一种自动机学习和符号化执行的软件自动测试方法

陈 曙[1] 叶俊民[1] 张 帆[2]
(华中师范大学计算机学院 武汉 430079)[1] (杭州电子科技大学通信工程学院 杭州 310018)[2]

摘 要 针对高可信软件提出一种软件脆弱性自动测试方法。与传统测试方法不同,该方法对待测试程序进行预处理,使用自动机学习算法构造软件与环境交互的抽象机模型,在符号化执行迭代过程中利用抽象机模型指导符号化执行,并动态生成测试数据,同时精化交互抽象机用于后继的符号化迭代测试。解决了传统符号化执行测试技术中缺乏指引、具有较高盲目性的问题,同时也提高了符号化执行测试的效率和代码覆盖率。

关键词 自动机,自动测试,符号化执行,抽象机
中图法分类号 TP309 **文献标识码** A

Automatic Program Testing with Dynamic Symbolic Execution and Model Learning

CHEN Shu[1] YE Jun-min[1] ZHANG Fan[2]
(School of Computer Science, Huazhong Normal University, Wuhan 430079, China)[1]
(Communication School, Hangzhou Dianzi University, Hangzhou 310018, China)[2]

Abstract An automatic testing approach was proposed with dynamic symbolic execution and model learning. The model that represents I/O interaction of program with its environment was constructed by stepwise learning algorithm. With abstract interaction model, the process of dynamic execution is guided by states of the model, and test data is automatically generated. Abstract interaction model is also refined by the test data and used for further execution. The problem that traditional symbolic execution lacks guidance is solved, and its speed and code coverage rate are also improved.

Keywords Automation, Auto testing, Symbolic execution, Abstract machine

1 引言

近年来,针对软件可信度要求越来越高的特点,面向软件脆弱性分析的测试理论和技术研究得到了广泛的关注。传统基于人工分析构造测试用例的黑盒或白盒测试方法存在效率低下、代码覆盖率低等缺点,而自动化测试是改进这些缺陷的一个有效途径[1,2]。自动化测试通过测试引擎自动或半自动生成测试用例,驱动程序执行,记录和监测程序的执行情况,若程序产生运行时异常则报警,从而达到测试的目的。自动测试方法虽然解决了人工测试效率低下的缺点,但仍存在一些固有的局限性,例如缺乏针对性,随机性较高,难以保证代码覆盖率等。

动态符号化执行技术在基于脆弱性分析的自动测试方法中产生了较为重要的影响。符号化执行的主要思想是使用代表任意含义的符号变量来代替具体的数据作为程序输入,在程序执行过程中利用符号变量作为逻辑变量构造执行分支约束表达式,并进行求解,得出满足约束的符号变量随机值,利用求解结果作为新的测试数据,驱动程序迭代执行,以覆盖不同的执行分支[2-4]。这种方法虽然改善了传统自动测试方法中的一些缺陷,但仍然存在一些不足。例如由于缺乏指引,符号化执行仅能掌握程序执行的当前局部状态,而无法获知全局的状态信息,因此存在较大盲目性,即无法知晓当前执行状态与测试者感兴趣的目标状态的距离;此外,可能陷入某些局部循环中而消耗大量时间和空间,从而直接导致代码覆盖率低[4]。另外,该方法无法利用当前符号执行迭代的信息作为辅助来加快后继迭代的速度[4]。

针对符号化执行中的不足之处,本文提出一种基于自动机学习的动态符号化执行测试方法。从操作语义角度看,软件测试的本质可看作通过构造不同的输入数据驱动程序执行以遍历程序的执行状态空间,而判断测试的代码覆盖率可等同为状态的遍历覆盖率,测试的效率等同为状态的遍历效率。基于此种观点,本方法针对待测试程序进行状态预处理,使用自动机学习算法构造软件与环境交互的抽象状态模型,并利用该模型指引符号化执行,以提高状态的覆盖率和遍历效率。该方法的主要贡献如下:

1)通过对程序进行前期建模处理,利用模型指导符号化执行的方向,解决了传统符号化测试过程中缺乏指引、具有盲目性的缺点。

到稿日期:2012-10-21 返修日期:2013-01-14 本文受中央高校自主科研基金(CCNU11A01012,CCNU11A02007),湖北省自然科学基金(2010CDB04001)资助。

陈 曙(1981—),男,博士,讲师,主要研究方向为软件工程、可信软件、信息安全;叶俊民(1965—),男,博士后,教授,主要研究方向为可信软件工程;张 帆(1978—),男,博士,讲师,主要研究方向为软件工程、可信计算。

2)通过建立系统交互模型,并在符号化执行迭代中对其精化,记录每次符号化迭代的信息,提高后继迭代的速度,进一步提高符号化执行的代码覆盖率。

3)该方法能够用于自动化单元测试和集成测试。

本文第2节简要介绍动态符号化执行;第3节介绍交互抽象模型的构造方法;第4节介绍如何利用交互模型指导符号化执行;最后总结。

2 动态符号化执行

符号化执行将代表数值输入的符号变量作为符号输入,构造符号表保存符号的变化,并利用符号变量在路径分支中构造分支约束表达式。通过对约束表达式部分取反来构造新的表达式以代表新的执行分支,同时求解生成新的输入数据,驱动程序执行新的分支。将分支约束条件看成节点,则程序执行过程直观上可表示为一棵树,利用深度优先遍历,覆盖测试所有分支路径。动态符号化执行过程简要描述如下:

1)首次执行时,利用任意具体数据输入驱动程序常规执行,同时构造与输入数据对应的符号变量。

2)如果出现执行路径分支,则利用符号变量作为逻辑变量来构造分支条件逻辑表达式。

3)利用现有的逻辑表达式按照规则构造新的分支条件逻辑表达式(例如将现有的表达式按右优先取反,若现有逻辑表达式为 $x>0\wedge y<0$,其中 x 和 y 均为符号化输入,则可生成新的逻辑表达式:$x>0\wedge y\geqslant 0$,$x\leqslant 0\wedge y<0$,$x\leqslant 0\wedge y\geqslant 0$ 等)。

4)对于所有新产生的表达式,如果其能够代表一个新的执行分支,则对这些新产生的逻辑表达式求解,用所得到的符号变量解作为新的输入数据,驱动程序常规执行以生成新的执行分支。

不断迭代上述过程,直到达到目标(例如满足一定的分支覆盖率或发现一定数量的缺陷)为止。

3 基于污点数据信息流的软件可信分析模型

本文利用 Dana Angluin[6-8] 提出的自动机学习算法 L^* 构造目标系统与外部环境的交互模型。首先进行如下假设:

1)系统的输入类型仅包括整数、浮点数、字符、字符串、指针、引用,并将其抽象为字符串形式表示。

2)目标系统输入输出字符集是已知的,且目标系统在任意执行点均能返回到初始状态(可通过重启目标系统等方法来完成)。

3)系统输出类型包括整数、浮点数、字符、字符串、指针,并将其抽象为字符串形式表示。

系统与环境的交互模型表示为一台抽象机。

定义 1(输入输出抽象函数) 输入抽象函数定义为 *Iabs*,将包括整数、浮点数、字符、字符串、指针、引用类型的输入数据抽象为字符串类型,并记录其原始类型。*Iabs* 的反函数表示为 $Iabs^{-1}$,将输入数据反射为原始类型数据。输出抽象函数定义为 *Oabs*,将输出数据抽象为字符串,并记录其原始类型。同理,反函数 $Oabs^{-1}$ 将抽象输出反射为原始类型数据。

对于函数 *Iabs* 和 *Oabs* 及其对应的反射函数,我们构造逻辑地址映射表 Π 建立逻辑地址到值的映射。其目的在于消除指针类型数据、引用类型数据或复杂类型的成员数据(例如指针类型可能表示某些复杂类型的数据,即可能指向一片具有特定结构的内存区域,如结构体、对象等)存在间接引用而导致符号化后直接进行逻辑判断时产生的偏差。令 N 为表示逻辑地址的自然数集,V 为目标程序的值集,逻辑地址映射表 Π 定义为 $\Pi:N\rightarrow N\cup V$。对于任意普通数据类型输入 x,Π 为其分配初始逻辑地址,记录其长度 $sizeof(type(x))$。同理,对于复杂数据类型 y,Π 为其分配初始逻辑地址,根据其成员类型定义记录长度 $\sum_{m\in member(y)} sizeof(type(m))$,并递归记录包括偏移地址、长度等成员信息。

定义 2(系统交互抽象机,以下简称交互抽象机) 交互抽象机被定义为一个六元组 $M=(S,s_0,I,O,Trans,Prod)$。其中 S 为状态集,s_0 为初始状态,I 为输入数据字符集,O 为包含空字符在内的输出字符集,$Trans$ 表示内部状态迁移函数,定义为 $Trans:S\times I\rightarrow S$,$Prod$ 表示输出函数,定义为 $Prod:S\times I\rightarrow O$。

交互抽象机可看作一个接受输入,并产生输出的黑盒。黑盒的状态由输入数据经过 *Iabs* 抽象后的名字排列组成,直观上可看作一张二维表,定义为 $Ot:(R,E,Entry)$,其中表示行的集合 R 定义为 $R=Rm\cup Rm\circ I$,Rm 为包括空串 ε 在内的、由输入数据的字符集 I 所组成的前缀封闭字符串集。表的列集合 E 为由输入数据的字符集 I 所组成的后缀封闭字符串集。初始状态下 $Rm=\{\varepsilon\}$,$E=I$。函数 $Entry$ 表示行列到经过 *Oabs* 符号化抽象后输出的映射,定义为 $Entry:R\times E\rightarrow O^*$。抽象机 M 的状态 S 表示为二维表 Ot 中所有的行元素。下面给出状态等价的定义。

定义 3(状态等价) 给定交互抽象机 M 及其对应的二维表 $Ot:(R,E,Entry)$,抽象机 M 中两个状态等价表示为 $s\cong_M t$,当且仅当其输出对应的实际类型数据相同。形式定义如下:

$$\forall e\in E:Oabs^{-1}(Entry(s,e))=Oabs^{-1}(Entry(t,e))$$

用 $[s]$ 表示所有与状态 s 等价的状态形成的等价类。根据上述定义,交互抽象机 M 的状态迁移表示为 $\forall s\in S,i\in I:Trans([s],i)=s\circ i$,输出函数表示为 $\forall s\in S,i\in I:Prod([s],i)=Entry(s,i)$。

定义 4(封闭性) 给定二维表 $Ot:(R,E,T)$ 满足封闭性,当且仅当所有在二维表中能够观察到的状态在抽象机中都能出现。形式定义为:

$$\forall s\in Rm\circ I:(\exists t\in Rm:s\cong_M t)$$

定义 5(稳定性) 给定二维表 $Ot:(R,E,T)$ 满足稳定性,当且仅当在所有等价状态下,对于相同输入,不存在不同输出。形式定义为:

$\forall s,t\in Rm:(s\cong_M t\Rightarrow\forall i\in I:s\circ i\cong_M t\circ i)$

利用算法 L^* 的思想来构造交互抽象机，构造算法简要描述如下：

1)观察系统的输入输出，更新二维表 $Ot:(R,E,Entry)$，并判断二维表是否封闭且稳定。如果不满足稳定性或封闭性，则进行步骤 2)，否则转到步骤 3)。

2)如果二维表 Ot 不满足定义 5 的稳定性，则找出二维表中所有满足 $s\cong_M t$，且 $Entry(s\circ i,e)\neq Entry(t\circ i,e)$(其中 $i\in I,e\in E$)的字符串 $i\circ e$，并将其加入到集合 E 中。如果二维表 Ot 不满足定义 4 的封闭性，则找到所有满足 $t\in R\circ I$，且对于所有的 $s\in R$，满足 $s\ncong_M t$ 的状态 t，并将 t 加入到 R 中，同时计算 $Entry(t\circ i,e)$。回到步骤 1)。

3)如果二维表 Ot 满足稳定性和封闭性，算法 L^* 根据定义 2 生成系统交互抽象机候选 M^i。

4)继续观察系统输入输出，如果观察到一组输入序列 i^*，且对应的输出序列 o^* 与 i^* 在 M^i 中的输出不一致，则称 i^* 为 M^i 的一个反例。

5)根据该反例，精化候选 M^i，得到 M^{i+1}。精化过程描述为，将反例 i^* 中所有的前缀字符串加入到符号集 Rm 中，并按照上述过程再次生成抽象机。精化后的抽象机比精化前具有更多的状态，即对系统进行更精确的等价划分。

6)不断迭代执行上述过程，直到在设定时间阈值内抽象机 M^n 不再产生反例，即可将候选 M^n 作为最终的系统交互抽象机。

假设在理想情况下抽象机 M^n 不再产生任意长度的反例，则称 M^n 中的状态集 S^n 是对程序中所有可能输入所产生精确状态的一个完整等价类划分。然而实际情况下很难做到如此，因此我们设定时间阈值，如果在阈值内不再产生反例，则停止构造 M^n。

令 $|S|$ 为输入字符集 S 的元素个数，m 和 n 分别为代表反例的平均输入长度和二维表行数，则构造交互抽象机 M^i 的平均算法复杂度为 $O(|S|mn)$。根据 M^n 中的状态迁移和输出函数，易获取从初态到各个状态的迁移路径、输入以及输出。有了这些信息即可利用其来指导符号化执行。

4 基于系统交互模型的动态符号化执行

动态符号化执行利用与输入数据对应的污点符号变量来构造分支条件表达式，并通过按右取逆的原则构造新的分支条件式，同时计算新表达式的符号变量解，将这些解作为新的测试数据驱动程序执行，以发掘新的执行分支。如图 1 所示，基于系统交互模型的动态符号化执行可描述为动态符号化执行与精化交互抽象机交替进行。其中交互抽象机产生达到各个状态所需的输入序列，用于指导符号化执行，而符号化执行过程中产生的符号解和反例又用于精化交互抽象机。监视器监视符号化执行情况，如果产生运行时错误，则报警，并输出导致错误的输入等相关信息。

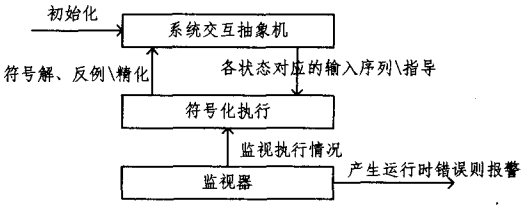

图 1 基于系统交互模型的动态符号化执行过程

令 $M=(S,s_0,I,O,Trans,Prod)$ 为由算法 L^* 在初态下生成的系统交互抽象机。由于对应的二维表满足封闭性和稳定性，因此根据迁移和输出函数可获得从 s_0 到每个状态 $s_i\in S$ 的迁移序列，记为 s_0*s_i，输入序列记为 i_0*i_i，以及输出序列记为 o_0*o_i。

对于任意状态迁移序列 s_0*s_i，将抽象输入序列 i_0*i_i 结合类型函数获得真实输入及其类型，并作为种子数据输入待测试程序中驱动程序执行，直到数据处理完毕。易见此时程序处于状态 s_i 下。由于状态 s_i 为交互抽象机中的抽象状态，根据交互抽象机的构造原理不难看出，状态 s_i 为一个具有相同特性的状态群集的等价类抽象。因此在状态 s_i 下进行符号化执行，能够有效遍历到 s_i 周围的状态。而在某时刻下，若状态集 S 及其状态迁移作为目标程序中所有输入所产生状态的一个近似完整的刻画，则称由该状态集及其迁移所构造的交互抽象机为系统在该时刻下的一个快照，如图 2 所示，其中大圆表示抽象状态，小圆表示抽象状态所代表的精确状态。在每个抽象状态 $s_i\in S$ 下分别进行符号化执行以遍历其周围的精确状态，能够有效提高状态的覆盖率，从而提高测试代码覆盖率。

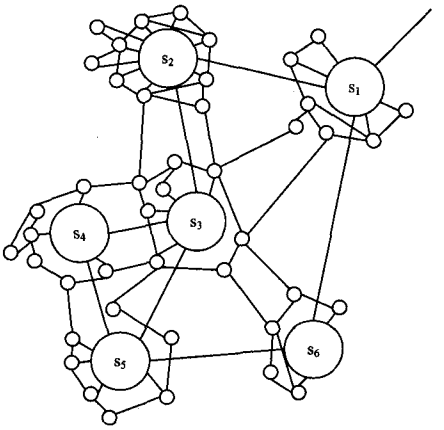

图 2 交互抽象机快照

在实际操作中，我们采用多线程的方式，将待测试目标程序分为一定量的副本，以 $i_0*i_i,i_0*i_{i+1},\cdots,i_0*i_{i+n}$ 作为种子数据，多线程驱动程序符号化执行，亦能够提高一定的效率。

观察上一节中系统交互抽象机的生成过程可知，如果产生反例，则需要对抽象机做进一步精化。令 M^i 为交互抽象机，在符号化执行过程中，对于由新产生的分支条件表达式计算得到的作为程序新输入符号变量解，以及对应的程序输出，如果发现这些解是 M 的一个反例，则利用该反例精化 M^i 得

到 M^{i+1}，并用 M^{i+1} 代替 M^i 指导符号化执行。由于每次符号化迭代均会因产生反例而对交互抽象机进行多次精化，即对系统进行更为精确的等价划分，从而产生新的交互抽象机快照，因此每次迭代的输入信息及其生成的状态均被记录在交互抽象机中。根据前面叙述，用当前抽象机指导后继符号化迭代时，利用抽象机记录的输入信息驱动目标程序直接进入相应状态，避免了传统符号化执行中可能存在的重复判断执行，提高了符号化执行效率。

5 实验分析

我们在 Dazhi-Zhang[9] 的符号化执行工具 SecTAC 源代码的基础上实现了基于本方法的系统原型 SMATS(SYMBOLIC MODEL AUTO TEST SUIT)。SecTAC 是针对 C 语言程序符号化单元测试的一个开源工具，通过扫描 C 语言程序代码，利用代码插装方式植入符号化执行函数，编译后进行动态符号化自动测试。根据第 3、4 节中介绍的方法，SMATS 在 SecTAC 基础上加入了交互抽象机生成和精化模块，与 SecTAC 联动执行，对类 C 语言程序进行自动测试。我们用 SMATS 和 SecTAC 对 LINUX 下的开源软件 EVOLUTION 分别进行了测试，在 EVOLUTION 源代码中插入了 10 个千万次的子循环陷阱，通过特定的种子输入引导测试进入陷阱。测试环境为：UBUNTU9.04，INTEL I7 860 3.46GHz，4GBDDR3-1600，在 10 次测试中分别生成 100 个输入，并记录测试时间、覆盖的代码块数量以及进入陷阱的次数。取相对典型的 3 次结果，如表 1 所列。

表 1 两种测试方法结果

测试方式	覆盖代码块数量	耗时(秒)	进入陷阱的次数	测试编号
符号化执行	1353908	2023	1	1
SMATS	2232823	3563		
符号化执行	1023456	5376	6	2
SMATS	2054324	3892		
符号化执行	1232231	7231	9	3
SMATS	1912234	4328		

由表 1 可看出，编号 1 的测试中 SMATS 方法共消耗 3562 秒，传统符号化执行方法消耗 2023 秒，虽然 SMATS 需要对程序前期建模处理而耗时较多，但测试覆盖的代码块数量却大大超出传统方法。编号 2 和 3 的测试中，传统方法由于陷入陷阱次数增多，因此耗时显著增加，而 SMATS 方法所耗费的时间仅略微增加。由此可看出，SMATS 方法不仅能够在可接受的范围内增加代码覆盖度，且在特殊情况下(如大规模子循环较多的情况)有更为稳定的表现。由于篇幅限制，将在后续文献中对该工具进行详细介绍。

结束语 传统的自动化测试方法通过测试引擎自动生成测试用例驱动并监视程序执行，捕获执行中产生的异常。这些方法虽然从一定程度上提高了测试效率，但也带来了一些缺点，例如缺乏针对性，难以保证代码覆盖率等。相对传统方法，符号化测试方法虽然能够提高测试路径覆盖率，但仍然存在缺乏指导、具有盲目性的问题，这就导致了难以对测试过程进行控制和预知，从而限制了测试中的代码覆盖率和测试效率。

由于提高测试代码覆盖率和效率的本质在于提高测试过程中系统执行状态遍历的覆盖率和效率，因此本文提出通过对待测试程序进行状态建模预处理的方法，即通过构造目标程序与环境的输入输出交互模型抽象机来指导符号化执行。由于交互模型记录了由各种输入组合所产生的系统抽象状态划分，因此在这些已知的抽象状态下进行符号化执行能够有效遍历其所代表的精确状态，从而提高状态覆盖率，进而提高测试代码覆盖率。此外，模型在每次符号化迭代所产生的符号解及输出中不断被精化，因此能够用其积累的信息指导符号化执行，并且加快后继符号迭代的执行速度。

由于加入了构造和精化交互抽象机这一环节，因此不可避免地将会在此处增加测试时间和空间开销，今后的工作之一是进一步优化抽象机构造算法。另外，将该方法应用于面向对象、面向服务等应用测试中也是今后的研究方向之一。

参考文献

[1] Godefroid P, Klarlund N. DART: Directed Automated Random Testing [C]//PLDI05. Chicago, Illinois, USA, June 2005: 12-15

[2] King J C, Wegbreit B. Symbolic Execution and Program Testing [J]. Communications of the ACM on Programming Languages, 1976, 19(7): 385-394

[3] Sen T, Mall R. State-Model-Based Regression Test Reduction for Component-Based Software [J]. ISRN Software Engineering, 2012(7): 15-26

[4] Alsmadi I. Advanced Automated Software Testing: Frameworks for Refined Practice [M]. Yarmouk University, Jordan, 2012: 209-224

[5] Cho C Y, Babi D, Poosankam P. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery[C]//Proceedings of the 20th USENIX conference on Security(SEC 11). Usenix, 2011: 78-90

[6] Angluin D. Learning regular sets from queries and counterexamples[J]. Information and Computation, 1987, 75(2): 87-106

[7] Shahbaz M, Groz R. Inferring Mealy machines [C]//FM'09: Proc. of the 2nd World Congress on Formal Methods. Springer, 2009: 207-222

[8] Shu Guo-qiang, Lee D. Testing Security Properties of Protocol Implementations-a Machine Learning Based Approach [C]// 27th International Conference on Distributed Computing Systems(ICDCS'07). 2007: 34-43

[9] Zhang Da-zhi, Liu Dong-gang, Wang Wen-hua. Detecting Vulnerabilities in C Programs Using Trace-Based Testing [C]// Proc. 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE Publisher, 2010: 241-250