一种基于原子控制流图的继承性安全漏洞检测方法

芮志清[1,2]，吴敬征[1]，罗天悦[1]，段旭[1,3]，赵航[1,4]，陈昶宇[1,3]

（1. 中国科学院软件研究所 智能软件研究中心 北京市 100190；2. 中国科学院大学 人工智能学院 北京市 100049；3.北京科技大学 计算机与通信工程学院 北京市 100083；4. 莫纳什大学 信息技术学院 墨尔本市 3800；）

摘要：当前，全球99%的IT系统都普遍使用了开源代码，开源代码的大量复用，导致继承性安全漏洞的广泛出现。继承性安全漏洞是指由代码复用导致的不同版本程序中出现的同一漏洞。典型的继承性漏洞检测方法均采用了基于CFG（Control Flow Graph，控制流图）的相似性检测技术，而针对大规模的CFG分析会导致漏洞检测漏报率较高。该文提出一种基于原子控制流图的改进的CFG分析方法，该方法首先将程序代码和已知漏洞代码中包含语义信息的最小不可分割代码段作为最小节点，生成原子控制流图，然后基于节点指纹进行快速哈希检索，最后进行函数漏洞率的计算。通过对该方法进行综合的分析评价可知，该方法可以有效降低继承性漏洞检测的漏报率，且适用于对千万行级的大规模代码进行漏洞检测。

关键字：原子控制流图；控制流图；漏洞挖掘；静态分析； 继承性漏洞

中图分类号：TP 309.1　　文献标志码：A

A new Approach for Detecting Inheritance Vulnerabilities based on Atomic Control Flow Graph

RUI Zhi-qing[1,2], WU Jing-zheng[2], LUO Tian-yue[2], DUAN Xu[2,3],

ZHAO Hang[2,4], CHEN Chang-yu[2,3]

(1. Intelligent Software Research Center, Institute of Software, Chinese Academy of Sciences, Beijing 100190, P.R.China; 2. Artifical Intelligence Academy, University of China Academy of Science, Beijing 100049, P.R.China; 3. School of Computer & Communication Engineering, University of Science and Technology Beijing, Beijing 100083, P.R.China;4. Faculty Of Information Technology, Monash University, Melbourne 3800, Australia)

Abstract: Nowadays, the open source project is used in 99% of organizations' IT systems. The open source

基金项目：国家自然科学基金（61772507）；国家重点研发计划（2017YFB0801902）

收稿日期：2018/11/30

作者简介：芮志清（1997）,男，硕士，漏洞挖掘。E-mail: Zhiqing.Rui@Gmail.com

code is reused everywhere, which induces the widespread emergence of inherited security vulnerabilities. The inherited vulnerabilities occur in different versions of the program due to the code reuse. Most methods use CFG (Control Flow Graph) similarity to detect the inheritance vulnerabilities. However, oversize CFG analysis will result in a high rate of false negatives in vulnerability detection. This article proposed an improved CFG comparison method based on the atomic control flow graph. The method first takes the least indivisible code segment, which contains semantic information in the program code and the known vulnerability code, as the minimum node. Then the atomic control flow graph will be generated. After that, executing the rapid hash detection based on node fingerprint. Finally, the function vulnerability rate is calculated. Through the analysis of the method, it can be concluded that this method can effectively reduce the false negative of inherited vulnerability detection, and it will be applicable for vulnerability detection of huge source code.

Key words: Atomic Control Flow Graph; Control Flow Graph; Vulnerability Detection; Static Analyze; Inherited Vulnerability

引言

由于开源技术的广泛应用，开源代码被大量复用，导致继承性漏洞引起的安全问题日益增多。继承性安全漏洞是指由代码复用导致的不同版本程序中出现的同一漏洞。例如，以CVE（通用漏洞披露库）发布的编号为 CVE-2016-7054 “脏牛”[1,2]漏洞为例，该漏洞存在于 Linux Kernel 中，影响范围覆盖 2.x 至 4.18.13 之间的所有版本。该漏洞源于 Linux kernel 中处理 Copy-On-Write 操作时存在竞争条件漏洞，攻击者可利用该漏洞以低权限向只读内存写入数据，从而提升至管理员权限, 最终实现对设备的完全控制。继承性漏洞是当前安全领域危害性高又易于利用的典型漏洞。

近年来，安全研究人员提出了大量方法来对继承性漏洞进行检测。2017 年 Wu 等提出 LaChouTi[3]工具，该工具采用补丁文件作为匹配模式，在源代码中采用滑动窗口算法以文本匹配的方式进行漏洞模式匹配。该方法速度快，误报率低，但其基于对文本的精准匹配，会导致漏报率增高，不适用于继承性漏洞检测。2015 年 Perl 等提出 VCCFinder[4]，该方法对有漏洞的代码提交抽取特征，训练 SVM 分类器，并使用此分类器对项目的每次提交进行特征检测，VCCFinder 有效降低了误报率，但漏报率较高。2016 年 Feng 等提出 Genius[5]方法，该方法使用一种小型的 CFG，以图搜索的方式检测二进制代码中的已知漏洞。基于 CFG 的继承性漏洞检测技术发展迅速，但对于规模较大的函数，生成的 CFG 也会增大，导致漏洞检测的漏报率较高[6-8]。

为了解决基于 CFG 的漏洞检测技术在继承性漏洞检测中漏报率高的问题，本文提出了一种称为原子控制流图的新型 CFG 结构（简称为μCFG），是具有最小语义结构的控制流图。基于μCFG的概念，本文实现了μCFG生成工具s2μ和继承性漏洞检测方法Diffμ。其中，s2μ首先将源代码进行递归拆分到最小顺序结构代码段，生成μCFG的节点和边，然后对变量名进行归一化处理，最后使用哈希算法生成节点指纹。Diffμ首先从漏洞平台获取项目和补丁的源代码，然后使用s2μ工具对源代码和补丁生成μCFG数据库，再对数据库进行快速哈希检索，最后分析挖掘，得到节点的漏洞率信息。

1 相关研究

1.1 漏洞挖掘方法

漏洞挖掘，是指从程序及源代码中寻找可能会被攻击者利用的代码安全缺陷。根据是否需要运行待测程序，漏洞挖掘方法主要分为两类：静态分析方法和动态分析方法[9,10]。静态分析的代表性方法有 Splint[11]、Fortify[12]、Checkmarx[13]、Coverity[14]、CPPCheck[15]等。其中，Splint 是一款经典的对 C 代码进行静态分析的工具，该方法可以快速检测典型的代码缺陷，如数组越界、内存泄露和死循环等。动态分析方法有模糊测试、渗透测试和符号执行等方法。Chen 等提出的 IOTFuzzer[16]，IOTFuzzer 是基于模糊测试的方式，从物联网设备的官方 APP 中提取与物联网设备的通信协议，并使用模糊测试的方式生成正常或异常的数据包，用以检测设备中是否存在如内存中断的漏洞。Yun 等提出的 APISan[17]，该方法对函数的错误调用进行分析的工具，采用了符号执行技术，对执行中 API 调用的模式进行统计分析，检查返回值、参数和约束条件，最后计算 API 误使用的概率。

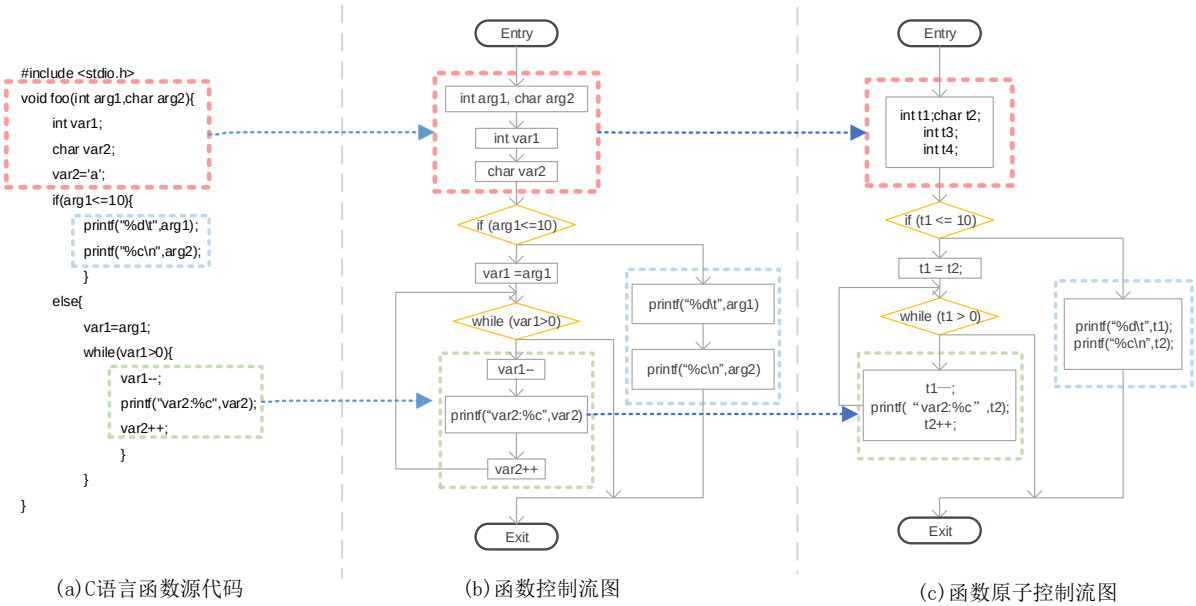
1.2 继承性漏洞检测方法

继承性漏洞检测，是检测安全漏洞代码是否在被测对象中被复用。ReDebug[18]工具，是将继承性漏洞检测问题转变为漏洞的代码克隆问题,首先从版本控制软件每次提交代码的补丁入手，找到初始的漏洞代码的版本，然后递归查找克隆未修复代码进行开发的代码，最后对这些代码进行报警。OSSPolice[19]工具，用于检测代码中是否存在开源协议未授权漏洞，是将代码继承关系转化为系统代码识别问题，将待测项目中模块特征进行提取，找到其所依赖的库和框架，最后分析出是否引用了使用某开源协议的代码，防止因为授权问题而引起纠纷。这些方法从项目的依赖关系来寻找继承关系，可以快速对项目进行扫描和检索，但是对于继承性漏洞代码本身不具有针对性。

1.3 基于CFG的函数相似性检测方法

CFG 是函数语句的运行流程，可以用来代表函数的结构特征。CFG的相似性问题是很多领域的核心技术[6,20]，如漏洞挖掘，恶意软件检测，软件抄袭检测等。CFG是一个用来表示函数控制流的有向图，可以表示为$CFG(V_{CFG}, E_{CFG})$。其中V_{CFG}为CFG中的节点的集合，每个 CFG 都只有 1 个进入节点和 1 个退出节点。E_{CFG}为 CFG 中边的集合。图的相似度可以用其最大同构子图的规模来度量[8]，而子图同构问题是一个 NP 完全问题。通过对函数的 CFG 进行比较，可以得到函数的相似性。若两个函数的相似性较高，则该函数有较大的几率是复用的函数。CFG 是对函数的抽象表示，通过变量归一化的方式，可以将函数的变量名差异引起的函数差异去除。近年也有安全研究人员利用机器学习和深度学习的方式进行函数相似性检测[21,22]。Xu 等提出的 Gemini[22]方法，把 CFG 简化为特征化的 ACFG，再使用 Structure2vc 算法将 ACFG 转为数字变量，使用神经网络训练的方式计算函数的距离，得到函数的相似性。Zhao 等提出的 DeepSim[23]方法，将 CFG 的语义特征抽取成为一个高维向量，该向量表示 CFG 的结构，输入输出，所含变量等信息。并将该向量编码在语义特征矩阵中，使用前馈神经网络的方式对该矩阵的进行比较。

综上，在检测继承性漏洞时，如果遇到 CFG 规模庞大的情况，基于 CFG 的函数相似性比较算法会出现较高的漏报率。为了解决该问题，本文提出一种基于μCFG 的继承性漏洞检测方法。



(a) C语言函数源代码　　(b) 函数控制流图　　(c) 函数原子控制流图

图 1 控制流图与原子控制流图的比较示意图

2 基于μCFG的继承性漏洞检测方法

通过对大量已公布的漏洞补丁文件进行分析，漏洞往往存在于一个函数中的较小行数的代码中。由于额外的代码量过多，因此在函数的 CFG 较为庞大的情况下，对整个函数进行图检索会造成漏报率过高。μCFG 首先将大的 CFG 进行拆分，以最小的顺序结构代码段为叶节点，形成嵌套式的CFG然后基于快速哈希表的方式，自底向上的从叶节点到整个 CFG 进行逐一比较，最终得到与漏洞函数的相似度。控制流图与原子控制流图的比较如图 1 所示。

2.1 μCFG模型描述

定义 1：定义 CFG 为一个有向图，其表达式为

$$\mathrm{CFG}(\mathrm{V_{CFG}}, \mathrm{E_{CFG}}); \quad \text{公式(1)}$$

其中$\mathrm{V_{CFG}}$是 CFG $node$（节点）的集合，每个节点都仅有一个入口点和一个出口点。$\mathrm{E_{CFG}}$是 CFG $edge$（边）的集合，每条边从一个$node$的出口点指向另一个$node$的入口点，代表一条从节点到节点的控制流。$\mathrm{V_{CFG}}$有且仅有一个$entry\ node$（入口节点），$entry\ node$只有后继节点，没有前驱节点，从该节点向后遍历，可以遍历到任意节点。$\mathrm{V_{CFG}}$有且仅有一个$entry\ node$，出口节点只有前驱节点，没有后继节点，从任意节点向后遍历，都可以遍历到该节点。使用$\mathrm{IE_i}$和OE_i来分别表示节点i的入边和出边的集合。使用IN_i和ON_i来分别表示以节点i入边和出边与节点i相连的集合[6]。

定义 2：令$\mathrm{CFG_{arr}}$为CFG的集合，$\mathrm{n(A)}$为集合A元素的数量，定义μCFG为$\mathrm{CFG(V_{\mu CFG}, E_{\mu CFG})}$，$\mathrm{\mu CFG_{arr}}$为μCFG的集合，则μCFG满足以下条件：

$$\forall\ \mu\mathrm{CFG} \in \mu\mathrm{CFG_{arr}},\ \mu\mathrm{CFG} \in \mathrm{CFG}; \quad \text{公式(2)}$$

$$\text{对于}\forall\ (node\ i, j \in \mu\mathrm{CFG}) \wedge (j \in \mathrm{IN}_i \cap \mathrm{ON}_i),$$

$$if\ \mathrm{n}\left(\mathrm{IE}_j \cap \mathrm{OE}_j\right) \leq 2 \Longrightarrow \mathrm{n}(\mathrm{IE}_i \cap \mathrm{OE}_i) \geq 3,$$

$$if\ \mathrm{n}\left(\mathrm{IE}_j \cap \mathrm{OE}_j\right) \geq 3 \Longrightarrow \mathrm{n}(\mathrm{IE}_i \cap \mathrm{OE}_i) \leq 2; \quad \text{公式(3)}$$

即μCFG中入度和出度都为 1 的节点不连续存在。

定义 3：定义节点变量归一化操作$\chi(v)$，对于节点v所表示的代码中出现的变量名，依次重命名为$\mathrm{t_1}, \mathrm{t_2}, \mathrm{t_3}, \ldots, \mathrm{t}_n$ (n 为正整数)。

定义 4：定义节点指纹$H(v)$，对于节点v，首先进行$\chi(v)$操作，然后使用 MD5 算法进行对节点v的代码进行哈希计算，得到一个唯一表示该代码段的固定位数的值。

定义 5：令KV为已知漏洞，定义$\mathrm{DB_{\mu CGF}}(KV)$是存储已知漏洞μCGF的数据库。

定义 6：令TC为待测代码，定义$\mathrm{DB_{\mu CGF}}(TC)$是存储待测代码μCGF的数据库。

2.2 μCFG生成工具：s2μ

基于定义 2，给出生成μCFG的原型工具s2μ，实现从源代码到生成μCFG生成。

首先，对于输入的源代码文件，s2μ按照函数边界将其拆分为不同的函数段。对于每个函数段，建立$entry\ node$和$exit\ node$，将其包含在$V_{\mu CFG}$中。按照分支结构和循环结构的分支点和汇聚点，递归的将函数段拆分为不同的最小语义段，每个最小语义段都是一段不包含分支的顺序结构代码，将最小语义段视为不可分割的最小代码段。以最小语义段为基础生成μCFG的节点，组成$V_{\mu CFG}$。在上述拆分的过程中，将控制流上相邻的代码段之间建立边的关系，组成$E_{\mu CFG}$。μCFG 生成流程如图 2 所示。

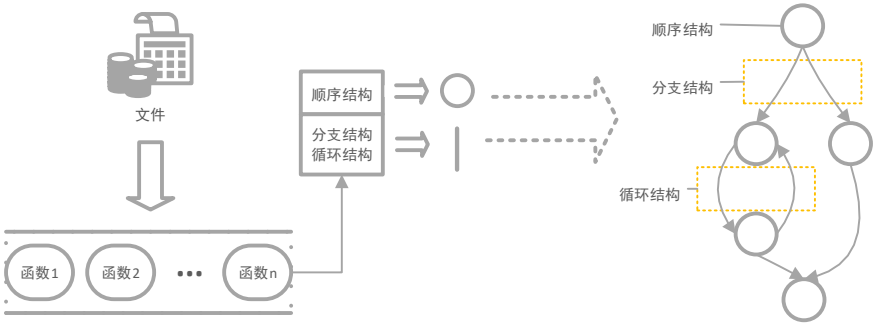


图 2 s2μ 流程示意图

基于定义 4，对于每个节点建立节点指纹。先将节点的代码部分以定义 3 的规则进行归一化操作$\chi(v)$，然后使用 MD5 算法对节点的文本进行哈希计算，得到能标识该节点的值。将该值作为节点v的属性封装到节点v中。

2.3 继承性漏洞检测工具：Diffμ

建立原型系统Diffμ，是基于μCFG进行继承性漏洞的漏洞挖掘的工具。Diffμ可以建立已知漏洞μCFG数据库$DB_{\mu CGF}(KV)$，待测代码μCFG数据库 $DB_{\mu CGF}(TC)$，并基于快速哈希表对控制流图进行匹配并生成漏洞检测报告。

生成$DB_{\mu CGF}(KV)$的过程如下：1）从漏洞信息发布平台，如 CNNVD（China National Vulnerability Database of Information Security）[24]，CVE（Common Vulnerabilities and Exposures），NVD（National Vulnerability Database）[25]，获取漏洞信息，然后获取补丁文件和代码源文件。对于存在补丁 Git 地址的漏洞，使用爬虫获取其提交编号，然后在 Git 项目中使用提交编号获取补丁文件和打补丁前的代码源文件。对于没有 Git 补丁地址的漏洞，使用其他方式抓取其补丁及打补丁前的代码源文件。2）使用μCFG生成工具s2μ对代码源文件生成μCFG，并存

储到图数据库（如 Neo4j）之中，即为$\mathrm{DB}_{\mu\mathrm{CGF}}(KV)$。

生成$\mathrm{DB}_{\mu\mathrm{CGF}}(TC)$的过程如下：1）对于待测项目，获取其源代码。2）使用s2μ对代码源文件生成μCFG，并存储到图数据库中，即为$\mathrm{DB}_{\mu\mathrm{CGF}}(TC)$。

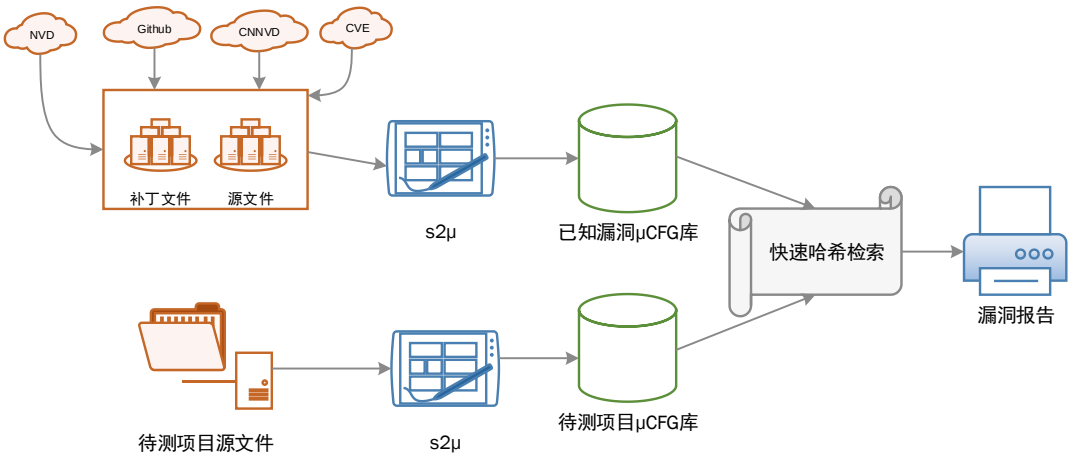


图 3 Diffμ 体系架构图

然后进行漏洞率的计算。使用快速哈希表的方式，比较$\mathrm{DB}_{\mu\mathrm{CGF}}(TC)$和$\mathrm{DB}_{\mu\mathrm{CGF}}(KV)$中的节点指纹$H(v)$。然后对每个μCFG中匹配的节点进行统计。对于每个匹配所对应的待测项目和已知漏洞节点，按照与补丁位置的距离计算权值，距离补丁位置越近的节点漏洞率权值越大。最后对每个函数中匹配到的节点进行统计，按照漏洞率权值大小进行排序，最后输出继承性漏洞检测报告。

3 基于μCFG的继承性漏洞检测方法分析

3.1 漏洞检测方法的性能指标

漏洞挖掘技术的性能评价指标主要有：1）待分析对象规模和复杂度。是指分析程序能够接受的输入程序的规模和复杂度的大小，待分析对象规模越大，复杂度越高，对检测算法的要求就越高。2）检测过程效率。指检测过程中需要消耗的时间和检测工具的自动化程度。对于相同规模的程序，如果其自动化程度越高，消耗时间越短，其检测效率越高。3）发现漏洞严重程度。发现的缺陷对信息安全造成的威胁的严重程度，对信息安全造成的威胁越大，则漏洞的严重程度越高。4）误报率。在工具发现的漏洞中，假阳性的概率。较低的误报率可以降低安全人员手动验证漏洞存在性的时间成本。5）漏报率。在工具未发现的漏洞中，假阴性的概率。漏报率的高低影响项目中存在的隐患数量多少[10]。

3.2 基于μCFG的继承性漏洞检测技术性能分析

本方法主要用来解决大规模的 CFG 搜索漏洞挖掘中漏报率高的问题。在使用传统 CFG 进行漏洞检测过程中，由于 CFG 过大，导致在对函数进行匹配时往往不能精准的匹配到漏洞所在位置，会导致漏洞的漏报率有所增加。但是μCFG方法比传统 CFG 方法的叶节点的规模有所放大，使用了变量的归一化，在匹配时不是基于每行代码进行匹配，而是对一个原子代码段进行匹配，能有效降低漏报率。

该工具可以对大规模高复杂度的代码进行分析，并且效率高。在 Intel Core i7-7700 8 核心 CPU，32GB 内存，Ubuntu 系统，使用 Docker 容器搭建运行环境。使用原型工具对 Linux Kernel 建立μCFG图，最后成功建立$\mathrm{DB}_{\mu\mathrm{CGF}}(TC)$。Linux Kernel 代码量为 2500 万行，生成的数据库文件大小为 10.8GB，耗时约 10 小时。实验说明对于大规模搞复杂度的代码进行分析具有可行性。但对于需要即时快速的大规模代码漏洞检测，需要进一步优化。本方法是基于快速哈希表的匹配，速度较快。且本方法无需人工参与，可以做到完全的自动化，该方法降低了人力成本，具有较高的效率。

发现漏洞的严重程度可控制。对于高危漏洞的补丁，其继承性漏洞有很大几率也是高危漏洞，若采用高危漏洞的补丁进行漏洞筛查，所能筛选出的漏洞为高危漏洞的可能性更大。

μCFG方法旨在解决漏洞挖掘的漏报率高的问题，同时，针对检测的误报率问题，本方法也具备一定的应对能力，比如针对补丁文件对应的代码位置对$\mathrm{DB}_{\mu\mathrm{CGF}}(KV)$中的节点进行筛查，可以减少检测的误报率。

4 结论

为了解决继承性漏洞挖掘中，使用大规模的CFG搜索算法时漏报率高的问题，本文提出了一种改进的CFG结构——μCFG。并且提出了生成μCFG的工具s2μ和基于μCFG进行继承性漏洞挖掘的工具Diffμ。该方法通过将CFG节点进行整合，建立μCFG，再通过快速哈希检索的方式对继承性漏洞进行检测，能够解决漏报率高的问题。μCFG 方法能应用在大规模代码上进行分析，效率高，漏洞危害等级可控制。且对于误报率的问题也提出了对应的解决方案。

后续的研究将会在μCFG的基础上，采用深度学习算法，进一步提高当前继承性漏洞挖掘的准确率，降低误报率和漏报率，从而提高继承性漏洞的检测能力。

参考文献：

[1] Saleel A P, Nazeer M, Beheshti B D. Linux kernel OS local root exploit[C]. 2017 IEEE Long Island Systems, Applications and Technology Conference, 2017.

[2] CVE-2016-7054(Common Vulnerabilities and Exposures)[EB/OL]. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054.

[3] Wu J, Yang M. LaChouTi: Kernel vulnerability responding framework for the fragmented android devices[C]. 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2017: 920-925.

[4] Perl H, Dechand S, Smith M, et al. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits[C]. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015: 426-437.

[5] Feng Q, Zhou R, Xu C, et al. Scalable Graph-based Bug Search for Firmware Images[C]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016: 480-491.

[6] Chan P, Collberg C. A Method to Evaluate CFG Comparison Algorithms[C]. International Conference on Quality Software, 2014: 95-104.

[7] 马锐, 高浩然, 窦伯文, et al. 基于改进 GN 算法的程序控制流图划分方法[J]. 清华大学学报(自然科学版), 2018.

MA R, GAO H, DOU B, et al. Control flow graph division based on an improved GN algorithm[J]. Journal of Tsinghua University (Science and Technology), 2018.

[8] Bunke H, Shearer K. A graph distance metric based on the maximal common subgraph[J]. Pattern Recognition Letters, 1998, 19(3): 255-259.

[9] 赵晖. 面向军工应用软件的源代码漏洞分析系统的研究与实现[D]. 北京交通大学, 2015.

ZHAO H. The research and implementation of source code vulnerability analysis system for military application [D]. Beijing Jiaotong University, 2015.

[10] 张健, 张超, 玄跻峰, et al. 程序分析研究进展[J]. 软件学报, 2019, 30(1): 1-31.

ZHANG J, ZHANG C, XUAN J, et al. Recent Progress in Program Analysis[J]. Journal of Software, 2019, 30(1): 1-31.

[11] Evans D, Larochelle D. Improving security using extensible lightweight static analysis[J]. IEEE Software, 2002, 19(1): 42-51.

[12] Fortify Static Code Analyzer[EB/OL]. [20181206]. https://software.microfocus.com/zh-cn/products/static-code-analysis-sast/overview.

[13] Checkmarx - Application Security Testing and Static Code Analysis[EB/OL]. https://scan.coverity.com/.

[14] Coverity Scan - Static Analysis[EB/OL]. https://www.checkmarx.com/.

[15] Cppcheck - A tool for static C/C++ code analysis[EB/OL]. http://cppcheck.sourceforge.net/.

[16] Chen J, Diao W, Zhao Q, et al. IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing[C]. Proceedings of the 2018 Network and Distributed System Security Symposium, 2018.

[17] Yun I, Min C, Si X, et al. APISan: Sanitizing API Usages through Semantic

Cross-Checking[C]. Proceedings of the 25th USENIX Security Symposium, 2016: 363-378.

[18] Jang J, Agrawal A, Brumley D. ReDeBug: Finding unpatched code clones in entire OS distributions[C]. 33rd IEEE Symposium on Security and Privacy, 2012: 48-62.

[19] Duan R, Bijlani A, Xu M, et al. Identifying open-source license violation and 1-day security risk at large scale[C]. Proceedings of 24th ACM SIGSAC Conference on Computer and Communications Security, 2017: 2169-2185.

[20] Yamaguchi F, Golde N, Arp D, et al. Modeling and Discovering Vulnerabilities with Code Property Graphs[C]. IEEE Symposium on Security and Privacy, 2014: 590-604.

[21] 邹权臣, 张涛, 吴润浦, et al. 从自动化到智能化:软件漏洞挖掘技术进展[J]. 清华大学学报(自然科学版), 2018: 1-16.

ZHOU Q, ZHANG T, WU R, et al. From automation to intelligence: Survey of research on vulnerability discovery techniques[J]. Journal of Tsinghua University (Science and Technology), 2018: 1-16.

[22] Xu X, Liu C, Feng Q, et al. Neural network-based graph embedding for cross-platform binary code similarity detection[C]. Proceedings of 24th ACM SIGSAC Conference on Computer and Communications Security, 2017: 363-376.

[23] Gang Z, Jeff H. DeepSim: deep learning code functional similarity[C]. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018: 141-151.

[24] 国家信息安全漏洞库(China National Vulnerability Database of Information Security)[EB/OL]. [20181206]. http://www.cnnvd.org.cn/.

[25] National Vulnerability Database[EB/OL]. https://nvd.nist.gov/.