计算机工程与应用
Computer Engineering and Applications
ISSN 1002-8331,CN 11-2127/TP

《计算机工程与应用》网络首发论文

题目：　符号执行技术及应用研究综述
作者：　吴皓，周世龙，史东辉，李强
网络首发日期：　2022-12-16
引用格式：　吴皓，周世龙，史东辉，李强．符号执行技术及应用研究综述[J/OL]．计算机工程与应用. https://kns.cnki.net/kcms/detail//11.2127.TP.20221215.1039.002.html

符号执行技术及应用研究综述

吴皓[1]，周世龙[2]，史东辉[1]，李强[2+]
1. 安徽建筑大学 电子与信息工程学院，合肥 230601
2. 国防科技大学 电子对抗学院，合肥 230037

摘 要：符号执行是一种程序分析技术，通过收集程序路径上约束条件并利用约束求解器生成高覆盖率的测试用例，能发现深层次程序错误的优势。首先，梳理了符号执行概念和发展历程，从符号执行系统核心设计切入，对符号执行技术的中间语言、路径搜索和约束求解进行分类阐述。接着，调研现有研究工作进展，选取应用最突出的安全漏洞方面，从漏洞利用与漏洞检测上系统地分析符号执行技术应用细节。最后，依据符号执行技术特点选取一些研究成果整理分析，探讨符号执行技术面临的局限与解决方案，并展望了未来趋势。

关键词：符号执行；符号执行系统；约束求解；漏洞利用；漏洞检测
文献标志码：A **中图分类号**：TP309; TP311 doi：10.3778/j.issn.1002-8331.2209-0359

A Review of Symbolic Execution Technology and Applications

WU Hao[1], ZHOU Shilong[2], SHI Donghui[1], LI Qiang[2+]
1. School of Electronic and Information Engineering, Anhui Jianzhu University, Hefei 230601
2. School of Electronic Countermeasures, National University of Defense Technology, Hefei 230037

Abstract: Symbolic execution is a program analysis technique that has the advantage of finding deep program errors by collecting constraints on program paths and generating high-coverage test cases using constraint solvers. First, the concept and development history of symbolic execution are sorted out, and the intermediate language, path search and constraint solving of symbolic execution techniques are categorized and explained from the core design of symbolic execution system. Then, we investigate the progress of existing research work, select the most prominent security vulnerabilities, and systematically analyze the details of the application of symbolic execution technology in terms of vulnerability exploitation and vulnerability detection. Finally, some research results are selected and analyzed according to the characteristics of symbolic execution technology, and the limitations and solutions faced by symbolic execution technology are discussed, and the future trends are foreseen.

Key words: symbolic execution; symbolic execution system; constraint solving; exploit generation; vulnerability detection

基金项目：“十四五”共用技术项目资助(0722-217FJ129YPF)。
作者简介：吴皓(1998-)，男，硕士研究生，研究方向为信息安全、漏洞检测；周世龙(1965-)，男，硕士，教授，研究方向为电路与系统、模拟集成电路设计；史东辉(1966-)，男，博士，教授，研究方向为数据挖掘、机器学习、人工智能、知识工程；李强(1962-)，通信作者，男，硕士，教授，研究方向指挥信息系统、信息安全、软件工程，E-mail：lychfeei@163.com。

符号执行是一种提高程序与软件质量的分析技术，使用符号化输入代替程序变量，执行中程序变量、语句和表达式会根据符号集模拟执行。符号执行具有探索不同的程序路径，为每个路径生成一组具体输入值以及检查程序中是否有错误，被广泛应用在软件测试和安全领域中。

自 70 年代开始，符号执行技术提出程序自动测试后便立刻引起了高度关注，不过短暂研究高潮后便陷入低谷，原因是当时计算能力有限无法遍历全部路径。随着现代计算机器件及工艺的提升，针对路径搜索策略、内存建模和非线性求解能力等进一步优化，缓解了符号执行路径爆炸和约束求解的问题，而重新获得了广泛关注[1-2]。实际项目中符号执行已有一些成功的测试套件，如 NASA(National Aeronautics and Space Administration)的 Symbolic Path Finder[3]，还有微软团队设计的符号执行引擎 SAGE[4]，该引擎宣称在 Windows7 系统中有接近 1/3 的漏洞数量都是通过该工具挖掘。目前来看，符号执行技术方向进展显著，尤其是安全漏洞利用生成和检测方面研究成果颇丰。

本文围绕符号执行技术及突出应用方面阐述，全文组织如下：第 1 节中从符号执行技术自身的概念、发展和系统核心设计研究切入介绍；第 2 节中从应用的角度讨论符号执行在安全漏洞上的研究，包含控制流劫持、堆栈和 Android 系统的漏洞利用，以及围绕漏洞检测中目标输入对象和漏洞类型的不同展开；第 3 节中，从技术特点的角度对符号执行研究工作整理分析并讨论所面临局限与解决方法；最后，总结全文并加以展望。

1 符号执行相关研究

1.1 执行技术与发展

符号执行[5-6]使用符号值作为输入值，根据程序语义利用符号集对程序变量、表达式和语句进行符号翻译，沿着程序路径执行，通过收集条件送往求解器进行求解，最后的输出值可表示为输入值的符号函数。在遇到状态分支时，相应的分叉探索每支路径状态，收集每支路径上的约束条件，通过约束求解器验证约束条件的可满足性问题。若路径约束是可满足的，则说明该路径是可达的，并生成测试用例，程序执行到该路径；若路径约束是不可满足的，则说明路径不可达，终止对该路径分析。

符号执行本质理解是一组输入值向前的符号表达式替换，算法 1 包含了赋值、条件 goto 语句和断言语句类型的符号分析。其中，函数Next用于选择下一个工作列表中的状态，follow 函数遵循分支的决定，程序的状态由三元组（PC，Π，Δ）决定， PC是下一条执行语句，Π表示路径约束，表示符号值在执行过程中由于到达PC的分支而产生的表达式； Δ是一种符号存储，包含具体值或符号值的表达式集合。

算法 1：符号执行

```
输入：具体值、符号值
数据：符号存储Δ、路径约束Π、初始位置 PC0、工作列表 W 、继承状态 S
1.  W ←{（ PC0 ,Π=true， Δ=∅）};
2.  S :=∅;
3. while  W  ≠∅  do
4.       （PC，Π，Δ）← Next（ W ）
5.      switch  typeAt( PC )  do
6.       case v:= e                  //赋值运算
7.          S ←{(succ( PC ),Π ,Δ [v→eval(Δ ， e)])}
8.       case if (e) goto  PC'      //条件语句
9.         if follow(Π ∧Δ ∧e) then
10.           S :={( PC' ,Π ∧e,Δ )};
11.        if follow(Π ∧Δ ∧¬e) then
12.           S :=  S ∪{(succ( PC ),Π ∧e,Δ )};
13.      case assert(e)              //断言判断
14.        if isSatisfiable(Π ∧Δ ∧¬e) then abort;
15.           else  S :={(succ( PC ),Π ,Δ )};
16. print ″no errors″
```

在第 1 行中，算法将W初始化为一个工作列表，在每次迭代中，算法从工作列表中选择一个新状态(第 4 行)，当遇到赋值 *v*:= *e* 时(第 6-7 行)将符号存储Δ从 *v* 映射到新符号表达式中同时更新符号存储Δ，并将新的状态添加进S中；条件 goto 语句分支符号分析(第 8-12 行)当程序遇到条件语句时，符号执行引擎将收集到的信息传递给求解器，以便在分支点产生所需结果作为程序输入；对于断言(第 13-15 行)将路径条件、符号存储和否定断言结合在一起并检查是否满足。

根据时间脉络来看，基于路径空间的符号执行技术发展出丰富的思想，大致分为经典符号执行、混合符号执行(Concolic Execution)、执行生成测试(Execution Generated Testing)、选择符号执行(Selective Symbolic Execution)、符号后向执行(Symbolic Backward Execution)和融合性符号执行。经典符号执行[7]于 1975 年提出，并不直接运行程序而是通过符号值遍历程序模拟执行；混合执行[8-9]于 2005 年提出，

采取具体值和输入的符号值混合执行；执行生成测试[10]于 2006 年被提出，也融合了具体值和符号值的执行，对程序执行中符号变量使用具体值和符号值替换，其中执行生成测试和混合执行是动态符号执行两种重要思想；选择符号执行[11-12]于 2009 年提出，可以针对局部区域进行选择符号值执行；同年，还有符号后向执行[13-14]从目标区域反向运行到程序入口探索不可达路径；最后，目前流行的融合性符号执行逐渐成为研究重点，典型代表 Driller[15]，通过符号执行和基于输入空间遍历的模糊测试融合使用，引导模糊测试随机变异，以提升准确率。

除此之外，符号执行还根据是否生成中间语言进行模拟执行，分为基于解释型符号执行和编译型符号执行两类[16]。基于解释型的符号执行系统根据目标程序转换为中间表示，然后对中间表示动态插桩送到解释器逐条执行；基于编译型的符号执行通过编译执行的方式避免了重复的插桩和翻译等步骤。

1.2 符号执行系统设计研究

符号执行技术自 1975 年提出后，经过几十年的发展，至今有着丰富的成果。通过现有文献梳理分析，给出符号执行系统框架，如图 1 所示主要包括前端、执行阶段、后端。前端主要针对目标输入对象(如源码和二进制文件等)处理与解析以完成后续生成中间语言等工作，后续符号执行操作中使用的符号也需要进行声明与定义；执行阶段由中间语言文件、内存建模、指令追踪、符号翻译和路径搜索等步骤构成；后端的功能主要针对状态分支中路径条件进行约束求解，通过收集程序执行路径分支上的约束条件传递给求解器，求解并生成测试用例。

不过现实中分析程序执行状态空间很大，因此构建符号执行系统会权衡运行时间、路径选择、内存消耗、性能、可靠性及完整性。另外，符号执行系统设计应遵循性能原则：进度控制，执行器应在不超过资源情况下进行任意时间执行；工作重复，避免多次启动程序分析具有共同的路径；分析重用，重复利用以前运行的分析结果。

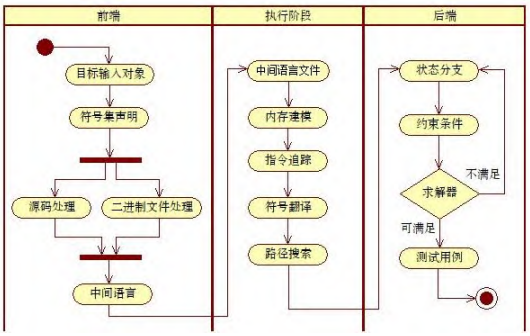

图 1 符号执行系统流程

Fig.1 Symbolic execution system flow

1.2.1 中间语言研究

符号执行引擎目标输入对象多数以二进制文件程序，然而二进制文件底层指令数量较多且缺少语法语义的类型信息导致难以分析。基于此，业界侧重于将输入分析对象转换成一些中间表示 IR(Intermediate Representation)。中间语言具有指令精简及语法语义丰富的优势，实际使用中常见使用编译器 Clang 和 GCC(GNU Compiler Collection)等生成中间语言。符号执行中备受青睐的中间语言有 LLVM[17-18]字节码、VEX[19]、Vine[20]、CIL[21]和 TCG[22-23]等。

为评估中间表示对符号执行影响，文献[24]于 2019 年选取流行且开源的符号执行工具进行验证，采用 CGC(Cyber Grand Challenge)程序和 coreutils 套件作为数据集，论证 IR 生成过程是如何影响符号执行各方面。研究重点放在中间表示从源码生成 IR 和从二进制提升到 IR 的符号执行各有什么好处，同程序生成的不同 IR 对符号执行有什么差异上面。实验表明，基于源码生成的 IR 通常比二进制生成的 IR 更有优势，在执行速度上无 IR 比有 IR 生成占有优势，不过也失去了平台移植性，表明 IR 语言的抽象程度会对符号执行引擎中有着差异。

中间语言在符号执行执行阶段，通过 IR 解释器循环执行程序指令，并根据符号输入跟踪语义，而文献[25]提供广泛的证据表明执行组件是现代符号执行技术的主要瓶颈，其中执行组件就是 IR 解释器。一般来说解释的效率低于编译的，因为执行时编译器只需工作一次。为改进执行组件的速度性能，文献[16]于 2020 年提出“要编译不是解释”的思想并实现针对源码的 symCC 系统，通过 LLVM 编译器生成 LLVM 字节码中间表示，插入用于符号处理的代码使符号执行阶段成为编译文件的一部分，

实现了符号执行中嵌入编译处理的能力，提高了程序执行的速度。此外，针对二进制文件编译优化上，文献[26]于 2021 年在 symCC 基础实现基于编译的二进制符号执行 SymQEMU 系统。该系统构建在 QEMU 上，将目标程序转换为 TCG 中间表示，使其符号执行阶段编译到二进制文件中，具有架构独立性的同时也获得相关性能优势，甚至比当前的无 IR 执行更快。

1.2.2 路径搜索策略研究

程序执行中路径与分支条件的数量成指数关系，增加分支的数量导致搜索空间会增大，进一步限制了符号执行的性能。因此，采取合适的搜索策略来提高符号执行的覆盖率，以及减少时间开销尤为重要。

本小节根据搜索风格以经典式、指导式和混合式搜索策略这三类回顾符号执行中应用的搜索策略。

经典式搜索策略使用执行树路径遍历算法，这些算法不依赖于先验知识所进行盲目地搜索，包括深度优先搜索 DFS(Depth First Search)、广度优先搜索 BFS (Breadth First Search)和随机搜索 RS(Random Search)。深度优先搜索是 2005 年最早在 DART[27]中实现，应用在动态符号执行的一种搜索算法，利用内存模型探索一条路径后把符号约束状态存储在内存中，使用求解器否定前面分支来生成一个新的路径约束，从而使程序执行走向一个不同的执行路径，提高求解的重用。不过 DFS 也有一些缺点，它增加了路径约束的长度，使得约束求解更加困难和耗时甚至会陷入循环和迭代的求值中[28]。其次，以广度优先顺序选择执行树中的分支，简称广度优先搜索策略[29-30]。这种策略倾向于探索不同的路径为重点，会使程序执行中频繁切换程序状态和符号状态，导致内存重用率低，路径状态爆炸更明显。总体上，DFS 和 BFS 搜索策略易于实现，并在许多符号执行工具[31–34]中实现，不过 DFS 倾向于最终分析而 BFS 倾向于初始程序中的分支。最后，随机搜索策略[35-36]从程序执行树的根节点出发随机选择一条路径继续执行，随机搜索重点放在路径选择、分支选择和输入选择，可利用它随机性来限制 DFS 和 BFS 的缺点，但稳定性较差，难以保证可靠的路径覆盖，应用该策略的符号执行工具有 KLEE[17]和 DART 等。

指导式搜索策略不同于以往盲目性搜索，而是获取上下文信息定义了路径的权重和启发式方法，引导搜索朝着目标前进。常见利用控制流和数据流中目标的距离[37-38]、路径长度[39]、测试目标的密度[40-41]等指标。其中，文献[42]提出 CarFast 启发式算法，基于控制流图 CFG (Control Flow Graph)的信息计算状态的数量，对分支状态信息排序决定先后执行的顺序。不过 CarFast 算法需要额外计算 CFG 信息并对分支排序，若遇到状态空间本身庞大时候会导致低效率。此外，文献[43]针对程序进行控制流静态分析指导符号执行来探索空间路径，以程序生成控制流图的每条边分别对跳转、分支和函数调用分别赋予不同的值，计算出可达路径的全部权重，优先选择权重最小的探索，这样可减少约束求解器的压力。

混合式搜索策略结合无指导式的路径覆盖率高和有指导式的开销小等优势，可以分为随机搜索导向[44-45]和面向蛮力搜索[46–49]两类。随机搜索导向将随机搜索和指导式启发方法结合一起来增强搜索，通过随机策略的廉价性降低了搜索的成本，但带来了不稳定性；面向蛮力搜索策略利用 DFS 和 BFS 的基础上使用测试对象信息引导搜索，适用于小型程序的穷举搜索，但难避免倾斜搜索区域。

1.2.3 约束求解研究

约束求解决定着符号执行的效率。约束求解是一种基于数学理论的分析方法，其问题可描述为一个三元组 $P=<V,D,C>$ 形式，V 表示变量集合 $V=\{v_1,v_2,v_3,...,v_n\}$；$D$ 表示中元素的值域集合，即变量的取值范围；C 表示约束条件。约束求解问题就是变量在值域范围内取值使得约束条件 C 可满足的，否则，这约束问题是不可满足的。

目前，约束求解问题主要依赖两种理论模型，SAT(Boolean Satisfiability Problem) 和 SMT(Satisfiability Modulo Theories)。SAT 理论基于布尔逻辑，检验一阶谓词公式的可满足性问题，是一个已经被证明的 NP-Complete 问题，不过 SAT 求解的变量类型只能以布尔类型和逻辑公式问题形式。于是在 SAT 基础上加入模块背景理论扩展出了 SMT，在求解范围上由布尔类型到一阶逻辑表达式的公式，不需要像 SAT 把问题转化为复杂的合取范式 CNF(Conjunctive Normal Form)，在一定程度上简化了求解问题。当前使用较广基于 SAT/SMT 理论模型的约束求解器有 Z3[50]、

CVC[51]和 STP[52]等。

约束求解在符号执行的运用，沿着程序一条路径，收集路径上所有的路径约束条件，通过约束求解器判定是否可满足并生成测试用例。若路径约束条件可满足，则说明路径是可达，程序可以运行到此处并生成测试用例；否则，路径则是不可达。

在符号执行中，约束求解以数学理论为基础，可以处理复杂的分支等约束条件，使得符号执行技术具有高可靠性。然而，约束求解问题也制约着符号执行的发展[53]，归结到底主要是求解能力与求解效率问题未提升。其中，求解能力表示求解器对非线性运算和浮点运算等复杂性约束条件的求解能力不足，导致符号执行处理数据类型有限制；求解效率表示遇到大量状态空间信息求解时所带来时间阻塞。

2 基于符号执行的安全漏洞研究

漏洞是信息技术产品在需求、设计、编码、测试、配置、运行维护等过程中所产生的缺陷，这些缺陷可以被攻击者控制并利用，进而破坏系统机密性、完整性和可用性等非预期危害[54]。对于漏洞数量的发展趋势，截止 2022 年 10 月底，国家信息安全漏洞库采集漏洞数量已达 195428，较 2021 年同比增加 24321 个，增速约为 14 百分点[55]。随着漏洞数量的快速增长，研究人员通过各种技术方法在漏洞的发现、分析以及利用中提出一系列解决安全漏洞的思路，而符号执行便是一种常用的方法。

本节进一步讨论关于符号执行在安全漏洞方面的利用与检测应用。

2.1 基于符号执行的漏洞利用研究

漏洞利用是以某种目的针对特定漏洞生成数据和代码的过程，期望获得系统的最高权限[56]，可导致数据篡改、信息泄露甚至任意代码执行等危害，漏洞的可利用性是区别于其他程序错误的明显特征。

近年来，漏洞利用研究主要包括：利用方法、自动利用生成、安全防护机制绕过和内存布局等。本文根据符号执行方法、利用数据和漏洞分类的差异，从控制流劫持漏洞、堆栈漏洞和 Android 系统漏洞等这三个方面，介绍基于符号执行的漏洞利用研究进展，从而分析漏洞利用发展趋势。

2.1.1 控制流劫持漏洞利用研究

控制流劫持是从漏洞触发开始，驱动被劫持的程序运行到可输入控制的一种状态，这些可输入控制的状态包括基于代码注入类攻击和基于代码重用类攻击的指令代码输入所控制。控制流劫持是属于任意代码执行漏洞的典型代表，大多数是高危漏洞类型，针对控制流劫持漏洞利用已有多年研究，相关成果如表 1 所示。

漏洞利用方面的自动利用生成最早出现针对内存溢出类漏洞方面，通过二进制补丁比较技术发现漏洞并生成利用代码。此后，文献[57]利用符号执行提出基于源码的自动利用生成方案 AEG。通过对源码进行编译预处理，利用符号执行提取文件中漏洞触发约束、环境约束、内存变量等属性，接着调用约束求解得到漏洞 PoC (Proof of Concept)。最后，将 PoC 通过程序插桩建立内存攻击代码条件，使用混合执行生成利用了代码。不过在试验中，所针对漏洞触发点生成利用代码只支持栈溢出和格式化字符串漏洞类型。

此外，文献[58]在 AEG 的基础上扩展出二进制程序的漏洞利用生成 Mayhem，可面向控制流劫持漏洞进行利用生成，该方法结合了符号执行在线和离线模式的优点形成混合符号执行方法，在符号翻译过程中构建约束条件，求解后生成利用代码。符号执行期间，还引入了索引内存建模技术，缓解了执行过程中路径爆炸问题，提升了符号执行效率。不过该系统未建模时会出现调用错误，漏洞范围仅支持栈溢出和格式化字符串漏洞利用。

除了常见源码和二进制文件的利用数据外，文献[59]提出对软件崩溃信息利用分析的自动利用生成方案 CRAX，构建在符号执行 S2E 平台，实现了控制流劫持漏洞的利用攻击生成。在执行过程中可回溯到程序崩溃的路径上进行混合符号执行并探索漏洞路径，过滤复杂和不可达的路径，监控指令寄存器和符号化变量内存，最后通过约束求解获得利用生成代码。具有在真实环境下对控制流劫持多种漏洞利用生成的优势，不过对于异常输入详细情况会决定着利用生成质量。

表 1 基于符号执行的控制流劫持漏洞利用生成研究

Table 1 A study on the generation of control flow hijacking vulnerability exploitation based on symbolic execution

分类	研究成果	研究问题	主要贡献	不足之处

文献[57]	AEG	针对源码的漏洞自动利用生成研究	首次提出针对源码的自动生成利用，实现了预处理符号和路径优先算法	仅针对漏洞触发点生成利用代码，局限在栈溢出和格式化字符串漏洞的利用生成
文献[58]	Mayhem	针对二进制程序中可利用漏洞研究	提出一种可扩展的二进制漏洞挖掘与利用生成框架，实现了混合符号执行	利用范围仅包括格式化字符串漏洞等，不能处理未建模的系统调用
文献[59]	CRAX	针对软件崩溃漏洞利用研究	提出一种软件崩溃的漏洞利用生成	不具备漏洞挖掘性能，异常输入的详细情况决定着利用生成质量
文献[60]	HeapHopper	针对堆元数据的漏洞自动利用生成研究	提出一种符号执行与模型检测面向堆漏洞的利用生成	仅针对堆漏洞实现利用生成，尚未突破安全防护机制
文献[61]	R2dlAEG	针对安全防护绕过机制研究	提出一种符号执行结合 Return-to-dl-resolve 的安全防护绕过机制	仅针对数据执行保护和地址空间布局随机化保护，突破安全保护机制不全面

针对内存布局的自动利用生成，文献[60]提出基于符号执行和模型检测的堆内存分配器的 HeapHopper，突出思想通过堆元数据破坏利用，可造成堆内存成控制流劫持、任意地址写入和任意数据的利用逻辑等，是第一次尝试将有限状态用于动态内存的漏洞可利用。

考虑安全防护机制保护情况，文献[61]提出基于符号执行与 Returned-to-dl-resolve 绕过安全防护机制实现漏洞利用生成的 R2dlAEG，该方法对 Returned-to-dl-resolve 提取漏洞利用要素进而对 ELF 文件提供符号执行环境，利用设计的崩溃状态引导程序执行，收集程序崩溃时运行的信息，用来构建约束条件并生成利用代码。利用 angr 和 QEMU 的基础上实现，成功突破系统安全防护机制数据执行保护和地址空间布局随机化 ASLR(Address Space Layout Randomization)开启并劫持程序。

对于控制流劫持的自动构造问题，研究人员也会利用符号执行和其他方法混合共同实现漏洞利用生成，如文献[62]针对由可控内存写漏洞到控制流劫持无法自动构造利用问题，采取动态污点分析检测可控内存写漏洞，再基于利用要素搜索使用约束求解器生成能触发控制流劫持的 POC。

总体上看，基于符号执行的控制流劫持漏洞利用的思路，以生成测试用例作为输入数据，建立控制流劫持点，关联可控制输入状态与执行路径的信息，通过程序插桩、监控指令寄存器以及崩溃信息等基础上建立程序路径的约束条件，最后调用求解器生成利用代码。存在问题是多数研究成果的前提以系统中安全机制关闭下进行，尤其是当前的 Stack Guard、ASLR、DEP(Data Execution Prevention)等安全机制可以对抗一定的程度攻击利用，如何考虑在安全保护机制开启下对多种类型漏洞的利用是需要深入研究的问题。

2.1.2 堆栈漏洞利用研究

堆栈是一种在计算机科学中经常用到的抽象数据类型，拥有先进后出的特点。堆栈漏洞也是一种常被攻击利用的类型，包括堆栈溢出、释放后重引用(Use-After-Free，UAF)、格式化字符串等。堆栈漏洞攻击利用的关键点是利用指向堆结构的垂悬指针来实现任意地址读取、写入和 shell 返回等。

近年来，研究人员针对堆栈漏洞使用符号执行技术进行分析与利用取得了相关成果，如表 2 所示。

针对二进制文件缓冲区溢出漏洞，文献[63]提出基于符号执行二进制文件的堆栈缓冲区溢出漏洞自动利用生成，通过处理输入数据的指令，使用切片算法定位到接收输入数据和异常崩溃点的信息来构造路径谓词，可以生成指定的 shell 代码攻击。不足的地方是实验中主动关闭了系统保护，尚未突破堆栈的保护。

还有面向 PoC 文件，文献[64]针对堆漏洞提出轻量模块化符号执行的利用生成 Revery 方法，与前期研究将概念性证明(PoC)输入转换为利用生成并搜索可利用的状态不同。核心思想是提出一种新的控制流拼接，将多种崩溃和正常的路径缝合在一起，最后用一个轻量级的符号执行产生漏洞利用代码。

针对 Linux 内核中的 UAF 漏洞，文献[65]针对 UAF 漏洞开发了自动利用生成平台 FUZE，利用符号执行和模糊测试来识别、分析和评估 UAF 漏洞并生成利用代码，主要思想是利用模糊测试输入异常崩溃的测试代码，以上下文的崩溃处生成约束条件并求解得到利用代码。

表 2 基于符号执行的堆栈漏洞利用生成研究

Table 2 A study on the generation of stack exploits based on symbolic execution

分类	研究成果	研究问题	主要贡献	不足之处

文献[63]	AEGSBOV	针对二进制堆栈缓冲区漏洞利用生成研究	提出一种基于符号执行的评估异常错误方法	没有克服堆栈保护,仅针对堆栈缓冲区溢出漏洞
文献[64]	Revery	针对堆漏洞的利用生成研究	提出一种 PoC 输入转换为漏洞利用生成方法，并引入轻量级符号执行	多种路径拼接,带来路径状态爆炸
文献[65]	FUZE	针对 UAF 漏洞自动利用生成研究	提出一种符号执行与 Linux 内核融合的 UAF 利用生成	仅针对释放后重引用漏洞,
文献[66]	Pangr	针对二进制程序中堆栈溢出漏洞	提出一种基于符号执行的行为漏洞建模	未考虑系统保护状态
文献[67]	FSAEG	针对格式化字符串漏洞利用生成研究	提出一种基于符号执行的格式化字符串漏洞利用生成	多种符号执行引擎,效率低
文献[68]	AngErza	针对缓冲区漏利用生成研究	提出一种动态符号执行识别漏洞点的方法	仅针对缓冲区溢出和格式化字符串的漏洞

针对二进制文件堆溢出漏洞，文献[66]针对二进制文件中存在的堆溢出漏洞提出 Pangr 利用生成框架。该框架在 angr 基础上建立堆溢出漏洞检测模型，可检测和利用格式化字符串、堆栈溢出和堆溢出漏洞，并生成相应的保护方法。存在问题是仅支持 32 位的 X86 的二进制程序，并在测试验证中未考虑系统保护状态，主动关闭堆栈保护和地址空间布局随机化。

针对格式化字符串漏洞，文献[67]提出基于符号执行对格式化字符串漏洞实现自动利用生成 FSAEG。该方案针对参数存储位置处于栈空间外所带来可利用性误判高的问题，提出参数存储分散在不同空间下构建符号约束并求解，实现格式化字符串漏洞的利用。不过测试验证中使用多种符号引擎 S2E、KLEE 和 QEMU 系统等实现,所产生的测试用例效率低。

除此之外，文献[68]提出面向二进制文件缓冲区溢出漏洞的利用生成框架 AngErza，并在 angr 利用生成框架基础上实现。具体思路上，使用动态符号执行来识别二进制文件中的缓冲区溢出和格式化字符串的漏洞，构建约束条件后调用 angr 的符号执行和约束求解等模块生成利用代码。

基于符号执行的堆栈漏洞研究，在利用方法、模型建立以及自动利用生成等取得了一些成果。通过采用程序切片方法、轻量化模块化符号执行、多种符号执行引擎融合以及综合使用污点分析、模糊测试等技术扩大符号执行搜索的路径；通过堆栈溢出、格式化字符串等漏洞数据的可控与可写入的特征属性，建立漏洞利用生成模型；通过约束求解生成测试用例作为堆栈漏洞利用代码等。堆栈漏洞可利用主要以破坏堆栈数据后，进一步分析数据是否可输入可控制成为利用的关键，与控制流劫持漏洞相比，堆漏洞的利用过程会复杂些，主要的原因是堆数据区域的动态性会导致多样性。

2.1.3 Android 漏洞利用研究

智能手机的流行促使 Android 系统逐渐成为市场占有率最高之一，随之而来 Android 系统的漏洞也逐年也增多。

针对 Android 的应用程序利用生成，文献[69]是第一次提出针对 Android 应用程序组件通信漏洞利用的方案 LetterBomb，研究的重点是 Android 应用程序组件间通信接口。使用符号执行用来分析 Android 系统应用程序中可能的漏洞，利用反向静态执行确定具体的漏洞类型并生成漏洞可利用性代码。

面向 Android 系统的利用生成框架研究，文献[70]设计并构建了第一个支持 Android 系统符号执行分析框架 CENTAUR，为避免初始化导致的状态空间爆炸而提出阶段性符号执行方法，CENTAUR 实现与 Android 解耦的优势，具体执行时提供的执行上下文从 Android 进程迁移到 Java VM。

为研究 Android 应用程序远程攻击利用问题，文献[71]针对 Android 中嵌入式浏览器远程攻击利研究，设计了 EOEDroid 方案，技术上选择符号执行和静态分析审查应用中漏洞并生成利用代码。使用选择符号执行探索所有可疑路径并收集路径约束，根据求解的测试用例引导静态分析状态并转换为利用代码。不足之处探索部分 UI 组件以及在隐式流问题中只关注数组索引类型的操作。

为研究 Android 应用程序问题，文献[72]基于路径敏感符号执行与静态分析针对 Android 应用程序的能力漏洞，设计并实现了 AGCLEAA 方案。主要思想是遍历 Android 和 API 接口之间的所有可能的路径，利用控制流图和函数调用图优化符号执行搜索过程，提取路径约束，将约束条件使用 Z3 约束求解器进行求解并利用。

表 3　基于符号执行的 Android 系统漏洞利用生成研究

Table 3 A study of symbolic execution-based vulnerability generation for Android systems

分类	研究成果	研究问题	主要贡献	不足之处
文献[69]	LetterBomb	针对 Android 应用程序的组件通信漏洞利用生成研究	提出一种依赖路径敏感的符号执行静态分析	仅对组件间通信接口可疑漏洞检测并利用
文献[70]	CENTAUR	针对 Android 应用程序漏洞分析与利用生成研究	提出一种上下文感知的信息构建符号执行系统，面向 Android 程序框架漏洞利用生成	执行上下文中的变量会为符号执行带来状态爆炸
文献[71]	EOEDroid	针对 Android 应用程序中嵌入式浏览器的漏洞利用生成研究	提出一种选择符号执行和静态分析混合处理程序	仅简单触发部分漏洞，在隐式流问题只关注数组索引类型的操作
文献[72]	AGCLEAA	针对 Android 应用程序的功能泄露利用生成研究	提出一种路径敏感静态分析符号执行工具，采取控制流图和函数调用图优化符号搜索过程	检测程序的功能泄露不具有大型代码量

表 3 列出符号执行运用在 Android 漏洞的利用生成所取得的成果。通过以上分析可以得出，目前符号执行对 Android 系统漏洞利用研究集中在 Android 应用程序的功能和组件通信，综合使用符号执行、控制流图和函数调用图等技术生成利用代码。不过，面向 Android 应用框架的利用生成研究较少，应用程序框架是 Android 系统不可或缺的基础部分，智能设备依赖于 Android 应用框架的系统服务来管理应用程序和系统资源，通过利用漏洞可对用户的安全和隐私造成严重的伤害。此外，Android 系统使用了 Linux 的内核，故对 Android 系统漏洞利用研究可参考 Linux 内核漏洞的利用研究。

2.2 基于符号执行的漏洞检测研究

基于符号执行的漏洞检测过程如图 2 所示，通过源码和二进制文件等目标输入对象进行预处理，根据处理得到中间语言，利用中间语言文件使用符号执行思想对程序中变量、表达式及语句翻译成符号化表达式，依据漏洞检测策略将路径约束和安全约束特征合取生成一阶逻辑公式。最后，通过约束求解对一阶逻辑公式寻求可满足解。漏洞判定需要判断路径约束同漏洞策略合取之后一阶逻辑公式是否存在可满足解。如果存在可满足解，说明当前路径中存在漏洞，否则当前路径中不存在漏洞。

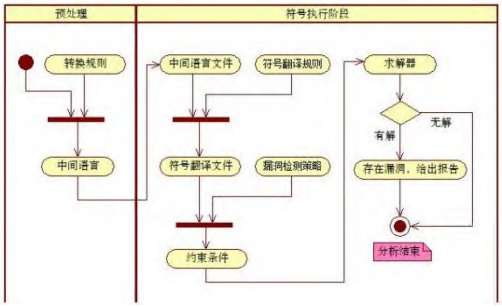

图 2 基于符号执行的漏洞检测过程

Fig.2 Symbolic execution based vulnerability detection process

基于符号执行的漏洞检测主要集中在基于源码和二进制文件的目标输入对象以及针对特定目标漏洞类型方面。

2.2.1 目标输入对象不同

对于符号执行检测输入对象的不同，检测技术主要分为源码和二进制文件检测。在实际检测过程中需将目标输入程序转换成中间语言，以便后续符号执行，不过由源码和二进制文件生成的 IR 语言抽象程度直接影响检测的结果。此外，其他对象输入的符号执行检测会复杂些，需进行汇编、反汇编和提升至二进制文件等操作，以便从输入目标对象中提取信息。

(1) 源码

源码是按照程序设计语言规范所书写的指令，通过编译器翻译成目标程序，常见有 C/C++、Java 和 Python 等高级语言所编写。源码中包含着函数、变量和类型等抽象信息，便于获取漏洞检测所需相关信息。

针对 C/C++源码，文献[73]第一次提出将符号执行技术用于 C 语言源码的漏洞检测。该方法集成了符号执行、模型检测和随机搜索技术，核心思想是执行过程中每生成输入向量沿不同执行路径，以达到最大路径覆盖率。但是在检测的过程中，C 语言所包含的一些复杂数据结构无法分析，且约束求解器使用线性求解器，带来求解精度低，极大地限制了测试用例的生成。因此，为改善 C/C++源码检测性能问题，得到了研究人员广泛关注。如文献[74]在 DART 的基础上扩展提出了 CUTE，可以对指针操作和复杂数据结构进行多线程处理，核心是近似表示指针运算约束表达式来回避指针分析的非准确性。同样，还有针对符号执行技术 EXE[75]的改进，文献[76]提出基于 EXE 的基础上设计出 KLEE，将 C/C++源码编译为 LLVM 字节码提升性能，突出点使用了覆盖率优化搜索和随机路径选择策略来应对路径爆炸问题，可以生成高覆盖率的测试集。相对于 EXE 主要有两方面改进：存储更大的并发状态

和处理程序与外界环境交互，而面对符号变量声明、加密函数和循环时处理有些棘手。

针对 Java 代码，文献[77]将符号执行与模型检测相结合，对程序 Java 字节码进行错误检测。通过模型检测来探索符号执行的数据重叠，具有处理字符串、复杂数据结构和调用等复杂约束。此外，文献[78]还实现对 Java 程序中复杂的堆输 入进行探测，提出一种新的堆探测逻辑以及先进的处理数据结构的方法。最后，还有在 DART 基础上加入多线程来处理 Java 程序的 JCUTE[79]和 JDART[80]。

针对 Python 代码，文献[81]使用 Python 语言编写的漏洞检测 Conpy 方案，旨在用选择符号执行思想对 Python 检测，直接在 Python 代码库中作为模块运行，不足之处是只能维持在小数量运行且不能对用户定义的变量进行检测。

除了常见的高级语言形式外，还有着第一个使用符号执行技术分析 JavaScript 的 Kudzu[82]；针对 Ruby 语言符号执行分析的 RubyX[83]；针对.NET 框架微软基于符号执行开发的 Pex[84]等成果。

不难发现，基于符号执行的源码检测通过利用源码中函数、变量、语句等语法语义信息便于符号执行。不过直接面向高级程序设计语言检测是需要一定数量代码维持，实用性、生态性方面受到限制。

(2) 二进制文件

二进制文件是源码经过预处理、编译、汇编和链接等形成的二进制序列文件，检测过程中分析二进制代码，具有准确性高，实用性广。因缺少语义与语法的类型信息，实际检测中通过提升器翻译为中间语言，以获得需要的信息。

面向 X86 二进制程序漏洞，SAGE[85]是第一个将动态符号执行技术运用到二进制程序分析上，采用离线跟踪指令进行约束生成，采用全新的导向性搜索方法。其覆盖程序路径最大化并生成测试用例，通过 Z3 求解后并对约束路径再取反得到新的测试输入，使得测试用例数量生成的效率大幅提高。

此外，针对二进制文件的数组越界、空指针引用错误以及函数参数错误的具体漏洞类型，文献[86]设计并实现了漏洞检测 SEVE 平台，并在 MP3 和 PDF 软件中得到验证。其采用 C++的 STL 数据结构分别对内存和寄存器字节构建 map 结构，建立变量映射关系。将程序输入符号化，利用动态插桩获得汇编语义信息同符号变量传播信息用来更新映射，收集约束条件调用求解器进行检测。

结合漏洞检测策略方面，文献[87]为提升二进制漏洞检测性能，该方法结合符号执行和漏洞匹配模式，将检测的过程分为两级：第一级限制可疑区域使用行为代数表达式快速遍历代码；第二层级使用符号执行对给定的行为执行以减少时间。实验表明在检测时间上有着高效率提升。

还有面向网络协议二进制文件，文献[88]给出了选择符号执行对网络协议的服务器端和客户端同步的测试方法，可减少符号执行可探测路径的数量，利用网络协议的交互信息有效检测漏洞，不过目前仅针对已知溢出漏洞具有成效。

由于二进制文件底层指令数量较多且缺少语法语义的类型信息，现阶段基于符号执行的二进制文件漏洞检测是将二进制程序翻译成中间语言进行分析。在符号执行过程中通过导向性搜索策略、内存映射符号变量、漏洞匹配模式等措施优化提升了检测效率。

随着业界对二进制文件分析需求的增长，面向动态符号执行结合二进制文件分析会变得越来越重要。考虑进一步实用性上，研究人员所提出检测方案多是原型系统，没有有机的结合，不能互为弥补，系统性、重用性、协同性不够，将程序分析技术集成到一个平台形成全面综合的二进制漏洞检测框架是一种趋势。

2.2.2 检测漏洞的类型不同

漏洞检测与具体的漏洞类型有着紧密联系。根据 CVE(Common Vulnerability And Exposures)收录的漏洞类型统计[89]，内存溢出类和 Web 注入类漏洞这两种漏洞数量在漏洞库中排名位列前几，分别占据 12.2%和 20.5%份额。内存溢出类也是安全漏洞研究的开端，是一种常见且高危害的漏洞类型，研究人员持续关注内存类漏洞研究。同时，随着 Web 技术应用和发展，满足用户与 Web 应用的实时性、响应性和多元化等需求所出现动态交互页面，大大增加了 Web 注入漏洞的产生。此外，其他漏洞数量份额较小的类型同样也具有研究意义。

(1) 溢出类漏洞

溢出类漏洞是程序执行过程中缓冲区溢出的漏洞，主要原因是程序获取数据时未检查目标缓冲区的长度大小，直接向目标缓冲区写入过多数据所导致溢出，常见有堆栈溢出和格式化字符串等。通常，溢出类攻击会破坏堆栈数据实施注入、跳转和执行攻击代码以便获得系统控制权限甚至程序崩溃。

对于堆而言，堆分配大小可控设置可以对外界写入的数据进行合理分配内存，不过攻击

者可通过构造非法关键参数进行输入控制。针对该问题，文献[90]提出堆分配大小可控安全的方案，提出一种静态路径分析结合导向性符号执行的检测。方案在 KLEE 引擎上实现原型系统 SCAD，通过对 LLVM 字节码进行静态路径扫描，并使用相应的权值计算函数和基本块信息，以权值大小优先选择路径进行符号执行来判定堆分配大小可控的漏洞，尤其对整型溢出漏洞检测效果良好，不过没有对函数指针作出分析，以致路径分析不完整。

对于易被破坏堆栈数据来说，在栈数据中 EBP(Extended Base Pointer)、RETADDR(下一条执行的指令的内存地址)和 LOVCAR(函数中本地变量)等容易被利用破坏的对象。同时，面向堆数据也有着多种攻击方法造成破坏，如 fastbin 和 Unlink 攻击，针对堆溢出的攻击检测是近些年研究热点。

面向堆溢出的 fastbin 攻击，文献[91]提出基于符号执行的 fastbin 攻击检测的方法。其中，fastbin 攻击是通过修改堆块头从而达到任意代码的读写操作。该方案从堆块溢出、溢出可控、fastbin 触发和指针数据可控特征建立 fastbin 攻击模型，运用污点分析技术监控符号数据抵达漏洞触发点的路径信息并构建约束条件，最后通过约束求解得到测试用例。另外面向堆溢出的 Unlink 攻击，文献[92]提出一种基于符号执行的 Unlink 攻击检测的方法。其中，Unlink 是 Linux 平台面向堆溢出的一种攻击方式，篡改已经释放堆块的链表指针，造成程序指针被覆盖已达到控制流劫持的危害。不过都未考虑系统保护机制的影响，仅单一性检测堆溢出的 fastbin 攻击和 Unlink 攻击，后续针对堆溢出的多方面攻击检测是研究重点。

除了直接检测内存数据和内存攻击外，文献[66]还提出基于堆栈溢出漏洞和格式化字符串漏洞的行为建模也是一种尝试，利用符号执行寻找内存漏洞。

(2) Web 注入类漏洞

Web 注入类漏洞是 Web 应用中常见的一种漏洞，漏洞产生主要原因是缺乏对用户输入内容进行验证而出现恶意构造代码注入问题，典型漏洞有 SQL(Structure Query Language)注入和跨站脚本攻击 XSS(Cross Site Scripting)。

SQL 注入攻击是指攻击者使用 Web 应用程序恶意执行插入到数据库操作语句中，以期望控制数据库和操作系统等目标。常见的 SQL 注入漏洞形式[93]：GET 注入型—构造恶意代码插入到 URL 链接参数中；POST 注入型—利用用户与数据库交互的地方注入恶意代码，如利用搜索框和留言对话框等输入控件。早期针对 SQL 注入方面，文献[94]利用符号执行静态检测 Web 应用程序中 SQL 注入漏洞问题，设计并实现了静态分析框架 SAFELI。接着，在 SAFELI 基础上，文献[95]提出针对 Java 网络应用程序的字节码进行检测，利用符号执行静态检测 SQL 注入漏洞，通过在应用程序中提交 SQL 查询的位置构建方程并使用混合字符串求解器得到测试用例。此外，文献[96]面向 Web 应用程序中服务器端的 HTTP 参数修改的问题，利用符号执行生成客户端输入表单的约束，并使用约束求解器产生测试用例，不过在测试中动态符号执行技术对接口的覆盖率不能保证。

另外，跨站脚本攻击(XSS)也是未对用户输入进行转义处理或过滤所导致攻击利用。跨站脚本攻击分为这几种典型的攻击形式[97]，即反射型跨、存储型、DOM(Document Object Model)型和 Flash 型的这四类的跨站脚本攻击。面向 XSS 漏洞检测，文献[98]利用符号执行技术检测 Web 应用程序中反射型和存储型的 XSS 漏洞。虽然目前有很多检测 XSS 漏洞的方法，但是在 Web 应用中检测出所有的 XSS 漏洞仍然是困难的[99]。

对于 Web 注入类漏洞的认识，往往以注入的路径、防御和触发这 3 个过程进行。注入路径指的是污染注入数据的流动过程；防御指 Web 应用中防御验证功能；触发是指恶意构造的代码突破防御功能所触发非预期功能。基于上述认识，文献[100]提出了基于符号执行的注入类漏洞分析技术，旨在利用符号化输入提取路径上的防御约束、安全约束和攻击约束建立注入类漏洞检测模型。

综上所述，基于符号执行的漏洞检测研究围绕特定漏洞实施检测技术，实现了堆栈溢出、格式化字符串等内存溢出类，以及 SQL 注入攻击、跨站脚本攻击 XSS 等 Web 注入类的漏洞检测。除此之外，仍有众多的漏洞类型值得研究，尤其是新兴场景应用下的漏洞，如智能合约字节码中漏洞[101]、物联网固件设备中漏洞[102]和网络协议中漏洞[103]等。

2.3 小结

基于符号执行的安全漏洞的检测、分析与利用生成是一系列复杂的过程。现阶段针对特定的目标漏洞进行检测、分析与利用生成是一个趋势，包括针对控制流劫持、堆栈、Android 系统、内存溢出类、Web 注入类、物联网中固件设备、智能合约等。基于这些特定的目标可获取特定的特征属性，方便建立漏洞模型来辅助漏洞检测、分析和利用的快速研究。

第 2 节介绍了符号执行在漏洞利用与检测研究。在漏洞利用方面，以控制流劫持漏洞、堆栈漏洞和 Android 漏洞的利用发展较成熟，

漏洞利用研究主要包括利用方法、自动利用生成、安全防护机制绕过和内存布局等。在漏洞检测方面，介绍了源码和二进制作为检测输入对象的特点，并对常见堆栈漏洞、SQL 注入攻击、跨站脚本攻击 XSS 等其它漏洞的检测方式总结。虽然基于源码和二进制的漏洞种类有所相似，但研究成果中原型系统采用的符号执行思想和技术细节有所不同。随着符号执行的发展，构建一种精细、有效和实用的基准测试用来评估符号执行系统性能是有必要的。

其次，随着新兴技术应用和攻击利用手段的进步，安全漏洞仍具有进一步研究的空间，其中涉及到多方面的知识与技术，为解决分析技术的扩展和重用，将相关先进有代表性分析方法和技术(如符号执行、模糊测试、污点分析、模型检测等)集成在一个平台上是主流的研究方向。代表性有 angr[19]，angr 是加州大学圣塔芭芭拉分校研究团队提出面向二进制文件漏洞检测、分析和利用生成等功能的符号执行分析框架平台。该框架具有：多种分析技术，可以具备符号执行、控制流图分析、污点分析、静态分析、值域分析和数据依赖分析等多种分析技术；多平台多架构，angr 的指令插桩由 QEMU 实现，采用 VEX 中间语言，一定程度避免不同架构指令对程序分析的影响；可扩展，采用 python 开发和插件设计并对已有的分析技术模块化实现，可以根据使用者需求有效组合分析。但 angr 目前只突出符号执行提供强力的分析，其它相关技术只是作为辅助，未能提供有效的技术优势互补，后面研究应构建一个运用全面技术的安全漏洞分析框架为重点。

3 分析与讨论

3.1 对比与分析

鉴于符号执行的实际研究方向在漏洞利用生成与检测方面广以及评估方式不同，实验部分结果无法全面说明优劣。因此，以符号执行技术特点切入对各项现有研究工作进行对比分析，整理分析工作中研究的方法和问题，为后续研究工作提供参考。

从上述符号执行相关研究与安全漏洞研究工作中挑选了具有代表性的研究。表 4 中第 1 列为研究成果文献，第 2 列为研究对象的代码形式，包括源代码和二进制文件等；第 3 列为研究成果实现的原型系统，第 4 列为该项研究采用的中间语言形式，主要包括 LLVM 字节码、VEX、Vine、CIL、TCG 和 BAP 等；第 5 列为研究主要采用的分析方法，有符号执行、模型检验、程序插桩、污点分析和模糊测试等；第 6 列为该项研究针对具体问题为出发点的考虑；第 7 列给出相关链接，方便研究者了解和获取；最后，研究成果不局限于以下表中，表中以选取代表性成果进行分析。

通过研究发现：

(1) 对于输入对象而言，符号执行技术研究集中在二进制文件和源码的分析，其他对象以翻译成中间语言表示，在中间代码的基础上进行符号计算。在众多的输入对象中，对二进制文件分析更加实用，因此，在分析对象中占比最大。而源码中以主流的 C/C++、Java、python 和 JavaScript 等编程语言为主，少量研究涉及到补丁信息[104]、PoC[105]和软件崩溃报告[106]等也取得不错的成果。

(2) 对于研究的方法而言，符号执行具有高覆盖率测试用例生成的优势，是安全漏洞领域常用的方法之一。此外，现有的程序分析技术种类众多，还有模糊测试[107]、污点分析[108]、程序切片[109]和机器学习[110]等分析技术应用在安全漏洞领域，如何有效的结合多种技术进行改进也是当前研究的重点。

(3)对于符号执行的漏洞利用生成研究问题，针对控制流劫持、堆栈和 Android 系统漏洞较成熟。不过在现实环境中面对资源有限，提升漏洞分析和利用在未来是一个趋势[111]，尤其是面对新兴场景下，如区块链智能合约、物联网的固件设备和 Linux 内核等有待提升。而且，利用生成的过程涉及到多种方法技术，目前工作大多数以实现特定漏洞开发原型系统，系统自动化和智能化程度不高，针对多种漏洞综合利用生成的可扩展和高效性成为主流。

(4) 对于符号执行的漏洞检测研究问题，以常见的内存溢出类、Web 注入类研究工作开展较多，少数研究可以实现多种漏洞类型检测。在检测的过程中常侧重于将分析对象转换成一些中间表示，不过中间表示的抽象程度会对符号执行有着影响。另外，其他漏洞类型不常见的同样也具有研究意义。

(5)对于符号执行技术而言，在程序分析和软件测试方面有着不小的成果，研究学者也将大部分精力集中在有效性(提高速度)和重要性(提高代码覆盖率)。然而，调研发现关于符号执行技术的正式论证较少[112]，一般性理论是缺少的，多数用于符号执行的工具缺乏明确的规范和基准测试。近些年研究人员渐渐重视符

号理论的逻辑论证，文献[113]于 2021 年首次提出根据符号转换流程对符号执行进行形式化解释，公式推理了具体程序状态和路径条件的符号表示与生成，提出一个通用形式化框架来证明面向对象语言符号执行的可靠性和完整性，不足之处是解释符号执行技术片面，没有进一步扩展到具体执行和符号执行的混合。

通过文献研究发现，基于符号执行技术逐渐成为安全漏洞领域的研究重点。研究人员先后根据漏洞的代码形式、类型和原理实现了符号执行系统，取得了一系列突破性的成果：

(1) 目标分析对象多元化：从常见的源码和二进制文件利用分析外，还有补丁信息、PoC 和软件崩溃报告等，目标分析对象形式多元化扩展了符号执行技术的广度。

(2) 技术迭代更加有效：从经典符号执行到现代符号执行技术发展出丰富的成果，现代符号执行技术关键要点是混合了具体值与符号值的执行，尤其是当前基于编译思想的符号执行的性能可以提高几个数量级。

(3) 适用场景具有普遍性：从安全漏洞的发现、分析以及利用生成出发，应用到智能合约、物联网硬件安全、最坏情况执行时间分析、网络协议逆向工程和性能测试等多数任务场景中，所具有高覆盖率的测试用例生成和发现深层次错误的能力。

表 4 研究工作小结

Table 4 Summary of research work

文献	分析对象	原型系统	中间语言	方法	研究问题	链接
文献[3]	Java	Symbolic Path Finder	——	符号执行 模型校验	针对 Java 程序的自动生成测试用例问题	https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
文献[80]	Java	JDART	——	符号执行 模糊测试	针对 Java 程序的符号执行框架	https://github.com/psyco-paths/jdart
文献[79]	Java	JCUTE	——	符号执行 程序插桩	针对 Java 语言的漏洞研究	https://github.com/osl/jcute
文献[74]	C	CUTE	——	符号执行 程序插桩	针对 C 语言的单元测试	https://github.com/osl/jcute
文献[27]	C/C++	DART	——	符号执行 程序插桩	针对 C/C++语言的自动测试	——
文献[10]	C/C++	EXE	CIL	符号执行	针对 C/C++语言的漏洞研究	——
文献[91]	C/C++	fastbin 检测	——	符号执行 污点分析	针对堆溢出 fastbin 攻击检测问题	——
文献[92]	C/C++	Unlink 检测	TCG LLVM	符号执行 污点分析	针对堆溢出的 Unlink 攻击检测问题	——
文献[17]	LLVM	KLEE	LLVM	符号执行	针对 C/C 语言的符号执行框架	http://klee.github.io/
文献[83]	Ruby	RubyX	Ruby	符号执行	针对 Ruby 语言的漏洞研究	http://www.cs.umd.edu/avik/papers/ssarorwa.pdf
文献[84]	.NET	Pex	——	符号执行 模型校验	针对.NEX 漏洞研究	http://research.microsoft.com/en-us/ projects/pex/
文献[82]	JavaScript	Kudzu	——	符号执行	针对 JavaScript 语言漏洞研究	——
文献[20]	Vine	BitBlaze	Vine	符号执行 污点分析等	综合统一的二进制漏洞分析平台	http://bitblaze.cs.berkeley.edu
文献[67]	Binaries	FSAEG	TCG LLVM	符号执行	针对格式化字符串漏洞的自动利用问题	——
文献[68]	Binaries	AngErza	VEX	符号执行	针对堆栈缓冲区溢出和格式化字符串的漏洞利用生成	——
文献[61]	Binaries	R2dlAEG	VEX	符号执行	针对系统安全防护机制的突破利用问题	——
文献[19]	Binaries	angr	VEX	符号执行 模糊测试 程序切片等	多架构的二进制漏洞检测与利用平台研究	http://angr.io/
文献[4]	Binaries	SAGE	——	符号执行 模糊测试	针对 X86 架构二进制漏洞研究	——
文献[22]	Binaries	S^2E	TCG LLVM	符号执行	针对二进制的符号执行框架	http://s2e.epfl.ch

文献[58]	Binaries	Mayhem	BAP	符号执行模糊测试	针对二进制程序的漏洞检测与利用	http://forallsecure.com/mayhem.html

3.2 局限与展望

分析现有研究成果不难发现，在符号执行技术领域中，虽然已经取得不错的研究成果，但是现阶段也有着发展不成熟的方面，主要还面临着以下挑战：探索路径太多——路径爆炸问题，路径条件太难解决——约束求解问题。

为缓解路径爆炸问题，研究人员尝试以下几种思路：一是路径剪枝[114-115]，对程序的路径数量进行限制，通过修剪不可实现的分支以减少调用求解器的次数，实施路径剪枝的前提需指明多余路径的断定条件；二是状态合并[116-117]，可以将不同的路径融合成一个状态，侧重将同一语句中两个符号状态非常相似进行状态合并；三是并行化策略[118-119]，符号执行探索程序路径并形成符号执行树，执行树每条约束路径都是独立的，可以将探索过程分配给多个符号引擎探索是合理的；四是搜索策略优化[120]，通过设计合理、高效的搜索策略，可提升基于路径空间遍历的符号执行的效率，以经典式、指导式和混合式搜索策略在符号执行应用为主；五是利用先决条件预测[121]，利用数据和特征提取一些先验知识，用来预测程序路径执行搜索并减少路径爆炸。

为缓解路径约束求解问题，研究人员尝试以下几种思路：一是精简约束输入[17]，将路径约束条件利用表达式重写、约束集简化和符号值具体化等措施，转换为更简单的形式输入到约束求解器中求解；二是缓存策略[122]，缓存求解是以重用以前计算的结果来提高 SAT/SMT 求解器的性能，主要分为存储部分简单的结果和存储整体方案的结果这两种策略；三是值具体化[123]，约束求解器难免会遇到无法求解的符号表达式时，通过将符号值换成具体值，利用具体值求解时的确定性，降低符号操作数带来的复杂性；四是延迟策略[124]，当执行器遇到涉及符号分支操作语句时，考虑接受 true 或 false 分支，可在路径条件中添加一个延迟约束，优先考虑可行的路径并延迟其它路径，不过延迟策略可能会导致大量路径冗余从而产生更多的求解器无用。

总体来看，路径爆炸和约束求解问题是制约着符号执行发展的主要因素。缓解路径爆炸方面，研究人员通过路径剪枝、状态合并、并行化策略、搜索策略优化、利用先决条件预测等措施，进行状态空间简化以及启发式搜索有效缓解路径爆炸问题；对约束求解方面，研究人员通过精简约束输入，实现求解器的外部优化，以及利用缓存策略、值具体化和延迟策略等措施进行求解器内部优化，进一步提升求解器对约束条件的处理效率与能力。除此之外，针对路径爆炸状态还可借助有限状态机并建立状态筛选规则，将路径中无关的状态剔除所达到状态空间简化；针对约束求解优化的问题，尽管 SAT 是一个已经被证明的 NP-Complete 问题，但数学的发展已经产生实际适用方法来求解，尤其是，符号计算可以提高非线性表达式的求解效率进而提升全局的约束求解能力。

尽管符号执行技术的发展存在一些挑战，但是挑战也带来了机遇，结合当下的研究热点，给出未来重点的研究方向：

(1) 针对路径爆炸优化方面，借助云服务器并行化策略。尽管符号执行的并行化策略使得执行树的每条路径都是独立于其他路径，使得路径探索工作分配减少，性能有所提升，但受限于场地与时效。通过借助云服务器的并行化策略可增大探索工作路径以减少探索时间，另一方面还从离线状态过渡到在线检测模式，从而推进在安全漏洞方面的广泛应用。

(2) 针对符号执行成果，需构建统一基准测试方面。现有评估方式以常见指标和少量开源的漏洞数据集进行实际评测，研究人员面临着研究成果的性能和效率的验证，以构建忽略目标对象的代码形式，可从符号执行的符号推理和路径状态入手进行基准测试，验证符号执行技术跨语言和项目的场景的实用性。

(3) 针对符号理论与实用性方面。在理论方面对符号执行的一般性理论进行数理逻辑论证。另外，在实用性方面，进一步与其它技术相结合，充分利用现有技术优势与符号执行进行优势互补，进而形成综合分析平台。

4 结束语

符号执行成功地应用在多任务场景中，尤其是在安全漏洞的检测、利用生成和测试用例等方面，受到越来越多的关注和重视。本文在符号执行相关研究基础上，总结了静态符号执行到动态符号执行技术的发展历程阶段，归纳了符号执行系统设计由前端、执行阶段和后端组成，其中核心设计环节的中间语言、路径搜索策略和约束求解也制约着符号执行的发展。在应用方面按照漏洞利用生成、漏洞检测等两个方面进行总结。漏洞利用生成方面，对控制流劫持漏洞、堆栈漏洞和 Android 系统漏洞等三个方向进行整理与分析；在漏洞检测方面，在总结符号执行漏洞检测的过程之后，对基于

源码、二进制文件、内存溢出类和 Web 注入类检测方法进行了归纳分析。最后从符号执行技术特点角度对各项研究整理分析，得出了目标分析对象多元化、技术迭代更加有效和适用场景具有普遍性等特点，并讨论了现有研究中存在的局限和未来发展方向。

通过总结符号执行技术与相关的应用可以看出：符号执行技术发展迅速，取得了不错的性能与效果，逐渐成为安全漏洞领域的研究热点之一。随着状态爆炸、约束求解、内存建模和非线性求解能力等进一步优化，一定程度提升符号执行面向大型程序的性能问题。

参考文献

[1] ZHANG T, WANG P, GUO X. A survey of symbolic execution and its tool KLEE[J]. Procedia Computer Science, 2020, 166: 330-334.

[2] PĂSĂREANU C S, KERSTEN R, LUCKOW K, et al. Symbolic execution and recent applications to worst-case execution, load testing, and security analysis[J]. Advances in Computers, 2019, 113: 289-314.

[3] PĂSĂREANU C S, RUNGTA N. Symbolic PathFinder : symbolic execution of Java bytecode [C]//Proceedings of the IEEE/ACM international conference on Automated software engineering. 2010: 179-180.

[4] BOUNIMOVA E, GODEFROID P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production[C]//2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013: 122-131.

[5] CADAR C, SEN K. Symbolic execution for software testing: three decades later[J]. Communi- cations of the ACM, 2013, 56(2): 82-90.

[6] CADAR C, GODEFROID P, KHURSHID S, et al. Symbolic execution for software testing in practice: preliminary assessment[C]//Proceedings of the 33rd International Conference on Software Engineering. 2011: 1066-1071.

[7] KING J C. Symbolic execution and program testing[J]. Communications of the ACM,1976,19 (7): 385–394.

[8] SEN K. Concolic testing[C]//Proceedings of the twenty-second IEEE/ACM international con- ference on Automated software engineering. 2007: 571-572.

[9] SEN K. Concolic testing: a decade later (keynote) [C]// Proceedings of the 13th International Workshop on Dynamic Analysis. 2015: 1-1.

[10] CADAR C, GANESH V, PAWLOWSKI P M, et al. EXE: Automatically generating inputs of death [J]. ACM Transactions on Information and System Security (TISSEC), 2008, 12(2): 1-38.

[11] CHIPOUNOV V, GEORGESCU V, ZAMFIR C, et al. Selective symbolic execution[C]//Pro- ceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep). 2009 (CONF).

[12] WEN S, FENG C, MENG Q, et al. Testing network protocol binary software with selective symbolic execution[C]//2016 12th International Conference on Computational Intelligence and Security (CIS). IEEE, 2016: 318-322.

[13] CHANDRA S, FINK S J, SRIDHARAN M. Snugglebug: a powerful approach to weakest preconditions[C]//Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009: 363-374.

[14] DINGES P, AGHA G. Targeted test input generation using symbolic-concrete backward execution[C]//Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. 2014: 31-36.

[15] STEPHENS N, GROSEN J, SALLS C, et al. Driller:augmenting fuzzing through selective symbolic execution[C]//NDSS. 2016, 16(2016): 1-16.

[16] POEPLAU S, FRANCILLON A. Symbolic execution with {SymCC}: Don't interpret, compile! [C]// 29th USENIX Security Symposium (USENIX Security 20). 2020: 181-198.

[17] CADAR C, NOWACK M. KLEE symbolic execution engine in 2019[J]. International Journal on Software Tools for Technology Transfer, 2020: 1-4.

[18] BUCUR S, URECHE V, ZAMFIR C, et al. Parallel symbolic execution for automated real-world software testing[C]//Proceedings of the sixth conference on Computer systems. 2011: 183-198.

[19] SHOSHITAISHVILI Y, WANG R,SALLS C, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis[C]//2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 138-157.

[20] SONG D, BRUMLEY D, YIN H, et al. BitBlaze: A new approach to computer security via binary analysis[C]//International conference on information systems security. Springer, Berlin, Heidelberg, 2008: 1-25.

[21] BURNIM J, SEN K. Heuristics for scalable dynamic test generation[C]//2008 23rd IEEE/ ACM International Conference on Automated Software Engineering. IEEE, 2008: 443-446.

[22] CHIPOUNOV V, KUZNETSOV V, CANDEA G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. Acm Sigplan Notices, 2011, 46(3): 265-278.

[23] CHIPOUNOV V, KUZNETSOV V, CANDEA G. The S2E platform: Design, implementation, and applications[J]. ACM Transactions on Computer Systems (TOCS), 2012, 30(1): 1-49.

[24] POEPLAU S, FRANCILLON A. Systematic comparison of symbolic execution systems: inter- mediate representation and its generation [C]// Proceedings of the 35th Annual Computer Security Applications Conference. 2019: 163-176.

[25] YUN I, LEE S, XU M, et al. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing[C]//27th USENIX Security Sym- posium (USENIX Security 18). 2018: 745-761.

[26] POEPLAUS, FRANCILLON A. SymQEMU: Compilation-based symbolic execution for binaries [C]//NDSS. 2021.

[27] GODEFROID P, KLARLUND N, SEN K. DART: Directed automated random testing[C]//Pro- ceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. 2005: 213-223.

[28] WANG H, LIU T, GUAN X, et al. Dependence Guided Symbolic Execution[J]. IEEE Transactions on Software Engineering, 2017, 43(3):252–271.

[29] BARDIN S, BAUFRETON P, Cornuet N, et al. Binary-level testing of embedded programs [C]//

2013 13th International Conference on Quality Software. IEEE, 2013: 11-20.
[30] DAVID R, BARDIN S, TAT D, et al. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis[C]//2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 2016, 1: 653-656.
[31] CHEN Z, CHEN Z, SHUAI Z, et al. Synthesize solving strategy for symbolic execution [C]// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 348-360.
[32] GIANTSIOS A, PAPASPYROU N, SAGONAS K. Concolic testing for functional languages[C]// Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming. 2015: 137-148.
[33] GUPTA A, MAJUMDAR R, RYBALCHENKO A. From tests to proofs[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2009: 262-276.
[34] VISHNYAKOV A, FEDOTOV A, KUTS D, et al. Sydr: Cutting edge dynamic symbolic execution[C]// 2020 Ivannikov ISPRAS Open Conference (ISPRAS). IEEE, 2020: 46-54.
[35] MAJUMDAR R, SEN K. Hybrid concolic testing[C]//29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 416-426.
[36] CLAESSEN K, HUGHES J. QuickCheck: a lightweight tool for random testing of Haskell programs[C]//Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. 2000: 268-279.
[37] MA K K, YIT PHANG K, FOSTER J S, et al. Directed symbolic execution[C]//International Static Analysis Symposium. Springer, Berlin, Heidelberg, 2011: 95-111.
[38] GERASIMOV A Y. Directed dynamic symbolic execution for static analysis warnings confirmation [J]. Programming and Computer Software, 2018, 44(5): 316-323.
[39] PAPADAKIS M, MALEVRIS N. A symbolic execution tool based on the elimination of infeasible paths[C]//2010 Fifth International Conference on Software Engineering Advances. IEEE,2010:435-440.
[40] SU T, PU G, FANG B, et al. Automated coverage-driven test data generation using dynamic symbolic execution[C]//2014 Eighth International Conference on Software Security and Reliability (SERE). IEEE, 2014: 98-107.
[41] DONG Q, YAN J, ZHANG J, et al. A Search Strategy Guided by Uncovered Branches for Concolic Testing[C]//2013 13th International Conference on Quality Software. IEEE, 2013: 21-24.
[42] PARK S, HOSSAIN B M M, Hussain I, et al. Carfast: Achieving higher statement coverage faster[C]//Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. 2012: 1-11.
[43] Online Computer Library Center. CREST: Automatic test generation tool for C[EB/OL]. (2009-03-06)[2022-05-31].https://code.google.com/p/crest/.
[44] NOLLER Y, KERSTEN R, PĂSĂREANU C S. Badger: complexity analysis with fuzzing and symbolic execution[C]//Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2018: 322-332.
[45] BÖTTINGER K, ECKERT C. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries[C]// International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer , Cham, 2016: 25-34.
[46] KUCHTA T, PALIKAREVA H, Cadar C. Shadow symbolic execution for testing software patches[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2018, 27(3): 1-32.
[47] MARINESCU P D, CADAR C. KATCH: High-coverage testing of software patches [C]// Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013: 235-245.
[48] LI M, CHEN Y, WANG L, et al. Dynamically validating static memory leak warnings[C]//Pro- ceedings of the 2013 International Symposium on Software Testing and Analysis. 2013: 112-122.
[49] KRISHNAMOORTHY S, HSIAO M S, LINGAPPAN L. Strategies for scalable symbolic execution-driven test generation for programs[J]. Science China Information Sciences, 2011, 54(9): 1797-1812.
[50] MOURA L, BJØRNER N. Z3: An efficient SMT solver[C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2008: 337-340.
[51] STUMP A, BARRETT C W, DILL D L. C VC: A cooperating validity checker[C]//Internat ional Conference on Computer Aided Verificati on. Springer, Berlin, Heidelberg, 2002:500-504.
[52] Online Computer Library Computer, Anon STP (simple theorem prover)[EB/OL].(2006-01-01)[2022-06-15].http://sourceforge.net/projects/stp-fast-prover.
[53] ZHANG Y, CHEN Z, SHUAI Z, et al. Multiplex symbolic execution: exploring multiple paths by solving once[C]//2020 35th IEEE/ACM Inter- national Conference on Automated Software Engineering (ASE). IEEE, 2020: 846-857.
[54] HU H, YE R, ZHANG H. Vulnerability life cycle oriented security risk metric method[J]. Journal of Software, 2018, 29(5): 1213-1229.
[55] Online Computer Library Computer.国家信息安全漏洞库[EB/OL].(2022-05-01)[2022-10-31]. http://www.cnnvd.org.cn/web/vulreport/queryListByType.tag?qnewtype=2.
[56] HUANG N, HUANG S, CHANG C. Analysis to heap overflow exploit in linux with symbolic execution[C]//IOP Conference Series: Earth and Environmental Science. IOP Publishing, 2019, 252(4): 042100.
[57] AVGERINOS T, CHA S K, REBERT A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.
[58] CHA S K, AVGERINOS T, REBERT A, et al. Unleashing Mayhem on Binary Code[C]//2012 IEEE Symposium on Security and Privacy. IEEE, 2012: 380–394.
[59] HUANG S K, HUANG M H, HUANG P Y, et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations[C]//2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE,

2012: 78-87.
[60] ECKERT M, BIANCHI A, WANG R, et al. {HeapHopper}: Bringing Bounded Model Checking to Heap Implementation Security[C]// 27th USENIX Security Symposium (USENIX Security 18). 2018: 99-116.
[61] FANG H, WU L, WU Z. Automatic return- to-dl-resolve exploit generation method based on symbolic execution[J]. Computer Science, 2019, 46(2): 127-132.
[62] 黄桦烽,苏璞睿,杨轶,等.可控内存写漏洞自动利用生成方法[J].通信学报,2022,43(01):83-95.
HUANG H F, SU P R,YANG Y, et al. Automatic exploitation generation method of write-what-where vulnerability[J].Journal on Communication, 2022,43(01):83-95.
[63] PADARYAN V A , KAUSHAN V V , FEDOTOV A N. Automated exploit generation method for stack buffer overflow vulnerabilities[J]. Pro- ceedings of the Institute for System Programming of RAS(Proceedings of ISP RAS), 2014, 26(3):127-144.
[64] WANG Y, ZHANG C, XIANG X, et al. Revery: From proof-of-concept to exploitable[C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 1914-1927.
[65] WU W, CHEN Y, XU J, et al. {FUZE}: Towards Facilitating Exploit Generation for Kernel {Use-After-Free}Vulnerabilities[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 781-797.
[66] LIU D, WANG J, RONG Z, et al. Pangr: A Behavior-Based Automatic Vulnerability Detec- tion and Exploitation Framework[C]//2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communi- cations/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/ BigDataSE). IEEE , 2018: 705-712.
[67] 王瑞鹏,张旻,黄晖,等. 基于符号执行的格式化字符串漏洞自动验证方法研究[J].空军工程大学学报(自然科学版),2021,22(3):82−88.
WANG R P, ZHANG M, HUANG H, et al. Research on Automatic Exploit Generation Method of Format String Vulnerability Based on Symbolic Execution[J]. Journal of Air Force Engineering University (Natural Science Edition), 2021,22(3): 82 – 88.
[68] DIXIT S, GEETHNA T K, JAYARAMAN S, et al. AngErza: Automated Exploit Generation[C]// 2021 12th International Conference on Computing Communication and Networking Technologies (ICCC N T). IEEE, 2021: 1-6.
[69] GARCIA J, HAMMAD M, GHORBANI N, et al. Automatic generation of inter-component communication exploits for Android applications[C]// Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 661-671.
[70] LUO L, ZENG Q, CAO C, et al. System service call-oriented symbolic execution of Android framework with applications to vulnerability discovery and exploit generation[C]//Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services. 2017: 225-238.
[71] YANG G, HUANG J, GU G. Automated generation of event-oriented exploits in Android hybrid apps [C]//Network and Distributed System Security Symposium. 2018.
[72] ZHOU M, ZENG F, ZHANG Y, et al. Automatic Generation of Capability Leaks' Exploits for Android Applications[C]//IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).IEEE, 2019: 291- 295.
[73] SEN K. Automated test generation using concolic testing[C]//Proceedings of the 8th India Software Engineering Conference. 2015: 9-9.
[74] SEN K, MARINOV D, AGHA G. CUTE: A concolic unit testing engine for C[J]. ACM SIGSOFT Software Engineering Notes, 2005, 30(5): 263-272.
[75] DURAIBI S, ALASHJAEE A M, SONG J. A Survey of Symbolic Execution Tools[J]. International Journal of Computer Science and Security (IJCSS), 2019, 13(6): 244-255.
[76] CADAR C, DUNBAR D, ENGLER D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs [C]// OSDI. 2008, 8: 209-224.
[77] PĂSĂREANU C S, VISSER W, BUSHNELL D, et al. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis[J].Automated Software Engineering, 2013, 20(3): 391-425.
[78] BRAIONE P, DENARO G, PEZZÈ M. JBSE: A symbolic executor for java programs with complex heap inputs[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016: 1018-1022.
[79] SEN K, AGHA G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools[C]//International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, 2006: 419-423.
[80] LUCKOW K, DIMJAŠEVIĆ M, GIANNAKO POULOU D, et al. JDART: a dynamic symbolic analysis framework[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2016: 442-459.
[81] CHEN T, ZHANG X, CHEN R, et al. Conpy: Concolic execution engine for python applications [C]//International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, 2014: 150-163.
[82] SAXENA P, AKHAWE D, HANNA S, et al. A symbolic execution framework for javascript[C]// 2010 IEEE Symposium on Security and Privacy. IEEE, 2010: 513-528.
[83] CHAUDHURI A, FOSTER J S. Symbolic security analysis of ruby-on-rails web applications[C]// Proceedings of the 17th ACM conference on Computer and communications security. 2010: 585-594.
[84] TILLMANN N, HALLEUX J. Pex–white box test generation for. net[C]//International con- ference on tests and proofs. Springer, Berlin, Heidelberg, 2008: 134-153.
[85] GODEFROID P, LEVIN M Y, MOLNAR D. SAGE: whitebox fuzzing for security testing[J]. Communications of the ACM, 2012, 55(3): 40-44.
[86] NIU W N,DING X F,LIU Z ,et al.Vulnerability Finding Using Symbolic Execution on Binary Programs[J].Computer Science,2013,40(10):119-121

+138.
[87] LETYCHEVSKYI O. Two-Level Algebraic Methodfor Detection of Vulnerabilities in Binary Code[C]//2019 10th IEEE International Con- ference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS). IEEE, 2019, 2: 1074- 1077.
[88] WEN S, FENG C, MENG Q, et al. Analyzing network protocol binary software with joint symbolic execution[C]//2016 3rd International Conference on Systems and Informatics (ICSAI). IEEE, 2016: 738-742.
[89] Online Computer Library Center. CVE Details [EB/OL].(2022-5-1)[2022-10-31].https://www. cvedetails.com/ vulnerabilities-by-types.php.
[90] QIXUE X, YU C, LANLAN Q I, et al. Detection and analysis of size controlled heap allocation[J]. Journal of Tsinghua University, 2015, 55(5): 572-578.
[91] 张超,潘祖烈,樊靖. 基于符号执行的堆溢出 fastbin 攻击检测方法[J].计算机工程, 2020, 46(10): 151–158.
ZHAO C,PAN Z L,FAN J.Detection Method for Heap Overflow fastbin Attack Based on Symbolic Execution[J].Computer Engineering,2020,46 (10) : 151–158.
[92] 黄宁,黄曙光,梁智超. 基于符号执行的 Unlink 攻击检测方法[J]. 华南理工大学学报(自然科学版), 2018, 46(08):81–87.
HUANG N,HUANG S G,LIANG Z C.Detection of Unlink Attack Based on Symbolic Execution [J].Journal of South China University of Technology (Natural Science Edition), 2018, 46(08): 81–87.
[93] KAREEM F Q, AMEEN S Y, SALIH A A, et al. SQL injection attacks prevention system technology[J]. Asian Journal of Research in Computer Science, 2021, 6(15): 13-32.
[94] FU X, LU X, PELTSVERGER B, et al. A static analysis framework for detecting SQL injection vulnerabilities[C]//31st annual international computer software and applications conference (COMPSAC 2007). IEEE, 2007, 1: 87-96.
[95] FU X, QIAN K. SAFELI: SQL injection scanner using symbolic execution[C]//Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications. 2008: 34-39.
[96] BISHT P, HINRICHS T, SKRUPSKY N, et al. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications[C]//Proceedings of the 17th ACM conference on Computer and communications security. 2010: 607-618.
[97] WANG D, ZHAO W B, DING Z M. Review of detection for injection vulnerability of Web application[J]. Journal of Beijing University of Technology,2016,42(12):1822-1832.
[98] HALFOND W G J, CHOUDHARY S R, ORSO A. Penetration testing with improved input vector identification[C]//2009 International Conference on Software Testing Verification and Validation. IEEE, 2009: 346-355.
[99] LIU M, ZHANG B, CHEN W, et al. A survey of exploitation and detection methods of XSS vulnerabilities[J]. IEEE access, 2019, 7: 182004- 182016.
[100] 孙基男,潘克峰,陈雪峰,等. 基于符号执行的注入类安全漏洞的分析技术[J].北京大学学报(自然科学版), 2018, 54(01):1–13.
SUN J N, PAN K F, CHEN X F, et al. Static Analysis of Injection Security Vulnerabilities Based on Symbolic Execution[J].Acta Scientiarum Naturalium Universitatis Pekinensis,2018,54 (01): 1–13.
[101] 刘双印,雷墨鹭兮,王璐,等.区块链关键技术及存在问题研究综述[J].计算机工程与应用, 2022, 58(03):66-82.
LIU S Y, LEI M X, WANG L, et al. Survey of Blockchain Key Technologies and Existing Problems[J]. Computer Engineering and Applica- tions, 2022,58(03):66-82.
[102] 张弛,司徒凌云,王林章. 物联网固件安全缺陷检测研究进展[J].信息安全学报,2021, 6(03): 141–158.
ZHANG C, SITU L Y, WANG L Z. Research Progress on Security Defect Detection of IoT Firmware[J].Journal of Cyber Security,2021,6(03): 141–158.
[103] SHAO J, VU M, ZHANG M, et al. Symbolic ns-3 for Efficient Exhaustive Testing: Design, Implementation, and Simulations[C]//Proceedings of the Workshop on ns-3. 2022: 49-56.
[104] QIANG W, LIAO Y, SUN G, et al. Patch-related vulnerability detection based on symbolic execution[J]. IEEE Access, 2017, 5: 20777-20784.
[105] FEIST J, MOUNIER L, BARDIN S, et al. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free[C]//Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering. 2016: 1-12.
[106] CHEN N, KIM S. Star: Stack trace based automatic crash reproduction via symbolic execution[J]. IEEE transactions on software engineering, 2014, 41(2): 198-220.
[107] SEREBRYANY K. Continuous fuzzing with libfuzzer and addresssanitizer[C]//2016 IEEE Cybersecurity Development (SecDev). IEEE, 2016: 157-157.
[108] SCHWARTZ E J, AVGERINOS T, BRUMLEY D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]//2010 IEEE symposium on Security and privacy. IEEE, 2010: 317-331.
[109] 梅瑞,严寒冰,沈元,等. 二进制代码切片技术在恶意代码检测中的应用研究[J].信息安全学报, 2021, 6(03):125–140.
MEI R,YAN H B,SHEN Y, et al.Application Research of Slicing Technology of Binary Executables in Malware Detection[J].Journal of Cyber Security, 2021, 6(03):125–140.
[110] LI Y, HUANG C, WANG Z, et al. Survey of software vulnerability mining methods based on machine learning[J]. Journal of software, 2020, 31(7): 2040-2061.
[111]黄桦烽,王嘉捷,杨轶,等.有限资源条件下的软件漏洞自动挖掘与利用[J].计算机研究与发展, 2019, 56(11):2299-2314.
HUANG H , WANG J , YANG Y , et al. Automatic

Software Vulnerability Discovery and Exploit Under the Limited Resource Conditions[J]. Journal of Computer Research and Development, 2019, 56 (11):2299–2314.
[112] BALDONI R, COPPA E, D'ELIA D C, et al. A survey of symbolic execution techniques[J]. ACM Computing Surveys (CSUR), 2018, 51(3): 1-39.
[113] BOER F S, BONSANGUE M. Symbolic execution formally explained[J].Formal Aspects of Computing, 2021, 33(4): 617-636.
[114] MUSTAFA A, WAN-KADIR W M N, IBRAHIM N, et al. Automated Test Case Generation from Requirements: A Systematic Literature Review[J]. Computers, Materials and Continua, 2021, 67(2): 1819-1833.
[115] BUSSE F, NOWACK M, CADAR C. Running symbolic execution forever[C]//Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020: 63-74.
[116] KAPUS T, CADAR C. A segmented memory model for symbolic execution[C]//Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019: 774-784.
[117] LUO S, XU H, BI Y, et al. Boosting symbolic execution via constraint solving time prediction (experience paper)[C]//Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 336-347.
[118] ZHENG P, ZHENG Z, LUO X. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution[C]//Pro- ceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 740-751.
[119] ZHENG L, LIAO X, JIN H. Efficient and scalable graph parallel processing with symbolic execution[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2018, 15(1): 1-25.
[120] SABBAGHI A, KEYVANPOUR M R. A Systematic Review of Search Strategies in Dynamic Symbolic Execution[J]. Computer Standards & Interfaces, 2020, 72: 103444.
[121] HE J, SIVANRUPAN G, TSANKOV P, et al. Learning to Explore Paths for Symbolic Execution[C]//Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communi- cations Security. 2021: 2526-2540.
[122] TALJAARD J, GELDENHUYS J, VISSER W. Constraint Caching Revisited[C]//NASA Formal Methods Symposium. Springer, Cham, 2020: 251-266.
[123] PANDEY A, KOTCHARLAKOTA P R G, Roy S. Deferred concretization in symbolic execution via fuzzing[C]//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019: 228-238.
[124] KAPUS T, BUSSE F, CADAR C. Pending constraints in symbolic execution for better exploration and seeding[C]//2020 35th IEEE/ ACM International Conference on Automated Software Engineering (ASE). IEEE, 2020: 115- 126.