中国科学 : 信息科学　2014 年　第 44 卷　第 1 期 : 108–129
www.scichina.com　info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
软件维护与演化技术专刊 · 论文

一种基于一阶逻辑的软件代码安全性缺陷静态检测技术

秦晓军*, 甘水滔, 陈左宁

江南计算技术研究所, 无锡 214083
* 通信作者. E-mail: xjqin@163.com

收稿日期: 2013–08–13; 接受日期: 2013–11–25

摘要　软件代码安全性缺陷是可能引发软件系统高危后果的一类重要缺陷, 针对该类缺陷的自动化检测和定位技术在软件维护和演化研究领域具有重要意义. 本文提出并实现了一种形式化检测方法 —— 基于一阶逻辑的软件代码安全性缺陷静态检测方法, 利用命题逻辑和谓词逻辑定义模式路径公式, 引入多个与依赖关系相关的谓词构造逻辑函数表达式, 作为模式路径节点产生的制导条件, 实现了对多类软件代码安全性缺陷的形式化描述, 把安全性缺陷检测问题转化成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. 实验结果表明, 该方法能适用于大多数类型的软件代码安全性缺陷检测, 在对 openssl, wu-ftpd 等 13 个开源程序的测试中, 准确重现了 10 个已公开安全漏洞, 发现 2 个未公开安全漏洞. 并且, 与现有的模型检验等形式化静态分析方法相比, 该方法的测试时间和代码规模成渐近线性关系.

关键词　软件代码安全性缺陷　一阶逻辑　模式路径　静态分析　形式化描述

1　引言

软件安全性缺陷是能够损害软件系统和运行环境的机密性、完整性、可用性、抗抵赖性的一类软件缺陷, 在信息安全领域, 软件安全性缺陷也被称之为 “软件脆弱性”[1,2]. 软件安全性缺陷可能导致对信息系统的非授权访问和拒绝服务等攻击, 引发严重后果, 对国防、外交、金融等领域至关重要. 对 CERT 公布的从 2000 年 1 月到 2013 年 1 月共约 2911 条脆弱性条目的审查结果表明, 95% 以上的软件安全性缺陷来自于程序代码的编写和调试过程 1). 随着社会对软件需求的日益旺盛, 软件代码规模不断增长, 每年都有大量的人力物力花费在安全性缺陷的检测和维护上, 高效的软件代码安全性缺陷自动化检测和定位技术在软件维护和演化研究领域具有重要意义.

软件代码的安全性缺陷静态分析方法一般基于缺陷属性描述模型或正确属性描述模型, 在不运行测试程序的情况下, 利用自动化分析方法检测缺陷相关的数值敏感, 流敏感、上下文敏感以及路径敏感等特征. 针对静态分析方法的软件代码安全性缺陷检测能力评估, 需要综合考虑软件安全性缺陷类型检测的完备性、虚报率、漏报率、检测速度、检测规模、资源要求等不同技术指标 [3,4]. 从 20 世

1) Cert Vulnerability Notes Database, http://www.kb.cert.org/.

引用格式: 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. 中国科学: 信息科学, 2014, 44: 108–129, doi: 10.1360/N112013-00095

纪 80 年代开始, 静态分析在词法/语法分析 [5]2)、控制流分析 [6]、数据流分析 [7]、类型理论 [8]、定理证明 [9]、模型检验 [10~12]、抽象解释 [13] 等方法的支持下, 取得了软件缺陷检测能力方面的显著进展, 但从代码安全缺陷检测的角度看, 这些分析方法存在各自的缺点 [4], 如: 基于词法和语法的检测方法虚报率高, 并且很难检测到流敏感相关的安全性缺陷; 数据流分析技术扩展了检测类型, 但对时序安全相关的安全性缺陷漏报率高, 同时解决不了整体上虚报率高的问题; 控制流分析技术解决了时序安全相关的安全性缺陷检测问题, 但引入了呈指数增长的路径遍历问题 [6]; 类型理论借助自动机推导函数和变量的类型, 判定对函数和变量的访问是否符合预先设定的缺陷类型规则, 适用于控制流敏感的程序分析, 能识别程序中指针变量的别名问题, 而且可判定部分条件竞争安全性缺陷, 但只能作为提升静态分析效率的辅助技术; 模型检验利用自动机或数值向量描述软件代码的安全缺陷, 类型覆盖率高, 但面临着程序状态空间的遍历问题, 在大规模代码的检测上困难较大, 并且仍然存在虚报率高的问题 [14]; 抽象解释通过定义不同的抽象域, 大大缩减了测试目标的状态空间, 解决了循环结构中变量边界值的估计, 但无法有效追踪变量之间的约束关系, 在路径可行性判定上依然存在不足 [15]; 以定理证明为基础的数据约束关系求解技术克服虚报率高的问题, 但空间消耗大, 目前只能适应小规模代码测试 [16].

随着研究的进展, 学术界陆续开发了一些具备一定实用性的静态分析原型系统, 也有多种商业化分析系统推出, 如: 斯坦福大学提出的 saturn3), 利用 CIL 前端对程序的抽象语法树进行编码, 并利用内部实现的 calypso 逻辑编程语言, 对中间代码进行进一步解释抽取程序约束条件、构造缺陷相关摘要信息并提交检测报告. saturn 引入 SAT 理论 (布尔可满足性) [17] 对路径可达性进行约束求解判定, 降低了虚报率, 一定程度上达到了商业静态分析工具的缺陷挖掘效率, 但由于过程间分析的不足造成了较大漏报隐患. Cousot 等提出的 ASTREE4) 是基于抽象解释理论的程序静态分析器, 在符号模型检验方法的基础上, 通过对空间状态建立多面体抽象域, 可以检测十万行代码规模程序中的数组越界访问、除零异常、浮点运算溢出和整数运算溢出等问题, 但由于对各种语法对象难以抽象限制了 ASTREE 的检测精度 [18,19]. rose 编译器工作组开发的 compass 静态分析工具 [20] 定义了统一的缺陷规则描述框架, 但只提供了少数代码缺陷类别的简单描述, 缺乏对代码缺陷的深层次语义特征抽象, 在检测效率方面也存在不足. 总的来说, 目前静态分析工具仍面临诸多需要克服或改进的技术难点, 主要包括以下 3 个方面.

1) 目前的静态分析方法大部分需要进行状态空间遍历, 所需的时空代价与测试代码行数呈指数级上升关系, 不适应软件代码行数快速增长的现状. 定理证明、模型检验等静态分析方法将缺陷检测规约成对程序设计语言的正确属性遍历的逆向问题, 但利用自动机遍历程序设计语言的正确属性构成的状态集合, 所需的时间复杂度呈指数级增长.

2) 现有的针对软件代码安全缺陷的形式化描述模型通用性或完备性不够, 因此, 所构造系统的检测能力具有很强的局限性, 这是无法验证和确保软件系统安全性的重要原因.

3) 现有的静态分析方法在缺陷类型覆盖能力、检测速度、虚报率、漏报率、可测代码规模等方面具有不同的特性, 没有一种静态分析方法能够在以上指标方面同时具备优势, 现在市场上的 coverity

2) RATs, http://www.securesw.com/rats/; Flawfinder, http://sourceforge.net/projects/flawfinder/.
3) http://saturn.stanford.edu/.
4) http://www.astree.ens.fr/.

prevent [21]5), klocwork insight6), codesonar7), polyspace8) 等商业化静态分析工具基本上集成了多数有代表性的静态分析方法, 虽然已体现出一定的实用价值, 但在上述评估指标上还有较大的改进余地 [22].

为解决测试程序的状态规模爆炸问题, 本文提出了一种模式路径 (pattern path, PP) 公式, 作为软件代码安全缺陷形式化描述的通用基础模型, 可以应用于文献 [22, 23] 中涉及的所有缺陷类别, 模式路径是对系统依赖图进行精简和转换之后的模型化描述, 利用模式路径公式对软件代码的安全性缺陷进行形式化描述和检测具备如下优势.

1) 模式路径确定了初始状态规模, 通过制导条件推导状态转移条件及下一个状态, 这样将缺陷检测问题规约成模式路径公式的存在性判定问题, 大大减少了检测的时间复杂度, 后文中的实验数据可以看到, 模式路径制导方法的检测时间和测试代码行数呈渐进线性增长关系;

2) 模式路径可以表达程序中变量、函数间的时序关系, 能够描述时序相关的软件代码安全缺陷类型;

3) 模式路径可以刻画程序控制流和数据流信息, 能够描述控制流敏感和数据流敏感的软件代码安全缺陷类型, 进一步的, 模式路径可以推广至由数据类型转换和运算所引发的缺陷类型;

4) 基于模式路径的缺陷形式化描述方法可用于模型检验、定理证明、符号执行、类型推断等静态分析方法的前端, 实现对上述方法的优化.

本文提出的 PPDdetector 软件代码安全性缺陷静态挖掘模型以模式路径为基础, 目标软件代码的模式路径集合生成后, 一方面可以用基于模式路径公式描述的缺陷规则作为缺陷检测的制导条件, 计算可能存在的安全缺陷, 即本文提出的模式路径制导 (pattern path directed, PPD) 方法; 另一方面, 模式路径集合中包含的状态空间比系统依赖图大大减少, 可使用模型检验、类型推断、定理证明等静态分析方法实现对代码的正确性验证, 反证代码中的安全缺陷, 大大优化这些方法的覆盖率、虚报率、漏报率等关键指标.

PPDdetector 模型的核心在于模式路径公式的定义和缺陷的形式化描述, 另外, 约束求解模块和类型分析模块也是模型的重要组成部分. 本文第 2 节重点描述 PPDdetector 模型框架、模式路径公式定义、多类缺陷的模式路径公式描述、核心模式路径公式相关的数据结构设计和时间复杂度分析, 第 3 节对 13 个不同代码规模的开源软件进行测试, 用实验数据对 PPDdetector 模型的缺陷类型覆盖能力和时间复杂度等指标进行具体的评估. 第 4 节总结全文.

2 基于模式路径制导的软件代码安全性缺陷检测模型

2.1 模型框架及功能模块分析

对于给定的被测软件代码, 基于模式路径制导的代码安全性缺陷检测模型 PPDdetector 的主要执行流程如下.

1) 利用 rose 编译器 9) 提供的中间代码生成前端 (EDG front-end parser) 解析生成抽象语法树 (AST), 从中提取控制依赖图 (CDG)、数据依赖图 (DDG) 和控制流图 (CFG), 并结合过程间分析构

5) http://www.coverity.com/.
6) http://www.klocwork.com/.
7) http://www.grammatech.com/products/codesonar/overview.html.
8) http://www.mathworks.cn/products/polyspace/.
9) http://rosecompiler.org/.

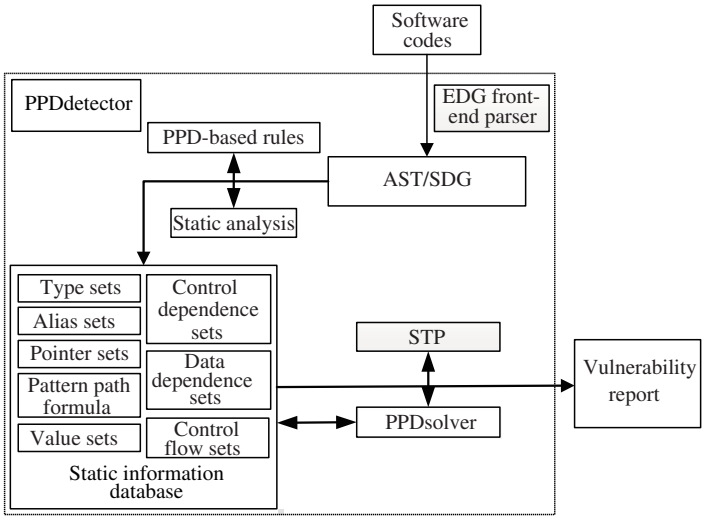

图 1 PPDdetector 模型

Figure 1 PPDdetector module

造系统依赖图 [24] (SDG);

2) 基于步骤 1) 中生成的中间代码表示形式 AST 和 SDG, 静态提取与模式路径初始节点相关的对象信息, 主要包括相关的变量类型信息 (type sets)、变量值信息 (value sets)、别名信息 (alias sets, 别名识别参考了文献 [25] 中的方法)、控制流信息 (CF sets)、数据依赖信息 (DD sets) 和控制依赖信息 (CD sets) 等, 这些静态程序信息以及中间代码形式存储在 BDB (SQL Berkley data base) 数据库 [10] 中;

3) 初始信息提取完成后, 利用模式路径公式求解器 PPDsolver (使用 Ocaml 语言 [11] 实现), 根据输入节点信息和制导条件 (以逻辑表达式形式存储在 BDB 数据库中) 计算出输出节点信息;

4) 如果系统依赖图中存在从输入节点开始, 满足模式路径制导条件的输出节点, 则将输出节点信息保存至 BDB 数据库中, 并将本轮的输出节点作为下一轮计算的输入节点, 回到步骤 3), 循环执行, 直到整条模式路径公式求解完毕; 如果不存在满足路径制导条件的输出节点, 求解过程结束;

5) PPDsolver 求解完成后, 在 BDB 数据库中将保存有被测代码的整个求解空间, 根据代码安全性缺陷判定规则判定代码中是否存在安全缺陷. 考虑到即使步骤 4) 中满足模式路径制导条件的输出节点集合不为空, 但被测软件代码在实际执行过程中, 有可能存在步骤 4) 中输入节点无法执行到输出节点, 即输出节点实际不可达的情况, 为提高检测效率, 在步骤 4) 中计算出输出节点后, 则立即抽取输入节点和输出节点间的约束条件, 使用 STP 约束求解器 [26] 进行求解 (PPDsolver 提供了自动符号映射作为 STP 约束求解器的输入), 如果有解 (代表从输入节点可达输出节点), 则按步骤 4) 正常流程将求解的节点信息集合保存至 BDB 数据库中, 继续执行下一步骤, 如果无解, 则计算过程结束.

图 1 示意了 PPDdetector 模型的主要流程. 与 saturn 相比, PPDdetector 中的 BDB 数据库存储的信息是动态变化的, 降低了空间开销 [27].

模式路径的核心是基于命题逻辑、谓词逻辑、以及一组以谓词逻辑为变量的逻辑函数定义构造模式路径公式, 在此基础上对软件代码的各类安全性缺陷进行形式化描述, 把代码缺陷的检测转化为有

10) http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

11) http://caml.inria.fr/ocaml/.

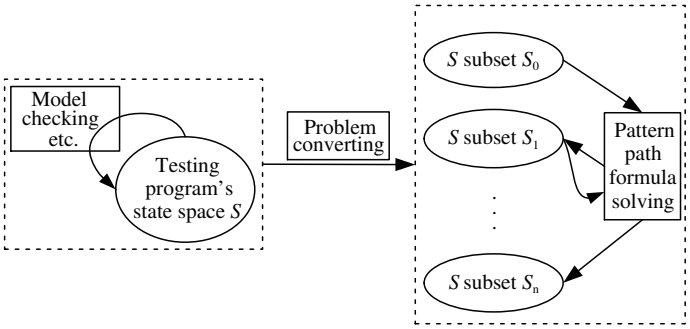

图 2　测试程序静态分析的问题转换

Figure 2　Problem converting of testing software's static analysis

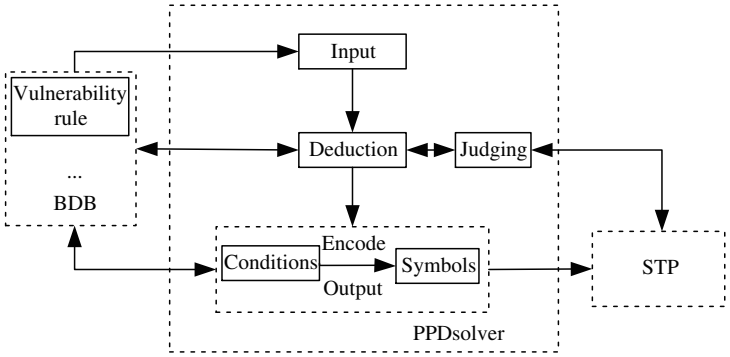

图 3　模式路径求解器组成结构和接口

Figure 3　Structure and interface of PP's solver

限状态空间的模式路径公式求解问题, 如图 2 所示.

PPDdetector 模型预先提取和存储模式路径公式中的初始语句节点和制导语义信息, 避免了对被测代码整个状态空间进行多次遍历和重复计算, 大大缩减了静态分析的时间开销, 模型主要包括如下关键技术环节.

1) 不同类别安全缺陷的机制原理分析. 参照 Tripathi 等 [23] 和 Howard 等 [28] 对代码安全缺陷的机制原理分析, 提取各类缺陷中涉及的词法、语法的数据依赖关系、执行顺序关系、控制依赖关系.

2) 模式路径公式定义. 在命题逻辑、谓词逻辑基础上, 定义制导蕴涵规则, 作为构造模式路径公式的基础, 模式路径公式明确了初始状态、制导条件 (状态转移条件) 以及终止状态, 确定了用于描述各类安全缺陷判定规则的统一框架.

3) 判定规则构造. 在模式路径公式基础上, 结合代码中各类安全性缺陷中涉及的语义, 利用数据依赖、控制依赖以及代码语句间的支配、执行次序等各种关系构造制导条件, 为缺陷语句的状态转移提供条件, 最后通过一组由谓词定义的判定规则判定缺陷的存在性.

4) 模式路径公式求解器 PPDsolver 设计. 主要包括关键程序状态数据结构设计和存储, 以及模式路径公式求解器的组件和接口设计. PPDsolver 由输入模块、推导模块、输出模块及判定模块四部分组成, 如图 3 所示. 输入模块把 BDB 数据库事先存储好的缺陷相关的模式路径公式初始状态集合作为输入, 然后推导模块按照模式路径公式中的制导条件 (逻辑表达式) 进行推导, 并根据推导结果决定

是否进行下一轮求解. 可达性验证过程中, 约束求解器中所需的路径可达约束条件从 BDB 数据库中抽取, 输入符号由输出模块中的符号映射组件生成. 整个过程存在多个交互的环节, 可以从下文中给出的模式路径公式定义解释其中涉及的输入、推导、输出、判定等功能.

5) 相关功能模块的选择. BDB 数据库可以很好的适应 PPDdetector 模型中频繁的数据访问和写入, 模式路径公式求解器的主要工作量集中在逻辑表达式的推导和约束条件的提取, 使用 BDB 数据库, 每个步骤都可以较快速的完成. 约束求解器的使用是为了进一步降低虚报率, 选择较为成熟的 STP 求解器而不是另行设计实现.

2.2 基于依赖关系逻辑表达式的模式路径制导公式

在本文的工作中, PPDdetector 模型已被证实能够检测内存释放后错误操作、指针误用、资源泄露、认证缺失、格式串溢出、不安全系统调用、整数溢出、竞争条件、资源保护不当等 9 类软件代码安全性缺陷 [23,28], 在命题逻辑基础上构造模式路径公式是描述各类缺陷判定规则的基础, 针对每一种缺陷, 利用模式路径公式对缺陷的语法规则进行描述, 并构造对应的谓词逻辑公式作为判定规则, 定位软件代码中具体的缺陷语句或执行路径. 2.3 小节对上述 9 类缺陷中的 4 类进行详细的形式化描述.

定义 1 (制导蕴涵规则) 制导蕴涵规则定义为带条件的蕴涵规则 $p \xrightarrow{c} q$, 表示 “如果 p 成立, 且满足条件 c, 那么 q 成立”, 必须满足以下 4 条规则:

1) 制导蕴涵的对等性规则: $p \xrightarrow{c} q \dashv\vdash q \xrightarrow{c} p$;

2) 制导蕴涵的去条件逻辑等价转换规则: $p \xrightarrow{c} q \dashv\vdash (p \wedge c \to q) \wedge (q \wedge c \to p)$;

3) 制导蕴涵的向左结合规则: 对于制导蕴涵规则公式 $p \xrightarrow{c} q \xrightarrow{d} g$, 满足 $p \xrightarrow{c} q \xrightarrow{d} g \dashv\vdash (p \xrightarrow{c} q) \xrightarrow{d} g \dashv\vdash g \xrightarrow{d} (p \xrightarrow{c} q)$;

4) 制导蕴涵的条件从右向左消去规则: 对于制导蕴涵规则公式 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f$, 如果 $\neg(e \to g)$, 满足 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash p \xrightarrow{c} q \xrightarrow{e} f$; 如果 $\neg(e \to g) \wedge \neg(e \to q)$, 满足 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash p \xrightarrow{e} f$.

定义 2 (模式路径的命题逻辑) 模式路径是一个由有限个原子语句和制导蕴涵规则构造的公式 $p_0 \xrightarrow{\mathrm{In}_1} p_1 \cdots p_{m-1} \xrightarrow{\mathrm{In}_m} p_m \xrightarrow{\mathrm{In}_\varepsilon} p_\varepsilon$, 其中 $p_0, \ldots, p_{m-1}, p_m, p_\varepsilon$,$\mathrm{In}_1, \ldots, \mathrm{In}_m, \mathrm{In}_\varepsilon$ 为原子语句, p_0 表示 “节点 n_0 隶属于一个确定的非终止节点有限集合 set”, p_i ($1 \leqslant i \leqslant m$) 表示 “节点 n_i 是非终止节点”, p_ε 表示 “节点是终止节点 ε”, In_i ($1 \leqslant i \leqslant m$) 表示 “制导动作为真”, In_ε 表示 “制导动作为非真”.

定义 3 (逻辑表达式制导的模式路径) 满足模式路径的命题逻辑定义, 对相应的原子语句中的对象赋予更具体的属性, 一条依赖关系制导的模式路径用 $n_0 \xrightarrow{\mathrm{InductRelation}_1(N_1,n_1)} n_1 \cdots n_{m-1} \xrightarrow{\mathrm{InductRelation}_m(N_m,n_m)} n_m \xrightarrow{\mathrm{InductRelation}_\varepsilon(N_{m+1},\varepsilon)} \varepsilon$ 描述, 具备以下几个约束条件:

1) 有且仅有一个初始路径节点 n_0 和一个终止节点 ε, 初始路径节点和终止节点中间包含有限 $(m \geqslant 0)$ 个路径节点的有序连接序列.

2) 路径模式中的 $m+2$ 个节点有序连接通过节点间的制导动作用制导蕴涵规则中的制导关系 InductRelation 完成.

3) 节点分 3 种类型, 非空节点、空节点 NULL、终止节点 ε. 符合命题逻辑定义, 非终止节点限制为非空节点、空节点 NULL.

4) 模式路径中从节点 n_i 到达非终止节点 n_{i+1} 当且仅当 $\mathrm{InductRelation}_{i+1}(N_{i+1}, n_{i+1}) == \mathrm{REAL}$. 其中 $\mathrm{InductRelation}_{i+1}$ 由一组二值函数构造的逻辑表达式表示, N_{i+1} 为已知的节点集合, $N_{i+1} \subseteq \{n_0, \ldots, n_i\}$, n_{i+1} 为 $\mathrm{InductRelation}_{i+1}$ 在有限状态空间存在的非终止节点. 如果 $\nexists n_{i+1}$, 则满足 $\mathrm{InductRelation}_{i+1}(N_{i+1}, n_{i+1}) == \mathrm{REAL}$, 那么 n_{i+1} 为空节点 NULL, 记为 $\mathrm{InductRelation}_{i+1}$ $(N_{i+1},$

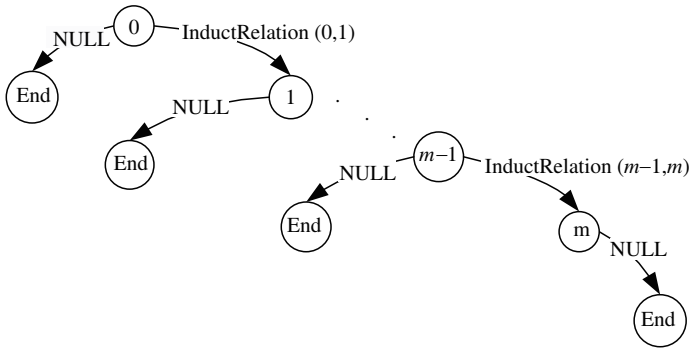

图 4　一颗逻辑表达式制导模式路径树

Figure 4　A directed PP tree based on logic expression

NULL) == REAL.

5) 模式路径中从节点 n_m 到达终止节点 ε 当且仅当 i) 制导动作为非真, 记为 InductRelation$_{m+1}$ (N_{m+1}, ε) == FALSE; ii) N_{m+1} 只包括初始节点 n_0.

6) 约定 InductRelation$_\varepsilon$ (n_0, ε) 记作为符号 "$\emptyset$".

定义 4 (逻辑表达式制导的模式路径树)　一棵逻辑表达式制导的模式路径树任何一条路径满足模式路径的命题逻辑定义, 并具备以下条件:

1) 模式路径树包含一条节点数量最多的主干模式路径, 主干模式路径节点数量为 $m+2$ ($m \geqslant 0$), 主干模式路径仅有一个非真制导动作;

2) 模式路径树由主干模式路径生成, 主干模式路径的初始节点为模式路径树的根节点, 对于主干模式路径上的非终止节点 $0, 1, \ldots, m-1$, 将生成一个由非真制导产生的终止节点 (图 4);

3) 一颗模式路径树的主干模式路径用 pp 表示, 那么模式路径树用 $\mathcal{L}(pp)$ 表示.

性质 1　针对一条主干模式路径公式可通过制导蕴涵的条件从右向左消去规则推导出模式路径树的任意路径构成的模式路径公式.

证明　假设该主干模式路径公式为 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f$, 根据该性质条件可知, 该公式满足初始节点和任意制导条件对应的命题为真, 即 p, c, d, e 为真. 根据定义 3 的第 4 条约束条件, 可得到 q, g, f 等节点对应的命题为真. 根据制导蕴涵的向左结合规则可得 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash \left(p \xrightarrow{c} q \xrightarrow{d} g\right) \xrightarrow{e} f$; 根据制导蕴涵的去条件逻辑等价转换规则, 满足 $\left(p \xrightarrow{c} q \xrightarrow{d} g\right) \xrightarrow{e} f \vdash \left(p \xrightarrow{c} q \xrightarrow{d} g\right) \wedge e \to f$; 根据制导蕴涵的对等性规则, 满足 $\left(p \xrightarrow{c} q \xrightarrow{d} g\right) \wedge e \to f \vdash f \wedge e \to \left(p \xrightarrow{c} q \xrightarrow{d} g\right)$; 结合条件 e, f 为真, 满足 $f \wedge e \to \left(p \xrightarrow{c} q \xrightarrow{d} g\right), f, e \vdash p \xrightarrow{c} q \xrightarrow{d} g$; 由以上结论, 满足 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash p \xrightarrow{c} q \xrightarrow{d} g$; 同理, 可推导出 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash p \xrightarrow{c} q \xrightarrow{d} g, p \xrightarrow{c} q \vdash p$; 最后, 可推导出 $p \xrightarrow{c} q \xrightarrow{d} g \xrightarrow{e} f \vdash \left(p \xrightarrow{c} q \xrightarrow{d} g\right) \wedge \left(p \xrightarrow{c} q\right) \wedge p$.

即可通过主干模式路径公式推导出其对应模式路径树的任意路径构成的模式路径公式. 该性质表示, 当一颗制逻辑表达式制导模式路径树的制导动作恒为真时, 那么它包含任意路径 (从根节点到任意叶子节点以及节点间的制导关系构成的路径) 是以主干模式路径命题逻辑公式为前提的结论.

定义 5 (模式路径的制导秩)　模式路径

$$\mathrm{pp}: n_0 \xrightarrow{\mathrm{InductRelation}_1(N_1,n_1)} n_1 \cdots n_{m-1} \xrightarrow{\mathrm{InductRelation}_m(N_m,n_m)} n_m \xrightarrow{\mathrm{InductRelation}_\varepsilon(N_{m+1},\varepsilon)==\mathrm{FALSE}} \varepsilon$$

算法 1 最优制导模式路径计算

输入: pp, 一条如定义 5 所示的模式路径

输出: order, 模式路径 pp 中每个节点产生的最优制导模式路径构成的制导秩集合

1: **for** $i = 1$ to m (m 表示 pp 的节点数量) **do**
2: order(OptimalPath$_i$) =$\ll n_0 \gg$ (OptimalPath$_i$ 表示 pp 中第 i 个节点的最优制导路径)
3: **end for**
4: order $\leftarrow$ {order(OptimalPath$_0$)}
5: **for** $i = 1$ to m **do**
6: **for** $j = 1$ to $i - 1$ **do**
7 **if** $n_j \in N_i$ **then**
8: order$(n_i) \leftarrow$ order$(n_i) \cup$ order(n_j)
9: 制导秩的有序合并操作如 $\ll n_0, n_1 \gg \cup \ll n_1, n_3 \gg = \ll n_0, n_1, n_3 \gg$
10: **end if**
11: **end for**
12: order $\leftarrow$ order $\cup$ {order(OptimalPath$_i$)}
13: **end for**
14: **return** order

的制导秩 order(pp) 用于描述模式路径 pp 中节点的制导秩序, 用 $\ll n_0, n_1, \ldots, n_m \gg$ 描述, 通过 order(pp) 可映射到该模式路径 pp.

性质 2 最优制导模式路径 (最少数量节点制导模式路径) . 针对上述定义的模式路径树, 结合性质 1 以及定义 3 的第 4 条性质, 每个节点的产生存在一条最优制导模式路径, 产生算法如算法 1 所示.

证明 该算法在计算最优制导路径的节点过程中, 采用的是动态规划算法, 每个节点的模式路径的制导秩需要做初始化, 算法 1 的第 1~3 条语句就是做初始化, 每个节点的模式路径的制导秩生成过程是必须以前一个节点的模式路径制导秩为前提, 所以需要先计算和存储好前一个节点的最优模式路径制导秩, 以计算下一个节点的最优模式路径制导秩, 算法 1 的第 5~10 条语句就是对这个过程的描述, 因此, 最后生成的每个节点的制导模式路径都是最优的.

性质 3 由模式路径公式 pp 产生的任一节点的最优制导模式路径 $\mathrm{pp}_{\mathrm{opt}}$, 满足 $\mathrm{pp} \vdash \mathrm{pp}_{\mathrm{opt}}$.

证明 如性质 1 证明过程.

推论 1 一颗主干模式路径节点为 $m + 2$ ($m \geqslant 0$) 的模式路径树包含模式路径数量 pathnum $\leqslant 1 + \sum_{i=1}^{m} \frac{m!}{(m-i)!}$.

证明 考虑极端情形, 当每个中间节点的制导关系只跟初始节点有关, 也就是中间节点无相关性, 这样产生的模式路径为 $C_m^0 + \sum_{i=1}^{m} C_m^i * i!$.

定义 6 (模式路径识别函数) 路径识别函数 PPI 定义如下:

$$\mathrm{PPI}\,(\mathcal{M}, \mathrm{PatternPath}) = \begin{cases} 1, \text{ if PatternPath} \in \mathcal{M}; \\ 0, \text{ if PatternPath} \notin \mathcal{M}, \end{cases} \tag{1}$$

其中 $\mathcal{M}$ 表示有限个节点、节点属性以及节点间的关系构成的有限状态信息空间, PatternPath 表示某条模式路径.

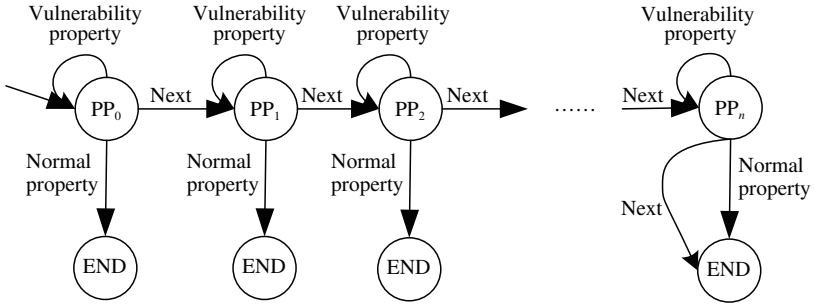

图 5　判定过程描述

Figure 5　Description of judging procedure

定义 7 (软件代码安全性缺陷描述规则)　软件代码安全性缺陷描述规则 VulnerabilityRule 由一个四元组表示, $\text{VulnerabilityRule} = (\Omega, n_0, \text{SyntaxRule}, \text{JudgeRule})$, Ω 表示描述规则应用的节点、语法以及制导信息构成的程序空间, SyntaxRule 表示语法规则, JudgeRule 表示判定规则, n_0 为初始程序节点. 规则构造如下:

• 语法规则. $\text{VulnerSyntaxRule} = \{\mathcal{L}(\text{pp}), \text{PP}\}$, $\mathcal{L}(\text{pp})$ 是一颗由主干模式路径 pp 生成的模式路径树, PP 是 $\mathcal{L}(\text{pp})$ 上的模式路径子集, 满足 $\forall \text{pp}_i \in \text{PP}, \text{pp} \vdash \text{pp}_i$. 模式路径树上的制导动作集合 $\text{Induct} = \{\text{CD}, \text{DD}, \text{Pred}, \text{Succ}, \text{PD}\}$, 包括控制依赖制导、数据依赖制导、前驱制导、后继制导、支配制导. 程序节点分三种状态的实体, 一种是正常的程序语句 $\text{statement} = \{n | n \in N \wedge N \subset \Omega\}$, 一种是空语句 NULL, 一种是终止语句 ε. 语句到语句之间的制导关系是一个或多个制导动作构造的逻辑表达式 InductRelation, 具体制导关系用如下产生式表示.

$$\text{statement} \xrightarrow{\text{InductRelation(statement,statement|NULL)==REAL}} \text{statement|NULL},$$

$$\text{NULL} \xrightarrow{\text{InductRelation(NULL,NULL)==REAL}} \text{NULL},$$

$$\text{statement|NULL} \xrightarrow{\text{InductRelation(statement|NULL,}\varepsilon\text{)==FALSE}} \varepsilon.$$

对于一条语法规则 $\text{SyntaxRule} = \{\mathcal{L}(\text{pp}), \text{PP}\}$, 模式路径子集 $\text{PP} = \{\text{pp}_0, \text{pp}_1, \ldots, \text{pp}_n\}$, 其中的任意模式路径 pp_i 包含语法属性 $\text{atti}(\text{pp}_i)$, 描述为

$$\text{atti}(\text{pp}_i) = \begin{cases} 1, \text{ 正常特性}, \\ 0, \text{ 脆弱特性}. \end{cases} \tag{2}$$

• 判定规则. SyntaxRule 是一个由谓词逻辑公式构成的集合, 记作 $\text{SyntaxRule} = \{\ \text{judge}(\text{pp}_i) \mid \text{pp}_i \in \text{PP} \wedge \text{PP} \subset \mathcal{L}(\text{pp}), 0 \leqslant i \leqslant n\ \}$, 针对语法规则中的每一条模式路径, 都会有相应的谓词逻辑公式, 根据 SyntaxRule 中的规则属性构造判定条件, 图 5 给出了判定过程示意图, 具体描述如算法 2.

性质 4 (初始节点的确定性)　任一程序代码安全性缺陷描述规则的初始节点 n_0 隶属于某个确定的有限个语句构成的集合 (定义 7 中已说明, 因为一个程序的语句集合是有限的).

性质 5 (制导动作的确定性)　任一语法规则实例中的特定源节点到目标节点转换的制导动作具有确定的依赖关系逻辑表达式 (定义 7 中的制导关系产生式已说明了制导动作的确定性).

性质 6 (规则的实例化)　一条完整的软件代码安全性缺陷描述规则可以包含多个规则实例, 规则实例的数量等于语法规则中模式路径的数量, 一个规则实例包括一条语法规则的模式路径和与之对应的判定规则 (定义 7 中的语法规则定义说明了规则的实例化).

算法 2 基于模式路径制导的软件代码安全性缺陷判定过程

输入: 描述规则 VulnerabilityRule (如定义 7 所示)

输出: 描述规则 VulnerabilityRule 的判定过程 judgeprocess (如定义 7 所示)

```
for i = 0 to n do
    if atti(pp_i) == 1 then
        while traverse ω do
            if condition then
                EXIT
            end if
        end while
    else if atti(pp_i) == 0
        while traverse ω then
            if condition then
                report vulnerable behaviour
            end if
        end for
    end while
end for
```

性质 7 (判定规则的优先排序) 整个判定过程对模式路径 $\mathrm{pp}_0, \mathrm{pp}_1, \ldots, \mathrm{pp}_n$ 有序的判定, 一旦出现某个语法规则描述正常特性的 pp_i 满足判定条件, 立即结束判定 (如图 5 所示).

定义 8 (软件代码安全性缺陷描述规则下的模式路径求解空间) 对于一条软件代码安全性缺陷描述规则 VulnerabilityRule $= (\Omega, n_0, \mathrm{SyntaxRule}, \mathrm{JudgeRule})$ 下的某条定义任意属性的模式路径

$$\mathrm{pp}: n_0 \xrightarrow{\mathrm{InductRelation}_1(N_1,n_1)} n_1 \cdots n_{m-1} \xrightarrow{\mathrm{InductRelation}_m(N_m,n_m)} n_m \xrightarrow{\mathrm{InductRelation}_\varepsilon(N_{m+1},\varepsilon)} \varepsilon,$$

其求解空间用 $\mathbb{S}(\mathrm{pp})$ 描述, 求解空间规模用 $|\mathbb{S}(\mathrm{pp})|$ 描述, 其每个节点 n_i 的解空间表示为集合 $\mathrm{set}(n_i)$, 对于解 $S \in \mathbb{S}(\mathrm{pp})$, 用 $(s_0, s_1, \ldots, s_m)$ 表示, 当且仅当

$$s_0 \in \mathrm{set}(n_0) \xrightarrow{\mathrm{InductRelation}_1(s_0,s_1)==\mathrm{REAL}} s_1 \cdots s_{m-1} \xrightarrow{\mathrm{InductRelation}_m(s_0,s_1,\cdots,s_{m-1},s_m)==\mathrm{REAL}}$$

$$s_m \xrightarrow{\mathrm{InductRelation}_\varepsilon(s_0,\varepsilon)} \varepsilon.$$

推论 2 (模式路径的解空间规模计算) 对于一条软件代码安全性缺陷描述规则 VulnerabilityRule $= (\Omega, n_0, \mathrm{SyntaxRule}, \mathrm{JudgeRule})$ 下的某条定义任意属性的模式路径

$$\mathrm{pp}: n_0 \xrightarrow{\mathrm{InductRelation}_1(N_1,n_1)} n_1 \cdots n_{m-1} \xrightarrow{\mathrm{InductRelation}_m(N_m,n_m)} n_m \xrightarrow{\mathrm{InductRelation}_\varepsilon(N_{m+1},\varepsilon)} \varepsilon,$$

满足如下性质:

1) 每个节点 n_i 的解空间表示为集合 $\mathrm{set}(n_i)$ 满足关系 $|\mathrm{set}(n_0)| \leqslant \cdots \leqslant |\mathrm{set}(n_m)|$;

2) 其解空间规模 $|\mathbb{S}(\mathrm{pp})| = |\mathrm{set}(n_m)|$;

3) 求解的制导操作总次数为 $\sum_{i=0}^{m} |\mathrm{set}(n_i)|$;

4) $\mathbb{S}(\mathrm{pp})$ 集合可以用图 6 中的树进行描述, 树的根节点为解空间的制导节点, 树的第 i 层包含的所有节点构成了集合 $\mathrm{set}(n_{i-1})$, 从树的根出发到叶子节点的任意一条路径描述为 $\mathbb{S}(\mathrm{pp})$ 集合中的一

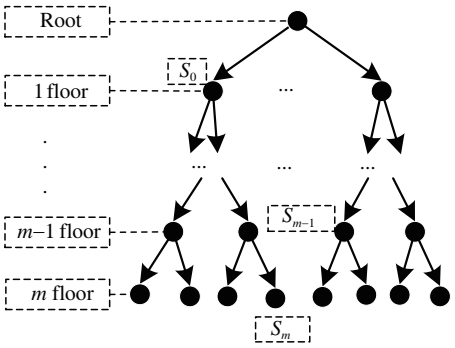

图 6　模式路径求解空间

Figure 6　PP solving space

个元素, 即为模式路径 pp 在 Ω 中的一个解. $\forall s_i \in \text{set}(n_i)$, $\text{children}(s_i)$ 表示为 s_i 节点的孩子集合, $\text{farther}(s_i, j)$ 表示 s_i 节点的 j 重父亲, 即 $s_0 = \text{farther}(s_1, 1) = \text{farther}(s_2, 2) = \text{farther}(s_m, m)$, 必满足 $\text{set}(n_{i+1}) = \bigcup_{s_i \in \text{set}(n_i)} \text{children}(s_i)$.

定义 9 (软件代码安全性缺陷描述规则中判定规则的谓词定义)　对于一条软件代码安全性缺陷描述规则 $\text{VulnerabilityRule} = (\Omega, n_0, \text{SyntaxRule}, \text{JudgeRule})$ 下的某条定义任意属性的模式路径

$$pp: n_0 \xrightarrow{\text{InductRelation}_1(N_1,n_1)} n_1 \cdots n_{m-1} \xrightarrow{\text{InductRelation}_m(N_m,n_m)} n_m \xrightarrow{\text{InductRelation}_\varepsilon(N_{m+1},\varepsilon)} \varepsilon,$$

对应定义的语法规则, 判定规则定义的一组谓词如下:

$\text{SET}_i(s)$: $s \in \text{set}(n_i)$,

$N(s)$: $s == \text{NULL}$,

$\text{CH}(s, y)$: $s \in \text{children}(y)$,

$\text{FA}(s, y)$: $s \in \text{farther}(y)$,

$\text{EQ}(s, y)$: $s == y$,

$\text{LT}(n)$: $\text{Lessthan}(n) == \text{REAL}$,

$\text{VUL}_1(s, y)$: $s, \ldots, y$ is a vulnerable trace,

$\text{VUL}_2(s)$: s is a vulnerable statement,

EXIT: exit the analysis.

在对应的有限状态空间中, 对模式路径进行求解有两种策略: 广度优先求解策略和深度优先求解策略, 这两种求解方式在时间开销上基本无差异, 本文中使用深度优先求解策略, 对于一条缺陷描述规则中语法规则 (模式路径公式) 对应的解空间, 验证每个解是否满足判定规则对应的谓词逻辑公式, 并提取其中可满足的解.

2.3　基于模式路径制导公式的软件代码安全性缺陷形式化描述

程序依赖图 (PDG) 由控制流图、控制依赖图和数据依赖图构成, 作为一种优化的中间代码表示形式, 程序依赖图能够以结构化模型方式表达软件代码的语法和语义等关键信息. 程序依赖图分为过程内依赖图 (PDG) 和过程间依赖图 (又称系统依赖图, SDG), 过程间依赖图是过程内依赖图的从单一过程到多个过程的扩展. 本节中定义 10~18 引用了文献 [29, 30] 中关于程序依赖图的基本定义, 在此基础上, 定义 19 ~ 25 基于模式路径公式给出了代码安全性缺陷形式化描述所需的部分定义.

定义 10 (模块和节点) 一个过程的节点表示为一组连贯的语句, 即连续顺序执行的语句, 每个节点只有一个入口和一个出口, 每次控制流到达只能从节点的入口进入, 出口离开.

定义 11 (CFG) 过程内控制流图 CFG 用四元组表示, 其中 N 表示该过程的节点集合, E 是边的集合, 每条边是一个表示从节点 n_i 到节点 n_j 可能存在控制流转移的有序节点对 $\langle n_i, n_j\rangle$.

定义 12 (控制流边) 节点 n_i 到节点 n_j 之间存在一条控制流边的充要条件是, 执行过程能从节点 n_i 的语句到达节点 n_j 的语句.

定义 13 (直接前驱－直接后继) 在 CFG 中, 若 $\langle n_i, n_j\rangle \in E$, 则称 n_i 是 n_j 的直接前驱, n_j 是 n_i 的直接后继; 节点 n 的所有直接前驱组成的集合称为 n 的直接前驱集, 记为 $\mathrm{DirPred}\,(n) = \{m \in N\,|\langle m, n\rangle \in E\}$; 节点 n 的所有直接后继组成的集合称为 n 的直接后继集, 记为 $\mathrm{DirSucc}\,(n) = \{m \in N\,|\langle n, m\rangle \in E\}$; CFG 的入口节点 Entry 是一个没有前驱的节点, CFG 的出口节点 Exit 是一个没有后继的节点.

定义 14 (执行路径) 在 CFG 中, $\mathrm{Path} = \langle n_1, n_2, \ldots, n_k\rangle$ 为一语句序列, 若满足 $\forall i, 0 < i < k, n_i \in \mathrm{DirPred}\,(n_{i+1})$, 则称 Path 为 CFG 的一条可执行路径.

定义 15 (前驱－后继) 在 CFG 中, 若从 n_i 到 n_j 之间存在一条可执行路径, 则称 n_i 是 n_j 的前驱, 相反 n_j 是 n_i 的后继, 记为 $(n_i, n_j) \in \mathrm{Pred}$ 或 $(n_j, n_i) \in \mathrm{Succ}$.

定义 16 (支配节点) 在 CFG 中, 若从 n_i 到程序出口节点 Exit 的任何路径经过 n_j, 则称 n_j 是 n_i 支配 PostDominate 节点, 记作 $(n_i, n_j) \in P$.

定义 17 (控制依赖) 控制依赖用于表示控制流引起的程序实体之间的关系, 对于 CFG 中的两个节点 n_i, n_j, 若 n_j 能否被执行由 n_i 执行状态满足, 则 n_j 控制依赖于 n_i, 记为 $(n_j, n_i) \in \mathrm{CD}$, 且满足从 n_j 到 n_i 之间存在一条可执行路径 P, 对于 P 上除 n_i, n_j 外的每个节点 n, 节点 n_j 都是 n 的后必经节点, 节点 n_j 都不是 n_i 的后必经节点.

定义 18 (数据依赖) 在 CFG 中, 对于其中的节点或语句 n_i, n_j, 变量 v, 若满足 $v \in \mathrm{DEF}\,(n_i)$, $v \in \mathrm{USE}\,(n_j)$, n_i 到 n_j 之间存在一条可执行路径, 且在该路径上, 没有语句对 v 进行重新定义. 那么称 n_i 关于变量 v 数据流依赖于 n_j, 记作 $(n_j, n_i) \in \mathrm{DD}\,(v)$.

定义 19 (可能的空指针变量集合) 程序 P 中一个可能的空指针变量集合 possibleNPV 满足 $\mathrm{possibleNPV} = \{v\,|\exists n \in P, \text{ such that } v \in \mathrm{DEF}\,(n) \wedge (v = \mathrm{function\_return\_value} \vee v = \mathrm{NULL} \vee v = v' \in \mathrm{possibleNPV})\}$.

定义 20 (定义－使用－核查) 对于某个程序 P, $\mathrm{DEF}\,(n/P)$ 表示程序 P 中在语句 n 处定义的变量集合, $\mathrm{USE}\,(n/P)$ 表示程序 P 中在语句 n 处使用的变量集合, $\mathrm{DEF}\,(v/P)$ 表示程序 P 中定义变量 v 的语句集合, $\mathrm{USE}\,(v/P)$ 表示程序 P 中使用 v 的语句集合, $\mathrm{CHECK}\,(v/P)$ 表示程序 P 中核查变量 v 的条件语句集合.

定义 21 (函数调用) 对于某个程序 P, $\mathrm{FUNCTIONCALL}\,(\mathrm{function}/P)$ 表示程序 P 中调用函数 function 的语句集合. $\mathrm{CallName}\,(n)$ 表示在语句 n 处调用的函数集合. ResourceAllocateFuctionList 表示资源分配相关的函数调用集合, ResourceAllocateFuctionList = { malloc, open, fopen, new, strdup, realloc, socket, calloc }, $\mathrm{ResourceRelease}\,(\mathrm{fuction})$ 表示资源分配函数 fuction 对应的资源回收函数, ResourceAllocateFuctionList 表示资源回收相关的函数调用集合, 且 ResourceReleaseFuctionList = {ResourceRelease (fuction) | function ∈ ResourcellocateFuctionList }.

定义 22 (指针别名) 对于某个指针变量 pv, 其相应的指针别名集合用 $\mathrm{PointerAlias}\,(\mathrm{pv})$ 表示.

定义 23 (变量作用域) 一个变量 v 的作用域表示为 $\mathrm{process}\,(v)$.

表 1　若干二值函数定义

Table 1　Some definitions of boolean functions

Function name	Function definition	Value	Conditions of value	Variable description
Boolean function of judging CD	$\mathrm{CD}(n_0,n_1)$	TRUE	$(n_0,n_1)\in \mathrm{CD}$	n_0,n_1 are statements
		FALSE	$(n_0,n_1)\notin \mathrm{CD}$	
Boolean function of judging DD	$\mathrm{DD}(n_0,n_1,v)$	TRUE	$(n_0,n_1)\in \mathrm{DD}(v)$	n_0,n_1 are statements, v is variable
		FALSE	$(n_0,n_1)\notin \mathrm{DD}(v)$	
Boolean function of judging CHECK	$\mathrm{CHECK}(n,v)$	TRUE	$n\in \mathrm{CHECK}(v)$	n is statement, v is variable
		FALSE	$n\notin \mathrm{CHECK}(v)$	
Boolean function of judging Pred	$\mathrm{Pred}(n_0,n_1)$	TRUE	$(n_0,n_1)\in \mathrm{Pred}$	n_0,n_1 are statements
		FALSE	$(n_0,n_1)\notin \mathrm{Pred}$	
Boolean function of judging IsNULL	$\mathrm{IsNull}(n)$	TRUE	$n\in \mathrm{NullStatement}$	n is variable
		FALSE	$n\notin \mathrm{NullStatement}$	
Boolean function of judging IsBelong	$\mathrm{IsBelong}(n,N)$	TRUE	$n\in N$	n is statement, N is a statement set
		FALSE	$n\notin N$	
Boolean function of judging IsCall	$\mathrm{IsCall}(n,\mathrm{function})$	TRUE	$\mathrm{function}\in \mathrm{CallName}(n)$	n is statements
		FALSE	$\mathrm{function}\notin \mathrm{CallName}(n)$	
Boolean function of judging Lessthan	$\mathrm{Lessthan}(n)$	TRUE	$\lvert\mathrm{children}(n)\rvert < \lvert\mathrm{DirSuss}(n)\rvert$	n is statement in FP
		FALSE	$\lvert\mathrm{children}(n)\rvert \geqslant \lvert\mathrm{DirSuss}(n)\rvert$	

定义 24 (终止语句)　终止语句集合用 NullStatement 表示, 为包含 exit 函数和 return 函数调用的语句集合. $\mathrm{NullStatement}=\{\mathrm{FUNCTIONCALL}(\mathrm{exit}),\mathrm{FUNCTIONCALL}(\mathrm{return})\}$. 某个程序 P 中的终止语句集合表示为 $\mathrm{NullStatement}(P)=\{\mathrm{FUNCTIONCALL}(\mathrm{exit}/P),\mathrm{FUNCTIONCALL}(\mathrm{return}/P)\}$.

定义 25 (不同制导相关的若干谓词函数定义)　若干二值函数定义如表 1 所示, 第 1 列给出函数名称, 第 2 列是对应的函数定义, 第 3, 4 列分别是二值函数取值和对应的取值条件, 第 5 列是函数定义的对象属性描述.

下面作为实例, 给出了空指针引用、资源泄露、内存多次释放和竞争条件缺陷的描述规则, 数组越界等缺陷规则可以此类推, 文中不再赘述.

1) 空指针引用缺陷描述规则. 空指针引用缺陷描述规则用 NullPointerRerenfenceRule = (Ω, n_0, SyntaxRule, JudgeRule) 表示, 其中 Ω = { SDG, NPcheck, possibleNPV, DEF, USE }, $n_0\in$ DEF (v/possibleNPV), SyntaxRule = $\{\mathcal{L}(\mathrm{pp}),\mathrm{PP}\}$, $\mathrm{PP}=\{\mathrm{pp}_0\}$, $\mathrm{atti(PP)}=\{0\}$, JudgeRule=$\{\mathrm{judge}(\mathrm{pp}_i)\,|\mathrm{pp}_i\in \mathrm{PP}\wedge \mathrm{PP}\subset\mathcal{L}(\mathrm{pp})\}$.

- $\mathrm{pp}: n_0 \xrightarrow{\mathrm{DD}(n_1,n_0,v)} n_1 \xrightarrow{\mathrm{CHECK}(n_2,v)\wedge\mathrm{CD}(n_1,n_2)} n_2 \xrightarrow{\emptyset} \varepsilon$;
- $\mathrm{pp}_0: n_0 \xrightarrow{\mathrm{DD}(n_1,n_0,v)} n_1 \xrightarrow{\mathrm{CHECK}(n_2,v)\wedge\mathrm{CD}(n_1,n_2)} n_2 \xrightarrow{\emptyset} \varepsilon$;
- $\mathrm{judge}(\mathrm{pp}_0): \forall s_1\left(\mathrm{SET}_1(s_1)\wedge\neg\exists s_2\left(\mathrm{CH}(s_2,s_1)\right)\rightarrow \mathrm{VUL}_2(s_1)\right)$.

由图 7 中的代码可以发现, 语句 3 −语句 14 −语句 7 −语句 22 是一条空指针引用的执行路径, 最后在语句 22 发生空指针引用. 该问题在本文定义的缺陷规则中如下文描述和判定.

在文中空指针引用脆弱性描述规则定义的语法规则为

$$n_0 \xrightarrow{\mathrm{DD}(n_1,n_0,v)} n_1 \xrightarrow{\mathrm{CHECK}(n_2,v)\wedge\mathrm{CD}(n_1,n_2)} n_2 \xrightarrow{\emptyset} \varepsilon,$$

```
1 FUNCTION *** () {
2   char * input="sssssssss";
3   char * string=funA(input);
4   if(!string){
5     funB(string);
6   }
7   FunC(string);
8 }
9 funA(char *x){
10   if(strlen(x)<10){
11     return x;
12   }
13   slse{
14     Return 0;
15   }
16 }
17 funB(char *y)}
18   ******
19 }
20 funC(char *src){
21   char des[10]="0";
22   strcpy(des;src);
23 }
```

图 7 一段包含空指针引用脆弱性的代码

Figure 7 A fragment code containing null pointer reference

其中 $n_0 =$ { 语句 3 }, $v =$ { 变量 string }; 根据制导条件 $\mathrm{DD}(n_1, n_0, v)$ (n_1 关于变量 v 数据流依赖于 n_0), 可计算出 $n_1 =$ { 语句 5, 语句 7 }; 根据制导条件 $\mathrm{CHECK}(n_2, v) \wedge \mathrm{CD}(n_1, n_2)$, 语句 n_2 对变量 v 进行相关核查, 并且语句 n_1 控制依赖于语句 n_2, 可计算出 $n_2 =$ { 语句 4 }. 判定规则为

$$\forall s_1 \left(\mathrm{SET}_1 \left(s_1\right) \wedge \neg \exists s_2 \left(\mathrm{CH}\left(s_2, s_1\right)\right) \rightarrow \mathrm{VUL}_2 \left(s_1\right)\right),$$

该判定规则的描述是, 对于节点 n_1 中的任意语句 s_1, 如果该语句不存在模式路径数上的孩子 (即根据这一条语句制导计算出下一个节点对应的解, 节点 n_2 对应的语句 s_2), 那么语句 s_1 发生了空指针引用. 在该判定规则下可以发现, 语句 7 为空指针引用语句, 并且可以去除可能存在的虚报语句 5.

2) 资源泄露缺陷描述规则. 资源泄露缺陷描述规则用 ResouceLeakRule 表示, ResourceLeakRule = (Ω, n_0, SyntaxRule, JudgeRule), 其中 $\Omega =$ { SDG, DEF, USE, ResourceAllocateFuctionList, NullStatement, procss, Pred, Succ, PD, CD, DD }, SyntaxRule = {$\mathcal{L}$ (pp), PP }, PP = $\{\mathrm{pp}_0, \mathrm{pp}_1\}$, atti (PP) = { 1,0 }, JudgeRule = { judge (pp_i) | $\mathrm{pp}_i \in$ PP $\wedge$ PP $\subset \mathcal{L}$ (pp) }.

• pp : $n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{(c)} n_3 \xrightarrow{\emptyset} \varepsilon$. 其中, ($a$): Pred ($n_0$, n_1) $\wedge$ IsNull (n_1), (b): PD(n_0, n_2) $\wedge$ IsBelong (ResourceRelease (CallName (n_0)), CallName (n_2)) $\wedge$ Pred (n_2, n_1) $\wedge$ DD (n_2, n_0, DEF (n_0)), (c): IsBelong (ResourceRelease (CallName (n_0)), CallName (n_3)) $\wedge$ DD (n_3, n_0, DEF (n_0)) $\wedge$ PD (n_3, n_1).

• pp_0 : $n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{\emptyset} \varepsilon$, 其中, ($a$): Pred ($n_0, n_1$) $\wedge$ IsNull (n_1), (b): PD (n_0, n_2) $\wedge$ IsBelong (ResourceRelease (CallName (n_0)), CallName (n_2)) $\wedge$ Pred (n_2, n_1) $\wedge$ DD (n_2, n_0, DEF (n_0));

• judge (pp_0) : $\exists s_2 \left(\mathrm{ST}_2 \left(s_2\right)\right) \rightarrow$ EXIT;

• pp_1 : $n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_3 \xrightarrow{\emptyset} \varepsilon$, 其中, ($a$): Pred ($n_0$, n_1) $\wedge$ IsNull (n_1), (b): IsBelong (ResourceRelease (CallName (n_0)), CallName (n_3)) $\wedge$ DD (n_3, n_0, DEF (n_0)) $\wedge$ PD (n_3, n_1);

• judge (pp_1) : $\forall s_1$ (SET_1 (s_1) $\wedge\neg\exists s_3$ (FA (s_1, s_3)) $\rightarrow \mathrm{VUL}_1$ (s_0, s_1)).

3) 内存多次释放缺陷描述规则. 内存多次释放缺陷描述规则用 MemoryDoubleReleaseRule 表示, MemoryDoubleReleaseRule = (Ω, n_0, SyntaxRule, JudgeRule), $\Omega =$ { SDG, DEF, USE, ResourceAllocateFuctionList, NullStatement, procss, Pred, Succ, PD, CD, DD }, $n_0 =$ { $n | n \in \mathrm{DEF}(v/\mathrm{PV}) \wedge \mathrm{CallName}\,(n) \in$ ResourceAllocateFuctionList }, SyntaxRule = { $\mathcal{L}$ (pp), PP }, PP = $\{\mathrm{pp}_0\}$, atti(PP) = {0}, JudgeRule = { judge (pp_i) | $\mathrm{pp}_i \in$ PP $\wedge$ PP $\subset \mathcal{L}$ (pp) }.

• pp : $n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{(c)} n_3 \xrightarrow{\emptyset} \varepsilon$, 其中, (a): DD $(n_0, n_1,$ DEF $(n_0))$, (b): DD $(n_0, n_2,$ DEF $(n_0)) \wedge \neg$ IsBelong $(n_2, n_1) \wedge$ Pred (n_1, n_2), (c): (Pred $(n_3, n_2) \wedge$ PD $(n_1, n_3) \wedge$ IsNull $(n_3)) \vee$ (IsNull $(n_3) \wedge$ PD $(n_0, n_3) \wedge$ Pred $(n_3, n_1))$;

• $\text{pp}_0 : n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{(c)} n_3 \xrightarrow{\emptyset} \varepsilon$, 其中, (a): DD $(n_0, n_1,$ DEF $(n_0))$, (b): DD $(n_0, n_2,$ DEF $(n_0)) \wedge \neg$ IsBelong $(n_2, n_1) \wedge$ Pred (n_1, n_2), (c): (Pred $(n_3, n_2) \wedge$ PD $(n_1, n_3) \wedge$ IsNull $(n_3)) \vee$ (IsNull $(n_3) \wedge$ PD $(n_0, n_3) \wedge$ Pred $(n_3, n_1))$;

• judge $(\text{pp}_0) : \forall s_1 \forall s_2$ $(\text{SET}_1\ (s_1) \wedge \text{SET}_2\ (s_2) \wedge$ FA $(s_1,\ s_2) \wedge \neg \exists s_3$ (FA $(s_2,\ s_3) \wedge \text{SET}_3\ (s_3))$ $\rightarrow \text{VUL}_1\ (s_0,\ s_2))$.

4) 竞争条件缺陷描述规则. 竞争条件缺陷描述规则用 RaceConditonRule 表示, RaceConditonRule = (Ω, n_0, SyntaxRule, JudgeRule), Ω = { SDG, DEF, USE, ResourceAccessFuctionList, ResourceCheck-FuctionList, process, Pred, Succ, PD, CD, DD }, n_0 = { n | CallName $(n) \in$ ResourceCheckFuctionList }, SyntaxRule = { $\mathcal{L}$ (pp), PP }, PP = $\{\text{pp}_0\}$, atti(PP) = $\{0\}$, JudgeRule = { judge (pp_i) | $\text{pp}_i \in$ PP $\wedge$ PP $\subset \mathcal{L}$ (pp) }.

• pp : $n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{\emptyset} \varepsilon$, 其中, (a): IsBelong $(n_1,$ USE (USE $(n_0))) \wedge$ IsBelong (CallName (n_1), ResourceAccessFunctionList) $\wedge$ CD (n_1, n_0), (b): Pred $(n_2, n_1) \wedge$ PD $(n_0, n_2) \wedge$ IsNull (n_2);

• $\text{pp}_0 : n_0 \xrightarrow{(a)} n_1 \xrightarrow{(b)} n_2 \xrightarrow{\emptyset} \varepsilon$, 其中, (a): IsBelong $(n_1,$ USE (USE $(n_0))) \wedge$ IsBelong (CallName (n_1), ResourceAccessFunctionList) $\wedge$ CD (n_1, n_0), (b): Pred $(n_2, n_1) \wedge$ PD $(n_0, n_2) \wedge$ IsNull (n_2);

• judge $(\text{pp}_0) : \forall s_0 \forall s_1\, (\text{SET}_0\, (s_0) \wedge \text{SET}_1\, (s_1) \wedge \text{FA}\, (s_0, s_1) \wedge \neg \exists s_2\, (\text{FA}\, (s_1, s_2)) \rightarrow \text{VUL}_1\, (s_0, s_1))$.

5) 格式串溢出缺陷、不安全系统调用、整数溢出、内存释放后使用、数组越界缺陷描述规则 (略).

2.4 算法与数据结构设计

在按照制导条件进行计算的过程中涉及多种数据, 考虑到对数据的访问较为频繁, 对访问性能有较高要求, 因此, 主要数据结构设计如下.

1) 初始节点的计算. 每个缺陷描述规则中模式路径公式的初始节点需要预先生成, 本模型采用数组形式存储初始节点, 结构体子项包括前驱节点、后继节点以及控制依赖节点和数据依赖节点.

2) 依赖关系存储和访问. 程序依赖图的节点间的数据依赖关系、控制依赖关系、直接前驱/后继关系的边构成的邻接矩阵是一个稀疏矩阵, 通过链表形式存储, 依赖关系采用数据依赖和控制依赖复合存储, 利用 4 位二进制数进行标识, 低 2 位表示数据依赖关系, 高 2 位表示控制依赖关系, 访问时通过与操作计算 2 节点间的依赖关系, 如图 8 所示. 其中, CD $(n_0, n_1) = r\ (n_0, n_1)$ & 0x0001; DD $(n_0, n_1) = r\ (n_0, n_1)$ & 0x0100; CD $(n_1, n_0) = r\ (n_0, n_1)$ & 0x0010; DD $(n_1, n_0) = r\ (n_0, n_1)$ & 0x1000.

3) 核查、支配等关系的计算. 在确定某条语句的终止语句后, 利用控制依赖、数据依赖、前驱、后继关系访问, 在线性时间内即可确定支配该语句的语句集合. 核查语句可以通过简单的词法分析判定.

4) 检测算法. 针对不同的缺陷类型, 模式路径的存在性检测算法具有相似性, 这里以空指针引用缺陷为例, 检测算法描述如算法 3, 其中函数 PPGEN 的伪码描述如算法 4.

5) 模型缺陷检测的时间复杂度分析. PPdetector 模型与全局状态的模型检验方法比较, 通过对相关数据的预存储, 以空间复杂度为代价大大缩减了被测程序节点遍历的规模, 根据定义 8 和推论 2 可知, 路径模式公式的求解规模能直接反映检测的时间复杂度, 通过分析, 可以确定影响模型检测过程中链表查找次数和比较次数的参数, 主要包括:

• 缺陷描述规则的初始节点集合中数量上限 n;

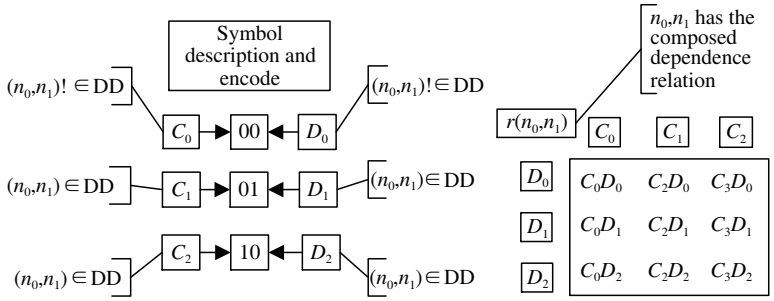

图 8 数据依赖关系和控制依赖关系二进制描述

Figure 8 Binary description of DDG and CDG

算法 3 空指针引用缺陷检测算法

输入: 空指针引用缺陷描述规则 NullPointerReferenceRule (如 2.3 小节的形式化描述)

输出: 空指针引用的缺陷语句集合 NPRVUL

```
NPRVUL ← {∅}
for each v ∈ possibleNPV do
    for each n0 ∈ DEF(v) do
        for each pp_i ∈ PP do
            PPinstance ← PPGEN (Ω, pp_i)
        end for
        for each instance ∈ PPinstance do
            judge(instance)
            NPRVUL ← NPRVUL ∪ {VUL1 or VUL2}
        end for
    end for
end for
return NPRVUL
```

算法 4 PPGEN

输入: 测试程序代码空间 Ω, 第 i 条模式路径公式 pp_i

输出: 程序代码空间 Ω 中存在的模式路径 pp_i 实例集合 PPinstance

```
PPinstance ← { ∅ }
if PPI (SDG, PP_i) == 1 then
    PPinstance ← travel (SDG, PP_i)
end if
return PPinstance
```

- 待检测模式路径中最长路径包含节点数量 m;
- 相关节点的依赖链表长度上限 λ_1;
- 构成制导逻辑表达式的谓词函数数量上限 λ_2;
- 所有缺陷描述规则对应的模式路径数量上限 λ_3.

表 2　缺陷类别和符号映射关系

Table 2　Vulnerability type and symbol mapping relationship

Vulnerability type of testing software	Symbol
Operating error after releasing memory	VA
Pointer misuse	VB
Memory leak	VC
Overflow of format string	VD
Calling insecure system	VE
Race condition	VF
Integer overflow	VG
Array overflow	VH
Improper protecting resource	VI

6) 链表查找次数计算. 一条模式路径求解过程中, 链表查找次数上限为 $(\lambda_1\lambda_2)^m$, 待检测模式路径数量上限为 $\lambda_3 n$, 链表查找次数上限为 $o\left((\lambda_1\lambda_2)^m \lambda_3 n\right)$; 节点状态比较次数和路径模式的求解规模相关, 小于链表查找次数, 因而节点状态比较次数上限为 $o\left((\lambda_1\lambda_2)^m \lambda_3 n\right)$. 通过对多个测试程序的观察以及模式路径公式的定义, λ_1, λ_2, λ_3, m 都是较小的常数, 相关节点的依赖链表长度上限 λ_1、初始节点集合中数量上限 n 和测试程序代码行数呈线性关系, 构成制导逻辑表达式的谓词函数数量上限 $\lambda_2 \leqslant 6$, 所有模式路径数量上限 $\lambda_3 \leqslant 10$, 最长模式路径包含节点数量 $m \leqslant 6$.

3　测试与分析

本文 PPDdetector 的实验平台配置为处理器 Intel(R) Core(TM) i72600, 3.4 GHz, 内存 DDR3 8 GB, 操作系统 Ubuntu 32 bits.

PPDdetector 对多个不同代码规模的开源程序进行测试, 表 2 给出了 PPDdetector 中模式路径公式下定义的 9 类缺陷描述规则的符号映射关系, 表 3 给出了 iodine[12)], bftpd[13)], wu-ftpd[14)], openssl[15)], dnsmasq[16)], net-snmp[17)], openssh[18)], cups[19)], busybox[20)], coreutil[21)], andorid-kernel[22)], ntp[23)], samba[24)] 等 13 个不同代码规模的网络服务、系统服务等安全相关度较高的测试软件信息, 包括软件版本、软件代码行数、分析文件数量、分析函数数量、系统依赖图生成路径数量、模式路径分析总数量以及 PPDdetector 原型系统的检测时间.

12) http://code.kryo.se/iodine/.
13) http://freecode.com/projects/bftpd/.
14) http://wu-ftpd.therockgarden.ca/.
15) http://www.openssl.org/.
16) http://www.thekelleys.org.uk/dnsmasq/doc.html.
17) http://www.net-snmp.org/.
18) http://www.openssh.org/.
19) http://www.cups.org/.
20) http://www.busybox.net/.
21) http://www.gnu.org/software/coreutils/.
22) http://source.android.com/source/building-kernels.html.
23) http://www.ntp.org/.
24) http://www.samba.org/.

表 3 测试软件相关代码信息和测试时间

Table 3 The code information and testing time of 13 softwares

Testing software	Lines of code	Testing time (s)	Version	File account	Function account	SDG path account	PP
iodine	18007	28	0.4.2	26	83	10052	1202
bftpd	19853	43	1.6	41	144	14724	984
wu-ftpd	38674	267	2.6.2	59	271	59841	3864
openssl	241310	2691	1.0d	799	6137	939937	25481
dnsmasq	29000	321	2.47	43	216	86944	4222
net-snmp	121992	1126	5.0.11.2	520	1789	423188	22490
openssh	106140	645	5.8p1	497	1605	246064	9812
cups	177445	1331	1.4.8	106	769	643654	32101
busybox	79630	579	1.11.2	347	1363	182089	52327
coreutil	100361	777	6.11	217	1445	309832	15666
android kernel (partial module)	151338	807	2.6.39.2	345	1859	212191	63897
ntp	138169	649	4.2.2	69	759	185001	50043
samba	662130	7850	3.6.2	1288	19265	3858984	100023

表 4 PPDdetector 对 13 个软件的可疑缺陷报告结果

Table 4 Doubtful vulnerability report of 13 softwares by PPDdetector

Target software	VA	VB	VC	VD	VE	VF	VG	VH	VI
iodine	0	1	0	0	4	0	0	1	0
bftpd	0	7	0	2	23	0	0	1	0
wu-ftpd	0	2	7	1	63	7	0	1	0
openssl	0	18	1	1	31	2	0	2	0
dnsmasq	0	4	13	0	31	0	0	0	0
net-snmp	7	30	59	12	153	6	1	10	0
openssh	4	6	6	0	9	9	0	1	0
cups	1	3	7	7	90	9	1	0	0
busybox	0	4	25	3	66	22	3	8	0
coreutil	1	9	2	2	54	9	1	7	0
android kernel (partial module)	2	3	3	6	58	0	0	3	0
ntp	0	1	0	2	57	0	0	0	0
samba	2	42	9	3	33	2	4	14	0

表 3 中各个软件的代码安全性缺陷报告数量情况如表 4 所示, 经初步人工审查, 验证了 10 个有 CWE[25)]编号的已公开漏洞, 表 5 给出了这些漏洞的信息, 包括 iodine 中 1 个漏洞、wu-ftpd 中 2 个漏洞、dnsmasq 中 2 个漏洞、net-snmp 中 1 个漏洞、cups 中 1 个漏洞、android-kernel 中 1 个漏洞、ntp 中 1 个漏洞、samba 中 1 个漏洞, 这 10 个漏洞基本覆盖了 PPDdetector 中利用模式路径公式定义的

25) http://cwe.mitre.org/documents/vuln-trends/index.html.

表 5　PPDdetector 对 13 个软件的缺陷报告结果

Table 5　Vulnerability report of 13 softwares by PPDdetector

Target software	Detected open vulnerability numbers	Vulnerability type	ID
iodine	1	VB	Bugtraq-34731
bftpd	0	–	–
wu-ftpd	2	VH	CVE-2003-1327
		VE	CVE-2003-0466
openssl	0	–	–
dnsmasq	2	VB	CVE-2009-2958
		VE	CVE-2009-2957
net-snmp	1	VE	CVE-2008-6123
openssh	0	–	–
cups	1	VA	CVE-2010-0302
busybox	0	–	–
coreutil	0	–	–
android kernel (partial module)	1	VB	CVE-2009-2692
ntp	1	VD	CVE-2009-1252
samba	1	VC	CVE-2012-0817

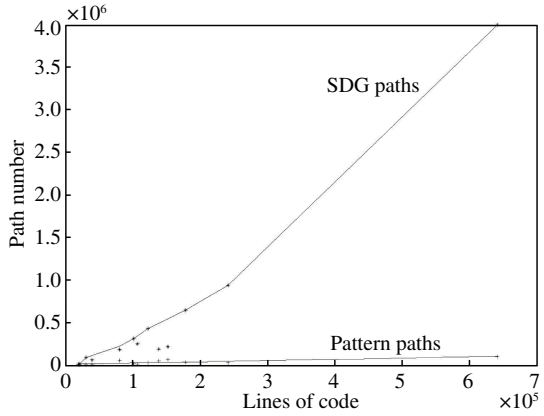

图 9　测试代码行数和模式路径数量、SDG 路径数量的关系图

Figure 9　Relationship between lines, PP number and SDG path number of testing code

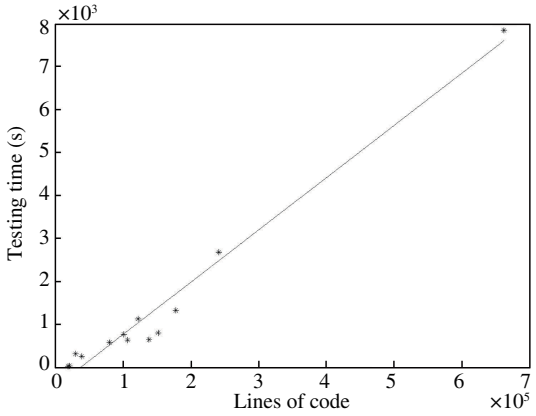

图 10　测试代码行数和测试时间的关系图

Figure 10　Relationship between lines and time of testing code

缺陷类别, 分别是 1 个内存释放后操作错误、3 个指针误用缺陷、1 个资源泄露、1 个格式串溢出缺陷、3 个不安全系统调用缺陷和 1 个数组越界缺陷, 结合各类缺陷报告数量可以发现, 对指针类错误所引发缺陷的检测准确率最高, 不安全系统调用缺陷准确率最低, 这和缺陷条件信息的健全度相关, 可以作为今后优化目标的参考数据之一.

图 9 和图 10 分别给出了 PPDdetector 的性能测试数据. 图 9 给出了被测代码行数和模式路径数量、SDG 路径数量的关系图, 这里的 SDG 路径指的是和模式路径相关的语句节点构成的 ICFG 上统

计出的路径数量, 从图上看出 SDG 路径数量和代码行数成非线性关系, 而转换为模式路径描述时, 压缩为低常量的线性关系, 这说明了模式路径公式在静态分析中发挥了显著缩减测试状态的作用. 图 10 给出了被测代码行数和测试时间的关系图, 图中二者基本呈线性增长关系, 总的来说, PPDdetector 在一定程度上缓解了时间复杂度的爆炸性增长问题. 最后, 在其它缺陷报告的审查过程中, 发现部分未公开的代码安全缺陷, 其中 2 个已通过了数据包可达性验证.

4 小结

本文提出并实现了一种基于命题逻辑和谓词逻辑的形式化检测方法－模式路径制导方法, 具备对内存释放后操作错误、指针误用、资源泄露、格式串溢出、不安全系统调用、竞争条件、整数溢出、资源保护不当和数组越界等软件代码安全性缺陷的检测能力, 并有效缓解了已有检测方法的时间复杂度爆炸性增长问题. 下一步工作中, 一方面可通过对更多代码安全性缺陷语义特征的研究, 进一步扩展本方法的缺陷类型覆盖能力; 另一方面, 可改造后作为模型检验等已有方法的前端模块, 实现检测能力和检测效率的双重提升.

参考文献

1 Krsul I V. Software Vulnerability Analysis. Dissertation for Ph.D. Degree. West Lafayette: Purdue University, 1998

2 Hui Z, Huang S, Ren Z, et al. Review of software security defects taxonomy. In: Yu J, Greco S, Lingras P, et al, eds. Rough Set and Knowledge Technology. Berlin: Springer, 2010. 310–321

3 Chess B, Wysopal C. Guest editors' introduction: software assurance for the masses. Secur Privacy, 2012, 10: 14–15

4 Di Penta M, Cerulo L, Aversano L. The evolution and decay of statically detected source code vulnerabilities. In: Proceedings of 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. Piscataway: IEEE, 2008. 101–110

5 Viega J, Bloch J T, Kohno Y, et al. ITS4: a static vulnerability scanner for C and C++ code. In: Proceedings of 16th Annual Conference on Computer Security Applications. Piscataway: IEEE, 2000. 257–267

6 Midtgaard J. Control-flow analysis of functional programs. ACM Comput Surv, 2012, 44: 10

7 Fosdick L D, Osterweil L J. Data flow analysis in software reliability. ACM Comput Surv, 1976, 8: 305–330

8 Foster J S, Fghndrich M, Aiken A. A theory of type qualifiers. ACM Sigplan Not, 1999, 34: 192–203

9 Darvas A, H. R, Sands D. A theorem proving approach to analysis of secure information flow. In: Proceedings of the Second International Conference on Security in Pervasive Computing. New York: ACM, 2003. 193–209

10 Clarke E, Grumberg O, Peled D. Model Checking. Cambridge: MIT Press, 1999

11 Ball T, Naik M, Rajamani S K. From symptom to cause: localizing errors in counterexample traces. ACM Sigplan Not, 2003, 38: 97–105

12 Ball T, Rajamani S K. The SLAM project: debugging system software via static analysis. ACM Sigplan Not, 2002, 37: 1–3

13 Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York: ACM, 1977. 238–252

14 Jhala R, Majumdar R. Software model checking. ACM Comput Surv, 2009, 41: 21

15 Cousot P, Cousot R. Temporal abstract interpretation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York: ACM, 2000. 12–25

16 Cadar C, Godefroid P, Khurshid S, et al. Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd International Conference on Software Engineering. New York: ACM, 2011. 1066–1071

17 Xie Y, Aiken A. Saturn: a scalable framework for error detection using boolean satisfiability. ACM Trans Program Lang Syst, 2007, 29: 16
18 Cousot P. Proving the absence of run-time errors in safety-critical avionics code. In: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software. New York: ACM, 2007. 7–9
19 Blanchet B, Cousot P, Cousot R, et al. A static analyzer for large safety-critical software. ACM Sigplan Not. New York: ACM, 2003, 38: 196–207
20 ROSE Team. Compass User Manual: A Tool for Source Code Checking Version 0.9.5a, 2013
21 Bessey A, Block K, Chelf B, et al. A few billion lines of code later: using static analysis to find bugs in the real world. Commun ACM, 2010, 53: 66–75
22 Emanuelsson P, Nilsson U. A comparative study of industrial static analysis tools. Electron Notes Theor Comput Sci, 2008, 217: 5–21
23 Tripathi A, Singh U K. Towards standardization of vulnerability taxonomy. In: Proceedings of 2010 2nd International Conference on Computer Technology and Development. Piscataway: IEEE, 2010. 379–384
24 Horwitz S B. Interprocedural slicing using dependence graphs. ACM Trans Program Lang Syst, 1990, 12: 26–60
25 Hackett B, Aiken A. How is aliasing used in systems software? In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2006. 69–80
26 Ganesh V, Dill D L. A decision procedure for bit-vectors and arrays. In: Damm W, Hermanns H, eds. Computer Aided Verification. Berlin: Springer, 2007. 519–531
27 Aiken A, Bugrara S, Dillig I, et al. An overview of the Saturn project. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York: ACM, 2007. 43–48
28 Howard M, LeBlanc D, Viega J. 19 Deadly Sins of Software Security Programming Flaws and How to Fix Them. Xiao F T, Yang M J, Trans. Beijing: Tsinghua University Press, 2006 [Howard M, LeBlanc D, Viega J. 软件的 19 个致命安全漏洞. 肖枫涛, 杨明军, 译. 北京: 清华大学出版社, 2006]
29 Weiser M D. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Dissertation for Ph.D. Degree. Michigan: University of Michigan Ann Arbor, 1979
30 Matsubara M, Narisawa R, Enshoiwa M, et al. Model checking with program slicing based on variable dependence. In: Proceedings of FTSCS 2012, First International Workshop on Formal Techniques for Safety-Critical Systems, Kyoto, 2012

A static detecting technology of software code secure vulnerability based on first-order logic

QIN XiaoJun*, GAN ShuiTao & CHEN ZuoNing

Jiangnan Computer Technique Institute, Wuxi 214083*, China*
*E-mail: xjqin@163.com

Abstract The secure vulnerability of software codes is an important vulnerability which may cause a disaster in the software system. The automatic detecting and locating technologies for this type of vulnerability have a significant meaning in the software preserving and evolution. This paper proposes and implements a formal detection method which is a static detecting method of software code secure vulnerability based on first-order logic. Our method defines the formula of pattern path by combining propositional logic and predicate logic. Some expressions of proposition logical construction function related to dependence relation is used as directed conditions for creating the nodes of pattern path. Then we formulate various types of software code secure vulnerabilities and turn the efforts in finding vulnerability to judging the existence of pattern path in limited state space among corresponding intermediate codes. The experiment results show that our method is fit for

detecting most types of software code secure vulnerability. It punctually finds out ten known and two 0-day vulnerabilities in 13 open source projects including “openssl”, “wu-ftpd”, etc. Comparing to existing static analysis methods, such as module checking, the test time in suing this model is almost in line with the size of the code.

Keywords software code secure vulnerability, first-order logic, pattern path, static analysis, formula description

QIN XiaoJun was born in 1975. He received the M.S. degree from Jiangnan Computer Technique Institute, Wuxi, in 2004. Currently, he is a Ph.D. candidate in Jiangnan Computer Technique Institute. He is a Senior Researcher at Jiangnan Computer Technique Institute. His main research interest includes software assurance, and computer network.

GAN ShuiTao was born in 1986. He received the M.S. degree from Jiangnan Computer Technique Institute, Wuxi, in 2010. Currently, he is a Ph.D. candidate in Jiangnan Computer Technique Institute. His main research interest includes software assurance, and computer network.

CHEN ZuoNing was born in 1957. Currently, she is a senior engineer in Jiangnan Computer Technique Institute. Her main research interest includes operating system, and software theory. She is an Academician of the Chinese Academy Engineering.