基于模式匹配的安全漏洞检测方法

缪旭东[1] 王永春[1] 曹星辰[2] 方 峰[2]
(海军大连舰艇学院作战软件与仿真研究所 大连 116018)[1]
(华中科技大学计算机科学与技术学院 武汉 430074)[2]

摘 要 针对现存的大部分软件漏洞静态检测工具无法灵活检测用户关心的漏洞的情况,提出了一种基于模式匹配的漏洞检测方法。首先,对待测程序源码进行解析,将其转化为中间表示并存放在自定义的数据结构中;然后,用安全规则语言描述漏洞并解析安全规则,将其转换成对应的自动机模型存放在内存中;最后,将源代码的中间表示与安全规则进行模式匹配,并跟踪自动机的状态转化,根据自动机状态向用户提交漏洞报告。实验结果表明,该方法的漏报率低、扩展性好。

关键词 安全规则,模式匹配,漏洞检测,静态分析
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.04.024

Detection Approach for Security Vulnerability Based on Pattern Matching

MIAO Xu-dong[1] WANG Yong-chun[1] CAO Xing-chen[2] FANG Feng[2]
(Operation Software and Simulation Research Institute, Dalian Naval Academy, Dalian 116018, China)[1]
(School of Computer Science & Technology, Huazhong University of Science and Technology, Wuhan 430074, China)[2]

Abstract For the conditions that most of the existing software vulnerability static detecting tools cannot detect vulnerabilities that users care, this paper proposed a vulnerability detection method based on pattern matching. First, the source code which is going to be tested is parsed, and the code is transformed into intermediate representation which is stored in the user-defined data structure. Then, the vulnerability is described and the safety rules is parsed by using safety rule languages, and they are converted into corresponding automata model which can be stored in memory. Finally, the source code intermediate representation and safety rule should be for pattern matching, and the automata state should be transformed. And we need to submit the report based on the automata state to users. The experimental results show that this method has a low missing report rate and good expansibility.

Keywords Safety regulations, Pattern matching, Vulnerability detection, Static analysis

1 引言

随着社会的信息化发展,计算机软件被越来越多地应用于各个行业。在软件开发中不可避免地存在设计缺陷或者编码错误,这都是产生漏洞的原因。一旦攻击者发现并且利用这些漏洞,就会对系统的完整性、可用性、保密性[1]造成影响。随着 Internet 的普及,攻击者利用软件漏洞的手段更加便捷,软件安全问题[2]变得越来越突出。

目前,软件安全性分析方法主要分为静态分析技术和动态分析技术两大类。动态分析[3]是一个将测试用例作为输入,然后运行可执行文件,通过程序运行的输出发现程序缺陷的过程;静态分析[4]则无需运行被测程序,直接通过分析程序源代码发现程序的缺陷。据相关报道,大部分软件安全性问题是在软件开发中产生的。因此,如何在软件开发的过程中发现这些缺陷或者错误具有重要的研究价值。在这一点上,静态分析技术拥有动态分析无可比拟的优势。

本文提出了一种针对源代码进行检测的静态分析技术。借鉴 Metal[5-6]定义了一套安全规则描述语言,采用模式匹配的方法,根据使用者自定义的规则对特定软件漏洞进行检测。

2 相关工作

目前静态分析方法研究的热点之一是规则检查分析。对于特定漏洞,可以总结出一套相对统一的源代码,即漏洞模式。规则检查方法将这些漏洞模式以特定语法进行描述,经过一系列处理后再将程序行为进行比对、检测。目前采用规则检查分析的系统主要有以下几种。

PMD[7]是一款采用 BSD 协议发布的 Java 程序代码检查工具,其提供了通过查看被测代码的抽象语法树来编写检测

到稿日期:2015-11-30 返修日期:2016-02-29 本文受湖北省自然科学基金(2014CFB1006),华中科技大学自主创新基金(2015QN062)资助。
缪旭东(1964—),男,博士,教授,主要研究领域为军用软件质量管理,E-mail:miao3399@sina.com;王永春(1977—),男,博士,助理研究员,主要研究领域为军用软件需求分析与模型构建;曹星辰(1988—),男,硕士,主要研究领域为软件测试;方 峰(1991—),男,硕士生,主要研究领域为软件测试。

规则或者使用 Java 代码重新定义检测模块的功能。虽然 PMD 支持自定义规则，但是由于高级功能需要查看源代码抽象语法树，因此存在规则编写难度大、对使用者要求高等缺点。

Coverity Prevent[8]是由 Coverity 公司开发的一款检测软件缺陷和安全隐患的源代码静态分析工具，来自于斯坦福大学的 Metal 研究项目。Coverity Prevent 利用 Metal 语言来自定义相关的安全规则，并采用跨过程的数据流分析和静态分析相结合的方法，根据自定义的安全规则检测漏洞，形成分析报告。但作为一款商用软件，用 Coverity Prevent 来扫描机密性较高的系统存在一定风险。

Metal 在本质上是一种自动机形式的描述语言，用户可以根据实际需要编写符合 Metal 语法的规则。但是 Metal 采用类 C 语言文法来描述，用户可使用的类型和表达式有限，使得用户在根据 Metal 描述漏洞特征时存在一定局限性。本文所使用的安全规则描述语言是在借鉴 Metal 设计思路的基础上，针对其不足做出改进，并增加模式约束等设计而成。本项目定义的安全规则描述语言能够检测源代码，检测效率得到了提升；支持自定义规则，规则编写难度降低，而扩展性得到提高。

虽然国内外研究者在静态分析技术上取得了一定的科研成果，但是还存在不完善的地方：在静态分析过程中，对漏洞产生的场景考虑不够全面，如大多数漏洞的攻击路径都是跨函数的，而有些分析技术只是考虑单一函数的情景；对源码中常见的分支语句的支持不够，导致一些代码不能被识别；对程序中函数的递归调用不能很好地处理从而直接导致检测失败。上述考虑不足之处都会造成检测结果的漏报率较高。因此，本文基于模式匹配的安全漏洞检测方法将从上述 3 个方面进行研究，这对于静态分析检测漏洞具有重要的研究意义和实用价值。

3 模式匹配研究

模式匹配的基本流程如下。首先，系统会读取安全规则源文件，并将其解析成自动机模型。然后，扫描源代码中间表示 IR (Intermediate Representation, IR)，与安全规则中的源码模式进行匹配，匹配成功之后再进行模式约束检测。在模式匹配与模式约束都成功的前提下，绑定触发模式匹配的变量到自动机，并转移自动机的状态。如果自动机到达不安全状态，就向用户提交漏洞报告；如果没有到达不安全状态，就在程序流程图的控制下继续扫描后续的源代码 IR，直到 IR 扫描结束。

3.1 模式匹配流程

安全规则经过规则解析器解析之后会产生一个自动机模型，每当发生 start 状态转换时，就会生成该模型对应的自动机。模式匹配的流程如图 1 所示。

1)程序入口点：扫描是从程序入口点开始的，程序入口点指的是程序运行最开始执行的那一个函数，最常见的入口点就是 main 函数。在 Java 中程序入口点通常不止一个，因此，安全规则扫描器会从每一个程序入口点扫描程序。

2)函数扫描：主要任务是在对该函数的函数体进行路径扫描之前进行信息收集等工作，并处理扫描过程中遇到的循环调用等问题。

3)路径扫描：包含两部分任务，首先，根据语句的类型调用不同的方法对其进行处理；在扫描完一条语句之后，根据源代码的控制流程继续扫描下一条语句。

4)函数调用语句、分支语句、顺序执行语句：路径扫描会对当前语句类型做出判断。如果是函数调用语句类型，则采用 DFS 方式进行函数扫描；如果是分支语句类型，则开辟新的路径进行路径扫描；如果是顺序执行语句类型，则进行模式匹配及模式约束。

5)转移状态、绑定变量：在匹配成功时转移自动机的状态，并将触发状态转移的变量绑定到自动机。

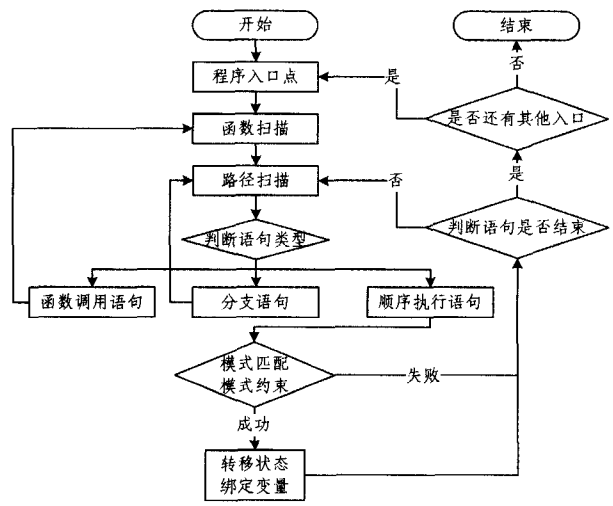

图 1 模式匹配流程图

安全规则可以给出模式匹配过程中的匹配语句。但是在实际程序中，由于分支语句、函数调用等语句的存在，从漏洞注入点到攻击点可能存在多条路径，安全规则必须指出哪条路径会引发漏洞，以方便系统管理员修补漏洞，因此，需要系统给出关键路径。关键路径是由能够唯一确定从漏洞注入点到攻击点路径的一系列语句构成的。关键路径应该包括两部分：1)在模式匹配过程中匹配成功的源代码语句；2)注入点到攻击点路径上的分支语句和函数调用语句。其中前者在安全规则匹配过程中就能够获得，而后者的获得则需要综合考虑分支路径扫描和函数扫描。

在模式匹配的流程中，有跨函数匹配、分支路径匹配、环路识别 3 个主要的问题需要解决。

3.2 跨函数匹配

大多数漏洞的攻击路径都是跨函数的。因此，除了在模式匹配成功时绑定触发状态转移的变量，还需要考虑函数调用中实参与形参的数据传递。每进入一层函数扫描，都将该函数的实参分为两部分：一部分与当前自动机中已经绑定的变量有数据依赖关系，另一部分没有数据依赖关系。后者在扫描过程中不予关注，重点是对前者的处理。将这些具有数据依赖关系的变量所对应的函数形参全部绑定到自动机中，之后继续按照图 1 所示的方式扫描源代码。当扫描退出该函数调用时，检查自动机的状态是否发生了改变，若没有发生改变则将函数的形参从自动机中删除，否则依然保留形参与自

动机的绑定关系。

在进入函数扫描之前，应当对函数语句进行判断，看其是否为系统调用(对没有源代码的系统函数或者第三方函数不予检测)。如果不是，则将函数调用语句加入到自动机的扫描路径中。扫描路径是指因关键路径生成需要，在模式匹配过程中记录一个分支语句和函数调用语句的集合，从本质上来说，就是一个关键路径的候选语句集合。从该函数扫描退出时，查看自动机扫描路径中的记录，如果没有新添加的路径或者自动机的状态没有发生改变，则将该函数调用语句从扫描路径中删除；否则，该调用语句是关键路径的一部分，予以保留。

3.3　分支路径匹配

下面以 if 语句为例进行分析，与跨函数匹配类似，在进入 if 分支扫描之前，也将分支语句加入到自动机的扫描路径中。对进入分支扫描后，开辟两条新的路径进行扫描(true 分支和 false 分支)，此时需要将自动机拷贝一份，以便区分两条路径分别对自动机状态产生的影响。除此之外，还应重新标明 end 语句，该语句用于告知路径扫描何时终止，避免重复地对同一条语句进行扫描，提高程序执行效率。在进入新的路径扫描之前会将该路径的首条语句加入到自动机的扫描路径中。对两条新开辟的路径都扫描完成之后，对比两个自动机的状态，如若两个自动机的状态都没有发生改变，则说明 if 分支中的语句不会对漏洞的产生造成任何影响。因此，应将之前添加的 if 语句以及新开辟路径的首条语句从扫描路径中删除。否则，说明该分支语句是关键路径的一部分，应当保留扫描路径现有的状态。详见算法 1。

算法 1　分支扫描

```
输入：源代码 IR
输出：关键语句集合
1. begin：
2.   for ∀branchStmt in IR
3.     automation←branchStmt
4.     for ∀branch in branchList
5.       copyAuto←automation
6.       bind firstStmt to copyAuto
7.       endStmt←getEndStmt()
8.       pathScanner(copyAuto)
9.       if copyAuto same as automation
10.          remove IfStmt and firstStmt
11.      end if
12.      automation←copyAuto∪automation
13.  end for
14. end
```

算法分析：算法 1 主要包含两层循环，第一层循环控制系统只对源代码 IR 中的分支语句进行扫描，第二层循环对每个分支调用一次路径扫描，pathScanner 即为路径扫描函数，分支语句有多少分支就调用多少次 pathScanner。假设源代码 IR 中分支语句的数量为 n 个，则此种情况的时间复杂度为 $O(n)$。综合以上分析，分支路径匹配的流程如图 2 所示。

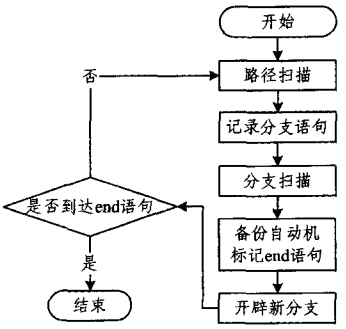

图 2　分支路径匹配

3.4　环路识别

静态分析中经常需要解决的一个问题是对循环的处理，例如，函数的递归调用，直接或者间接地调用函数自身的函数被称为递归函数。在递归函数中，一定有一个分支没有调用自身，在程序运行的过程中，很容易判断是否应该退出递归调用。但是从静态分析的角度来看，无法判断何时应该退出递归调用。通行的做法是对循环调用展开两层，这主要是因为展开两层可以覆盖大部分的调用序列。本文借鉴了这一思想，结合源代码 IR 的生成过程以及安全规则的扫描特点，给出以下解决方案。

假设函数 A 调用函数 B，函数 B 调用函数 C，函数 C 又调用函数 A，如果在静态分析中只考虑调用关系 $A->B->C$，肯定会造成路径丢失。因为循环调用意味着从任意一条语句开始按照程序的控制流程扫描都能回到原点。只考虑上述调用关系，必然导致扫描至函数 C 时无法再回到函数 A，因为自动机的关键路径中没有记录 $C->A$ 的调用关系。若考虑调用关系 $A->B->C->A->B->C$，则不会造成路径丢失，但是存在重复记录调用关系的现象，从而产生额外的时间和空间开销。在调用关系 $A->B->C->A$ 中，存在函数 C 到函数 A 的调用关系，不会造成路径丢失，也没有额外的时间和空间开销。因此，对循环调用只需要展开到调用起点即可。详见算法 2。

算法 2　循环调用

```
输入：调用路径 callPath，起始值为空链表
输出：关键语句集合
1. begin：
2.   if !method∈callPath && isn't systemCall
3.     add method to callPath
4.     pathScanner(callPath)
5.     if !method∈automation.getScanPath()
6.       bind method to automation
7.     end if
8.   end if
9.   remove method from callPath
10. end
```

算法复杂度分析：算法 2 主要有两个分支，第一个分支用于识别循环调用，并排除系统调用的情况；第二个分支判断当前函数调用语句是否已经存在于自动机所对应的扫描路径中。该算法的实现流程在图 1 中已经详细地给出，pathScanner 调用各个分支语句扫描，最终又会调用算法 2 本身。从整

体角度来看，算法的主要开销依然在于源代码 IR 的扫描上，因此，时间复杂度为 $O(n)$。

本项目在分支路径扫描中只考虑与函数扫描相互调用的函数调用扫描，排除系统调用，不将其加入到扫描路径当中。对于非系统调用，在进行 DFS 递归扫描之前，需要判断是否已经构成了环路，一旦构成了上文中分析的情况，就停止 DFS 扫描分析。将函数调用加入扫描路径时还应该附加一个标记，以区分扫描路径中的语句与模式匹配中匹配的语句。循环调用匹配的详细流程如图 3 所示。

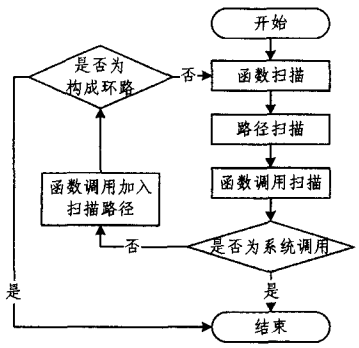

图 3 循环调用匹配

4 系统框架

为了验证本文提出的基于模式匹配的安全漏洞检测方法的有效性与可用性，设计并实现了 SDTPM(Security Vulnerability Detection Tool based on Pattern Matching)系统。SDTPM 对漏洞的检测是建立在源代码静态分析基础之上的，静态分析引擎运用 ANTLR 编译技术对源代码进行信息收集并将信息存放在中间表示 IR 中。安全规则是一种支持自定义的漏洞检测描述语言，规则解析器可以将其解析为自动机模型，然后与源代码的中间表示进行模式匹配，最后根据匹配结果提交漏洞报告。SDTPM 的系统框架如图 4 所示。

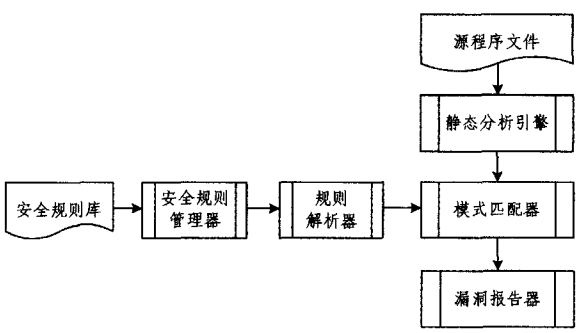

图 4 SDTPM 的系统框架

静态分析引擎：利用开源编译工具 ANTLR 对源代码进行信息收集，并将收集到的源代码信息存放到 IR 中。在 IR 的生成过程中就已经产生了控制流信息，生成了控制流程图 CFG。在 CFG 的基础上再进行控制依赖分析和数据依赖分析就能够得到控制依赖图 CDG 和数据依赖图 DDG。

安全规则库：主要用来存放用户自定义的安全规则，提供字段来记录每条规则的名字、描述、存放路径、适用语言和推荐修复方案等信息。

安全规则库管理器：主要通过提供界面来完成安全规则的增删改查，系统支持安全规则的分类查询和模糊查询。

规则解析器：与静态分析引擎类似，使用 ANTLR 对安全规则进行解析，并将解析获得的信息存放在自动机模型中。

模式匹配器：将规则解析器解析获得的自动机模型和静态引擎分析得到的源代码 IR 进行比对，当匹配成功时，根据相应的规则转换自动机的状态，如果自动机达到不安全状态，就向系统提交漏洞报告。模式匹配分为两大结构：与源代码模式的比对和模式约束。模式约束是在模式比成功之后进行的，它增强了模式匹配的描述能力，也使得模式匹配结果更加精确。

漏洞报告器：其主要功能是将基于安全规则的静态分析器检测出来的漏洞数据进行整理和重组，然后按照标准的漏洞报告格式生成安全漏洞报告展现给用户。漏洞报告中除了包括基本的安全漏洞信息外，还会根据用户自定义的安全规则附上推荐的修复方案。

5 实验分析

5.1 实验环境

本实验在如下环境中进行：Intel Core(TM) i3-2100 CPU，内存 2.92GB，Windows 7 旗舰版(32bit)，Eclipse 4.2 SR2(Juno)，JDK 1.6_45，ANTLRWorks 1.4.2，antlr 4.3。

本实验采用的数据集是 2012 年 9 月发布的 Juliet Test Suite for Java 的 1.1.1 版本。Juliet Test Suite 是一个专门用于测试漏洞检测工具的数据集，该数据集分别对 113 种漏洞构建了测试用例。本文从中选取了最常见的若干种漏洞，抽象出这些漏洞的特征并用安全规则描述，再使用 SDTPM 系统检测这些漏洞。该数据集中共包含 341 个文件、353 个类、2278 个方法和 113513 行代码。

5.2 实验结果分析

分别使用 SDTPM 和 PMD 针对实验数据集中的 12 种安全漏洞编写相应的安全规则来检测数据集中源代码的潜在漏洞，主要记录了编写的规则行数、检测时间、内存使用情况以及漏洞检测数，得到了如表 1 所列的实验原始数据。然后，通过对实验数据进行统计和分类，得到如表 2 所列的实验统计数据，以便于实验对比分析。

表 1 中的规则行数是按照两个系统中的标准规则定义格式进行统计的，可以看出 SDTPM 在规则定义的简易程度上明显优于 PMD，且 PMD 在编写 XPath 规则时需要查看源代码的抽象语法树结构，易用性较差。需要特别说明的是，表 1 中针对最后 5 种安全漏洞 PMD 的实验数据均为 0，是因为 PMD 无法使用 XPath 规则来编写上下文相关的规则，只能通过编写 Java 源代码来添加规则类的方式实现，这已经超出了本实验的对比范围，所以本文未采集此部分实验数据。

表 2 的实验统计数据给出了 SDTPM 系统的检测情况，从表中数据可以看出，对于上下文无关规则，SDTPM 无误报和漏报情况，而 PMD 存在少量漏报的情况。而对于上下文相关规则，SDTPM 还是会存在少量的误报和漏报的情况，主要原因是源代码转成中间表示以后在少量不常用的数据结构中丢失了一些源代码信息，从而导致控制流信息不完全。

表 1　实验原始数据

漏洞名称	标准漏洞数	SDTPM			PMD		
		规则行数	检测时间/s	漏洞检测数	规则行数	检测时间/s	漏洞检测数
CWE78_OS_Command_Injection	95	6	0.028	95	23	1.080	95
CWE89_SQL_Injection	172	9	0.031	172	35	2.233	158
CWE209_Information_Leak_Error	36	7	0.020	36	20	0.802	36
CWE327_Use_Broken_Crypto	18	8	0.015	18	24	0.094	18
CWE330_Insufficiently_Random_Values	18	8	0.016	18	27	0.984	18
CWE547_Hardcoded_Security_Constants	18	9	0.013	18	23	0.916	18
CWE597_Wrong_Operator_String_Comparison	18	7	0.018	18	21	0.354	18
CWE614_Sensitive_Cookie_Without_Secure	18	14	0.036	17	0	0	0
CWE698_Redirect_Without_Exit	35	13	0.041	34	0	0	0
CWE764_Multiple_Locks	4	13	0.047	6	0	0	0
CWE765_Multiple_Unlocks	2	14	0.041	4	0	0	0
CWE832_Unlock_Not_Locked	4	12	0.034	8	0	0	0

表 2　实验统计数据

	标准漏洞数	漏洞检测数	误报数	漏报数	误报率/%	漏报率/%	准确率/%	检测种类数	内存使用情况/MB
SDTPM	438	444	8	2	1.8	0.5	98.2	12	48
PMD	375	361	0	14	0	3.7	100	7	15

注：误报率＝误报数/漏洞检测数；漏报率＝漏报数/标准漏洞数；准确率＝(漏洞检测数－误报数)/漏洞检测数

从表 1、表 2 中可以看出，SDTPM 系统在规则的简洁性、检测效率、检测结果的漏报率以及检测漏洞种类方面都优于 PMD，两者的准确率相差不大。SDTPM 系统采用基于模式匹配的安全漏洞检测方法，在检测过程中同时考虑到源码中可能存在的跨函数、分支路径和环路等漏洞产生的场景，使得检测结果的漏报率大大降低。但在内存使用情况方面，SDTPM 却大大超出了 PMD，这主要是因为 SDTPM 使用了源代码中间表示存储器，占用内存较大，且随着源代码规模的增加，存放源代码的中间表示所占用的内存空间也会越来越大。SDTPM 系统检测的漏报率和效率是以空间的消耗为代价的，这也是本系统的不足之处。

结束语　通过分析常用的静态分析方法，在吸取其优点的基础上，提出了一种基于模式匹配的安全漏洞检测方法，并实现了原型系统 SDTPM。实验结果表明，SDTPM 系统检测漏洞时具有漏报率低、扩展性好的特点。同时从实验结果来看，较大的空间消耗成为了影响检测程序源代码规模的一个制约条件。后续工作：1)研究针对大规模程序源代码的中间表示的存储方式来降低系统内存的消耗；2)对源代码的中间表示进行改进，去掉不必要的数据结构和源代码信息以减小内存消耗。

参 考 文 献

[1] JUENEMAN R R. Securing wireless medicine confidentiality, integrity, nonrepudiation, & malware prevention[C]//2011 8th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT). IEEE, 2011: 1-5.

[2] ALBREIKI H H, MAHMOUD Q H. Evaluation of static analysis tools for software security[C]// 2014 10th International Conference on Innovations in Information Technology (INNOVATIONS). IEEE, 2014: 93-98.

[3] EGELE M, SCHOLTE T, KIRDA E, et al. A survey on automated dynamic malware-analysis techniques and tools [J]. ACM Computing Surveys (CSUR), 2012, 44(2): 6.

[4] STANCU C, WIMMER C, BRUNTHALER S, et al. Comparing points-to static analysis with runtime recorded profiling data[C]// Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools. ACM, 2014: 157-168.

[5] CHELF B, ENGLER D, HALLEM S. How to Write System-specific, Static Checkers in Metal[C]// Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering. Charleston, SC, USA. ACM, 2003: 51-60.

[6] HALLEM S, CHELF B, XIE Y, et al. A system and language for building system-specific, static analyses[C]// Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation. ACM, 2002: 69-82.

[7] ARAUJO J E, SOUZA S, VALENTE M T. Study on the relevance of the warnings reported by Java bug-finding tools [J]. IET Software, 2011, 5(4): 366-374.

[8] KIM Y, KIM M, KIM Y J, et al. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE[C] // 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012: 1143-1152.

(上接第 74 页)

王瑞. 基于 SAT 的符号化模型检验技术研究[D]. 长沙：国防科学技术大学，2014

[4] YANG J J. Study on SAT-based Bounded Model Checking and its Applications [D]. Guangzhou: Sun Yat-sen University, 2008. (in Chinese)
杨晋吉. 基于 SAT 的有界模型检验及其应用研究[D]. 广州：中山大学，2008.

[5] SHEN S Y. Explaining Counter Example of Model Checking [D]. Changsha: University of Defense Technology, 2005. (in Chinese)
沈胜宇. 模型检验的反例解释[D]. 长沙：国防科学技术大学，2005

[6] PNUELI A. The Temporal Logic of Programs [C]//Proc. of 18th IEEE Symposium on Foundation of Computer Science (FOCS' 77). 1977: 46-57.

[7] EMERSON E A, CLARKE E M. Characterizing Correctness Properties of Parallel Programs Using Fixpoints [C]// Proc. of the 7th Int. Colloquium on Automata, Languages and Programming (ICALP'80). 1980: 169-181.