软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(5):1258−1274 [doi: 10.13328/j.cnki.jos.005499]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

可编程模糊测试技术*

杨梅芳[1,2,3,4], 霍 玮[1,2,3,4], 邹燕燕[1,2,3,4], 尹嘉伟[1,2,3,4], 刘宝旭[1,2,3,4], 龚晓锐[1,2,3,4], 贾晓启[1,2,3,4], 邹 维[1,2,3,4]

[1](中国科学院 信息工程研究所,北京 100195)

[2](中国科学院 网络测评技术重点实验室(中国科学院 信息工程研究所),北京 100195)

[3](网络安全防护技术北京市重点实验室(中国科学院 信息工程研究所),北京 100195)

[4](中国科学院大学 网络空间安全学院,北京 100049)

通讯作者: 邹燕燕, E-mail: zouyanyan@iie.ac.cn

摘 要: 模糊测试是一种有效的漏洞挖掘技术.为了改善模糊测试因盲目变异而导致的效率低下的问题,需要围绕输入特征、变异策略、种子样本筛选、异常样本发现与分析等方面不断定制模糊测试器,从而花费了大量的定制成本.针对通用型模糊测试器(即支持多类输入格式及目标软件的模糊测试器)的低成本定制和高可扩展性需求,提出了一种可编程模糊测试框架,基于该框架,漏洞挖掘人员仅需编写模糊测试制导程序即可完成定制化模糊测试,在不降低模糊测试效果的基础上,可大幅提高模糊测试器开发效率.该框架包含一组涉及变异、监控、反馈等环节的模糊测试原语,作为制导程序的基本语句;还包含一套编程规范(FDS)及 FDS 解析器,支持制导程序的编写、解析和模糊测试器的生成.基于实现的可编程模糊测试框架原型 Puzzer,在 26 个模糊测试原语的支持下,漏洞挖掘人员平均编写 54 行代码即可实现当前主流的 5 款万级代码模糊测试器的核心功能,并可覆盖总计 87.8%的基本操作.基于 Puzzer 实现的 AFL 等价模糊测试器,仅用 51 行代码即可达到与 AFL 相当的模糊测试效果,具有良好的有效性.

关键词: 模糊测试;漏洞挖掘;可编程;制导程序;抽象语法树

中图法分类号: TP311

中文引用格式: 杨梅芳,霍玮,邹燕燕,尹嘉伟,刘宝旭,龚晓锐,贾晓启,邹维.可编程模糊测试技术.软件学报,2018,29(5):1258−1274. http://www.jos.org.cn/1000-9825/5499.htm

英文引用格式: Yang MF, Huo W, Zou YY, Yin JW, Liu BX, Gong XR, Jia XQ, Zou W. Programmable fuzzing technology. Ruan Jian Xue Bao/Journal of Software, 2018,29(5):1258−1274 (in Chinese). http://www.jos.org.cn/1000-9825/5499.htm

Programmable Fuzzing Technology

YANG Mei-Fang[1,2,3,4], HUO Wei[1,2,3,4], ZOU Yan-Yan[1,2,3,4], YIN Jia-Wei[1,2,3,4], LIU Bao-Xu[1,2,3,4], GONG Xiao-Rui[1,2,3,4], JIA Xiao-Qi[1,2,3,4], ZOU Wei[1,2,3,4]

[1](Institute of Information Engineering, The Chinese Academy of Sciences, Beijing 100195, China)

[2](Key Laboratory of Network Assessment Technology (Institute of Information Engineering, The Chinese Academy of Science), The Chinese Academy of Sciences, Beijing 100195, China)

* 基金项目: 中国科学院网络测评技术重点实验室资助项目; 网络安全防护技术北京市重点实验室资助项目; 中国科学院重点实验室基金(CXJJ-17S049); 国家重点研发计划(2016QY071405)

Foundation item: Program of Key Laboratory of Network Assessment Technology, the Chinese Academy of Sciences; Program of Beijing Key Laboratory of Network Security and Protection Technology; Foundation of Key Laboratory of Network Assessment Technology, the Chinese Academy of Sciences (CXJJ-17S049); National Key Research and Development Program of China (2016QY071405)

本文由软件安全漏洞检测专题特约编辑王林章教授、陈恺研究员、王戟教授推荐.

收稿时间: 2017-07-01; 修改时间: 2017-08-29; 采用时间: 2017-11-21; jos 在线出版时间: 2018-01-09

CNKI 网络优先出版: 2018-01-11 17:24:46, http://kns.cnki.net/kcms/detail/11.2560.TP.20180111.1724.009.html

[3](Beijing Key Laboratory of Network Security and Protection Technology (Institute of Information Engineering, The Chinese Academy of Sciences), Beijing 100195, China)

[4](School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Fuzzing is an effective vulnerability discovery technology. In order to solve the inefficiency problem caused by blind mutation in fuzzing, safety engineers need to customize fuzzer from all aspects, such as input characteristics, mutation method, seed samples screening, abnormal samples found and analysis, which will result in huge expenditure. To meet the need of low cost customization and high scalability of the universal fuzzer (i.e. fuzzer that supports multi-type input formats and softwares), this paper first proposes a programmable fuzzing framework. Based on the framework, the only thing safety engineers need to do is writing directive programs when they want to customize fuzzing. It can sharply improve the efficiency of developing fuzzer without reducing effectiveness of fuzzing. The framework contains a set of fuzzing primitives, fuzzing directive specification (FDS) and FDS parser. Fuzzing primitives which involve mutation, monitoring and guiding are basic statements of directive program. FDS and FDS parser can support writing and parsing directive programs, as well as generating fuzzers. Based on the implementation of a prototype framework called Puzzer, safety engineers can accomplish core functions and cover 87.8% of total basic operations of five mainstream fuzzers with only about 54 lines of code. A fuzzer which has equivalent function of AFL can be accomplished using Puzzer to achieve the same effectiveness with only 51 lines of code.

Key words: fuzzing; vulnerability discovery; programmable; directive program; abstract syntax tree

模糊测试是一种通过向目标系统或软件提供非预期的输入并监视异常结果来发现软件漏洞的方法,由于模糊测试技术可以将大量的手工测试转化为高度的自动化测试[1],因此该技术获得了漏洞挖掘人员的广泛使用,仅 Github 上与模糊测试器(关键词“fuzzer”)相关的项目超过 1 500 个.由于其自动化程度高、缺陷可重现等优点,该技术被广泛应用于网络协议、浏览器、操作系统内核、图像处理、音视频文件解析等各类软件的漏洞挖掘工作中,并产生了良好的效果.使用模糊测试器 AFL[2](american fuzzy lop),发现了 143 个软件中的超过 400 个漏洞;Google 推出的 OSS-Fuzz[3]项目自建立以来,发现并指导开发者修复开源软件中超过 1 000 个缺陷.随着模糊测试技术的普遍应用及测试目标类型的多元化,模糊测试器的种类和数量也在不断增加.为了能够对特定目标程序进行高效的模糊测试,漏洞挖掘人员不断定制全新的模糊测试器.

传统的模糊测试技术由于盲目变异,存在测试效率低下的问题.近年来,研究人员围绕程序特征[4−9]、变异策略[10,11]、种子样本筛选[12−15]、异常样本发现与分析[16,17]等方面完善并定制模糊测试器(fuzzer),提高测试效率.Peach(2.3)[18]为了支持目标程序运行中的弹出窗口,在 Windows 平台引入 Popup Watcher;AFL 使用了代码覆盖率反馈的方式指导种子样本的筛选,从而使模糊数据覆盖目标程序的深层逻辑;Honggfuzz[19]添加了基于 CPU 指令的硬件反馈方法提升反馈效率.

当前,定制模糊测试器、扩展模糊测试器功能需要较高的开发成本,特别是模糊测试器不合理的架构设计,导致了即使仅增加“小”功能也需要“从头”实现模糊测试器,这进一步增加了开发成本.如何使模糊测试器能够快速定制、易于扩展、功能共享,从而达到降低开发成本、提高开发效率的目标?漏洞挖掘人员对于上述问题一直缺少相关研究.

本文首次尝试解决该问题,并作为第一步探索,主要围绕通用型黑/灰盒模糊测试器的快速定制方法开展研究.通用型模糊测试器是指支持多类输入格式及目标软件的模糊测试器,如 AFL,Peach,Driller[6]等,是使用最为广泛的一类模糊测试器,也是定制最为活跃的一类模糊测试器,其分类见表 1.

Table 1 Classification of universal fuzzers

表 1 通用型模糊测试器分类

	基于变异的模糊数据生成	基于生成的模糊数据生成
黑盒	Taof, Zzuf, Radamsa	Peach, Sulley, kitty
灰盒	AFL, Libfuzzer, Honggfuzz	Choronzon, Tavor, Fuddly
白盒	Vuzzer, Sage, Driller	

本文主要针对表 1 所示的通用型黑盒及灰盒类模糊测试器提出了一种可编程模糊测试框架,基于该框架,

漏洞挖掘人员仅需编写模糊测试制导程序即可完成定制化模糊测试,在不降低模糊测试效果的基础上,可大幅提高模糊测试器开发效率.该框架包含一组涉及变异、监控、反馈等环节的模糊测试原语,作为制导程序的基本语句;还包含一套编程规范(fuzzer directive specification,简称 FDS)及 FDS 解析器,支持制导程序的编写、解析和模糊测试器的生成,有效降低了开发成本.本文的贡献如下:

(1) 提供一种可编程模糊测试框架,支持通用型黑/灰盒模糊测试器的全流程、快速、定制化的构建;

(2) 归纳、抽象模糊测试基本环节,构建模糊测试原语库;同时,支持漏洞挖掘人员通过可编程接口实现模糊测试原语库的扩展;

(3) 设计了模糊测试制导程序的编程规范及 FDS 解析器,可在解析漏洞挖掘人员编写的模糊测试制导程序的同时支持对制导程序的容错和调试;支持对模糊测试全流程的监控,辅助漏洞挖掘人员优化模糊测试器.

本文第 1 节分析现有模糊测试器在构建、扩展等方面存在的问题.第 2 节在归纳总结现有模糊测试器设计、实现的基础上,阐述可编程模糊测试技术.第 3 节介绍原型框架的实现.第 4 节通过实验对原型工具 Puzzer 进行评估.第 5 节给出本文的总结.

1 研究动机

通过对 3 个典型模糊测试器以及 Github 中 49 个高评价的模糊测试器的调研分析发现:尽管大部分模糊测试器实现复杂,但其基本工作流程和功能的相似度较高,存在较多共性.同时,由于功能扩展及适应性需求,需要对模糊测试器进行定制而导致全新开发.这将导致大量开发成本重复投入或无效投入,亟需建立统一且易于扩展的框架,支持模糊测试器的开发和定制.

1.1 模糊测试器复杂度高

本文选取了 20 款典型模糊测试器,包括 Peach(2.3)等商用模糊测试器、AFL 等开源典型模糊测试器以及 Honggfuzz 等 Github 上热门模糊测试器(项目好评度大于 100),覆盖文件解析、网络协议、浏览器、操作系统内核等 4 大类目标程序的模糊测试能力,统计其代码行数(使用“wc”命令,包含空行,不包括头文件),统计发现全部模糊测试器代码行数均在 2 000 行以上,其中有 9 款模糊测试器的代码量超过万行(如图 1 所示).

Fig.1 LOC of fuzzers

图 1 模糊测试器代码行数

1.2 模糊测试器相似度高

为了使模糊测试器支持不同的目标程序,各式各样的模糊测试器层出不穷.它们基于已有的模糊测试器进行功能增加,模糊测试器的功能增加包含两个方面:一是增加模糊测试流程的测试步骤,如 Choronzon[20]比 Peach 增加了反馈步骤;二是改进模糊测试器已有模块的功能,如 Choronzon 在 Peach 已有的模糊数据生成方法基础上进行了优化,提出了基于数据模板“重组”的模糊数据生成方式.Peach 在模糊测试流程上的固定,导致无法以较小的开销添加反馈步骤或改进模糊数据生成方法,Choronzon 重新实现了超过 4 000 行的代码重写已有功能以及增加新的功能.Kitty 在 Sulley[21]的基础上进行模块化,支持使用 python-pip 进行安装,同时增加了对

OS X 的监控支持,这使得 Kitty 重新实现了超过 10 000 行的代码.上述模糊测试器的对比见表 2.

Table 2 Comparison 1 of universal fuzzers

表 2 通用型模糊测试器比较 1

	编程语言	数据模板	模糊数据生成		监控	反馈
Kitty	Python	无明显差异	二者一致		支持Linux,Windows,OS X平台	二者均无反馈
Sulley	Python				支持Linux,Windows平台	
Choronzon	Python	无明显差异	无明显差异	提出重组器	支持Linux,Windows平台	基于Intel Pin
Peach(2.3.7)	Python			无重组器	支持Linux,Windows,OS X平台	无反馈

1.3 定制模糊测试器成本高

综上所述,由于现有的模糊测试器的扩展开销大、测试流程固化等问题,导致了定制模糊测试器的成本巨大.以模糊测试器的平台适应的定制为例,模糊测试器的实现与其支持的目标程序的操作系统平台息息相关.操作系统平台的限制,使得不同平台的模糊测试器在监控、反馈等步骤的实现上有巨大差异.以 AFL 与 WinAFL 为例,WinAFL 的目的是将 AFL 的模糊测试思想用于 Windows 平台,但 AFL 的设计中未考虑 Windows 平台的兼容性,使得 WinAFL 使用了超过 9 000 行的代码实现了 AFL 在 Windows 平台上的支持.同样的情况也出现在 NaFl 中,NaFl 与 WinAFL 功能相似,仅将反馈部分的插桩工具更改为 Pin,变异方法略有区别,这使得 NaFl 重新实现了超过 2 000 行的代码.Honggfuzz 重新实现了超过 12 000 行的代码.上述模糊测试器的对比见表 3.

Table 3 Comparison 2 of universal fuzzers

表 3 通用型模糊测试比较 2

	编程语言	变异策略	模糊数据生成	监控	反馈
NaFl	Python	无确定性变异策略	仅含 6 种方法	支持Windows平台	基于动态插桩Pin
WinAFL	C/C++	有确定性变异策略	无明显差异	支持Windows平台	基于动态插桩DynamoRIO
AFL				支持Linux平台	基于编译时插桩afl-gcc;qemu
Honggfuzz		无确定性变异策略		支持Linux,Windows,OS X平台	基于编译时插桩SANCOV;硬件反馈

1.4 可编程模糊测试框架

基于以上分析,为了从快速构建契合目标程序的模糊测试器这一新的角度来实现模糊测试器的全流程定制,本文提出了可编程模糊测试技术并实现可编程模糊测试框架,使用该框架可实现以下功能.

(1) 针对通用型黑/灰盒模糊测试器,基于制导程序实现模糊测试流程中各个环节间的全流程、快速、定制化构建;

(2) 基于可编程接口,实现模糊测试原语库的扩展;

(3) 基于 FDS 解析器,支持制导程序的容错和调试,实现模糊测试全流程的监控,辅助漏洞挖掘人员优化模糊测试器.

2 可编程模糊测试技术

2.1 方法概述

传统的模糊测试器构建方式如图 2(a)所示,漏洞挖掘人员使用 C,C++,Python 等编程语言实现模糊测试器.采用传统方式,由于测试需求的不同,漏洞挖掘人员需要花费大量的精力在模糊测试器代码编写以及先进方法集成的过程中.而本文所提出的可编程模糊测试技术构建方式如图 2(b)所示,漏洞挖掘人员通过模糊测试器制导程序编程规范编写程序,使用 FDS 解析器(FDS parser,简称 FDSP)解析程序,最终快速生成基于 Python 代码的模糊测试器.此外,该技术还支持对模糊测试技术方法的灵活、便捷扩展.

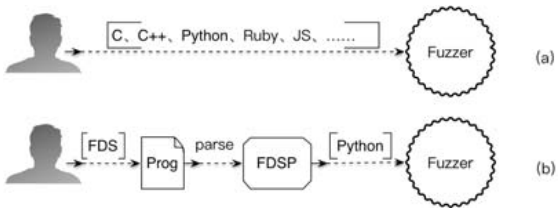

Fig.2 Traditional method vs. programmable fuzzing technology

图 2 传统方法与可编程模糊测试技术

可编程模糊测试技术对传统的模糊测试实现过程进行高层次抽象,将模糊测试各环节抽象为不同的模糊测试原语(fuzzing primitive,简称 FP),使用 FP 编写符合 FDS 规范的程序以达到降低编写模糊测试器的门槛、加快构建速度以及实现技术方法扩展集成的目的.本文首先提出了一种基于模糊测试原语的模糊测试器编写语法规范,使用该规范编写出制导程序(prog);然后,通过 FDS 解析器中的预处理器(preprocessor)对制导程序进行解析,生成模糊测试抽象语法树(fuzzing abstract syntax tree,简称 FAST);模糊测试引擎(fuzzing engine,简称 FE)通过遍历语法树并与模糊测试原语库(fuzzing primitives library)进行交互,最终生成一个基于 Python 语言的模糊测试器.该过程如图 3 所示.

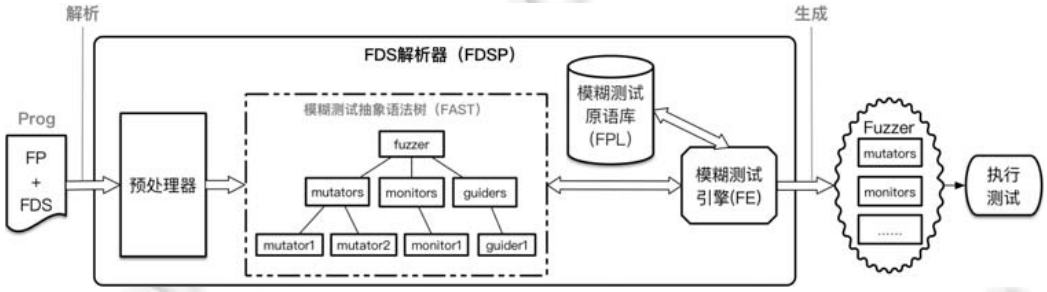

Fig.3 Architecture of programmable fuzzing technology

图 3 可编程模糊测试技术架构

Prog 中,每个模糊测试原语都对应着最终生成的模糊测试器中的一个代码块,如图 4 所示,FlipMutator 原语将对应 FlipMutator 类的实现,该类用于实现位翻转变异;FlipMutator 原语参数 pos 以及 step 将传递给 FlipMutator 类,控制位翻转变异的起始位置以及步长.

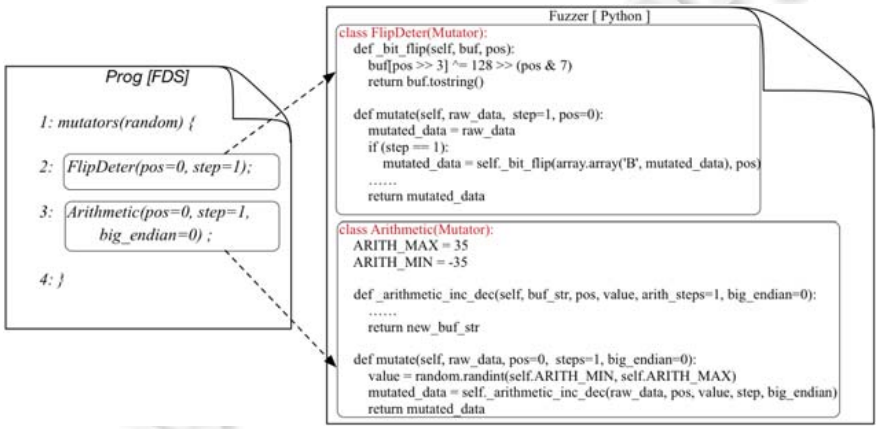

Fig.4 Covert prog to fuzzer

图 4 Prog 到 Fuzzer 的转化

下文将从 3 个部分来阐述可编程模糊测试技术:一是模糊测试原语及语法规范的提出;二是 FDS 解析器的设计;三是模糊测试原语库中基础原语的设计.

2.2 语法规范及模糊测试原语

本文提出的 FDS 规范将从形式化的角度对模糊测试器的内部实现进行描述.通过对 50 余个主流开源模糊测试器的分析,并结合传统模糊测试实现流程,制导程序由变异、监控、反馈这 3 个部分组成.

制导程序 *Prog* 可以用公式(1)进行表示:

$$Prog:=Mu^{+}Mo^{+}Gu^{*} \tag{1}$$

其中,*Prog* 为制导程序;*Mu* 表示变异部分代码块,是 *Prog* 中必须实现的部分;*Mo* 表示监控部分的代码块,是 *Prog* 中必须实现的部分;*Gu* 表示反馈部分代码块,是 *Prog* 中可选部分.各部分语法规范见表 4,其中,斜体字符表示非终结符,非斜体字符表示终结符.

Table 4 Fuzzer directive program specification

表 4 模糊测试器制导程序编程规范

	变异	监控	反馈

规范	*Mu*:=*mutators*(*Type*) {
	Body
	};
	Type:=*random*|*determine*
	Body:=FP_{Mu};*Body*|*Body* FP_{Mu};|ε
	FP_{Mu}:=*pName*(*pArgs*)
		Mo:=*monitors*() {
		Body
		};
		Body:=FP_{Mo};*Body*|*Body* FP_{Mo};|ε
		FP_{Mo}:=*pName*(*pArgs*)
			Gu:=*guiders*() {
			Body
			};
			Body:=FP_{Gu};*Body*|*Body* FP_{Gu};|ε
			FP_{Mu}:=*pName*(*pArgs*)

释义	*Type*:变异方法选取策略 FP_{Mu}:变异原语 *pName*={*fp*.*pName*|*fp*∈*FPS* & *fp*.*pCls*=*MUTATOR*}, 其中,*fp*.*pCls* 表示原语 *fp* 的 *pCls* 属性, *fp*.*pName* 表示原语 *fp* 的 *pName* 属性, *FPS* 表示模糊测试原语集合	FP_{Mo}:监控原语 *pName*={*fp*.*pName*|*fp*∈*FPS* & *fp*.*pCls*=*MONITOR*}, 其中,*fp*.*pCls* 表示原语 *fp* 的 *pCls* 属性, *fp*.*pName* 表示原语 *fp* 的 *pName* 属性, *FPS* 表示模糊测试原语集合	FP_{Gu}:反馈原语 *pName*={*fp*.*pName*|*fp*∈*FPS* & *fp*.*pCls*=*GUIDER*}, 其中,*fp*.*pCls* 表示原语 *fp* 的 *pCls* 属性, *fp*.*pName* 表示原语 *fp* 的 *pName* 属性, *FPS* 表示模糊测试原语集合

本文通过将模糊测试各个环节归纳为基本操作以及基本操作类,进而抽象出模糊测试原语.

定义 1(基本操作). 基本操作是模糊测试系统变异、监控、反馈环节中所选用的各个技术方法的具体实现.

定义 2(基本操作类). 基本操作类是实现同一功能的基本操作的集合.

定义 3(模糊测试原语). 模糊测试原语(fuzzing primitive,简称 FP)与基本操作类一一映射,是一个基本操作类所代表的模糊测试技术方法的抽象及封装.本文将模糊测试原语表示成一个由原语名称 *pName*、原语类别 *pCls* 以及原语参数 *pArgs* 这 3 个属性构成的三元组:

$$FP:\langle pName,pCls,pArgs\rangle,$$

其中,*pName* 用于模糊测试测试原语的编程调用,支持使用者的自定义;*pCls* 标记了原语所属类别,一个原语属于且仅属于一个类别,*pCls*={*MUTATOR*,*MONITOR*,*GUIDER*};*pArgs* 为原语参数键值集合(可为空集),其中可以包含 0 个或多个参数键值,参数键值使用二元组⟨*key*,*value*⟩进行表示.

2.3 FDS解析器

FDS 解析器(FDSP)中包含 3 个组件:预处理器、模糊测试引擎以及模糊测试原语库.为了能够将基于 FDS 的制导程序 Prog 映射到基于 Python 语言的模糊测试器,并实现对 Prog 的语法检查以及无效语句过滤,本文中使用模糊测试抽象语法树作为中间表示.FDS 解析器的 3 个组件相互调用实现以下 3 个过程:(1) 通过预处理器解析基于 FDS 编写的制导程序,生成模糊测试抽象语法树;(2) 通过模糊测试器引擎遍历语法树,依据语法树在原语库中匹配模糊测试原语;(3) 通过模糊测试引擎模块化整合原语方法,生成基于 Python 语言的定制模糊测试器.

2.3.1 预处理器

预处理器基础的功能为完成制导程序 Prog 到模糊测试语法树的转换,如图 5 所示.模糊测试原语位于语法

树的叶子节点,而非叶子节点由 FDS 规范中的语法组成.一个模糊测试器可以划分为变异、监控、反馈等模块,每个模块中包含相应的原语,语法树实质表征着一个模糊测试器.

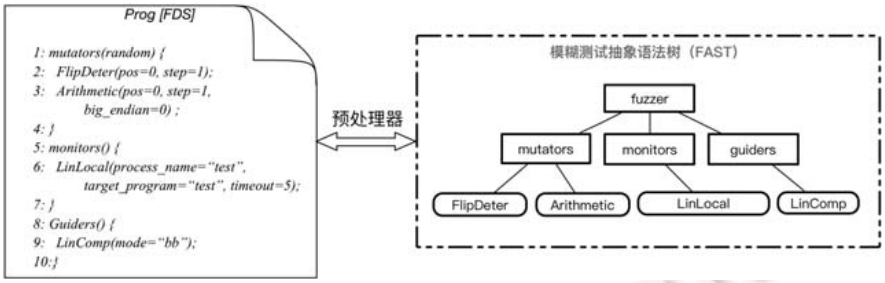

Fig.5 Prog converts to FAST

图 5 Prog 转换为 FAST

预处理器还支持对指导程序的查错、容错.查错主要对制导程序的编程规范进行检查,每个原语的调用是否符合制导程序编程规范.在符合编程规范的基础上,容错用于处理同一功能的不同原语多次调用的情况.

2.3.2 模糊测试引擎

本文中,将针对不同类型目标程序的测试架构抽象为模糊测试引擎,通过不同引擎来支持不同类型目标程序的测试.模糊测试原语库是全部模糊测试原语实例的集合,原语实例即是原语所代表的模糊测试技术方法的具体实现.模糊测试引擎将遍历语法树,并调用模糊测试原语库中的原语实例构建定制的模糊测试器.如图 6 所示,从树的根节点到叶子节点实质上是一个对模糊测试器不断拆分细化的过程.首先使用深度优先的遍历方式,根据对制导程序的预处理结果初始化对应类别的模块,直到遇到原语叶子节点,获取原语叶子节点的原语方法参数(pArgs),初始化原语;然后,通过从原语库中匹配相应原语实例,调用原语方法接口,返回所有完整的原语以及模糊测试器的各模块;最终,将生成的模块集合返回模糊测试引擎,构建完整的模糊测试器.

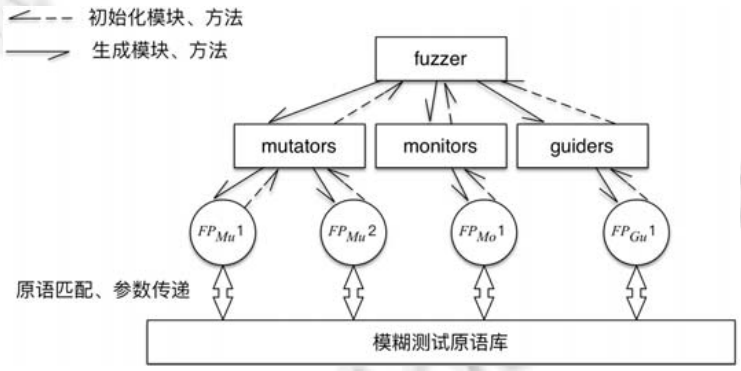

Fig.6 Parse of fuzzing abstract syntax tree

图 6 模糊测试语法树的解析

由于在解析抽象语法树的过程中涉及各模块的调度,对于不同类型的测试目标,模糊测试过程中各模块的调度方法不同.如测试协议的时候需要有协议状态控制、协议格式描述,而测试输入为文件的目标程序时则不关注诸如此类的描述.根据 Github 的好评度从高到低进行排序,加之现下使用广泛但未在 Github 上面公开的开源模糊测试器,本文从中选取了 52 个开源模糊测试器,按照模糊测试器测试的目标程序类型的不同,可将模糊测试器分为文件类、协议类、浏览器类、库(API、内核)、生成测试用例、测试特定语言、测试命令行、其他.文件类、协议类的模糊测试器占据总数的 52%,本文设计了 3 种模糊测试引擎来支持可编程模糊测试技术在这

些类别软件的模糊测试中的应用,分别为描述类测试引擎、非描述类测试引擎以及状态控制引擎.在之后的工作中,还将扩展其他测试引擎来支持其他类别软件的模糊测试需求.

2.3.3 模糊测试原语库

模糊测试原语库中存储了一系列模糊测试原语,其中封装了每个原语的具体实现,其中包含本文归纳的基础原语的实现.基础原语的具体设计见第 2.4 节.同时,原语库可通过可编程原语接口进行自定义扩展.模糊测试引擎通过匹配每个原因的唯一标识——原语名称 *pName*.每个原语有一个调试日志生成器与之相匹配,模糊测试引擎在原语库中识别原语,并将当前输入至原语的相关信息记录.调试日志生成器会记录原语运行过程中的中间信息以及运行结束后的返回信息,将中间信息以及返回信息进行过滤、合并,生成调试信息反馈至模糊测试引擎中的全局日志记录器.最后,由全局日志记录器负责输出至 stdout 或是 log 文件.全流程调试日志生成的原理如图 7 所示.

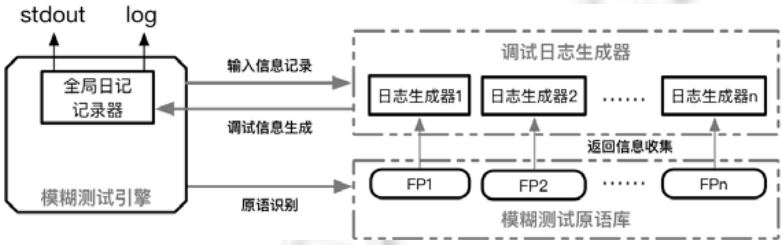

Fig.7 Full process debug log generation

图 7 全流程调试日志生成

2.4 基础原语设计

和程序设计语言中的基本类型同样重要,可编程模糊测试技术中的基础原语的设计直接影响使用该技术生成的模糊测试器所能支持的目标程序的广泛度以及构建一个模糊测试器所需的代码量.本文从通用性、完整性的角度来设计各模块所必须的基础原语.基础原语的定义将支持漏洞挖掘人员为目标程序编写定制的模糊测试器.同时,由于测试需求的不同,模糊测试基础原语的定义无法兼顾所有可能的模糊测试场景或需求.为了实现对不同场景或需求的支持,本文为各种类别的原语设计了自定义接口,从而实现漏洞挖掘人员根据实际情况自定义模糊测试原语,实现扩展模糊测试的技术方法.

2.4.1 变异基础原语

本文对上述测试目标为文件类、协议类、浏览器类以及 API 类的 38 款模糊测试器所使用的变异方法进行详细的分析统计,其中涉及了 32 种变异基本操作,根据每种方法的特性及作用效果,本文将这 32 种基本操作归纳为 16 种变异原语,归纳出的变异原语如下:确定长度翻转(FlipDeter)、随机长度翻转(FlipRand)、任意字节加减(Arithmetic)、数字字节加减(ArithmeticDigit)、随机值替换(ReplaceRand)、特殊值替换(ReplaceSpec)、随机值插入(InsertRand)、特殊值插入(InsertSpec)、随机长度删除(DeleteRand)、确定长度删除(DeleteDeter)、置乱(Shuffle)、交换(Swap)、重置大小(ChangeSize)、行变换(ChangeLine)、重复(Repeat)、拼接(Splicing).

可编程模糊测试技术将基于上述 16 种变异原语进行定义与描述见表 5.

模糊数据生成方法在很大程度上影响着模糊测试的效率.

- 一方面,对于对输入的有效性进行严格检查的目标程序,盲目变异将无法通过格式检查,导致程序过早的走向错误分支,致使模糊测试无法覆盖目标程序的深层逻辑.对于上述情形,基于数据模板的模糊数据生成方法将体现其优势,其通过数据模板对数据格式进行详细描述,以此通过目标程序对数据格式的检查,使输入的模糊数据能更深的覆盖到目标程序逻辑.
- 另一方面,对于难以获取数据模板的目标程序,模糊测试器也需要支持基于变异的“黑盒”模糊数据生成方式,以达到对难以获取数据模板的目标程序的支持.

变异原语通过原语参数来实现对上述两种方式的支持,依托可编程模糊测试技术中的原语参数传递,实现对变异过程的灵活控制.

Table 5 Mutation primitives

表 5 变异原语

pName	pArgs 及释义			
	Arg1	Arg2	Arg3	Arg4
FlipDeter	pos:起始位置	step:步长	–	–
FlipRand	pos:起始位置	–	–	–
Arithmetic	pos:起始位置	big_endian:是否为大端模式	value:加减数值	step:步长
ArithmeticDigit	pos:起始位置	big_endian:是否为大端模式	value:加减数值	step:步长
ReplaceRand	pos:起始位置	–	–	–
ReplaceSpec	pos:起始位置	step:步长	–	–
InsertRand	pos:起始位置	–	–	–
InsertSpec	pos:起始位置	–	–	–
DeleteRand	pos:起始位置	step:步长	–	–
DeleteDeter	pos:起始位置	–	–	–
Shuffle	pos:起始位置	step:步长	–	–
Swap	pos1:起始位置 1	step1:步长 1	pos2:起始位置 2	step2:步长 2
ChangeSize	value:改变量	mode:模式(add/subtract)	–	–
ChangeLine	pos:起始位置	mode:模式(repeat/remove/swap)	–	–
Repeat	pos:起始位置	step:步长	times:重复次数	–
Splicing	–	–	–	–

综上所述,变异部分的框架设计如图 8 所示.

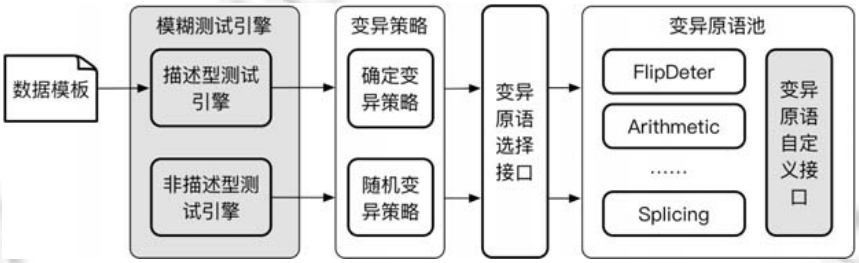

Fig.8 Framework of mutation

图 8 变异部分框架

2.4.2 监控基础原语

监控目标程序的进程状态是最直接、准确的监控方式,除了与目标程序进程直接相关的监控方法以外,对于协议类、服务类、内核类的目标程序,还可以使用间接的方式进行监控,如,通过端口扫描对协议类目标程序进行监控;通过网络抓包、流量分析对服务类目标程序进行监控;通过系统日志对操作系统内核进行监控.

本文以进程监控为主,间接监控为辅来设计监控原语.监控原语与模糊测试器所能支持的操作系统息息相关,同时,监控原语提供的不同的监控粒度也影响着漏洞挖掘人员分析、确认软件缺陷的效率.本文将监控原语从 3 个角度进行分析.

- 一是可支持的操作系统;
- 二是监控粒度;
- 三是是否支持远程监控以及其他间接监控方式.

本文将能够获取目标程序异常时堆栈、寄存器、指令、内存等信息的监控原语称为细粒度监控方式,反之称为粗粒度监控方式.

本文同时兼顾这 3 个角度,分析了测试目标为文件类、协议类、浏览器类以及 API 类的 38 款模糊测试器,总结出 10 种监控基本操作,监控基本操作在各模糊测试器中的选用比例,如图 9 所示.其中,58%的模糊测试器选

用了监控进程返回值(process exit code)的方法,以该方式进行监控无法获取栈、寄存器等信息,属于粗粒度监控.16%的模糊测试器使用了 Sanitizer(如 ASAN[22,23]等)这样的编译器提供的监控缺陷的方式,但这种方式只适用于可以修改编译参数的开源目标程序.是否能够进行细粒度监控,决定了是否可以获取更多的程序缺陷信息,同时也影响了是否能够对监控到的缺陷进行有效去重.这些因素将直接影响漏洞挖掘人员分析软件缺陷的效率.除此之外,在38个模糊测试器中,有32%的模糊测试器可以进行远程监控,支持远程监控的模糊测试器几乎都使用于协议类目标程序的测试.是否能够进行远程监控,对于协议、服务类型的软件有重大影响.

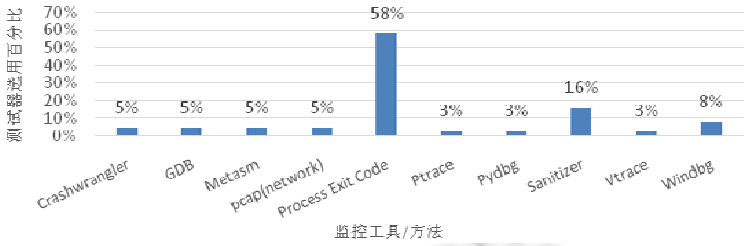

Fig.9 Choice percentage of monitoring basic operations

图 9 监控基本操作选用百分比

根据上述分析,本文将监控原语归纳为如表 6 所示的 6 种原语:Windows 本地监控(WinLocal)、Windows 远程监控(WinRemote)、Windows 网络监控(WinNet)、Linux 本地监控(LinLocal)、Linux 远程监控(LinRemote)、Linux 网络监控(LinNet).

Table 6 Monitoring primitives

表 6 监控原语

pName	pArgs 及释义					
	Arg1	Arg2	Arg3	Arg4	Arg5	Arg5
WinLocal	process_name: 进程名称	target_program: 目标程序 启动命令	timeout: 监控超时	–	–	–
WinRemote	process_name: 进程名称	procmon_options: 目标程序 控制命令	target_host: 目标程序 主机地址	target_port: 目标程序 端口号	procmon_host: 进程监控 主机地址	procmon_port: 进程监控 端口号
WinNet	process_name: 进程名称	netmon_options: 目标程序 控制命令	target_host: 目标程序 主机地址	target_port: 目标程序 端口号	newmon_host: 网络监控 主机地址	netmon_port: 网络监控 端口号
LinLocal	process_name: 进程名称	target_program: 目标程序 启动命令	timeout: 监控超时	–	–	–
LinRemote	process_name: 进程名称	procmon_options: 目标程序 控制命令	target_host: 目标程序 主机地址	target_port: 目标程序 端口号	procmon_host: 进程监控 主机地址	procmon_port: 进程监控 端口号
LinNet	process_name: 进程名称	netmon_options: 目标程序 控制命令	target_host: 目标程序 主机地址	target_port: 目标程序 端口号	newmon_host: 网络监控 主机地址	netmon_port: 网络监控 端口号

以上监控原语将实现对 Windows,Linux 操作系统、本地、远程进程的粗细粒度监控以及网络监控的支持,依托于原语参数传递,实现对监控过程的控制.监控部分的框架设计如图 10 所示.

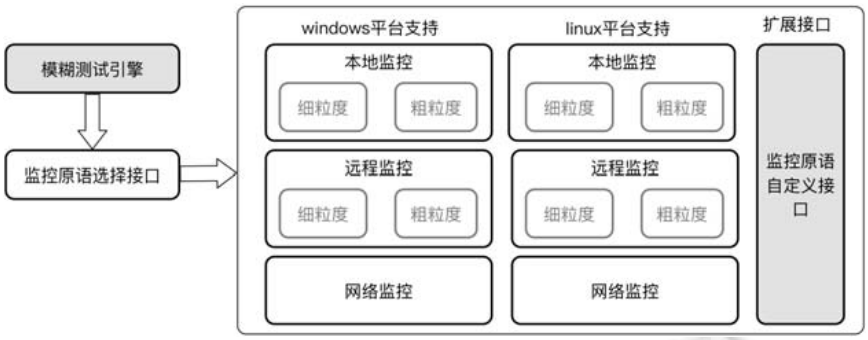

Fig.10 Framework of monitor

图 10 监控部分框架

2.4.3 反馈基础原语

反馈模块在模糊测试中主要用于对生成模糊数据进行指导,使生成的模糊数据更趋向于触发目标程序缺陷.反馈模块为变异选择优势样本,变异部分在优势样本上进行进一步变异,使变异生成的模糊数据触发更深层次的目标程序缺陷.当下流行的反馈方法为通过插桩的方式计算代码覆盖率,代码覆盖率能够直观地体现出模糊测试对目标程序测试的全面程度.针对上文所述的 38 款模糊测试器,本文分析归纳出 11 种反馈基本操作.

目标程序是否开源、运行平台都影响了插桩方式的选择,对于非开源目标程序的插桩方式多选用运行时插桩的方法(如 Pin,dynamoRio 等),对于开源目标程序的插桩方式多选用编译时插桩的方法(如 SANCOV[24],afl-gcc 等).同时,插桩粒度的不同也影响了覆盖率的准确度以及目标程序运行速度,常见的插桩粒度有 basic block(bb),edge,function.本文将 11 种反馈基本操作总结为表 7 所示的 4 个反馈原语:Windows 平台运行时插桩(WinRun)、Windows 平台编译时插桩(WinComp)、Linux 平台运行时插桩(LinRun)、Linux 平台编译时插桩(LinComp).

Table 7 Guider primitives

表 7 反馈原语

pName	pArgs 及释义	
	Arg1	Arg2
WinRun	guider_options:反馈相关的命令及路径参数	mode:插桩粒度(bb/edge/function)
WinComp	mode:插桩粒度(bb/edge/function)	–
LinRun	guider_options:反馈相关的命令及路径参数	mode:插桩粒度(bb/edge/function)
LinComp	mode:插桩粒度(bb/edge/function)	–

所以,本文在反馈基础原语的设计上兼顾了开源、闭源目标程序、目标程序运行平台以及插桩粒度,将反馈部分进行了如图 11 所示的设计.

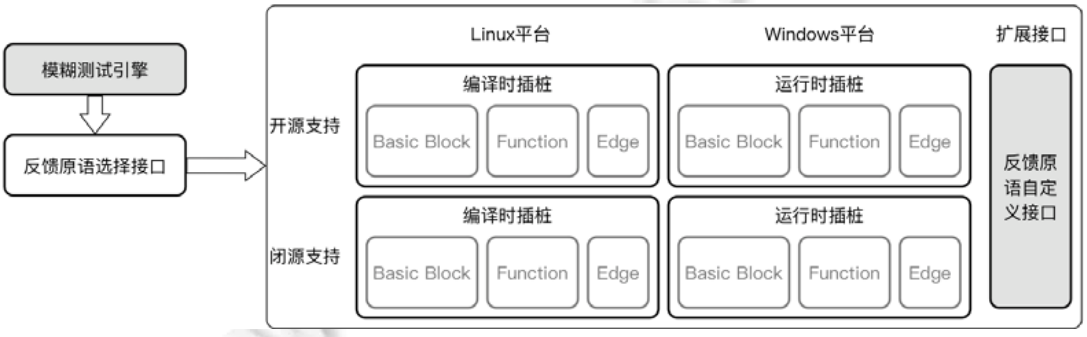

Fig.11 Framework of guider

图 11 反馈部分框架

2.5 小 结

在可编程模糊测试技术中,

- 原语的抽象以及 FDS 规范的提出,让漏洞挖掘人员可以使用更高层次的代码来编写模糊测试器,极大地减少了编写模糊测试器需要使用的代码量;
- 原语的封装及基础原语的制定,提升了模糊测试技术方法的可集成以及可扩展的能力,实现了测试需求覆盖的最大化;
- 原语的方法参数的设置,使得漏洞挖掘人员可以依据测试需求的不同对模糊测试中选用的不同原语方法进行灵活的控制;
- 模糊测试引擎的设计,解决了现有模糊测试器架构单一的问题,尽可能多地支持不同类型目标程序的测试.

3 原型框架的实现

基于可编程模糊测试技术,本文使用 python 语言实现了原型框架工具 Puzzer.总代码行数为 39 968 行(包含空行),其中,58.8%的代码用于实现模糊测试原语库,41.2%的代码用于实现预处理器以及模糊测试引擎.漏洞挖掘人员利用模糊测试原语编写符合 FDS 规范的制导程序,通过 Puzzer 的解析生成基于 python 代码的定制化模糊测试器,然后启动模糊测试器对目标程序进行测试.

Puzzer 实现了如图 12 所示的 FDS 解析器:(1) 解析制导程序生成抽象模糊测试语法树;(2) 遍历语法树、调用模糊测试原语库中定义的方法;(3) 对调用的原语方法进行模块化整合,生成基于 python 语言的定制模糊测试器.

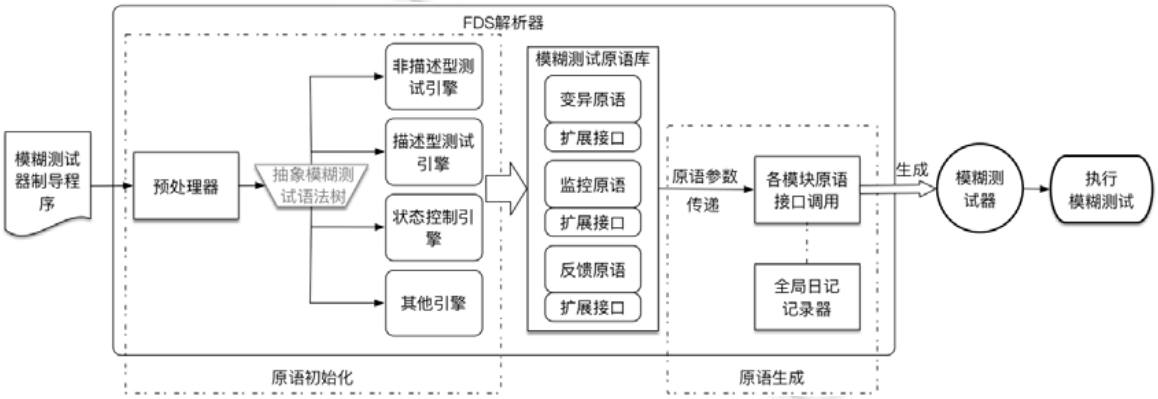

Fig.12 Implementation of Puzzer

图 12 Puzzer 的实现

在 Puzzer 中涉及到的模糊测试引擎包括:描述型测试引擎、非描述型测试引擎、状态控制引擎.模糊测试引擎首先遍历模糊测试抽象语法树对原语进行初始化,然后通过对原语实例接口的参数传递以及调用生成模糊测试的各模块的原语,最终构建出定制的模糊测试器.其中,

- 非描述型测试引擎主要用于测试没有给出数据模板的目标程序;
- 描述型测试引擎主要用于测试给出数据模板的目标程序;
- 状态控制引擎主要用于测试类似网络协议有明确的状态转换的目标程序.

对于描述型模糊测试引擎,Puzzer 中定义了多种数据类型,如 String,Number, Block 等,描述型测试引擎在解析数据模板时将不同的数据类型解析到与之对应的类中,根据类型以及数据模板中设置的类型属性调用不同类型的变异原语.

模糊测试引擎向各模块的原语生成接口中传递了相关控制参数的同时,还传递了日志记录器的实例,实现在测试过程中详细记录所需的相关信息以及模糊测试的执行状态,为模糊测试器的异常处理以及目标程序的状态分析提供支持.

模糊测试原语库中实现了文中归纳的 16 种变异原语、6 种监控原语以及 4 种反馈原语.

- 监控原语基于 Ptrace,Pydbg,ASAN,pcap 等工具实现了对 Windows 平台和 Linux 平台的粗粒度或细粒度的远程、本地、网络监控.
- 反馈原语基于 Pin,SANCOV 等工具实现了对 Windows 平台和 Linux 平台的函数、基本块、控制流边等粒度的执行信息记录.

4 实验验证

本文从 Github 上好评度排前 5 名的模糊测试中依据应用平台以及适用的目标程序种类不同选择了 Honggfuzz,WinAFL,Sulley 这 3 款模糊测试器,同时还选择了未在 Github 上面公开的主流模糊测试器 AFL 以及 Peach(2.3).所选对比模糊测试器支持的目标程序覆盖了 Windows 平台和 Linux 平台上开源及闭源程序,包含协议服务类以及文件类目标程序.

基于上述 5 款模糊测试器,本文从原语覆盖完整度、模糊测试器构建复杂度、模糊测试器扩展能力、漏洞发现能力这 4 个角度评价可编程模糊测试技术.

Windows 平台以及 Linux 平台的实验环境分别为:

(1) Windows 7 x86,内存 4G,Intel Xeon E312xx(Sandy Bridge) 2.40GHz;

(2) Linux Ubuntu x86_64,内核 4.4.0,内存 8G,Intel Xeon CPU E5-2620 v3 2.40GHz.

4.1 原语方法覆盖完整度

基于上文中归纳的 16 种变异方法,将 Puzzer 中实现的变异方法与 AFL、Honggfuzz、Peach(2.3)、Sulley、Winafl 进行比较,见表 8.

Table 8 Integrity of mutation methods

表 8 变异方法完整度

变异原语名称	AFL	Peach(2.3)	Honggfuzz	Sulley	Winafl	Puzzer
FlipDeter	✔	–	✔	–	✔	✔
FlipRand	–	✔	–	–	–	✔
Arithmetic	✔	–	✔	–	✔	✔
ArithmeticDigit	–	✔	–	–	–	✔
ReplaceRand	–	✔	✔	✔	–	✔
ReplaceSpec	✔	✔	✔	✔	✔	✔
InsertRand	✔	✔	–	✔	✔	✔
InsertSpec	–	✔	–	–	–	✔
DeleteRand	–	–	–	–	–	✔
DeleteDeter	✔	–	–	–	✔	✔
Shuffle	–	–	–	–	–	✔
Swap	–	–	–	–	–	✔
ChangeSize	–	✔	✔	–	–	✔
ChangeLine	–	–	–	–	–	✔
Repeat	✔	✔	–	–	✔	✔
Splicing	–	–	–	–	–	✔
16 种	6/16=37.5%	9/16=56%	5/16=31.25%	3/16=18.75%	6/16=37.5%	16/16=100%

对于监控方法的完整度,本文将从平台支持、粗细粒度监控、本地远程或网络监控这个几个方面来进行评估,见表 9.

对于反馈方法的完整度,主要从平台支持、运行时插桩、编译时插桩以及插桩粒度等角度进行评价,见表 10.

Table 9 Integrity of monitoring methods

表 9 监控方法完整度

监控原语名称(监控粒度)		AFL	Peach(2.3)	Honggfuzz	Sulley	Winafl	Puzzer
WinLocal	细粒度	–	✔	✔	–	–	✔
	粗粒度	–	✔	✔	✔	✔	✔
WinRemote	细粒度	–	✔	–	–	–	✔
	粗粒度	–	✔	–	✔	–	✔
WinNet		–	✔	–	✔	–	✔
LinLocal	细粒度	–	✔	✔	–	–	✔
	粗粒度	✔	✔	✔	✔	–	✔
LinRemote	细粒度	–	✔	–	✔	–	✔
	粗粒度	–	✔	–	–	–	✔
LinNet		–	✔	–	✔	–	✔

Table 10 Integrity of feedback methods

表 10 反馈方法完整度

反馈原语名称(插桩粒度)		AFL	Peach(2.3)	Honggfuzz	Sulley	Winafl	Puzzer
WinRun	edge	–	–	–	–	✔	✔
	bb	–	–	–	–	✔	✔
	function	–	–	–	–	✔	✔
WinComp	edge	–	–	✔	–	–	✔
	bb	–	–	✔	–	–	✔
	function	–	–	✔	–	–	✔
LinRun	edge	✔	–	✔	–	–	✔
	bb	–	–	✔	–	–	✔
	function	–	–	✔	–	–	✔
LinComp	edge	✔	–	✔	–	–	✔
	bb	–	–	✔	–	–	✔
	function	–	–	✔	–	–	✔

4.2 模糊测试器构建开销

使用 Puzzer 框架编写平均 54 行代码,实现了 AFL,Honggfuzz,Peach(2.3),Sulley,Winafl 原有基本操作的百分比见表 11.上述 5 款模糊测试器的代码量分别为 13 695 行、12 210 行、54 480 行、10 644 行、9 553 行.可编程模糊测试技术可以通过平均 54 行代码实现 5 款主流模糊测试器的平均 87.8%的基本操作.

Table 11 Costs of constructing fuzzers

表 11 模糊测试器构建开销

	AFL	Honggfuzz	Peach(2.3)	Sulley	Winafl
Puzzer 编写行数	51	46	60	53	62
基本操作覆盖百分比(%)	100	71.4	80	87.5	100

4.3 模糊测试器扩展能力

本实验将以两个实例阐述基于 Puzzer 所构建的模糊测试器的可扩展能力.

- 实例 1:若模糊测试原语库中已经含有需要扩展方法的原语实例,漏洞挖掘人员仅需通过在制导程序中添加需要扩展的原语.如图 13 所示,若需要为一个模糊测试器添加 Arithmetic 变异方法,仅需在 Prog1 中添加一行代码得到 Prog2,Puzzer 通过对 Prog2 的解析即可生成扩展后的模糊测试器.
- 实例 2:若模糊测试原语库中未含有需要扩展方法的原语实例,漏洞挖掘人员首先需要按照原语自定义接口的要求自定义新的原语实例,然后在制导程序中添加相应的扩展代码.如图 14 所示,漏洞挖掘人员首先使用 python 语言编写 NewMutator 原语实例,然后在原有制导程序 Prog1 中添加对该原语的调用代码形成 Prog2,Puzzer 通过解析 Prog2 实现原有模糊测试器的扩展.

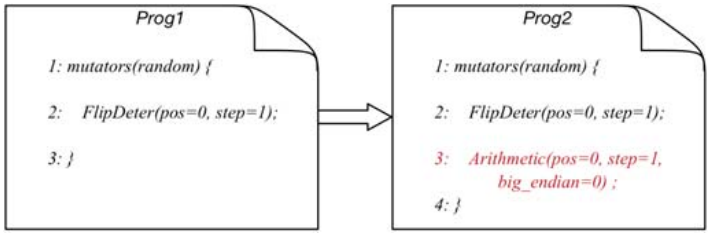

Fig.13 Case 1

图 13 实例 1

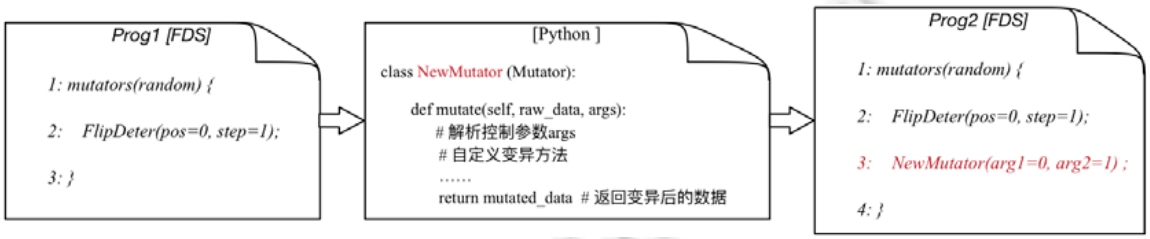

Fig.14 Case 2

图 14 实例 2

4.4 漏洞发现能力

基于 FDS 编程实现具备 AFL 全部功能的模糊测试器,其漏洞发现能力与 AFL 的对比见表 12.本实验中选取了 8 个目标程序,使用相同种子集合,在相同的系统环境下运行 24 小时,实验结果表明,使用可编程模糊测试技术生成的模糊测试器与原始模糊测试器测试效果相当.由于数据变异的随机性以及测试时间的限制,crash 数目与 AFL 比较稍有差异,差异数均在 1 以内.此外,本实验中发现的新的软件缺陷均已向 CVE(CVE-2017-9084, CVE-2017-9335)和软件作者报告.

Table 12 Comparison of vulnerability discovery capabilities

表 12 漏洞发现能力的比较

异常数量	bsdtar	fax2ps	jhead	tidy	tiff2ps	tiffset	pdftohtml	pdfunite
Puzzer	0	1	7	0	1	2	0	1
AFL	0	1	8	0	1	1	0	1

4.5 小 结

综上所示,Puzzer 框架基础原语中包含的变异、监控、反馈原语覆盖了现下开源模糊测试器中绝大多数的基本操作,支持 Windows 平台和 Linux 平台的开源或者闭源的多种类型的目标程序.

- Puzzer 能够通过百行之内的制导程序,快速构建不同功能的模糊测试器.
- Puzzer 能够支持漏洞挖掘人员依据具体测试需求的不同自定义新的原语方法,实现模糊测试器的便捷扩展.
- 使用 Puzzer 框架,基于 AFL 构建生成的模糊测试器的测试效果与 AFL 相当.

可编程模糊测技术降低了开发模糊测试器的开销及门槛,具备良好的扩展能力,能够根据不同测试需求快速构建契合目标程序的模糊测试器,保证了漏洞挖掘的时效性.

5 总 结

为了减少“构建模糊测试器”在整个模糊测试流程中的时间开销、降低模糊测试器开发门槛、增强模糊测试器的扩展性,本文对模糊测试流程进行分解,对 38 款开源模糊测试进行分析,归纳出 53 种基本操作,提出了一

种可编程模糊测试技术.本文抽象出 26 个模糊测试原语,构建了一套用于编写模糊测试器制导程序的语法规范 FDS,并设计了 FDS 解析器以及解析器中的预处理器、模糊测试器引擎和模糊测试原语库,以此支持漏洞挖掘人员通过编写制导程序实现通用型黑/灰盒模糊测试器的全流程、快速、定制化构建,同时实现模糊测试全流程的调试.基于可编程模糊测试原型框架 Puzzer 的实验结果表明:可编程模糊测试技术将模糊测试器的开发成本降至原有的 1%,降低了模糊测试器的开发门槛、增强了模糊测试器的可扩展性,实现了模糊测试器的快速定制化构建.

References:

[1] Sutton M, Greene A, Amini P, Wrote; Huang L, Yu LL, Li H, Trans. Fuzzing: Brute Force Vulnerability Discovery. Beijing: China Machine Press, 2009 (in Chinese).

[2] American fuzzy lop (AFL). 2017. http://lcamtuf.coredump.cx

[3] OSS-Fuzz. 2017. https://github.com/google/oss-fuzz

[4] Pham VT, B¨ohme M, Roychoudhury A. Model-Based whitebox fuzzing for program binaries. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). 2016. 552−562. [doi: 10.1145/2970276.2970316]

[5] Rawat S, Jainz V, Kumarz A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-Aware evolutionary fuzzing. In: Proc. of the NDSS 2017. 2017. 1−16. [doi: 10.14722/ndss.2017.23404]

[6] Stephens N, Grosen J, Salls C, Dutcher A, Wang RY, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the NDSS 2016. 2016. 1−16. [doi: 10.14722/ndss.2016.23368]

[7] Wang MT, Wei T, Gu G, Zou W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (IEEE S&P 2010). 2010. 497−512. [doi: 10.1109/SP.2010.37]

[8] Gascon H, Wressnegger C, Yamaguchi F, Arp D, Rieck K. PULSAR: Stateful black-box fuzzing of proprietary network protocols. In: Proc. of the SecureComm. 2015. 330−347. [doi: 10.1007/978-3-319-28865-9_18]

[9] Tsankov P, Dashti MT, Basin D. SECFUZZ: Fuzz-Testing security protocols. In: Proc. of the Automation of Software Test (AST). 2012. 1−7. [doi: 10.1109/IWAST.2012.6228985]

[10] Woo M, Cha SK, Gottlieb S, Brumley D. Scheduling black-box mutational fuzzing. In: Proc. of the 20th ACM Conf. on Computer and Communications Security (CCS 2013). 2013. 511−522. [doi: 10.1145/2508859.2516736]

[11] Huang Y, Zeng FP, Cao Q. Fuzzing test approach based on dynamic tracking of library functions. Computer Engineering, 2010, 36(16):39−41 (in Chinese with English abstract).

[12] Rebert A, Cha SK, Avgerinos T, Foote J, Warren D, Grieco G, Brumley D. Optimizing seed selection for fuzzing. In: Proc. of the 23rd USENIX Security Symp. (USENIX Security 2014). 2014. 861−875.

[13] Böhme M, Pham VT, Roychoudhury A. Coverage-Based greybox fuzzing as Markov chain. In: Proc. of the 23rd ACM Conf. on Computer and Communications Security (CCS 2016). 2016. 1−12. [doi: 10.1145/2976749.2978428]

[14] Zhao YH, Kan JJ. Research and design of symbol execution-based test data generation method. Computer Applications and Software, 2014,31(2):303−306 (in Chinese with English abstract).

[15] Ma JX, Zhang T, Li ZJ, Zhang JX. Improved fuzzy analysis methods. Journal of Tsinghua University, 2016,56(5):478−483 (in Chinese with English abstract).

[16] Wu ZY, Xia JJ, Sun LC, Zhang M. Survey of multi-dimensional fuzzing technology. Application Research of Computers, 2010, 27(8):2810−2813 (in Chinese with English abstract).

[17] Wang ZQ, Zhang YQ, Liu QX, Huang TP. Algorithm for discovering SNMP protocol vulnerability. Journal of Xidian University, 2015,42(4):20−26 (in Chinese with English abstract).

[18] Peach fuzzer platform. 2017. http://www.peachfuzzer.com/products/peach-platform/

[19] Honggfuzz. 2017. https://github.com/google/honggfuzz

[20] Choronzon. 2017. https://github.com/CENSUS/choronzon

[21] Sulley fuzzer. 2017. https://github.com/OpenRCE/sulley

[22] ASAN. 2017. https://github.com/google/sanitizers/wiki/AddressSanitizer

[23] Serebryany K. Libfuzzer: A library for coverage-guided fuzz testing (within llvm). 2017. http://llvm.org/docs/LibFuzzer.html

[24] SANCOV. 2017. http://clang.llvm.org/docs/SanitizerCoverage.html

附中文参考文献:

[1] Sutton M, Greene A, Amini P,著;黄陇,于莉莉,李虎,译.模糊测试:强制性安全漏洞发掘.北京:机械工业出版社,2009.

[11] 黄奕,曾凡平,曹青.基于库函数动态跟踪的 Fuzzing 测试方法.计算机工程,2010,36(16):39−41.

[14] 赵跃华,阚俊杰.基于符号执行的测试数据生成方法的研究与设计.计算机应用与软件,2014,31(2):303−306.

[15] 马金鑫,张涛,李舟军,张江霄.Fuzzing 过程中的若干优化方法.清华大学学报,2016,56(5):478−483.

[16] 吴志勇,夏建军,孙乐昌,张旻.多维 Fuzzing 技术综述.计算机应用研究,2010,27(8):2810−2813.

[17] 王志强,张玉清,刘奇旭,黄庭培.一种简单网络管理协议漏洞挖掘算法.西安电子科技大学学报,2015,42(4):20−26.

杨梅芳(1993－),女,内蒙古通辽人,硕士生,主要研究领域为软件安全,程序分析.

霍玮(1982－),男,博士,副研究员,博士生导师,CCF 专业会员,主要研究领域为软件漏洞挖掘和安全评测,基于大数据的软件安全分析,智能终端系统及应用安全分析.

邹燕燕(1989－),女,助理研究员,CCF 专业会员,主要研究领域为软件安全,程序分析.

尹嘉伟(1994－),男,硕士生,CCF 学生会员,主要研究领域为软件安全,程序分析.

刘宝旭(1972－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为网络攻防技术,安全态势感知技术.

龚晓锐(1973－),男,高级工程师,主要研究领域为网络攻防,软件逆向分析,Web 安全,移动互联网安全.

贾晓启(1982－),男,博士,研究员,博士生导师,主要研究领域为网络攻防技术,操作系统安全,云计算安全.

邹维(1964－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件安全分析理论与技术,网络安全评测.