软件漏洞静态检测模型及检测框架

王 涛 韩兰胜 付 才 邹德清 刘 铭
(华中科技大学计算机科学与技术学院 武汉 430074)

摘 要 软件漏洞静态分析是信息安全领域的重点研究方向,如何描述漏洞及判别漏洞是漏洞静态分析的核心问题。提出了一种用于描述和判别漏洞的漏洞静态检测模型。首先对软件漏洞的属性特征进行形式化定义,并对多种软件漏洞和其判定规则进行形式化描述;其次,针对传统的路径分析存在的状态空间爆炸问题,提出了一个新的程序中间表示——漏洞可执行路径集,以压缩程序状态空间。在该模型的基础上,设计了一个基于漏洞可执行路径集的软件漏洞静态检测框架,利用定义的漏洞语法规则求解漏洞可执行路径集上的漏洞相关节点集,利用漏洞判定规则对漏洞相关节点集进行判别得出漏洞报告。实验分析验证了该漏洞检测模型的正确性和可行性。

关键词 静态分析,漏洞检测,形式化描述,状态空间爆炸,中间表示
中图法分类号 TP311.53 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.5.015

Static Detection Model and Framework for Software Vulnerability

WANG Tao HAN Lan-sheng FU Cai ZOU De-qing LIU Ming
(School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract Static analysis of source-oriented software vulnerabilities has already been a research focus of information security in recent years. The core problem of vulnerability static detection is how to describe these vulnerabilities and how to detect them. We proposed a static analysis model to describe and detect software vulnerabilities. Firstly, formal definition is used to describe the attributes of several common software vulnerabilities, and these vulnerabilities and its discrimination rules are formulated with formal description. Secondly, a new program intermediate representation called vulnerability executable path set is proposed which is used to take place of traditional path analysis in order to reduce the program state space and avoid state explosion. Based on this model, we designed a static detection framework for software vulnerability based on vulnerability executable path set to solve vulnerability relation nodes with vulnerability syntax rule on vulnerability executable path set and detect vulnerabilities on vulnerability relation nodes by the vulnerability discrimination rules. The results show the correctness and feasibility of the static analysis model.

Keywords Static analysis, Vulnerability detection, Formal description, State explosion, Intermediate representation

1 引言

近年来,随着信息化和互联网的高速发展,软件需求日益旺盛,由软件漏洞引发的计算机安全问题越来越多[1,2]。软件中的任何安全漏洞都可能导致非常严重的后果。绝大多数的黑客攻击都是由软件自身的漏洞引起的[3]。在许多大型软件公司,为了保证软件的安全性而花费的人力、物力和资金已经超过软件研发环节。

从20世纪80年代开始,程序静态分析技术(Program Static Analysis)逐渐发展起来,成为一种重要的软件漏洞分析和检测技术[4]。它在不实际运行代码的方式下,使用编译技术进行代码扫描或编译来查找相关的判断信息,验证代码是否满足规范性、安全性、可靠性、可维护性等指标,具有自动化程度高、检测速度快、效率高等优点。面向源代码的软件漏洞静态分析能尽早地发现和消除软件漏洞,大大降低了软件开发成本和漏洞产生的风险,提高了软件的安全性和可靠性。

传统的静态分析技术存在很多不足。Viega等人[5]设计的漏洞静态扫描工具IST4利用语法分析技术把程序与漏洞数据库进行比较,以确定是否存在漏洞,其检测效率高,但可以检查的漏洞较少,误报率很高。ESC/Java[6]是一个基于定理证明的静态检测工具,具有较低的误报率,但自动化程度不高,需要人工分析来确定漏洞,可扩展性不强。Edmund Clarke等人提出一种软件漏洞测试的形式化验证技术[7],通过遍历状态空间来判定一个给定的程序模型是否满足某些预先定义的特性,但它只适用于小规模的代码检测,当程序规模较大时,会产生状态空间爆炸问题。

到稿日期:2015-05-09 返修日期:2015-09-13 本文受基于任务的木马关联行为识别研究(61272033),移动网络行为的多态聚类及其演化研究(61272405),云计算安全基础理论与方法研究(2014CB340600)资助。

王 涛(1985—),男,硕士,主要研究方向为信息安全,E-mail:leida8813@hust.edu.cn;韩兰胜(1972—),男,博士,副教授,主要研究方向为信息安全、网络安全、恶意代码,E-mail:hanlansheng@hust.edu.cn(通信作者);付 才(1976—),男,博士,副教授,主要研究方向为信息安全、网络安全;邹德清(1975—),男,博士,教授,主要研究方向为信息安全、可信计算;刘 铭(1976—),男,博士,讲师,主要研究方向为信息安全、恶意代码。

近年来，随着研究工作的不断深入，软件漏洞检测的研究重点主要集中在如何描述漏洞特征以及如何设计高效的检测框架进行漏洞定位和判别。Compass[8]是 Lawrence Livermore 实验室 ROSE 编译器工作组开发的一个源代码静态分析工具。Compass 定义和描述了部分代码的安全特性，并提供了一个通用的缺陷规则描述框架，但只对少数代码缺陷进行简单描述，没有对代码缺陷和漏洞的深层次特征进行描述，在检测效率方面存在不足。J. Wilander[9]提出用漏洞依赖图作为一种通用的方式对代码的安全特性进行建模，并对整数溢出和内存多次释放缺陷进行了建模，但该模型可扩展性不强，精度不高。

在国内，梁彬等提出了一个扩展有限状态机模型[10]，对程序中可能执行路径进行静态遍历并识别当前操作，以检测操作数据是否具有期望的安全状态，若出现与期望安全状态不符的情况，则表示发现了一个可能的安全漏洞，一定程度上降低了误报率，但其需要遍历程序的所有可能执行路径，检测效率有待提高。秦晓军等人提出了一种基于一阶逻辑的安全性缺陷静态检测方法[11]，利用命题逻辑和谓词逻辑定义模式路径公式，引入谓词构造逻辑函数表达式，把安全性缺陷检测问题转化为在中间代码对应的有限状态空间中是否存在相应模式路径公式的判定问题。该方法的漏报率和误报率较低，测试时间和代码规模呈渐近线性关系，但该模型对漏洞的形式化描述的通用性仍然不足，漏报率和误报率很大程度上取决于第三方的约束求解器 STP。曾福萍等给出了软件缺陷模式的定义并通过缺陷模式来描述软件漏洞和缺陷[12]，但只从语义上对常见漏洞的缺陷模式进行了文字描述，没有统一的形式化描述。宫云战等人提出了一种基于缺陷模式的软件测试方法[13]，定义和描述了软件缺陷模式的状态机模型，根据状态转换集合中是否可能存在 error 状态来报告潜在漏洞，但检测要遍历整个程序状态空间，因此效率不高，报告的潜在漏洞还需人工来二次确认。

总之，目前针对源代码的静态分析方法和检测工具仍存在下列不足：

1）目前的静态分析方法大都需要遍历状态空间，当程序规模较大时，容易出现状态空间爆炸问题，检测效率不高。

2）目前静态分析技术对漏洞的描述还不够通用和完善，漏洞检测工具如 FindBugs、RATs、klocwork 等在误报率、漏报率等方面仍有很大提高空间。

针对以上问题，本文提出一种判别漏洞的形式化描述模型，利用定义的谓词对软件漏洞的特征属性进行描述，利用各种依赖关系来描述漏洞的判别规则，对软件漏洞及其判定规则进行统一的形式化定义和描述。针对传统静态分析中的状态空间爆炸问题，提出用漏洞可执行路径集 VEPS（Vulnerability Executable Path Set）作为程序的分析路径来代替完整的路径分析，减少路径数量和路径上的节点数，压缩程序状态空间；然后设计了一个基于 VPES 的漏洞静态检测框架。与文献[13]相比，本文通过漏洞判定规则来判别漏洞，不需要进行二次人工判定；与文献[11]相比，本文在 VEPS 上推导求解漏洞节点集并进行漏洞判定，整个检测过程都是路径敏感的，避免了过滤路径不可达情况导致的额外计算开销，把漏洞判定规则分为前置判定条件和后置判定条件，分别从程序语句执行前和执行后的状态来判别漏洞，对漏洞的特征描述更加通用，更能反映出漏洞节点在不同时刻的状态。

2 相关定义

任何一个软件漏洞都是由软件中的一个或一些缺陷导致的[14]。软件漏洞也称为软件安全性缺陷。程序源代码中存在安全缺陷是产生软件漏洞的直接原因。对源代码进行漏洞分析也就是查找源代码中存在的安全缺陷，通过消除源代码缺陷来减少软件中潜在的漏洞[15]。采用 CWE（Common Weakness Enumeration）对漏洞进行分类和描述，用来描述和检测的软件漏洞（安全性缺陷）如表 1 所列。

表 1 本文描述的软件漏洞

名称	英文名称	CWE 编号
空指针引用	NPD(Null Pointer Dereference)	CWE-476;CWE-690
缓冲区溢出	BF(Buffer Overflow)	CWE-119;CWE-120
格式化字符串	UFS(Uncontrolled Format String)	CWE-134
资源操作相关缺陷	RRF(Resource Relation Flaws)	CWE-401;CWE-404;CWE-415;CWE-416
竞争条件	RC(Race Condition)	CWE-362

表 1 中的资源操作相关缺陷包含资源泄露（Memory Leak）、不正确的资源关闭或释放（Improper Resource Shutdown or Release）、内存多次释放（Double Free）和内存释放后重用（Use After Free）。

2.1 控制流图和程序依赖图

控制流图 CFG（Control Flow Graph）[16]和程序依赖图 PDG（Program Dependency Graph）[17]是大多数程序分析和测试必不可少的基础数据结构。CFG 是程序每个函数结构的图形化表示，程序中的每个语句对应图中的一个节点，它既表示了函数的控制结构信息，也表示了程序语句执行的流向。PDG 是程序语句间依赖关系的图形表示，能够清晰地表示出程序中所有操作的控制依赖关系及数据依赖关系，是一种经过优化的程序表示形式。下面给出 CFG 上的一些基本定义和性质。

设 n_i，n_j 为 CFG 中的两个节点。

定义 1（可执行路径） 在 CFG 中，若存在一个序列 $P=\langle n_0,\cdots,n_m\rangle$，满足 $(n_{i-1},n_i)\in E$，其中 $i=1,\cdots,m$，那么称 n_0 到 n_m 之间存在可执行路径，记为 $EP(n_0,n_m)$。

定义 2（前驱节点和后继节点） 若 n_i 到 n_j 存在一条可执行路径 $EP(n_i,n_j)$，则称 n_i 是 n_j 的前驱节点，记为 $Pred(n_i,n_j)$；称 n_j 是 n_i 的后继节点，记为 $Succ(n_i,n_j)$。对于一条语句 n，其所有前驱节点组成的集合称为 n 的前驱节点集合，记为 $Pred(n)$，n 的所有后继节点组成的集合称为 n 的后继节点集合，记为 $Succ(n)$。

定义 3（后必经节点） 从 n_i 到 CFG 出口 *exit* 都要经过 n_j，则称 n_j 是 n_i 的后必经节点，又称作后支配节点，记为 $PD(n_j,n_i)$。需要注意的是，n_i 不能成为自己的后必经节点。语句 n 的所有后必经节点构成后必经节点集，记为 $PD(n)$。

定义 4（数据依赖关系） 若满足：(1)存在 n_i 到 n_j 的可执行路径 $EP(n_i,n_j)$；(2)存在一个变量 v，n_j 的执行中使用了在 n_i 处定义的 v 的值；(3)v 在 $EP(n_i,n_j)$ 上没有被重新定义，则称语句 n_j 数据依赖于语句 n_i，记为 $DD(n_j,n_i,v)$。

定义 5（控制依赖关系） 若满足：(1)存在 n_i 到 n_j 的可执行路径 $EP(n_i,n_j)$，对于 $EP(n_i,n_j)$ 上除 n_i 和 n_j 外的每个

节点 n，n_j 都是其后的必经节点；(2)若 n_j 的执行与否完全取决于 n_i 的执行结果，则称 n_j 控制依赖于 n_i，用 $CD(n_i,n_j)$ 表示。

2.2 漏洞属性的谓词定义

下面给出描述软件漏洞所需的一组谓词定义。

定义 6(定义-使用-核查关系) 对于程序中某一变量 v 的使用方式，可描述为定义-使用-核查关系。用谓词 $DEF(v,n)$表示 v 在语句 n 处的声明、定义或赋值；谓词 $USE(v,n)$表示 v 在语句 n 处被使用或引用；谓词 $CHECK(v,n,Statement)$表示在语句 n 上对 v 进行状态判定。如 $CHECK(v,n,Null)$表示对 v 在 n 上的状态是否为 $Null$ 进行核查，核查结果为真或假。

定义 7(变量类型) 对于一个程序 M 中的参数，用相应的谓词来表示其参数类型。如：指针变量 $v=\{v\mid \exists v\in M, type\ of\ v\ is\ Pointer\}$用 $Pointer(v)$表示；函数 f 用 $Function(f)$表示等。

定义 8(资源分配和回收函数集) 用 $ResourceAllocateFunctionList$ 表示资源分配相关的函数集合。用谓词 $ResourceAllocateFunctionList(n)$表示节点 n 上的资源分配函数集合，简写为 $RAF(n)$。用谓词 $ResourceRelease(n)$表示对节点 n 处资源分配函数 $RAF(n)$进行相应的资源回收操作，简写为 $RR(n)$。

定义 9(格式化系列函数) 格式化系列函数用 $FormatFunctionList$ 表示。用谓词 $FormatFunction(n)$表示在语句 n 处调用的格式化系列函数，简写为 $FF(n)$。

定义 10(缓冲区相关函数) 缓冲区相关函数用 $BufferFunctionList$表示。

定义 11(函数调用) 用谓词 $CallFunction(n)$表示在语句 n 上的函数调用。

定义 12(共享资源) 用谓词 $SharedResource(v,n)$表示在程序语句 n 上的共享资源。

定义 13(集合包含) 用谓词 $IsIn(n_1,n_2)$表示 $n_1\subseteq n_2$；用$\neg IsIn(n_1,n_2)$ 表示 $n_1\not\subset n_2$。

3 漏洞形式化描述模型

软件缺陷是软件的一个固有属性并经常出现[12]，安全性缺陷也不例外。也就是说，软件安全性缺陷有一些特定的模式或规律。程序中经常发生的安全性缺陷所呈现出的语法或语义特征就是软件漏洞的特征。如 C 语言中的缓冲区溢出漏洞，若在执行字符串拷贝时没有对 $strcpy()$这类缓冲区相关函数进行长度检查，那么就可能存在缓冲区溢出漏洞[18]；申请的资源在使用完后必须进行正确释放，否则可能会造成资源泄露或资源释放不正确缺陷等。因此可以通过对程序中漏洞特征属性进行抽象来建立漏洞的检测模型，若违反该特征属性，则产生一个漏洞。

3.1 漏洞静态检测模型

用一个五元组来描述软件漏洞的检测模型，即 $Vulnerability=\{n_0,F,S,P,Q\}$。其中：

1)漏洞初始节点集 n_0，指漏洞的初始特征节点集是漏洞检测的入口节点。对于一个程序来说，若存在漏洞初始节点，那么在包含该漏洞初始节点的程序路径上，可能存在漏洞。比如，若存在一个缓冲区系列函数 $memcpy()$，那么在 $memcpy()$的执行路径上，就可能会存在缓冲区溢出漏洞。漏洞初始节点集 n_0 是从目标程序的源代码中静态提取的节点语句。对于程序 M 来说，其程序语句集合是有限的[11]，因此漏洞初始节点 n_0 也是有限的和确定的。

2)程序状态空间 F，是由程序中的可执行路径集 EP 以及 EP 上各节点间的控制依赖关系、数据依赖关系、前驱节点、后继节点、后必经关系以及变量的定义-使用-核查等信息构成的程序状态空间。F 可看做是目标程序的中间表示(Intermediate Representation)，其包含的信息都是漏洞检测的必要信息。

3)漏洞语法规则集 S，是漏洞在程序可执行路径 EP 上的行为特征，是通过漏洞初始节点 n_0 推导漏洞节点集 N 的状态转移规则。即在程序状态空间 F 中 EP 的任一路径 EP_i 上，$\forall n_1$ 满足 $n_0\xrightarrow{S}n_1$，那么 $n_1\in N$。漏洞节点集 N 中的节点是可能存在漏洞的程序语句节点。

4)前置判定条件 P，定义了漏洞节点集 N 中任一节点 n 执行前要满足的状态条件，不满足 P 则产生漏洞。

5)后置判定条件 Q，定义了漏洞节点集 N 中任一节点 n 执行后要满足的状态条件，不满足 Q 即产生漏洞。

从一阶逻辑的角度来看，任何程序漏洞都可看作判定性问题(Decision Problem)或功能性问题(Function Problem)。借助 Hoare 逻辑验证程序正确性的思想，漏洞判别的表示形式为 $F:\{P\}N\{Q\}$。这样就把漏洞检测变为判定 N 在 EP 上是否满足 P 和 Q 的判定性问题。即当 N 中的任一节点 n 在执行前必须满足 P，在执行后必须满足 Q 时，才能判定为不存在漏洞，否则存在漏洞。前置判定条件 P 和后置判定条件 Q 不能同时为空。

根据上述的漏洞检测模型，漏洞检测过程可表示为 $F:\{P\}n_0\xrightarrow{S}N\{Q\}$。该过程包含漏洞的粗定位和漏洞的精准定位两部分。

(1)漏洞的粗定位，即 $F:n_0\xrightarrow{S}N$。在 F 中的可执行路径集 EP 上，从 n_0 开始，查找各路径上满足漏洞语法规则集 S 的节点。若在 F 中的任一条可执行路径 EP_i 上，$\forall n_1$ 满足 $n_0\xrightarrow{S}n_1$，那么 n_1 即为漏洞点，$n_1\in N$。漏洞点 N 就是可能产生漏洞的程序语句节点。

(2)漏洞的精准定位，即 $F:\{P\}N\{Q\}$。对 F 中的任一可执行路径，用 P 和 Q 分别对漏洞节点集 N 进行判别，从而得出漏洞结果。若在 F 中的任一条路径 EP_i 上，在 n_1 执行前，$\forall n_2$ 满足 $n_1\xrightarrow{P}n_2$，且在 n_1 执行后，$\forall n_3$ 满足：$n_1\xrightarrow{Q}n_3$，判定结果为 TRUE，那么节点 n_1 不存在漏洞；在 n_1 执行前，$\neg\forall n_2$ 满足：$n_1\xrightarrow{P}n_2$，或在 n_1 执行后，$\neg\forall n_3$ 满足 $n_1\xrightarrow{Q}n_3$，判定结果为 FALSE，那么节点 n_1 存在漏洞。

3.2 软件漏洞的形式化描述

在对 CWE 公布的多种常见漏洞实例及文献[19]中各种漏洞的成因、导致后果、特征、预防措施进行研究分析的基础上，实现了对表 1 所列的软件漏洞的形式化描述。

1)空指针引用。对于目标程序 M，M 中定义或声明的指针 $Pointer(v)$的语句 n 构成的集合即为漏洞初始节点集。即 $n_0=\{n\mid \exists n\in M\wedge DEF(Pointer(v),n)\}$。

在任一条程序可执行路径 EP_i 上，若存在 n_0 的后继节点

n_1,且在节点 n_1 上调用了 $Pointer(v)$且关于 $Pointer(v)$数据依赖于 n_0,那么 n_1 就是漏洞点,即 $S=Succ(n_1,n_0)\wedge DD(n_1,n_0,Pointer(v))\wedge USE(Pointer(v),n_1)$。

在该 EP_i 上,若存在节点 n_2 关于 $Pointer(v)$数据依赖于 n_0 且漏洞点 n_1 控制依赖于 n_2,在 n_2 上对 $Pointer(v)$是否为空进行核查,即 $P=DD(n_2,n_0,Pointer(v))\wedge CD(n_2,n_1)\wedge CHECK(Pointer(v),n_2,NotNull)$。空指针引用的后置判定条件 Q 为空。

2)缓冲区溢出。目标程序 M 中调用缓冲区相关函数 $BufferFunctionList$ 的语句 n 构成的集合即为漏洞初始节点集。即 $n_0=\{n\mid \exists n\in M\wedge CallFunction(n)\subseteq BufferFunctionList\}$。

在任一条程序可执行路径 EP_i 上,n_0 数据依赖于其前驱节点 n_1 上定义的缓冲区变量 $Buffer(v_1)$,则 n_0 为漏洞点。故 $S=DD(n_0,n_1,Buffer(v_1,n_0))\wedge DEF(Buffer(v_1),n_1)\wedge Pred(n_1,n_0)$。

在该 EP_i 上,若存在节点 n_2,n_0 控制依赖于 n_2,且 n_1 是 n_2 的后必经节点,则在 n_0 或 n_1 上对缓冲区大小 $Size$ 和输入数据长度 $input(v_2)$进行核查,即漏洞前置判定条件 $P=CD(n_2,n_0)\wedge PD(n_2,n_1)\wedge CHECK(buffer(v_1),input(v_2),n_1,Size)\cup CHECK(buffer(v_1),input(v_2),n_0,Size)$,后置判定条件 Q 为空。

3)格式化字符串。目标程序 M 中调用格式化系列函数 $FormatFunctionList$ 的语句 n 构成的集合即为漏洞初始节点集。即 $n_0=\{n\mid \exists n\in M\wedge CallFunction(n)\subseteq FormatFunctionList\}$。

在任一条程序可执行路径 EP_i 上,n_0 数据依赖于其前驱节点 n_1 上定义的变量 v,则 n_0 为漏洞点。故 $S=Succ(n_1,n_0)\wedge DEF(v,n_1)\wedge DD(n_0,n_1,v)$。

在该 EP_i 上,对 n_0 处调用的格式化字符串函数的参数个数和类型进行核查,若匹配为 TURE,则不存在漏洞;反之,存在漏洞。$P=CHECK(FF(n_0),n_0,Parameter)$,后置判定条件 Q 为空。

4)资源相关缺陷。目标程序 M 中定义或声明的属于资源分配函数集的变量 v 的语句 n 构成的集合为漏洞初始节点集。$n_0=\{n\mid \exists n\in M\wedge DEF(v,n)\subseteq ResourceAllocateFunctionList\}$。

在任一条程序可执行路径 EP_i 上,n_0 的后继节点 n_1 上使用了 n_0 上的资源分配函数 $RAF(n_0)$且 n_1 关于 $RAF(n_0)$数据依赖于其前驱节点 n_0,则 n_1 为漏洞点。故 $S=USE(RAF(n_0),n_1)\cup Succ(n_1,n_0)\wedge DD(n_1,n_0,RAF(n_0))$。

在该 EP_i 上,若存在 n_1 的前驱节点 n_2,n_2 中不存在与资源分配函数 $RAF(n_0)$所对应的资源回收操作 $RR(n_2)$,则 $P=Pred(n_2,n_1)\wedge\neg IsIn(RR(n_2),RAF(n_0))$。

在该 EP_i 上,若存在 n_1 的后必经节点 n_3 关于 $RAF(n_0)$数据依赖于其前驱节点 n_0,且在 n_1 和 n_3 中存在资源分配函数 $RAF(n_0)$所对应的所有资源回收操作 $RR(n_1)$或 $RR(n_3)$,那么 $Q=PD(n_3,n_1)\wedge DD(n_3,n_1,RAF(n_0))\wedge IsIn(RAF(n_0),RR(n_3))\cup IsIn(RAF(n_0),RR(n_1))$。

5)竞争条件。目标程序 M 中共享资源 v 所在的语句 n 构成的集合为竞争条件的漏洞初始节点集。即 $n_0=\{n\mid \exists n\in M\wedge SharedResource(v,n)\}$。

在任一条程序可执行路径 EP_i 上,节点 n_0 的后继节点 n_1 使用了共享变量 v,且 n_1 关于 v 数据依赖于 n_0,则 n_1 为漏洞点,即 $S=Succ(n_1,n_0)\wedge DD(n_1,n_0,v)\wedge USE(v,n_1)$。

在该 EP_i 上,在以 n_1 为后必经节点的节点 n_2 上对共享变量 v 是否加锁进行核查,即 $P=PD(n_1,n_2)\wedge CHECK(v,n_2,Lock)$。

在该 EP_i 上,对 n_1 的后必经节点 n_3 的共享变量 v 是否解锁进行核查,即 $Q=PD(n_3,n_1)\wedge CHECK(v,n_3,UnLock)$。

4 漏洞可执行路径集 VEPS

误报率和漏报率是衡量静态分析方法的两大关键指标。路径敏感(path-sensitive)的静态分析方法能避免路径不可达的情况,从而降低了漏报率和误报率。在传统的路径敏感分析方法中,完整路径分析是最准确的方法。完整路径分析能准确记录和描述每条不同路径上的"程序执行状态"。但当程序规模较大时,尤其是当程序功能和逻辑关系极其复杂时,其需要分析的路径规模会十分巨大,会出现与模型检测等静态分析技术类似的状态空间爆炸问题。针对这个问题,本文提出一个新的程序中间表示 IR(Intermediate Representation)——漏洞可执行路径集 $VEPS$,用 $VEPS$ 来代替程序完整路径集 EP 进行漏洞分析。

$VEPS$ 是由 CFG 中的漏洞初始节点 n_0 到 $exit(entry)$节点的可执行路径组成的集合,是描述程序中漏洞的可执行路径的一个中间表示。对于程序 M,$VEPS$ 是 M 上的所有可执行路径 EP_M 的子集,$VEPS\subseteq EP_M$。也就是说,$VEPS$ 上的节点是可执行路径 EP_M 包含漏洞初始节点 n_0 的节点子集。$VEPS$ 中包含漏洞在可执行路径上的所有节点。各种漏洞的 $VEPS$ 和程序可执行路径集 EP 的关系如表 2 所列。

表 2 各种漏洞的 $VEPS$ 与可执行路径集 EP 的关系

漏洞名称	可执行路径集 EP
NPD	EP(n_0,exit)
BF	EP(entry,n_0)
UFS	EP(entry,n_0)
RRF	EP(n_0,exit)
RC	EP(n_0,exit)

从表 2 可看出,$VEPS$ 中只包含程序可执行路径集 EP 中从 $entry$ 到 n_0,或 n_0 到 $exit$ 之间的节点,从而大大减少了漏洞分析的路径数量和路径上的节点数。与传统的路径分析相比,$VEPS$ 和软件代码的规模不成正比,其规模首先取决于代码中的漏洞初始节点 n_0 的个数,因此用 $VEPS$ 能压缩程序的状态空间。

求解 $VEPS$ 其实就是对 CFG 的简化。下面以 $EP(n_0,exit)$为例在 CFG 上采用深度搜索优先的思想来求解 $VEPS$,如算法 1 所示。

算法 1 $VEPS$ 求解算法

输入:控制流图 G=〈V,E,entry,exit〉

输出:漏洞可执行路径集 EP(n_0,exit)

```
1. Initialization: EP(n0,exit)=〈n0,exit〉;
2. DirSucc(n0); //查找 n0 的直接后继节点;
3.   IF(DirSucc(n0) != NULL) THEN
4.     FOR each n∈DirSucc(n0)
5.       DO WHILE
6.         IF n is not in EP(n0,exit) THEN
```

```
7.            InsertSort(EP(n0, exit),n0,vi);
8.            n0=n;
9.          END IF
10.       IF n in EP(n0, exit) THEN //vi 在 EP(n0, exit)中;
11.         n0= DirSucc(n);
12.       END IF
13.       DirSucc(n0);
14.     END WHILE(DirSucc(n)==exit)
15.   END FOR
16. END IF
17. RETURN EP(n0, exit)
```

由于 CFG 采用了邻接表的存储方法，每个节点和边只遍历一次，因此该算法的时间复杂度为 $O(m+e)$，其中 m 为 CFG 中从 n_0 到 *exit* 的节点数，e 代表这些节点的边数。由于 m 小于 CFG 的节点数，因此 e 必然小于 CFG 中的边数。求解 $EP(entry,n_0)$时，需将算法 1 中的 $DirSucc(v_i)$换成 $DirPred(v_i)$，*exit* 换成 *entry* 即可。

图 1 是一个控制流图 CFG，该 CFG 的程序可执行路径集共有 6 条路径，即：

路径 1：⟨1,2,3,4,*exit*⟩；

路径 2：⟨1,2,3,4,5,*exit*⟩；

路径 3：⟨1,2,5,*exit*⟩；

路径 4：⟨1,6,7,*exit*⟩；

路径 5：⟨1,6,7,8,9,*exit*⟩；

路径 6：⟨1,6,8,9,*exit*⟩。

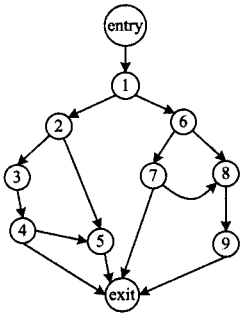

图 1　一个程序流程图

假设漏洞初始节点为节点 2，求解 *VEPS* 的详细节点信息如表 3 所列。

表 3　CFG 中的直接后继节点集

节点	直接后继节点集	VEPS 上的节点
2	3,5	⟨2,3⟩ ⟨2,5⟩
3	4	⟨2,3,4⟩
4	5,exit	⟨2,3,4,5⟩ ⟨2,3,4,exit⟩
5	exit	⟨2,3,4,5,exit⟩ ⟨2,5,exit⟩

首先查找 CFG 上节点 2 的直接后继节点，得到节点 3 和节点 5，此时 *VEPS* 分为两条路径，节点分别是⟨2,3⟩和⟨2,5⟩；按照深度优先搜索，先遍历节点 3 的直接后继节点，得到节点 4，此时该 *VEPS* 为节点⟨2,3,4⟩；以此类推，得到两条漏洞可执行路径⟨2,3,4,*exit*⟩和⟨2,3,4,5,*exit*⟩；再查找节点 5 的直接后继节点，得到一条漏洞可执行路径⟨2,5,*exit*⟩。最后得出的 *VEPS* 共 3 条可执行路径，即：

*VEPS*1：⟨2,3,4,5,6,*exit*⟩

*VEPS*2：⟨2,3,4,5,6,7,*exit*⟩

*VEPS*3：⟨2,3,7,*exit*⟩

对比 *EP* 和 *VEPS* 可以发现，无论是路径数还是各条路径上的节点数，*VEPS* 都少于 *EP*。节点 1 的右侧所有的子节点都不用考虑，这就大大缩减了路径分析的数量。即使考虑最坏的情况，即节点 1 为漏洞初始节点，那么此时 *VEPS* 也才完全等于 *EP*。因此，相对于程序可执行路径集 *EP*，程序的漏洞可执行路径集 *VEPS* 能减少路径数和路径上的节点数，压缩程序状态空间。

5　漏洞静态检测框架设计

以 *VEPS* 作为程序中间表示，设计了一个基于 *VEPS* 的静态分析框架，该框架如图 2 所示。

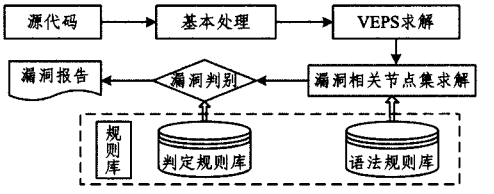

图 2　漏洞检测框架

该框架包括 5 个部分：基本处理模块、*VEPS* 求解模块、漏洞相关节点集求解模块、漏洞判别模块和漏洞规则库。

5.1　基本信息处理

基本信息处理模块用来生成和提取目标程序中的一些基本静态信息，如图 3 所示。

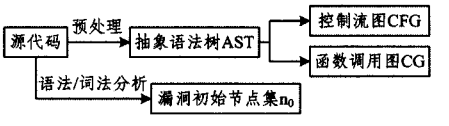

图 3　基本处理模块流程

首先，对目标程序源码进行词法/语法分析以提取缺陷初始节点集 n_0；然后针对不同的程序语言，通过编译器的前端（如 GCC、Java 编译器等）来生成抽象语法树 AST 并构造程序的 CFG 和函数调用图 CG(Call Graph)。

5.2　VEPS 求解

VEPS 的求解是漏洞静态分析框架的基础。该模块包括：*VEPS* 的求解、PDG/SDG 的生成、过程间分析和 *VEPS* 上的节点信息提取，如图 4 所示。

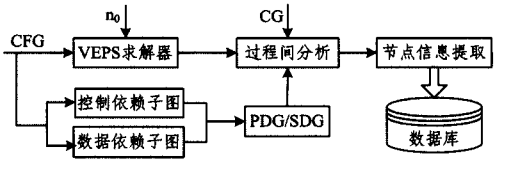

图 4　*VEPS* 求解模块流程

5.2.1　VEPS 求解

在 CFG 上，按照算法 1 从 n_0 节点开始进行深度优先遍历算法求解出 *VEPS*。此时 *VEPS* 是过程内的漏洞可执行路径集。

5.2.2　PDG 和 SDG 的生成

在 CFG 上根据定义 4 和定义 5 得到各节点的控制依赖关系和数据依赖关系，然后再提取其控制依赖子图 CDS (Control Dependence Subgraph)[17,20] 和数据依赖子图 DDS (Data Dependence Subgraph)[17]，构造其程序依赖图

PDG[17,21]。由多个 PDG 和程序依赖子图 PrDS(Procedure Dependence Subgraph)构造对应的系统依赖图 SDG(System Dependence Graph)[22]。

5.2.3 过程间分析

对于静态分析,过程间分析能大大降低误报率。对于过程内 *VEPS*,通过过程间分析来构建完整的过程间 *VEPS*。过程间分析就是分析过程内 *VEPS* 上节点是否与程序其他路径存在函数调用关系和节点依赖关系。

对于 *VEPS* 中的任一条漏洞可执行路径 $VEPS_i$:

1)分析函数调用图 CG,若其上任一节点 n 包含其他路径的节点函数调用;

2)若漏洞初始节点 n_0 与被调用函数 H 中的任一节点 h 有控制依赖或数据依赖关系。

若不能同时满足上述两个条件,则不用构建过程间可执行路径。此时过程内 *VEPS* 就是完整的漏洞可执行路径集。条件 2)规定漏洞初始节点 n_0 与被调用函数 H 中的任一节点 h 有控制依赖或数据依赖关系,其好处是:保证被调用函数 H 中的节点仍然和漏洞初始节点 n_0 有关,避免增加与漏洞无关的路径。

若同时满足上述两个条件,那么需要重新构建这条 $VEPS_i$ 的过程间可执行路径,步骤如下:

1)采用深度优先搜索算法遍历 H 的 CFG,得到 H 的完整可执行路径集 $EP_H(entry,exit)$;

2)得到 $EP_H(entry,exit)$ 中包含 h 节点的路径集 $EP_h(entry,exit)$;

3)把 $VEPS_i$ 看成一个单链表,把 $EP_h(entry,exit)$ 插入到 $VEPS_i$ 中。其中,*entry* 节点作为节点 n 前驱节点的后继节点,*exit* 作为节点 n 后继节点的前驱节点,然后删除节点 n。

5.2.4 节点信息提取

经过过程间分析后,静态提取 *VEPS* 上各节点的节点信息,把 *VEPS* 和其上的节点信息加入到存储程序状态空间 F 的数据库中。节点信息包括:控制依赖关系、数据依赖关系、前驱节点、后继节点和定义-使用链等。在提取节点信息时,根据漏洞语法规则集、前置判定条件和后置判定条件中描述的信息来提取对应的节点信息,这样可以避免额外提取多余的信息。如资源操作相关缺陷,其漏洞语法规则集、前置判定条件和后置判定条件中没有控制依赖关系,那么在提取节点信息时,就不需要提取节点间的控制依赖关系。

5.3 漏洞相关节点集 N 的求解

在 *VEPS* 上,通过漏洞初始节点 n_0 查找程序状态空间 F 中满足漏洞语法规则集 S 的节点,将其加入到漏洞相关节点集 N。

对于空指针引用、资源操作相关漏洞和竞争条件,在 $VEPS_i$ 上:$\forall n_1$ 满足 $n_0 \xrightarrow{S} n_1$,那么 $n_1 \in N_{VEPS_i}$。

对于缓冲区溢出和格式化字符串,在 $VEPS_i$ 上:$\forall n_1$ 满足 $n_0 \xrightarrow{S} n_1$,那么 $n_0, n_1 \in N_{VEPS_i}$。

漏洞相关节点集 N 的求解算法如算法 2 所示。

算法 2 漏洞相关节点集 N 的求解算法

输入:$VEPS_i$,S_i,程序状态空间 F

输出:漏洞相关节点集 N_{VEPS_i};

1. Initialization: $N_{VEPS_i}=\{\emptyset\}$
2. FOR each $n \in VEPS_i$, except n_0
3. IF(Relation(n_0, n)$\subseteq$F && Relation(n_0, n)==S_i) THEN
4. $N_{VEPS_i}=N_{VEPS_i} \cup n$
5. END IF
6. END FOR
7. RETURN N_{VEPS_i}

该算法的时间复杂度为 $O(m)$,m 为漏洞可执行路径上的节点数。对于缓冲区溢出和格式化字符串,N_{VEPS_i} 中还包含漏洞初始节点 n_0。

5.4 漏洞的判别求解

漏洞判别模块是漏洞分析的精准定位过程。对于 $VEPS_i$,$\forall n_2, n_2 \in N_{VEPS_i}$ 满足 $n_1 \xrightarrow{P} n_2$,且 $\forall n_3, n_3 \in N_{VEPS_i}$ 满足 $n_1 \xrightarrow{Q} n_3$,则判定结果为 TRUE,n_1 节点不存在漏洞;$\neg \forall n_2$,$n_2 \in N_{VEPS_i}$ 满足 $n_1 \xrightarrow{P} n_2$,或 $\neg \forall n_3, n_3 \in N_{VEPS_i}$ 满足 $n_1 \xrightarrow{Q} n_3$,则判定结果为 FALSE,n_1 节点存在漏洞。判别算法如算法 3 所示。

算法 3 漏洞判别算法

输入:$VEPS_i$、程序状态空间 F、N_{VEPS_i}、前置和后置判定条件 P、Q

输出:漏洞节点 Vulnerability Nodes

1. Initialization: Vulnerability Nodes=$\{\emptyset\}$
2. FOR each n$\in N_{VEPS_i}$
3. FOR each m$\in VEPS_i$, m$\neq$n
4. IF(Relation(n, m)$\subsetneq$F) THEN
5. Vulnerability Nodes = Vulnerability Nodes$\cup$n
6. ELSE IF(Relation(n, m)$\in$F &&(Relation(n, m) !=P
|| (Relation(n, m)!=Q)) THEN
7. Vulnerability Nodes = Vulnerability Node$\cup$n
8. END IF
9. END FOR
10. RETURN Vulnerability Nodes

该算法的时间复杂度为 $O(n^2)$,n 为 $VEPS_i$ 上的节点数。

5.5 漏洞规则库

漏洞规则库是一个数据库,其中包含漏洞语法规则 S 和漏洞前置判定条件 P、后置判定条件 Q,还包含与漏洞相关的 API 函数,如缓冲区系列函数、资源分配系列函数、格式化系列函数。常见的格式化系列函数包括 *printf*( )、*strncpy*( )、*fwprintf*( )、*fprint*( )、*snwscanf*( )、*printf*( )等。常见缓冲区的相关 API 函数有:*memcpy*( )、*strcpy*( )、*sprintf*( )、*vsprintf*( )、*gets*( )、*scanf*( )、*strcat*( )等。

6 实验与结果分析

实验共 3 部分:对开源项目 Wireshark 1.2.0 进行漏洞静态分析,并与美国国家标准与技术研究院 NIST[23] 公布的漏洞进行分析比较;对 4 个开源项目中已公开的漏洞进行正确性验证;最后对开源项目 Tomcat 4.1 中的空指针引用进行静态分析,并与 FindBugs 进行结果比较。

6.1 Wireshark 的检测结果

Wireshark 是一个开源的网络封包分析软件。Wires 1.2.0 源码共 1840685 行,总文件数为 1518。采用本文方法对开源项目 Wireshark 1.2.0 进行了漏洞静态分析。Wireshark 1.2.0 中的 CFG 个数和 *VEPS* 数如表 4 所列。

表 4　Wireshark 的 CFG 和 *VEPS* 路径数

漏洞名称	漏洞初始节点个数	CFG 的路径数	VEPS 路径数
NPD	6910	10897	8822
BF	1211	4699	2609
UFS	822	691	854
RRF	436	2794	2073
RC	109	149	122
总计	9488	19230	14480

表 4 中统计的 CFG 路径数只是包含漏洞初始节点集的路径数量，不是整个程序源代码的 CFG 路径数。从表 4 能够看出，*VEPS* 的路径数为 14480，小于包含漏洞初始节点集的 CFG 的路径数 19230。另外，表 4 中格式化字符串漏洞的 CFG 路径数小于 *VEPS* 的路径数，这是因为在统计 CFG 路径数时，只统计了包含格式化函数的 CFG，而在实际的路径分析中，进行完整 CFG 遍历的路径数要远远大于 *VEPS* 的路径数。

将漏洞分析结果与 NIST 公布的公开漏洞进行比较，结果如表 5 所列。

表 5　Wireshark 实验结果

漏洞名称	NIST 公布	本文方法	相同漏洞	误报/漏报	误报率/漏报率
NPD	26	43	20	17/6	39.5%/23%
BF	106	121	99	15/7	12.4%/6.6%
UFS	0	0	0	0/0	0%/0%
RRF	6	8	6	2/0	25%/0%
RC	0	0	0	0/0	0%/0%
总计	138	172	125	34/13	19.7%/9.4%

从表 5 可以看出，以 NIST 的数据为标准，本文空指针引用的检测结果误报率较高。究其原因，是由于本文制定的漏洞判别规则中要求在调用每个指针之前都要对其进行显式的非空判定，而在实际程序中，部分函数在返回指针时已经进行了非空判定，并进行了相应处理，而不在调用该指针前进行判定。但我们认为，在指针返回时判定是否为空并不能避免空指针引用的出现，因为指针在下次调用前，不能保证指针的值没有改变；缓冲区溢出的检测结果有较低的误报率和漏报率；资源操作相关漏洞经过二次验证，发现了 2 个误报的漏洞，但没有漏报的漏洞。总体来看，本文方法的漏报率和误报率都较低。

6.2　公开漏洞的验证

采用本文方法对 4 个开源项目在 NIST 中已公开的漏洞进行了验证和确认，结果如表 6 所列。

表 6　已公开漏洞的验证结果

漏洞	Chrome 5.0		Wireshark1.8		ABM1.0		Asterisk10.2		确认率
	NIST	确认	NIST	确认	NIST	确认	NIST	确认	(%)
NPD	0	0	1	1	0	0	5	5	100
BF	1	1	43	40	58	49	14	14	88.9
UFS	0	0	1	1	8	8	0	0	100
RRF	5	5	0	0	9	8	0	0	92.8
RC	0	0	0	0	3	1	0	0	33.3

从表 6 可看出，选取的开源项目涵盖了本文所描述的各种软件漏洞。结果表明，本文方法适用于大多数漏洞。从结果来看，其对资源操作相关漏洞、空指针引用和格式化字符串有较高的检测准确率，缓冲区溢出有较低的误报率和漏报率，竞争条件的检测准确率较低。静态分析由于不实际运行程序，因此对竞争条件的检测存在很大瓶颈，这也是今后需要改进和优化的地方。

6.3　FindBugs 结果比较

针对 6.1 节中空指针引用误报率较高的情况，对 Tomcat4.1 的空指针引用进行检测，并与 FindBugs 进行比较，结果如表 7 所列。

表 7　Tomcat4.1 实验结果

检测工具	检测结果	确认	不能确认	误报	误报率	检测率
FindBugs	36	2	12	22	61.1%	5.6%
本文方法	42	13	8	21	50%	30.9%

从检测结果发现，本文方法对于空指针引用的误报率为 50%，检测率为 30%，但与 FindBugs 相比，本文方法的检测率和误报率都较低。

结束语　针对当前漏洞静态分析方法中存在的不足，本文提出了一个形式化描述的漏洞检测模型，对多种常见漏洞特征和判定规则进行了统一的形式化定义和描述；然后提出用漏洞可执行路径集 *VEPS* 代替传统的完整路径分析，以压缩程序状态空间，并给出 *VEPS* 的求解算法；最后设计了一个基于 *VEPS* 的漏洞静态检测框架，并对多个开源项目进行了验证分析。该漏洞检测模型可应用于软件测试和漏洞分析等领域，*VEPS* 可作为一种新的程序中间表示，用于其他静态分析技术。在 *VEPS* 上求解和判定漏洞，消除了路径不可达的情况，降低了误报率；把整个代码空间转化为 *VEPS*，对 *VEPS* 上有限的漏洞相关节点集进行判定，压缩了检测代码空间，提高了检测效率。

下一步工作：一方面，继续研究更多漏洞的语法和语义特征，并在漏洞检测模型上对其进行形式化描述；另一方面，应用该模型，开发自动化的静态检测工具，继续对各大型开源项目进行检测研究，在检测效率上进行实验分析。

参 考 文 献

[1] Sandu R S, Samaratiy P. Access Control Principles and Practice [J]. IEEE Communications Magazine, 1994, 32(9): 40-48

[2] Krsul I V. Software Vulnerability Analysis[D]. West Lafayette: Purdue University, 1998

[3] Li Peng, Cui Bao-jiang. A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools [C] // 2010 IEEE International Conference on Information Theory and Information Security. Beijing, China: IEEE Press, 2010: 521-524

[4] Chess B, McGraw G. Static Analysis for Security[J]. IEEE Security & Privacy, 2004, 10(3): 53-56

[5] Viega J, Bloch J T, Kohno Y, et al. ITS4: A Static Vulnerability Scanner for C and C++ code[C]// 16th Annual Conference on Computer Security Applications. Piscataway, USA: IEEE, 2000: 257-267

[6] Flanagan C, Leino K R M, Lillibridge M, et al. Extended Static Checking for Java[C] // 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation. Berlin, Germany: ACM Press, 2002: 234-245

[7] Clarke E, Grumberg O, Peled D. Model Checking [M]. Cambridge: MIT Press, 1999

[8] Quinlan D, Panas T. Source Code and Binary Analysis of Softw are defects[C] // 5th Annual Workshop on Cyber Security and Information Intelligence Challenges and Strategies. New York, USA: AMC Press, 2009: 1-4

[9] Wilander J. Modeling and Visualizing Security Properties of Code Using Dependence Graphs[C] // 5th Conference on Software Engineering Research and Practice in Sweden (SERPS' 05). Vasteras, Sweden: ACM Press, 2005: 65-74

(下转第 116 页)

Service[C]// 2014 International Conference on Parallel, Distributed and Grid Computing. 2014:99-104

[2] Wu Ji-yi, Ping Ling-di, Pan Xue-zeng, et al. Cloud Computing: Concept and Platform [J]. Telecommunications Science, 2009, 25(12):23-30(in Chinese)
吴吉义,平玲娣,潘雪增,等. 云计算:从概念到平台[J]. 电信科学,2009,25(12):23-30

[3] Lin Wei-wei, Qi De-yu. Survey of Resource Schedulingin Cloud Computing [J]. Computer Science, 2012, 39(10): 1-6(in Chinese)
林伟伟,齐德昱. 云计算资源调度研究综述[J]. 计算机科学,2012,39(10):1-6

[4] Hou Hong-feng, Wang Can, Wang Li-juan. Research on Information Security Issues of Cloud Computing [J]. Electronic Design Engineering, 2012, 20(22):60-62(in Chinese)
侯洪凤,王璨,王立娟. 云计算信息安全问题探讨[J]. 电子设计与工程,2012,20(22):60-62

[5] Marozzo F, Talia D, Trunfio P. A Cloud Framework for Parameter Sweeping Data Mining Applications[C]// 2011 Third IEEE International Conference on Cloud Computing Technology and Science. 2011:367-374

[6] Garcia T, Wang T. Analysis of Big Data Technologies and Methods-Query Large Web Public RDF Datasets on Amazon Cloud Using Hadoop and Open Source Parsers[C]// 2013 IEEE Seventh International Conference on Semantic Computing. 2013:244-251

[7] Yang Hang, Fong S. Countering the Concept-drift Problem in Big Data Using iOVFDT[C]// 2013 IEEE International Congress on Big Data. 2013:126-132

[8] Zhang Wei-dong, Zhang Kai, Dong Qing, et al. Calculation of Information Entropy in Data Mining with Decision Tree [J]. Computer Engineering, 2001, 27(3):71-73(in Chinese)
张维东,张凯,董青,等. 利用决策树进行数据挖掘中的信息熵计算[J]. 计算机工程,2001,27(3):71-73

[9] Zheng Xian. Status and Development of Web Mining[J]. Technology and Market, 2013, 20(3):64-65(in Chinese)
郑弦. Web 挖掘的现状和展望[J]. 技术与市场,2013,20(3):64-65

[10] Wang Zhen-qi, Li Hai-long. Research of massive Web log data mining based on cloud computing[C]// 2013 International Conference on Computational and Information Sciences. 2013:591-594

[11] Dev H, Sen T, Basak M, et al. An Approach to Protect the Privacy of Cloud Data from Data Mining Based Attacks[C]// 2012 SC Companion: High Performance Computing, Networking Storage and Analysis. 2012:1106-1115

[12] Chen Ke-you. The Research on Data Security and Privacy Issues in Hybrid Cloud Computing [D]. Nanchang: Jiangxi Normal University, 2013(in Chinese)
陈科有. 混合云计算数据安全与隐私保护问题研究[D]. 南昌:江西师范大学,2013

[13] LiLing-juan, Zheng Shao-fei. Analysis of Data MiningPrivacy Preserving Technology Based on Data Processing [J]. Computer Technology and Development, 2011, 21(3):94-97(in Chinese)
李玲娟,郑少飞. 基于数据处理的数据挖掘隐私保护技术分析[J]. 计算机技术与发展,2011,21(3):94-97

(上接第 86 页)

[10] Liang Bin, Hou Kan-kan, Shi Wen-chang, et al. A Static Vulnerabilities Detection Method Based on Security State Tracing and Checking[J]. Chinese Journal of Computers, 2009, 32(5):899-909(in Chinese)
梁彬,侯看看,石文昌,等. 一种基于安全状态跟踪检查的漏洞静态检测方法[J]. 计算机学报,2009,32(5):899-909

[11] Qin Xia-jun, Gan Shui-tao, Chen Zuo-ning. A Static Detection Technoogy of Software Code Secure Vulnerabiity Based on First-order Logic[J]. Scientia Sinica Informationis, 2014, 44:108-219(in Chinese)
秦晓军,甘水滔,陈左宁. 一种基于一阶逻辑的软件代码安全性缺陷静态检测技术[J]. 中国科学:信息科学,2014,44:108-219

[12] Zeng Fu-ping, Jin Hui-liang, LU Min-yan. Study on Software Defect Patterns[J]. Computer Science, 2011, 38(2):127-130(in Chinese)
曾福萍,靳慧亮,陆民燕. 软件缺陷模式的研究[J]. 计算机科学,2011,38(2):127-130

[13] Gong Yun-zhan, Yang Chao-hong, Jin Da-hai, et al. Software Defect Patterns and Testing[M]. Beijing: Science Press, 2011:21-22(in Chinese)
宫云战,杨朝红,金大海,等. 软件缺陷模式与测试[M]. 北京:科学出版社,2011:21-22

[14] Chen Z Q, Zhang Y, Chen Z R. A Categorization Framework for Common Vulnerabilities and Exposures[J]. Computer Journal Archive, 2010, 53(5):551-580

[15] Wu Shi-zhong, Guo Tao, Dong Guo-wei, et al. Software Vulnerability Analysis Technology[M]. Beijing: Science Press, 2014:3-6(in Chinese)
吴世忠,郭涛,董国伟,等. 软件漏洞分析技术[M]. 北京:科学出版社,2014:3-6

[16] Allen F E. Control Flow Analysis[J]. ACM SIGPLAN Notices, 1970, 5(7):1-19

[17] Ferrante J, Ottenstein K J, Warren J D. The Program Dependence Graph and Its Use in Optimization[J]. ACM Transactions on Programming Languages and Systems, 1987, 9(3):319-349

[18] Chen Yong-yan, Shu Hong-chun, Dai Wei. Function Vulnerability Detection Method Based on Parse Tree[J]. Computer Science, 2013, 40(8):119-123(in Chinese)
陈永艳,束洪春,戴伟. 基于语法解析树的函数漏洞发现方法[J]. 计算机科学,2013,40(8):119-123

[19] Howard M, LeBlanc D, Viega D J. 24 Deadly Sins of Software Security: programming flaws and how to fix them[M]. 董艳,包战,程文俊,译. 北京:清华大学出版社,2006

[20] Lv Lei, Liu Hong, Li Xin. Method of Building Control Dependence Sub-graph[J]. Computer Engineering, 2009, 35(15):50-52(in Chinese)
吕蕾,刘弘,李鑫. 一种建立控制依赖子图的方法[J]. 计算机工程,2009,35(15):50-52

[21] Zheng Bian-hong. Generating of Static Call Graph and Use Case Model[D]. Xi'an: Xidian University, 2007(in Chinese)
郑变红. 静态程序依赖图和用例模型的生成[D]. 西安:西安电子科技大学,2007

[22] Horwitz S, Reps T, Binkley D. Interprocedural Slicing Using Dependence Graphs[J]. ACM Transactions on Programming Languages and Systems, 1990, 12(1):26-60

[23] NIST[OL]. http://samate.nist.gov/SARD/view.php