软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006971]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于局部路径图的自动化漏洞成因分析方法*

余媛萍[1,2], 苏璞睿[1,2], 贾相堃[1,2], 黄桦烽[1,2]

[1](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)

[2](中国科学院大学 计算机科学与技术学院, 北京 100049)

通信作者: 苏璞睿, E-mail: purui@iscas.ac.cn

摘　要: 快速的漏洞成因分析是漏洞修复中的关键一环, 也一直是学术界和工业界关注的热点. 现有基于大量测试样本执行记录进行统计特征分析的漏洞成因分析方法, 存在随机性噪声、重要逻辑关联指令缺失等问题, 其中根据测试集测量, 现有统计方法中的随机性噪声占比达到了 61% 以上. 针对上述问题, 提出一种基于局部路径图的漏洞成因分析方法, 其从执行路径中, 提取函数间调用图和函数内控制流转移图等漏洞关联信息. 并以此为基础筛除漏洞成因无关指令 (即噪声指令), 构建成因点逻辑关系并补充缺失的重要指令, 实现一个面向二进制软件的自动化漏洞成因分析系统 LGBRoot. 系统在 20 个公开的 CVE 内存破坏漏洞数据集上进行验证. 单个样本成因分析平均耗时 12.4 s, 实验数据表明, 系统可以自动剔除 56.2% 噪声指令和补充并联结 20 个可视化漏洞成因相关点指令间的逻辑结构, 加快分析人员的漏洞分析速度.

关键词: 漏洞分析; 成因分析; 函数间调用图; 函数内控制流转移图; 统计分析

中图法分类号: TP311

中文引用格式: 余媛萍, 苏璞睿, 贾相堃, 黄桦烽. 基于局部路径图的自动化漏洞成因分析方法. 软件学报. http://www.jos.org.cn/1000-9825/6971.htm

英文引用格式: Yu YP, Su PR, Jia XK, Huang HF. LGBRoot: Local Graph-based Automated Vulnerability Root Cause Analysis. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6971.htm

LGBRoot: Local Graph-based Automated Vulnerability Root Cause Analysis

YU Yuan-Ping[1,2], SU Pu-Rui[1,2], JIA Xiang-Kun[1,2], HUANG Hua-Feng[1,2]

[1](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Fast vulnerability root cause analysis is crucial for patching vulnerabilities and has always been a hotspot in academia and industry. The existing vulnerability root cause analysis methods based on the statistical feature analysis of a large number of test sample execution records have problems such as random noise and missing important logical correlation instructions. According to the test set measurement in this study, the proportion of random noise in the existing statistical methods reaches more than 61%. To solve the above problems, this study proposes a vulnerability root cause analysis method based on the local path graph, which extracts vulnerability-related information such as the inter-function call graph and intra-function control flow transfer graph from the execution paths. The local path graph is utilized for eliminating irrelevant instruction (i.e., noise instructions) elimination, constructing the logic relations for vulnerability root cause relevant points, and adding missing critical instructions. An automated root cause analysis system for binary software, LGBRoot, has been implemented. The effectiveness of the system has been evaluated on a dataset of 20 public CVE memory corruption vulnerabilities. The average time for single-sample root cause analysis is 12.4 seconds. The experimental data show that the system can automatically eliminate 56.2% of noise instructions, and mend as well as visualize the 20 logical structures of vulnerability root cause relevant points, speeding up the vulnerability analysis of analysts.

Key words: vulnerability analysis; root causes analysis; inter-function call graph; intra-function control flow graph; statistical analysis

---

* 基金项目: 国家自然科学基金 (62232016, 62102406, 61902384); 中国科学院战略性先导科技专项 (XDC02020300); 前沿科技创新专项 (2019QY1403)

收稿时间: 2022-05-30; 修改时间: 2022-11-03, 2023-03-15; 采用时间: 2023-04-19; jos 在线出版时间: 2023-10-18

快速的漏洞修复是软件漏洞防治环节中的关键一环, 而全面、准确的漏洞成因分析是快速确定漏洞修复方案的基础. 在现实场景中, 由于软件规模庞大, 逻辑复杂, 漏洞模式多样化, 软件漏洞的成因分析工作仍主要以人工分析为主, 分析效率低、能力差, 严重制约了软件漏洞发现后的快速响应能力. 探索自动化的漏洞成因分析方法是解决这一问题的重要思路, 也是当前研究的热点问题.

自动化的漏洞成因分析主要是辅助分析人员快速确定漏洞的错误代码位置, 以及错误代码与造成影响的逻辑关系, 以帮助分析人员快速理解程序错误原因, 确定修复方案. 当前自动化漏洞成因分析技术思路主要分为两大类, 即基于程序依赖关系分析的分析方法和基于执行记录对比的分析方法. 其中, 依赖关系分析是通过分析程序的控制依赖或者数据依赖关系提取与程序漏洞发生相关的指令操作, 代表性成果包括 RDDG[1] 、TaintCheck[2], 该方法主要面临的问题是依赖分析技术例如程序切片、符号执行存在固有的分析精度不高、分析路径爆炸等问题, 无法适应大规模样本量和复杂软件的分析需求; 执行记录对比通过直接比较正常执行和错误执行的指令差异区分出与漏洞成因相关的指令[3], 该方法能更好地适应软件规模、复杂度的需求, 但其面临程序中大量随机性因素带来的干扰问题 (噪声), 即软件正常执行过程中, 也会引入一些执行记录的差异. 为了解决噪声问题, 研究者提出了利用模糊测试技术建立测试集, 用于触发大量的执行记录, 通过对大量执行记录的统计分析手段来进一步确定关键差异、消除噪声, 其核心思想是运用测试集上正常与异常执行记录的统计特征信息对比, 建立程序特定指令与漏洞成因的关联性, 这方面代表性成果包括 AURORA[4], VulnLoc[5].

基于大量执行记录的统计分析方法对漏洞关联错误指令的筛选具有重要帮助, 但其仍然面临以下问题: 1) 随机性因素带来的噪声问题仍未有效解决. 统计分析方法将执行记录以“文本”方式处理, 根据执行信息的统计特征判定与漏洞是否相关, 实际执行记录中会有大量由于程序本身执行随机性带来的指令, 满足统计特征差异, 从而产生大量随机性噪声. 据本文数据集评估, 筛选的分析结果依然存在高达 61% 的随机性噪声指令, 严重干扰漏洞分析; 2) “平凡指令”缺失, 造成漏洞逻辑难以理解. 由于统计分析方法根据统计特征的差异筛选与漏洞成因相关的指令, 实际有些跟漏洞逻辑相关的重要指令在正常执行中也会频繁的出现, 按照现有的统计方法这些“平凡”指令会遗漏, 直接造成漏洞成因逻辑不完整而难以理解.

漏洞的成因是与软件自身的功能逻辑息息相关的, 其与程序执行的上下文环境联系紧密, 因此漏洞成因分析无法脱离程序功能逻辑. 程序功能逻辑在代码结构上就是各种代码基本块的组合, 具体包括函数内的分支跳转关系和函数间调用关系等; 同时, 漏洞触发时的程序状态一般表现为读或者写访问异常, 对内存的分配、释放等操作也是与程序异常紧密相关的行为. 因此, 在成因分析中结合软件的功能逻辑等上下文信息将有助于解决分析中的噪声等问题, 提升成因分析的准确性.

本文结合局部路径中的功能语义信息和逻辑结构关联信息实现漏洞成因关联指令的自动识别. 首先, 该方法通过从执行记录中收集漏洞相关指令的上下文信息, 包括函数间调用图 (inter-function call graph, InterCG) 和函数内控制流转移图 (intra-function control flow graph, IntraCFG) 关联, 建立局部路径图 (partial graph, PartG); 然后, 提出了基于局部路径图的噪声筛除方法, 该方法利用局部路径图的依赖关系识别与漏洞上下文环境有关的指令, 从而有效删除了漏洞成因噪声指令; 最后, 提出了基于局部路径图的漏洞成因点逻辑关系恢复方法, 补充与漏洞相关的缺失代码指令, 并联结漏洞成因相关点的逻辑结构. 最终, 形成了一个基于局部路径图的自动化漏洞成因分析方案, 不仅有效消除了随机性造成的噪声, 还可提供漏洞成因的可视化逻辑图, 对辅助漏洞成因分析具有重要帮助.

本文的贡献包括: 1) 指出了基于统计分析的漏洞成因分析方法存在噪声干扰, 并对现有方法进行了评估测量, 在实验数据集中, 存在高达 61% 的噪声; 2) 提出了基于局部路径图恢复的成因噪声筛除方法, 删除了统计分析结果中 56.2% 的函数间和函数内的成因噪声; 3) 提出了漏洞成因相关点逻辑关系的联结方法, 对数据集上 20 个漏洞形成了漏洞成因路径的可视化图; 4) 开发了一个面向二进制的自动化漏洞成因分析系统 LGBRoot, 并在 20 个 CVE 漏洞组成的数据集上验证了系统的效果. 该系统在平均开销 12.4 s 情况下, 自动剔除 56.2% 的噪声指令和联结 20 个可视化漏洞成因相关点组成的逻辑结构图, 加快分析人员的漏洞分析速度.

本文第 1 节介绍自动化漏洞成因分析方法的研究现状. 第 2 节引入样例分析介绍目前方法存在的问题, 并提出论文解决思路. 第 3 节介绍本文基于局部路径图的自动漏洞成因分析方法. 第 4 节通过实验验证了方法的有效

性. 最后讨论并总结全文.

1 研究现状

本文所提的方法主要解决漏洞成因统计分析中的问题, 下面就研究现状予以介绍.

漏洞成因分析主要用于指导漏洞理解和修复, 是软件漏洞分析中的关键需求. 漏洞隐藏于大规模程序代码中, 从代码的角度缺少明显异常特征, 仅在注入特定输入的情况下会触发软件执行异常[6], 漏洞成因分析需要从程序执行路径上的大量指令中, 确定与漏洞形成直接相关的指令, 即漏洞的成因, 以快速分析漏洞并确定修复方案. 早期的漏洞成因定位主要是依靠人工分析, 包括程序日志记录、断言、断点和运行时分析[7]. 然而, 这些方法严重依赖专家经验且效率低下不适用于大规模、复杂软件的分析[8].

自动化漏洞成因分析是探索自动化确定漏洞成因相关指令的过程, 可以分为基于程序依赖关系的分析方法和基于执行记录对比的分析方法. 基于程序依赖关系分析方法的基本思想是: 通过对错误对象与成因语句之间的依赖关系分析, 比如控制流、数据流等分析, 确定漏洞相关指令的范围. 常用的依赖分析技术有污点分析, 程序切片、符号执行等[9]. 其典型的分析流程如下: 首先, 基于漏洞验证输入 (POC), 运用指令插桩工具得到程序执行记录 (trace); 然后, 选定关键数据标记为污点 (如被破坏的指针等) 进行污点传播分析; 最后, 运用污点传播分析结果, 从执行指令记录中确定与污点相关的指令及变量值的修改关系, 从而确定漏洞成因[10]. 该方法理论上可以准确地分析漏洞成因[11], 但是实际分析效果受限于污点传播、程序切片、符号执行等固有的精度不高、路径爆炸等问题, 无法适应大规模漏洞软件样本量分析需求.

执行记录对比分析是另一种漏洞成因分析方法. 该方法基本思想是利用异常执行与正常执行的差异来识别漏洞成因相关指令. 其最基本的假设是触发漏洞的异常执行路径中存在与正常路径明显的代码差异, 这些差异性代码就是漏洞的成因. 然而, 在实际场景中, 由于系统的随机性干扰等问题, 正常执行路径中也存在大量的执行代码差异, 仅比较单条正常/异常执行记录会有大量的噪声干扰. 针对该问题, Liblit05[12] 、SOBER[13] 和 Hu 等人[14] 提出了基于测试集样本触发的多条执行记录进行统计分析的噪声消除方法. 其核心思想是漏洞触发路径中的差异性指令与正常路径中的指令有明显的统计特征差异, 其不同于一般性的系统随机性引入的一些指令差异. Zhang 等人[15] 运用该思想实现了一种针对 Java 虚拟机的控制流信息统计分析漏洞成因识别方案. 统计分析技术效果依赖于测试集, 而 CVE 漏洞在发布时仅包含一个触发漏洞的 POC, 传统手工构建或从网上爬取样本构建测试集的方法费时费力. 最近, AURORA[4] 、VulnLoc[5] 提出运用模糊测试技术优化了用于统计分析的测试集的产生. 其典型的分析流程如下: 首先, 利用模糊测试等方法, 构建大量与待分析漏洞 POC 相关的正常执行样本和触发异常执行的样本集合; 然后, 分别获取集合中样本的执行信息, 并在程序执行信息中构建指令谓语 (predicate, 例如指令操作、条件跳转等组成的布尔表达式信息); 最后, 利用各条谓语在测试集数据上的统计特征规律, 比如重要度分数, 定位与漏洞相关的谓语及其关联指令或者代码, 按照顺序生成成因指令的统计分析报告 (例如 AURORA 重要度分数阈值设置为 0.9). 基于统计分析方法的漏洞成因分析技术不依赖于重量级程序分析, 具有更好的适用性, 本文采用基于统计分析的漏洞成因分析方法并解决当前方法的局限性.

基于统计分析的漏洞成因分析方法较好地解决了大规模软件成因分析中的性能瓶颈问题, 但其在准确性方面仍难于满足现实需求, 其面临的问题主要包括以下两个方面.

1) 噪声问题仍未有效消除, 严重干扰成因的分析和确定. 统计分析主要根据正常执行和异常执行集合中指令出现的频率差异对比等方式去掉频率差异性特别小的指令, 但这一原则并不能作为判断漏洞引入的指令差异和随机性引入的指令差异的依据. 在实际的分析过程中, 大量软件运行随机性引入的指令差异与漏洞引入的指令差异在统计特性上非常的相近. 以本文实验数据统计为例, 按照现有的统计分析方法得到的漏洞成因指令中, 经过最后的分析确认, 仍有 61% 的噪声指令.

2) “平凡指令”缺失, 造成漏洞成因逻辑不完整, 理解困难. 在漏洞成因的完整代码逻辑中, 均少不了会出现一些“平凡指令”, 即在正常程序运行中的广泛使用的控制流 (过程间/过程内) 转移指令, 常见的有 call、jmp、jnz、

jz 等指令, 基于统计分析的基本思想可知, 这些指令不会被识别为异常指令, 即不被认为是与漏洞成因相关的指令. 这样就造成最后识别的漏洞成因指令是零散的, 逻辑是不完整的, 直接影响对漏洞成因的理解和分析.

2 样例分析及论文思路

本节以漏洞样例分析介绍基于统计分析的漏洞成因分析方法存在的问题, 并提出本文的解决思路.

2.1 样例分析

本文以图片编解码工具 FFJPEG 的空指针解引用漏洞 CVE-2019-16351 为例, 漏洞代码片段如图 1(a) 所示. 工具的执行功能流程包括图片加载、图片对象存储空间分配及赋值和图片解码. 此处的空指针解引用漏洞 (漏洞执行路径代码行号为 24-223-232-25-600-601-602-603-604-625-371) 成因是: 图片加载函数 jfif_load() 在加载图片时, 没有给图片对象 jfif 的结构体成员 jfif->phcac 正确的分配空间和赋初始值 (即正确分配赋初始值时 232 行的 if(fac) 分支条件被满足), 后续 jfif_decode() 函数对图片进行哈夫曼解码 (即第 625 行执行并调用 huffman_decode_step() 函数) 引用该结构体对象的操作出现错误. 漏洞修复补丁在 huffman.c 的第 371 行增加被错误解引用的结构指针的有效性判断.

在利用统计分析系统 (以 AURORA 为例) 对该漏洞分析后得到成因报告, 例如重要度阈值为 0.90 时报告指令条数 276 条, 部分数据如图 1(b) 所示, 对漏洞成因报告逐条进行手工分析, 并综合漏洞分析经验判断指令与漏洞成因的相关性, 发现: 1) 统计分析报告中存在大量分散在不同函数、不同位置的噪声指令 (例如 jfif.c 中 630、631 等); 2) 漏洞成因相关点零散分布在报告中, 且缺失部分对漏洞逻辑信息有重要关联作用的指令, 例如重要度阈值设置为 0.9 时, 缺失了阈值为 0.82 的 jfif_load() 漏洞成因函数以及阈值为 0.73 的 main() 函数中调用漏洞成因函数的入口点. 为了方便展示, 本文在图 1(b) 中标注了每条指令对应的源代码行数 (本文系统不依赖待分析程序源代码), 用“√”表示指令与成因相关, “×”表示指令与成因不相关.

```
//huffman.c
int huffman_decode_step(HUFCODEC *phc)
{
371:   if (!phc->input) return EOF; //patch
       while (1) {
375:     if(...)   bit = bitstr_getb(phc->input);
376:     if ( bit == EOF) return EOF;
379:     if (code - phc->first[len] < phc->huftab[len]) break;
} }
//jfif.c
int jfif_decode(JFIF *jfif)
{
                ...
600:  for (mcui=0; mcui<mcuc*mcur; mcui++) {
601:    for (c=0; c<jfif->comp_num; c++) {
602:      for (v=0; v<jfif->comp_info[c].samp_factor_v; v++) {
603:        for (h=0; h<jfif->comp_info[c].samp_factor_h; h++) {
604:          HUFCODEC *hcac = jfif->phcac[jfif->comp_info[c].htab_idx_ac];
 ...
624:          for (i=1; i<64; ) {
625:                code = huffman_decode_step(hcac);
                  if (code <= 0) break;
630:              code = bitstr_get_bits(bs  , size);
631:              code = category_decode(code, size);
                  if (i < 64) du[i++] = code;
          } } } } }
}
void* jfif_load(){
       JFIF *jfif  = NULL;
223:   case 0xc4: // DHT
         while (size > 0 && dht + 17 < end) {
           int idx = dht[0] & 0x0f;
232:       if (fac) {
233:         if (!jfif->phcac[idx]) jfif->phcac[idx] = calloc(1, sizeof(HUFCODEC));
                ...
           } }
       return jfif;
}
//ffjpeg.c
int main(int argc, char *argv[])
{
24:   jfif = jfif_load(argv[2]);
25:   jfif_decode(jfif);
}
```

(a) CVE-2019-16351 样例关键代码

```
× 0x00006284 -- 0x006284 edge_only_taken_to 0x006b3f -- 1 -- bnd jmp qword ptr [rip+0x9d3d]
× 0x000061b4 -- 0x0061b4 edge_only_taken_to 0x007429 -- 1 -- bnd jmp qword ptr [rip+0x9da5]
√ 0x0000c9e4 -- rax min_reg_val_less 0xff -- 0.992 -- mov rax, qword ptr [rbp-0x18] //huffman.c:371
√ 0x00008262 -- memory_value min_reg_val_less 0xff -- 0.977 -- mov qword ptr [rbp-0x1e8], rax //jfif.c:604
√ 0x00008658 -- max_sign_flag_set -- 0.950 -- cmp dword ptr [rbp-0x278], eax //jfif.c:602
√ 0x00008218 -- is_visited -- 0.950 -- mov dword ptr [rbp-0x27c], 0x0 //jfif.c:603
√ 0x0000823a -- max_interrupt_flag_set -- 0.950 -- add rax, rax //jfif.c:604
× 0x000082f0 -- rax min_reg_val_less 0xffffffff -- 0.950 -- mov rax, qword ptr [rbp-0x1e0] //jfif.c:611
...
√ 0x0000839f -- rax min_reg_val_less 0x0055575288 -- 0.925 -- mov rax, qword ptr [rbp-0x1e8] //jfif.c:625
√ 0x00008209 -- is_visited -- 0.922 -- mov dword ptr [rbp-0x278], 0x0 //jfif.c:602
√ 0x00008630 -- rcx min_reg_val_less 0xffffffff -- 0.922 -- mov rcx, qword ptr [rbp-0x208] //jfif.c:602
× 0x0000838a -- memory_value max_reg_val_less 0xffffffff -- 0.919 -- mov dword ptr [rbp-0x110], eax //jfif.c:620
√ 0x0000c9ef -- max_interrupt_flag_set -- 0.918 -- test rax, rax //huffman.c:371
× 0x0000c9fe -- rax min_reg_val_less 0xffffffff -- 0.918 -- mov rax, qword ptr [rbp-0x18] //huffman.c:375
× 0x0000a6e7 -- is_visited -- 0.918 -- nop edx, edi //bitstr.c:230
× 0x0000a6f7 -- is_visited -- 0.918 -- mov dword ptr [rbp-0x10], 0x0 //bitstr.c:231
× 0x00008302 -- memory_value min_reg_val_less 0xff -- 0.918 -- mov dword ptr [rbp-0x238], eax //jfif.c:611
× 0x0000ca0c -- 0x00ca0c edge_only_taken_to 0x00a6e7 -- 0.918 -- call 0x00a6e7 //huffman.c:375
× 0x0000a7d2 -- 0x00a7d2 edge_only_taken_to 0x00a7d5 -- 0.918 -- mov eax, dword ptr [rbp-0xc] //bitstr.c:254
× 0x0000a756 -- num_successors_greater 0 -- 0.918 -- jz 0x00a766 //bitstr.c:240
√ 0x0000839a -- 0x00839a edge_only_taken_to 0x00844d -- 0.917 -- jmp 0x00844d //jfif.c:624
√ 0x000083a9 -- 0x0083a9 edge_only_taken_to 0x00c9bf -- 0.917 -- call 0x00c9bf //jfif.c:625
× 0x0000a797 -- num_successors_greater 0 -- 0.917 -- jnz 0x00a7a0 //bitstr.c:246
√ 0x0000c9f2 -- 0x00c9f2 edge_only_taken_to 0x00c9fe -- 0.917 -- jnz 0x00c9fe //huffman.c:371
× 0x0000a092 -- num_successors_greater 0 -- 0.918 -- jl 0x00a09b //bitstr.c:43
× 0x0000ca18 -- num_successors_greater 0 -- 0.918-- jnz 0x00ca24 //huffman.c:376
√ 0x000081fa -- is_visited -- 0.904 -- mov dword ptr [rbp-0x280], 0x0 //jfif.c:601
√ 0x00008204 -- 0x008204 edge_only_taken_to 0x00866b -- 0.904 -- jmp 0x00866b //jfif.c:601
× 0x000069b0 -- rdx min_reg_val_greater_or_equal 0x1e -- 0.896 -- mov edx, dword ptr [rbp-0x4] //jfif.c:122
× 0x00009ff6 -- max_interrupt_flag_set -- 0.874 -- mov qword ptr [rbp-0x8], rax //bitstr.c:24
× 0x00006fbc -- max_parity_flag_set -- 0.874 -- mov qword ptr [rbp-0x18], rax //jfif.c:202
√ 0x0000868b -- rax min_reg_val_greater_or_equal 0x1 -- 0.871 -- mov eax, dword ptr [rbp-0x254] //jfif.c:600
× 0x0000a79e -- 0x00a79e edge_only_taken_to 0x00a7d5 -- 0.870 -- jmp 0x00a7d5 //bitstr.c:247
× 0x00006 9ad -- memory_value max_reg_val_greater_or_equal 0xc -- 0.870 -- mov dword ptr [rbp-0x8], esi //jfif.c:120
× 0x0000ca1a -- is_visited -- 0.857 -- mov eax, 0xffffffff //huffman.c:376
× 0x0000ca1f -- 0x00ca1f edge_only_taken_to 0x00cab9 -- 0.857 -- jmp 0x00cab9 //huffman.c:376
× 0x0000ca2d -- rax min_reg_val_less 0xffffffff -- 0.854 -- mov rax, qword ptr [rbp-0x18] //huffman.c:379
× 0x0000ca34 -- rdx min_reg_val_less 0xff -- 0.854 -- movsxd rdx, edx //huffman.c:379
√ 0x000071f9 -- rdx max_reg_val_less 0x1 -- 0.824 -- mov edx, dword ptr [rbp-0x40] //jfif.c:233
...
√ 0x0000642e -- min_reg_val_greater_or_equal 0x2aab0950 -- 0.727 -- mov qword ptr [rbp-0x28], rax //ffjpeg.c:24
√ 0x00006436 -- rax min_reg_val_greater_or_equal 0x2aab0950 -- 0.727 -- mov rax, qword ptr [rbp-0x28] //ffjpeg.c:25
```

(b) CVE-2019-16351 漏洞成因分析报告

图 1 CVE-2019-16351 样例分析

2.2 论文思路

漏洞的成因与程序执行的功能逻辑联系紧密, 由于漏洞发生涉及复杂操作, 而功能多样性的软件漏洞样本的

差异性相对比较大, 因此目前相关研究工作难以找到完备的方法来分析所有漏洞情况. 现有基于统计分析的漏洞分析方法根据正常与异常执行记录信息的统计特征差异性识别漏洞成因相关指令, 存在大量程序执行随机性引入的噪声, 并缺失重要逻辑关联指令. 程序执行的功能逻辑对于漏洞成因分析中去除随机性噪声和联结成因功能逻辑至关重要, 但是, 基于全面的程序功能逻辑分析来消除噪声又会引入很高的分析成本. 如何针对漏洞成因的上下文局部环境构建轻量级的逻辑关联结构信息, 并以此为基础, 消除噪声, 完善成因逻辑是本文的主要解决思路.

本文是在统计分析结果基础上, 首先从执行记录中收集漏洞相关指令的上下文信息, 主要包括函数间调用图和函数内控制流转移图, 构建漏洞相关的局部路径图; 然后, 基于局部路径图筛除漏洞相关范围以外的无关噪声指令; 最后, 基于局部路径图补充遗漏的成因指令, 并联结漏洞成因相关点之间的逻辑关联关系. 本文中, 局部路径图主要包括漏洞触发时造成程序读或者写访问异常的代码逻辑信息, 以及对内存的分配、释放等与程序异常紧密相关的程序操作行为的代码逻辑信息, 是包含整体程序代码逻辑信息的程序执行路径图的子图. 以样例程序为例, 程序代码执行逻辑信息包括整体函数间调用 (例如 main->bmp_save, main->jfif_decode 等) 和函数内控制流转移 (例如 huffman_decode_step() 函数内基本块分支跳转) 关系组成的程序执行路径图, 如图 2 所示; 样例程序的局部路径图是与漏洞成因相关的图片加载、图片对象空间分配及赋值和图片解码等操作相关的代码逻辑信息. 具体而言, 样例程序的局部路径图包括从 main() 函数调用 jfif_load() 函数加载图片; 然后, 执行函数内基本块跳转操作 (例如, 在加载函数 jfif_load() 内, 通过第 223 行的 case 基本块分支, 跳转进入的第 232 行 if 判断基本块分支, 决定图片对象空间分配及赋值操作等); 最后, main() 函数调用解码函数 jfif_decode(), 从而调用图片解析函数 huffman_decode_step() 出错. 为方便对比说明和展示, 在图 2 的程序执行图中, 以浅色字体简化标识了样例程序的局部路径图的组成和边界范围.

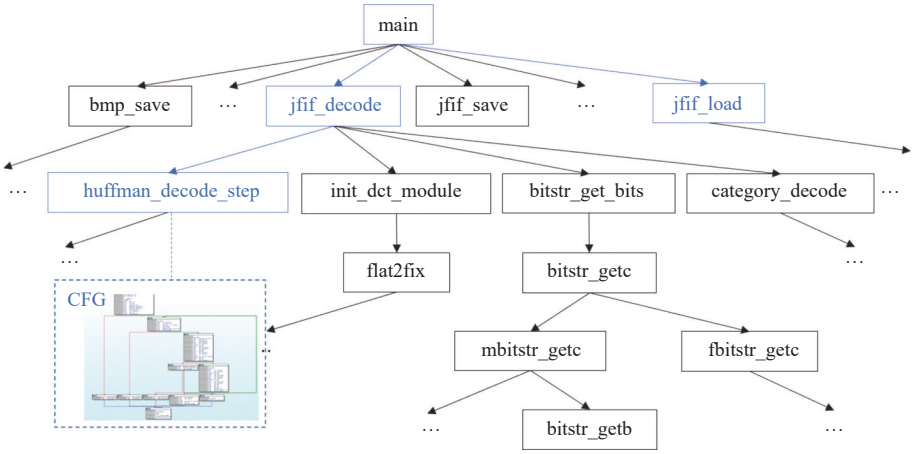

图 2 程序执行路径图

基于上述局部路径图中的信息, 可以自动化识别出统计分析结果中导致漏洞发生的指令, 从而筛除了与漏洞成因无关的指令, 即图 1 中的“×”表示的噪声指令; 利用局部路径图的逻辑结构信息, 可以将漏洞成因相关指令 (即图 1 中“√”表示的漏洞成因指令) 组成的逻辑结构补充完整 (例如修补阈值 0.9 时缺失的 jfif_load() 成因函数的调用关系), 并联结成后文图 3 所示的漏洞成因路径图 (vulnerability path graph, V). 因此, 从图中实例可以看出, 基于程序执行路径建立漏洞成因相关点组成的局部路径图, 可以筛除统计结果中多余噪声指令, 联结修复漏洞成因相关点之间的逻辑关联关系, 提高统计分析结果的准确性和漏洞成因逻辑的可理解性.

3 基于局部路径图的自动漏洞成因分析方法

鉴于漏洞成因统计分析技术不依赖于重量级程序分析和其对复杂软件的可拓展能力, 本文结合轻量级漏洞成因局部路径图和指令统计特征信息, 设计和实现了一个自动化漏洞成因分析系统 LGBRoot. 系统的流程是: 首先,

根据漏洞验证输入 POC 构造局部路径图, 并利用统计分析技术提取统计特征差异指令集合; 然后, 利用局部路径图剔除统计分析结果中无关噪声; 最后, 利用局部路径图补充统计成因结果中缺失的指令, 联结构造完整的漏洞成因逻辑报告, 总体系统框架如图 4 所示. 系统以待测程序 APP 和漏洞验证输入 POC 为输入, 最终输出漏洞成因上下文环境组成的成因路径图. 本文在系统的改进主要包括局部路径图建立、成因噪声筛除和逻辑关系联结 3 个部分, 见图 4 所示.

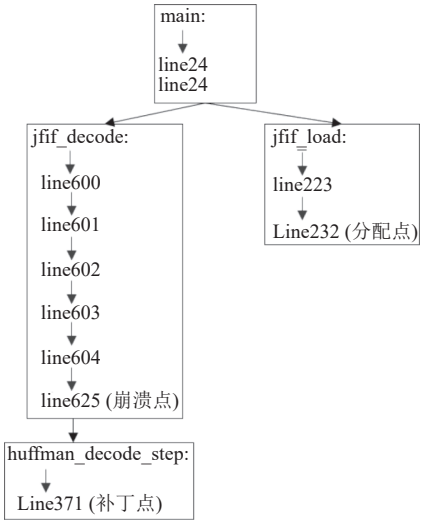

图 3 漏洞成因路径图

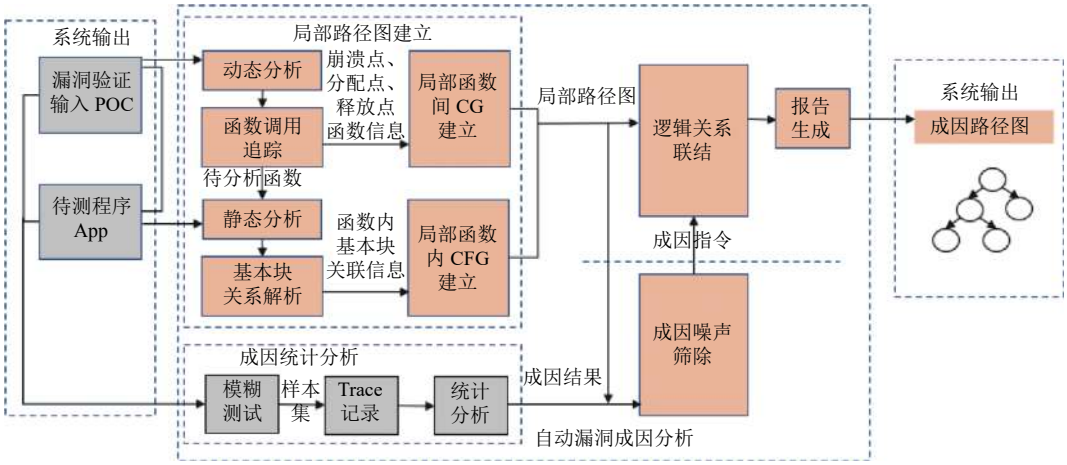

图 4 LGBRoot 系统框架图

首先, 根据漏洞验证 POC 从程序执行上下文环境中恢复漏洞成因相关点组成的局部路径, 即含有代码结构关联信息的函数间调用图和函数内控制流转移图, 建立局部路径图. 同时, 利用统计分析系统获取统计成因分析结果. 局部路径图中包含漏洞成因相关点的功能语义信息和逻辑结构关联信息, 可以用于统计分析结果中成因相关指令的自动识别. 一般而言, 程序执行的上下文环境是复杂的, 包含大量的函数调用关系和函数内基本块分支转移关系, 恢复程序完整上下文环境然后截取局部路径的方式是不可行的.

其次, 利用局部路径图的依赖关联信息, 自动化删除统计成因分析结果中与漏洞上下文环境无关的噪声点. 此处结合局部路径图上代码逻辑关联信息和漏洞成因统计分析技术提取的差异指令集合, 删除统计分析结果中不在

局部路径图结构范围中的随机性噪声指令, 并将剩余的漏洞成因指令输出到系统下一阶段.

最后, 利用局部路径图的结构关联信息, 补充统计成因分析结果中与漏洞相关的缺失指令, 并联结形成完整的漏洞成因分析报告. 对于结构化关联的局部路径图和零散的统计漏洞成因指令, 遍历联结统计结果中各条指令在图中涉及的逻辑关系, 并修补逻辑结构联结过程中新添加的指令 (例如用于联结函数调用的前必经结点), 将分析结果生成可视化的漏洞逻辑结构关联的成因路径图.

3.1 局部路径图的构建

局部路径图包含与程序异常状态紧密相关的程序功能逻辑信息, 在代码功能逻辑层面局部路径范围是与漏洞异常访问和内存的分配、释放等操作信息相关的函数调用和函数内基本块转移. 具体而言, 漏洞相关点信息是程序异常发生时的函数崩溃栈信息和内存分配和释放路径相关的函数调用列表信息, 以及上述信息涉及的各个函数的基本块分支跳转关系组成的集合.

定义 1. 局部路径图 (partial graph, PartG). 局部路径图是由漏洞成因相关点组成的局部函数间调用图和局部函数内控制流转移图组成.

定义 2. 局部函数间调用图 (inter-function call graph, InterCG). 漏洞程序 *Prog* 的局部函数间调用图 InterCG 是 (N, E) 组合, 其中, N 代表漏洞相关的所有函数节点集合, E 代表有向边集合, 而边 (n_i, n_j) 则表示从节点 n_i 到节点 n_j 的函数调用关系.

定义 3. 局部函数内控制流转移图 (intra-function control flow graph, IntraCFG). 漏洞程序 *Prog* 的局部函数内控制流转移图 IntraCFG 是 (N, E) 组合, 其中, N 代表漏洞函数内所有基本块结点集合, E 代表有向边集合, 而边 (n_i, n_j) 则表示从节点 n_i 到节点 n_j 的基本块跳转关系.

局部路径图 PartialGraph 的构建如算法 1 所示. 对于给定的漏洞程序 *Prog*, 漏洞触发样本 I_e, 首先, 从动态触发漏洞的程序执行 ExecutionTrace() 中获取函数调用关系 CallRetPair 和执行过的函数列表 FuncList; 然后, 通过对漏洞程序 *Prog* 反汇编 Dissgraph(), 获取函数列表中各个函数内的 CFG 图, 即基本块间转移关系; 最后, 通过对函数列表中各个函数名遍历, 得到由一一对应的函数名和函数 CFG 组成的集合 (FuncList, CFGraphs). 基于上述获取的信息, 本文对顺序执行的函数列表 FuncList 中的函数, 通过 IsCall() 和 IsRet() 匹配函数的调用和返回, 用 AddTo() 和 RemoveFrom() 维护调用栈 callstack 中函数调用的添加和删除, 并维护函数调用边集合 callbranch. 与此同时, 用 LabelAlloc() 函数标记对堆分配函数 (例如 malloc()、calloc() 和 realloc() 等) 有调用能力的函数列表 funcMlist, 以及用 LabelFree() 函数标记对堆释放函数 (例如 free() 等) 有调用能力的函数列表 funcFlist. 按照上述操作, 在漏洞触发造成程序异常结束时, callstack 中只有调用而无返回的剩余函数则认为是崩溃栈函数列表 backtrace; 列表 funcMlist 和 funcFlist 则分别是分配和释放路径相关的函数调用列表信息. 基于上述漏洞相关信息, 通过在集合 (FuncList, CFGraphs) 中取出崩溃栈函数列表 backtrace、分配函数列表 funcMlist 和释放函数列表 funcFlist 中函数所涉及函数内基本块转移图组成的列表 CFGList. 最终, 形成漏洞路径相关的包含崩溃栈路径, 对象分配、释放路径, 及其函数调用边, 函数内基本块跳转路径, 组成的局部路径图 PartialGraph.

算法 1. 构建局部路径图.

输入: 漏洞触发样本 I_e, 待测漏洞程序 *Prog*;
输出: 局部路径图 PartialGraph.

```
1. CallRetPair, FuncList←ExecutionTrace(Prog, I_e)
2. (FuncList, CFGraphs)←Dissgraph(Prog)
3. Function GetPartialGraph (CallRetPair, FuncList, (CFGraphs, Functions))
4.     for each func in FuncList do
5.         if IsCall(CallRetPair, func) then
6.             AddTo(callstack, func)
```

```
7.              AddTo(callbranch) /*函数调用边*/
8.          if IsRet(CallRetPair, func) then
9.              RemoveFrom(callstack, func) /*崩溃栈*/
10.         if IsAlloc(func) then
11.             LabelAlloc(funcMlist, callstack, func) /*分配栈*/
12.         if IsFree(func) then
13.             LabelFree(funcFlist, callstack, func) /*释放栈*/
14.   for each (function, CFG) in (FuncList , CFGraphs) do
15.         if function in (backtrace or funcMlist or funcFlist) then
16.             AddTo(CFGList, CFG) /*函数内 CFG*/
17.   PartialGraph←Graph(callbranch, backtrace, funcMlist, funcFlist, CFGList)
18. End Function
```

局部路径图可以用于漏洞成因分析出于两方面的考虑: 1) 漏洞成因指令的功能逻辑联系紧密, 漏洞关联指令在代码结构上分布范围集中; 2) 全局路径图代价高. 一方面, 漏洞触发时, 程序状态一般表现为读或者写访问异常, 对内存的分配、释放等操作也是与程序异常紧密相关的行为. 当程序执行被异常输入触发崩溃状态时, 发生错误的函数调用没有正常返回, 一般认为发生程序崩溃时的函数调用栈与漏洞直接相关. 除了触发程序崩溃路径上的函数调用, 漏洞成因可能也与漏洞对象的分配、释放路径的函数相关联, 例如堆溢出漏洞成因可能是错误的堆地址偏移访问, 也可能是堆空间分配异常. 因而本文在局部路径图中也会建立内存空间分配释放相关的路径, 用于方便分析复杂内存漏洞, 比如堆溢出, 释放后重引用 (UAF) 和空指针解引用 (null-pointer-deference) 等漏洞. 另一方面, 在程序代码逻辑中, 函数的调用涉及多个函数的调用与被调用关系, 静态分析缺乏函数实际调用信息无法准确画出函数调用图, 需要采用动态执行方式构建; 而动态构建整体函数调用图依然存在结点多、时间开销大的问题, 本文采取动态局部路径图恢复方法. 在动态构建图的过程中, 由于崩溃函数调用栈信息有可能被错误对象破坏而无法直接获取, 本文从动态执行的函数的调用和返回 (call-return 对) 信息中还原程序的函数崩溃栈, 加入局部路径图. 除了函数调用图, 函数内的基本块跳转也是执行路径图的一部分, 由于软件开发中的函数复杂度指导[16], 函数功能不会过于复杂, 因而本文通过静态分析提取漏洞相关的函数内 CFG 图, 包含基本块点和控制流转移边关联, 并加入局部路径图.

3.2 基于局部路径图的成因噪声筛除

结合局部路径图信息, 筛除统计成因分析报告中与漏洞无关的噪声指令. 基于统计分析的漏洞成因分析结果中存在大量程序执行随机性因素带来的噪声, 噪声指令零散、孤立 (无控制流关联) 分布在不同的函数或者基本块中. 局部路径图中引入了与漏洞发生相关的语义和逻辑关联信息, 可以识别出统计分析结果中存在于局部路径图上且不孤立存在的漏洞成因指令, 从而筛除与漏洞上下文环境无关联性的随机性噪声. 在实现上, 局部路径图从函数间和函数内两个维度去噪. 一方面, 在局部路径图上的函数间调用图 InterCG 中引入了漏洞相关的崩溃栈函数以及对出错对象具有分配、释放能力的函数信息, 筛除统计分析结果中不存在于局部路径图函数中的噪声指令. 另一方面, 在局部路径图上的函数内基本块分支跳转图 IntraCFG 中引入了结构化关联的基本块分支跳转关系, 筛除与其他指令无分支跳转关联性的噪声指令. 由于上述 InterCG 图中的函数与漏洞发生有关联 (即函数执行后存在程序正常执行和触发漏洞异常两种程序状态), 因此触发漏洞的样本在执行该函数时基本块跳转一般存在相似的关联性特征 (例如当漏洞点在嵌套的 if 条件判断语句中时, 执行到下层判断语句的条件时已经满足了上层 if 分支条件判断语句); 而统计分析结果来源于多条执行记录, 存在随机性孤立噪声指令 (也就是, 指令所在基本块与任何其他指令所在基本块无控制流关联的指令), 本文利用漏洞成因相关点存在分支跳转关联性, 筛除统计分析结果中在函数间 InterCG 上但孤立于函数内 IntraCFG 分支跳转关联结构的噪声指令.

基于局部路径图在函数间 InterCG 和函数内 IntraCFG 两个维度的去噪, 设计了漏洞成因噪声筛除方法如算

法 2 所示. 对于给定的统计成因分析指令集合 *Rr*, 局部路径图 PartialGraph, 统计阈值 θ, 一方面, 根据动态与静态程序基址偏移的对应关系, 通过函数 GetInstrAttribute($Instr_i$) 静态获取指令集合 *Rr* 中每条指令 $Instr_i$ 在程序中的基本信息, 包括指令所属函数名 Fname, 指令的重要度分数 IScore, 指令地址 IAddr 和所在基本块地址 BBAddr 等; 另一方面, 从局部路径图 PartialGraph 中获取结构关联信息, 即函数间和函数内的点和边列表, CGnodeList, CFGnodeList, CGedgeList, CFGedgeList, 用于去噪. 在函数间的去噪中, 筛除在统计阈值 θ (经验设置例如 AURORA 取值 θ=0.90) 之下以及指令所在函数 Fname 不在局部图 InterCG 的函数列表 CGnodeList 中的指令. 对于函数内去噪, 筛除指令所在基本块地址在控制流边 CFGedgeList 中无基本块分支边关联的指令. 因此, 基于局部路径图的函数间和函数内的上下文语义信息, 有效筛除了与漏洞成因不相关的随机性噪声指令, 完成去噪精简后的统计成因分析指令集合 *Rn* 的获取.

算法 2. 漏洞成因噪声筛除方法.

输入: 统计成因分析指令集合 *Rr*, 局部路径图 PartialGraph, 统计阈值 θ;
输出: 去噪精简后的统计成因分析指令集合 *Rn*.

```
1. Function TrimNoise (PartialGraph, Rr)
2.     for each node in PartialGraph do
3.         AddTo(CGnodeList, CFGnodeList, node)
4.     for each edge in PartialGraph do
5.         AddTo(CGedgeList, CFGedgeList, edge)
6.     for each Instr_i ∈ Rr do
7.         (FName, IScore, IAddr, BBAddr, Instr_i)←GetInstrAttribute(Instr_i)
8.         if IScore <θ or (FName not in CGnodeList) then /*函数间去噪*/
9.             continue
10.        else if (FName in CGnodeList) then
11.            dict[FName].append(BBAddr)
12.            dictIns[FName].append( Instr_i)
13.            for each func in dict.keys() do
14.                for each (bb0, bb1) in CFGedgeList do
15.                    if (bb0 in dict[func]) and (bb1 in dict[func]) then
16.                        Rn.add(dictIns[func]) /*函数内去噪*/
17. End Function
```

3.3 基于局部路径图的成因逻辑关系联结

利用局部路径图的结构关联特性, 恢复并联结统计成因分析报告中零散混乱的漏洞成因相关指令的逻辑关系. 统计分析结果中, 缺失了与漏洞统计特征无关, 但与漏洞执行功能逻辑结构有重要关联性的控制流转移指令. 局部路径图是包括函数间调用关系和函数内基本块转移边的连通性结构关联图, 可以区分图上指令执行先后顺序 (例如函数调用顺序) 和逻辑关联关系 (例如成因复杂的漏洞或是存在多处与成因相关指令的漏洞的执行逻辑), 从而识别和修复统计分析结果中缺失的用于联结各条指令逻辑的结点和边 (例如修补函数调用中前必经节点), 最终, 形成一个完整的漏洞关联逻辑.

基于局部路径图在函数间 InterCG 和函数内 IntraCFG 两个维度联结指令关系, 设计了漏洞成因逻辑关系联结方法如算法 3 所示. 对于去噪后的统计成因分析指令集合 *Rn*, 局部路径图 PartialGraph, 首先, 对指令集合 *Rn* 中的每条指令通过 GetInstrAttribute() 函数获取指令基本信息 (同第 3.2 节), 从 PartialGraph 分别获取函数间和函数内的点和边列表, CGnodeList, CFGnodeList, CGedgeList, CFGedgeList (同第 3.2 节), 用于联结指令关系. 然后, 在函

数间 InterCG 的逻辑关系联结中, 对于指令集合 *Rn* 中指令所在的函数通过 TraverBranch() 函数迭代遍历局部路径图函数调用边 CGedgeList, 联结至程序的入口函数 mainFunc, 并将迭代遍历中缺失的函数调用边添补至漏洞成因路径图 *V*. 最后, 对于函数内 IntraCFG 的指令基本块跳转逻辑关系联结, 将指令集合 *Rn* 中指令所在的基本块, 通过 TraverBranch() 函数迭代遍历局部路径图基本块跳转边列表 CFGedgeList, 至函数的入口地址 funcStartbb, 将存在于指令集合 *Rn* 中的指令涉及的基本块跳转边加入漏洞成因路径图 *V*, 对于联结中缺失的基本块跳转边, 由于基本块粒度过细 (见实验部分基本块粒度修补的评估), 此处不添补具体基本块粒度的关系进漏洞成因路径图 *V*, 而仅对缺失的联结边添补一条入边保持图的连通性. 最终, 通过画图工具 GraphPlot(), 将图 *V* 中的函数间和函数内的边可视化, 形成包含指令间逻辑关联关系的漏洞成因路径图.

算法 3. 漏洞成因逻辑关系联结方法.

输入: 去噪后的统计成因分析指令集合 *Rn*, 局部路径图 PartialGraph;
输出: 漏洞成因路径图 *V*.

```
1. Function UnionGraph (PartialGraph, Rn)
2.    for each node in PartialGraph do
3.          AddTo(CGnodeList, CFGnodeList, node)
4.    for each edge in PartialGraph do
5.          AddTo(CGedgeList, CFGedgeList, edge)
6.    for each Instr_i ∈ Rn do
7.          (FName, IAddr, BBAddr, Instr_i)←GetInstrAttribute(Instr_i)
8.          AddTo(FNameList, FName)
9.          AddTo(BBAddrList, BBAddr)
10.         for each (fb0, fb1) ∈ CGedgeList do /*函数间联结*/
11.               if IsFind(FNameList, fb1) and ((fb0, fb1) not in G) then
12.                     AddTo(G, (fb0, fb1))
13.                     While fb0 != mainFunc
14.                           (fbm, fb0)←TraverBranch(CGedgeList, fb0)
15.                           if (fbm, fb0) not in G then
16.                                 AddTo(G, (fbm, fb0))
17.                           fb0←fbm
18.         for each (bb0, bb1) ∈ CFGedgeList do /*函数内联结*/
19.               if IsFind(BBAddrList, bb1) then
20.                  While(bb0 != funcStartbb)
21.                        (bbm, bb0)←TraverBranch(CFGedgeList, bb0)
22.                        if IsFind(BBAddrList, bbm) then
23.                              AddTo(G, (bbm, bb0))
24.                        else
25.                              AddTo(G, (funcStartbb, bbm))
26.                              break
27.                       b0←bbm
28.   GraphPlot(G)
29. End Function
```

4 评估及实验分析

4.1 实验设置

漏洞类型种类繁多, 类似命令注入、SQL 注入、XSS 攻击等漏洞, 当漏洞触发点即漏洞成因点时, 成因分析难度相对低, 但作为高危害性漏洞之一的内存破坏类漏洞[17], 因其漏洞触发点与成因点的逻辑关系相对复杂, 分析难度大. 本文根据 CWE 的官方分类, 按照内存漏洞的成因特点, 收集有代表性的空间类和时间类漏洞组成数据集. 需要提到的是, 由于方法研究基于 POC 崩溃路径展开, 部分漏洞的触发类型会表现在其他漏洞类别上, 例如整数溢出, 内存泄露等漏洞最终会表现为堆溢出或者空指针解引用的漏洞类型, 因此文中未单列漏洞类型. 以整数溢出漏洞为例, 其发生时不会造成程序执行出错, 但在以溢出变量 (包括上溢和下溢) 作为地址偏移值计算堆内存的访问地址时, 造成堆溢出漏洞崩溃类型, 此类情况归为堆溢出漏洞.

实验数据集: 为了测试实验方法的有效性, 选择软件功能多样化 (包括: 图片、视音频处理软件, 汇编器, 反汇编器, JS、RUBY 语言框架, 样本分析软件和 ELF 文件格式处理软件等), 漏洞类型多样化 (包括典型的时序类内存漏洞 UAF (use-after-free)、DF (double-free)、 NPD (null-pointer-deference) 和空间类内存漏洞: 整数溢出 IO (integer-overflow)、缓冲区溢出 BO (buffer-overflow)、堆溢出 HBO (heap-buffer-overflow)、栈溢出 SO (stack-overflow)、未初始化 UV (uninitialised-value) 等类型). 漏洞发现和复现是复杂的过程, 据漏洞分析经验, 已公开披露的漏洞中只有少数漏洞能找到可复现的 POC 和补丁, 但为了方便验证漏洞成因定位的准确性, 选取用于测试集合的公开漏洞符合以下原则: 1) 漏洞 POC 可复现用于分析 2) 漏洞补丁可获取用于判定漏洞成因识别效果. 最终, 本文获取 20 个漏洞集作为实验数据集, 包括: wireshark、binutils 等百万行代码规模的复杂商用程序, 表 1 给出了数据集所对应的详细信息, 包括软件所在的程序包, 软件名称, 软件代码行数 LOC, 漏洞在本文测试集编号, 漏洞的公开 CVE 号, 漏洞类型以及软件功能描述.

表 1 实验数据集

程序包	软件名	LOC	No.	CVE号	漏洞类型	软件描述
GPAC	mp4box	85.4k	#1 #2 #3	CVE-2020-35980 CVE-2020-35979 CVE-2020-35981	UAF BO/HBO NPD	视音频处理软件
NASM	nasm	45.5k	#4	CVE-2018-19216	UAF	汇编器
MJS	mjs	36.9k	#5	CVE-2021-31875	UAF	JS 引擎
YARA	yara	42.8k	#6	CVE-2017-5924	UAF	恶意样本分析软件
MRUBY	mruby	43.6k	#7 #8	CVE-2018-10191 CVE-2018-10199	BO/HBO UAF	轻量级RUBY语言框架
LIBJPEG	djpeg	61.5k	#9	CVE-2018-19664	BO/HBO	JPEG 图像编解码器
BINUTILS	objdump	3 377k	#10	CVE-2017-14745	IO/SEGV	反汇编软件
	nm		#11	CVE-2017-15020	BO/HBO	库符号列表分析工具
	readelf		#12	CVE-2017-6965	BO/HBO	ELF文件格式分析工具
LIBTIFF	tiffcrop	85.5k	#13	CVE-2016-10093	BO/HBO	TIFF图像处理工具
	tiffcrop		#14	CVE-2016-10270	BO/HBO	
	tiffsplit		#15	CVE-2016-9273	BO/HBO	
JPEGOPTIM	jpegoptim	2.2k	#16	CVE-2018-11416	DF	图片压缩工具
NGIFLIB	git2tga	1.4k	#17	CVE-2019-20219	HBO	GIF图片格式解码工具
FFJPEG	ffjpeg	2.8k	#18	CVE-2019-16351	NPD	JPEG编码和解码工具
LIBMING	listmp3	95k	#19	CVE-2017-16898	IO/SO	视频文件处理库
WIRESHARK	tshark	3 621k	#20	Bug-15152	UV/BO	网络包分析工具

实验平台: 实验测试的主机是 64 位系统 Ubuntu LTS 16.04, 配置是 Intel(R) Xeon(R) CPU E5-2630, v3 处理器 (2.40 GHz, 32 核 32 GB).

实验效果评估标准: 漏洞的成因定位是复杂的, 可能涉及到多个不连续的代码行或者代码模块. 漏洞定位的目

标是提供尽可能多的正确信息帮助程序员理解和修复漏洞. 本文以补丁所在位置为漏洞成因识别成功的 ground truth 标准, 以 top-50 的假阳性指令数 (即噪声指令数量) 评估系统的识别效果.

4.2 系统实现

本文使用 Dynamorio 动态二进制代码分析框架追踪函数间调用信息, 使用 IDAPython[18] 静态分析函数内基本块跳转关系和统计分析报告中指令的基本信息, 用 Dot 工具完成漏洞成因路径图的可视化. 本文开发的漏洞分析系统 LGBRoot 共包含 1 450 行 Python 代码, 其中 970 行用于提取函数间局部 InterCG 图、去噪和联结功能, 480 行用于提取函数内局部 IntraCFG 图、去噪和联结功能. LGBRoot 系统不依赖于待分析程序的源代码, 为了方便与漏洞补丁做对比, 本文编译了源代码版本用于验证分析方法的有效性. 本文运用开源的统计分析系统 AURORA 作为原始漏洞成因指令的统计报告语料来源.

4.3 实验结果及分析

为了评估基于局部路径图的自动化漏洞成因分析系统 LGBRoot 的有效性, 本文研究了以下 2 个问题.

① RQ1: LGBRoot 能否比其他同类自动漏洞分析系统获得更好的效果?

② RQ2: LGBRoot 对于噪声筛除及成因逻辑联结效果如何?

● RQ1: LGBRoot 能否比其他同类漏洞自动分析系统获得更好的效果?

为了验证这个问题的结果, 本文将 LGBRoot 与基于统计分析方法的漏洞成因分析系统 AURORA 进行了对比实验, 在重要度阈值分别为 0.90 和 0.85 时, 从手工待确认指令数量、补丁点是否识别以及漏洞系统识别的 top-50 假阳性指令数量 3 个方面进行评估, 实验的结果见表 2.

表 2 LGBRoot 与自动化统计漏洞成因分析方法性能比较

No.	Total	AURORA						LGBRoot					
	θ=0.0	θ=0.90			θ=0.85			θ=0.90			θ=0.85		
	Ins	Ins	P	FP	Ins	P	FP	Ins	P	FP	Ins	P	FP
#1	19 728	744	N	—	2 062	Y	30	337	N	—	633	Y	3
#2	27 409	1 526	Y	20	7 690	Y	20	479	Y	4	2 750	Y	4
#3	14 165	1 966	Y	46	2 062	Y	46	870	Y	2	872	Y	2
#4	13 748	98	Y	27	169	Y	27	55	Y	0	69	Y	0
#5	10 715	23	N	—	133	Y	21	8	N	—	32	Y	0
#6	10 064	64	Y	15	281	Y	15	6	Y	0	6	Y	0
#7	40 219	852	Y	18	3 259	Y	18	372	Y	0	1 383	Y	0
#8	46 486	530	Y	24	1 518	Y	24	250	Y	6	458	Y	6
#9	755	13	N	—	231	Y	11	7	N	—	107	Y	5
#10	9 894	64	Y	7	263	Y	7	51	Y	0	225	Y	0
#11	16 113	95	Y	6	388	Y	6	84	Y	0	281	Y	0
#12	9 789	236	Y	11	525	Y	11	152	Y	0	222	Y	0
#13	2 966	20	Y	0	84	Y	0	19	Y	0	55	Y	0
#14	2 253	13	Y	0	163	Y	6	2	Y	0	105	Y	0
#15	831	0	N	—	295	Y	2	0	N	—	182	Y	0
#16	1 028	14	N	—	38	Y	11	1	N	—	24	Y	0
#17	1 035	4	N	—	9	Y	0	4	N	—	9	Y	0
#18	2 288	276	Y	3	406	Y	3	143	Y	0	255	Y	0
#19	166	0	N	—	3	Y	3	0	N	—	2	Y	0
#20	422	3	N	1	25	Y	2	3	N	0	23	Y	0
Sum	230 074	6 541	—	178	19 604	—	263	2 843	—	12	7 693	—	20

注: P表示补丁点是否识别; FP表示前50条指令中假阳性指令的数量

与目前效果最好的二进制自动化统计成因分析系统 AURORA 相比, 从表 2 中的数据可以看出本文的方法有着明显的优势. 在缩减手工待确认漏洞相关指令数 (Ins) 方面, LGBRoot 对测试集上所有样本 (20/20) 都能明显减

少漏洞分析人员的待分析指令数, 其中当正常执行与异常执行的指令统计特征差异阈值设置为 0.90 时 (θ=0.90) 指令总数相对于 AURORA 减少 2.3 倍 (6541/2843); 当阈值设置为 0.85 时 (θ=0.85) 指令总数减少 2.6 倍 (19604/7693), 虽然两个系统都从整体有差异 (θ=0.00) 的指令 (230074) 中明显缩减了指令量, 但 LGBRoot 更突出.

在漏洞补丁指令位置识别 (patch) 方面, LGBRoot 和 AURORA 在相同的统计阈值范围内能达到相同的补丁点指令识别效果. 然而, 当统计特征阈值设置为 0.85 时所有 (100%) 样本的补丁都可以识别, 而设为 0.90 时有 40% (8/20) 的样本不能具体识别到补丁指令, 此处可以看出阈值设置高虽然减少了需要人工检查的指令量 (即阈值 0.90 时平均手工分析指令数 2843/20=142.1 条而 0.85 时 7693/20=384.6 条), 但会造成漏洞成因指令缺失甚至识别不出漏洞, 而设置低会引入部分噪声指令, 因此需要权衡手工检查指令的效率和补丁识别的准确性来适应性的设置统计阈值. 本文局部路径图方法具有去噪能力, 为了获取更多漏洞信息, 本文在实验中适当降低阈值到 0.85, 虽然下降阈值增加了一些待分析指令量, 但能达到获取更多漏洞成因点信息的效果, 因此后文以 0.85 作为实验阈值 θ 进行方法探索.

在漏洞系统识别的假阳性方面, 本文分别手工验证了 LGBRoot 和 AURORA 系统识别的与漏洞相关的前 50 条指令中假阳性指令的数量 (top-50 FP, θ=0.85), LGBRoot 比 AURORA 的假阳性指令少 13.1 (263/20) 倍, 即指令识别准确率提高了 92.2%, 其中有 15 个样本 (即#4, #5, #6, #7, #10, #11, #12, #13, #14, #15, #16, #17, #18, #19, #20) 上的漏洞指令识别没有假阳性而 AURORA 只有 2 个 (#13, #17) 无假阳性, 因此 LGBRoot 能更准确地识别漏洞相关指令. 换句话说, 删除了 56.2% (即 (1 – 7693/19604 × 100%) × 92.2% = 61% × 92.2% = 56.2%) 的噪声指令. 在对假阳性指令进行分析, 数据如表 3 所示, 发现 AURORA 的假阳性指令产生原因主要包括分配释放内存操作及垃圾处理、复杂数据结构处理 (例如算数运算)、环境变量解析、错误处理及代码覆盖差异等, 而 LGBRoot 的假阳性主要集中在与数据的分配、释放等相关的指令或者函数里, 这也符合本文方法中在路径图上保留对内存空间具有分配 (例如 malloc)、释放 (例如 free) 能力的分支, 来联结复杂漏洞的产生和触发路径的逻辑关联关系, 从而留下的少量假阳性指令.

表 3　AURORA 与 LGBRoot 重要度阈值 0.85 的 top-50 FP 噪声分布

No.	Top-50 FP		内存垃圾处理		复杂数据结构处理		环境变量解析		错误处理		控制流未执行到		分配释放内存操作	
	A	L	A	L	A	L	A	L	A	L	A	L	A	L
#1	30	3	√	—	—	—	—	—	√	√	√	—	√	—
#2	20	4	√	—	—	—	—	—	√	—	√	—	√	√
#3	46	2	√	—	√	—	—	—	√	√	√	—	√	—
#4	27	0	—	—	—	—	—	—	√	—	√	—	—	—
#5	21	0	—	—	√	—	—	—	—	—	√	—	—	—
#6	15	0	—	—	√	—	—	—	—	—	√	—	—	—
#7	18	0	—	—	√	—	—	—	—	—	√	—	√	—
#8	24	6	—	—	√	—	—	—	—	—	√	—	√	√
#9	11	5	√	—	—	—	—	—	—	—	√	—	√	√
#10	7	0	—	—	√	—	—	—	—	—	√	—	—	—
#11	6	0	—	—	√	—	—	—	—	—	—	—	—	—
#12	11	0	—	—	√	—	—	—	—	—	—	—	—	—
#13	0	0	—	—	—	—	—	—	—	—	—	—	—	—
#14	6	0	—	—	—	—	√	—	—	—	—	—	—	—
#15	2	0	—	—	—	—	—	—	—	—	√	—	—	—
#16	11	0	—	—	√	—	√	—	√	—	—	—	—	—
#17	0	0	—	—	—	—	—	—	—	—	—	—	—	—
#18	3	0	—	—	—	—	—	—	—	—	√	—	—	—
#19	3	0	—	—	—	—	√	—	—	—	√	—	—	—
#20	2	0	—	—	√	—	—	—	—	—	√	—	—	—

注: A表示基准对比工具AURORA, L表示本文漏洞分析工具LGBRoot; √表示前50条指令中假阳性指令涉及该噪声类别, —表示前50条指令中假阳性指令不涉及此类别噪声

因此, LGBRoot 比 AURORA 从指令缩减、补丁点识别和指令识别准确率上获得更好的漏洞成因分析效果, 可以更好地帮助漏洞分析人员分析和理解漏洞成因.

● RQ2: LGBRoot 对于噪声筛除及成因逻辑联结效果如何?

为了回答该问题, 本文将所提的局部路径图方法按照方法的实现步骤, 分为函数级的局部 InterCG 图和基本块级的局部 IntraCFG 图的噪声消除和成因逻辑联结 2 个部分的实验结果进行评估, 其实验结果分别见表 4 和表 5, 并测试了系统各部分的时间开销见表 6.

表 4　函数间局部路径 InterCG 图作用效果

No.	Ins			Function					
	Total	Statistics	InterVCG	Total	Statistics	Remove	Left	Add	InterVCG
#1	19728	2062	648	424	89	64	25	35	60
#2	27409	7690	2575	579	241	209	32	7	39
#3	14165	2062	828	275	83	64	19	3	22
#4	13748	169	71	120	15	9	6	10	16
#5	10715	133	82	194	26	10	16	21	37
#6	10064	281	14	123	43	41	2	3	5
#7	40219	3259	1662	474	131	97	34	17	51
#8	46486	1518	786	624	85	53	32	34	66
#9	755	231	188	106	7	2	5	3	8
#10	9894	263	237	129	16	5	11	6	17
#11	16113	388	291	215	21	9	12	9	21
#12	9789	525	293	70	31	22	9	6	15
#13	2966	84	84	137	5	0	5	2	7
#14	2253	163	145	155	12	6	6	1	7
#15	831	295	196	88	5	2	3	1	4
#16	1028	38	27	18	5	3	2	4	6
#17	1035	9	9	13	2	0	2	2	4
#18	2288	406	260	21	11	7	4	2	6
#19	166	3	3	74	1	0	1	1	2
#20	422	25	21	12	6	1	5	3	8
Sum	230074	19604	8420	3851	835	604	231	170	401

注: Statistics表示统计分析的指令数; InterVCG表示函数间调用图; Remove、Left和Add分别表示删除、剩下和添加的函数数量; Ins和Function分别表示指令和函数

从表 4 可以看出, 局部函数调用图 InterCG 图可以明显减少指令噪声, 同时在漏洞成因函数间调用图 InterVCG 中添补一些图中缺失的前必经函数调用点, 用于联结逻辑关联关系. 从筛除噪声角度, InterCG 将统计成因分析后的总待检指令量从 19604 条减少到 8420 条, 总待检函数量从 835 个降低到 231 个, 其中有 30% (6/20, 即#1, #2, #3, #6, #7, #12) 的样本函数筛除量达到 70% 以上. 从联结逻辑关联关系角度, 20 个程序共添补 170 个联结逻辑关系的函数进 InterVCG, 加上图上原有的 231 个, 新的 InterVCG 共有 401 个函数.

由此可见, 噪声指令在函数级别的分布面比较广不利于手工排查, 需要借助漏洞成因的图结构关联性来缩减待检函数范围. 在利用 InterCG 中的函数调用关系联结成因逻辑关系方面, 可以将缺失的必经函数点添补进漏洞成因路径图, 关联漏洞成因相关的函数, 这样对于漏洞发生逻辑, 比如 UAF 漏洞可以更直观地判断出错对象的崩溃点, 分配、释放位置的先后顺序, 从而确定漏洞修补方案.

从表 5 来看, 局部基本块级转移图 IntraCFG 也能减少指令噪声. IntraCFG 将 InterCG 处理的成因分析的指令数从的 8420 条缩减到 7706 条, 基本块量缩减了 33% (1 – 1160/1721), 虽然 IntraCFG 的指令筛除量没有 InterCG 大, 但是其一方面原因是 InterCG 提前处理了一部分满足 IntraCFG 筛减条件的指令, 另一方面从剩余基本块和剩余指令数比例 6.64% (7706/1160) 可以看出, 指令在基本块层面比较集中, 噪声相对较少, 这也符合漏洞成因的聚集性逻辑关联特征. 需要提到的是, 在迭代遍历 IntraCFG 控制流边联结逻辑关系的过程中, 发现缺失的前必经基本块数量 PreNode 是 6397 个, 远远多于原始待分析基本块数量 1160, 造成可视化时图中信息阅读理解不

方便, 在考虑可用性和效果方面, 基本块级别指令修复粒度过细, 因此本文不添补具体缺失的基本块进漏洞成因函数内基本块转移图 IntraVCFG, 仅恢复统计成因报告中已有 jmp, jnz 等控制流转移指令所在基本块之间的函数内控制流转移逻辑关联局部图信息, 当局部图与函数入口不连通时, 将原有的基本块转移边 (left edge) 在 IntraVCFG 中通过函数入边简单联结起来, 方便漏洞理解和分析.

表 5　函数内基本块级的局部 IntraCFG 图作用效果

No.	Ins		BasicBlock					
	InterVCG	IntraVCFG	Total	Statistics	Remove	Left edge	PrevNode	IntraVCFG
#1	648	633	257506	168	46	88	722	122
#2	2575	2750	257624	493	75	316	742	418
#3	828	872	257644	208	52	111	783	156
#4	71	69	14012	24	6	13	62	18
#5	82	32	4813	29	23	3	181	6
#6	14	6	11329	8	5	2	14	3
#7	1662	1383	16741	250	101	101	1688	149
#8	786	458	16732	87	48	24	215	39
#9	188	107	1019	22	8	10	57	14
#10	237	225	59028	42	19	16	228	23
#11	291	281	42502	69	31	23	237	38
#12	293	222	17526	81	35	28	238	46
#13	84	55	3319	35	16	11	217	19
#14	145	105	3319	39	31	5	210	8
#15	196	182	305	30	14	10	44	16
#16	27	24	917	6	6	0	155	0
#17	9	9	440	4	4	0	82	0
#18	260	255	875	28	14	10	89	14
#19	3	3	234	24	4	24	111	20
#20	21	35	1473	74	23	40	322	51
Sum	8420	7706	967358	1721	561	835	6397	1160

注: Statistics表示统计分析的指令数; InterVCG表示函数间调用图; IntraVCFG 表示函数内基本块转移图; Remove和Left-edge分别表示删除的和剩下的基本块边; PreNode表示前必经基本块数量. Ins和BasicBlock分别表示指令和基本块

在系统时间开销方面, 从表 6 可以看出, LGBRoot 的时间开销较小. 在样本集上的数据中, 从局部函数间调用图 InterCG 的建立到可视化输出漏洞成因路径图 V 的总体平均时间为 12.4 s (248.61/20), 其中 75% (15/20) 的样本程序的开销小于 9 s, 主要时间开销是局部路径图中 InterCG 和 IntraCFG 的建立.

表 6　LGBRoot 各部分时间开销 (s)

No.	InterCG build	InterCG trim	InterVCG union	IntraCFG build	IntraCFG trim	IntraVCFG union	V plot	Total
#1	1.1734	0.0006	0.0013	6.891	~0	0.422	0.0003	8.4886
#2	4.6749	0.002	0.0091	7.194	~0	0.31	0.0002	12.1902
#3	0.8512	0.0007	0.0029	6.7739	0.0049	0.3959	0.0002	8.0297
#4	0.5195	0.0001	0.0003	0.531	~0	0.012	0.0001	1.063
#5	0.6582	0.0002	0.0003	0.3409	~0	0.031	0.0001	1.0307
#6	0.2813	0.0001	0.0003	0.411	~0	0.002	0.0074	0.7021
#7	29.2697	0.0021	0.0044	0.7599	0.008	1.2319	0.0013	31.2773
#8	97.3483	0.0025	0.0065	0.8159	0.005	0.2569	0.0004	98.4355
#9	4.7125	0.0001	0.0003	0.0439	~0	0.0119	0.0046	4.7733
#10	0.6272	0.0002	0.0003	2.5219	~0	0.0309	0.0001	3.1806
#11	0.6214	0.0003	0.0005	1.7769	0.0009	0.0319	0.0001	2.432
#12	0.1429	0.0002	0.0002	0.6019	0.0009	0.065	0.0001	0.8112
#13	17.8659	~0	~0	0.1819	0.001	0.0649	0.0035	18.1172
#14	53.1158	0.0001	~0	0.17	0.002	0.038	0.0105	53.3364

表 6 LGBRoot 各部分时间开销 (s)(续)

No.	InterCG build	InterCG trim	InterVCG union	IntraCFG build	IntraCFG trim	IntraVCFG union	V plot	Total
#15	0.4509	0.0001	~0	0.022	~0	0.01	0.0146	0.4976
#16	2.0352	~0	~0	0.1339	~0	0.1589	0.0031	2.3311
#17	0.1009	~0	~0	0.0189	~0	0.0119	~0	0.1317
#18	0.8176	0.0001	0.0031	0.0559	~0	0.015	0.0001	0.8918
#19	0.0123	~0	~0	0.018	~0	0.011	~0	0.0413
#20	0.1916	~0	0.0001	0.4322	~0	0.2201	0.0001	0.8441
Sum	215.4707	0.0094	0.0296	29.695	0.0227	3.3312	0.0468	248.61

注: Build、Trim和Union分别表示图建立、筛除噪声和联结逻辑结构图; Plot表示将图可视化, ~0表示约等于0

综上所述, 在少量的时间开销下, InterCG 图从函数调用层面判断统计指令所属函数是否符合漏洞成因函数域范围, 用于筛除噪声和联结指令所在函数间的逻辑关系, 而 IntraCFG 图在基本块层面筛除孤立的噪声指令, 并关联成因相关指令. 通过两个步骤的漏洞成因路径图 V 的建立, 在不用逐条指令地址排查下可以更快地确定指令与漏洞的关联性, 并提供指令所在上下文信息. 因此, 基于局部路径图的方法通过加入了漏洞相关点的图结构关联信息, 一方面可以自动剔除噪声指令减少分析人员的手工检验分析的工作量, 另一方面可以关联指令逻辑关系更方便理解漏洞成因, 是一个具有指导意义和实践价值的自动漏洞成因分析方法.

5 案例分析及未来展望

在实验中本文发现了两个有趣的实验现象.

案例 1: 漏洞多处成因点之间的图关联性. 对于 CVE-2019-16351 漏洞样例的空指针解引用, 漏洞的发生来源于 jfif_load() 函数中的 jfif->phcac 成员没有得到正确分配处理, 后续 jfif_decode() 调用的 huffman_decode_step() 解码引用该结构体对象的操作出现错误. 当阈值 θ 设置为 0.90 的时候, jfif_load() 函数由于干扰指令较多而未被统计分析识别出来, 但本文的局部路径图方法能识别出 jfif_load() 有对堆的分配操作因此在局部路径图中修补了该分配函数相关的逻辑信息. 当后续阈值降为 0.87 时 jfif_load() 漏洞成因函数被识别出来, 也验证了本文图结构化联结修补漏洞成因相关逻辑关联关系方案的可行性.

案例 2: 漏洞间的漏洞成因图的关联性. 在测试样本集 LIBTIFF 中的 CVE-2016-10093 和 CVE-2016-10270 堆溢出漏洞, 漏洞的发生都来源于文件 tiffcrop.c 的 3 701 行 bufp += bytes_read 中处理图片格式的字节数 bytes_read 在图片处理失败时可为–1, 而后续的指针 bufp 的根据错误的 bytes_read 偏移量访问造成堆下溢错误, 上述两个漏洞是在不同的函数调用路径和访问点触发了堆溢出, 但是本文在识别漏洞成因路径图时发现两个样本的局部 InterCG 图一样, 而且局部 IntraCFG 图的差异不大. 由此可见, 漏洞成因相同的不同漏洞触发点的漏洞成因路径图具有相似性. 基于此发现认为本文的局部漏洞路径图方法可以用于漏洞成因的识别和分类.

6 讨 论

通过前面的测试集评估, 本文基于局部路径图的自动漏洞成因分析系统可以识别和解释漏洞成因, 并大大缩减了漏洞分析人员的手工验证漏洞成因的工作量. 然而, 由于本文系统是基于已有的统计分析系统生成的结果报告来填充漏洞成因路径图, 其效果受限于统计分析系统的能力, 特别是最近发布的统计分析系统, 例如 Aurora 等, 其基于模糊测试构建成因分析的语料存在因引入多漏洞 DiGiuseppe[19]产生的差异指令而影响实验结果的问题. 除此以外, 当软件程序的设计中, 其对象的分配、释放路径很分散, 且中间的联结函数都在统计分析的差异指令列表中时, 局部路径图无法很好地区分与漏洞成因无关的分配和释放函数从而留下了一些无法去除的指令噪声.

7 总 结

本文根据漏洞成因相关点组成的局部路径符合程序功能逻辑执行中的代码结构特性, 提出了基于局部路径图

的漏洞成因点噪声筛除方法和成因点逻辑关系恢复方法, 解决了基于统计分析进行漏洞成因分析的随机性噪声和漏洞成因相关点逻辑关联关系缺失的问题, 并在数据集上验证了方法的有效性. 基于局部路径图, 一方面可以自动剔除噪声指令, 减少分析人员的手工检验分析的工作量; 另一方面可以修补和关联指令逻辑关系, 方便理解漏洞成因, 从而加快漏洞分析效率.

References:

[1] Agrawal H, Horgan JR. Dynamic program slicing. ACM SIGPLAN Notices, 1990, 25(6): 246–256. [doi: 10.1145/93548.93576]

[2] Newsome J, Song DX. Dynamic taint analysis for automatic detection, analysis, and SignatureGeneration of exploits on commodity software. In: Proc. of the 2005 Network and Distributed System Security Symp. San Diego: The Internet Society, 2005. 3–4.

[3] Agrawal H, Horgan JR, London S, Wong WE. Fault localization using execution slices and dataflow tests. In: Proc. of the 6th Int'l Symp. on Software Reliability Engineering. Toulouse: IEEE, 1995. 143–151. [doi: 10.1109/ISSRE.1995.497652]

[4] Blazytko T, Schlögel M, Aschermann C, Abbasi A, Frank J, Wörner S, Holz T. AURORA: Statistical crash analysis for automated root cause explanation. In: Proc. of the 29th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2020. 14.

[5] Shen SQ, Kolluri A, Dong Z, Saxena P, Roychoudhury A. Localizing vulnerabilities statistically from one exploit. In: Proc. of the 2021 ACM Asia Conf. on Computer and Communications Security. Hong Kong: ACM, 2021. 537–549. [doi: 10.1145/3433210.3437528]

[6] Cai J, Zou P, Yang SF, He J. Vulnerable spots localization methods for software vulnerability analysis. Journal of National University of Defense Technology, 2015, 37(5): 141–148 (in Chinese with English abstract). [doi: 10.11887/j.cn.201505022]

[7] Vessey I. Expertise in debugging computer programs: A process analysis. Int'l Journal of Man-machine Studies, 1985, 23(5): 459–494. [doi: 10.1016/S0020-7373(85)80054-7]

[8] Wong WE, Gao RZ, Li YH, Abreu R, Wotawa F. A survey on software fault localization. IEEE Trans. on Software Engineering, 2016, 42(8): 707–740. [doi: 10.1109/TSE.2016.2521368]

[9] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. of the 2010 IEEE Symp. on Security and Privacy. Oakland: IEEE, 2010. 317–331. [doi: 10.1109/SP.2010.26]

[10] Cui WD, Ge XY, Kasikci B, Niu B, Sharma U, Wang RY, Yun I. REPT: Reverse debugging of failures in deployed software. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 17–32.

[11] Kusumoto S, Nishimatsu A, Nishie K, Inoue K. Experimental evaluation of program slicing for fault localization. Empirical Software Engineering, 2002, 7(1): 49–76. [doi: 10.1023/A:1014823126938]

[12] Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. ACM SIGPLAN Notices, 2005, 40(6): 15–26. [doi: 10.1145/1064978.1065014]

[13] Liu C, Fei L, Yan XF, Han JW, Midkiff SP. Statistical debugging: A hypothesis testing-based approach. IEEE Trans. on Software Engineering, 2006, 32(10): 831–848. [doi: 10.1109/TSE.2006.105]

[14] Hu PF, Zhang ZY, Chan WK, Tse TH. Fault localization with non-parametric program behavior model. In: Proc. of the 8th Int'l Conf. on Quality Software. Oxford: IEEE, 2008. 385–395. [doi: 10.1109/QSIC.2008.44]

[15] Zhang YL, Rodrigues K, Luo Y, Stumm M, Yuan D. The inflection point hypothesis: A principled debugging approach for locating the root cause of a failure. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 131–146. [doi: 10.1145/3341301.3359650]

[16] McCabe TJ. A complexity measure. IEEE Trans. on Software Engineering, 1976, SE-2(4): 308–320. [doi: 10.1109/TSE.1976.233837]

[17] Mitre Corporation. 2022 CWE top 25 most dangerous software weaknesses. 2022. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[18] EAGLE C. The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. 2nd ed., San Francisco: No Starch Press, 2011: 276–302.

[19] DiGiuseppe N, Jones JA. On the influence of multiple faults on coverage-based fault localization. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Ontario: ACM, 2011. 210–220. [doi: 10.1145/2001420.2001446]

附中文参考文献:

[6] 蔡军, 邹鹏, 杨尚飞, 何骏. 软件漏洞分析中的脆弱点定位方法. 国防科技大学学报, 2015, 37(5): 141–148. [doi: 10.11887/j.cn.201505022]

余媛萍(1994－), 女, 博士, 主要研究领域为系统安全, 漏洞挖掘, 漏洞分析.

贾相堃(1990－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为系统安全, 漏洞挖掘与分析.

苏璞睿(1976－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为系统安全, 恶意代码分析, 漏洞挖掘.

黄桦烽(1988－), 男, 博士, 工程师, 主要研究领域为计算机系统安全, 漏洞自动挖掘与利用.