符号执行技术研究

林锦滨[1,2] 张晓菲[1] 刘 晖[1]
（1.中国信息安全测评中心南亚 北京 100085）
（2.中国科学技术大学 安徽合肥 230026）

摘 要：本文首先介绍了符号执行的过程内分析和程序全局分析的基本原理；接着介绍了其中的路径状态空间爆炸、复杂结构语义和操作语义建模及程序全局分析这三个关键技术的难点及相关研究；最后介绍了一些使用符号执行技术开发的研究型工具。本文从基本原理到技术难点，从理论基础到工具，系统全面地展现了符号执行技术。

关键词：符号执行，静态分析，源代码安全

1 简介

软件的安全性越来越受到人们的重视，如何检测软件中存在的安全问题也逐渐成为软件工程领域的研究热点。《Software Security》一书的作者Gary McGraw估计，在引发安全问题的错误中，大约有一半是实现时的疏忽、遗漏或误解造成的[1]。基于源代码的静态分析工具是检测软件编码安全问题的一种有效方法。

静态分析是一种在不执行程序的情况下对代码进行评估的技术，它通过对当前状态及趋势的分析来预判未来所有软件运行可能的情况。源代码静态分析方法涉及的理论包括指称语义、公理化语义、操作语义、抽象解释理论等，常见的实现技术有模型检验、数据流分析、抽象解释、谓词转换、定理证明、类型推导、符号执行等。其中，符号执行是目前源代码静态分析工具常采用的主要技术之一，本文将重点介绍符号执行技术的基本原理、关键技术及发展现状。

2 符号执行的基本原理

2.1 基本原理

符号执行是指在不执行程序的前提下，用符号值表示程序变量的值，然后模拟程序执行来进行相关分析的技术，它可以分析代码的所有语义信息，也可以只分析部分语义信息（如只分析“内存是否释放”这一部分的语义信息）。

符号执行分为过程内分析和过程间分析（又称全局分析）。过程内分析是指只对单个过程的代码进行分析，全局分析指对整个软件代码进行上下文敏感的分析。所谓上下文敏感分析是指在当前函数入口点要考虑当前的函数间调用信息和环境信息等。程序的全局分析是在过程内分析的基础上进行的，但过程内分析中包含了函数调用时就也引入了过程间分析，因此两者之间是相对独立又相互依赖的关系。

过程内分析流程如图1所示。首先，对待分析的单个过程代码对象构建控制流图（Control Flow Graph, CFG）。控制流图（CFG）是编译器内部用有向图表示一个程序过程的一种抽象数据结构，图中

作者简介：林锦滨（1985— ），男，福建，中国信息安全测评中心实习研究生，中国科学技术大学硕士，ljb@mail.ustc.edu.cn

的节点表示一个程序基本块，基本块是没有任何跳转的顺序语句代码块，图中的边表示代码中的跳转，它是有向边，起点和终点都是基本块。在CFG上从入口节点开始模拟执行，在遇到分支节点时，使用约束求解器判定哪条分支可行，并根据预先设计的路径调度策略实现对该过程所有路径的遍历分析，最后输出每条可执行路径的分析结果。其中，约束求解是数学上的判定过程，形象地说是对一系列的约束方程进行求解。

如果要进行源代码的安全性检测，则需要在过程内分析时，根据具体的安全知识库来添加安全约束。例如，如果要添加缓冲区溢出的安全约束，则在执行时遇到对内存进行操作的语句时，就要对该语句所操作的内存对象的边界添加安全约束。以上面的方式来进行安全约束的添加，并且每次在添加之后就使用约束求解器对所有的安全约束进行求解，以判定当前是否可能潜在一个安全问题。

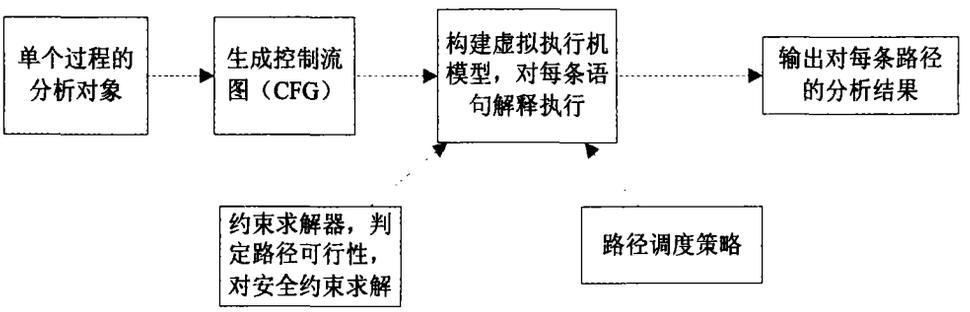

图 1 过程内分析原理流程图

程序全局分析流程如图1所示。首先，为整个程序代码构建函数调用图（Call Graph，CG），在函数调用图中，节点表示函数，边表示函数间的调用关系。根据预设的全局分析调度策略，对CG中的每个节点（对应一个函数）进行过程内分析，最终给出CG每种可行的调用序列的分析结果。

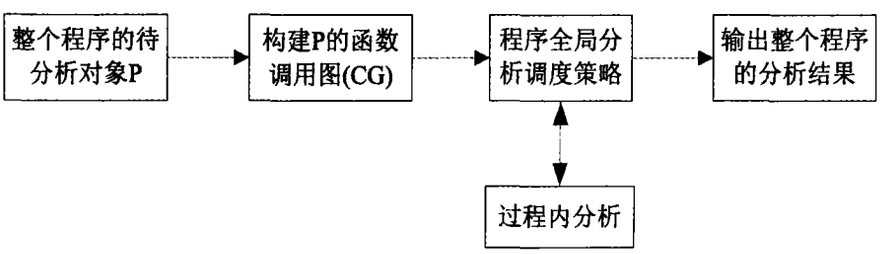

图 2 全局分析原理流程图

符号执行在发展过程中出现了一种叫做动态符号执行的方法。动态符号执行是以具体数值作为输入来模拟执行程序代码，与传统静态符号执行相比，其输入值的表示形式不同。动态符号执行使用具体值作为输入，同时启动代码模拟执行器，并从当前路径的分支语句的谓词中搜集所有符号约束。然后修改该符号约束内容构造出一条新的可行的路径约束，并用约束求解器求解出一个可行的新的具体输入，接着符号执行引擎对新输入值进行一轮新的分析。通过使用这种输入迭代产生变种输入的方法，理论上所有可行的路径都可以被计算并分析一遍。

动态符号执行相对于静态符号执行的优点是每次都是具体输入的执行，在模拟执行这个过程中，符号化的模拟执行比具体化的模拟执行的花销大很多；并且模拟执行过程中所有的变量都为具体值，而不必使用复杂的数据结构来表达符号值，使得模拟执行的花销进一步减少。但是动态符号执行的结果是对程序的所有路径的一个“下逼近”，即其最后产生路径集合应该比所有路径集合小，但这种情况在软件测试中是允许的。

2.2 符号执行示例

本节将通过一个示例代码来具体说明符号执行的分析原理。如图3所示，先对一段简单的代码

函数 test 构建控制流图 CFG，可以看出该 CFG 只包含了 2 条路径。接着，以符号值为输入，模拟执行代码，在遇到分支语句时，使用约束求解器判定这两条路径的可行性，本示例使用的是“深度优先”的路径调度策略，详细的分析过程见图 3。在对两条路径模拟执行并收集路径约束后，使用约束求解器可以求解出分别触发这两条路径的两个测试例输入及对应的返回值。测试例输入为“i=11”时，返回值“10”；测试例输入为“i=10”时，返回值为“9”。

通过这个简单的例子可以看出使用符号执行的方法进行分析可以达到很高的路径覆盖率，并且结合约束求解器还可以实现测试例的自动生成。但是这仅仅是符号执行技术和约束求解结合的具体应用的一个方面，符号执行技术还可以有其他的应用。

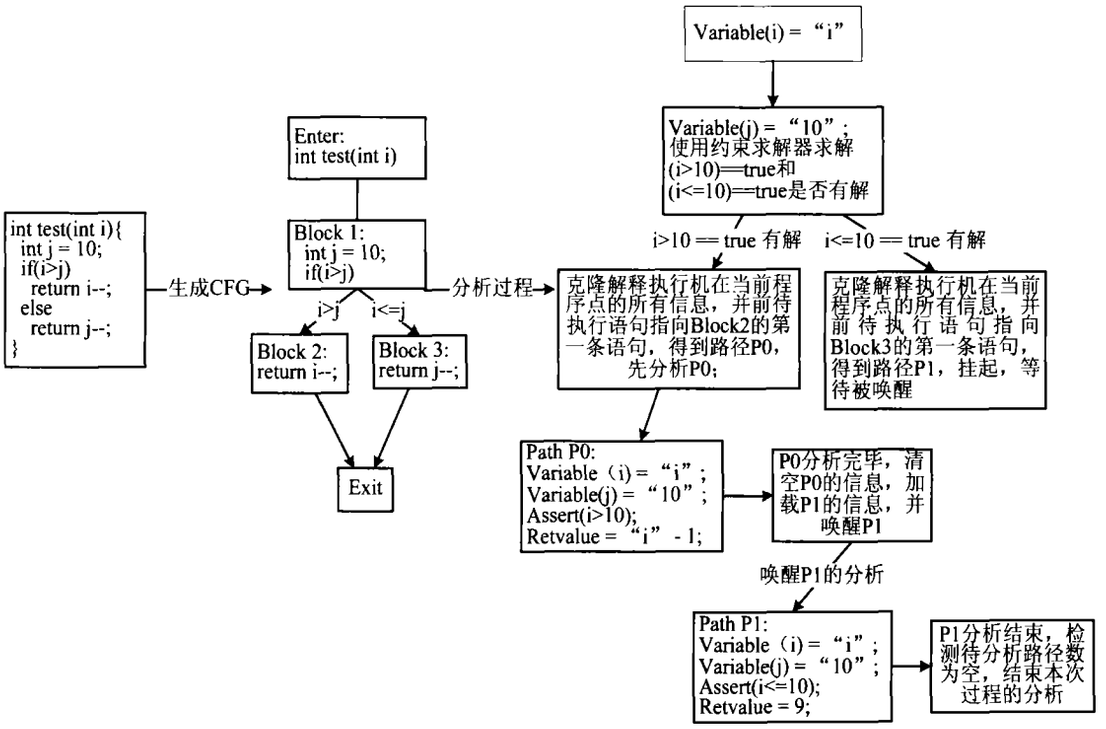

图 3 符号执行原理示例

3 符号执行的关键技术及发展现状

上节介绍了符号执行的基本原理，本节将重点介绍符号执行的关键技术及其解决方法，此外，还将介绍使用该技术开发的工具。

3.1 符号执行的关键技术

3.1.1 路径状态空间爆炸

符号执行技术在理论上面临着路径状态空间的爆炸问题，其主要形成原因是每一个分支条件语句都可能会使当前的路径再分支出一条新的路径，而这是“指数级”增长的。相对于静态符号执行在分支语句时都是符号值的约束，动态符号执行在分支语句时则都是具体值的约束。动态符号执行的当前路径执行过程中不会产生新路径，只在执行结束后再去产生新路径，且一次只产生一条。但是它们都没法从根本上避免路径状态空间爆炸所造成的影响。

一方面，在具体的实现中有使用限定每个过程内的分析路径数目上限的方法来缓解该问题所产生的影响，如Coverity，也可以使用设置时间上限或者内存空间上限的方法来缓解路径爆炸问题所可能造成分析工具崩溃的恶劣影响，如Fortify。另一方面，符号执行工具的实现者总是希望能设计出好的路径遍历策略，以在有限的时间和空间范围内达到最大的代码检测覆盖率。人们通过改进路径调度算法来提高符号执行的分析性能，但是这些路径调度算法都只是局部改进，很难从根本上解决这一问题。

3.1.2 复杂结构语义和操作语义的建模

符号执行实现时需要对被分析代码的结构语义进行建模，然后再对被分析代码的操作语义进行

建模，最后构建一个虚拟机模型。由于符号执行是路径敏感的分析方法，因此一般为每条路径都会创建一个专属的虚拟机模型，以保证路径之间的相互独立性。该模型的准确程度将直接影响静态分析结果的精度。由于编程语言中使用的复杂数据结构和复杂操作语句具有较高的灵活性，使得它们的建模变得十分困难。

人们提出了一种“惰性初始化”[2]的方法，其思想是为每个数据结构（特别是复杂数据结构）建模时，在声明或者定义时只为其构建类型信息，直到被使用的时候，才根据使用的需要来初始化该变量的对象信息。此外，人们还[3,4,5,6]分别对不同复杂对象的建模方法提出了解决方案。

3.1.3 程序全局分析

在程序全局分析过程中，当对一个规模较大、包含很多的过程间调用的程序进行上下文敏感的分析时，每当一个过程调用了另一个过程时都进入子过程进行分析，虽然会很精确，但这种方式可能会造成大量的时间空间花销，而使分析过程异常中止或在用户可接受的时间内无法完成。

一种比较好的全局分析方法叫做“函数摘要”。函数摘要的方法是在过程内分析的基础上对已分析过的函数进行一个摘要记录的操作。在以后的分析中遇到调用其他函数时，如果已存在被调用函数的摘要，则直接调用该函数的摘要并对该摘要行为进行解释；如果不存在被调用函数的摘要信息，则进入被调用函数进行分析，并在分析之后进行摘要保存。函数摘要是一种相对折中的办法，所创建的函数摘要也可能很不准确。另一个问题是对无法获得源代码的第三方库函数的摘要只能通过人工的编写来完成，这也可能是不精确的，而这两点都会影响到最终的分析结果的精度。

对于函数摘要的构建策略也是一个研究内容，文章[7]使用的是在函数调用图上的自顶向下的需求驱动的策略来为所有子过程创建函数摘要。犹豫循环语句可能会使被摘要函数的路径空间爆炸，而使得无法对函数的所有路径都进行分析摘要，该文章介绍了结合使用循环不变量来进行函数摘要以解决循环语句所引入的问题。

3.2 符号执行的研究型工具

符号执行应用比较多的方面有：单元测试、代码漏洞静态检测等。前者是只进行过程内的分析，后者是进行程序全局的分析且更侧重于代码的安全性相关的检测。符号执行技术与约束求解器结合使用来产生测试例输入是一个比较热门且比较实用的研究点。相关的较新的研究性工具有EXE[7]，CUTE[8]，JPF-SE[9]，DART[10]，SAGE[11]，SMART[12]，PEX[13,14]，KLEE[15]等；其他相关的研究文章有[16~21]。这些研究也是从前面的三个难点提出自己的改进算法，如采用动态符号执行技术，对约束求解器性能的优化，程序全局分析策略的优化，路径搜索调度策略优化，以及基于x86汇编指令的分析等几个方面。

4 结语

本文从符号执行的原理、关键技术及相关工具三个方面系统地介绍了符号执行技术。相对于其他的静态分析方法，符号执行方法的分析结果要更为精确，因此该技术也越来越多地被应用于实际的工具中。但是该技术目前还处于一个发展的阶段，包括符号执行的理论改进研究和具体应用的研究。符号执行技术可以用于测试例的自动生成，也可以用于源代码安全性的检测。而这两项工作的成效都十分依赖于约束求解器的性能，同时还受硬件设备处理能力的影响。符号执行技术在实际应用中的效果还是主要取决于前面提到的三个关键技术，因此更多的研究团体也都主要致力于解决本文前面提到的几个问题。随着符号执行理论及约束求解理论的发展，该技术的实用价值性也会越来越高，将会在静态分析领域逐渐的占据主导地位。

源代码静态分析的目标是要找出代码中潜在的漏洞，使用多种静态分析技术结合来开发静态分析工具将是一个可行且有效的方法。虽然这些静态分析工具还无法完美，甚至说效果比较一般，但随着静态分析技术的发展，或者结合其他技术（如动态分析），未来的静态分析工具一定会在软件的安全保障中起到越来越重要的地位。

参考文献

[1] McGraw, Gary. Software Security: Building Security In.

Boston, MA: Addison-Wesley, 2006.

[2] Visser W, Păsăreanu C S, Khurshid S. Test input generation with java PathFinder, ISSTA'04, July 11–14, 2004.

[3] Rugina R, Rinard M C. Symbolic bounds analysis of pointers, array indices, and accessed memory regions, ACM Transactions on Programming Languages and Systems (TOPLAS) Volume 27 ,(March 2005) .

[4] Jian Zhang. Symbolic execution of program paths involving pointer structure variables, Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on Publication Date: 8-9 Sept. 2004: 87- 92.

[5] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies and Hongseok Yang, Shape Analysis for Composite Data Structures , Lecture Notes in Computer Science,2007.

[6] A Ermedahl, C Sandberg, J Gustafsson, S Bygde, and Bjorn Lisper, Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis , Workshop on Worst-Case Execution Time Analysis,(WCET'2007).

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In Proc. ACM CCS, 2006: 322–335.

[8] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. In Proc. ESEC/FSE, 2005: 263–272.

[9] Anand S, Pasareanu C S, Visser W. JPF-SE: A symbolic execution extension to Java Pathfinder. In Proc. TACAS, 2007: 134–138.

[10] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In Proc. PLDI, 2005: 75–84.

[11] GODEFROID P, LEVIN M Y, MOLNAR D. Automated whitebox fuzz testing. In NDSS '08: Proceedings of Network and Distributed Systems Security, 2008:151–166.

[12] Godefroid P. Compositional dynamic test generation. In Proc. POPL, 2007:47–54.

[13] Microsoft Research Foundation of Software Engineering Group, Pex: Dynamic Analysis and Test Generation for .NET, 2007. http://research.microsoft.com/Pex/.

[14] Tillmann N, de Halleux J. Pex-white box test generation for .NET. In Proc. TAP, 2008: 134–153.

[15] Cristian Cadar, Daniel Dunbar, Dawson Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

[16] CADAR C, ENGLER, D. Execution generated test cases: How to make systems code crash itself. In Proceedings of the 12th International SPIN Workshop on Model Checking of Software (August 2005).

[17] BOONSTOPPEL P, CADAR C, ENGLER D. RWset: Attacking path explosion in constraint-based test generation. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) (2008).

[18] COSTA M, CROWCROFT J, CASTRO M, ROWSTRON A, HOU L, ZHANG L, BARHAM P. Vigilante: end-toend containment of Internet worms. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP) (October 2005).

[19] COSTA M, CASTRO M, ZHOU L, ZHANG L, PEINADO M. Bouncer: Securing software by blocking bad input. In Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP) (October 2007).

[20] BRUMLEY D, NEWSOME J, SONG D, WANG H, JHA S. Towards automatic generation of vulnerability-based signatures. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006).

[21] EMMI M, MAJUMDAR R, SEN K. Dynamic test input generation for database applications. In International Symposium on Software Testing and Analysis (ISSTA'07) (2007), ACM.