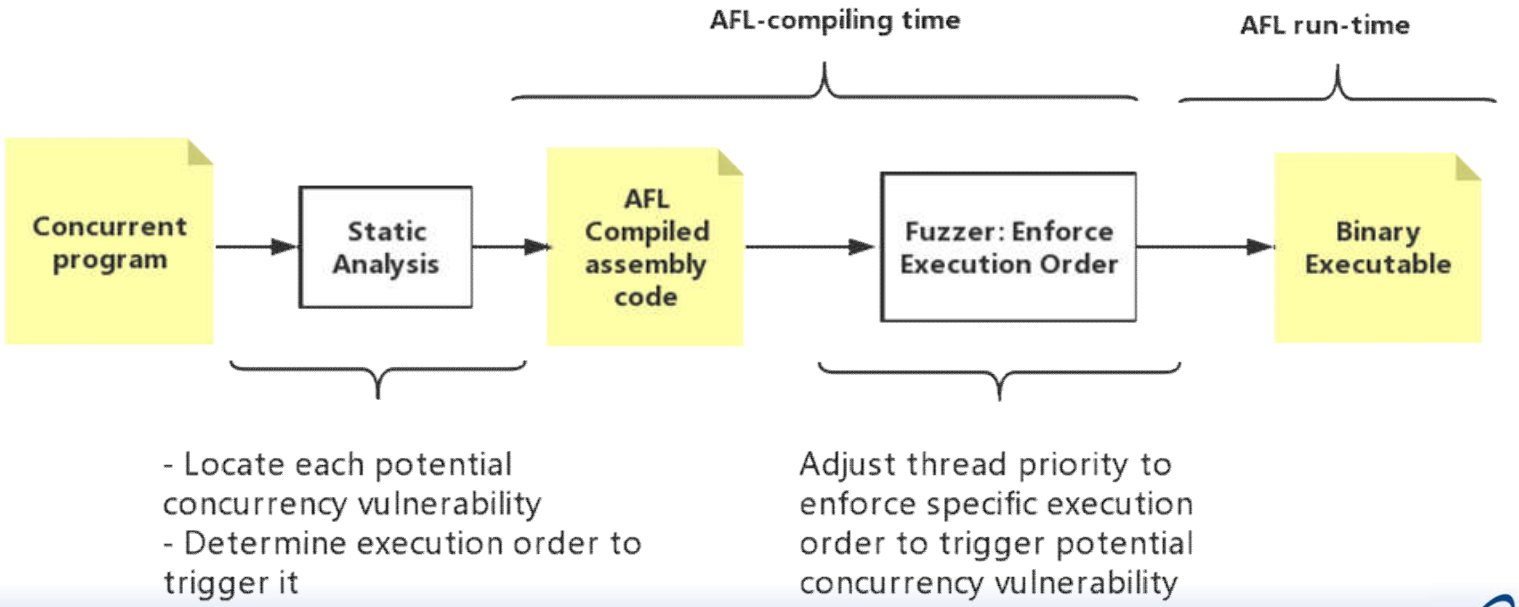
2. 专注某个特定的线程调度，通过反复测试来触发静态分析发现的并发漏洞
   - 并发漏洞检测方案

1. 线程调度探索—并发错误检测

- 模糊测试中的线程级别的优先级调度
  - 通过动态调整线程优先级来探索更多的线程调度
  - 和传统的读/写级别的调度相比，开销更小，但同样有效
- 探索尽可能多的线程调度
  - 每种线程调度只会执行特定次数，之后会产生新的未测试的线程调度进行测试

2. 并发漏洞检测

- 静态分析定位到某个特定的并发漏洞，确定其线程调度
- 通过在模糊测试中固定这个线程调度方式来增加触发漏洞的概率
- 针对4个主要并发漏洞的模式
  - buffer-overflow, use-after-free, use-after-null, double-free



部分实验结果

Application	LOC	*Exploring Priority*	Vulnerability Detected				Performance Overhead
		# of new crashes	Time	Vuln. type	# found by static analysis	# detected by targeted priority	
boundedbuf	0.4k	0	2.3s	Buffer Overflow	1	0	272%
swarm	2.2k	0	3.5s	Double-Free	1	0	109%
bzip2smp	6.3k	2	1500s	Double-Free	3	1	51%
pfscan	1.1k	1	1.2s	None	0	0	98%
ctrace	1.5k	0	2.9s	Double-Free	3	1	59%
qsort	0.7k	0	0.5s	Buffer Overflow	2	0	104%

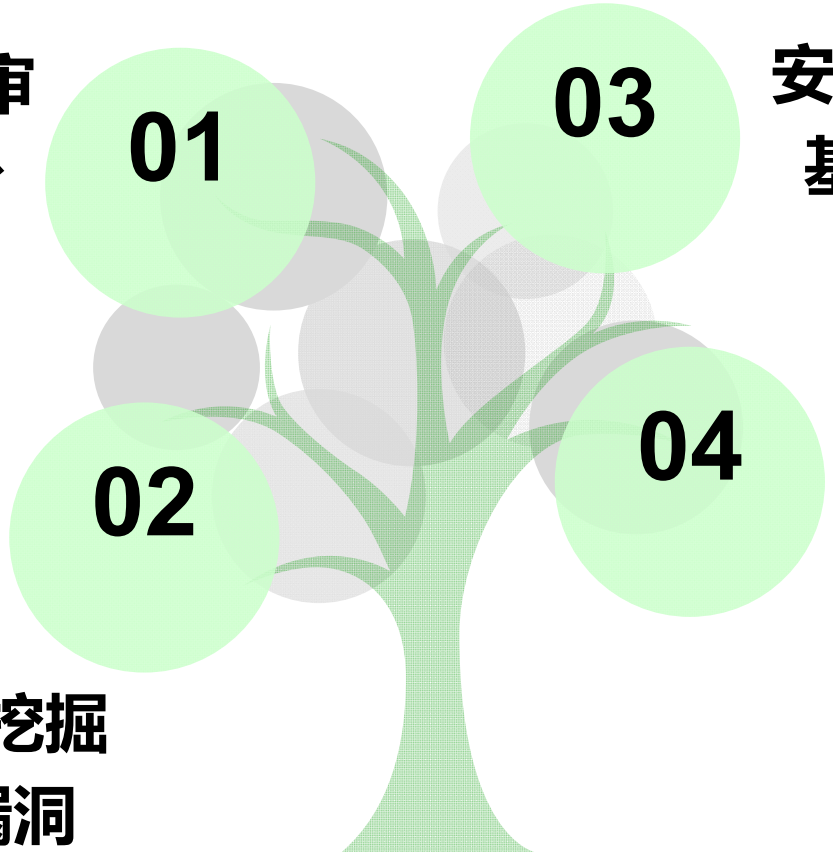
在检测并发错误方面，本方法通常10分钟左右会报一个错误，与此相比AFL则几天不会报一个错误

延伸到其他有价值的研究

主要涉及的方面

01 软件源代码安全审查公共服务平台

02 动态污点跟踪挖掘二进制程序漏洞



03 安全可控的电力信息基础设施风险评估

04 车联网安全

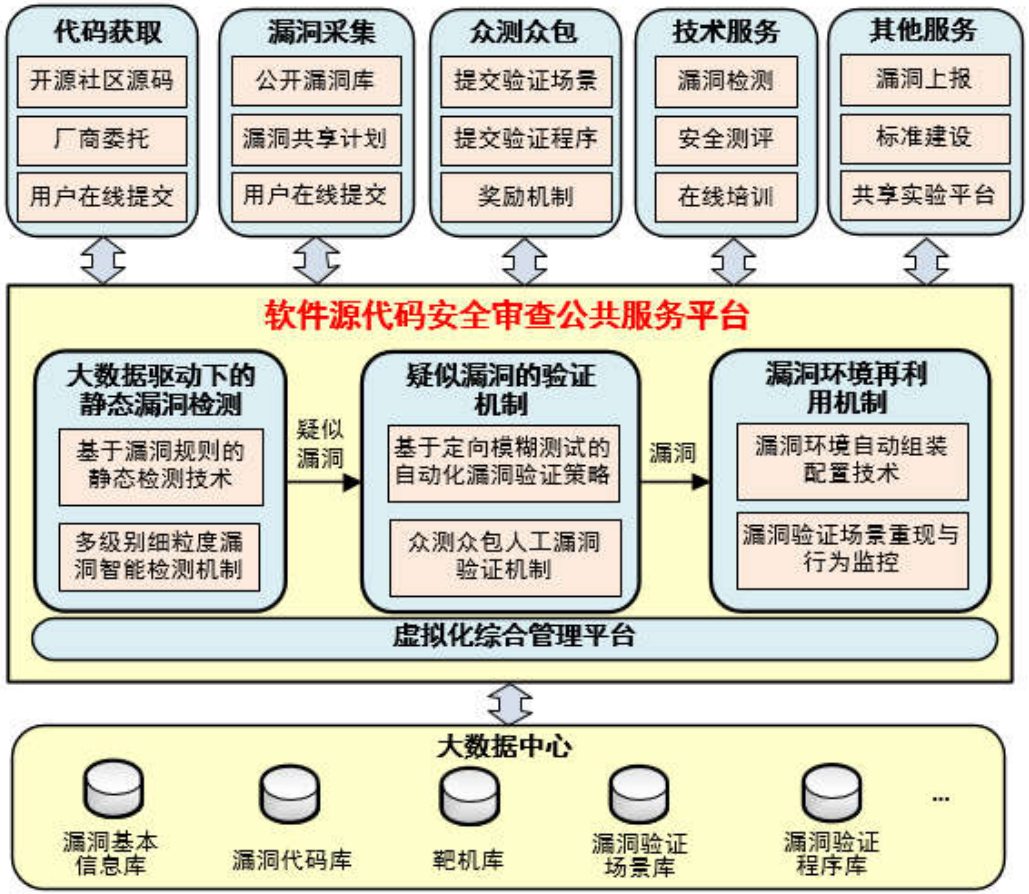
软件源代码安全审查公共服务平台

- 研究目标

构建国家级的软件源代码安全审查公共服务平台，为开源代码社区和企业提供安全审查服务

自行研发有效的软件代码漏洞检测与验证工具

打造专业的源代码安全实践基地，构建以高校为后盾的开源软件安全审查联盟



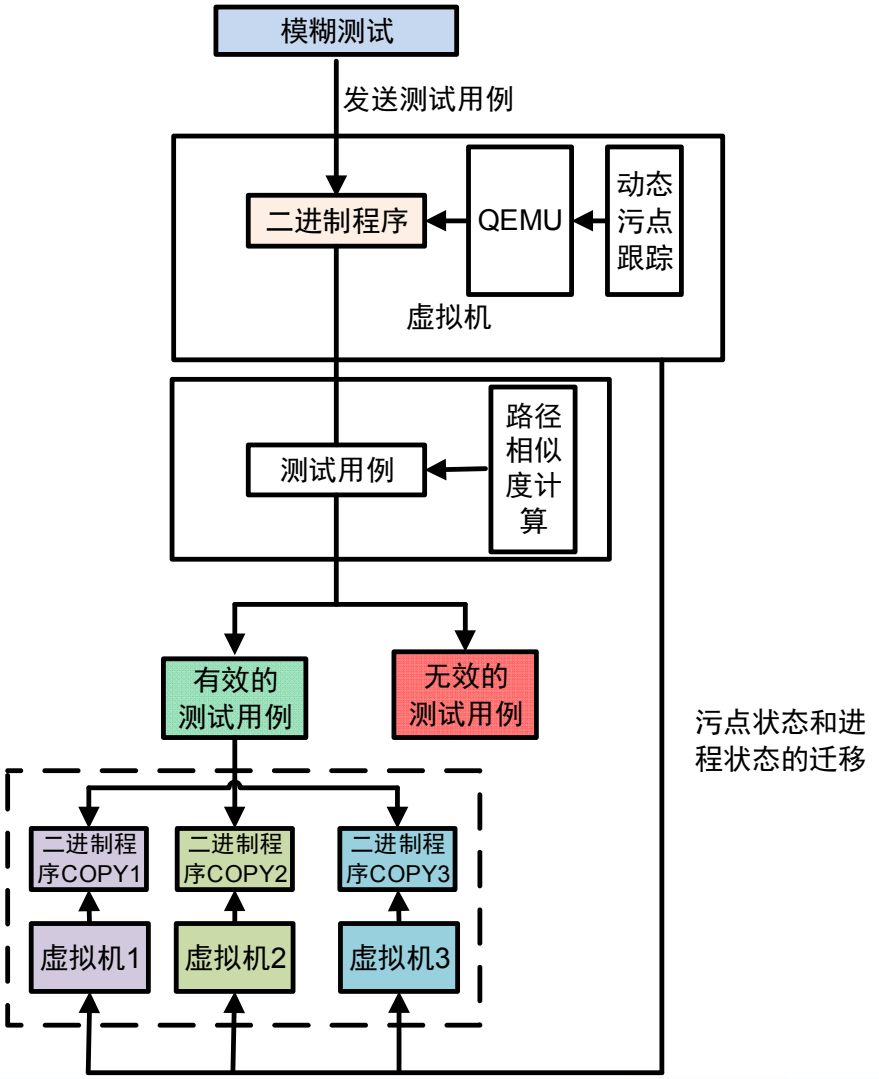
利用动态污点跟踪挖掘二进制程序漏洞

- 研究意义

对二进制程序进行动态污点分析是下一代系统安全的主要技术，这种技术在理论上能够检测已知和未知攻击，对提高主机系统安全水平具有重要意义

- 研究内容

围绕下一代系统安全中的动态污点分析技术，基于污点数据的污点传播对二进制程序的数据流进行细粒度分析，并辅助控制流分析和符号化执行，还原输入数据的高层语义结构，将数据流的访问特性和临界条件应用到二进制程序的漏洞挖掘中



谢谢！

deqingzou@hust.edu.cn