doi:10.11887/j.cn.202302005 http://journal.nudt.edu.cn

混合模糊测试中混合符号执行优化策略评估与分析*

陶 静[1],糜娴雅[2],王宝生[1],王鹏飞[1]
(1. 国防科技大学 计算机学院,湖南 长沙 410073;2. 军事科学院 智能博弈与决策实验室,北京 100071)

摘 要:针对传统混合模糊测试提升技术多聚焦于利用多种动静态分析手段辅助而忽略了混合符号执行自身性能的问题,提出一种混合模糊测试平衡点模型,并基于该模型对主流混合符号执行方案进行剖析,包括污点分析辅助模糊测试、混合模糊测试以及混合符号执行,归纳了6种符号执行方案,基于混合符号执行引擎Triton复现了6种符号执行方案,并通过10个典型真实程序进行了测试评估。从效率、内存、覆盖率三个维度对各个方案进行性能对比与影响因素分析。实验证明,优化方案都可以消除不必要的约束并减少时间和空间开销,但约束缩减会造成信息丢失,造成覆盖率降低。基于实验数据分析,提出了一个优化方案的性能序列,并提出三种针对不同测试需求的优化方案。

关键词:软件安全;软件漏洞挖掘;混合符号执行;混合模糊测试
中图分类号:TP311 **文献标志码**:A **文章编号**:1001-2486(2023)02-045-10

Evaluation and analysis of concolic execution optimizations in hybrid fuzzing

TAO Jing[1], *MI Xianya*[2], *WANG Baosheng*[1], *WANG Pengfei*[1]
(1. College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China;
2. Intelligent Game and Decision Lab, Academy of Military Sciences, Beijing 100071, China)

Abstract: Aiming at the problem that the traditional hybrid fuzzy test promotion technology focuses more on the use of multiple dynamic and static analysis methods to assist and ignores the performance of concolic execution, a hybrid fuzzing balance-point model was proposed. Based on the model, the popular concolic execution tools were analyzed, including taint-assist fuzzing, hybrid fuzzing and concolic execution, and 6 symbol execution schemes were summarized. Based on the hybrid symbol execution engine Triton, 6 symbol execution schemes were reproduced, and tested and evaluated through 10 typical real programs. Performance comparison and impact factor analysis of each scheme were conducted from three dimensions of efficiency, memory consumption and coverage. Experiment results show that all of the optimization patterns can basically reduce the unnecessary constraints and thus reduce time and space consumption. However, the reduction of constraints can cause information loss and lead to coverage decrease. Based on the analysis of experimental data, the performance sequence of an optimization scheme was proposed, and three optimization schemes for different test requirements were proposed.

Keywords: software security; software vulnerability discovery; concolic execution; hybrid fuzzing

随着软件系统的规模和复杂性不断增加,软件漏洞问题层出不穷。根据Skybox Security最新发布的2020年漏洞和威胁趋势报告[1],2020年已发现的漏洞数量有可能突破新的纪录——超过20 000个。因此自动化软件漏洞挖掘近年来成为软件安全研究的热点。从是否运行实际程序角度,目前自动化软件漏洞挖掘方法研究主要有三种模式:静态分析、动态分析和混合分析。

早期的静态分析方法将程序代码模型作为分析对象,然后建立一系列基于源代码或者二进制反汇编代码的漏洞分析规则作为检测程序漏洞的依据。这种方法将漏洞挖掘看作一个遍历程序代码并进行规则匹配的过程,并不需要对程序进行实际执行。后来出现的静态符号执行[2]将程序的输入符号化,程序中变量的值表示为符号值和常量组成的计算表达式,随着程序控制流过程,程序的输入被逐步表示为符号值的表达式。对于每一条路径,都会生成一个包含符号值的符号路径约束,判断路径条件的可满足性可转化为判断对符号取值的约束是否可满足的问题。由于软件系

* **收稿日期**:2021-04-09
基金项目:国防科技大学校科研计划基金资助项目(ZK20-17)
作者简介:陶静(1971—),女,山东昌邑人,副研究员,硕士,E-mail:ellen5702@aliyun.com;
糜娴雅(通信作者),女,江苏镇江人,助理研究员,博士,E-mail:mixianya09@nudt.edu.cn

统的规模性和复杂性,随着执行路径的深入,静态符号执行将面对两个问题:约束过大难以求解和路径爆炸。

不同于静态分析,动态分析对程序进行实际执行,因此需要提供一些测试用例,也称为初始种子。基于动态分析的漏洞挖掘方法又称为模糊测试[3],通过对输入字节进行随机变异,生成更多测试用例,以期待执行更多的程序路径。例如,当对程序提供大小为10个字节的输入,其变异的输入空间就是2^{80},所以完全随机变异的模糊测试输入空间会呈指数级增加。因此,高效且有目的性的变异策略成为模糊测试的研究重点。污点分析[4]是另一种典型的动态分析技术,旨在发现程序输入与程序特定部分之间的联系。在程序运行开始时,程序的输入被设定为污点源,在程序执行过程中,如果某个值的计算依赖于被设定为污点源的输入字节,那么这个值就会被看作是受污染的,而其他不依赖于污染源的值则被看作是未受污染的。污点传播策略定义了污点数据如何在程序执行的过程中传播,比如哪些操作引入了新的污点以及被污染的值上应该施加何种检查等。

混合分析将静态分析、动态分析二者各自的优势结合,弥补劣势,有助于漏洞挖掘效率和准确性的提升。一方面,动态分析具有执行快、效率高、开销小的优势,可以在很短的时间内执行大量的测试用例并计算覆盖率,生成反馈;另一方面,静态分析可以提供大量有关程序自身的信息,以此指导动态分析的反馈机制和变异策略。DART[5]是第一个尝试结合实际执行与符号执行的工作,从此有了混合符号执行(concolic execution,即concrete execution与symbolic execution的结合)的叫法。混合符号执行从一个特定输入出发分析程序执行的单条路径,同时进行实际执行和符号执行,从而在收集该条路径的符号化约束的同时,也维护实际值的状态。当执行触发一个条件分支,通过翻转此前的约束,执行路径也被翻转,从而执行新的路径。

近年来,研究人员将混合符号执行和模糊测试相结合,并辅以污点分析、静态分析等分析手段,称为混合模糊测试,成为混合分析方向的研究热点。混合模糊测试结合了模糊测试和混合符号执行各自的优点。通过模糊测试,程序的大部分路径可以在短时间内被迅速遍历;混合符号执行可以求解并通过复杂的条件检查,让程序执行走向更深的路径;而常规静态分析手段可以辅助提供程序信息,从而更有针对性、目的性地引导模糊测试和混合符号执行。

污点分析辅助的模糊测试和混合符号执行辅助的模糊测试是混合模糊测试的两个组合方向。前者基于“变异”,旨在通过回答“变异哪些字节”[6]和“变异成为什么”[7-10]的问题来缩小搜索空间;后者基于“求解”,旨在通过回答“什么时候用求解”和“求解指导的目标在哪里”的问题来更有效地提高覆盖率和引导模糊测试。基于变异的方案只需要合理的资源,但缺乏指向性;基于求解的方案更精确,但开销更大,存在可用性问题。两种方案各有优劣,于是很多研究者试图结合二者的优点,弥补缺点。前一种方案的典型实现是污点分析辅助的混合符号执行[11-13]。通过使用污点分析,可以识别程序或者输入的“重要的”(与目的相关)部分,然后混合符号执行可以只聚焦于这些重要的部分,从而减少开销。反过来,混合符号执行可以求解相关的约束并生成新的测试用例,提升了精确性。然而,由于污点分析自身的精确性问题,即存在污点传播的过度污染和污染不足的问题,会影响到分支约束的准确性。因此,对于大型真实程序,污点分析辅助的混合符号执行方案很难避免用精确性换取可用性的妥协。后一种方案的典型实现是符号执行辅助模糊测试的混合模糊测试[14-18]。对于基于覆盖率引导的模糊测试来说,混合模糊测试的目的在于尽可能多地覆盖执行路径。Driller[19]提出当模糊测试“卡”住,也就是一定时间段无法发现更多新路径时,触发混合符号执行,这样就可以求解那些用模糊测试很难通过的复杂分支检查。对于引导型的模糊测试来说,混合模糊测试的目标在于将测试导向更可能存在问题的代码。

本文首先提出混合模糊测试中的平衡点理论,来阐述混合符号执行与模糊测试在混合模糊测试中的作用。基于平衡点理论,本文对软件测试中的3种主流方法进行归纳和分析,对多个当前主流的混合符号执行辅助的混合模糊测试方案的优化策略模式进行了归类,总结出6种混合符号执行方案。然后基于Triton复现了6种方案,并选取10个典型的真实被测程序对6种方案进行了系统性测评。最后,通过测试数据对6种方案的性能影响进行了深入分析,提出了一个优化方案的性能序列,并提出3种针对不同测试需求的优化选择方案,指出了未来的优化方向。

1 混合符号执行优化策略分类

混合模糊测试基于混合符号执行与模糊测试

相结合而实现。为更清楚地阐述混合符号执行与模糊测试在混合模糊测试中的作用,本文提出一种混合模糊测试的平衡点模型,如图 1 所示,混合符号执行与模糊测试分别为模型在不同方向的两端。如果其中混合符号执行按照一定顺序探索完毕程序的所有路径,则可以将测试过程看作白盒测试;同理,如果采用基于变异而没有任何关于程序的信息的模糊测试,则可以看作黑盒测试。从图 1 的左端到右端,即从混合符号执行端到模糊测试端,待探索的某条路径的搜索空间会逐渐增大,需处理的约束复杂性也逐渐增大,而测试需要的资源开销逐渐变小,求解一条分支的准确性也逐渐降低。从右往左则相反。

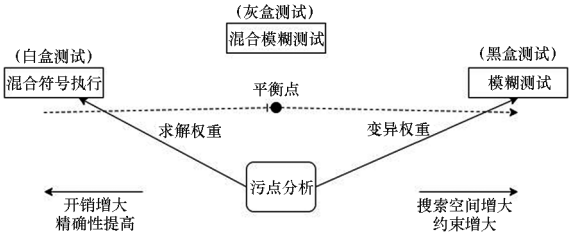

图 1　混合模糊测试的平衡点模型

Fig. 1　Balance point model of hybrid fuzzing

对于大多数现有的混合模糊测试工作来说,如何平衡白盒测试和黑盒测试的比例成为关键点,即程序的哪些部分应该由混合符号执行探索,哪些部分应该由模糊测试探索。通常来说,当分支的约束比较小的时候,能够被轻易求解而不需要很大的开销,则使用混合符号执行;当分支的约束比较大的时候,特别是大到混合符号执行很难在有限的时间和资源内求解,这个时候就要依靠基于变异的模糊测试来处理。在混合符号执行和模糊测试之间,有一个平衡点,决定了混合符号执行和模糊测试各自的工作量。混合模糊测试可以视为一个平衡点移动的问题:寻找平衡点的最佳位置,让混合符号执行和模糊测试的工作量得到最佳配比,则在有限的时间和资源预算下,混合模糊测试的性能能够达到最佳。

基于平衡点模型,本文对软件测试中的 3 种主流方法进行归纳和分析,包括污点分析辅助的模糊测试、混合模糊测试、混合符号执行。如图 2 所示,系统性归纳涉及工作超过 50 项,从 20 世纪 70 年代的早期工作,一直到当前的 state-of-the-art(由于文章篇幅限制,本文不对每项工作进行详细分析)。基于这些工作的传承关系、发展趋势及技术特点,例如是否涉及约束求解、符号化表示的范围、收集约束的方式、是否借助污点分析等,本文总结出 6 种混合模糊测试中的混合符号执行方案,每种方案各自用一个单词来表示,并列出了各自的代表性工作,如表 1 所示。

1)original:代表混合符号执行的原始实现模型。基于混合符号执行根本原理的原始实现,没有增加任何优化方法,代表工作有开山之作 DART[5]、最常用的符号执行引擎 KLEE[20]。

2)taint:基于链式追踪的污点分析辅助。指令级别的污点追踪方法,只收集污染后的相关约束,可以减少因额外指令引起的多余约束,代表工作有 QSYM[11] 和 STINGER[21]。

3)symbolized:只收集符号化的字节相关约束。这种方法不收集符号化之外的约束,可以减少未符号化的部分引入约束,代表工作有 COLOSSUS[22]。

4)last:“乐观求解”,即只求解最后一条分支的约束。当约束过大的时候,由于在指定时间内无法求解,所以只求解最后一条与分支相关的约束,把前序的所有约束都丢弃,代表工作有 QSYM[11] 中的 optimistic solving 模式。

5)function:函数摘要。对函数端进行模拟,可以去掉因为函数与环境交互引入的约束,而改为实际化执行,代表工作有 S2E[23] 和 angr[24]。

6)part:只符号化程序中与目标有关的一部分变量,而不是全部,可以减少与目标无关的约束,代表工作 Dowser[13]。

2　混合符号执行测试方案设计

为了对 6 种混合符号执行方案进行系统性测评和分析,本文首先基于混合符号执行引擎 Triton[12] 复现了 6 种方案。使用统一的平台实现方案可以消除因为平台差异引入的其他误差,更准确高效地比较不同混合符号执行辅助的混合模糊测试方案之间的特点和优劣。

2.1　实验配置

Triton 是一个基于插桩的二进制动态分析工具,提供了丰富且方便使用的 Python API 接口,可以实现混合符号执行原始模型和其他优化方案。Triton 支持所有 Linux x86/64 系统的系统调用,所以环境相关的约束可以通过追踪系统调用来引入,只要符号化文件读入系统调用相关的内存地址即可,这与大部分混合符号执行引擎的原理是相同的。Triton 与 QSYM 非常相似(实际上它比 QSYM 提出的时间要早三年),提供了基于链式追踪的指令级别污点分析引擎,也提供了只分析最后一条指令约束的“乐观求解”功能。Triton 本身并不提供函数摘要功能,但利用其 API 可以实现

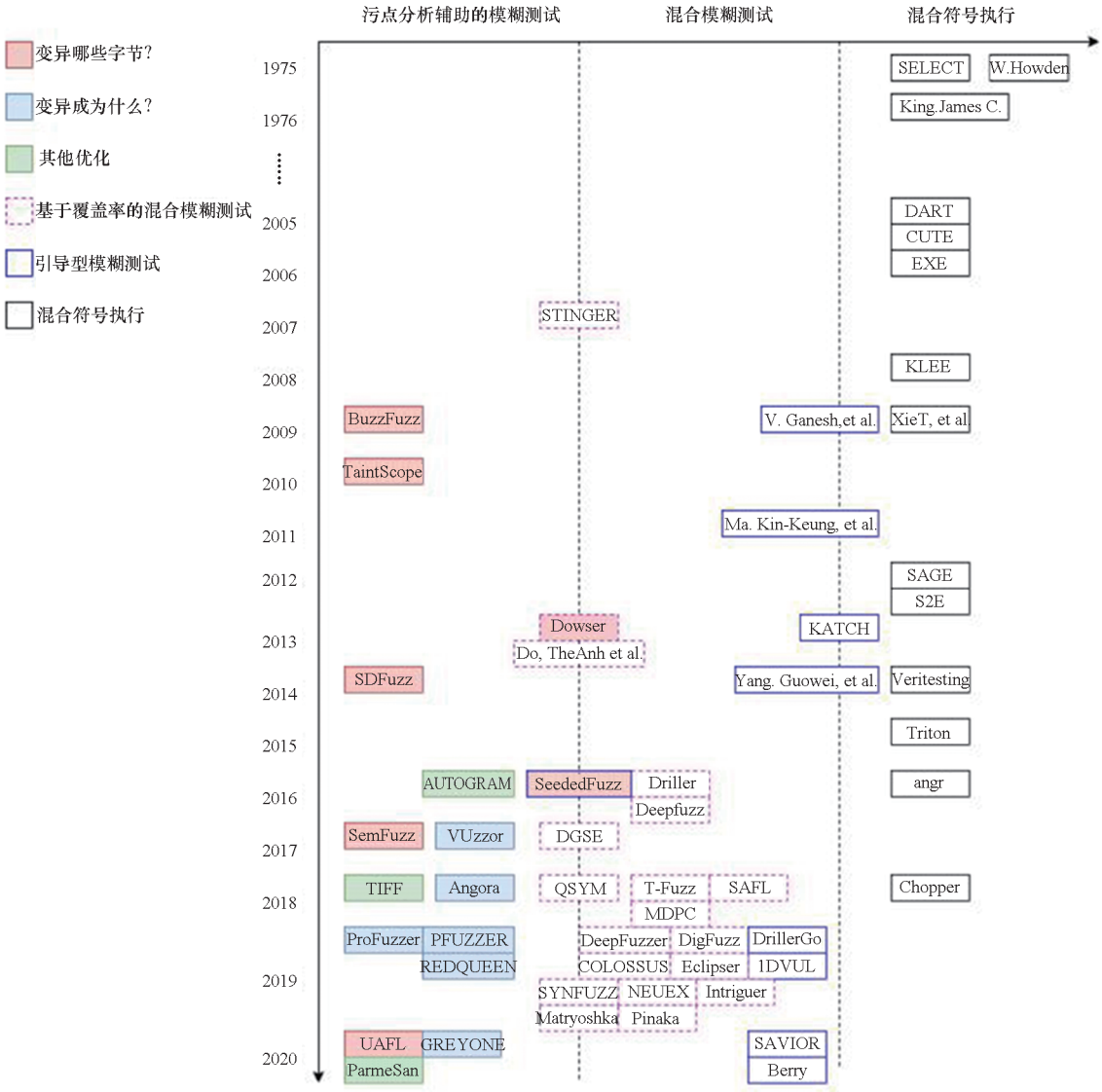

图 2 混合模糊测试发展历程

Fig. 2 Development course of hybrid fuzzing

表 1 本文归纳的 6 种混合符号执行方案

Tab. 1 Six patterns of different concolic execution solution

名称	优化技术描述	代表工作
original	混合符号执行的原始实现模型(无优化)	DART[5], KLEE[20]
taint	基于链式追踪的污点分析辅助	QSYM[11], STINGER[21]
symbolized	只收集符号化的字节相关约束	COLOSSUS[22]
last	“乐观求解”,即只求解最后一条分支的约束	QSYM[11]
function	函数摘要	S2E[23], angr[24]
part	只符号化程序的一部分	Dowser[13]

基于函数追踪插桩的函数摘要,只是不能自动化,需要每个函数都实现一次。其中用来分析分支的污点分析模块基于 libdft64[25] 实现,这是一个基于 Pin 插桩编写的二进制污点分析引擎,实现了 Triton 不具备的分支分析功能。

2.2 评价指标

实验选取 10 个常见的拥有不同功能的真实 Linux 二进制软件进行测试,程序名称、版本和具体功能如表 2 所示。这些程序的功能基本可以涵盖常用的程序类型,也可以保证约束的多样性,有助于测试结果更客观、正确地反映实际情况。在实验测试中主要记录了 4 种结果数据,用于形成评价指标:

1)运行时间:该方案在被测程序的一条执行路径上进行混合符号执行所用的时间,是开销的

表 2　10 个被测软件的相关信息

Tab. 2　Information of 10 tested software

程序名称	版本	大小/KB	功能	输入大小/B
boringssl	2016 - 01 - 02	2 244	服务器程序	132
guetzli	2017 - 3 - 30	3 505	图像处理	52
libpng	1.2.56	429	图像处理	218
openssl	1.0.2d	291	服务器程序	10
libxml2	v2.9.2	4 771	XML C 解析器	15
sqlite	2016 - 11 - 14	895	SQL 数据库引擎	50
woff2	2016 - 05 - 06	1 725	woff2 格式解析器	66
re2	2014 - 12 - 09	4 551	正则表达式解析引擎	12
pcre2	10.00	808	Perl 适用的正则表达式解析	116
proj4	2017 - 08 - 14	3 395	CRS 库	15

一个方面,称为时间开销。

2)内存消耗:该方案执行一次混合符号执行需要使用的内存大小,是开销的一个方面,称为空间(内存)开销。

3)失败分支:在规定的"超时"时间限制内无法完成求解的分支数量,在本文实验中,这个限制被设置为 10 s,即 10 s 内无法求解出的分支数量。分支越复杂时,失败分支的数量就会越多。

4)覆盖路径:由新产生的测试用例所引入的覆盖路径,本文评价主要依据的是代码(行)覆盖率。

本文使用 3 种评价指标来评价每种方案的正确性和效率:

1)时间效率:表示该方案在分析一个输入对应的路径上的时间开销相比于原始模型的提升。通常认为时间开销越小越好,这样在相同时间内可以分析更多的输入,因此时间开销减少的百分比越多则越好。

2)内存开销:表示该方案在分析一个输入对应的路径上的空间开销相比于原始模型的提升。通常认为空间开销越小越好,否则会影响到程序其他部分分析的效率,且受到硬件条件的制约,因此空间开销减少的百分比越多则越好。

3)覆盖率提升:表示该方案在分析一个输入对应路径结束后产生的新测试用例能够引入的代码覆盖率相比原输入代码覆盖率的提升。不同的优化方案由于牺牲了不同的约束,可能对代码覆盖率造成不同影响。通常认为覆盖率越高越好,因此代码覆盖率的提升百分比越多越好。

3　性能影响因素分析

使用设计的综合测试平台对选取的 10 个被测程序进行实验。如表 2 所示,实验所用测试程序来自领域内广泛使用的测试集 Google fuzzer-test-suite,类型涵盖服务器程序、图像程序、数据库引擎、格式解析器等,大小从 291 KB 到 4 771 KB,具有广泛代表性,能够综合反映各个方案的效果。

实验分别从运行时间、内存消耗、失败分支数以及代码路径覆盖四个方面对 6 个混合符号执行方案进行测评,每项实验重复 3 次取平均值,结果如图 3 ~ 6 所示,详细实验数据记录在表 3 ~ 6。

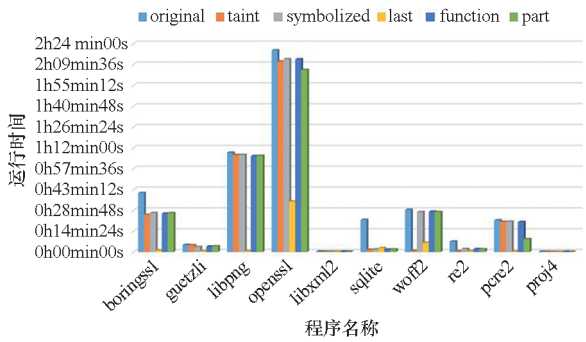

图 3　各方案的运行时间对比

Fig. 3　Comparison of runtime among different optimizations

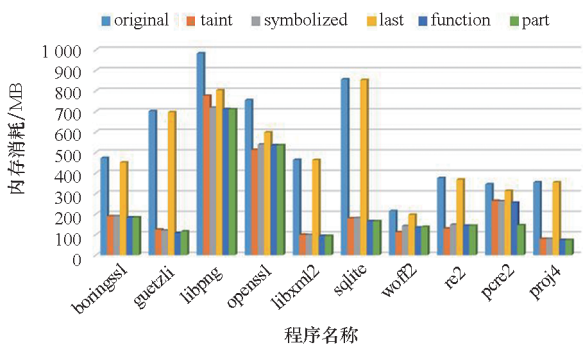

图 4　各方案的内存消耗对比

Fig. 4　Comparison of memory consumption among different optimizations

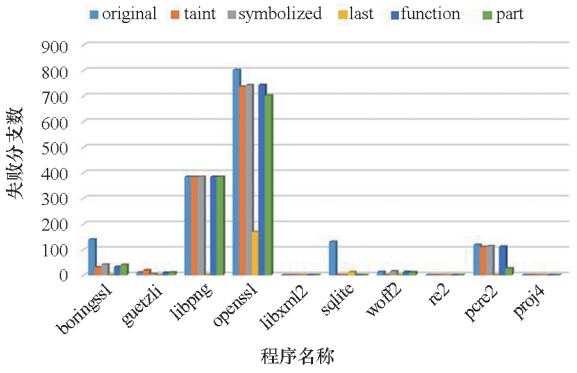

图5 各方案的失败分支数对比

Fig. 5 Comparison of failed branches among different optimizations

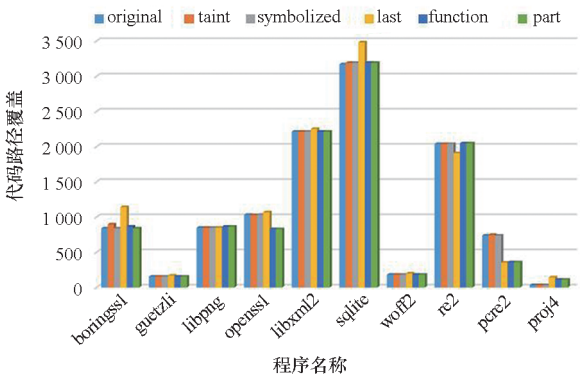

图6 各方案的代码路径覆盖对比

Fig. 6 Comparison of path coverage among different optimizations

表3 各优化方案的运行时间数据对比

Tab. 3 Comparison of runtime among different optimizations

程序名称	original	taint		symbolized		last	
	运行时间	运行时间	减少率/%	运行时间	减少率/%	运行时间	减少率/%
boringssl	0h40min45s	0h25min40s	37.01	0h27min02s	33.66	0h01min07s	97.26
guetzli	0h04min54s	0h04min41s	4.42	0h03min29s	28.91	0h00min36s	87.76
libpng	1h08min41s	1h07min06s	2.31	1h07min10s	2.21	0h00min37s	99.10
openssl	2h19min29s	2h11min43s	5.57	2h13min20s	4.41	0h35min00s	74.91
libxml2	0h00min21s	0h00min13s	38.10	0h00min14s	33.33	0h00min16s	23.81
sqlite	0h22min12s	0h01min35s	92.87	0h01min50s	91.74	0h02min47s	87.46
woff2	0h29min12s	0h00min47s	97.32	0h27min41s	5.19	0h06min22s	78.20
re2	0h07min14s	0h00min35s	91.94	0h02min06s	70.97	0h00min34s	92.17
pcre2	0h21min57s	0h20min52s	4.94	0h21min01s	4.25	0h00min25s	98.10
proj4	0h00min16s	0h00min11s	31.25	0h00min10s	37.50	0h00min09s	43.75

程序名称	original	function		part	
	运行时间	运行时间	减少率/%	运行时间	减少率/%
boringssl	0h40min45s	0h26min33s	34.85	0h26min51s	34.11
guetzli	0h04min54s	0h03min48s	22.45	0h04min01s	18.03
libpng	1h08min41s	1h06min20s	3.42	1h06min33s	3.11
openssl	2h19min29s	2h13min15s	4.47	2h05min49s	9.80
libxml2	0h00min21s	0h00min13s	38.10	0h00min13s	38.10
sqlite	0h22min12s	0h01min45s	92.12	0h01min47s	91.97
woff2	0h29min12s	0h27min54s	4.45	0h27min39s	5.31
re2	0h07min14s	0h01min59s	72.58	0h01min54s	73.73
pcre2	0h21min57s	0h20min47s	5.32	0h08min57s	59.23
proj4	0h00min16s	0h00min10s	37.50	0h00min10s	37.50

表 4　各方案的内存消耗数据对比

Tab. 4　Comparison of memory consumption among different optimizations

程序名称	original	taint		symbolized		last		function		part	
	内存消耗/MB	内存消耗/MB	减少率/%	内存消耗/MB	减少率/%	内存消耗/MB	减少率/%	内存消耗/MB	减少率/%	内存消耗/MB	减少率/%
boringssl	473	189	60.04	189	60.04	451	4.65	183	61.31	183	61.31
guetzli	701	124	82.31	119	83.02	696	0.71	107	84.74	115	83.59
libpng	983	776	21.06	718	26.96	803	18.31	711	27.67	710	27.77
openssl	755	512	32.19	539	28.61	598	20.79	535	29.14	535	29.14
libxml2	464	99	78.66	98	78.88	463	0.22	93	79.96	93	79.96
sqlite	856	179	79.09	180	78.97	853	0.35	165	80.72	165	80.72
woff2	214	111	48.13	142	33.64	196	8.41	134	37.38	137	35.98
re2	375	129	65.60	148	60.53	368	1.87	143	61.87	143	61.87
pcre2	345	264	23.48	262	24.06	313	9.28	255	26.09	145	57.97
proj4	354	79	77.68	79	77.68	354	0.00	73	79.38	73	79.38

表 5　各方案的失败分支数据对比

Tab. 5　Comparison of failed branches among different optimizations

程序名称	original	taint		symbolized		last		function		part	
	失败分支	失败分支	增长率/%	失败分支	增长率/%	失败分支	增长率/%	失败分支	增长率/%	失败分支	增长率/%
boringssl	139	30	-78.42	41	-70.50	0	-100.00	32	-76.98	40	-71.22
guetzli	10	19	90.00	4	-60.00	0	-100.00	9	-10.00	10	0
libpng	384	384	0.00	384	0.00	0	-100.00	384	0.00	384	0.00
openssl	802	737	-8.10	743	-7.36	169	78.93	743	-7.36	702	-12.47
libxml2	0	0		0		0		0		0	
sqlite	130	0	-100.00	0	-100.00	12	-100.00	0	-100.00	0	-100.00
woff2	12	0	-100.00	15	25.00	0	-100.00	12	0	11	-8.33
re2	0	0		0		0		0		0	
pcre2	118	110	-6.78	113	-4.24	0	-100.00	111	-5.93	26	-77.97
proj4	0	0		0		0		0		0	

表 6　各方案的代码路径覆盖数据对比

Tab. 6　Comparison of path coverage among different optimizations

程序名称	original	taint		symbolized		last		function		part	
	代码覆盖	代码覆盖	提升率/%	代码覆盖	提升率/%	代码覆盖	提升率/%	代码覆盖	提升率/%	代码覆盖	提升率/%
boringssl	839	896	6.79	839	0.00	1 141	36.00	865	3.10	839	0.00
guetzli	154	154	0.00	154	0.00	168	9.09	154	0.00	154	0.00
libpng	847	847	0.00	847	0.00	848	0.12	863	1.89	863	1.89
openssl	1 032	1 028	-0.39	1 032	0.00	1 069	3.59	829	-19.67	829	-19.36
libxml2	2 213	2 213	0.00	2 213	0.00	2 251	1.72	2 213	0.00	2 213	0.00
sqlite	3 170	3 191	0.66	3 191	0.66	3 482	9.84	3 193	0.73	3 191	0.66
woff2	181	181	0.00	181	0.00	199	9.94	181	0.00	181	0.00
re2	2 038	2 037	-0.05	2 038	0.00	1 908	-6.38	2 047	0.44	2 047	0.44
pcre2	737	746	1.22	737	0.00	353	-52.10	357	-51.56	357	-51.56
proj4	33	33	0.00	33	0.00	143	333.33	110	233.33	110	233.33

表7所示为10个被测程序经5个不同优化方案在3个评价指标上与原始方案相比平均提升的百分比统计。

表7 5种优化方案与原始方案平均提升效果比较

Tab.7 Comparison of 5 optimization patterns with original pattern

%

优化方案	时间效率	内存开销	覆盖率提升
taint	40.57	56.82	0.82
symbolized	31.22	55.24	0.07
last	78.25	6.46	34.52
function	31.53	56.83	16.83
part	37.09	59.77	16.54

3.1 实验结果分析

首先,从时间效率指标来看,symbolized 和 function 两种优化方案有差不多的提升效果,分别是31.22% 和31.53%;taint 和 part 比前二者的效果略好,分别为40.57%和37.09%;在这个指标上表现最好的是 last 方案,提升了78.25%,是 symbolized 和 function 的两倍多。

其次,从内存开销指标来看,part、function、taint、symbolized 这四种方案提升的效果差不多,依次为59.77%、56.83%、56.82% 和55.24%。而 last 方案在这个评价指标上表现最差,只有6.46% 的提升,收效甚微。

最后,从覆盖率提升指标来看,各方案的表现差别比较大。last 的表现最好,提升高达34.52%;function 次之,达到了16.83%;处于中等的是 part,提升了16.54%;效果比较差的是 taint 和 symbolized,平均提升只有0.82% 和0.07%,几乎只是维持了相同的覆盖率。

3.2 差异性原因分析

深入分析每一个被测程序的具体测试数据能更清楚反映不同方案的提升和不足在哪里以及造成差异性的具体原因。

3.2.1 失败分支

对于不同的输入,有的测试会产生“失败分支”,这是因为当分支约束过于复杂时,测试方案没有办法在指定的“超时”限制(10 s)内完成约束求解,所以只能放弃。而对于5种优化方案,因为约束减少,会有很多之前无法求解的“失败分支”被求解出来,进而造成覆盖率提升。而有的输入不会产生“失败分支”是因为它们的约束本来就较为简单,之前所有的分支都能够被求解,所以当施加5种优化方案时,理论上并不会产生新的覆盖率,只会减少时间和空间开销。此外,因为 last 方案丢弃掉了除了正在分析的分支之前的所有约束,其约束缩减的程度是所有方案中最大的,所以时间效率的提升非常大,对于有“失败分支”的输入,其覆盖率的提升也很大。从 boringssl 运行的结果可以看到,last 方案覆盖率提升达36%,“失败分支”也从原始模型中的139条减少到0条,即这些“失败分支”都被 last 方案解决了,这说明 last 方案更适合于约束过大导致很多分支不能在规定时间内求解的输入。然而对于没有“失败分支”的输入,由于分支约束准确性遭到了破坏,往往会造成覆盖率的大幅丢失。

3.2.2 覆盖率降低

有的情况下,由于优化方案通常以丢失上下文信息为代价,会导致部分约束求解不正确,进而导致覆盖率减少。这种因为丢失信息导致约束求解不正确进而影响覆盖率的情况,在 last 方案中尤为突出。从表6中 pcre2 运行的结果可以看出,last 方案造成了52.10%的覆盖率损失,这说明 last 方案的确对上下文信息缺失太多,以至于会损害约束的正确性,造成无法正确求解,导致覆盖率降低。此外, taint、function、part 方案也均会造成覆盖率损失。表6中 taint 方案在 openssl 和 re2 上分别造成了0.39% 和0.05% 的覆盖率损失,这是由污点分析自身的污染不足问题所导致的。由于污染不足,有些本该被识别为污点数据的指令,没有显示为“污染的”,因此在 taint 方案中,这些指令的约束就会被忽略,从而造成信息丢失,影响约束求解,导致覆盖率下降。从 openssl 程序测试结果还可以看到,function 方案和 part 方案分别产生了高达19.67% 和19.36% 的覆盖率损失,这些都是由上下文缺失造成约束不完整而导致的问题。对于 function 方案来说,其省略了与系统调用交互的部分,即这一部分的约束被视为实际化的,一旦环境有所变化,就会对实际化值的准确性造成影响,也就是因为忽略了本该计算的约束而导致约束不完整。由于 part 方案也依赖污点分析去发现程序模块与输入之间的关系,所以其也受困于污染不足的问题。

综上所述,得出以下结论:

1)last 方案在时间效率和覆盖率提升方面都展现出了最优的效果,但在内存开销方面表现不佳;而覆盖率方面,由于缺失大量信息,即使覆盖率提升,也有很多约束求解不正确,导致新增加的

程序执行路径并没有全部覆盖在期望的地方;另外,此方案更适合约束较为复杂、可能产生更多“失败分支”的输入。

2)taint 方案在时间效率和内存开销方面都有很大提升,但对于覆盖率效果不升反降。symbolized 方案表现相近,虽然对覆盖率提升非常有限,但对时间效率和内存开销的提升都比较明显,更适合约束不复杂且不会产生“失败分支”的输入。

3)function 和 part 方案是能力最均衡的两个方案,不仅对时间效率和内存开销都有提升,覆盖率的提升也比较稳定,而且相对于 last 方案来说损失的信息并没有那么大,属于一个比较折中的选项。

4　结论

从上述测试结果中可以得出结论:基于混合符号执行的以上 5 种优化方案都可以消除不必要的约束并减少时间开销和空间开销。对于大多数情况,由于本来求解失败的分支在约束缩减后可以得到求解,覆盖率也有所提升。然而,大部分时候约束缩减会造成信息丢失,导致约束求解的正确性遭到破坏,进而造成覆盖率相比本来的目标有所减少。尤其是对于“乐观求解”策略,损失的情况最为严重。总的来说,已有工作对于混合符号执行的优化,实际上是一个用正确性来换时间和空间效率的折中方案。从目前研究进展看,还难以提出一种完美的解决方案,只能视情况做适当的折中选择。针对不同测试需求,本文提出如下三种选择方案。

选择方案一:针对一些比较小的输入,由于约束比较简单,不会有太多求解失败的分支,方案选择应该更重视正确性而不是时间效率。而对于一些较大的输入,由于约束较为复杂,在求解中会产生较多失败分支,可以适当牺牲正确性,尽量提升效率,从而求解更多的分支。

选择方案二:根据混合符号执行的使用目的来定。如果使用混合符号执行是为了辅助模糊测试,即混合模糊测试,那么发现新路径数量的优先级应当高于质量,只要能够在一定时间内产生足够多新的测试用例,损失部分精确性是完全可以接受的,此时方案选择应重视效率多过正确性。然而,如果混合符号执行被用于漏洞分析、自动化利用生成等强调单条路径分析准确性的场景,那么应该牺牲效率换取分析的正确性,因为一旦分析存在偏差,整个测试都会受到影响。

选择方案三:根据对 3 个评价标准的重视程度来定。基于本文测试数据,可以给出如下不同评价标准的优先排名次序:对于时间效率指标来说,last > taint > part > function > symbolized > original;对于内存开销指标来说,part > function > taint > symbolized > last > original;对于覆盖率提升指标来说,last > function > part > taint > symbolized > original。因此,在选择方案时,可以根据最看重的评价指标来决定。

以上三种方案选择虽然没有解决效率与正确性无法兼得的问题,但对于目前的混合符号执行优化来说,依然存在显著的提升空间,即在提升执行效率和求解效率的同时,仍然保持了一定程度上的约束正确性。从本质上来说,解决这个问题的关键在于,去除不必要的约束,并确保这些约束真的是“不必要的”。

参考文献(References)

[1] GROVE W. Vulnerability and threat trends report 2020: key findings[R/OL]. (2020 - 02 - 12)[2021 - 02 - 02]. https://lp. skyboxsecurity. com/rs/440MPQ510/images/Skybox_Report_2020VT_Trends. pdf. 2020.

[2] BOYER R S, ELSPAS B, LEVITT K N. SELECT—a formal system for testing and debugging programs by symbolic execution[C]//Proceedings of the International Conference on Reliable Software, 1975.

[3] MILLER B P, FREDRIKSEN L, SO B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32 - 44.

[4] 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]//Proceedings of IEEE Symposium on Security and Privacy, 2010.

[5] GODEFROID P, KLARLUND N, SEN K. DART: directed automated random testing[J]. ACM SIGPLAN Notices, 2005, 40(6): 213 - 223.

[6] WANG H J, XIE X F, LI Y, et al. Typestate-guided fuzzer for discovering use-after-free vulnerabilities [C]// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE), 2020.

[7] MATHIS B, GOPINATH R, MERA M, et al. Parser-directed fuzzing [C]//Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019.

[8] ASCHERMANN C, SCHUMILO S, BLAZYTKO T, et al. REDQUEEN: fuzzing with input-to-state correspondence [C]// Proceedings of Network and Distributed System Security Symposium, 2019.

[9] YOU W, WANG X Q, MA S Q, et al. ProFuzzer: on-the-fly input type probing for better zero-day vulnerability discovery [C]//Proceedings of IEEE Symposium on Security and Privacy (SP), 2019.

[10] GAN S T, ZHANG C, CHEN P, et al. GREYONE: data flow sensitive fuzzing [C]//Proceedings of USENIX Security

Symposium, 2020.

[11] YUN I, LEE S, XU M, et al. QSYM: a practical concolic execution engine tailored for hybrid fuzzing[C]//Proceedings of the 27th USENIX Security Symposium, 2018.

[12] SAUDEL F, SALWAN J. Triton: a dynamic symbolic execution framework[C]// Proceedings of Symposium sur la Sécurité des Technologies de l'Information et des Communications, SSTIC, France, Rennes, 2015: 31 – 54.

[13] HALLER I, SLOWINSKA A, NEUGSCHWANDTNER M, et al. Dowsing for overflows: a guided fuzzer to find buffer boundary violations[C]//Proceedings of the 22nd USENIX Conference on Security, 2013.

[14] ZHAO L, DUAN Y, YIN H, et al. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing [C]//Proceedings of Network and Distributed System Security Symposium, 2019.

[15] KIM J, YUN J. Poster: directed hybrid fuzzing on binary code [C]//Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2019.

[16] PENG J Q, LI F, LIU B C, et al. 1dVul: discovering 1-day vulnerabilities through binary patches [C]//Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2019.

[17] CHEN Y, LI P, XU J, et al. SAVIOR: towards bug-driven hybrid testing [C]//Proceedings of IEEE Symposium on Security and Privacy (SP), 2020: 1580 – 1596.

[18] LIANG H, JIANG L, AI L, et al. Sequence directed hybrid fuzzing [C]// Proceedings of IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020: 127 – 137.

[19] STEPHENS N, GROSEN J, SALLS C, et al. Driller: augmenting fuzzing through selective symbolic execution [C]//Proceedings of the 23rd Annual Network and Distributed System Security Symposium, 2016.

[20] CADAR C, DUNBAR D, ENGLER D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs [C]//Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008.

[21] ANAND S, ORSO A, HARROLD M J. Type-dependence analysis and program transformation for symbolic execution[C]// Proceedings of the 13th International Conference On Tools and Algorithms for the Construction and Analysis of Systems, 2007.

[22] 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.

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

[24] SHOSHITAISHVILI Y, WANG R Y, SALLS C, et al. SOK: (state of) the art of war: offensive techniques in binary analysis[C]//Proceedings of IEEE Symposium on Security and Privacy (SP), 2016: 138 – 157.

[25] VUzzer(64) Version 1.0 [CP/OL]. [2021 – 02 – 02]. https://github.com/vusec/vuzzer64/.