计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200600128

SymFuzz:一种复杂路径条件下的漏洞检测技术

李明磊　黄　晖　陆余良　朱凯龙
国防科技大学电子对抗学院　合肥 230037
网络空间安全态势感知与评估安徽省重点实验室　合肥 230037
(921519263@qq.com)

摘　要　当前漏洞检测技术可以实现对小规模程序的快速检测,但对大型或路径条件复杂的程序进行检测时其效率低下。为实现复杂路径条件下的漏洞快速检测,文中提出了一种复杂路径条件下的漏洞检测技术 SymFuzz。SymFuzz 将导向式模糊测试技术与选择符号执行技术相结合,通过导向式模糊测试技术对程序路径进行过滤,利用选择符号执行技术对可能触发漏洞的路径进行求解。该技术首先通过静态分析获取程序漏洞信息;然后使用导向式模糊测试技术,快速生成可以覆盖漏洞函数的测试用例;最后对漏洞函数内可以触发漏洞的路径进行符号执行,生成触发程序漏洞的测试用例。文中基于 AFL 与 S2E 等开源项目实现了 SymFuzz 的原型系统。实验结果表明,SymFuzz 与现有的模糊测试技术相比,在复杂路径条件下的漏洞检测效果提高显著。

关键词:模糊测试;符号执行;静态分析;污点分析;漏洞检测
中图法分类号　TP309

SymFuzz:Vulnerability Detection Technology Under Complex Path Conditions

LI Ming-lei,HUANG Hui,LU Yu-liang and ZHU Kai-long
College of Electronic Engineering,National University of Defense Technology,Hefei 230037,China
Anhui Key Laboratory of Cyberspace Security Situation Awareness and Evaluation,Hefei 230037,China

Abstract　The current vulnerability detection technology can realize the rapid detection of small-scale programs,but it is inefficient when performing vulnerability detection on programs with large or complex path conditions. In order to achieve a rapid detection of vulnerabilities under complex path conditions,this paper proposes a vulnerability detection technology SymFuzz under complex path conditions. SymFuzz combines guided fuzzing technology and selected symbolic execution technology,filters program paths through guided fuzzing technology,and uses selected symbolic execution technology to solve paths that may trigger vulnerabilities. This technology first obtains program vulnerability information through static analysis. Then it uses guided fuzzy test technology to quickly generate test cases that can cover the vulnerability function. Finally,it executes symbolic execution on the path that can trigger the vulnerability within the vulnerability function to generate a test case that triggers the program vulnerability. This paper implements the prototype system of SymFuzz based on open source projects such as AFL and S2E. The comparison experiments show that SymFuzz significantly improves the effectiveness of vulnerability detection under complex path conditions compared with existing fuzzy testing techniques.

Keywords　Fuzzy testing,Symbol execution,Static analysis,Stain analysis,Vulnerability detection

1　引言

程序漏洞严重威胁着我国网络空间的安全,如何快速对程序漏洞进行检测是网络空间安全的重要研究内容。现阶段程序漏洞检测的主要技术手段为模糊测试技术,其本质是通过随机生成的测试用例对程序进行测试,并对程序运行状态进行检测。研究人员通过对程序运行状态的分析,来判断程序中漏洞所在的位置及类型[1]。该方法因其部署简单、检测快速的特性被广泛应用于漏洞检测中。但随着程序规模的增长与开发人员安全意识的提高,现在的程序漏洞多存在于程序路径深处且路径条件较为复杂。此时,模糊测试技术随机生成的测试用例难以快速对复杂路径条件下的漏洞进行检测。

因此,本文以复杂路径条件下的漏洞快速检测为主要研究内容,其中文中将复杂路径条件分为两种情况:一种为该漏洞需同时覆盖程序的多个基本块,如 UAF 漏洞;另一种为漏

到稿日期:2020-06-22　返修日期:2020-08-21　本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划项目(2017YFB0802905)
This work was supported by the National Key R&D Program of China (2017YFB0802905).
通信作者:陆余良(mr.lee.public.cn@gmail.com)

洞所在路径的分支条件复杂。本文分析了不同的模糊测试技术与符号执行技术的利弊，并基于此提出了一种复杂路径条件下的漏洞检测技术。

本文第 2 节对模糊测试技术与符号执行技术进行了简要分析；第 3 节介绍了 SymFuzz 的技术框架与本文所做的改进；第 4 节介绍了距离计算模型与种子能量动态调控模型；第 5 节介绍了导向式符号执行技术；第 6 节实现了原型系统并进行对比实验；最后总结全文。

2 背景

2.1 模糊测试技术

模糊测试技术于 1990 年被首次提出[2]，发展至今形成了 3 类模糊测试技术，即黑盒模糊测试、白盒模糊测试与灰盒模糊测试，不同类型的模糊测试技术有其独特的优势与应用场景。黑盒模糊测试主要应用于无源码的程序，仅通过随机生成的测试用例对程序进行测试，这使得黑盒模糊测试具有很好的时间效率，但其随机生成的测试用例由于没有针对性，因此难以检测程序深层路径。白盒模糊测试应用于有源码的程序，根据源码分析的结果有针对性地生成测试用例，可以覆盖到程序深层路径，但对程序源码的分析需花费大量的时间与系统资源，这限制了白盒模糊测试的时间效率。灰盒模糊测试引入轻量级的程序分析技术对程序状态进行分析，指导测试用例的生成，在保证模糊测试的时间效率的同时提高了测试用例的覆盖率[3-5]。

为提高模糊测试技术的覆盖率与时间效率，研究人员做出了大量努力，在模糊测试技术中引入了污点分析技术[6]、符号执行技术[7]以及静态分析技术[8]，并取得了不错的效果。但随着现代程序的规模不断增大以及开发人员安全意识的提高，如今程序漏洞通常隐藏在程序深处，单纯的提高模糊测试技术的覆盖率已经无法满足现实的需要。

为实现对程序深处的漏洞检测，Böhme 提出了导向式模糊测试技术[9]，导向式模糊测试技术不再以提高测试用例覆盖率为指导，而是以覆盖程序特定位置为指导。在实际应用中，研究人员首先对程序中可能存在漏洞的位置进行定位，然后将漏洞位置信息输入到导向式模糊测试器中，并以测试用例到该位置的距离为指导，快速生成可以覆盖程序漏洞所在位置的测试用例。

导向式模糊测试技术将系统资源集中在程序存在潜在威胁的位置，提高了模糊测试的效率。基于导向式模糊测试的思想，研究人员开发出了 AFLGO，Hawkeye[10] 等工具。在实际使用中，这些工具可以快速检测出程序中的栈溢出、除零错误引起的漏洞，但其对堆溢出等需要测试用例同时覆盖多个特定位置才可以触发的漏洞检测的效果并不理想。如何提高导向式模糊测试技术在复杂路径条件下的漏洞检测效果是当前模糊测试领域的热点问题。

2.2 符号执行技术

符号执行技术使用符号变元代替程序的具体输入，监控程序控制流并记录程序执行轨迹。其在程序中的每一个分支节点，收集符号变元的路径约束，每执行完一条路径，系统恢复到上一个分支节点并探索新的路径。其通过全路径分析的方式定位程序漏洞位置，求解程序入口到程序漏洞的路径约束来得到触发漏洞的测试用例[11-13]。

早期的符号执行技术受限于计算机性能主要采取静态分析的方式，静态分析的方式无法保证污点数据传播的完整性，导致代码覆盖率不高。随着计算机性能的快速提升，研究人员提出了动态符号执行技术，通过动态运行程序来收集程序执行过程中的信息，以检测程序中存在的漏洞。动态符号执行解决了静态符号执行代码覆盖率低的问题，其在小型程序的漏洞检测中取得了良好的效果。但动态符号执行技术由于需要同时保持程序不同状态下的符号信息，随着程序规模的扩大出现了路径状态空间爆炸的问题[14]。

2009 年选择符号执行技术的出现在一定程度上解决了路径状态空间爆炸的问题。选择符号执行技术的核心思想是将具体执行与符号执行相结合，如图 1 所示，对程序核心函数进行符号执行而不对核心函数或系统函数调用进行具体执行。这不仅有效降低了系统运行开支，还降低了约束求解器的求解难度。但选择符号执行技术对初始具体值输入的要求较高，当初始具体值输入无法到达符号执行域时，符号执行引擎将不会对程序进行符号执行。

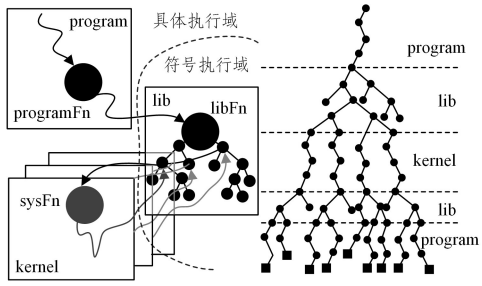

图 1 选择符号执行技术

Fig. 1 Selection symbolic execution technique

3 方法概述

为提高模糊测试技术在复杂路径条件下的漏洞检测能力，本文提出了一种将符号执行技术与导向式模糊测试相结合的方法，并基于该方法形成了原型系统 Symfuzz。为便于表述文中将包含漏洞的函数统称为漏洞函数，该技术首先通过静态分析对程序中的漏洞函数进行定位，利用导向式模糊测试技术对程序路径进行探索，快速生成到达漏洞函数的测试用例；然后符号执行引擎依靠导向式模糊测试器生成的测试用例具体执行到危险函数后，收集程序污点信息并在函数内部进行导向式符号执行；最后求解能覆盖程序特定路径的测试用例并触发漏洞。该技术的整体框架如图 2 所示。

该框架由导向式模糊测试与符号执行两部分组成。在导向式模糊测试部分中，对静态分析过程中的距离计算模型（见图 2 中的①）和模糊测试过程中种子能量调控过程（见图 2 中的②）进行改进；在符号执行部分中，对选择符号执行进行优化，实现了导向式符号执行（见图 2 中的③）。框架的具体执行过程如下：

(1)框架调用静态分析模块，读取二进制程序和漏洞信息生成程序控制流程图与函数调用流程图，计算程序各基本块

到漏洞函数的距离，将距离信息插桩到二进制程序中；

(2)模糊测试模块对插桩后的程序进行测试，根据种子执行路径上的程序基本块计算种子距离并动态调控种子能量，根据种子能量决定不同种子的变异数量，直到产生可以到达漏洞函数入口的测试用例；

(3)静态控制流分析模块读取漏洞信息、程序控制流图和函数调用图，分析得到漏洞函数内部的导向信息；

(4)符号执行引擎根据导向式模糊测试生成的测试用例和函数内导向信息，在漏洞函数内部进行导向式符号执行，生成可以触发程序漏洞的测试用例。

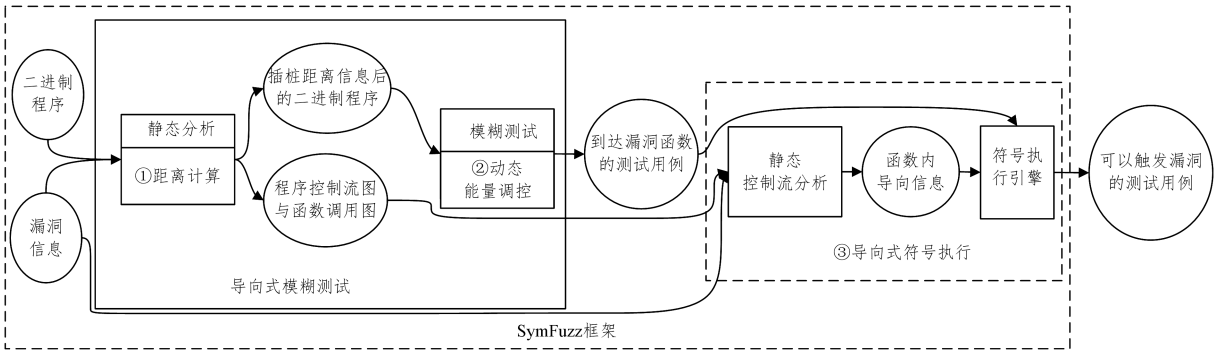

图 2 SymFuzz 的框架图

Fig. 2 Framework of SymFuzz

4 距离计算与种子能量调控

导向式模糊测试技术的核心在于建立合理的距离计算模型与种子能量调控策略。合理的距离计算模型可以保证快速生成覆盖漏洞函数的测试用例，合理的种子能量调控策略可以缓解距离计算模型的局部最优解。

4.1 距离计算模型

距离计算模型由基本块权重计算、函数距离计算、基本块距离计算和种子距离计算 4 个部分组成，其中函数权重计算、函数距离计算和种子距离计算在静态分析阶段完成，种子距离计算在模糊测试阶段完成。表 1 列出了距离计算模型中出现的变量及其含义。

表 1 变量表

Table 1 Variable list

Symbol	Name	Implication
W_{BB}	基本块权重	基本块 BB 在控制流中的重要程度
$Sidelen(i,j)$	基本块边长	边(i,j)的长度
$L(i,j)$	基本块间距离	同属一个函数的基本块 i 和基本块 j 之间的最短路径距离
D_{BB}	基本块距离	程序基本块到漏洞函数的距离
D_{Fun}	函数距离	程序函数到漏洞函数的距离
R	种子距离	种子到漏洞函数的距离
Q	归一化种子距离	归一化后的种子距离
P	种子能量	种子变异后产生的测试用例的数量

4.1.1 基本块间距离计算

基本块间距离表示同属一个函数的两个基本块之间的距离，是基本块距离计算的基础。目前，已有的导向式模糊测试工具在计算基本块间距离时，都没有对程序中的基本块进行区分，这导致基本块间的距离计算不够准确。

如图 3 所示，以函数内控制流图中基本块 1 和基本块 2 为例进行分析。基本块 1 有 3 个后继节点，基本块 2 只有 1 个后继节点。当种子执行轨迹经过基本块 1 时，种子变异后覆盖到其他路径的概率大于经过基本块 2 的种子。而导向式模糊测试的目的是在短时间内尽可能多地生成通过不同路径覆盖漏洞函数的测试用例，因此基本块位置对种子变异的影响也应被考虑到基本块间距离计算中。

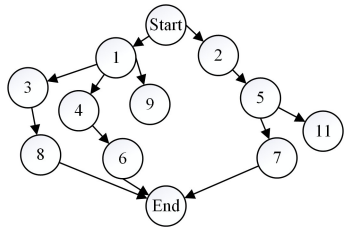

图 3 函数内控制流图

Fig. 3 Control flow graph within a function

本文以基本块出度为基本块的权重，如式(1)所示：

$$W_{BB}=\mathrm{Outdegree}(BB) \tag{1}$$

其中，W_{BB} 表示基本块 BB 的权重，Outdegree()表示计算出度。利用式(1)可以得到基本块 1 的权重为 3，基本块 2 的权重为 1。计算出程序中的基本块权重后，利用式(2)对属于一个函数的基本块边进行赋值，即：

$$Sidelen_{ij}=\frac{1}{W_i} \tag{2}$$

其中，$Sidelen_{ij}$ 表示基本块 i 与基本块 j 之间的边长，其长度为基本块 i 权重的倒数。利用式(2)对图 3 中的边进行赋值，其结果如图 4 所示。

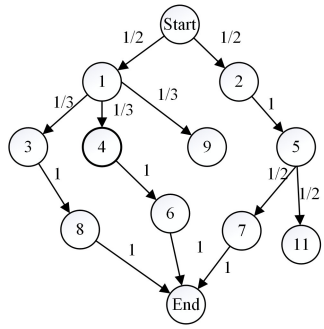

图 4 边长赋值

Fig. 4 Side length assignment

对基本块边赋值后，使用 Djikstra 算法[15]求解同属一个函数的基本块 i 和基本块 j 之间的最短路径距离，以作为基本块 i 和基本块 j 的基本块间距离 $L(i,j)$。

4.1.2 种子距离计算

在计算种子距离之前，首先要计算程序基本块间距离并将其插桩到程序中，然后在模糊测试的过程中根据种子执行轨迹计算种子距离。在计算基本块距离时将基本块分为 3 类进行讨论，如图 5 所示，图中方框表示函数，圆圈表示基本块。

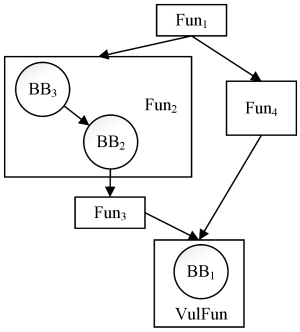

图 5 基本块的分类

Fig. 5 Basic block classification

情况 1 基本块属于漏洞函数，此时该基本块的基本块距离为 0，如图 4 中的基本块 BB_1。

情况 2 基本块不属于漏洞函数，但该基本块可直接调用其他函数到达漏洞函数，此时该基本块的基本块距离为基本块所在函数到漏洞函数的距离，如图 5 中的基本块 BB_2 的基本距离为函数 Fun_2 到函数 VulFun 的距离。

情况 3 基本块不属于漏洞函数，但该基本块可以通过同属一个函数的基本块到达漏洞函数，此时该基本块的基本块距离为该基本块到跳板基本块的距离加上跳板基本块的基本块距离，如图 5 中的基本块 BB_3 的基本距离为基本块 BB_3 到基本块 BB_2 的距离加上 BB_2 的基本块距离。

式(3)为基本块距离的计算公式，即：

$$D_{BB}=\begin{cases}0, & BB\in VulFun\\ D_{Fun}BB\in Fun, & BB\in T\\ L(BB,t)+D_tBB\in Fun, & t\in T\end{cases} \tag{3}$$

其中，$VulFun$ 表示漏洞函数，D_{Fun} 表示函数距离，$D_{Fun}=Djikstra(Fun,VulFun)$，集合 T 由可以直接到达漏洞函数的基本块构成。

利用式(3)计算出程序的基本块距离后，在模糊测试阶段，利用式(4)计算种子距离，即：

$$R=\frac{\sum_{BB\in Seed}D_{BB}}{Num(Seed)} \tag{4}$$

其中，$Seed$ 表示种子执行路径覆盖到基本块的集合，$Num(Seed)$ 表示集合 $Seed$ 中的元素数量。

通过式(5)对种子距离进行归一化处理，即：

$$Q=\frac{R-R_{\min}}{R_{\max}-R_{\min}} \tag{5}$$

其中，$R_{\min}$ 表示种子距离的最小值，$R_{\max}$ 表示种子距离的最大值。

4.2 种子能量动态调控模型

种子能量决定了种子的变异次数，当种子执行轨迹越接近漏洞函数时，种子能量越高，变异生成的测试用例到达漏洞函数的概率就越大。但当程序有多条可到达漏洞函数的路径时，种子距离计算模型可能会导致种子能量计算陷入局部最优，使生成的测试用例集中在最优路径周围，而无法覆盖其他到达漏洞函数的路径。

本节通过建立种子能量动态调控模型，来缓解种子距离计算模型在计算种子能量时的局部最优。能量动态调控模型引入执行轮次(t)作为敏感系数，以控制模型对种子距离的敏感程度，使得模型在测试初期对种子距离不敏感，但随着测试轮次的增加将提高对种子距离的敏感程度。模型的表达式如式(6)所示：

$$P=A*((1-\frac{1}{t})*\frac{1}{Q}+\frac{1}{t}) \tag{6}$$

其中，A 为常数系数，Q 为种子距离，t 为执行轮次。当 $t=1$ 时，$P=A$；当 t 趋近正无穷时，$P=A/Q$。由式(6)可知，当模糊测试器进行第一轮测试时，模型对种子距离不敏感，种子能量与种子距离无关；随着测试轮次的增加，模型对种子能量逐渐变得敏感，种子能量受种子距离的影响逐渐增大。图 6 给出了种子能量的变化曲线。当种子距离一定时，随着种子迭代次数的增加，种子能量趋于稳定，种子迭代次数对种子能量的影响逐渐下降。纵向对比各条曲线，当种子迭代次数相等时，种子距离越小，种子能量就越高。

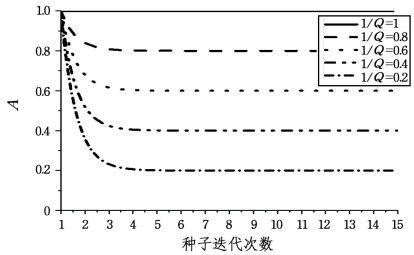

图 6 种子能量的变化曲线

Fig. 6 Seed energy curve

5 导向式符号执行

通过第 3 节的导向式模糊测试技术可以快速生成到达漏洞函数的测试用例，但不能保证一定会触发漏洞。本节对选择符号执行技术进行改进，以可到达漏洞函数的测试用例为驱动，快速生成可触发漏洞的测试用例。

选择符号执行技术会对符号执行区域内的路径进行遍历并求解。本文只希望对能够到达漏洞点的路径进行求解，因此对函数内的控制流进行分析并生成导向信息，使符号执行引擎只对函数内的部分路径进行遍历求解，以提高执行效率。

首先根据模糊测试阶段传入的函数调用图与控制流程图，对漏洞函数进行内联分析[15-16]。未进行内联分析前的流程图如图 7 所示，VulFun 为漏洞函数，漏洞函数内的基本块 BB_4 调用函数 CalledFun。

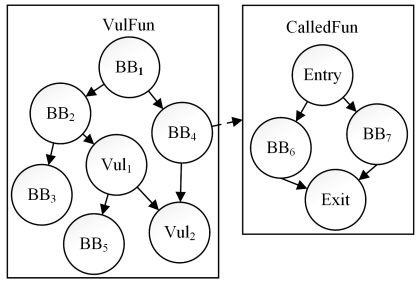

图 7 内联前的 VulFun 与 CalledFun 的控制流程图

Fig. 7 Control flowchart of VulFun and CalledFun before inlining

根据算法 1，将 VulFun 与 CalledFun 的控制流程图内联，合并为一个控制流程图。算法 1 以深度遍历的方式对漏洞函数的控制流程图中的每个节点进行判断，若该节点调用了其他函数，则对被调函数递归调用算法 1，递归结束后将被调函数的控制流程图内联到漏洞函数的控制流图中。通过算法 1 内联分析后得到的 VulFun 控制流程图如图 8 所示。

算法 1 控制流程图内联算法

输入：漏洞函数的控制流程图 CFG_VulFun

输出：漏洞函数内联控制流程图 CFG_Inline

```
1. Inline(CFG_VulFun){
2. While(True){
3.   BB=Depth first algorithm(CFG_VulFu)
4.   If(BB==Null){
5.     break}
6.   CFG_Inline->AddBB(BB)
7.   If(BB->HaveCallFun){
8.     Inline(BB->CallFun)//该基本块有函数调用，对被调用函数
       进行内联分析
9.     CFG_Inline->AddCFG(BB->CallFun)}
10.  Return CFG_Inline}
```

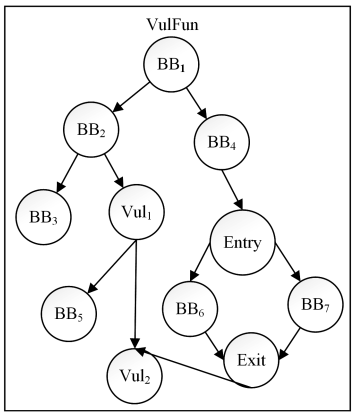

图 8 内联后的 VulFun 控制流程图

Fig. 8 Control flow chart after inline of VulFun

使用广度优先算法对内联后的控制流程图进行分析，分别求解漏洞点所在基本块的前驱基本块集合，并取其交集生成导向信息，如图 9 所示。

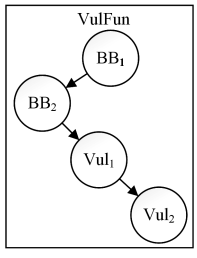

图 9 VulFun 导向信息

Fig. 9 Guidance information of VulFun

获得导向信息后，以导向式模糊测试提供的测试用例为驱动，导向式符号执行过程如图 10 所示。图 10 中，虚线表示控制流程图上存在而未执行的路径，实线表示测试用例执行的路径。在具体执行域中，程序严格按照导向式模糊测试提供的种子进行单路径具体执行。在符号执行域中，程序按照导向信息进行导向式符号执行，对部分路径进行覆盖。

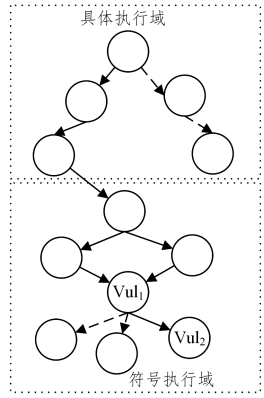

图 10 导向式符号执行

Fig. 10 Guided symbolic execution

6 实验及分析

6.1 原型系统设计与环境设置

为验证本文技术的有效性，基于模糊测试工具 AFL[17] 与符号执行平台 S2E[12]，设计并实现了原型系统 SymFuzz。原型系统使用 Angr 对程序进行分析并生成程序控制流图；使用 Python 的 networkx 包对控制流程图进行处理，计算程序基本块距离。SymFuzz 对 AFL 插桩模块与种子能量计算模块进行扩展，将基本块距离插桩到程序中，并在模糊测试的过程中计算种子距离与能量；对 S2E 平台的 S2EExecutor 模块进行扩展，使得符号执行引擎可以进行导向式符号执行。

实验环境的实验平台为 Ubuntu 16.04LTS 64 位操作系统，处理器为 Intel Core i7-8650U @1.90GHz，内存为 16GB。

6.2 实验验证

为验证本文技术的有效性，以近年公开的 CVE、美国国防部高级研究计划局（Defense Advanced Research Projects Agency）举办的网络安全挑战赛（Cyber Grand Challenge）中的部分测试集[18] 和部分 UAF 测试集为测试目标进行实验。其中，CVE-2018-7867 和 CVE-2019-12982 为 0.4.8 版本 libming 中的堆溢出漏洞，CVE-2018-8807 和 CVE-2018-8962 为 0.4.8 版本 libming 中的 UAF 漏洞，CVE-2020-6628 为 0.4.8 版本 libming 中的越界读漏洞。UAF 测试集为手工构造的数据集，在 UAF 漏洞的基础上加入复杂路径约束条件，以检测 SymFuzz 对复杂路径下漏洞的检测能力。实验以工具 AFL 和 AFLgo 为对比目标，设计了目标点覆盖、漏洞检测两组实验。

为验证本文对距离计算模型与种子能量调控模型改进的有效性，对 5 个 CVE 目标进行站点覆盖实验，实验进行了 5 次，每次实验 24 小时。表 2 列出了站点覆盖结果，其中，Reach 表示覆盖到目标站点的次数，Time 表示覆盖到目标站点花费的平均时间，Enhance 表示性能提升水平，AFLgo 的 Enhance 值为 SymFuzz 的 Time 与 AFLgo 的 Time 的商，AFL 的 Enhance 值为 SymFuzz 的到达时间与 AFL 的到达时间的商。由表 2 可知，SymFuzzz 对目标站点覆盖的速度最快。SymFuzzz 与没有导向性的模糊测试器 AFL 相比，到达目标站点的速度平均提高了 4.54 倍，与导向式模糊测试器 AFL-

go 相比，到达目标站点的速度平均提高了 1.87 倍。这表明本文改进的距离计算模型和动态种子调控模型，可以有效提高模糊测试器对复杂路径的探索能力。

表 2 站点覆盖

Table 2 Site coverage

CVE-ID	Tools	Reach	Time/s	Enhance
2018-7867	SymFuzz	5	17726	—
	AFLgo	5	30603	1.72
	AFL	3	63055	3.56
2018-8807	SymFuzz	5	29105	—
	AFLgo	3	43436	1.49
	AFL	2	67714	2.33
2018-8962	SymFuzz	5	14484	—
	AFLgo	5	17252	1.19
	AFL	4	45022	3.11
2019-12982	SymFuzz	5	434	—
	AFLgo	5	1563	3.60
	AFL	5	3674	8.47
2020-6628	SymFuzz	5	322	—
	AFLgo	5	436	1.35
	AFL	5	1684	5.23

为了进一步说明在导向模糊测试过程中引入符号执行技术对复杂路径条件漏洞挖掘的影响，在网络安全挑战赛测试集中选取 2 个目标并在 UAF 测试集中选取 3 个目标进行漏洞检测实验。漏洞检测实验目标选取的目标与目标站点覆盖实验选取的目标相比，路径数量较少但漏洞路径条件复杂，可以更好地检测原型系统对复杂路径条件下的漏洞检测性能。实验重复进行 5 次，每次实验 5 小时。表 3 列出了漏洞检测实验的结果。

表 3 漏洞检测

Table 3 Vulnerability detection

Program	Tools	Reach	Time/s	Enhance
UAF_2018_316	SymFuzz	5	168	—
	AFLgo	5	13358	79.51
	AFL	5	14040	83.57
UAF_2018_504	SymFuzz	5	340	—
	AFLgo	5	14214	41.80
	AFL	5	14650	43.088
UAF_2018_703	SymFuzz	5	471	—
	AFLgo	5	11873	25.20
	AFL	5	12036	25.55
CGC_Pwn_01	SymFuzz	5	434	—
	AFLgo	0	—	—
	AFL	0	—	—
CGC_Pwn_02	SymFuzz	5	322	—
	AFLgo	0	—	—
	AFL	0	—	—

在漏洞检测实验中，SymFuzz 可以对 5 个测试程序实现快速检测，而 AFLgo 与 AFL 只能对其中 3 个程序完成检测。通过对程序的分析发现，测试用例需要检测到程序 CGC_Pwn_01 和 CGC_Pwn_02 中的漏洞，需要通过复杂的数学变化，AFL 和 AFLgo 生成的测试用例始终无法通过该变化，导致 AFL 和 AFLgo 无法检测到漏洞。对 SymFuzz，AFLgo 与 AFL 在检测过程中所花费的时间进行更细粒度的划分，如表 4 所列。$T1$ 表示测试用例到达函数入口花费的时间，$T2$ 表示模糊测试生成的测试用例到达漏洞函数后触发漏洞的时间，T 为模糊测试器触发漏洞花费的总时间。

表 4 漏洞检测阶段的时间花费

Table 4 Time spent in vulnerability detection phase

(单位：s)

Program	Tools	$T1$	$T2$	T
UAF_2018_316	Symfuzz	23	145	168
	Aflgo	32	13326	13358
	Afl	57	13983	14040
UAF_2018_504	Symfuzz	31	309	340
	Aflgo	53	14161	14214
	Afl	64	14586	14650
UAF_2018_703	Symfuzz	98	373	471
	Aflgo	161	11712	11873
	Afl	194	11842	12036
CGC_Pwn_01	Symfuzz	41	393	434
	Aflgo	43	—	—
	Afl	23	—	—
CGC_Pwn_02	Symfuzz	64	258	322
	Aflgo	61	—	—
	Afl	55	—	—

由表 4 可知，每组实验 $T1$ 阶段的时间差异与其 $T2$ 阶段的时间差异相比并不明显，在 T 中起决定性作用的是 $T2$ 阶段所花费的时间。以 UAF_2018_316 为例，SymFuzz，AFLgo 与 AFL 在 $T1$ 阶段的时间花销分别为 23 s，32 s 和 57 s，在 T2 阶段的时间花销分别为 145 s，13 326 s 和 13 983 s，这表明 SymFuzz 使用符号执行技术大幅减少了在 $T2$ 阶段花费的时间，由此可以看出在复杂路径条件下符号执行技术对于漏洞检测起到了良好的效果。

此外，在对 CGC_Pwn_01 和 CGC_Pwn_02 的漏洞检测过程中，AFL 在 $T1$ 阶段花费的时间小于 AFLgo 和 SymFuzz 的。这是因为 CGC_Pwn_01 和 CGC_Pwn_02 漏洞函数所在位置距离主函数极近，所以不需要进行导向式模糊测试就可以实现快速覆盖，AFLgo 和 SymFuzz 进行导向式模糊测试时引入了时间开销，反而降低了测试速度。

由上述实验结果可知，相比传统的模糊测试技术，SymFuzz 可以实现对漏洞函数的快速覆盖，并对复杂路径条件下的漏洞实现快速检测。这证明 SymFuzz 通过改进种子距离模型与能量动态调控模型并引入符号执行技术的方式，实现了在复杂路径条件下的漏洞快速检测。

结束语 本文对导向式模糊测试技术与选择符号执行技术进行研究，综合两种技术的优势，提出了一种复杂路径条件下的漏洞检测技术。该技术改进了距离计算模型与种子能量计算模型，对符号执行路径空间进行约束，使得该技术可以实现复杂路径条件下的快速漏洞检测。但该方法对静态分析的准确性要求较高，目前的静态分析工具还无法有效解决控制流程图中间接调用的问题，影响了分析的准确性。

在下一步的工作中，将着重解决静态分析过程中间接调用的问题，并在符号执行过程中引入漏洞检测模型以进一步提高符号执行的速度，减小系统开销。

参 考 文 献

[1] WU S Z, GUO T, DONG G W, et al. Advances in software vulnerability analysis techniques [J]. Journal of Tsinghua University (Natural Science Edition), 2012, 52(10): 1309-1319.

[2] BARTON P M, LOUIS F, BRYAN S. An Empirical Study of the

Reliability of UNIX Utilities[J]. Communications of the ACM, 1990,33(12):32-44.

[3] GODEFROID P,LEVIN M Y,MOLNAR D. SAGE:whitebox fuzzing for security testing[J]. Communications of the ACM, 2012,55(3):40-44.

[4] LI J,ZHAO B,ZHANG C. Fuzzing:a survey[J]. Cybersecurity, 2018,1(1):6.

[5] MANÈS V J M,HAN H S,HAN C,et al. The art,science,and engineering of fuzzing:A survey[J]. IEEE Transactions on Software Engineering,2019,arXiv:1812.00140.

[6] REN Y Z,ZHANG Y W,AI C W. Review of Stain Analysis Technology Research [J]. Journal of Computer Applications. 2019,39(8):2302-2309.

[7] CHEN J M,SHU H,XIONGX B. Fuzzing test method based on symbolic execution [J]. Computer Engineering,2009,35(21): 33-35.

[8] ZOU Q C,ZHANG T,WU R P,et al. From automation to intelligence: software vulnerabilities mining technology progress [J]. Journal of Tsinghua University (Science and Technology), 2018,58(12):1079-1094.

[9] BÖHME M,PHAM V T,NGUYEN M D,et al. Directed Greybox Fuzzing[C]// Acm Sigsac Conference on Computer & Communications Security. 2017:2329-2344.

[10] CHEN H,XUE Y,LI Y,et al. Hawkeye:Towards a desired directed grey-box fuzzer[C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018:2095-2108.

[11] CHIPOUNOV V,KUZNETSOV V,CANDEA G. S2E:A platorm for invivo multipath analysis of software systems [J]. ACM SIGPLAN Notices,2011,47(4):265-278.

[12] CHIPOUNOV V,GEORGESCU V,ZAMFIR C,et al. Selective symbolic execution[C]// The Workshop on Hot Topics in System Dependability. 2009:1286-1299.

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

[14] 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.

[15] MEHLHORN K. Data structures and algorithms:1. Searching and sorting [J]. Springer,1984,84:90.

[16] HUANG H,LU Y L,LIU L T,et al. Research on the symbolic execution technology of control flow stain information[J]. Journal of University of Science and Technology of China, 2016, 46(1):21-27.

[17] ZALEWSKIM. American Fuzzy Lop [OL]. http://lcamtuf.coredump.cx/afl/.

[18] DARPA. DARPA cyber grand challenge [EB/OL]. [2017-02-01]. https://github.com/CyberGrandChallenge.

LI Ming-lei, born in 1996, master's degree. His main research interests include cyberspace security, binary software analysis and program vulnerability mining and analysis.

LU Yu-liang, born in 1964, professor, Ph.D supervisor. His main research interests include cyberspace security, vulnerability mining and utilization and network situational awareness.