软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(11):3421−3435 [doi: 10.13328/j.cnki.jos.005833]

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

函数级数据依赖图及其在静态脆弱性分析中的应用*

陈 千[1,2,3], 程 凯[1,2], 郑尧文[1,2], 朱红松[1,2], 孙利民[1,2]

[1](物联网信息安全技术北京市重点实验室(中国科学院 信息工程研究所),北京 100093)

[2](中国科学院大学 网络空间安全学院,北京 100049)

[3](北京奇虎科技有限公司,北京 100015)

通讯作者: 朱红松, E-mail: zhuhongsong@iie.ac.cn

摘 要: 数据流分析是二进制程序分析的重要手段,但传统数据依赖图(DDG)构建的时间与空间复杂度较高,限制了可分析代码的规模.提出了函数级数据依赖图(FDDG)的概念,并设计了函数级数据依赖图的构建方法.在考虑函数参数及参数间相互依赖关系的基础上,将函数作为整体分析,忽略函数内部的具体实现,显著缩小了数据依赖图规模,降低了数据依赖图生成的时空复杂度.实验结果表明,与开源工具 angr 中的 DDG 生成方法相比,FDDG 的生成时间性能普遍提升了 3 个数量级.同时,将 FDDG 应用于嵌入式二进制固件脆弱性分析,实现了嵌入式固件脆弱性分析原型系统 FFVA,在对 D-Link、NETGEAR、EasyN、uniview 等品牌的设备固件分析中,发现了 24 个漏洞,其中 14 个属于未知漏洞,进一步验证了 FDDG 在静态脆弱性分析中的有效性.

关键词: 数据流分析;函数级数据依赖图;脆弱性分析;固件

中图法分类号: TP311

中文引用格式: 陈千,程凯,郑尧文,朱红松,孙利民.函数级数据依赖图及其在静态脆弱性分析中的应用.软件学报,2020,31(11):3421−3435. http://www.jos.org.cn/1000-9825/5833.htm

英文引用格式: Chen Q, Cheng K, Zheng YW, Zhu HS, Sun LM. Function-level data dependence graph and its application in static vulnerability analysis. Ruan Jian Xue Bao/Journal of Software, 2020,31(11):3421−3435 (in Chinese). http://www.jos.org.cn/1000-9825/5833.htm

Function-level Data Dependence Graph and Its Application in Static Vulnerability Analysis

CHEN Qian[1,2,3], CHENG Kai[1,2], ZHENG Yao-Wen[1,2], ZHU Hong-Song[1,2], SUN Li-Min[1,2]

[1](Beijing Key Laboratory of IOT Information Security Technology (Institute of Information Engineering, Chinese Academy of Sciences), Beijing 100093, China)

[2](School of Cyber Security, University of Chinese Academy of Science, Beijing 100049, China)

[3](Qihoo 360 Technology Co. Ltd., Beijing 100015, China)

Abstract: Data flow analysis plays an important role in binary code analysis. Due to consuming too much time and space, constructing the traditional data dependence graph (DDG) limits the size of the analyzed code thoroughly. This study introduces a novel graph model, function-level data dependence graph (FDDG), and proposes a corresponding construction method. The key insights behind FDDG lie in the following two points. First, FDDG focuses on the relationships between function parameters; Second, FDDG treats a function as a whole and ignores the details inside the function. As a result, the size of the data dependence graph is reduced significantly. Also, the time and space are saved greatly. The experimental results show the time performance of the method is improved by about three orders of

* 基金项目: 国家自然科学基金(U1766215, U1636120); 中国科学院信息工程研究所国际合作项目(Y7Z0451104); 国家电网公司科学技术项目(52110417001B)

Foundation item: National Natural Science Foundation of China (U1766215, U1636120); International Cooperation Project of Institute of Information Engineering, Chinese Academy of Sciences (Y7Z0451104); Science and Technology Project of State Grid Corporation of China (52110417001B)

收稿时间: 2018-10-09; 修改时间: 2018-11-29, 2019-01-18; 采用时间: 2019-02-28

magnitude compared to the method in angr. As an instance, FDDG is employed to analyze the vulnerability of embedded firmwares, and a firmware vulnerability analysis prototype system called FFVA is implemented. The implemented FFVA system is used to analyze firmwares from real embedded devices, and find a total of 24 vulnerabilities in the devices from D-Link, NETGEAR, EasyN, uniview, and so on, among which 14 are unknown vulnerabilities, thus validating the effectiveness of function-level data dependence graph in static vulnerability analysis.

Key words: data flow analysis; function-level data dependence graph; vulnerability analysis; firmware

数据流分析是一种用来获取相关数据沿着程序执行路径流动的信息分析技术[1],分析对象是程序执行路径上的数据流动或可能的取值,最初被广泛应用于程序的编译优化过程.作为程序静态分析的辅助支撑技术,该技术也逐渐被运用于程序切片、变量别名分析、信息泄露检测等中.

由于程序数据流的某些特点或性质与程序漏洞紧密相关,也可以将数据流分析技术用于程序漏洞的检测中[2].比如对 SQL 注入、系统命令注入等漏洞进行检测,主要关心数据的流动或数据的性质,即需要知道某个变量的取值是否来源于某个非可信的数据源.文献[3]采用上下文敏感的数据流分析方式来发现 Web 应用中的脆弱点,适用于 SQL 注入、XSS 和命令注入等污点类漏洞的检测.而针对缓冲区溢出漏洞的检测,还需要知道某个程序变量可能的取值范围.目前,数据流分析已被广泛应用于各种面向源代码的漏洞检测分析工具中,典型的工具包括 FindBugs[4]、Fortify SCA[5]、Coverity[6]、IBM Appscan Source Edition[7]和 Pinpoint[8]等.类似地,数据流分析技术也可以用于二进制代码的漏洞检测与分析中,如 CodeSonar[9].

在进行数据流分析时,由于构建传统数据依赖图(data dependence graph,简称 DDG)的时间与空间复杂度高,严重限制了可分析代码的规模.如何对数据依赖图进行改进,对于提升静态分析效率具有重要意义.本文提出了函数级数据依赖图(function-level data dependence graph,简称 FDDG)的概念,设计了 FDDG 的构建方法,在保证数据流准确性的同时,缩小数据依赖图的规模,显著降低数据依赖图的构建时间.实验结果表明:基于 FDDG 对嵌入式固件进行脆弱性分析,在保证分析能力的情况下,能够极大地提升分析效率.

1 相关工作

近年来,数据依赖分析成为程序自动并行化、运行时调度和性能优化等应用中的研究热点.在程序自动并行化的研究中,数据依赖分析是指令可并行性判别的重要方法.Ye 等人[10]通过离线分析代码块间数据依赖关系,从而节省运行时的计算资源和开销.Abbas 等人[11]通过自适应采样方式生成近似的数据依赖图,避免通过代码插装方式带来的运行时性能损失过高问题,但该方式会带来一定的精度损失.而 Li 等人[12]则提出了一种快速数据依赖分析技术,允许跳过循环中重复执行的内存操作,在保证精度的同时,降低数据依赖分析的时间开销.上述方法虽然降低了数据依赖分析的时间开销,但构建得到的数据依赖图包含大量与脆弱性分析无关的变量信息,大大降低了变量回溯的效率,不适用于程序静态脆弱性分析.

在数据流分析中,函数摘要技术是过程间分析常用的技术[13].通过为函数生成摘要信息,避免在不同上下文对同一函数进行多次分析,从而简化过程间分析的复杂度.Rountev 等人[14]采用生成库代码摘要的方式来加快数据流分析,但其时间复杂度较高,同时未考虑存在间接调用的情形.而 Tang 等人[15]提出了一个用于程序分析的 TAL(tree-adjoining language)可达性框架,在降低开销的同时,解决了存在间接调用时库代码的摘要生成问题.利用函数摘要技术,可以在一定程序上能够降低数据依赖分析的时间开销.但该技术与数据依赖图生成过程无关,不能从结构上优化数据依赖图.

也有一些学者对数据依赖图进行了简单改进,以满足特定应用场景的需求.杨轶等人[16]在研究恶意代码相似性比较方法时,对传统的数据依赖图进行了扩展——在数据依赖图中增加了恶意代码行为统计信息记录;之后,通过对依赖图进行预处理以缩减图的规模;然后,基于扩展的依赖图进行相似性比较,以提升结果的准确性,同时避免由于混淆、加壳等反制手段引起的干扰.

本文从依赖图构建的角度,提出了一种新的数据依赖图——函数级数据依赖图(FDDG).其核心思想是:在考虑函数参数及参数间相互依赖关系的基础上,将函数作为整体进行分析,忽略函数内部的具体实现,从结构上

重新定义数据依赖图,从而达到缩减数据依赖图规模、降低数据依赖图生成的时空复杂度的目的.

2 函数级数据依赖图

2.1 数据流分析

数据流分析主要关注程序执行路径上的数据流流动或可能的取值,其目的是在程序的一定范围内,确定变量定义和引用间的关系.变量定义指程序中的某条语句对变量进行赋值,除定义之外的其他变量出现称为变量引用.

数据依赖表示程序中对某变量进行引用的语句(或基本块)对定义该变量语句的依赖,即一种“定义-引用”依赖关系(如图 1 所示).若 *a* 和 *b* 分别是程序控制流图中的两个节点,*v* 为程序中的一个变量,当节点 *a* 和 *b* 满足以下条件时,称节点 *b* 关于变量 *v* 直接数据依赖于节点 *a*:(1) 节点 *a* 对变量 *v* 进行定义;(2) 节点 *b* 中引用了变量 *v*;(3) 节点 *a* 到节点 *b* 之间存在一条可执行路径,且在此路径上不存在其他语句对变量 *v* 进行定义.

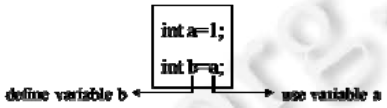


Fig.1 Example of variable definition and use

图 1 变量定义与引用示例

数据依赖图 DDG[17]是一种表示程序语句(或基本块)间数据依赖关系的有向图.根据数据依赖关系可以构建数据依赖图 *DDG*=(*V*,*E*),其中,*V* 表示程序中所有语句对应节点的集合,*E* 表示语句之间数据依赖关系的集合.图 2 为代码片段及其对应的数据依赖图示例.对于二进制代码而言,数据依赖图中的节点均为指令,因此本文将这种传统的数据依赖图称为指令级数据依赖图.

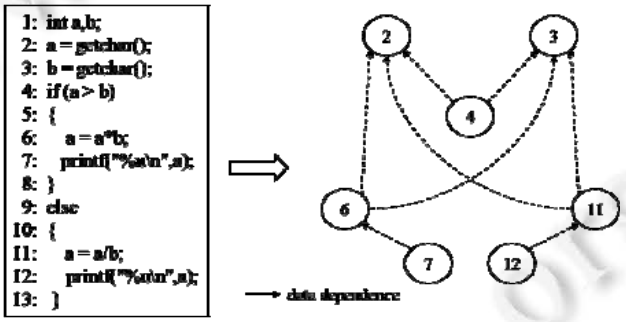


Fig.2 Example of code snippet and its data dependence graph

图 2 代码片段及其数据依赖图示例

2.2 函数级数据依赖图定义

指令级数据依赖图在数据流分析中应用广泛,但其资源开销限制了可分析代码的规模和分析的效率.鉴于此,本文提出函数级数据依赖图 *FDDG*=(*FV*,*FE*)的概念,通过降低构图粒度提升分析效能.与指令级数据依赖图不同的是,*FV* 表示程序中函数对应节点的集合,*FE* 表示函数之间参数依赖关系的集合.

2.2.1 函数级数据依赖图中的节点

在函数级数据依赖图中,节点代表函数,其记录了函数参数或返回值之间的“定义-引用”关系,同时包含函数参数的大小、函数名等信息.节点可以采用(*func_name*,⟨[*def_arg*1,*def_arg*2,…],[*use_arg*1,*use_arg*2,…]⟩)这种形式来表示,其中,*func_name* 为函数名称,⟨[*def_arg*1,*def_arg*2,…],[*use_arg*1,*use_arg*2,…]⟩表示函数参数或返回值之间的“定义-引用”关系,[*def_arg*1,*def_arg*2,…]为定义参数的集合,[*use_arg*1,*use_arg*2,…]为引用参数的集合.

对于常见的函数,均可以采用上述形式进行表示.根据函数功能的不同,可以分为如下几类.

(1) *retval*=*strstr*(*arg*1,*arg*2,…)类函数:这类函数实质上是对返回值 *retval* 进行定义,而对 *retval* 的定义又引用了函数参数 *arg*1,*arg*2,….由于返回值 *retval* 可能会在后续的数据流中用到,因此可以表示为(*strstr*,⟨[*retval*],[*arg*1,*arg*2,…]⟩),其中,[*retval*]为定义参数的集合,[*arg*1,*arg*2,…]为引用参数的集合.

(2) *memcpy*(*arg*1,*arg*2,*arg*3,…)类函数:这类函数实质上是对某个参数如 *arg*1 进行定义,而对该参数的定义又引用了其他参数如 *arg*2,*arg*3,…;同时,这类函数没有返回值或返回值不会体现在后续的数据流中,因此可以表示为(*memcpy*,⟨[*arg*1],[*arg*2,*arg*3,…]⟩),其中,[*arg*1]为定义参数的集合,[*arg*2,*arg*3,…]为引用参数的集合.

(3) *retval*=*strcpy*(*arg*1,*arg*2,…)类函数:这类函数实质上是对某个参数如 *arg*1 进行定义,而对该参数的定义又引用了其他参数如 *arg*2,…;同时,也对返回值 *retval* 进行了定义,而且返回值可能会体现在后续的数据流中,因此可以表示为(*strcpy*,⟨[*arg*1,*retval*],[*arg*2,…]⟩),其中,[*arg*1,*retval*]为定义参数的集合,[*arg*2,…]为引用参数的集合.

(4) *system*(*arg*1,…)类函数:这类函数只是引用函数参数如 *arg*1,…;同时,该函数没有返回值或返回值不会体现在后续的数据流中,因此可以表示为(*system*,⟨[·],[*arg*1,…]⟩),其中,[·]为定义参数的集合,[*arg*1,…]为引用参数的集合.

除了程序中的函数外,也可以将程序中的某些语句片段(或某条语句)抽象成函数,进而表示为 FDDG 中的节点,最典型的就是实现循环拷贝的语句片段.循环拷贝是指在某个循环体内向不断改变的地址中写入数据,文献[18]给出了识别循环拷贝代码片段的方法.本文方法将循环拷贝代码片段直接抽象成函数,并将存在循环依赖关系的变量作为函数的参数,同时记录参数之间的"定义-引用"关系.在图 3 中,将实现循环拷贝的代码片段抽象成 *loop_copy* 函数,同时保存循环间隔和循环终止条件等信息.这种处理方法拓展了函数级数据依赖图的表达能力.

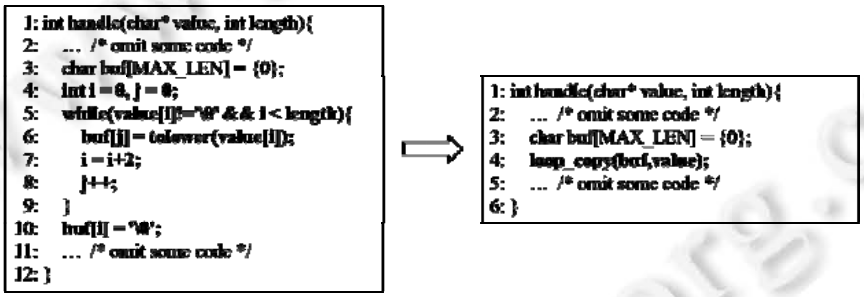


Fig.3 Example of abstracting a code snippet into a function

图 3 代码片段抽象为函数示例

2.2.2 函数级数据依赖图中的边

函数级数据依赖图中的边表示函数之间参数的依赖关系,主要包括"引用"和"消灭"两种关系.若 *a* 和 *b* 分别为程序 *P* 中的两个函数,*v* 为两个函数共同的参数,这里将函数的返回值也看作函数的参数,当 *a* 和 *b* 满足以下条件时,称函数 *b* 和函数 *a* 之间关于参数 *v* 存在"引用"关系.

(1) 函数 *a* 对参数 *v* 进行定义.

(2) 函数 *b* 引用参数 *v*.

(3) 函数 *a* 到函数 *b* 之间存在一条可执行路径,且此路径上不存在其他对参数 *v* 的定义.

类似地,当 *a* 和 *b* 满足下列条件时,称函数 *b* 和函数 *a* 之间关于参数 *v* 存在"消灭"关系.

(1) 函数 *a* 对参数 *v* 进行定义.

(2) 函数 *b* 对参数 *v* 进行重新定义.

(3) 函数 *a* 到函数 *b* 之间存在一条可执行路径,且此路径上不存在其他对参数 *v* 的定义.

图 4 所示为代码片段及其对应的函数级数据依赖图示例.其中,函数 *fread*(·)对参数 *buf* 进行定义,函数 *base*64_*decode*(·)引用了参数 *buf*,同时,两个函数之间存在一条可执行路径,因此,函数 *base*64_*decode*(·)和 *fread*(·)之间关于参数 *buf* 存在"引用关系".同理,函数 *strstr*(·)和 *base*64_*decode*(·)、函数 *printf*(·)和 *copy_lower*(·)之间关于参数 *cmd* 均存在"引用"关系.另外,函数 *copy_lower*(·)对参数 *cmd* 进行重新定义,而函数 *base*64_*decode*(·)和 *copy_lower*(·)之间存在一条可执行路径,且路径上没有对参数 *cmd* 的其他定义,因此,函数 *copy_lower*(·)和 *base*64_*decode*(·)之间关于参数 *cmd* 存在"消灭"关系.

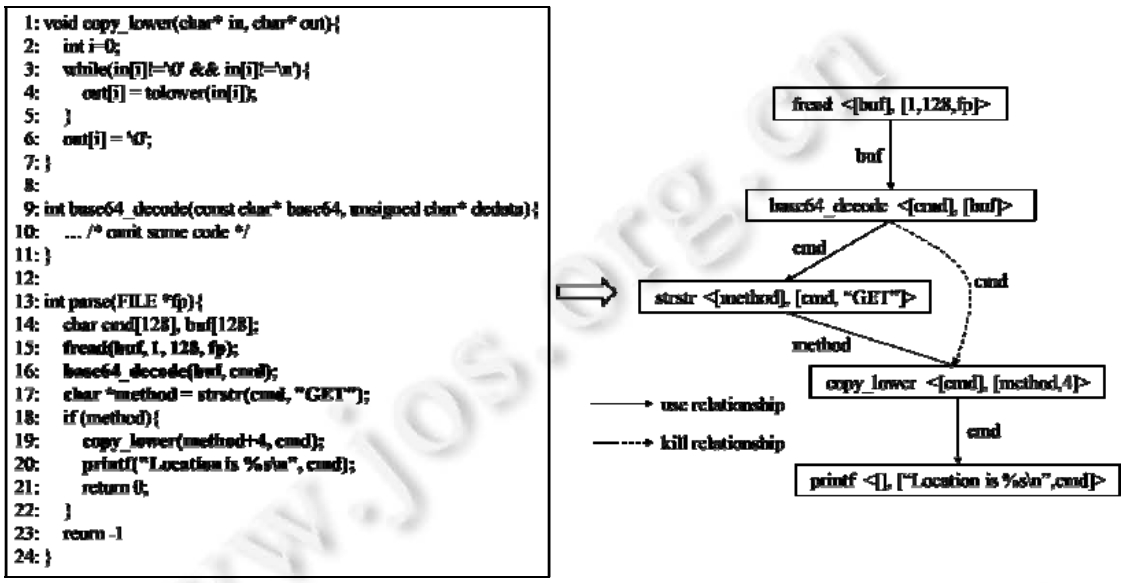


Fig.4 Example of code snippet and its function-level data dependence graph

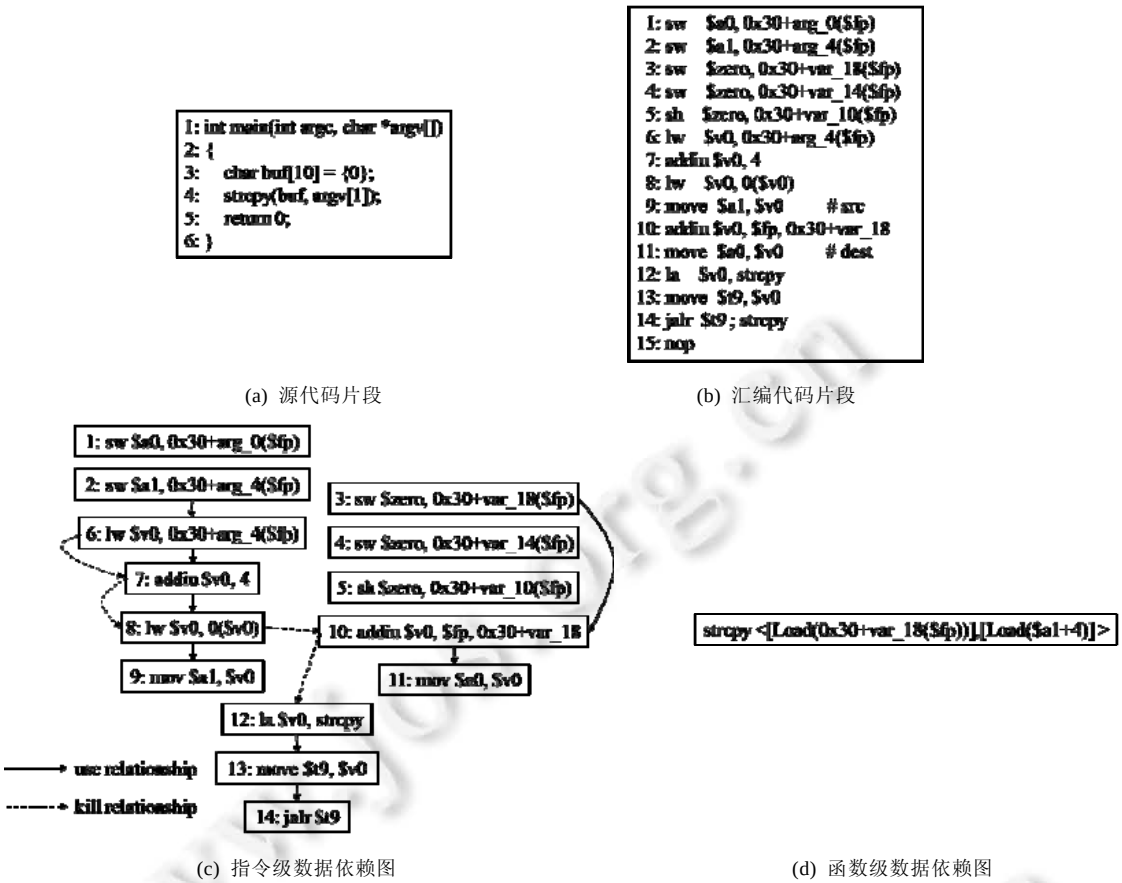
图 4 代码片段及其函数级数据依赖图示例

2.3 函数级数据依赖图特性分析

指令级数据依赖图 DDG 以语句为节点,在构建过程中会考虑所有的变量,即在所有相关变量之间建立"定义-引用"依赖关系,关系精确、完整,适用于程序切片、程序数据结构恢复等应用,但在程序静态脆弱性分析方面却不高效.一方面,DDG 将变量与变量所处的语境(如变量为函数 *strcpy*(·)的参数)分开,变量的上下文信息变得不明确;另一方面,考虑所有变量会造成 DDG 的规模极大,而很多变量信息对脆弱性分析没有帮助,导致后续的数据依赖分析效率低下,浪费计算时间和空间.

函数级数据依赖图 FDDG 以函数为节点,在构建过程中仅考虑函数之间参数的相互依赖关系,得到的依赖图粒度较"粗",难以用于程序切片.但 FDDG 将变量与其所处的语境保存在一起,同时保留了变量路径分析所需要的全部信息,适用于程序静态脆弱性分析,如对缓冲区溢出和命令注入等漏洞进行检测.此外,由于 FDDG 的规模比较小,数据流比较清晰,因此对参数进行回溯效率很高.

图 5 所示为针对同一段汇编代码构建的两种数据依赖图示例.由图 5 可知,指令级数据依赖图 DDG 的规模明显大于函数级数据依赖图 FDDG.同时,在 DDG 中,函数与函数参数是分离的;而 FDDG 则直接将函数参数和函数保存在一起.此外,针对图 5(a)中的 *strcpy*(*buf*,*argv*[1])语句,在进行静态脆弱性分析时,需要对参数 *buf* 和 *argv*[1]进行溯源,以获取参数指向的内存空间大小、参数来源等信息.基于图 5(c)所示的 DDG 进行回溯,查找路径包括"11→10"和"9→8→7→6→2";而在图 5(d)所示的 FDDG 中,函数参数中已经包含了所需的信息,不用进行回溯.



(a) 源代码片段 (b) 汇编代码片段

(c) 指令级数据依赖图 (d) 函数级数据依赖图

Fig.5 Example of two kinds of data dependence graph

图 5 两种数据依赖图示例

与指令级数据依赖图相比,函数级数据依赖图中的数据流更清晰直观,对变量的回溯效率更高,同时保存了很多额外信息如变量大小、路径约束等,适用于程序静态脆弱性分析,尤其是针对“污点类”漏洞[19]的检测.需要说明的是,由于函数级数据依赖图忽略了一些变量和指令级的信息,如与函数参数无关的局部变量,在编译优化、程序切片、指针别名分析等需要完整数据流信息的应用场景中并不适用.

3 函数级数据依赖图构建

为保证数据流分析的准确率,通常需要进行流敏感的分析,即考虑程序语句可能的执行顺序,因而构建 FDDG 需要控制流分析的支持.

控制流分析基于程序的控制流图(control flow graph,简称 CFG),用于确定一个程序的控制结构,可用于探究程序可能的执行路径、提取循环分析结构等.在构建程序控制流图的过程中,记录函数参数之间的“定义-引用”关系以及函数名、参数大小等信息,并将这些信息保存在对应的控制流图节点中.

在得到函数调用图和程序控制流图后,构建函数级数据依赖图 FDDG 的方法如下述算法 1 表述,具体流程如下.

(1) 按后序方式对函数调用图进行遍历.

(2) 针对函数调用图中的每个函数,对其控制流图进行层次遍历.

(3) 针对单个函数控制流图中的节点,获取节点中保存的函数参数关系记录,根据函数之间参数的依赖关

系,先后在当前函数和全局变量定义列表中查找函数参数定义中引用参数的定义,若查找成功,则将对参数进行定义的函数和对其引用参数进行定义的函数作为节点、参数之间的依赖关系作为边添加到数据依赖图中,然后跳转到步骤(5).

(4) 将对该参数进行定义的函数作为节点添加到数据依赖图中.若在当前函数和全局变量定义列表中均无法找到该定义中引用参数的定义,则将该参数的定义更新到所有直接调用当前函数的父函数中.

(5) 若当前函数的控制流图中某个节点的出度为 0,则需要将函数内对函数参数的定义更新到所有直接调用该函数的父函数中;同时,将对全局变量的定义更新到全局变量定义列表中.

(6) 若函数控制流图遍历完毕,则跳转到步骤(7);否则跳转到步骤(2).

(7) 若函数调用图遍历完毕则停止,否则跳转到步骤(1).

其中,在步骤(3)中,针对某个关键参数的定义,如果该定义中引用参数的类型为整数类型,则不查找引用参数的定义,只有当其类型为堆/栈变量、返回值、函数参数或全局变量时才进行查找.此外,在全局变量定义列表中查找其引用参数的定义时,如果存在多个对引用参数的定义,则在对关键参数进行定义的函数和每个对其引用参数进行定义的函数之间都建立一条边.在步骤(5),由于函数内部对函数参数的定义会影响父函数的行为,当遇到出度为 0 的节点时,需要更新函数参数定义,同时也要更新全局变量定义列表.

算法 1. 函数级数据依赖图构建算法.

输入:函数调用图 CG,控制流图 CFG.

输出:函数级数据依赖图 FDDG.

```
1  GenerateFDDG(CG,CFG);
2  for each node n_i in CG
3    for each block_node b_i in CFG_{n_i}
4      for each dst_def_info dd_i in b_i
5        sd←search_src_def(dd_i);   /*在局部或全局变量定义列表中查找引用参数的定义*/
6        if sd
7          FDDG.add_edge(sd,dd_i);
8        else
9          FDDG.add_node(dd_i);
10         save_forward_def_info(n_i,dd_i);   /*保存参数用于后续在父节点中继续查找*/
11       end if
12       save_def_info(n_i,dd_i);
13     end for
14     for each node s_i in successors(b_i)
15       add_def_info(s_i);   /*将 b_i 中的信息添加到后继节点中*/
16     end for
17     if out_degree_bi==0
18       for each node p_i in predecessors(n_i)
19         update_def_infos(p_i);   /*将 n_i 中的信息更新到父节点中*/
20       end for
21     end if
22   end for
23   for each node p_i in predecessors(n_i)
24     search_forward_def(p_i);   /*对于未查找到的定义,在父节点中进行查找*/
25   end for
```

26 **end for**

在构建 FDDG 时,过程间分析同样采用摘要分析的方式.对于系统函数或库函数,采用“符号摘要”方法来更好地描述函数的行为及语义[20];对于用户自定义函数,采用普通的函数摘要技术来构建其输入状态和输出状态等信息[21],也可以采用其他更精确的函数摘要技术,如基于 TAL(tree-adjoining-language)可达性的摘要技术[15].需要说明的是,FDDG 本质上还是一种数据依赖图,其与函数摘要技术属于两个不同的范畴.FDDG 中节点的处理方式,重点关注的是参数或返回值之间的“定义-引用”关系,而函数摘要技术则无法直接得到这种关系.

4 函数级数据依赖图在静态脆弱性分析中的应用

函数级数据依赖图 FDDG 保存了函数参数及函数之间参数的依赖关系,所以最直观的应用是分析函数参数在程序调用过程中的传递路径.本节将以固件中二进制程序的静态脆弱性分析为例分析 FDDG 的可用性.

4.1 二进制程序静态脆弱性分析

二进制程序静态脆弱性分析的基本流程如图 6 所示.其中,通过控制流分析可以得到程序的控制流图 CFG 和函数调用图(call graph,简称 CG),通过数据流分析可以得到数据依赖图 DDG.在此基础上,根据漏洞模式或规则对程序中的潜在脆弱点进行分析,以判断程序是否存在安全缺陷.

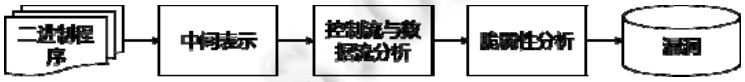


Fig.6 Flow of static vulnerability analysis

图 6 静态脆弱性分析流程

在进行静态代码分析时,由于存在函数间接调用、指针别名等机制,构建 DDG 的过程中会出现信息损失.基于 DDG 进行静态脆弱性分析,实质是对 DDG 中包含的信息进行挖掘,从而判断是否存在漏洞.整个过程可以用图 7 所示的模型表示.为了便于描述,将与脆弱点处的变量存在依赖关系的其他变量称为相关变量,这些变量所蕴含的信息称为有效信息,记为 A;而与脆弱点处的变量不存在依赖关系的其他变量称为无关变量,其信息为无效信息,记为 B(或Θ–A,Θ表示全部信息).具体地,在对脆弱点进行分析时,除了关心数据的流动或数据的性质,既需要知道某个变量的取值是否来自非可信的数据源,还需要知道某个变量可能的取值范围.因此,在对脆弱点处的变量进行回溯时,与其存在依赖关系的相关变量的信息包括变量取值、大小、来源及约束等都属于有效信息.以图 5 所示的代码片段为例,在图 5(a)中的脆弱点处包含 *buf* 和 *argv*[1]两个变量,从脆弱点向上回溯时,只要与变量 *buf* 或 *argv*[1]存在依赖关系的变量都属于相关变量,这些变量的取值、大小、来源及约束等信息都属于有效信息.而变量 *argc* 与脆弱点的分析无关,故其属于无关变量,其信息为无效信息.

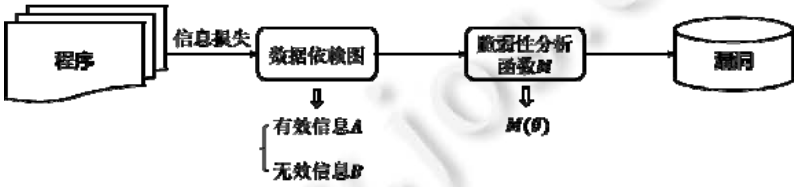


Fig.7 A vulnerability mining model based on data dependence graph

图 7 基于数据依赖图的漏洞挖掘模型

由图 7 可知,脆弱性分析的结果可以看作是将脆弱性分析函数 M 作用于信息θ上的输出,记为 $M(\theta)$.通常,包含的有效信息 A 越多,检测出漏洞的可能性越大,$M(\theta)$越接近 $M(\Theta)$;包含的无效信息 B 越多,检测不出漏洞的可能性也越大,$M(\theta)$越接近 $M(\Phi)$(Φ表示空信息).在理想情况下,脆弱性分析函数 M 可以根据有效信息 A 检测出所有的漏洞,即 $M(A)=M(\Theta)$.而实际会存在根据有效信息 A 检测不出漏洞的情形,和根据无效信息 B 检测出漏洞

的情形,即所谓的漏报和误报.由表 1 可知,脆弱性分析的结果与包含的有效信息量和漏洞分析函数 M 这两个因素有关.在脆弱性分析函数 M 相同的前提下,基于数据依赖图进行脆弱性分析的结果只取决于图中所蕴含的有效信息量.

Table 1 Relationship between the results of vulnerability analysis and information

表 1 脆弱性分析结果与信息的对应关系

$M(\theta)$	有效信息 A	无效信息 B
检测出漏洞	$M(A)$	$M(\Theta)-M(B)$
检测不出漏洞	$M(\Theta)-M(A)$	$M(B)$

在进行脆弱性分析时,以“污点类”漏洞为例,当对脆弱点处的变量进行回溯时,需要分析与其存在依赖关系的相关变量,获取相关变量的取值、大小、约束以及是否来自非可信的数据源等信息,这些信息构成了有效信息.由函数级数据依赖图的定义和构建方法可知,函数级数据依赖图是对传统数据依赖图进行进一步精简与抽象.两者的主要区别在于:DDG 中的节点表示指令,而 FDDG 中的节点表示函数.在进行依赖分析时,无论是指令还是函数,主要关注其中包含的变量及其依赖关系.根据 FDDG 节点的定义,其节点既可以表示系统函数如 *strcpy*(·),也可以表示用户自定义函数(即非系统函数)如 *custom_copy*(·).在对程序进行分析时,针对系统函数,由于其为一个最小单元,FDDG 中的处理方式与 DDG 中的一致,故不存在信息损失.而针对用户自定义函数,在构建 FDDG 时可能会将其当作一个整体而忽略函数内部的信息.同样,由 FDDG 节点的定义可知,当某个函数能够用节点表示时,说明该函数参数或返回值之间的依赖关系是明确的.因而没有必要对函数内部进行分析以获取这种依赖关系,损失的函数内部信息对参数依赖关系分析的影响很小.总的来说,FDDG 中包含了变量的信息及“精简”的依赖关系,通过这种“精简”的依赖关系能够获取到相关变量的信息,故 FDDG 包含的有效信息量与 DDG 相近.

4.2 固件脆弱性静态分析系统

为验证 FDDG 在程序脆弱性静态分析中的效能,本文在开源框架 angr[22,23]的基础上,结合污点分析、符号执行等技术,实现了一个固件脆弱性静态分析原型系统 FFVA.

固件通常是对嵌入式设备软件系统的一种称谓,其由固件头、启动引导程序、操作系统内核、根文件系统以及附加数据等组成.与传统软件类似,固件中也存在安全缺陷或漏洞.由于很难获取固件源码,因此,嵌入式设备脆弱性分析的重点研究对象是固件文件系统中的二进制应用程序.

在固件二进制程序中,常见的程序漏洞包括缓冲区溢出、格式化字符串漏洞、命令注入、认证缺失、硬编码/弱密钥等.通常,缓冲区溢出、格式化字符串漏洞和命令注入这 3 类漏洞的形成原因主要是与不安全函数调用有关,可以尝试基于 FDDG 对这 3 类漏洞进行检测.以 *strcpy* 函数为例,如果源缓冲区的大小超过目的缓冲区,且源缓冲区的内容为外部可控,将导致缓冲区溢出漏洞.这里,源缓冲区的大小及内容来源是评判是否存在漏洞的关键,适用于 FDDG 分析.

表 2 所示为 OWASP 嵌入式应用安全项目[24]提供的 Top 10 清单.

Table 2 Top 10 best practices for embedded application security provided by OWASP

表 2 OWASP 嵌入式应用安全 Top 10 实践清单

序号	名称
1	Buffer and stack overflow protection
2	Injection prevention
3	Firmware updates and cryptographic signatures
4	Securing sensitive information
5	Identity management
6	Embedded framework and C-based hardening
7	Usage of debug code and interfaces
8	Transport layer security
9	Data collection usage and storage—Privacy
10	Third party code and components

由表 2 可知,缓冲区溢出和命令注入漏洞对嵌入式设备的威胁较大.对这两类漏洞进行检测,FFVA 系统的主要思想如下:将这两类漏洞视为一种特殊的“Source-Sink”问题[25,26],分析程序中所有 Source 点到 Sink 点之间的数据流,然后对 Sink 点的安全性进行判断,同时进行路径敏感性分析.其中,Source 点表示程序中的外部数据输入点,如网络流读取、文件读取和用户输入等;Sink 点表示程序中可能会存在安全风险的代码片段,如对 *strcpy*(·)等不安全函数的调用(如图 8 所示).

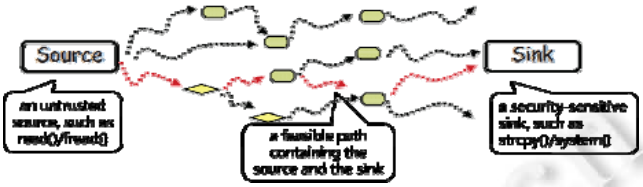


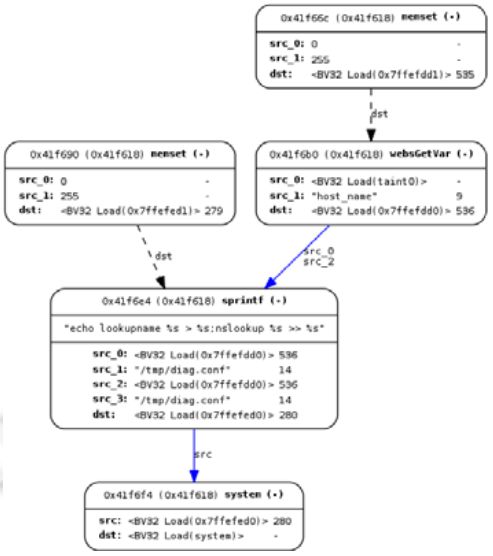
Fig.8 “Source-Sink” problem

图 8 “Source-Sink”问题

以 CVE-2017-6334 漏洞为例,在 NETGEAR DGN2200 型号的设备中存在一个命令注入漏洞,漏洞触发点位于 *diagCgiDnslookup*(·)函数中.利用 FFVA 系统对该二进制程序样本进行分析,分析流程如图 6 所示,针对 *diagCgiDnslookup*(·)函数构建得到的函数级数据依赖图如图 9(b)所示.由图 9(b)可知,在 *diagCgiDnslookup*(·)函数内部,先调用 *websGetVar*(·)函数获取请求包中 *host_name* 字段的值,然后调用 *sprintf*(·)函数进行格式化输出,之后将结果传递给 *system*(·)函数.由于未对 *host_name* 字段的值进行有效校验,故存在命令注入漏洞,其中可以将 *websGetVar*(·)函数看作 Source 点,*system*(·)函数看作 Sink 点.这种分析思路对 FDDG 和 DDG 都适用,但基于 FDDG 进行分析,其数据流更清晰,效率更高.由于指令级数据依赖图的规模比较大(包含 156 个节点、165 条边),不便展示,故这里未给出对应的指令级数据依赖图.

```
addiu   $s0, $sp, 0x238+var_218
la      $a1, aProfile_status  # "profile_status(2, \"my_lan_access\");"
la      $t9, websGetVar
move    $a0, $s3
move    $a2, $s0
jalr    $t9 ; websGetVar
addiu   $a1, (aSmb_host_name+4 - 0x490000)  # "host_name"
lw      $gp, 0x238+var_220($sp)
addiu   $s1, $sp, 0x238+var_118
la      $v1, unk_4A0000
la      $a1, unk_4A0000
addiu   $v1, (aTmpDiag_conf - 0x4A0000)  # "/tmp/diag.conf"
la      $t9, sprintf
move    $a2, $s0
sw      $v1, 0x238+var_224($sp)
addiu   $a1, (aEchoLookupname - 0x4A0000)  # "echo lookupname %s > %s;nslookup
move    $a3, $v1
sw      $s0, 0x238+var_228($sp)
jalr    $t9 ; sprintf
move    $a0, $s1
lw      $gp, 0x238+var_220($sp)
la      $t9, system
jalr    $t9 ; system
move    $a0, $s1
```

(a) 汇编代码片段



(b) 函数级数据依赖图

Fig.9 Code snippet of *diagCgiDnslookup*(·) and its function-level data dependence graph

图 9 *diagCgiDnslookup*(·)函数代码片段以及对应的函数级数据依赖图

5 实验与结果

利用 FFVA 系统,本节通过两个实验来评估函数级数据依赖图 FDDG 构建方法的效率和在固件静态脆弱性

分析中的有效性.

- FDDG 构建时间性能评估实验:对不同固件中的二进制程序分别构建 FDDG 和 DDG,对比两种依赖图的构建时间.
- FDDG 在固件静态脆弱性分析中的有效性评估实验:运用 FFVA 系统对二进制程序进行脆弱性分析,尝试发现程序中存在的安全缺陷或漏洞.

实验采用的软硬件环境见表 3.

Tabel 3 Experimental environment of software and hardware

表 3 实验软硬件环境

名称	详情
处理器型号	Intel(R) Xeon(R) CPU E5-2687W v3@3.10Ghz,10 Cores
处理器个数	4
内存大小	126GB
操作系统	Ubuntu 16.04.3 LTS x86_64
软件模块	Python 2.7.12,angr 7.7.12.16

对固件进行脆弱性分析,主要分析对象是固件中提供网络服务的程序模块,如 HTTP,UPnP 等.因为一旦在这些程序中发现漏洞,其很可能会被远程利用,进而带来严重的安全隐患.考虑到不同指令架构、不同品牌、文件大小以及是否有真实可用设备等因素,本文从路由器和摄像头设备固件中最终选取表 4 中给出的 13 个二进制程序样本作为分析对象.

Table 4 Information of binary samples

表 4 二进制样本相关信息

名称	大小(k)	指令架构	设备品牌	设备型号
wlancmd	38	MIPS	Huawei	HG532e
stupid-ftpd	53	MIPS	D-Link	DIR-882_A1
upnp	90	MIPS	Huawei	HG532e
jjhttpd	118	MIPS	D-Link	DIR820LB1
lighttpd	145	ARM	Honeywell	HIVDC-F100V
cgibin	150	ARM	D-Link	DIR-890L
cgibin	155	MIPS	D-Link	DIR-645
miniupnpd	165	MIPS	D-Link	DIR-882_A1
upnpd	213	MIPS	NETGEAR	DGN2200
setup.cgi	324	MIPS	NETGEAR	DGN1000
httpd	994	MIPS	NETGEAR	DGN2200
ipc_server	2 048	ARM	EasyN	B18EN
mwareserver	4 812	ARM	uniview	IPC_6201

5.1 FDDG构建时间性能评估

本文采用 FFVA 系统构建函数级数据依赖图,然后利用 angr 框架构建指令级数据依赖图,对两种依赖图的构建时间进行对比.为了保证两种方案分析函数的一致性,本文采用程序的入口地址作为分析的起始地址,将从入口地址可达的所有非系统函数作为待分析函数列表.

由于控制流分析采用并行化加速,CFG 的构建时间在整个脆弱性分析中的占比很小,因此,数据依赖图的构建时间将直接影响脆弱性分析的效率.由表 5 可知,与 DDG 相比,FDDG 的构建时间基本在 1min 内,除 *wlancmd* 这种分析函数个数少、内部逻辑简单的程序样本外,构建时间性能普遍提升约 3 个数量级.同时,构建时间与分析函数的数量、样本内部的流程复杂度密切相关.以两种硬件架构(ARM 和 MIPS)的 cgibin 样本为例,对 ARM 架构的 cgibin 分析了 356 个函数,多于 MIPS 架构,FDDG 生成时间也长一些,但 FDDG 的构建时间差异(3 倍)小于 DDG 的构建时间差异(11 倍).

从实现原理上看,angr 框架中的 DDG 构建方法考虑了程序中的所有变量,当一个函数内部存在循环时,由于很多变量之间存在循环依赖,故需要频繁对变量的数据依赖进行更新,耗费大量的时间.本文提出的 FDDG 构建方法仅关心与特定函数(如不安全函数)相关的参数,不考虑无关变量,分析变量大大减少,大大降低了在循环

内对变量进行数据依赖更新的频度.本文对比了两种数据依赖图中节点与边的规模,如表 6 所示,节点与边的数量分别相差约 3 个数量级.

Table 5 Comparison of time consumed in constructing the data dependence graph

表 5 数据依赖图构建时间对比

名称	大小(k)	设备型号	分析函数个数	耗费时间(min)			时间性能提升(倍)
				CFG	DDG	FDDG	
wlancmd	38	HG532e	7	0.07	0.03	0.002	**15**
stupid-ftpd	53	DIR-882_A1	24	0.16	6.99	0.005	**1 500**
upnp	90	HG532e	56	0.43	113.54	0.038	**3 000**
jjhttpd	118	DIR820LB1	45	0.47	120.14	0.027	**4 400**
lighttpd	145	HIVDC-F100V	136	0.98	468.02	0.091	**5 100**
cgibin	150	DIR-890L	356	1.24	600.72	0.115	**5 600**
cgibin	155	DIR-645	88	0.90	53.21	0.038	**1 400**
miniupnpd	165	DIR-882_A1	147	1.02	728.91	0.080	**9 100**
upnpd	213	DGN2200	114	0.65	346.93	0.108	**3 200**
setup.cgi	324	DGN1000	39	0.92	17.17	0.016	**1 050**
httpd	994	DGN2200	168	1.59	831.41	0.138	**6 000**

注:(1) DDG 对应 angr 框架中的方法,FDDG 对应本文提出的方法;(2) 时间性能提升结果为取整后的结果.

Table 6 Comparison of the data dependence graph scale

表 6 数据依赖图规模对比

名称	大小(k)	设备型号	分析函数个数	节点数		边数	
				DDG	FDDG	DDG	FDDG
stupid-ftpd	53	DIR-882_A1	24	102 037	57	120 825	12
upnp	90	HG532e	56	494 854	372	585 385	266
jjhttpd	118	DIR820LB1	45	544 243	198	612 616	157
lighttpd	145	HIVDC-F100V	136	2 318 991	428	2 646 642	491
cgibin	150	DIR-890L	356	2 845 062	1 223	3 132 910	742
cgibin	155	DIR-645	88	634 415	502	766 330	205
miniupnpd	165	DIR-882_A1	147	3 184 594	697	3 490 395	505
upnpd	213	DGN2200	114	1 850 608	1 291	2 047 508	1 242
setup.cgi	324	DGN1000	39	157 698	458	185 527	177
httpd	994	DGN2200	168	2 517 499	801	2 795 711	566

注:DDG 对应 angr 框架中的方法,FDDG 对应本文提出的方法.

总的来说,本文提出的 FDDG 构建方法极大地降低了分析所需的时间和空间开销,实现在相同计算资源条件下分析更大规模的程序,同时适用于对批量固件的快速分析.

5.2 固件脆弱性分析系统有效性评估

在前面分析中强调函数级数据依赖图保留了关键变量等有效信息,但能否在固件静态脆弱性分析中发挥效用,需要通过实验进行评估.

本文首先利用 FFVA 系统对表 4 中的二进制程序样本进行分析,然后借助逆向分析工具如 IDA Pro[27],对系统的分析结果进行人工审查,利用真实设备或 QEMU 模拟器[28,29]对脆弱点进行验证.实验结果如表 7 所示,共发现漏洞 24 个,14 个属于未知漏洞.部分未知漏洞已经提交给厂商并得到确认,剩余的未知漏洞则利用真实设备或通过固件仿真的方式进行了验证.

在表 7 中,以 Honeywell 摄像头固件中的 lighttpd 程序为例,其主要负责处理与 HTTP 协议相关的请求,在其中发现的脆弱点数量为 0.借助 IDA Pro 工具对该程序进行逆向分析,发现程序中几乎没有使用 *strcpy*(·), *sprintf*(·)等不安全函数,大部分的拷贝操作使用 *memcpy*(·)函数进行完成,并且对拷贝长度进行校验.通过对 stupid-ftpd 和 miniupnpd 程序进行逆向分析,发现其很可能是直接来源于开源软件,对脆弱点进行人工验证后没有发现漏洞.

Table 7 Results of vulnerability analysis on binary samples

表 7 二进制样本脆弱性分析结果

名称	大小(k)	设备型号	分析函数个数	耗费时间(min)	发现脆弱点数量	发现漏洞数量
wlancmd	38	HG532e	86	0.52	9	**4**
stupid-ftpd	53	DIR882_A1	56	0.46	6	**0**
upnp	90	HG532e	298	3.32	1	**1**
jjhttpd	118	DIR820LB1	281	3.25	1	**0**
lighttpd	145	HIVDC-F100V	327	2.42	0	**0**
cgibin	150	DIR-890L	445	2.77	5	**2**
cgibin	155	DIR-645	318	2.30	7	**4**
miniupnpd	165	DIR-882_A1	229	2.03	2	**0**
upnpd	213	DGN2200	268	4.47	7	**0**
setup.cgi	324	DGN1000	732	11.54	19	**5**
httpd	993	DGN2200	796	30.72	14	**2**
ipc_server*	2 048	B18EN	383	3.62	11	**5**
mwareserver*	4 812	IPC_6201	430	5.67	10	**1**

注:针对二进制样本 ipc_server 和 mwareserver,仅选取了部分函数进行分析.

由表 7 可知,采用 FFVA 系统对二进制程序进行静态脆弱性分析时,其结果同样存在误报.从代码分析的角度,存在误报的主要原因是由于函数回调机制和间接调用的存在,导致进行数据依赖分析时会出现“断链”现象,即从潜在脆弱点无法直接回溯到外部输入点如 recv/recvfrom 等,进而无法判断参数的来源,造成对潜在脆弱点安全评估的不准确.此外,函数内由于不同的执行路径可能会对应不同返回值,对多个返回值的处理不当,也会对数据依赖分析造成影响.

在表 7 中,分析二进制程序所耗费的时间包括程序分析和结果可视化两部分.以 NETGEAR httpd 为例,由于该程序内部函数较多且流程比较复杂,整体耗费时间约为 30min,但真正进行程序分析所耗费的时间大约为 7min,占比为 23.3%.二进制程序分析所耗时间占比如图 10 所示,其中大部分分析样例的整体耗费时间低于 5min,而对于耗时超过 5min 的分析样例,真正进行程序分析所耗费的时间在整体中的占比均低于 35%.

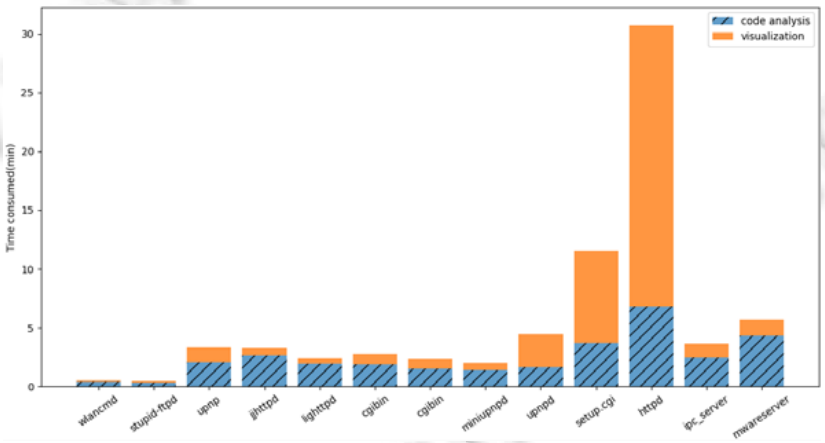


Fig.10 Proportion of time consumed in analyzing vulnerability of binary samples

图 10 二进制样本脆弱性分析所耗时间占比

目前,在实验部分没有提供基于两种数据依赖图进行漏洞挖掘的对比结果,原因如下:(1) 针对部分二进制程序样本,构建 DDG 的时间与空间复杂度过高(单个程序样本耗费的分析时间超过 13 个小时,内存消耗高达 84G),而构建 FDDG 耗费的时间在数分钟内,内存消耗在 5G 左右;(2) 基于 DDG 对与脆弱点相关的变量进行回溯的效率很低.以图 5 为例,在对图 5(a)中,*strcpy*(·)函数的参数 *buf* 和 *argv*[1]进行溯源时,在 DDG(图 5(c))中的查找路径包括“11→10”和“9→8→7→6→2”,而在 FDDG(图 5(d))中则不用进行回溯,因为函数参数中已经包含了所需的信息.在对真实程序样例进行分析时,由于 DDG 的规模约为 FDDG 的 1 000 倍,两者之间的差距会更加明显.

鉴于以上两点,基于 DDG 进行静态脆弱性分析的效率非常低,故没有提供基于 DDG 进行漏洞挖掘的实验结果.

综上,利用 FFVA 系统对固件进行静态脆弱性分析,虽然结果存在误报,但其时间开销较低;同时能够发现程序中存在的安全缺陷或漏洞,表明了函数级数据依赖图在静态脆弱性分析中的有效性.

6 总结与展望

本文提出了函数级数据依赖图 FDDG 的概念,同时设计了 FDDG 的构建方法.与指令级数据依赖图相比,该图的粒度更“粗”,数据流更清晰,同时包含丰富的变量及语义信息.进一步,基于 angr 框架,在 FDDG 的基础上实现了一个固件脆弱性静态分析原型系统 FFVA.实验结果表明:FDDG 能够高效地应用在程序静态脆弱性分析中,并在 D-Link、NETGEAR、EasyN、uniview 等品牌的设备中发现了 24 个漏洞,其中 14 个为未知漏洞.本文验证了 FDDG 在静态分析中的有效性,但对 FDDG 是否会因为忽略信息而导致脆弱性分析不全的问题只做了定性分析,后续将尝试通过模型方法论证.

References:

[1] Aho AV, Lam MS, Sethi R, *et al*. Compilers: Principles, Techniques, and Tools. 2rd ed., Boston: Addison-Wesley Longman Publishing Co. Inc., 2006.

[2] Wu SZ, Guo T, Dong GW. Software Vulnerability Analyses. Beijing: Science Press, 2014 (in Chinese).

[3] Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting Web application vulnerabilities. In: Proc. of the IEEE S&P. 2006. 258−263.

[4] The findbugs Java static checker. 2015. http://findbugs.sourceforge.net/

[5] The HP Fortify Static Checker. 2016. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

[6] The coverity code checker. 2016. http://www.coverity.com/

[7] IBM appscan. 2016. https://www.ibm.com/security/application-security/appscan

[8] The sourcebrella pinpoint. 2016. https://www.sourcebrella.com/pinpoint/

[9] The grammatech codesonar static checker. 2016. https://www.grammatech.com/codesonar-sast-binary

[10] Ye JM, Chen J, Chen T, *et al*. Offline data dependence analysis to facilitate runtime parallelism extraction. In: Proc. of the Computational Science and Engineering, 2014. 698−703.

[11] Abbas MM, Elmahdy A. Approximate data dependence graph generation using adaptive sampling. In: Proc. of the Int’l Conf. on Parallel Processing. 2016. 329−337.

[12] Li Z, Beaumont M, Jannesari A, *et al*. Fast data-dependence profiling by skipping repeatedly executed memory operation. In: Proc. of the Int’l Conf. on Algorithms and Architectures for Parallel Processing. 2015. 583−596.

[13] Sharir M, Pnueli A. Two Approaches to Interprocedural Data Flow Analysis. New York: Courant Institute of Mathematical Sciences, Computer Science Department, New York University, 1978.

[14] Rountev A, Kagan S, Marlowe T. Interprocedural dataflow analysis in the presence of large libraries. In: Proc. of the Int’l Conf. on Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2006. 2−16.

[15] Tang H, Wang X, Zhang L, *et al*. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proc. of the POPL. 2015. 83−95.

[16] Yang Y, Su PR, Ying LY, Feng DG. Dependency-based malware similarity comparison method. Ruan Jian Xue Bao/Journal of Software, 2011,22(10):2438−2453 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3888.htm [doi: 10.3724/SP.J.1001.2011.03888]

[17] Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. ACM Trans. on Programming Languages and Systems, 1987,9(3):319−349.

[18] Rawat S, Mounier L. Finding buffer overflow inducing loops in binary executables. In: Proc. of the IEEE Conf. on Software Security and Reliability. 2012. 177−186.

[19] Yamaguchi F, Maier A, Gascon H, *et al*. Automatic inference of search patterns for taint-style vulnerabilities. In: Proc. of the IEEE S&P. 2015. 797−812.

[20] Shoshitaishvili Y, Wang R, Hauser C, *et al*. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proc. of the NDSS. 2015.

[21] Cheng S, Yang J, Wang J, *et al*. Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code. In: Proc. of the IEEE Conf. on Trust Security and Privacy in Computing and Communications. 2011. 150−159.

[22] Shoshitaishvili Y, Wang R, Salls C, *et al*. (State of) The art of war: Offensive techniques in binary analysis. In: Proc. of the IEEE S&P. 2016. 138−157.

[23] Angr, a binary analysis framework. 2016. http://angr.io/index.html

[24] OWASP Foundation, Inc. OWASP embedded application security. 2017. https://owasp.org/www-project-embedded-application-security/

[25] 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 IEEE S&P. 2010. 317−331.

[26] Wang TL. Research on binary-executable-oriented software vulnerability detection [Ph.D. Thesis]. Beijing: Peking University, 2011 (in Chinese with English abstract).

[27] Hex-rays. 2016. https://www.hex-rays.com

[28] Bellard F. QEMU, a fast and portable dynamic translator. In: Pai V, ed. Proc. of the USENIX Annual Technical Conf. Berkeley: USENIX Association, 2005. 41−46.

[29] Wu SH, Wang W, Zhao X. Revealing the Art of 0 Day Mining in Home Router. Beijing: Publishing House of Electronics Industry, 2015 (in Chinese).

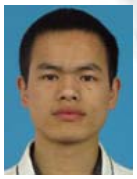
附中文参考文献:

[2] 吴世忠,郭涛,董国伟.软件漏洞分析技术.北京:科学出版社,2014.

[16] 杨轶,苏璞睿,应凌云,冯登国.基于行为依赖特征的恶意代码相似性比较方法.软件学报,2011,22(10):2438−2453. http://www.jos.org.cn/1000-9825/3888.htm [doi: 10.3724/SP.J.1001.2011.03888]

[26] 王铁磊.面向二进制程序的漏洞挖掘关键技术研究[博士学位论文].北京:北京大学,2011.

[29] 吴少华,王炜,赵旭.揭秘家用路由器 0day 漏洞挖掘技术.北京:电子工业出版社,2015.



陈千(1993－),男,硕士,主要研究领域为嵌入式设备安全.

程凯(1991－),男,博士生,主要研究领域为物联网安全,二进制固件的脆弱性分析.

郑尧文(1990－),男,博士,主要研究领域为物联网安全,固件逆向分析与漏洞挖掘.

朱红松(1973－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为物联网安全,漏洞分析,网络测绘,安全大数据分析.

孙利民(1966－),男,博士,研究员,博士生导师,CCF 杰出会员,主要研究领域为物联网及其安全,工业控制系统安全,区块链安全.