硕士学位论文

模糊测试与符号执行相结合的漏洞发现技术研究

RESEARCH ON VULNERABILITY DETECTION TECHNOLOGY BASED ON FUZZING TEST AND CONCOLIC EXECUTION

宋博宇

哈尔滨工业大学

2017 年 6 月

国内图书分类号：TP315　　学校代码：10213
国际图书分类号：681.5　　密级：公开

工程硕士学位论文

模糊测试与符号执行相结合的漏洞发现技术研究

硕士研究生：宋博宇
导　　师：张伟哲 教授
申请学位：工程硕士
学　　科：计算机技术
所在单位：计算机科学与技术学院
答辩日期：2017 年 6 月
授予学位单位：哈尔滨工业大学

Classified Index: TP315
U.D.C: 681.5

Dissertation for the Master Degree in Technology

RESEARCH ON VULNERABILITY DETECTION TECHNOLOGY BASED ON FUZZING TEST AND CONCOLIC EXECUTIONU

Candidate： Song Boyu
Supervisor： Prof. Zhang Weizhe
Academic Degree Applied for： Master of Science
Speciality： Computer Technology
Affiliation： School of Computer Science and Technology
Date of Defence： June, 2017
Degree-Conferring-Institution： Harbin Institute of Technology

摘 要

内存损坏漏洞是软件中始终存在的风险，攻击者可以利用其获得未经授权的权限访问机密信息。随着软件对敏感数据的访问变得越来越普遍，潜在的可利用漏洞的系统也越来越多，导致越来越需要软件工具进行自动检查。

现有的寻找潜在漏洞的技术包括静态分析，动态分析，和符号执行分析，各有其优缺点。这些系统创建输入以触发缺陷的设计大多都只能找到浅层的漏洞，并且它们都努力尝试探索可执行文件更深层的路径。

本文给出的漏洞挖掘系统，是一种混合的漏洞挖掘工具，它以一种互补的方式平衡了模糊测试（Fuzzing）和可选择的符号执行（Selective Concolic Execution），用来找到更深层的漏洞。低成本的模糊测试用来探索应用隔区，符号执行测试用来生成通过隔区检查且复杂的输入。

通过结合两种技术各自的优势，我们弥补了他们各自特定的缺陷：既有效避免了符号执行原有的路径爆炸，又充分解决了模糊测试的盲目性、不完备性。本系统使用可选择的符号执行，只去探索被模糊器认为感兴趣的路径，并为模糊器无法通过的检查生成有效输入，并对其作出了预约束控制、探索缓存等优化，大大提升了系统的执行性能。最后本文在 CGC 大赛上发布的 126 个含有漏洞的应用程序上对本系统进行了测试评估，在有效时间内展现了比模糊测试、符号执行单独两种方法更良好的性能。

关键词：漏洞挖掘，模糊测试，混合符号执行

Abstract

Memory corruption vulnerabilities are an ever- present risk in software, which attackers can exploit to obtain unauthorized access to confidential information. As products with access to sensitive data are becoming more prevalent, the number of potentially exploitable systems is also increasing, resulting in a greater need for automated software vetting tools.

Current techniques for finding potential bugs include static, dynamic, and concolic analysis systems, which each having their own advantages and disadvantages. A common limitation of systems designed to create inputs which trigger vulnerabilities is that they only find shallow bugs and struggle to exercise deeper paths in executables.

We present a hybrid vulnerability excavation tool which leverages fuzzing and selective concolic execution in a complementary manner, to find deeper bugs. Inexpensive fuzzing is used to exercise compartments of an application, while concolic execution is used to generate inputs which satisfy the complex checks separating the compartments.

By combining the strengths of the two techniques, we mitigate their weaknesses, avoiding the path explosion inherent in concolic analysis and the incompleteness of fuzzing. Driller uses selective concolic execution to explore only the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer cannot satisfy. We evaluate Driller on 126 applications released in the qualifying event of the DARPA Cyber Grand Challenge and show its efficacy by identifying more number of vulnerabilities.

keywords: vulnerabilities defect, fuzzing, concolic execution

目 录

摘 要.....I
Abstract.....II
第 1 章 绪 论.....1
1.1 课题来源及研究的背景和意义.....1
1.1.1 课题研究的背景.....1
1.1.2 课题研究的目的和意义.....2
1.2 国内外研究现状.....3
1.2.1 有指导的模糊测试.....3
1.2.2 白盒模糊测试.....4
1.2.3 混合符号执行.....5
第 2 章 漏洞发现系统概述.....7
2.1 系统概念.....7
2.2 系统模块构成.....7
2.3 系统流程实例.....8
2.4 系统局限性分析.....10
2.5 本章小结.....13
第 3 章 模糊测试.....14
3.1 模糊测试关键技术及实现.....15
3.1.1 协议及文本数据的生成.....15
3.1.2 异常输入数据的构造.....16
3.1.3 误报自动分析.....17
3.2 模糊器的特点.....17
3.3 模糊器的局限性.....18
3.4 模糊器模块设计实现.....19
3.4.1 模糊器模块架构设计.....19
3.4.2 模糊器模块组成.....20
3.4.3 模糊器模块执行流程.....20
3.5 模糊器跳转到符号执行.....21
3.6 本章小结.....22
第 4 章 混合符号执行.....23
4.1 符号执行引擎.....23
4.2 符号执行关键技术及实现.....23
4.2.1 代码插桩.....23
4.2.2 符号树生成.....25
4.2.3 约束生成.....26
4.2.4 路径空间探索.....26
4.2.5 约束条件优化.....28
4.3 符号执行的局限性.....29

4.4 漏洞发现系统符号执行设计及优化.....30
4.4.1 符号执行前的预约束控制.....31
4.4.2 引入缓存探索器.....33
4.4.3 符号执行中的地址转换.....34
4.5 本章小结.....34
第 5 章 漏洞发现系统任务管理.....35
5.1 系统配置管理.....35
5.2 模糊器到符号执行过程.....35
5.3 任务协作中的数据流转移.....37
5.4 本章小结.....38
第 6 章 漏洞发现系统评估实验.....39
6.1 实验数据集.....39
6.2 实验过程.....39
6.3 漏洞发现量比较.....40
6.4 状态转移覆盖分析.....42
6.5 程序隔区覆盖分析.....43
6.6 具体实例分析.....45
6.7 本章小结.....50
结 论.....51
参考文献.....52
哈尔滨工业大学学位论文原创性声明和使用权限.....56
致 谢.....58

第 1 章 绪　论

1.1 课题来源及研究的背景和意义

1.1.1 课题研究的背景

尽管针对安全漏洞在增强软件的韧性方面做了很大的努力，软件中的缺陷仍是普遍存在的。事实上，近年来安全漏洞的出现已经增加到了历史新高[28]。此外，尽管引入了内存损坏和执行重定向缓解技术，软件缺陷仍占去年发现的所有的漏洞的三分之一[14]。

这些漏洞以前被想要推动安全限制并揭露保护无效的独立黑客利用，然而在现代世界，已经转移到国家和网络犯罪分子，他们利用这些漏洞来获得战略优势或利益。此外，随着物联网的兴起，运行着潜在的易受攻击软件的设备数量激增，并且在这些设备运行的软件中越来越多的发现了漏洞[29]。

虽然许多漏洞是人工发现的，但手动分析并不是一种可扩展的漏洞评估方法。为了跟上必须审查漏洞的软件数量，必然需要一个自动化的方法。DARPA 最近通过赞助两项工作来支持这一目标：VET，一个致力于二进制固件分析开发技术的程序，以及 Cyber Grand Challenge（CGC），参与者设计和部署自动化漏洞扫描引擎，它将通过利用二进制软件漏洞相互竞争。表明了其对开发可行的自动化二进制分析方法的强烈兴趣。

安全研究人员一直在积极地设计自动化漏洞分析系统，存在许多方法，分为三个主要类别：静态，动态和符号执行分析系统。这些方法具有不同的优点和缺点。静态分析系统可以提供可证明的保证，即静态分析系统可以确定地表明给定的二进制代码段是安全的。然而，这样的系统具有两个基本缺点：它们不精确，导致大量错误的判断，并且它们不能提供“可操作的输入”（即可以触发检测到的漏洞的特定输入的示例）。动态分析系统，例如“模糊器”（Fuzzers），监控应用程序的本地执行以识别缺陷。当检测到缺陷时，这些系统可以提供可操作的输入来触发它们。然而，这些系统需要“输入测试用例”来驱动执行。没有全面的测试用例集，这需要大量的人工来生成，这种系统的可用性是有限的。最后，混合符号执行（Concolic Execution）引擎利用程序解释和约束求解技术来生成输入，以探索二进制的状态空间，以尝试到达和触发缺陷。然而，因为这样的系统能够在二进制程序中触发大量的路径（即对于条件分支，它们通常创建使得分支

被采取的输入和不被采取的输入），它们受“路径爆炸”约束，这极大限制了它们的可扩展性。

1.1.2 课题研究的目的和意义

由于这些缺点，现代自动化分析系统产生的大多数用来触发缺陷的输入仅能表现软件中的浅层错误。在模糊器的情况下，这是因为模糊器随机地向应用产生新的输入，并且它们可能不能成功地通过输入处理代码。另一方面，符号执行引擎通常能够重新创建正确格式化的输入以通过输入处理代码，但是由于约束于路径爆炸，限制了它们可以分析的代码的“深度”。 因此，位于应用的更深层逻辑中的缺陷往往被这些工具所遗漏，并且通常通过人类专家的人工分析来发现[3,9,13]。

通过模糊测试和符号执行找到的不用漏洞类型之间的差异，也可以通过应用程序处理用户输入的方式来查看。我们提出两种不同类别的用户输入：一般输入，其具有大范围的有效值（例如，用户的名称）和特定输入，其具有有限的有效值集合（例如，上述用户名称的散列值）。应用程序对特定输入的特定值的检查有效地将应用程序分割成由这样的检查分隔的隔区。模糊测试精通于在一个区域内探索一般输入的可能值，但是很难找出所需的精确值，以满足对特定输入的检查和驱动隔区之间的执行流程。另一方面，选择性符号执行精确地确定这种特定检查所需的值，并且如果路径爆炸问题被解决，则可以推动隔区之间的执行流程。

例如，考虑处理来自用户的命令的应用程序：应用程序读取命令名称，将其与命令列表进行比较，并将用户提供的参数传递给适当的命令处理器。在这种情况下，复杂检查将是命令名称的比较：模糊器随机变化的输入将有非常小的机会发送正确的输入。另一方面，符号执行引擎将非常适合于找到正确的命令名称，但可能在参数处理代码中遭受路径爆炸。一旦确定了正确的命令名称，模糊器就更适合于探索可以发送的不同命令参数，而不会遇到路径爆炸。

我们意识到这中场景可以结合两种分析技术，利用他们的优势，同时减轻他们的弱点。例如，模糊器可以用于探索应用的初始隔间，并且当其不能进一步深入时，可以利用符号执行引擎来引导它到下一个隔间。 一旦到达，模糊器可以再次接管，探索可以提供给新隔区的可能输入。当模糊器再次停止运行时，符号执行引擎可以恢复并将分析引导到下一个隔间。通过重复这样做，执行被驱动的

越来越深入程序，限制了符号执行固有的路径爆炸，并且改善动态分析的不完备性。

在这种理念的引导下，我们创建了一个漏洞挖掘系统，结合遗传输入-突变模糊器和选择性的符号执行引擎，以识别二进制中的深层漏洞。 结合这两种技术，系统可以以可扩展的方式运行，并绕过输入测试用例的需要。在本文中，我们将描述漏洞挖掘系统的设计和实施，并评估其在 DARPA Cyber Grand Challenge 的资格赛发布的 126 个应用程序上表现的性能。

本系统不是第一个结合不同分析方法的工作。 然而，现有技术大多数只支持非常特定类型的漏洞（而本系统现在可以检测到可能导致程序崩溃的任何漏洞）[21,25],没有充分利用动态分析提供的功能（特别是 fuzzing）[19]，或受路径爆炸问题的影响[4,8,10,20].。我们可以通过对比与模糊测试或者符号执行单独运行，本系统在二进制文件中可以识别更多的漏洞，最后我们通过实验数据，以表明如果没有本系统的作用（即使用传统的模糊测试或符号执行方法）这是不可能的。

1.2 国内外研究现状

1.2.1 有指导的模糊测试

模糊测试最初是作为测试 UNIX 系统程序的几个工具之一引入的[23]。 从那时起，它已广泛用于应用的黑盒安全测试。然而，模糊测试缺乏引导 - 基于先前输入的随机突变产生新的输入，而不能控制应用中的哪个路径应该被定向。

模糊测试是一种自动、黑盒测试的形式，其使用模糊器来随机生成或突变样本输入。这种技术已经显示出令人惊讶地有效地揭露软件系统中的错误[17,23,16]。它特别适用于测试输入解析组件，随机生成的输入通常会在初始解析和错误检查代码中执行忽略的角落。

为了更好地将模糊器指向特定类别的漏洞，产生了有指导的模糊测试的概念。例如，许多研究尝试通过选择性地选择最优测试用例来改进模糊测试，在目标二进制中包含的代码的感兴趣区域上进行提升[21,25]。具体来说，Dowser [21]使用静态分析首先识别可能导致涉及缓冲区溢出的漏洞的代码区域。为了分析代码，Dowser 对可用的测试用例进行污点跟踪，以确定这些代码区域处理哪些输入字节，并对代码区域进行符号化的探索。不幸的是，Dowser 有两个缺点：需要测试用例到达包含内存损坏漏洞的代码区域，并且它只支持缓冲区溢出漏洞。与 Dowser 不同，本系统支持任意漏洞规范（虽然当前实现的重点是导致崩溃的

漏洞），并且不需要输入测试用例。此外，Dowser 仍然遭受符号执行的路径爆炸问题，而本系统通过使用模糊测试来缓解这个问题。

与 Dowser 类似，BuzzFuzz [17]对抽样输入测试用例应用污点跟踪，以发现哪些输入字节由系统定义的“攻击点”处理，通常是系统调用参数和库代码。 与 BuzzFuzz 不同，本系统不依赖于到达易受攻击代码的输入测试用例，也不依赖于系统定义的攻击点。2009 邵林结合 fuzzing 技术、数据流动态分析技术以及异常自动分析技术提 出一种新的缓冲区溢出漏洞发掘思路[8]，2014 年李彤，黄轩等人完成了基于 Fuzzing 的软件漏洞发掘技术[7]。

在改进模糊状态的另一个尝试中，Flayer[15]允许系统在目标应用程序中随意跳过复杂的检查。这允许系统在应用程序中进行更深地模糊逻辑，而不需要编制符合目标所要求格式的输入，减掉了花费在寻找防止崩溃的合法输入的时间。类似地，Taintscope 使用校验和检测算法从应用程序中删除校验和代码，有效地“修补”分支断言，这是很难通过突变方法满足的[30]。这使得模糊器能够处理特定类别的困难约束。然而，这两种方法中，Flayer 的方法需要大量人工指导，另一种需要人工来确定在崩溃分类期间的错误正面判断。本系统不修改目标应用程序的任何代码，意味着发现的崩溃不需要深入搜索，此外本系统不需要人为干预，因为它试图使用其符号执行后端来发现良好形式的输入。

另一种方法是混合模糊测试，其中有限的符号探索被用来找到“边界节点”[26]。然后采用模糊测试来执行具有随机输入的程序，随机输入被预先约束以遵循通向边界节点的路径。此方法对于确保模糊输入在二进制程序执行的早期采用不同路径很有用，但是它不处理程序中分离隔区时更深入的复杂的检查。

1.2.2 白盒模糊测试

其他系统尝试将模糊与符号执行混合以获得最大代码覆盖[6,7,19,20]。 这些方法倾向于通过符号执行由模糊引擎产生的输入，收集置于该输入上的符号约束，然后否定这些约束以产生将采取其他路径的输入来增加模糊。然而，这些工具不具有本系统的关键点，符号执行最好用于复原输入以驱动代码在应用程序隔区之间执行。没有这种关键点，符号执行的独特能力被浪费在隔间内创建分歧路径。这些工具本质上是以串行化方式工作的符号执行引擎，每次一个路径，因此，它们深深受到路径爆炸问题的影响。

本系统在许多实现细节中类似，可以将主要的独特的路径发现装载到模糊引擎中。本系统限制了高消耗的符号执行调用，以满足允许我们的模糊器进入另外

的隔区。由于我们只使用符号执行来生成模糊器无法自己生成的基本转换块，所以符号执行引擎只处理可管理的输入数量。相反，上述工具使用符号执行重复地否定约束，缓慢地分析指数级增加的转换次数，其中大多数可以由模糊器更有效地分析。

但是，模糊测试（或简单的模糊测试）在生成语法合法输入时不太有效，这些输入在程序中暴露更深层次的语义错误[23]，对于许多程序，几乎所有随机生成的输入都不能满足表征良好的基本语法约束形成的输入，因此不能使其超过初始解析阶段来执行剩余代码。

1.2.3 混合符号执行

随着近年来计算能力的不断增加，动态符号执行已经普及。由 EXE 引入[5]，于 KLEE 完善[4]，并由 Mayhem[8]和 S2E[10]应用于二进制代码，符号执行引擎解释应用程序，使用符号变量的模拟用户输入，跟踪条件跳转产生的约束，并使用约束求解器创建输入以驱动程序代码按特定路径向下执行。虽然这些系统是强大的，但是它们存在一个根本问题：如果条件分支依赖于符号值，则通常满足已采用和未采用条件。因此执行状态必须分叉，并且必须探索两个路径。这快速导致了众所周知的路径爆炸问题，这是符号执行技术的主要抑制者。

2012 年梁晓兵提出的基于关联矩阵和邻接矩阵的双向路径搜索算法,可以识别出从二进制程序输入位置到关键代码区域的路径，提高了符号执行的效率[4]。最近，研究者提出了混合符号执行测试[14]作为符号执行的变体，其中符号执行与实际执行相同，即程序在具体和符号值上等效执行，并且沿着生成的符号约束 使用相应的具体值简化路径。符号约束再次用于递增地生成测试输入以通过结合具有否定条件的路径前缀的符号约束来实现更好的覆盖。混合符号执行的主要优点：纯粹的符号模拟是存在具体的数据和地址值，它们可以用于推论复杂数据结构以及约束约束超出基础约束的能力时求解。

然而实际上，无论是对于符号执行还是混合符号执行，必须被执行的符号路径的可能数量是如此之大，以至于方法最终只能探索程序状态空间的一小部分，以便可以在合理的时间内运行。此外，因为符号执行的深度增长，维护和解决执行路径的符号约束十分耗费时间。因此，以前这些技术的应用已经被限制在小的程序当中[25]，最多约五万个基本块的路径深度[18]。也就是说，尽管广泛地探索了不同的程序路径，但是符号执行和混合符号执行在探索只有在长时间执行之后才达成的深层次态度。

研究员们已经尝试了各种方法来减轻路径爆炸问题。Veritesting [1]提出了一种优化的路径合并技术来减少正在执行的路径数量，Firmalice[29]执行大量的静态分析，并将符号执行限制在小的代码段，并且与欠约束的符号执行互换精度来支持伸缩性[16,27]。然而，这些技术要么不能缓解路径爆炸问题（例如，Veritesting 的方法会延迟爆炸，但这种爆炸仍然最终发生）要么产生的输入不能直接可操作（例如，由 Firmalice 实验产生的输入满足一种特定切片约束，但是没有提供可以直接到达代码缺陷的输入）。

最后，混合式符号执行测试具有基于符号执行的测试生成的相同限制：发现未覆盖的点取决于约束求解器的可扩展性和表现力，对未覆盖点的详尽搜索受到路径数量的限制。因此，一般来说，混合符号执行可能无法实现 100％的覆盖率，但可以大大改善随机模糊测试。。

本系统尝试通过将大多数路径探索任务装载到其模糊引擎来减轻这种情况，使用符号执行只是为了满足应用程序中保护各个隔区之间的复杂检查的过渡。

第 2 章 漏洞发现系统概述

2.1 系统概念

漏洞发现系统设计背后核心是系统处理两种不同类型的用户输入：

（1）**一般输入**：表示可以采用大范围有效值的输入；

（2）**特定输入**：表示必须采用几个可能值之一的输入。

应用程序对后一种类型输入的检查，将应用程序拆分成各个隔区。执行流在隔间之间通过针对特定输入的检查来移动，而在隔间内，应用处理一般输入。

系统通过将模糊测试的速度与符号执行的输入推理能力结合起来工作。这允许系统快速探索二进制文件，不对用户输入施加复杂的要求，同时还能够处理纯粹的符号执行中存在的可扩展性问题，以及模糊测试对特定输入的复杂检查问题。我们定义“复杂”检查为那些具体的，通过模糊器对输入进行变异不能满足的检查。

2.2 系统模块构成

系统由以下多个组件组成：

（1）**测试用例生成模块**　　系统可以在没有测试用例输入的情况下运行。然而输入测试用例可以通过引导模糊器指向某些隔区来加快初始模糊测试步骤。

（2）**模糊测试模块**　　当系统执行时，它通过启动模糊引擎开始。模糊引擎探索应用程序的第一个隔区，直到它达到第一个复杂检查。在此，模糊引擎被“卡住”并且不能识别能够在程序中搜索新路径的输入。

（3）**符号执行模块**　　当模糊引擎卡住时，系统调用它的符号执行组件。该组件分析应用程序，采用先前的模糊测试步骤发现的唯一输入，来约束用户输入以防止路径爆炸。接管模糊器产生的输入之后，符号执行组件利用其约束求解引擎来识别输入，以强制执行先前未被探索的后续路径。如果模糊引擎在卡住之前遍历了之前的隔区，则这些路径代表了到新隔区的执行流。

(4)**任务管理模块**　　一旦符号执行组件识别到新的输入，则执行权被传递回模糊执行组件，其继续在这些输入上突变以探索新的隔区。系统继续在模糊测试和符号执行之间循环，直到发现使应用程序崩溃的输入。

各模块之间数据流转移如图 2-1 所示：

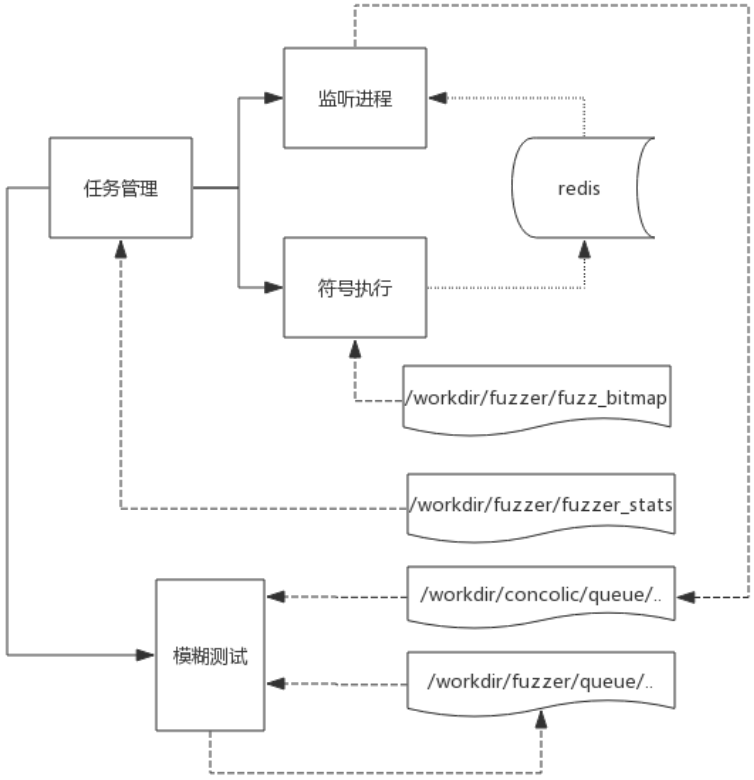


图 2-1 各模块之间数据流转移

2.3 系统流程实例

在此表 2-1 的示例中，应用程序解析通过输入流接收到配置文件，该文件中包含特定魔数。如果接收到的数据包含语法错误或不正确的数字，程序退出。否则控制流基于各隔区之间的检查切换，其中一些包含内存崩溃缺陷。

系统通过调用其模糊引擎在程序的第一个隔区开始操作。这些模糊节点在图 2-2 的程序的控制流程图中以阴影示出。该模糊步骤探索第一隔区并且停留在第一个复杂检查，即与特定魔数进行比较。然后，系统执行符号执行引擎，以识别能够通过该检查的输入，进入其他隔区。对于该示例，由图 2-3 中示出了由符号执行组件发现的额外转换。

随后，系统再次进入其模糊阶段，覆盖第二个隔间（初始化代码和对配置文件中的键的检查）。 第二模糊阶段的覆盖如图 2-4 所示。如图所示，除了默认值之外，模糊器找不到任何关键值。当第二次模糊调用被阻塞时，系统利用其符号执

行引擎来发现“*crashstring*”和“*set_option*”输入，如图 2-5 所示。前者直接导致二进制中的错误。

表 2-1. 需要模糊测试和符号执行一起工作的示例

```
1   int main(void){
2           config_t* config = read_config();
3       if(config == NULL){
4               puts("Configuration syntax error");
5               return 1;
6           }
7           if(config->magic != MAGICNUMBER){
8               puts("Bad magic number");
9               return 2;
10          }
11          initialize(config);
13          char* directive = config->directives[0];
14          if(!strcmp(directive, "crashstring")){
15              program_bug();
16          } else if(!strcmp(directive, "set_option")){
18              set_option(config->directives[1]);
19           } else{
21              default();
22          }
24          return 0;
25  }
```

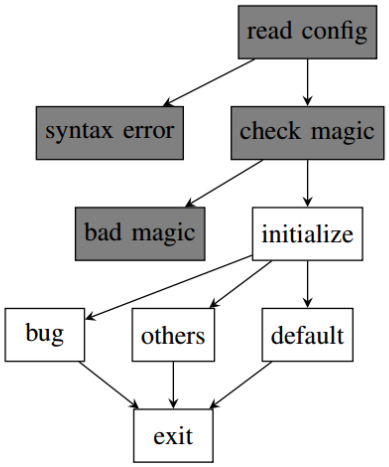


图 2-2. 最初由模糊器找到的节点

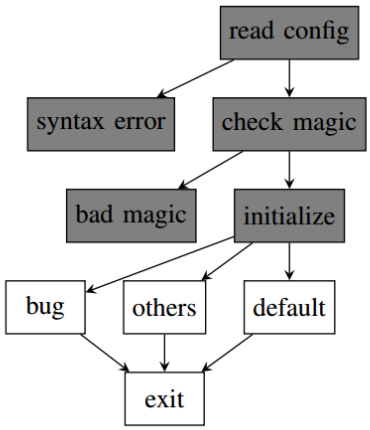


图 2-3. 第一次调用符号执行所找到的节点

虽然符号执行和模糊测试单独都不能发现这个 bug，但结合后的新系统可以。在这个例子中有几个区域需要系统的混合方法。解析配置和初始化代码具有大量

的复杂控制流推理，这将导致路径爆炸，从而将符号执行减慢到毫无作用。此外如前所述，通过魔数检查要求高度特定的输入，很难在有限时间内在其搜索空间中发现，这导致传统的模糊测试方法不可能成功，阻碍模糊测试的其他常见技术还包括使用散列函数来做输入验证等。因此，符号执行和模糊测试结合具有获得更好结果的能力。

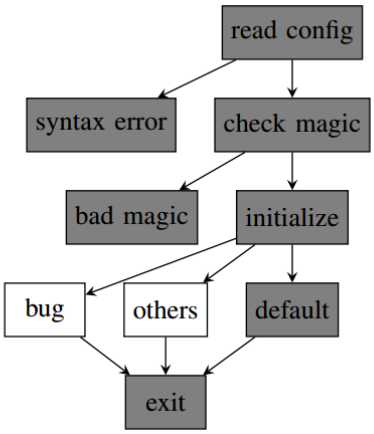


图 2-4. 模糊器发现的节点，补充了第一次结果

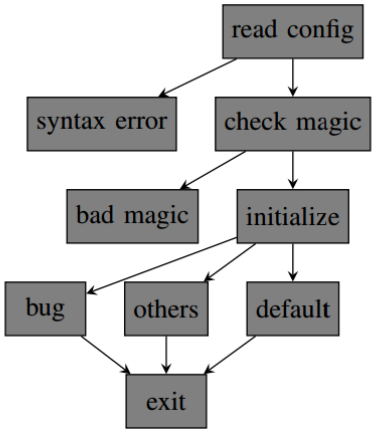


图 2-5. 第二次调用符号执行找到的节点

2.4 系统局限性分析

本系统通过利用符号执行和模糊进行统一的分析。 这允许系统用其补充来解决每个分析的一些缺点。 在本节中，我们将讨论本系统的局限性和未来的研究方向，以进一步增强自动化漏洞提取。

本系统的一个优点也是缺陷，是它借用了 AFL 的状态空间解释。AFL 通过简单地跟踪状态转换元组计算粗略的“命中计数”（遇到状态转换的次数）来表示状态。种适度的轻量状态表示，是 AFL 如此有效的原因，因为每个路径的状态仅由它遇到的状态转换元组的集合，以及它们遇到了多少次来定义。系统使用这个相同的数据结构来确定哪些状态转换值得解决。 我们在表 2-2 中举一个例子来说明本系统这方面的局限性。

表 2-2 展示了在多个命令处理程序中发生的状态转换。由于每个分支都依赖于 *static_strcmp*，AFL 本身将不能在 *static_strcmp* 的不同调用内区分状态转换。因此，系统不会尝试在第 3 行上多次解析 *if* 语句，即使它用于不同的比较。此外，具有一个或两个额外匹配字符的输入将不被 AFL 视为感兴趣。当然，如果整个字符串是由本系统发现的，AFL 会对它感兴趣，并采纳它。系统试图在每个

新的状态转换时调用的符号探索器桩（在 4.3.2 中描述）减轻这个问题的影响。然而，我们认为这是一个不完美的解决方案，最终可能需要更好的状态表示。

表 2-2. 限制发现新状态转换的最小状态表示的示例

```
1    int static_strcmp(char *a, char *b) {
2     for(; *a; a++,b++) {
3           if(*a != *b)
4                 break;
5     }
6
7     return *a − *b;
8    }
9
10   int main(void) {
11    read(0, user_command, 10);
12
13    if (static_strcmp("first_cmd", user_command) == 0) {
14          cmd1();
15    }
16    else if (static_strcmp("second_cmd", user_command) == 0) {
17          cmd2();
18    }
19    else if (static_strcmp("crash_cmd", user_command) == 0) {
20          abort();
21    }
22
23    return 0;
24   }
```

本系统的另一个限制是用户输入在一个组件中被视为通用输入，在另一个组件中被视为特定输入。考虑表 2-3 中提供的程序，此应用程序从用户读取命令和散列，并验证散列。此隔区（跨越第 1 到 11 行）将命令视为通用输入，将散列视为特定输入。然后，应用程序在多个阶段检查提供的命令是否是“CRASH！”。基本上，这会将 *user_command* 重新分类为特定输入，因为它必须完全匹配。 这触发了将系统降到符号执行引擎的情况，如下所述。

第一阶段，直接进入到隔区 3 中，系统的符号执行引擎将识别以“CRASH”开始的输入及其对应的散列（因为这是散列的向前计算，所以不必担心需要“破解”哈希; 系统只需要计算它）。然而，在这之后，模糊器将不再用于探索这个隔区。 这是因为对哈希或输入的任何随机突变将可能导致执行从分区 1 无法进行。因此，模糊器将很快被阻塞，并且系统将再次调用符号执行引

擎。这个调用将引导系统到在第 16 行的隔区 4，执行回到模糊器。 但是，模糊器将再次失败，决定它被卡住，并触发符号执行。

表 2-3. 在一个地方用作通用输入并在另一个地方用作特定输入的输入示例。崩溃输入是“CRASH！”，后面是它的哈希

```
1   int main(void) {
2    char user_command[10];
3    int user_hash;
4
5    read(0, user_command, 10);
6    read(0, user_hash, sizeof(int));
7
8    if (user_hash != hash(user_command)) {
9          puts("Hash mismatch!");
10         return 1;
11   }
12
13   if (strncmp("CRASH",usercommand,5) == 0) {
14         puts("Welcome to compartment 3!");
15         if (user_command[5] == '!') {
16               path_explosion_function();
17               if(user_command[6] == '!') {
18                     puts("CRASHING");
19                     abort();
20               }
21         }
22   }
23
24   return 0;
25  }
```

这个循环将继续，使得模糊组件无用并且基本上将系统降为符号探索。 更糟的是，在本应用程序中，隔区 4 调用导致路径爆炸的函数（*path_explosion_function*）。没有模糊引擎的缓解效果，系统将不能到达隔间 5（18 和 19 行）并触发漏洞。

这表示系统中的限制：在某些情况下，模糊组件可以被有效地禁用，这抢夺了系统的优势。减轻这个问题的潜在未来阶段是产生“半符号”模糊输入的能力。例如，符号执行引擎可以将一组约束传递给模糊器，以确保它生成的输入符合一些规范。这将利用代生模糊的概念[18]创建“输入生成器”，以帮助模糊器到达和探索应用程序隔区。

表 2-3 所示的限制显示了特定输入如何防止模糊器有效地改变通用输入。然而，对于其他类型的特定输入，即使具有多个组件，AFL 仍可以模糊较深的组件。即使在最困难的情况下，例如哈希检查，系统仍然能够改变与哈希无关的任何输入，例如检查哈希后的输入。我们已经料到系统发现多个组件后性能会有所下降。 这是因为 AFL 不知道来自符号执行引擎的约束，所以将有一小部分的模糊周期浪费在了试图突变特定输入。

2.5 本章小结

本章介绍了系统中涉及的一些自定义概念，如一般输入、特定输入、程序隔间等。并相机介绍了构成系统的各个大模块划分以及具体工作原理：测试用例生成模块、模糊测试模块、符号执行模块、任务管理模块。之后通过一个具体的例子描述了系统工作的具体流程，在这个例子中有几个区域需要系统的混合方法。解析配置和初始化代码具有大量的复杂控制流推理，这将导致路径爆炸，从而将符号执行减慢到毫无作用。此外如前所述，通过魔数检查要求高度特定的输入，很难在有限时间内在其搜索空间中发现，这导致传统的模糊测试方法不可能成功。因此，符号执行和模糊测试结合具有获得更好结果的能力。在本章最后，分析了当前系统存在的局限性。

第 3 章 模糊测试

漏洞发现系统利用了一个比较流行的开源模糊器，American Fuzzy Lop（AFL）。系统的改进主要涉及将模糊器与符号执行引擎集成，没有改变 AFL 原有的逻辑。AFL 是一个暴力方法的 fuzzer，搭配了一个极其简单但是绝对可靠的，插桩代码导向的遗传算法。它根据一种自定义的分支覆盖率，来毫不费力地识别局部的程序控制流。简单来说，整个算法的逻辑如下：

（1）将用户提供的初始测试用例加载到队列中；

（2）从队列中取下一个输入；

（3）尝试将测试用例修剪到不改变测量行为的最小尺寸；

（4）使用平衡的各种传统模糊策略重复地改变输入；

（5）如果任何产生的突变导致新的状态转换，将突变输出添加到队列中；

（6）转到（2）。

AFL 模糊器生成的状态文件中各属性意义如表 3-1 所示：

表 3-1 fuzzer_stats 文件各属性意义

属性	描述
start_time	afl-fuzz 执行的开始时间戳
last_update	对于此文件的最后更新时间戳
fuzzer_pid	Fuzzer 进程的 PID
cycles_done	到目前为止已完成的队列循环
execs_done	尝试执行 execve() 的数量
execs_per_sec	当前每秒的执行数
paths_total	队列中的路径总数
paths_found	通过模糊测试发现的路径数
paths_imported	从其他实例导入的路径数
max_depth	生成的数据集中的层数
cur_path	当前处理的路径编号
pending_favs	仍等待模糊测试的感兴趣路径数
pending_total	仍等待模糊测试的所有路径数
stability	表现一致的位图字节的百分比
variable_paths	显示不同表现的测试用例数
unique_crashes	记录的崩溃输入数量
unique_hangs	遇到的超时挂起输入数量

该工具可以在编译时引入或通过 QEMU 的用户级仿真实现，系统选择了 QEMU 实现，以消除对源代码可用性的依赖。fuzzing 过程中，会输出文件到以下目录：

（1）queue/ - 每个单独执行路径的测试用例，以及用户给出的所有开始文件。这就是所谓的合成语料库。在使用语料库用于任何其他任务时，可以使用 afl-cmin 工具将其缩小到较小的大小。

（2）crashes/ - 使得被测试程序接收致命信号（例如，SIGSEGV，SIGILL，SIGABRT）的测试用例。文件由接收的信号分组。

（3）hangs/ - 导致测试程序超时的测试用例。当默认（主动）超时设置有效时，由于延迟和其他自然现象，这可能会有轻微的噪声。

3.1 模糊测试关键技术及实现

3.1.1 协议及文本数据的生成

模糊测试的重要问题之一是初始化数据的生成，这很大程度上影响到模糊测试是否可行以及模糊器运行时间的长短。

模糊测试根据接收何种格式输入的二进制文件主要可以分为两大类：第一类是接收输入为文件格式的模糊测试（主要对象为图片格式、二进制格式、文档格式等）。第二类位接收输入为协议类型的模糊测试（主要对象为 HTTP、TCP/UDP、SMTP、FTP、ICMP 协议等）。

首先，对于接收输入为文件格式的模糊测试，本系统会生成一个初始化文件，然后通过对文件中的字符进行不同更换来产生大量的文件输入。针对文件格式的模糊测试流程如图 3-1 所示。

针对输入为协议格式的模糊测试，初始化测试数据的生产要考虑到具体接收输入的协议。否则随机生产的普通无格式输入会是模糊测试难以得到预期结果，因为程序检查接收的数据时，一旦发现其结构与特定的协议规则不相符就会将数据包舍掉，虽然模糊器生成了大量的输入，但是由于都是无格式的杂乱数据，因此不能有效的进入程序的内容分析过程和具体的解析处理过程，这样模糊器的性能将会大大降低，运行时间大幅度延长，缺陷漏报严重。因此有必要针对不同的目标程序接收协议数据的格式要求，对应生产不同格式的输入数据。

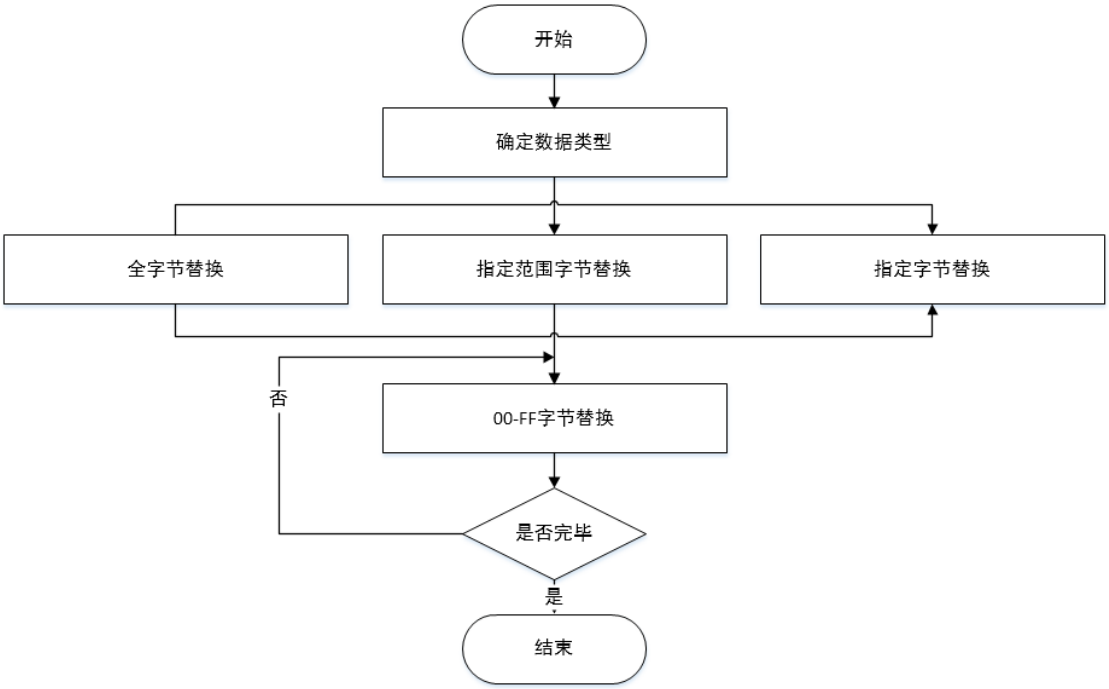


图 3-1 文件数据构造

本系统采用的做法是在生成输入之前，制定目标协议的正则表达式文件，要求严格符合目标程序的协议格式，然后模糊器按照正则表达式的要求，依次生成每一个目标字符，以达到有效进入程序命令处理过程的目的。

3.1.2 异常输入数据的构造

异常数据是指严格满足目标程序所需输入的协议格式，但是无意义且产生的结果无效的输入数据。这些数据的目的即是充分模拟异常输入，尽可能覆盖程序执行中所有的执行路径，这样才能尽量多的探索程序隔区以发现更多的漏洞。异常输入数据的构造的常用方法主要有以下几种：

（1）针对漏洞类型为格式化字符串漏洞的探索，生成数据时构造字符串通常可以为“%d%d%d”、”%s%s%d%n”等等。这样类型的字符串源词一般为“%d”“%n”“%n”等，可以通过配置文件来设定并随机组合。

（2）针对漏洞类型为缓冲区溢出漏洞的探索，异常数据的构造方法比较简单，只要输入较长的普通杂乱字符串就可以。为了便于误报自动分析，通常可以填充相同字符的超长字符串。

（3）针对漏洞类型为整数溢出漏洞的探索，采用的异常数据构造通常是填充整数 int 类型边界值，例如 0xffff、-1、1、0 等等，这样可以尽可能除法整数溢出漏洞。

以上三种特定格式的异常数据构造即是模糊测试技术产生输入时最长使用的方法，根据模糊测试中程序对象的不同还可采用其他不同的构造方法，例如除 0 异常等等。针对不同漏洞，模糊器通常采用不同的异常数据构造方法。本系统主要针对缓冲区溢出漏洞。

3.1.3 误报自动分析

通过模糊测试所找到的潜在软件漏洞往往可能会是一些小的缺陷，所以模糊测试技术都会存在一定的误报，也就是并不能构成漏洞，所以通常涉及模糊器时需要加入误报自动分析功能。误报自动分析功能主要是检测目标二进制文件出现崩溃时记录下崩溃产生的所有寄存器内容，记录下的寄存器信息是否出现了明显的缺陷特点。例如当二进制程序发生缓冲区溢出时，系统发生崩溃，EIP 寄存器存储的内容通常会被输入的杂乱数据所覆盖，由于输入数据是我们刻意生成的都是重复字符子字符串，所以这时模糊器刻意自动分析当前 EIP 寄存器中存储的值是否都是我们生成的特定字符，如果存储的都是特定字符，模糊器才会标记下这次测试为发现漏洞的测试，这样刻意大大提升模糊器发现程序漏洞的准确度。

3.2 模糊器的特点

（1）遗传模糊测试　　AFL 通过遗传算法生成输入，根据遗传规则（转录，插入等）突变输入并通过适应度函数　对它们排序。 对于 AFL 来说，适应度函数基于唯一的代码覆盖率，即触发不同执行路径的能力。

（2）状态转换跟踪　　AFL 跟踪它从输入得到的控制流转移集合（源地址和目的地址元组）。基于新控制流转移的发现，导致程序以不同方式执行的输入在之后的生成过程中获得高优先级。

（3）循环分桶　　处理循环对于模糊引擎和符号执行引擎是一个复杂的问题。为了帮助减少循环的路径空间大小，执行以下启发式算法：

a) 当 AFL 检测到路径包含循环迭代时，触发辅助计算以确定该路径是否应当有资格进行育种。

b) AFL 确定执行中的循环迭代的数量，并将其与之前的，能产生通过相同循环路径的输入进行比较。

c) 这些路径都根据它们的循环迭代计数的对数（1,2,4,8 等等）被放置到“桶”中。 每个桶中选出一条路径被考虑用于遗传算法中的育种。

d) 这样，与 *N* 条路径的普通方法相比，每个循环只需要考虑 *log(N)*调路径。

（4）去随机化 程序随机化会干扰遗传模糊器对输入的评估 - 在给定的随机数种子下产生感兴趣路径的输入，可能不会在另一随机数种子下产生。所以系统预先设置 AFL 的 QEMU 到一个特定的随机种子，以确保一致的执行。之后，当发现崩溃输入时，系统使用符号执行引擎来复原依赖于随机性漏洞的缺陷。

3.3 模糊器的局限性

因为模糊器随机对输入进行变异，所以它们能够快速发现处理“一般”输入而产生的不同路径（即许多不同值的输入都可以触发后续执行的检查）。然而，生成通过程序中复杂检查的“特定”输入对于模糊器是非常困难的。

表 3-2. 模糊测试难以处理的程序

```
1   int main(void)
2   {
3           int x;
4           read(0,&x,sizeof(x));
5
6           if(x==0x0123ABCD)
7                   vulnerable();
8   }
```

表 3-2 中的应用程序从用户输入读取一个值，并将其与特定值进行比较。如果提供正确的值，应用程序将崩溃。由于模糊测试的性质，模糊器满足该断言是几乎不可能的。选择随机值作为输入的模糊器，其发现错误的可能性是极小的 $1/2^{32}$。对于基于编译时注入的模糊器，该二进制的控制流布局将被发现为单一的路径。基于 QEMU 的模糊器在现有路径上应用随机突变，这在本质上与编译时注入的情况相同，几乎没有可能成功。

符号执行在解决此问题时好于模糊器。由于通过检查需要精确的输入，保护了对弱点函数的调用，因此模糊测试在合理的时间内不能探索那段代码。然而，符号执行引擎将能够轻易地满足该检查并触发缺陷函数。对于这个例子，符号执

行只需要探索一小部分路径来找到一个在这个例子中达到缺陷的路径，但是对于更大的二进制和现实的例子，会有太多的路径以相同的方式探索。

3.4 模糊器模块设计实现

总结模糊器的以上限制，并深入分析模糊测试技术的根本原理以及程序框架之后，本文提出基于 AFL 的改进模糊器设计，改进的模糊器主要有以下特点：

（1）能够自动生成自定义的输入数据，并对程序的异常情况进行监控

（2）无论在有源码还是无源码的情况，系统都可以对软件进行有效的漏洞挖掘，尽量多的发现使系统发生崩溃的缺陷。

（3）能够对程序的崩溃进行及时定位和记录

3.4.1 模糊器模块架构设计

模糊器设计的根本目的为一下两点：

（1）能够在有限时间内高效挖掘使二进制程序发生崩溃的软件缺陷；

（2）方便对设计模型进行增加功能和后续扩展。

为了实现以上两点根本目的，在设计时首先要做设计好模糊器系统的构架，使各功能模块之间相互独立互不影响，模块可以单独运行互不依赖，同时又可方便地进行增加功能和后续扩展。所以，在漏洞发现系统构架设计时，需要考虑以后漏洞发现系统的的伸缩性和扩展性。优化的模糊器的总体结构如图 3-2 所示。

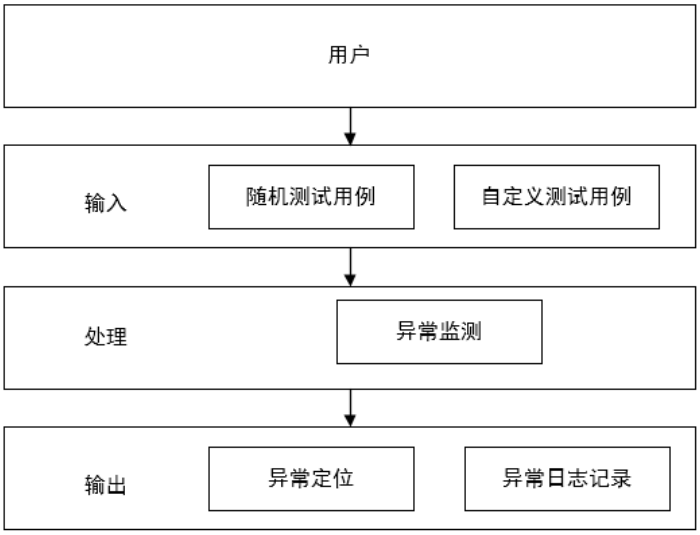


图 3-2 模糊器总体设计

3.4.2 模糊器模块组成

根据模糊器的优化目标，即模块之间的独立性和扩展性，优化的模糊器主要包含五个模块：数据生成，数据输入执行，漏洞产生监控，误报自动分析，挖掘结果记录。如图 3-3 所示。

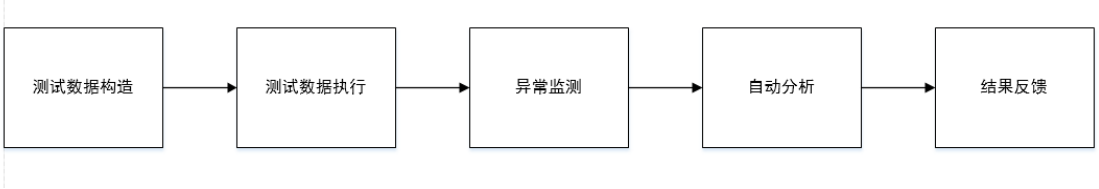


图 3-3 模糊器的模块组成

模糊器的各个模块描述如下：

（1）数据生成模块。功能为产生随机数据，以便用于探索系统，可以根据配置的协议格式生产指定格式的数据。

（2）数据输入执行模块。读取测试数据构造模块生成的数据，输入到目标二进制程序中，使用不同输入不断重复执行二进制程序。

（3）漏洞产生监控模块。监控二进制程序执行不同输入数据后，是否发现指定的 signal 产生异常情况。

（4）误报自动分析模块。定位漏洞产生监控模块发现的漏洞，并对发现的缺陷进行分析是否为可利用的有价值的缺陷。

（5）挖掘结果记录模块。将程序中可能存在软件漏洞的执行序列记录生成日志文件并输出，自动去除重复的记录。

3.4.3 模糊器模块执行流程

系统的改进主要涉及将模糊器与符号执行引擎集成, 没有改变 AFL 原有的逻辑。为了加快模糊器执行速度，系统采用 AFL 的 master-slave 并行执行模式，进程数量在配置文件中配置。系统中模糊器包装后的执行引擎初始化时加载参数，检查三个系统文件以保证 AFL 正常执行，创建工作目录中的输入目录以及输出目录。其初始化的主要流程如图 3-4 所示：

当模糊器引擎执行 *start()*方法后，系统执行 afl_count 参数个 AFL 命令行，其中第 0 个进程为 fuzzer-master，其余为 fuzzer-n。当探索非 afl-gcc 编译生成的二进制文件时，添加-Q 参数，表示 QEMU 模式执行。

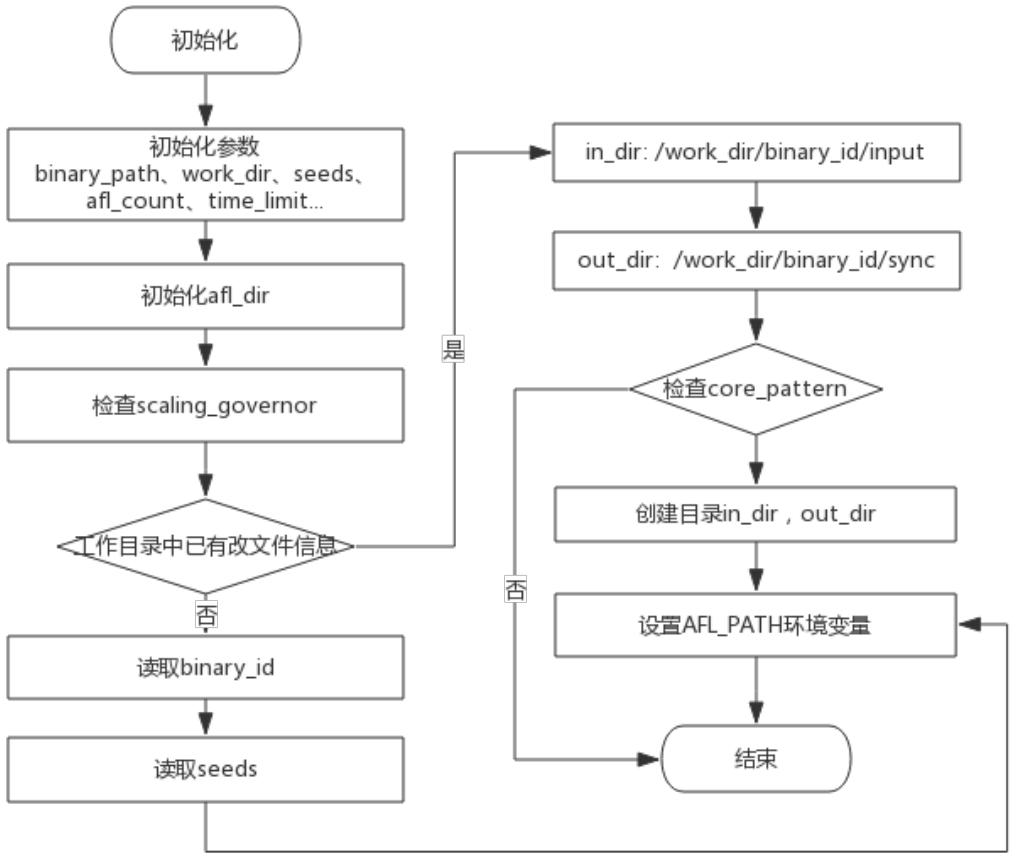


图 3-4 模糊器引擎初始化流程

当检查是否崩溃时，任务管理模块调用模糊引擎 *found_crash()*方法，它返回找到的崩溃输入文件数量，其过程为：查找输出目录下 crashs 目录，检查崩溃输入文件名并提取 AFL 产生崩溃的 signal，默认认定为崩溃的 signal 包括 *SIGSEGV*, *SIGILL*，可由配置文件配置。

3.5 模糊器跳转到符号执行

漏洞发现系统旨在弥补模糊测试的根本弱点，通过符号执行的力量来确定通过复杂检查所需的特定输入。当模糊组件经过预定值（与输入长度成正比）而没有识别新的状态转换时，我们认为它“卡住”。然后，系统检索模糊器在当前分区中认为“感兴趣”的输入，并在它们上调用符号执行引擎。

其中如果输入满足以下两个条件之一，则模糊器将其识别为感兴趣：

1）使应用程序采取的路径，触发了状态转换。

2）使应用程序采取的路径，放置到了唯一的“循环桶”。

漏洞发现系统中，AFL 执行的进度可以在 fuzzer_stats 文件中找到的“pending_favs”属性确定，其表示等待执行模糊测试的感兴趣路径。当该属性为 0 时，调用符号执行。

3.6 本章小结

本章首先介绍了模糊测试涉及的关键技术以及系统的对其的具体实现：包括协议以及文本数据的生产、异常输入数据的构造、误报自动分析等。之后列举了本系统采用的 AFL 模糊器的具体特点：包括遗传模糊测试、状态转换跟踪、循环分桶等。3.3 小节通过一个具体的实例来分析了模糊器的局限性，即因为模糊器随机对输入进行变异，所以它们能够快速发现处理一般输入而产生的不同路径。然而，生成通过程序中复杂检查的特定输入对于模糊器是非常困难的。3.4 小节展示了本系统模糊器模块的具体设计实现，包括架构设计、具体组成、以及初始化流程的描述。本章最后描述了由模糊器跳转到符号执行的具体条件。

第 4 章 混合符号执行

4.1 符号执行引擎

系统采用开源的符号执行引擎 angr 。该引擎基于 Mayhem 和 S2E 改进的模型。 首先引擎将二进制代码转换为 Valgrind 的 VEX 中间表示，其被解释成符号状态来决定对程序代码的影响。这种符号状态使用符号变量来表示来自用户的输入或其他数据，例如环境变量、文件等。

“符号变量”是可以产生许多可能具体值的变量（例如数字 5 可能表示为 X）。而其他的例如程序中的硬编码常量，被建模为“具体值”。随着执行的进行，符号约束被添加到这些符号变量上。约束是对符号值的限制语句（例如，X <100）。

分析引擎在整个执行过程中跟踪内存和寄存器中的所有具体值和符号值。在引擎到达的程序中的任何点，可以执行约束确定可能输入。这样的输入当被传递到应用的正常执行时，将驱动程序运行到该点。符号执行的优点是它可以探索和找到约束求解器可以满足的任何路径的输入。这使得它识别复杂比较时十分有效（甚至包括某些哈希函数）。

漏洞发现系统的符号化内存模型可以存储具体值和符号值。它使用基于索引的内存模型，其中读取地址可以是符号化的，但写入地址总是具体化的。这种由 Mayhem 提出的方法是一种重要的优化：如果读取和写入地址都是符号的，使用相同符号索引的重复读取和写入将导致符号约束的次方级增加。因此，符号写地址总是具体化为单个有效解。

符号存储器的优化增加了符号执行引擎的可伸缩性，但是可能导致不完整的状态空间，导致产生较少可能的解决方案。然而对实际的二进制文件进行分析时，这是一个必然要做出的平衡。

4.2 符号执行关键技术及实现

4.2.1 代码插桩

动态符号执行的目的是对目标二进制程序的执行状态进行监测。获取其运行时的状态信息和语义，并对其动态语义今夕分析。因此，能否准确高效的获取程序执行时的动态状态，将严重关系到程序分析的可靠性。程序代码插桩技术是实

现获取程序执行状态的重要技术手段，目前主要的代码插桩技术主要为以下三种：

（1） **动态二进制插桩** 当不能获取到程序源代码时，我们可以使用动态插桩工具对二进制程序做代码插桩，现在比较流行的几种二进制动态插桩工具有Valgrind[11],Pin[12]等。以这种方式工作的系统，如 BitScope[13]等，可以在执行二进制程序时，监控执行代码序列，根据不同指令，调用不同的符号执行引擎函数并对其进行分析。

（2） **二进制离线插桩** 对于很多大型程序来说，指令数可能达到上千万之多，之多执行序列的捕获以及状态分析工作提出了极大的挑战，大大超过了普通求解器的求解能力。针对大型程序，微软推出的 SAGE 工具提出了一种新的二进制离线插桩的分析方法。其工作原理的大致流程如下：a)获取程序的流程序列；b)对其镜像进行插桩并重放；c)根据离线插桩之后程序运行的轨迹获取需要的约束。

（3） **基于中间代码** 在可以获取源代码的二进制情况下，漏洞检测通常可以使用工具先将源代码编译成一种特定的中间语言代码例如 CIL[9],LLVM[10]等，然后对生成的中间语言插入符号执行引擎函数的调用语句，最后再将中间语言插桩后得到的代码重新编译成二进制程序，在实际执行的过程中进行动态符号执行。

本系统采用中间代码插桩技术，首先对二进制文件的动态执行过程监测，获取动态执行的实时状态，将单条的机器指令扩展，得到多条特定中间语言指令构成的序列，这样的操作可以使单条机器语言操作可以更加直观简洁的表示程序动态执行中对物理机器内容以及寄存器的操作，更加方便的对代码进行分析。之后再在翻译之后生成的中间代码上进行代码插桩，完成之后再将中间语言代码编译生产可执行程序实际执行。例如本节以下的单条机器指令，对中间代码插桩的流程进行分析。

在一个实际程序中，假设存在以下指令，使用中间语言表示“变量 p1 值写入第 0 号寄存器”如下所示：

```
reg[0] = p1
```

中间语言表示中间代码插桩之后的状态如下所示：

```
reg[0] = p1
regfunc{0x00510121}(p1, 0x0, 0x8)
```

其中，regfunc 为中间代码插桩的辅助函数，对写寄存器操作进行处理；0x00510121 为调用的辅助符号引擎函数的内存地址；后面为 regfunc 函数的 3 个参数：p1 表示变量名称；0x0 表示操作第 0 号寄存器；0x8 表示操作记录节点的长度为 8 字节长度。 进行如上辅助代码插桩之后，编译中间代码生成可执行程序代码并实际执行，表示的代码如下所示：

```
mov eax,p1
push  8
push  p1
push  0
call  0x00510121
```

4.2.2 符号树生成

本系统采用二叉树数据结构作为符号执行流程的存储结构，叶子节点中存储符号值或者为数值，各层非叶子节点存储内容为符号运算的操作。当得到输入数据 n 个字节之后，本系统将输入数据符号化表示为 t_0、t_1、…、$t_{(N-1)}$，然后将其符号化之后的内容写入内存之中，设置相应的 bitmap 位置为 1，当对符号化的内容进行执行操作时，采用二叉树记录执行序列。图 4-1 展示了符号二叉树的生成过程以及结果。

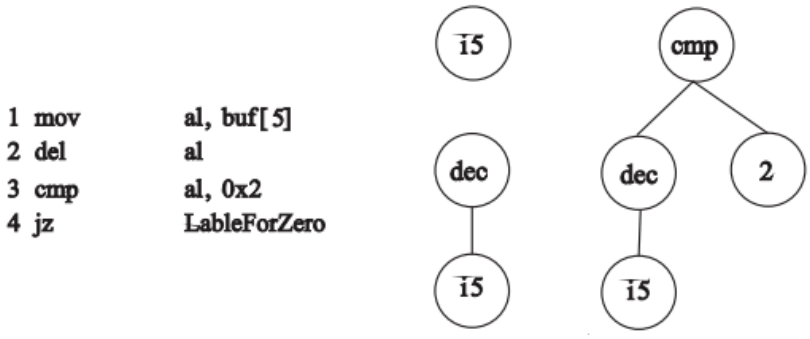


图 4-1 符号二叉树生成过程图

指令 1 作用时读取内容缓冲区第 5 个字节的内容，系统创建二叉树符号值节点 i5，大小为 1 字节，并且将符号值写入 al 寄存器，设置 bitmap 相应的位置为 1。指令 2 将 al 寄存器存储的内容减 1，al 寄存器中存储的内容为符号之，所以系统创建操作类型为 dec 的节点，保存本次操作。由于操作目的存储位置还是 al 寄存器，所以讲 dec 操作之后的符号之更新到 al 寄存器中存储起来。指令 3 将 al 寄存器中存储的符号值与 0x2 相比较，系统创建双值操作类型节点 cmp，保存本次操作，由于 cmp 操作中包含数值类型 0x2，所以还需要创建数值类型节点。由

于 cmp 比较操作结果没有目的存储位置，所以将 cmp 操作产生的结果保存到临时寄存器 EFLAGES 中。之后的指令动态监测模块会监测 EFLAGES 中存储的值。

4.2.3 约束生成

在系统执行序列遇到条件跳转时，根据临时寄存器 EFLAGS 中存储的临时变量值生成特定的约束条件。如果当前指令是条件跳转指令，系统根据 EFLAGES 寄存器中存储值得变化确定最终约束。例如 EFLAGS 寄存器 ZF 为为 1，表示条件跳转结果为 0，然后生成特定约束条件时对约束条件取反。相反如果 ZF 位为 0，则生成相反的约束条件。如果遇到其他条件的指令，根据以上描述，同样根据零食寄存器 EFLAGS 中符号位的值生成对于约束条件，并取反生成相反的约束条件。

4.2.4 路径空间探索

软件安全漏洞检测过程会遍历二进制程序执行二叉树中的所有路径，但是因为符号执行存在路径爆炸的特有问题，完全遍历整个执行路径二叉树往往在现实运行中式不可能的。因此设计好路径空间的探索算法，去覆盖尽可能多的程序隔区。覆盖更多的代码空间，才能最大可能的运行到触发程序崩溃的目标路径，这对符号执行是否能够成功尤其重要。研究者提出了很多种启发式算法，例如基本块权重打分、深度优先、广度优先、按代探索等等方法，这其中按代探索算法一次探索可以生成多组新的约束，在现实实现的过程中便于实现并行化加速，可以提升漏洞发现系统的整体效率，因此本系统选用按代探索的路径空间探索方法。

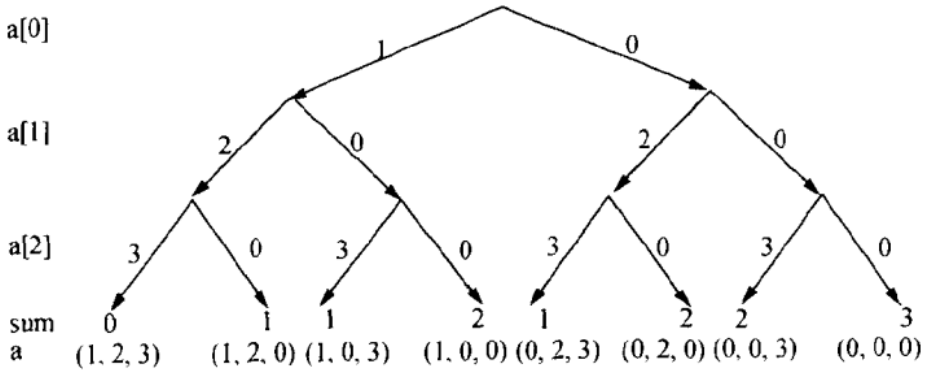
假设待测程序为 P，初始输入为 I_0，首先以 I_0 执行 P，在混合符号执行过程中，收集 I_0 所经过的每一个分支约束，最终可得到一组路径约束。假设输入 I_0 对 应着 n 个分支语句，其约束条件分别为 pc_0、pc_1、…、pc_{n-1}，则 PC = pc_0+pc_1+…+pc_{n-1}，即为 I_0 所对应的路径约束。对 pc_i (0<=i<=k—1)取反，则得到一组新的路径约束，若其可求解，则它对应于一条新的执行路径。

深度优先算法按照树的深度优先搜索顺序遍历执行树，对每条路径的最后一个约束取反，其余约束条件不变，以此得到一组新的约束。这种方法的缺点是不

利于实现搜索的并行化，代搜索比较利于并行化程序执行，以图 4-2 所示代码片段为例：

```
1 int function(int a[3])
2 {
3     int sum = 0;
4     if(a[0] == 0) sum++;
5     if(a[1] == 0) sum++;
6     if(a[2] == 0) sum++;
7     if(sum  == 3) error;
8 }
```

a)示例代码



b)执行树

图 4-2 示例代码及其执行树

代搜索算法根据 Pc 中的每一个分支约束生成新的测试用例，过程如下：

（1） 合取分支约束 j 之前的所有约束，即 Pc[0…(j 一 1)]；

（2） 对分支约束 j 取反；

（3） 合取(1)和(2)的结果。

假设初始输入为 a[0]=1、a[1]=2、a[2]=3．图 4-2 b 中最左侧的一条执行路径代表了程序的首次执行。以此路径作为父测试任务，代搜索算法会生成 3 组新约束，分别对应于图 4-2 b 中叶子节点值为 1 的 3 条执行路径。这 3 组约束都是通过对最左侧执行路径的某个分支约束取反所得到。将其作为第 1 代搜索结果，可以通过代搜索算法生成第 2 代新路径约束。第 2 代路径约束对映的叶节点值为 2，一共有 3 组。继续使用代搜索算法，生成第 3 代路径约束，对映的叶节点值为 3，一共有 1 个。至此，所有的执行路径已经搜索完毕。

显然，代搜索算法每次可以得到多组不相容的约束，分别对应于多条可能的执行路径。对各组约束的求解可以完全可以并行进行，最终能够缓解系统的性能瓶颈，提升系统的整体性能。

本系统为了减少符号执行的执行时间，提升系统的整体性能，对目标二进制程序以基本块为基本单位进行中间代码插桩，并且先前已经执行过的代码本系统将不会进行插桩。本文中所指的基本代码块表示一个代码序列，其特点是只有一个入口和出口。中间代码插桩函数根据基本代码块的起始地址和基本代码的内存大小，对相应代码 bitmap 进行动态设置。代码段中通常会包含无关的数据，但是这些数据仅为标记并不会实际执行，设置 bitmap 时应设置为 0。这将会导致符号执行的代码覆盖率大大降低，但是这并不会对路径空间选择算法的结果造成影响。采用该算法经过多次符号执行之后，代码覆盖率将达到最大限度。

4.2.5 约束条件优化

路径空间探索算法解释了以当前程序执行状态的约束为起始状态，得到新的约束条件的执行流程，但是这些刚刚得到的约束条件并不能立即放到求解器中进行约束求解，本小节描述了在将约束条件输入到求解器并生成新数据之前，对约束条件进行的优化，优化内容主要包括以下两方面：

(1) 约束条件插桩缓存

本系统采用的基于代的路径空间探索，约束条件导致生成的两种不同约束路径在取反之前都是一样的，不同点仅为是否取反，为了方式在符号执行的流程中对相同约束条件进行重复的采集，本系统采用了一种缓存的方式，中间代码中插入的辅助插桩程序对之前已经缓存过的约束不再重复生成。这加速了二进制程序隔间的探索，大大加速了漏洞挖掘系统的运行速度。

(2) 约束条件简化

本系统采用的基于代的路径空间探索，约束条件导致生成的两种不同约束路径在取反之前都是一样的，不同点仅为是否取反。在如上过程执行当中，只有部分关键输入发生了替换，导致二进制可执行程序运行时导向不同隔间。所以在路径约束进行求解时，只考虑影响当前约束条件、会导致路径变异的关键输入即可。

对以上输入进行简化约束条件的时候，关注输入处理主要为以下两个方面：

a) 剪枝无关约束：假设所有变量的关键属性不为规定值，即相关的污点属性都不包含有关键输入，则可以去除当前约束条件。

b) 具化变量：在剩下的约束条件中，对关键属性不包含关键输入的无关变量，使用二进制程序实际执行时得到的具体数值进行变换，以此来简化生成的路径约束条件。

优化之后生成的路径约束条件将输入到求解器进行约束求解，假如得不到结果我们即可认为该路径对应的隔间不能到达；假设该路径约束条件可以得到可行解，则将求解器得到的输入数据加入到目标队列中，模糊器将从队列中得到输入执行，或者符号执行也会从队列中进行选择，作为下一个输入数据继续执行。

4.3 符号执行的局限性

传统的符号执行方法，从程序起始就开始执行符号执行，探索路径状态，以找到尽可能多的缺陷。然而这种方法有两个主要的限制。

（1）**符号执行速度缓慢**　　这是由于需要解释应用程序代码（与使用模糊器原生执行代码相反）以及约束求解步骤中涉及的开销。特别是后一操作，涉及NP 完全问题的解决，使得潜在输入的生成以及确定哪些条件跳转是可行的十分耗时。

（2）**符号执行的路径爆炸问题**　　随着符号执行引擎探索该程序，路径数量呈指数增长，并且它很快变得不可能再探索下去。例如表 4-1 中的示例。在此程序中，当用户输入 25 个 B 字符时，将触发 *vulnerable()*，但这是一个在符号执行框架中难以表达的条件。这个程序的符号执行将导致巨大的状态爆炸，因为模拟 CPU 将递归调用进入到 *check()*函数中。每次执行与字母 B 比较的三元操作都将模拟状态一分为二，最终导致 2^{100} 种可能的状态，这是现实中不可能实现的处理量。

模糊器选择输入是基于状态转换的，不去推测程序的整个状态空间，只是仅仅考虑由输入触发的状态转换。也就是说，它将主要关注次数，例如，第 5 行的 check 函数执行成功。也就是说，不管输入中 B 字符在哪里，状态的判断都将基于输入中的它们的数目来确定，因此模糊器不会出现路径爆炸问题。

本系统的符号化内存模型可以存储具体值和符号值。它使用基于索引的内存模型，其中读取地址可以是符号化的，但写入地址总是具体化的。这种由 Mayhem 提出的方法是一种重要的优化：如果读取和写入地址都是符号的，使用相同符号索引的重复读取和写入将导致符号约束的次方级增加。因此，符号写地址总是具体化为单个有效解。

表 4-1. 导致在符号执行产生路径爆炸的程序

```
1   int check(char* x, int depth){
2     if(depth >= 100){
3           return 0;
4     }else{
5           int count = (*x == 'B') ? 1 : 0;
6           count += check(x+1, depth+1);
7           return count;
8     }
9   }
10
11  int main(void){
12    char x[100];
13    read(0, x, 100);
14
15    if(check(x,0) == 25)
16          vulnerable();
17  }
```

符号存储器的优化增加了符号执行引擎的可伸缩性，但是可能导致不完整的状态空间，导致产生较少可能的解决方案。然而对实际的二进制文件进行分析时，这是一个必然要做出的平衡。

4.4 漏洞发现系统符号执行设计及优化

在大多数情况下，模糊测试单独就可以充分地探索大部分路径，只要简单地通过随机的位翻转和其他突变策略就可以找到它们。由于模糊器基于原生代码执行，在大多数情况下，它的性能优于符号执行。因此，大多数探索工作交给了模糊器，这将快速地找到许多路径，符号执行引擎只是去解决更困难的约束。符号执行模块调用查找新路径方法调用引擎，若找到可以通过检查的输入，它将返回二进制文件可执行的下一步路径集合，否则返回空集合。其主要执行流程如下：

（1） 检查 redis 中该二进制的已跟踪集合中是否已经包含当前输入；

（2） 将当前输入写入该二进制文件 redis 已跟踪集合中；

（3） 初始化符号执行跟踪引擎；

（4） 设置输入具化参数，包括内存阈值、寄存器阈值等等；

（5） 更新已经过的路径；

（6） 当下一分支的 active 数量大于 0 且基本块数量小于路径数，执行以下操作：

a) 检查 redis 中该二进制的已完成集合中是否包含该输入；

b) 对于每条丢失路径，如果当前地址与该路径最后跟踪地址未相遇，

c) 移除该路径的预约束；

d) 当前转移与缓存探索器发现的转移都写回到 redis。

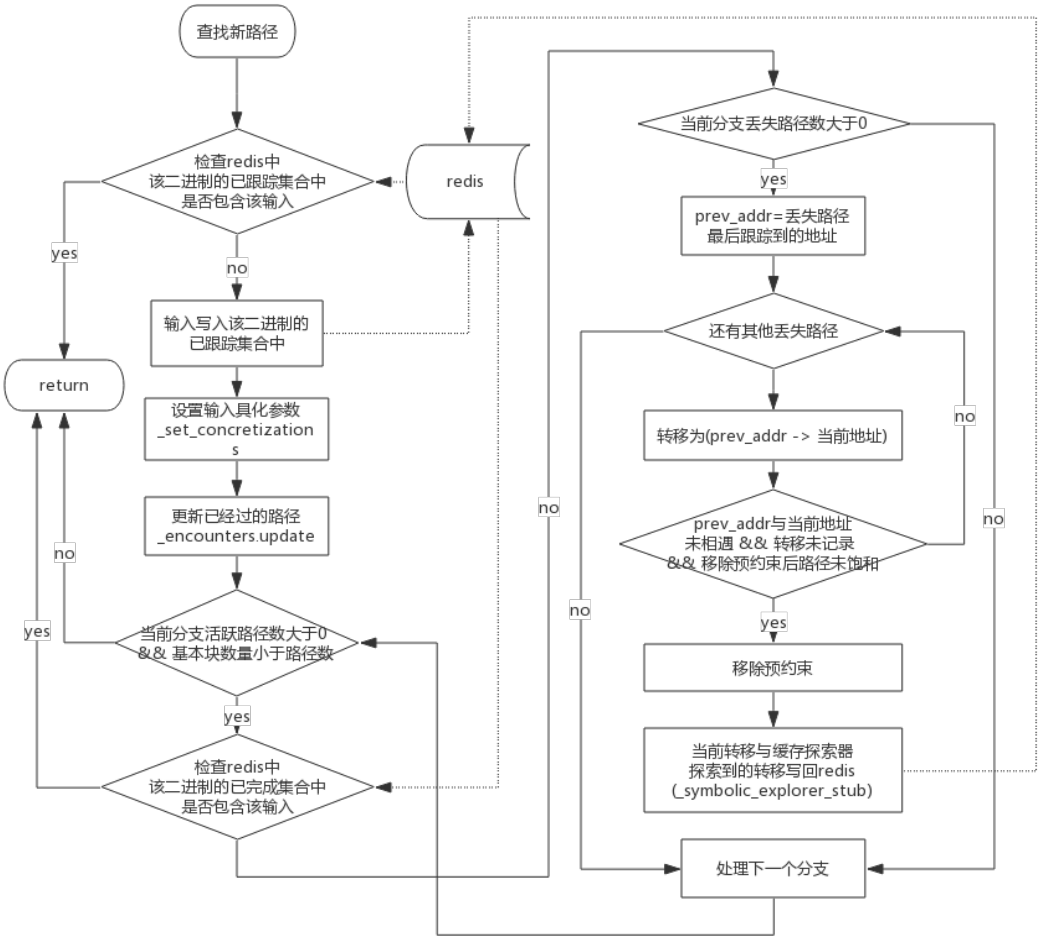


图 4-3 系统中符号执行查找新路径流程图

4.4.1 符号执行前的预约束控制

系统使用预约束控制以确保符号执行引擎的结果与模糊器结果相同，同时保持发现新的状态转换的能力。在预约束执行中，输入的每个字节被约束与模糊器输出的每个实际字节匹配，例如/dev/stdin[0] == 'A'。当发现新的可能的基本块转换时，暂时地去除预约束，允许系统求解偏离该状态转换的输入。预约束对于在符号执行引擎中生成相同的轨迹是必要的。

使用表 4-2 中的示例来演示系统中的输入预约束是如何工作的，为了达到弱点函数，系统必须在第 18 行提供一个魔数（0x42d614f8）。在输入模糊测试后，系统最终认识到它没有发现任何新的状态转换，因为单独的模糊器不能猜测正确的值。

表 4-2 需要预先约束符号输入的程序

```
1   int check(char* x,int depth){
2       if(depth >= 100){
3           return 0;
4       }else{
5           int count = (*x == 'B') ? 1 : 0;
6           count += check(x+1, depth+1);
7           return count;
8       }
9   }
10
11  int main(void){
12      char x[100];
13      int magic;
14      read(0, x, 100);
15      read(0, &magic, 4);
16
17      if(check(x,0)==25)
18          if(magic==0x42d614f8)
19              vulnerable();
20  }
```

当调用符号执行跟踪输入时，系统首先约束符号输入中的所有字节以匹配跟踪到的输入字节。由于程序是以符号方式执行的，因此每个分支只有一种可能性，因此只跟随一个路径，这防止了路径爆炸。然而，当执行到达行 18 时，系统识别出在模糊测试期间从未采取的备选状态转换。然后，系统删除在执行开始时添加的预约束。字符数组 x 中的字节被路径部分约束，magic 的值只受等式检查 if（magic == 0x42d614f8）约束。因此，符号执行引擎创建一个包含 25 个 B 和一个魔数 0x42d614f8 的输入。 这通过了 18 行中的检查并成功到达缺陷函数 vulnerable()。

4.4.2 引入缓存探索器

为了减少昂贵的符号引擎调用的次数，系统还引入了一个缓存探索器，以发现更多的状态转换，直接位于新发现的状态转换之后。这个缓存探索器探索状态转换的周围区域，直到探索器遍历配置数量（系统默认配置为 1024）的基本块。探索达到了预定数量后，符号执行就会为探索器发现的所有路径确定输入。

往往一个状态转换后很快就有另一个状态转换，这将导致模糊器立即卡住，并转到符号执行，这将导致系统性能大大降低。引入了缓存探索器可以防止模糊器在接受符号执行生成的输入后被卡住。缓存探索器在确定当前转移后执行，接收参数为当前发现的路径，并将当前路径周围区域的转移同时写回 redis。缓存探索器的执行流程如图 4-4 所示：

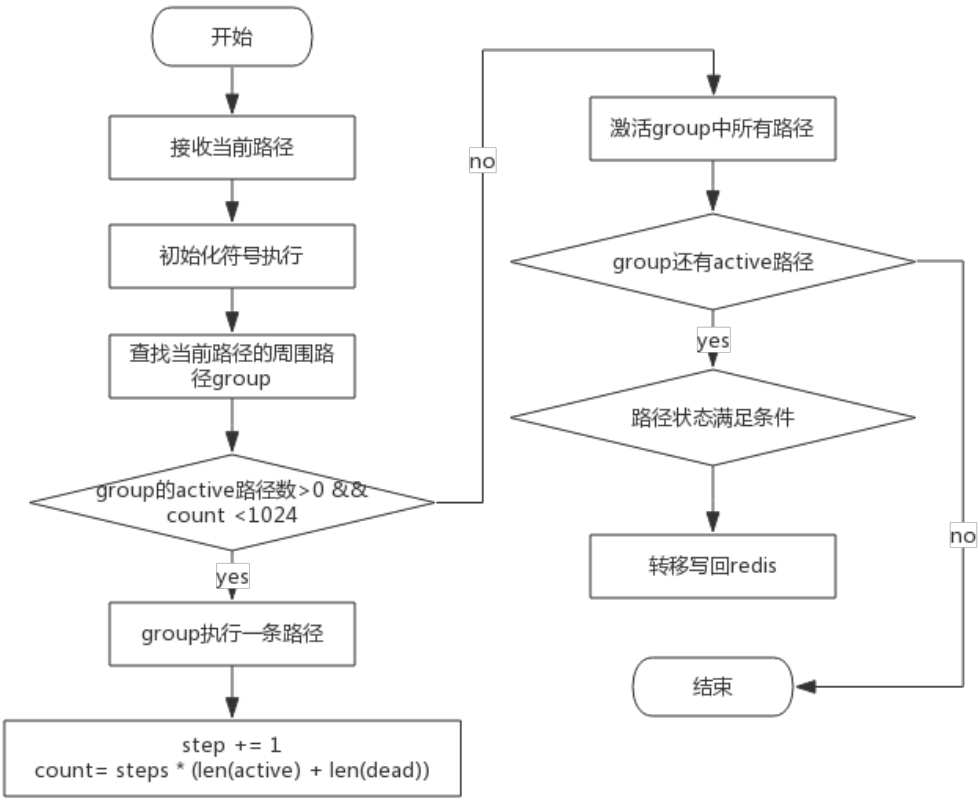


图 4-4 符号执行缓存探索器的执行流程

4.4.3 符号执行中的地址转换

其中符号执行继续模糊器探索到的地址进行探索，依靠模糊器传递来的输入来实现。每次继续探索之前会判断当前地址与丢失分支的最后一个跟踪地址（符号执行前一个跟踪到的地址）是否相遇，依靠 bitmap 实现。alf-fuzz 将会每分钟将当前内存执行状态写入到 *fuzz_bitmap* 文件中。

地址转换到 bitmap 中位置下标的过程如下：

（1） loc = addr

（2） loc = (loc >> 4) ^ (loc << 8)

（3） loc = loc & (bitmap_size -1)

（4） loc = loc >> 1

判断当前地址与前一地址是否相遇的公式如式(4-1)：

$$hit = bool(ord(bitmap[cur \wedge prev]) \wedge 0xff) \tag{4-1}$$

4.5 本章小结

本章首先介绍了本系统采用符号执行引擎以及相关概念，并介绍了符号执行技术中相关知识。之后描述了符号执行中的一些关键技术以及系统中对其实现：包括代码插桩、符号树生成、约束生成、路径空间探索、符号条件优化等。4.3 小节讨论了符号执行的局限性：即执行速度慢以及存在路径爆炸问题。本章最后描述了本系统中对符号执行模块的设计实现以及相关优化：包括符号执行前的预约束控制、引入缓存探索器、bitmap 地址空间转换等。

第 5 章 漏洞发现系统任务管理

5.1 系统配置管理

系统可配置项如表 5-1 所示：

表 5-1 系统可配置项

分类	配置项	描述	示例
Redis 配置	REDIS_HOST	Redis 地址	"173.26.100.209"
	REDIS_PORT	Redis 端口	6379
	REDIS_DB	Redis 数据库	0
Celery 配置	BROKER_URL	任务队列（amqp）	"amqp://guest@127.0.0.1:5672//"
	CELERY_ROUTES	任务分发映射	{'driller.tasks.conc:‘concolic’ 'driller.tasks.fuzz': 'fuzzer'}
目录配置	QEMU_DIR	Qemu 程序目录	"/usr/bin/qemu"
	BINARY_DIR	二进制扫描目录	"/root/defect-mining/binarys"
	FUZZER_WORK_DIR	模糊器工作目录	"/root/defect-mining/output"
超时配置	CONCOLIC_TIMEOUT	符号执行超时时间（秒）	600
	FUZZ_TIMEOUT	模糊器超时时间（秒）	600
	CRASH_CHECK	崩溃输入检查间隔（秒）	10
	FUZZER_INSTANCES	AFL 并行进程数	6
	MEM_LIMIT	符号执行内存限制（M）	1024 * 8

5.2 模糊器到符号执行过程

系统采用 celery 分布式任务队列进行任务分发与管理，其中分发队列采用 rabittmq 消息队列，任务完成或中断时采用 redis 存储任务状态，恢复后可继续执行。系统包含两个任务队列：

（1）**模糊器（fuzz）** 接收二进制文件路径作为参数，输入为默认初始化或从文件读入，并初始化模糊器，开启监听进程监听符号执行产生的新输入。循环检查模糊器，当没有发现崩溃也没有超时，检查 *fuzzer_stats* 文件中的

pending_favs 属性，为 0 时表示模糊器卡住，将任务交给符号执行处理。当发现崩溃，将崩溃信息写入 redis，并撤销仍在执行的符号执行任务。

（2） **符号执行（concolic）** 读取模糊器输出目录中的 *fuzz_bitmap* 文件，并写入 redis。对于 fuzzer 输出目录中的每一个未被跟踪过的输入文件，读取 redis 中的模糊器 bitmap，并开启符号执行引擎。

模糊器到符号执行的内部工作流程图如图 5-1 所示：

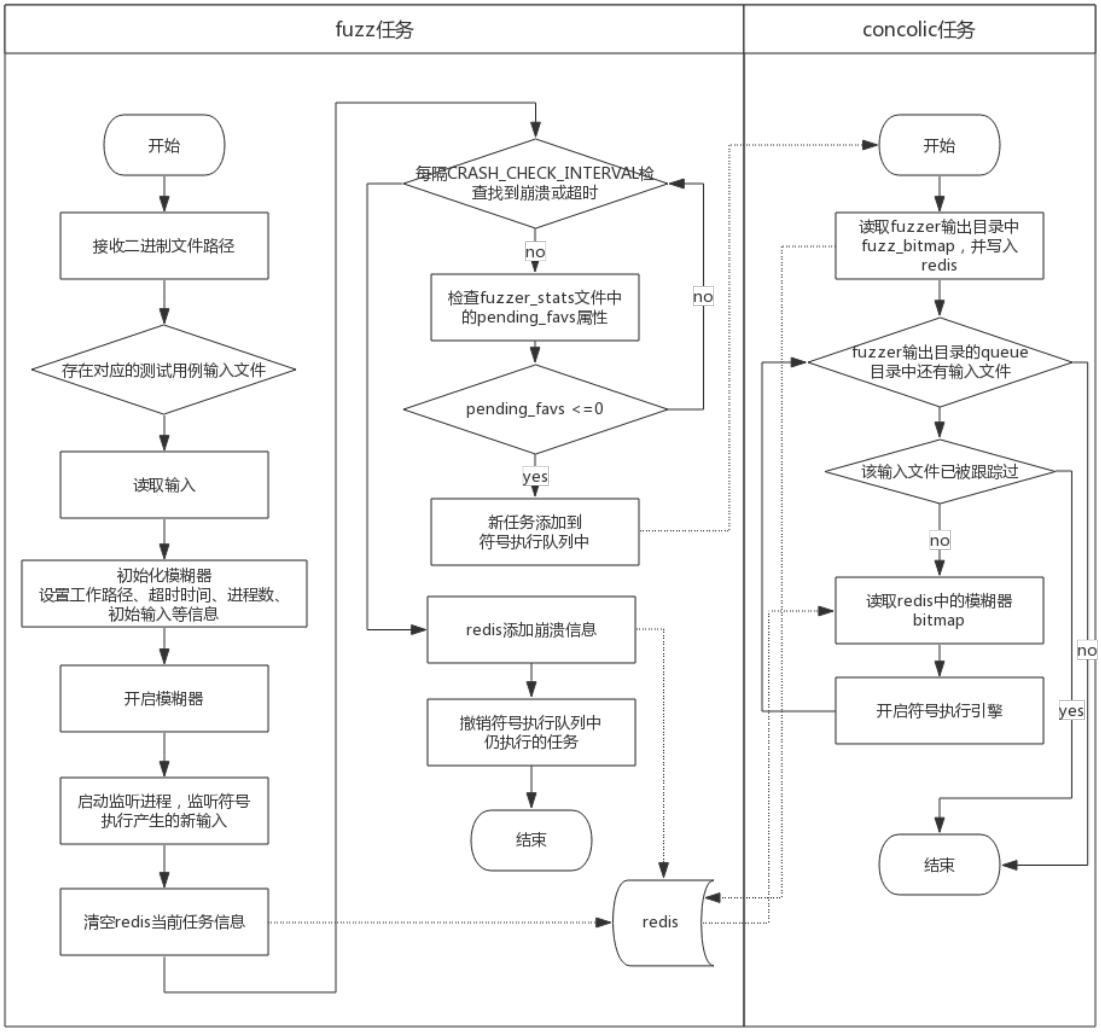


图 5-1 系统任务管理中模糊器到符号执行的内部工作流程图

5.3 任务协作中的数据流转移

监听进程接收两个参数：redis 管道、输出队列目录，其功能为监听 redis 管道，符号执行若找到新的可驱动程序继续执行的输入，会将输入 publish 到该管道中。监听到新的输入数据时，监听进程将会把输入数据写入到输入队列目录的新建文件中。之后，模糊器读取到新建的输入文件后，继续执行模糊测试，直到再次卡住。

其中模糊器任务产生的输出记录到日志文件 fuzzer-out.log，符号执行任务产生的输出记录到日志文件 concolic-out.log。监听进程、模糊器、符号执行协同工作的控制流转移时序图如图 5-2 所示：

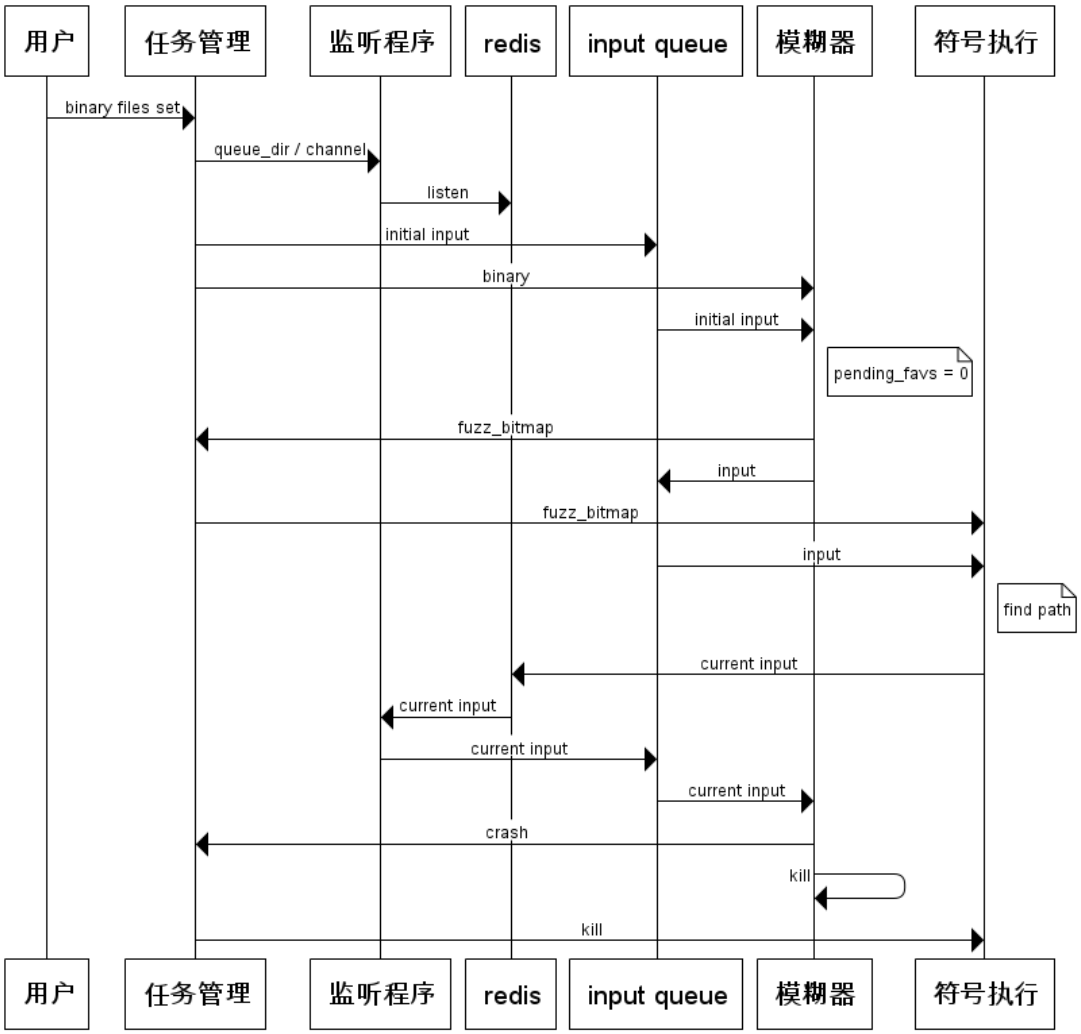


图 5-2 监听进程、模糊器、符号执行协同工作时序

5.4 本章小结

本章给出了漏洞发现系统的任务管理模块具体实现过程，首先介绍了系统的配置管理，之后详细描述了模糊器跳转到符号执行的详细过程，采用 celery 分布式任务队列进行任务分发与管理，系统包含两个任务队列：即模糊器任务队列与符号执行任务队列，两个队列之间相互对立交替工作。本章最后描述了任务协作中的数据流转移：监听进程接收两个参数：redis 管道、输出队列目录，其功能为监听 redis 管道，符号执行若找到新的可驱动程序继续执行的输入，会将输入 publish 到该管道中。监听到新的输入数据时，监听进程将会把输入数据写入到输入队列目录的新建文件中。之后，模糊器读取到新建的输入文件后，继续执行模糊测试，知道再次卡住。

第 6 章 漏洞发现系统评估实验

为了确定本文方法的有效性，对大型二进制数据集进行了评估。本文的评估目标是展示两方面：首先，系统显著扩展了无辅助模糊器实现的代码覆盖率，其次，这种增加的覆盖率增加了发现的漏洞数量。

6.1 实验数据集

本文从 DARPA 网络大挑战（CGC）[11]的资格赛对本系统进行了评估，该比赛旨在“测试新一代全自动化网络防御系统的能力”。在资格赛期间，选手有 24 小时自主发现内存损坏漏洞，并通过给出输入，使相关应用程序处理时导致崩溃来证明。在 CGC 资格赛数据集中有 131 个应用，但其中 5 个涉及多个二进制文件之间的通信。由于这样的功能超出了本系统的能力范围，本文只考虑 126 个单二进制应用程序，留下多二进制应用程序作为以后的工作。

这 126 个应用程序包含各种各样的障碍，使得二进制分析变得困难，例如复杂的协议和庞大的输入空间。它们专门用于突出程序分析技术的能力，而不仅仅是玩具应用程序用于黑客娱乐。这些二进制文件的种类和深度允许对高级漏洞挖掘系统进行广泛测试。

本文在 AMD64 处理器的计算机集群上运行我们的实验。每个二进制有四个专用的模糊器节点，当模糊器需要符号执行帮助时，它将作业发送到 64 个所有二进制文件共享的符号执行节点池中。由于可用内存的限制，我们将每个符号执行作业限制为 2 GB 的内存。在所有测试中，本文分析单一的二进制文件最多 10 小时。分析每个二进制文件直至发现崩溃或超时。

所有崩溃都使用赛事提供的二进制测试工具收集和重放，以验证提交的崩溃在实际的 CGC 环境中是可重现的。因此，这些结果是真实可验证的，并且可以与竞赛的实际结果相比。

6.2 实验过程

在本文的评估中总共运行了三个实验。 首先，为了以现有技术的基准性能来评估本系统，我们尝试使用纯符号执行引擎和纯粹的模糊器进行漏洞挖掘。然后，我们在同一数据集上评估本系统。

实验设置如下：

（1）基本模糊测试 在这个测试中，每个二进制被分配 4 个内核用于进行 AFL 模糊测试，但是符号执行节点被关闭。 当模糊器无法发现新路径时，符号执行不执行。注意，本系统对 AFL 的 QEMU 后端进行了更改以提高针对 CGC 二进制文件的性能，但是如前所述，本系统没有对 AFL 的核心逻辑进行更改。

（2）基本符号执行 本文使用一个现有的符号执行引擎，基于 Mayhem[8] 提出的想法，用于符号执行测试。为了确保对现有技术的公平测试，优化的状态合并技术被用于帮助限制状态爆炸的影响，如 Veritesting[1]中提出的。 我们通过符号探索状态空间，从入口点开始分析每个二进制文件，检查内存损坏。当发生状态爆炸时，我们使用启发式方法来对程序进行深入探索的路径进行优先级排序，以最大化代码覆盖率。

（3）漏洞发现系统 当测试本系统时，每个二进制文件被分配 4 个内核用于模糊引擎，总共 64 个内核用于符号执行组件。当系统确定模糊器被“卡住”时，由于轨迹是由模糊节点请求的，所以符号执行池处理先进先出队列中的符号执行作业。 符号执行跟踪被限制为一小时周期和 4GB 字节的内存限制，以避免当分析大的轨迹时资源耗尽。

本章将讨论对系统评估的几个不同方面。首先将讨论三个实验的结果，即系统对我们在数据集中找到漏洞数量的贡献。接下来，将讨论系统在代码覆盖率方面对现有技术的贡献。最后我们将重点介绍数据集中的一个示例应用程序，进行深入的案例研究，以讨论系统如何提高代码覆盖率并识别该应用程序中的漏洞。

6.3 漏洞发现量比较

在本小节中，本文将讨论三个实验发现的漏洞的数量，以及本系统在这方面发挥的作用。

基本符号执行实验在这个数据集上表现不佳。 在 126 个应用程序中，符号执行只发现了 16 个漏洞。在本文的实验数据集中的 126 个 CGC 应用程序中，模糊测试足以发现 68 个的崩溃。在剩余的 58 个二进制文件中，41 个被“卡住”（即 AFL 无法识别任何新的“感兴趣路径” 如第四章所讨论，不得不求助于随机输入突变），17 个尽管继续找到新的感兴趣的输入，但没有发现崩溃。

在系统的运行中，模糊器调用了“卡住”的 41 个二进制文件上的符号执行组件。图 7 显示了对这些二进制文件调用符号执行的次数。其中，系统的符号执行组件能够为其中 13 个应用程序生成总共 101 个新输入。 利用这些额外的输

入，AFL 能够恢复额外的 9 次崩溃，使系统实验中确定的总崩溃数达到 77 次，这意味着本系统发现的漏洞相对于基本模糊测试实现了 12%的改进。

当然，在本系统实验中发现崩溃的大多数应用程序是与基本模糊器一起发现的。 对于通过不同方法识别的唯一崩溃，基本模糊器发现了 55 个符号执行未能发现的崩溃。 其中的 13 个漏洞与基本符号执行共享。另外 3 个漏洞基本符号执行与系统恢复的漏洞重叠，剩下应用程序，其中基本符号执行单独发现了 1 个漏洞，剩下 6 个应用程序，本系统是唯一找到漏洞的方法。 基本上，系统有效地合并和扩展了基线模糊和基线符号执行提供的功能，获得了比单独执行更多的结果。 这些结果示于图 5 中。

总而言之，本系统能够识别 77 个独特应用程序中的崩溃，与基础实验的并集（71 个漏洞）相比，改善了 6 个崩溃（8.45%）。 这与竞赛中得分最高的团队确定的崩溃数量相同（并且明显高于任何其他对手）。在相同的时间内，如果没有本系统（即使用两种基本方法），我们无法达到这样的结果。虽然如此，但与参与团队的比较只是指示性的，并不意味着是定性的。 参与团队在严格的时间限制下运行，只有很少或没有错误的空间。 本文的实验受益于更多的准备时间，本文提出的技术还可以在系统的发展过程中改进。

这些结果表明，优化的具有选择符号执行的模糊器改善了其在发现崩溃中的性能。 通过提高漏洞挖掘中的技术水平，系统能够找到比通过模糊测试和通过符号执行分别发现的并集更多的崩溃程序。虽然 6 个唯一漏洞的贡献可能看起来远低于 CGC 限定事件中的应用程序总数，但这些崩溃代表其各自二进制文件中的深层漏洞，其中许多漏洞需要多个符号执行调用来穿透几个隔区。

表 6-1 实验结果总述

方法	发现崩溃数
模糊测试	68
模糊测试 ∩ 本系统	68
模糊测试 ∩ 符号执行	13
符号执行	16
符号执行 ∩ 本系统	16
本系统	77

维恩图 6-1 显示了基本模糊（AFL），符号执行和本系统在 CGC 数据集中查找崩溃的相对覆盖率。 标记为 F 的圆圈表示通过模糊查找发现的崩溃，S 表示通过符号执行发现的崩溃，D 表示由本系统发现的崩溃。 该表根据不同方法的

相对有效性及其相对于彼此的改进提供了这些结果。 可以看到，本系统识别了由模糊测试和符号执行发现的崩溃的超集。

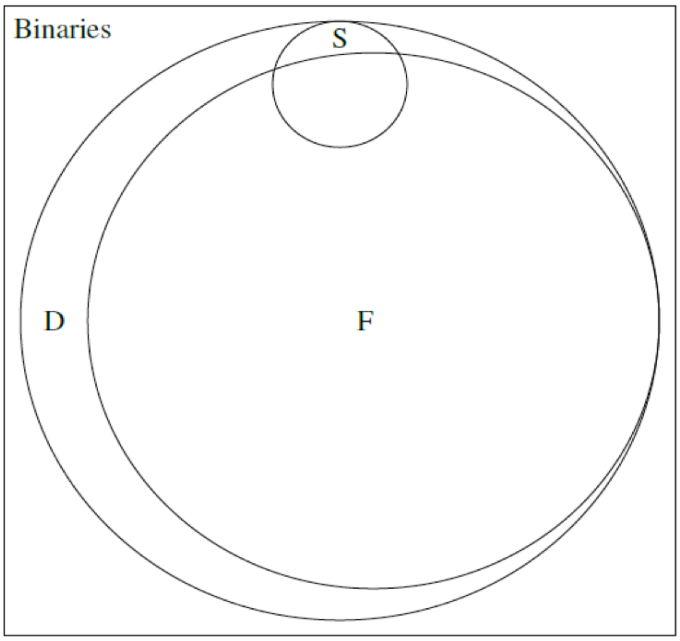


图 6-1. 实验结果的构成

6.4 状态转移覆盖分析

选择性符号执行能够克服在处理“魔数”常数和其他复杂输入检查时模糊器的根本弱点。这意味着，在模糊器不能识别新的感兴趣输入（例如，由于无法猜测散列或魔数）之后，符号执行引擎可以生成允许模糊器继续探索已被卡住的路径的输入，本系统的这个方面可以在表 I 中观察到，其示出了在执行期间如何发现状态转换。在符号执行能够找到新路径的应用中，单独的模糊测试只发现了平均 28.5％的块转换。

如所预期的，符号跟踪在这些二进制中仅占少量新的状态转换（平均约 15.1％），因为符号探索在范围上受限并且主要用于识别和通过感兴趣的检查。然而，由符号执行引擎产生的输入帮助模糊引擎成功地穿透这些状态转换。模糊引擎对这些输入的后续修改允许它平均找到额外的 56.5％的状态转换。总的来说，对于模糊器被阻塞并且最终符号执行找到新路径的应用，71.6％的状态转换由基于符号跟踪期间生成的那些的输入产生。

事实上，小数量的有价值输入将导致更大的模糊器可以探索的状态转换集合，表明本系统的符号执行引擎产生的输入刺激了更深入的应用程序的探索。

重要的是，这个数字仅适用于 41 个应用程序中的 13 个，这些应用程序变得“卡住”，并且能够通过符号执行识别新的路径。这些百分比在我们在实验过程中的基本块总量上归一化，因为生成完整的控制流图静态地需要超出本文范围的重量级静态分析。

如在第四章中所讨论的，我们将状态转换考虑为基本块（A，B）的有序对，其中块 B 在块 A 之后立即执行。换句话说，状态转换是控制流图中的边，其中每个节点表示程序中的基本块。很明显，如果我们发现每个状态转换，我们就有完整的代码覆盖。类似地，如果我们只发现很少状态转换，那么我们可能具有非常低的覆盖。因此，使用不同状态转换的数量作为代码覆盖的度量是合理的。在图 6-2 中，我们通过显示本系统的模糊器单独找不到的额外基本块数量，来展示系统随时间推移提升了多少基本块覆盖。

如图 6-2 所示。执行时间显示为根据二进制文件执行时间归一化后的结果，因为二进制文件执行时间由于它是否与何时崩溃而各不相同。此图包括调用的 13 个受益于符号执行的二进制文件。

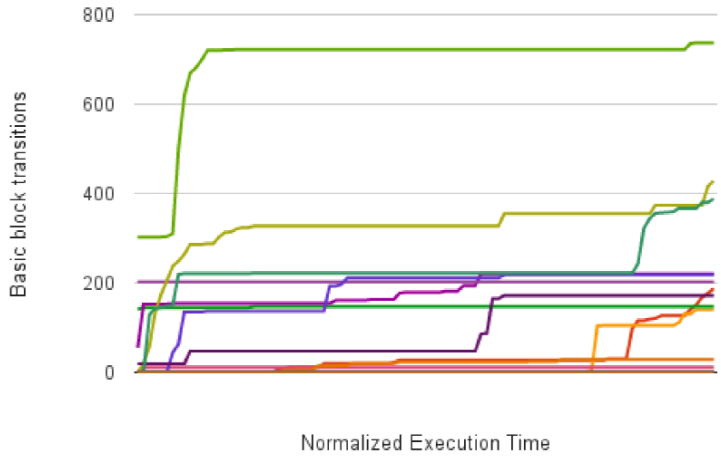


图 6-2. 系统随时间发现的模糊器无法单独找到额外基本块的数量

6.5 程序隔区覆盖分析

系统中的符号轨迹的目标是使得模糊器能够探索二进制中的各个隔区，其可以通过对用户输入进行复杂检查来分离。期望看到通过调用符号跟踪器产生的输入符合在应用程序中找到新的隔间。也就是说，由符号执行引擎产生的输入应该使得模糊器能够到达并探索新的代码区域。

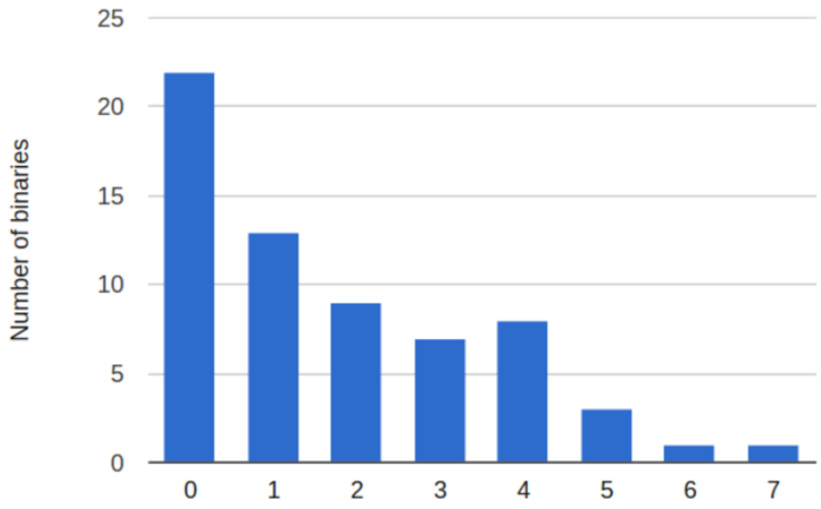


图 6-3. 模糊测试不会使其崩溃的二进制程序中调用的符号执行的次数

如图 6-1 所示，数据集中 126 个应用程序中的 68 个，没有任何需要系统符号执行的困难检查。这些对应模糊测试可以独立发现崩溃输入，或从未被卡住的应用。这些往往是具有简单协议和较少复杂检查的应用。另一方面，本系统能够至少在 13 个二进制文件中找到一个困难检查，并在 4 个二进制文件中找到多个困难检查。这些隔间由于分离它们的特定检查，所以对于基本的模糊器很难进入，但是可以通过本系统使用的混合方法解决。

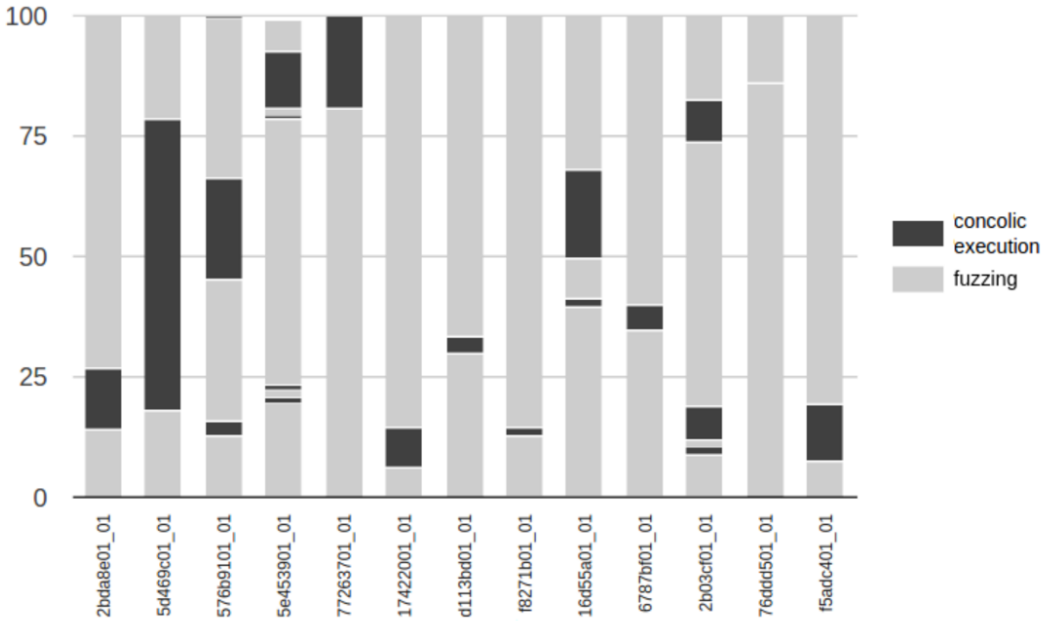


图 6-4. 每次调用符号执行导致发现更多的基本块转换

每次调用符号执行都有可能将执行引导到应用程序中的一个新隔区。这可以通过在一个模糊循环被“卡住”，并且在随后的一轮模糊实现的覆盖上调用符号执行之前，且在符号执行组件将执行推送到下一个隔区之后，分析本系统的基本块覆盖来度量。在图 6-4 中，显示了在分析的每个阶段，调用了符号执行的每个二进制文件中，总数标准化为整个实验中发现的基本块总数的基本块比例。该图表明本系统确实将应用程序中的执行驱动到了新的隔区，允许模糊器快速探索更大量的代码。

6.6 具体实例分析

对于二进制文件 2b03cf01，本系统在大约 2.25 小时内崩溃，图 6-5 显示了随时间发现的基本块的数量。 每条线代表不同数量的符号执行调用（从 0 到 3 次）。在每次调用符号执行之后，模糊器能够找到更多的基本块。

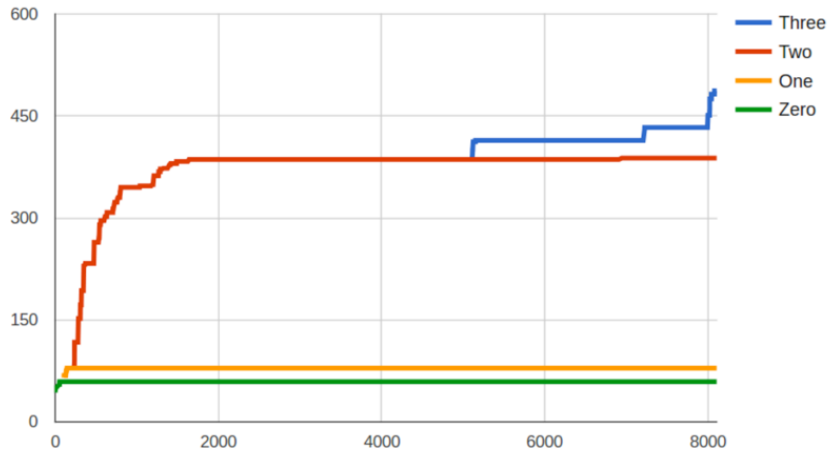


图 6-5. 二进制文件 2b03cf01 实验结果

表 6-2. 在所有调用了符号执行的二进制文件中，发现状态转移方式的百分比

状态转移类型	所有二进制文件中发现基本块的百分比	在符号执行至少发现了一个输入的二进制文件中发现基本块的百分比
初始化模糊器发现	84.2	28.4
符号执行发现	3.3	15.1
模糊器运行符号执行产生的输入	12.5	56.5
总量	100	100

本节将重点介绍单个应用程序，以深入解释系统的操作。我们将关注标识符为 2b03cf01 的二进制文件。另外，我们提供这个二进制的调用图，我们将在这个案例中参考图 10。该图显示了系统的模糊测试和符号执行组件连续调用的性能 - 通过连续的模糊测试调用发现的节点被绘制为逐渐变暗的颜色，并且由不同样式绘制的边，展示由符号执行组件重新发现的转换。节点的不同颜色表示二进制中的不同隔间。

表 6-3. 2b03cf01 应用程序中的第一个复杂检查

```
1    enum {
2     MODE_BUILD = 13980,
3           MODE_EXAMINE = 809110,
4    };
5
6    ...
7
8    RECV(mode, sizeof(uint32 t)) ;
9
10   switch(mode[0]) {
11         case MODE_BUILD:
12           ret = do_build();
13           break;
14    case MODE_EXAMINE:
15           ret = do_examine();
16           break;
17    default:
18           ret = ERR_INVALID_MODE;
19   }
```

该应用表示电源测试模块，其中客户端向服务器提供电气设计，并且服务器建立电连接的模型。这不是一个简单的二进制：它为用户提供了各种复杂的功能，并要求格式正确的输入，有许多复杂的检查。

当系统对这个二进制进行模糊测试时，第一个复杂的检查导致模糊器在应用程序的一个相当小的区域中只找到 58 个基本块后立即卡住，包括一些包含初始化代码的函数。模糊引擎卡在对用户输入的检查。为了方便，对应于图 6-7 中的节点“A”的所讨论的片段在表 6-3 中再现，尽管系统直接在二进制代码上操作。

来看源代码，我们看到从主循环调用的两个主要命令，需要用户给出一个特定的 32 位数字来选择“操作模式”。要调用函数 *do_build()*，用户必须提供数字

13980，要调用函数 *do_examine()*，用户必须提供数字 809110。虽然这些检查对于人来说看起来很简单，但实际上一个模糊器必须暴力破解他们。因此，模糊器猜测这些魔数的机率是微乎其微的，结果模糊组件被卡住。

在模糊器无法识别新的感兴趣路径之后，系统调用符号执行组件来跟踪模糊器目前收集的输入，并找到新的状态转换。系统找到将驱动执行上述两个函数的输入，并将它们返回到模糊器进行探索。再次，模糊器被相当快地阻塞，这次在图 6-7 中的节点“B”处的另一个复杂检查。系统的符号执行引擎被第二次调用，产生足够的新输入来通过这些检查。从这一点开始，模糊器能够在处理通用输入的应用程序大隔间内找到 271 个附加基本块，对于该应用，通用输入由与用户提供的电气设计分析相关的解析代码组成。最终，模糊器找到它在该隔间中可能存在的所有有趣的路径，并决定它不会产生进一步进展，导致调用另一个 Driller 的符号执行引擎。

这一次，系统找到 74 个新的基本块，并生成到达它们的输入，这依靠成功地通过模糊器以前没有产生满足的输入的检查。这些附加的基本块（由图 6-7 中的黑色节点表示）包括添加特定电路组件的功能。表 6-5 给出了对于模糊器有麻烦的，包含针对用户输入的特定检查的函数。表示这些组件的输入，必须遵守电路组件的确切规范，并且这些规范的检查是系统的符号执行引擎的第三次调用所发现的。在函数检查中使用的这些常量在表 6-4 中重现的代码中定义。

模糊器不消耗巨大的搜索空间是不能猜测到这些常量的，因为它们是 32 位整数的特定值。然而，系统的符号执行可以很容易地找到这些常量，因为代码中的比较动作，在采用这些分支的路径上产生易于解决的条件。

如表 6-4 所示，为了猜测这些常数，这些特定值必须从 232 个数字的搜索空间中猜出。

表 6-4.具有显式常量的枚举定义

```
1    typedef enum {
2            FIFTEEN_AMP = 0x0000000f,
3            TWENTY_AMP = 0x00000014,
4    } CIRCUIT_MODELS_T;
```

如图 6-6 所示为二进制执行刘所通过的隔间序列。系统进入第四隔间（由黑色节点表示）的能力对于生成崩溃输入是至关重要的。

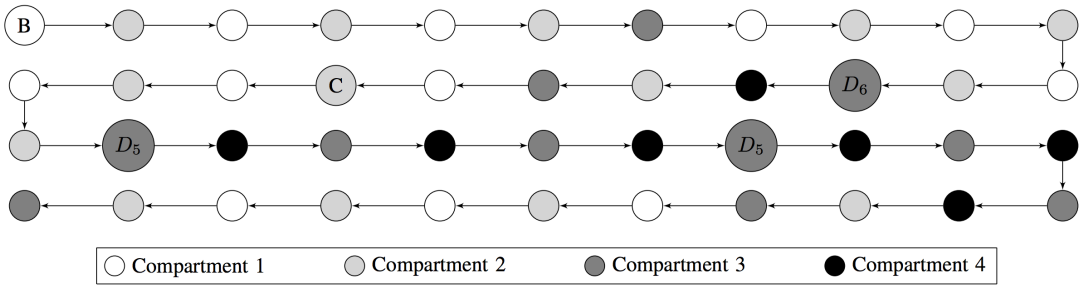


图 6-6. 对二进制文件 2b03cf01 的崩溃输入跟踪的执行流程

如图 6-7 所示位系统发现新隔区的过程。每个节点是一个函数，每个边都是一个函数调用，但返回边被排除以保持可读性。节点 A 是入口点。节点 B 包含一个魔数检查，需要符号执行组件来解析。节点 C 包含另一个魔数检查。

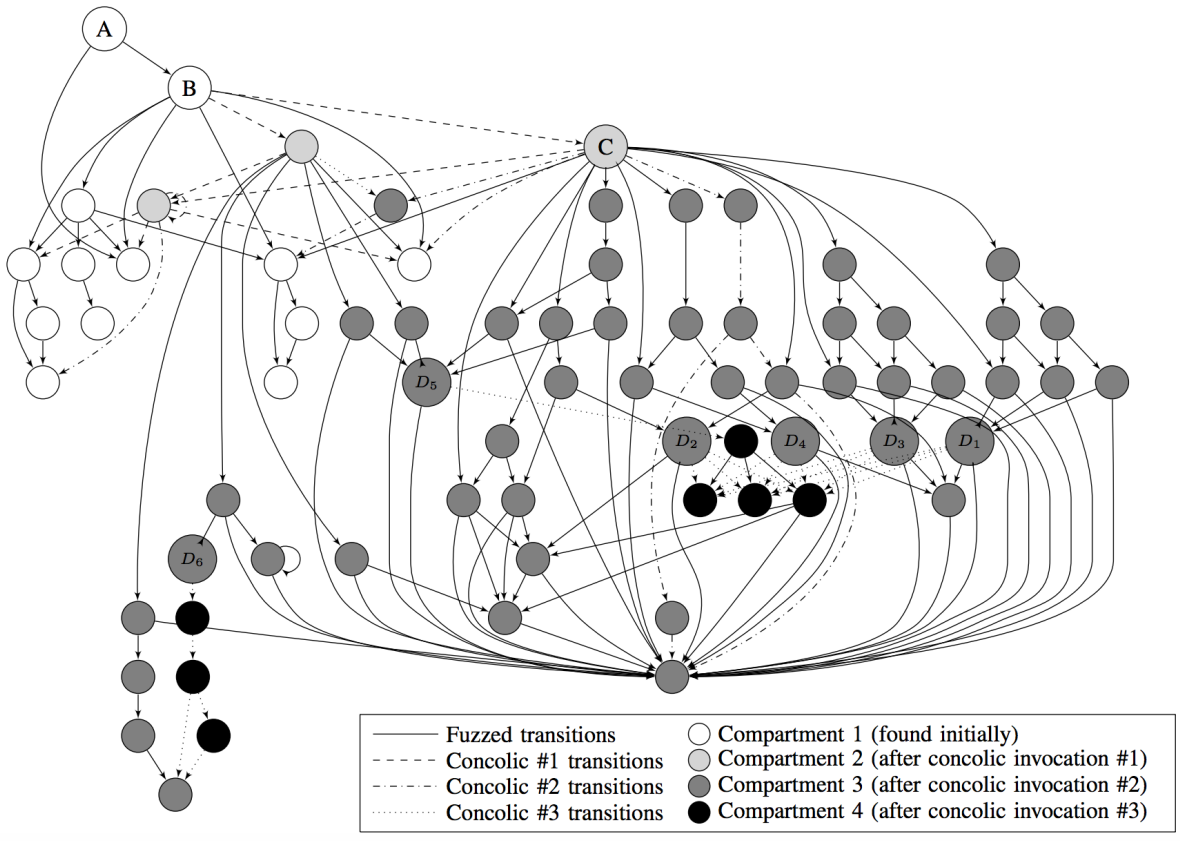


图 6-7. 系统发现新隔区的过程

系统的新输入之后被传递回到模糊器，以便快速评估由变化产生的新覆盖。表 6-6 显示了作为新输入的结果第一次执行的代码。这个新组件处理的用户输入

不再是具体的，而是一般的，使其适合于模糊器。从这一点开始，模糊器继续改变这些输入，直到它触发由应用程序中缺少的清理检查引起的漏洞。

表 6-5. 一个带有 switch 语句的函数，用于对多个特定值测试用户输入

```
1    int8_t get_new_breaker_by_model_id
                (CIRCUIT_MODELS_T model_id, breaker_t *
                breaker_space, uint8_t breaker_space_idx) {
2     int8_t res = SUCCESS;
3     switch(model_id) {
4           case FIFTEEN_AMP:
5                 create_breaker(15, breaker_space, breaker_space_idx);
6                 break;
7           case TWENTY_AMP:
8                 create breaker(20, breaker_space, breaker_space_idx);
9                 break;
10          default:
11                //invalidmodelid
12                res=−1;
13    }
14    return res;
15   }
```

在语义上，该漏洞涉及在用户创建的电路中初始化新的断路器对象。稍后，电路将被测试连接性等，并且将根据组成电路的材料来调用特定于组件的逻辑。满足检查以添加断路器将扩展错误搜索覆盖范围，以包括断路器特定的代码。触发此漏洞需要在提供的电路图中包含特制的断路器组件。触发创建这些组件所需的输入是系统在第三个符号执行调用中恢复的输入，最后的模糊测试调用使它们足够多，以触发缺陷的边缘情况。

表 6-6.由于传递特定检查而执行的代码

```
1    static void create_breaker
                (uint8_t amp_rating, breaker_t *
                breaker_space, uint8_t breaker_space_idx) {
2     breaker_space−>id = breaker_space_idx;
3     breaker_space−>amp_rating = amp_rating;
4     breaker_space−>outlets = list_create_dup();
5     if(breakerspace−>outlets == NULL) { _terminate(ERRNO_ALLOC); }
6    }
```

崩溃输入所采用的最后路径如图 6-6 所示。从入口点开始，该路径经过逐渐更难到达的隔区（由节点的不同颜色表示），直到触发边缘的条件被创建。这个二进制在基本实验中没有崩溃 - 独立的模糊器不能通过复杂检查到达保护代码

的区域，并且符号探索引擎在输入处理代码中经历了几乎立即发生的路径爆炸。通过结合模糊测试和符号执行的优点，本系统对这个二进制文件大约两个小时发现崩溃。

6.7 本章小结

本章对漏洞发现系统进行了评估实验，采用 DARPA 网络大挑战（CGC）的资格赛提供的 126 个应用程序作为数据集，设置了对单独模糊测试、单独符号执行、以及本文实现的漏洞挖掘系统三个对象的对比实验，本系统能够识别 77 个独特应用程序中的崩溃，与基础实验的并集（71 个漏洞）相比，改善了 6 个崩溃（8.45%）。之后描述了实验过程中本系统状态转移覆盖分析，与程序隔区覆盖分析。本章最后对二进制文件 2b03cf01 的漏洞发现过程进行了详细的描述，包括分条语句执行过程解析，以及每个执行过程发现的基本块数量对比。

结　论

在本文中，我们提出了漏洞挖掘系统这样一个工具，结合动态模糊测试和符号执行，能够有效地找到潜藏在二进制中的漏洞。本文介绍一个二进制隔区的概念，这在很大程度上分离了功能和代码。在本系统中，模糊器提供了快速且廉价的隔区探索，有效地探索循环和简单的检查，但通常不能在隔区之间转换。当考虑到循环和内部检查时，选择性的符号执行将进入状态爆炸，但它在找到二进制中隔区之间的路径时非常有效。通过结合这两种各自都具有一定缺点的技术，系统能够探索二进制内更大的功能空间。

我们给出的漏洞挖掘系统，是一种混合的漏洞挖掘工具，它以一种互补的方式平衡了模糊测试（Fuzzing）和可选择的符号执行（Selective Concolic Execution），用来找到更深层的漏洞。低成本的模糊测试用来探索应用隔区，符号执行测试用来生成复杂的，用来分隔隔区检查的输入。

在大多数情况下，模糊测试单独就可以充分地探索大部分路径，只要简单地通过随机的位翻转和其他突变策略就可以找到它们。由于模糊器基于原生代码执行，在大多数情况下，它的性能优于符号执行。因此，大多数探索工作交给了模糊器，这将快速地找到许多路径，符号执行引擎只是去解决更困难的约束。符号执行模块调用查找新路径方法调用引擎，若找到可以通过检查的输入，它将返回二进制文件可执行的下一步路径集合，否则返回空集合。

通过结合两种技术各自的优势，我们弥补了他们各自特定的缺陷：既有效避免了符号执行原有的路径爆炸，又充分解决了模糊测试的盲目性、不完备性。本系统使用可选择的符号执行，只去探索被模糊器认为感兴趣的路径，并为模糊器无法通过的检查生成有效输入，并对其作出了预约束控制、探索缓存等优化，大大提升了系统的执行性能。最后本文在 CGC 发布的 126 个应用程序上对本系统进行了测试评估，系统发现 77 次崩溃，与基本模糊器发现的 68 次相比，有了实质性的改进。相信这种技术，在对所有类别二进制文件进行通用漏洞发现上，具有良好的前景。

参考文献

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancingsymbolic execution with veritesting. In Proceedings of the InternationalConference on Software Engineering (ICSE), pages 1083–1094. ACM,2014.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In USENIXAnnual Technical Conference, FREENIX Track, pages 41–46, 2005.

[3] S. Bucur. Improving Scalability of Symbolic Execution for Softwarewith Complex Environment Interfaces. PhD thesis, Ecole Polytechnique ´Fed´ erale de Lausanne, 2015. ´

[4] 梁晓兵. 面向二进制程序漏洞挖掘的相关技术研究[D]. 北京: 北京邮电大学, 2012.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted andautomatic generation of high-coverage tests for complex systemsprograms. In Proceedings of the USENIX Symposium on OperatingSystems Design and Implementation (OSDI), 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler.EXE: Automatically generating inputs of death. ACM Transactions onInformation and System Security (TISSEC), 12(2):10, 2014.

[7] 李彤, 黄轩, 刘海燕, 等. 基于 Fuzzing 的软件漏洞发掘技术[J]. 价值工程, 2014, 33(3): 197-199.

[8] 邵林, 张小松, 苏恩标. 一种基于 fuzzing 技术的漏洞发掘新思路[J]. 计算机应用研究, 2009, 26(3): 1086-1088.

[9] G. Campana. Fuzzgrind: un outil de fuzzing automatique. In Actes du7eme symposium sur la s ` ecurit ´ e des technologies de linformation et ´des communications (SSTIC), 2013.

[10] D. Caselden, A. Bazhanyuk, M. Payer, L. Szekeres, S. McCamant, andD. Song. Transformation-aware exploit generation using a HI-CFG.Technical report, UCB/EECS-2013-85, 2015.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. UnleashingMayhem on binary code. In Proceedings of the IEEE Symposium onSecurity and Privacy, 2012.

[12] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide,and J. Regehr. Taming compiler fuzzers. In Proceedings of theACM SIGPLAN Conference on

Programming Language Design andImplementation (PLDI), volume 48. ACM, 2013.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform forin-vivo multi-path analysis of software systems. In Proceedings ofthe Sixteenth International Conference on Architectural Support forProgramming Languages and Operating Systems, ASPLOS XVI, 265–278. ACM, 2011.

[14] DARPA. Cyber Grand Challenge. http://cybergrandchallenge.com.

[15] DARPA. Cyber Grand Challenge Challenge Repository. https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges.

[16] J. DeMott. Understanding how fuzzing relates to a vulnerability likeHeartbleed. http://labs.bromium.com/2014/05/14/understanding-how-fuzzing-relates-to-a-vulnerability-like-heartbleed/.

[17] C. Details. Vulnerability distribution of CVE security vulnerabilitiesby type. http://www.cvedetails.com/vulnerabilities-by-types.php.

[18] W. Drewry and T. Ormandy. Flayer: Exposing application internals.In Proceedings of the USENIX Workshop on Offensive Technologies(WOOT), 2015.

[19] 魏瑜豪,张玉清.基于 fuzzing 的 MP3 播放软件漏洞发掘技术[J].计算机工程,2007,33(24):158-167.

[20] D. Engler and D. Dunbar. Under-constrained execution: Makingautomatic code destruction easy and scalable. In Proceedings of theInternational Symposium on Software Testing and Analysis (ISSTA).ACM, 2007.

[21] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whiteboxfuzzing. In Proceedings of the International Conference on SoftwareEngineering (ICSE), 2013.

[22] P. Garg. Fuzzing - mutation vs. generation. http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automatedrandom testing. In Proceedings of the ACM SIGPLAN Conferenceon Programming Language Design and Implementation (PLDI),volume 40, pages 213–223. ACM, 2005.

[24] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzingfor security testing. Communications of the ACM, 55(3):40–44, 2012.

[25] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsingfor overflows: A guided fuzzer to find buffer boundary violations. InProceedings of the USENIX Security Symposium, 2013.

[26] LegitBS. DEFCON Capture the Flag. https://legitbs.net/.
[27] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliabilityof UNIX utilities. Communications of the ACM, 33(12):32–44, 1990.
[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweightdynamic binary instrumentation. In Proceedings of the ACM SIGPLANConference on Programming Language Design and Implementation(PLDI), volume 42, pages 89–100. ACM, 2007.
[29] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. TheBORG: Nanoprobing binaries for buffer overreads. In Proceedings ofthe ACM Conference on Data and Application Security and Privacy(CODASPY). ACM, 2015.
[30] B. S. Pak. Hybrid fuzz testing: Discovering software bugs via fuzzingand symbolic execution. Master's thesis, School of Computer Science,Carnegie Mellon University, May 2012.
[31] D. A. Ramos and D. Engler. Under-constrained symbolic execution:Correctness checking for real code. In Proceedings of the USENIXSecurity Symposium, 2015.
[32] Secunia. Resources vulnerability review 2015. http://secunia.com/resources/vulnerability-review/introduction/.
[33] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna.Firmalice - automatic detection of authentication bypass vulnerabilitiesin binary firmware. In Proceedings of the Symposium on Network andDistributed System Security (NDSS), 2015.
[34] Newsome J. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signatu re Generation of Exploit Attacks on Commodity Software[C]//Proceedings of the 12th Annual Network and Distributed System Security Symposium. San Diego, C alifornia, USA: [s. n.], 2005.
[35] MILLER B P, FREDRIKSEN L, SO B. An empirical study of the reliabilityof UNIX utilities [J] . Communications of the ACM ,1990, 33(12) : 32- 44.
[36] MILLER B P, KOSKI D, LEE C P, et al. Fuzz revisited: a reexaminationof the reliability of UNIX utilities and services[R] . Madison:University of Wisconsin-Madison, 1995 .
[37] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-awaredirected fuzzing tool for automatic software vulnerability detection. InProceedings of the IEEE Symposium on Security and Privacy, 2010.
[38] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[39] Oehlert P. Violating Assumptions with Fuzzing[J]. IEEE Security & Privacy, 200 5, 3(2): 58-62.

[40] Molnar D, Wagner D. Catchconv: Symbolic Execution and Runtime Type Inferen ce for Integer Conversion Errors[Z]. [S. l.]: UC Berkeley EECS, 2007.

[41] Godefroid P, Levin M, Molnar D. Automated Whitebox Fuzz Testing[Z]. [S. l.]: Microsoft Research, 2007.

[42] King J C. Symbolic Execution and Program Testing[J]. Journal of the ACM, 1976 , 19(7): 385-394.

攻读硕士学位期间发表的学术论文

[1] Zhang W, Song B, Bai E. A trusted real-time scheduling model for wireless sensor networks[J]. Journal of Sensors, 2016, 2016. UT WOS:000372976300001, Impact Factor 1.704

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《模糊测试与符号执行相结合的漏洞发现技术研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。

作者签名：　　日期：　　年　　月　　日

学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。

作者签名：　　日期：　　年　　月　　日

导师签名：　　日期：　　年　　月　　日

致　谢

在哈工大硕士期间的学习和生活这两年，实验室老师、同学、工大规则严格功夫到家的精神很大程度影响了我，对事或者对人，在人生阅历或者认知上，都有所收获。

首先我要由衷的感谢我的导师张伟哲。在求学的这两年，老师在学问上和做人上都给了我很大启发。老师在学术上的成就很高，致力于研究的方向和领域都处在前沿，在课题遇到棘手问题束手无策时，老师总能给我注入新的思路和见解，让我有机会接触到新领域的知识；老师对待学术上严肃认真的态度，也让我受益匪浅，撰写论文一丝不苟，做实验讲究实事求是，理论推导要有理有据，老师这种对学术精神的坚守我在课题研究中体会很深，也正是老师的这种学术精神才能让我做出更好更完善的paper，同时也让我更加明白为人处世也应该明明白白实事求是。记得老师说过我做学术追求的是学术本身，因为喜欢学术而去做并坚持。老师对待学术的热忱，对追求的坚守，让我在人生成长了也懂得动力和激情在做自己喜欢做的事时是最大的。

同时，还要感谢实验室老师张宏莉在学习和生活上给我的指导和帮助，感谢何慧老师在生活中亲切的关心和问候给了我妈妈一样的温暖，感谢左老师给了我一个和谐舒适学习的实验室环境，感谢实验室王岳、李肖强、赵尚杰、杨鹏等一大群一起在实验室中生活学习欢乐的同学，在遇到问题时帮我耐心的分析，在需要帮助的时候热情的伸出双手。

我要感谢一起做毕设的师妹在课题上对我的帮助，没有他们的坚持我的毕设也不能顺利下去。感谢师兄在课题上的帮助和指导。感谢朋友在我心情低谷毕设做不下去的时候，给我打气让我相信我可以。

要感谢的人太多太多，家人、朋友、老师，让我能在挫折中坚持，在跌倒后爬起来，在哭过后继续前行。是他们给了我勇气和力量，让我能一直向前。最后，祝愿这些出现在我的人生中与我同行的，能一辈子幸福快乐。我会继续努力，朝着我的未来越走越远。