分类号 TP391 学校代码 10590
UDC 620 密 级 公开

深圳大学博士学位论文

融合程序分析与测试的内存安全漏洞检测技术研究

学位申请人姓名 文成

专 业 名 称 计算机科学与技术

学院（系、所） 计算机与软件学院

指导教师姓名 秦胜潮

深圳大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文 融合程序分析与测试的内存安全漏洞检测技术研究 是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律后果由本人承担。

论文作者签名：文成　　日期：2022年 11月 28日

学位论文使用授权说明

（必须装订在印刷本首页）

本学位论文作者完全了解深圳大学关于收集、保存、使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属深圳大学。学校有权保留学位论文并向国家主管部门或其他机构送交论文的电子版和纸质版，允许论文被查阅和借阅。本人授权深圳大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（涉密学位论文在解密后适用本授权书）

论文作者签名：文成　　导师签名：秦胜渊

日期：2022 年 11月 28日　　日期：2022 年 11月 28日

摘　要

随着软件的广泛应用和人们对软件质量需求的不断提升，软件的安全性和可靠性越来越受到重视。内存安全漏洞是指在软件中对内存进行不恰当的分配、释放、读取和写入等操作而引入的缺陷和不足，它是一种高危且普遍出现的漏洞，经常引发软件系统的崩溃或被恶意利用实施网络攻击，甚至可能导致严重经济损失或人员伤亡。因此，研究有效自动化的内存安全漏洞检测技术，尽早发现软件中的漏洞，具有重要的现实意义。

近年来，内存安全漏洞的检测在软件工程和计算机安全等领域得到了广泛的关注，尽管研究人员从多种不同角度对内存安全漏洞检测进行了广泛且深入的探索，但在特定场景和技术局限性两个方面仍存在较大的挑战。一方面，内存安全漏洞的类型多种多样，不同类型的漏洞有其独特的检测难点，有些漏洞的触发涉及内存的复杂状态以及连续的状态变迁，通常十分隐蔽，难以被发现；内存安全漏洞不仅存在于串行程序当中，也可能存在于并发程序中，而当前主流的漏洞检测技术主要面向串行程序，无法有效检测出并发程序中由于多线程交错执行而引发的内存安全漏洞。另一方面，一些漏洞检测技术本身也存在一定的局限性，例如静态分析方法虽然代码覆盖率较高，但分析过程中会引入实际不可行路径和不可达状态，面临着误报率高的问题；动态测试方法虽然准确性高，但通常覆盖率低，对程序状态空间的探索不够充分，故漏报率较高。因此，仍迫切需要有效且实用的内存安全漏洞检测方法。

本文先对内存安全漏洞检测相关的研究工作进行了系统性的归纳与分析，并按照从易到难、由浅到深的顺序对其中的三类关键漏洞（即内存消耗漏洞、内存时序漏洞和内存并发漏洞）的检测进行了研究：（1）内存消耗漏洞是指软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终导致可用内存的耗尽。这一类漏洞的触发不仅依赖于程序的执行路径，还依赖于执行路径上的内存消耗。（2）内存时序漏洞是指程序在使用内存时，违反了内存使用的安全时序规则。这一类漏洞的触发往往涉及一系列内存操作，这些内存操作可能并不都位于某一个代码块中，仅当以某种特定的顺序执行这些内存操作时才触发漏洞。（3）内存并发漏洞是指两个或两个以上的线程因线程交错交互地对内存进行操作而引起程序崩溃、挂起，或产生与串行执行不同的结果。这一类漏洞的触发涉及多个线程之间的复杂交错执行，其执行结果具有不确定性，且执行交错空间庞大。

本文围绕当前内存安全漏洞检测面临的问题与挑战，以提高内存安全漏洞检测技术

的有效性和实用性作为目标，提出了一套通用可扩展的内存安全漏洞检测框架。该框架先采用轻量级的程序静态分技术析定位疑似漏洞以及可能出错的程序语句，后利用自动化测试技术有目的地发现特定的内存安全漏洞，通过把轻量级的程序分析技术与自动化测试技术相结合，取长补短，提高了发现漏洞的能力和效率。在该检测框架的理论指导下，本文对于内存安全漏洞的检测问题采取分而治之的策略，重点针对内存消耗漏洞、内存时序漏洞和内存并发漏洞的检测问题提出切实有效的解决方案。本文的主要贡献和创新点总结如下：

（1）针对内存消耗漏洞的检测问题，本文提出了一种融合程序分析与内存消耗导向的模糊测试的方法。该方法首次提出增加内存消耗这个新维度来指导模糊测试，使得模糊测试能逐步生成能造成过量内存消耗的测试用例，弥补了当前模糊测试技术在检测内存消耗漏洞方面的盲区。实验结果表明，相比业界先进的方法（如 AFL、AFLfast、PerfFuzz、FairFuzz、Angora、QSYM），本文提出的方法发现的内存消耗漏洞数量至少要多 17.9%，并且速度上更快，至少是其它方法的 2.07 倍。

（2）针对内存时序漏洞的检测问题，本文提出了一种融合程序分析与操作序列导向的模糊测试的方法。该方法首次提出将违反内存使用的安全时序规则的操作序列用于引导模糊测试，提升了模糊测试在发现内存时序漏洞方面的有效性和效率。实验结果显示，相比业界先进的方法（如 AFL、AFLFast、FairFuzz、MOpt、Angora、QSYM），本文提出的方法发现的内存时序漏洞的数量要多 20% 以上，并且实现了至少 2.63 倍的提速。

（3）针对内存并发漏洞的检测问题，本文提出了一种融合程序分析与受控并发测试的方法。该方法创新性地提出了一种新型的系统性测试方法，采用一种基于周期性执行的调度方案来主动控制线程调度，能根据历史执行信息循序渐进地高效探索调度空间，很大程度提升了发现并发漏洞的效率。实验结果表明，相比业界先进的方法（如 IPB、IDB、PCT、Maple、ConVul 等），本文所提的方法更有效，在同样时间内发现的内存并发漏洞的数量至少要多于 29%。

（4）本文将提出的内存安全漏洞检测技术转化为一套实际可用的工具集，在主流开源软件中发现了 28 个内存消耗漏洞、7 个内存时序漏洞，5 个内存并发漏洞。发现的漏洞也都采取了负责任的披露，其中 36 个漏洞已被国际 CVE 漏洞数据库收录。

简言之，本文针对三类隐藏深、难发现、危害大的内存安全漏洞提出了一套新颖实用的分析方法与检测技术，为减少业界实际应用程序中的内存安全问题、提升软件质量提出了行之有效的解决思路和方案。

关键词： 内存安全漏洞，程序分析，模糊测试，并发测试，漏洞检测

Abstract

Nowadays, the security and reliability of software systems are of concern while the software is spreading all over the world. Memory safety vulnerability refers to the defects and deficiencies introduced by improper allocation, release, reading, or writing of memory in software. Memory safety vulnerability is a type of high-risk and widespread vulnerability, which often result in software systems crashing or being affected by malicious attacks, leading to serious economic losses or casualties. Therefore, it is of great practical significance to study and develop effective and automatic memory safety vulnerability detection techniques, improving the quality, reliability, and security of software.

In recent years, researchers have carried out extensive and in-depth research on memory safety vulnerability detection from a variety of perspectives. However, there are still great challenges in specific memory safety vulnerability detection scenarios and technical limitations of current vulnerability detection techniques. On one hand, there are many types of memory safety vulnerabilities, and different types of vulnerabilities have their own characteristic. Some vulnerabilities involve complex program states and continuous state transitions on memory, which make it difficult to be detected. Memory safety vulnerabilities exist not only in serial programs, but also in concurrent programs. Current mainstream vulnerability detection techniques are mainly for serial programs, which cannot effectively detect memory safety vulnerabilities caused by threading interleaving in concurrent programs. On the other hand, vulnerability detection techniques also have their own limitations. For example, static analysis inevitably invokes some infeasible paths or unreachable program states, so as to face the problem of high false positives. Dynamic testing often achieves low code coverage and explores insufficient program state space, leading to missing the witness of vulnerabilities. Therefore, effective and practical methods for detecting memory security vulnerabilities are still urgently needed.

In this paper, we have systematically summarized the related work on memory safety vulnerability detection, and deeply studied the detection of three types of memory safety vulnerabilities, that is, memory consumption vulnerability, temporal memory safety vulnerability, and concurrency memory safety vulnerability. (1) Memory consumption vulnerability refers to that the program does not properly control the allocation and maintenance of limited memory re-

sources, so an actor can affect the amount of memory consumed, eventually leading to the exhaustion of available resources. The occurrence of such vulnerabilities depends not only on the program's execution path but also on the memory consumption on the execution path. (2) Temporal memory safety vulnerability refers to that a program execution violates the temporal safety rules of memory usage when using memory. This type of vulnerability often involves a series of memory operations, which may not all be located in the same code block. The vulnerability is triggered only when these memory operations are executed in a specific order. (3) Concurrency memory safety vulnerability refers to two or more threads interleaved and interactive operations on memory, causing a program to crash or hang or produce different results from serial execution. The triggering of this type of vulnerability involves complex interleaving between multiple threads, with uncertain execution results and huge interleaving space.

To address the aforementioned challenges, we propose a general framework for detecting memory safety vulnerabilities. The framework first uses lightweight program static analysis to analyze and locate potential vulnerabilities and program statements that may make mistakes, and then perform automated testing to discover specific memory safety vulnerabilities guided by the results from static analysis. By combining lightweight program analysis with automated testing, they can learn from each other's strengths to complement each other's weaknesses, and improve the ability and efficiency of vulnerability detection. Under the theoretical guidance of the framework, we adopt the divide-and-conquer strategy for memory safety vulnerability detection, focusing on detecting memory consumption vulnerabilities, temporal memory safety vulnerabilities, and concurrency memory safety vulnerabilities. The main contributions of this paper are summarized as follows:

(1) We propose a method for detecting memory consumption vulnerabilities, namely MemLock, based on program analysis and memory usage-guided fuzzing. To the best of our knowledge, MemLock is the first dedicated fuzzing technique that is guided by the memory consumption in a program path, which is complementary to the coverage guidance. The experiment results show that MemLock substantially outperforms state-of-the-art tools (*i.e.*, AFL、AFLfast、PerfFuzz、FairFuzz、Angora and QSYM). It finds 17.9% more vulnerabilities, and it also can discover memory consumption vulnerabilities at least 2.07 times faster than other baseline tools.

(2) We propose a method for detecting temporal memory safety vulnerabilities, namely UAFL, based on program analysis and operation sequence-guided fuzzing. To the best of our knowledge, UAFL is the first dedicated fuzzing technique that uses the operation sequence

to guide fuzzing, which improves the effectiveness and efficiency of fuzzing in discovering temporal memory safety vulnerabilities. The experimental results show that UAFL significantly outperforms the state-of-the-art tools (*i.e.*, AFL, AFLFast, FairFuzz, MOpt, Angora, and QSYM). It finds 20.9% more temporal memory safety vulnerabilities and achieves at least 2.63 times speedup.

(3) We propose a method PERIOD for detecting concurrency memory safety vulnerabilities based on program analysis and controlled concurrency testing. We innovatively propose a new systematic testing technique, which uses a scheduling scheme based on periodic execution to actively control thread scheduling. It can efficiently explore scheduling space step by step according to historical execution information, and greatly improves the efficiency of concurrency vulnerability detection. The experimental results show that PERIOD is significantly better than other state-of-the-art concurrent program analysis and testing techniques (*e.g.*, IPB, IDB, PCT, Maple, ConVul) in detecting concurrency memory safety bugs, resulting in 29% more vulnerability findings.

(4) We have implemented the proposed method as a toolkit and applied the toolkit to a good number of real-world open-source software. We successfully found 28 memory consumption vulnerabilities, 7 temporal memory safety vulnerabilities, and 5 concurrency memory safety vulnerabilities. These vulnerabilities were disclosed responsibly. 36 of them were included in the CVE security vulnerability database.

In short, we propose a set of novel and practical analysis and detection techniques for three types of memory safety bugs/vulnerabilities that are often hidden deep, difficult to find, and can be security-critical. We have paved an effective way to reduce memory safety issues and help improve software quality in real-world programs.

Key word: Memory Safety Vulnerabilities; Program Analysis; Fuzz Testing; Concurrency Testing; Vulnerability Detection

目　　录

第 1 章　绪论

1.1　研究背景和动机

软件漏洞是指软件在设计或实现过程中存在的缺陷和不足。这些缺陷和不足可能会导致软件在运行过程中出现错误或发生崩溃，也可能会导致软件被恶意利用，从而使攻击者能在未授权的情况利用这些漏洞对软件安全造成威胁或破坏[1,2]。就软件的生命周期来说，在设计和开发的过程中不引入任何漏洞几乎是不可能的。随着软件应用规模的不断扩大和软件复杂性的日益增长，由于软件漏洞而导致严重经济损失或人员伤亡的安全事故层出不穷，例如阿丽娜亚号航天火箭的爆炸[3]、北美大停电事故[4]、波音 737-MAX-8 的坠毁[5]、“永恒之蓝” 安全事件[6]、雅虎 30 亿邮箱信息泄漏[7] 等。

在众多潜在的软件漏洞中，内存安全漏洞出现的非常普遍并且产生的危害巨大。根据微软公司的统计，每年微软产品中通过安全更新修复的漏洞中，约 70% 的漏洞为内存安全漏洞[8]；同样，谷歌公司在 Chromium 浏览器的漏洞报告中也称，Chromium 浏览器中发现的漏洞约有 70% 为内存安全漏洞[9]。所谓内存安全漏洞，是指在软件中对内存进行不恰当的分配、释放、读取和写入等操作而引入的漏洞。常见的内存安全漏洞包括缓冲区溢出、释放后使用、空指针解引用、不受控的内存消耗、内存并发漏洞等[10]。在权威机构 MITRE 发布的 “2022 年最危险的 25 种软件漏洞” 中，内存安全漏洞就占 8 种[11]。值得注意的是，缓冲区溢出、释放后使用和空指针解引用常年被列入最危险的 25 种漏洞类型，而不受控的内存消耗和内存并发漏洞在 2022 年新进入最危险的 25 种漏洞类型榜单中。各式各样的内存安全漏洞持续威胁着软件系统的正确性和安全性，因此内存安全漏洞的检测一直都是学术界和工业界研究的热点和重点。

为了应对内存安全漏洞对软件正确性和安全性造成的威胁，研究人员陆续开始研究各种类型的内存安全漏洞检测技术[12,13]。从对内存安全的保障程度来看，当前的内存安全漏洞挖掘技术总体上可以分为两类，一类技术是以定理证明、模型检测等方法为代表的形式化验证技术，能基于数学推理的方法严格证明程序的某类性质。定理证明方法[14,15] 将程序是否满足某种内存安全性质是看作是一个逻辑命题，一般使用 Hoare 逻辑[16] 等公理系统从语法推导的角度证明程序的公理语义是否满足待检验性质；模型检测[17] 则使用有穷自动机表示程序的状态迁移系统，并从语义的角度验证所建立的状态迁移系统是否为待检验性质的一个模型。这一类技术的效果明显依赖于技术人员的专业能力，并且需要消耗大量的时间与精力，

常应用于不计成本的特定受限场景（航天航空、轨道交通、操作系统微内核等）。例如，不到一万行代码的 seL4 微内核耗费了 20 人年以上的投入成本后，才声称是世界上第一个通过数学方法被证明安全的操作系统内核[18]。另一类技术是以发现漏洞为目的的程序分析与测试技术，主要包括静态分析和动态测试等[19]。这类技术侧重于自动化和可扩展性，往往不要求对程序进行完全精确的分析，这就可能带来误报、漏报等问题。近些年静态分析技术已经在数据流分析[20]，基于摘要的过程间分析[21]、符号执行[22]、抽象解释[23] 等多个方面取得了长足的进步。同时，出现了许多可扩展的自动化静态分析工具，帮助人们在开源软件中找到了很多软件漏洞；有些工具在大公司里得到应用（例如 Infer[24]、RacerD[25]、Pinpoint[26] 等），并发挥了重要作用。但静态分析方法为了构建用于分析的模型需要对程序的动态语义做某种形式的抽象，其抽象结果不可避免地会引入实际不可达状态与不可行路径，而在有限时间内判定抽象状态的可达性是非常困难的，因此会导致大量的误报。动态测试技术只探索程序的实际可达状态或可行路径，这是确保分析结果没有误报的最主要原因。但动态测试技术的有效性一方面依赖于有效的测试预言，例如谷歌公司开发的内存错误动态检测工具 AddressSanitizer[27]（ASAN）能基于指定的测试用例分析程序的运行，在运行时分析内存安全漏洞；另一方面还取决于测试用例的质量，高质量的测试用例通常可以覆盖被测程序的大多数行为，故有更大的概率发现软件漏洞，然而要构造高质量的测试用例要求专业人员深刻地理解被测程序的行为，并且耗费一定的精力与时间才能完成。此外，由于动态测试技术大多时候并不能遍历程序的所有可行路径，因此可能会错过某些可以引发程序错误的执行路径，从而导致漏报。

迄今为止，工业界采用最多的仍是动态测试方法。近些年兴起的模糊测试[28,29]是一种有效的自动化软件测试技术，其能自动生成海量的输入来反复执行程序，通过监视程序运行过程中的异常来检测软件错误/漏洞。以 AFL[30]（American Fuzzy Lop）为首的代码覆盖引导的灰盒模糊测试技术利用程序执行的反馈信息指导模糊测试，提升了测试用例生成的方向感和质量，并能与各类动态内存漏洞检测工具（例如 ASAN[27]，MSAN[31]）结合使用并自动化进行测试，在主流开源软件中挖掘到了大量的零日漏洞，进一步证明了模糊测试的实际价值。由于覆盖引导的灰盒模糊测试技术不要求测试人员深刻理解程序源代码或拥有特定领域的知识，可扩展性强，且能较快发现和确认软件漏洞，因此在工业界被广泛部署。国际安全会议和软件工程会议上关于模糊测试的研究工作也越来越多，如 Vuzzer[32]、Angora[33]、GreyOne[34]、QSYM[35]、SAVIOR[36] 等，这些工作将经典的程序分析技术与模糊测

试技术相结合，在检测内存安全漏洞的有效性和实用性上有着较大的突破。

尽管目前对内存安全漏洞检测的研究已经取得了一定的进展，但鉴于程序性质的多样性、被分析程序的规模和复杂度，以及对漏洞的误报和漏报要求，当前的程序分析与测试技术研究仍面临着一定的挑战。

1. 不同类型的内存安全漏洞，其产生方式和原因不同，其检测原理也不同。有些漏洞的触发涉及内存的复杂状态以及连续的状态变迁，通常非常隐蔽，难以被发现。例如，对内存空间不受限制的过量使用，可能导致内存资源耗尽，这类漏洞强调的是对内存空间的消耗（后文简称内存消耗漏洞）；进行内存操作时未遵守特定的时序规则，在释放内存后继续使用或再次释放已释放的内存，可能导致未定义行为或程序崩溃，这类漏洞强调的是内存操作的时序（后文简称内存时序漏洞）。现有的针对内存安全漏洞的静态分析方法大多针对程序的执行路径进行建模，动态测试方法大多以代码覆盖率作为覆盖评估标准，即在有限的时间预算内检查尽可能多的程序路径以发现错误，然而内存消耗漏洞这一类漏洞的触发不仅依赖于程序的执行路径，还依赖于执行路径上的内存消耗，因此当前的程序分析与测试方法在检测内存消耗漏洞方面存在一定的盲区。对于内存时序漏洞而言，这类漏洞往往涉及一系列内存操作，这些内存操作可能并不都位于同一个代码块中，仅当以某种特定的顺序执行这一系列内存操作时才触发错误。内存时序漏洞的形成条件十分隐蔽，且需要跟踪较长的操作序列才能发现该漏洞，因此通常很难被发现。

2. 对于不同性质和不同复杂程度的程序来说，其面临的挑战也完全不同。当前多线程的并发程序在现代多核系统中已经广泛应用，与顺序程序的内存安全漏洞相比，并发程序的内存安全漏洞难以理解、难以检测、难以重现[37,38]。并发程序涉及多个线程之间复杂的交互执行，其执行路径具有不确定性，执行交错的空间十分庞大，当多个线程交互地作用于同一块内存区域且未正确使用的同步机制进行约束时，可能引发程序错误、挂起或崩溃，这类漏洞的发生体现在线程的并发执行交错上（后文简称内存并发漏洞）。现有的内存安全漏洞检测技术主要面向串行程序，大多数面向并发程序的漏洞检测研究的关注点仍然停留在检测数据竞争这类可能引发错误的行为上[39–43]，对于由线程交错引起的内存并发漏洞的研究仍然不足。

因此对于内存消耗漏洞、内存时序漏洞、内存并发漏洞这三大类漏洞而言，当前的程序分析与测试技术仍存在一定的局限性和不足，本文的工作重点是围绕这三类漏洞检测所面临的问题与挑战，在深入地分析和表征漏洞特点的情况下，研

究有效且实用的程序分析与测试技术，提供一整套通用可扩展的内存安全漏洞检测方法，并孵化出一套可用于实际应用程序的工具集。

1.2 国内外研究现状

本章首先对内存安全漏洞通用检测技术的研究现状进行总结，然后，再进一步梳理和归纳面向内存消耗漏洞、内存时序漏洞和内存并发漏洞的专用检测技术的研究现状。

软件漏洞检测技术的研究是一个长期的、全方位、多层次的工作，其相关工作有着很长的历史积累。早在 20 世纪 70 年代中期，美国就重点针对操作系统的安全漏洞进行研究，启动了 PA（Protection Analysis Project）研究计划[44] 以增强计算机操作系统的安全性。上世纪 90 年代，美国在漏洞分类、挖掘、评级等领域的研究都较为详尽，逐步形成行业内有影响力的标准，为漏洞库的发展奠定了稳固的基础。2006 年，美国政府建立了美国国家安全漏洞库 NVD[45]（National Vulnerability Database），专门针对漏洞的名称、来源、CVE（Common Vulnerabilities & Exposures）标号，CVSS 分数（Common Vulnerability Scoring System）等信息进行描述。NVD 与学术界、工业界持续维持密切合作，将漏洞信息广泛应用于安全风险评估，终端安全配置检查等领域，为国家信息安全保障做出了巨大贡献。中国的软件漏洞检测研究最早开始于科研机构。到了 2009 年，我国的软件漏洞检测研究也取得了突破性进展，相继构建了中国国家漏洞库、国家信息安全漏洞共享平台，在学术界和工业界引起了强烈的反响。随着人们对软件质量需求的提升和信息安全的发展，部分科研机构与安全公司根据自身的需求，开始研发自己的漏洞检测工具，如北大软件 CoBOT、源伞科技的静态分析工具 Pinpoint[26]、水木羽林科技的的开源模糊测试工具 Healer[46] 等，并产生了较大的影响。

经过多年的发展，软件漏洞检测研究已经在多个方面取得了长足的进步，对内存相关漏洞的特征分析和理解也不断加深[47]，正朝着自动化与智能化的方向发展。目前，主流的内存安全漏洞检测技术包括静态分析、动态分析、模糊测试等。

可扩展的静态分析工具常被开发者和安全研究人员用来检查代码错误或协助分析漏洞。Cppcheck[48] 是个采用错误模式匹配的静态分析工具，具有效率高和易用等特点。但错误模式的识别是对代码行为的简单匹配，分析效果会随着程序类型以及经验模型波动，并受到别名、堆抽象精度等因素的影响。Infer[24] 是 Meta/-Facebook 的一款开源的程序静态分析工具，基于分离逻辑[49] 和 Bi-abduction[50] 实现。Infer 能够分析 C、Java 或者 Objective-C 代码，主要应用于分析移动平台（iOS/安卓）项目，并报告潜在的问题。使用 Infer 对代码进行静态分析已经成为 Meta 开

发流程中的一个重要环节，每次代码变更，都要经过Infer的检测。Meta还开源了一款针对并发程序设计的数据竞争检测工具RacerD[25]，RacerD工具是以Infer静态分析平台为基础，采用程序分析框架Infer.AI来侦测程序错误，并产生程序中可能发生数据竞争情况的报告以提供给开发者。RacerD能对大规模程序快速地完成分析，原因在于它在检测并发问题时并没有去检查整个代码库，而是仅检查那些它认为可能并发执行的代码。2018年，源伞科技发布了核心产品Pinpoint[26]，其在工业级静态漏洞检测技术中引入了精确的指针分析技术，在极大的提升精度和召回率的同时，也保证了在可接受的时间内完成百万行甚至千万行代码的检测扫描。因此，Pinpoint得到了国内外同行专家的高度认可。以上静态分析工具都能在大规模的现实应用程序中较好地应用，它们在确保可扩展性的同时尽可能提升分析精度。但静态分析的保守抽象会产生很多的误报，低精度的静态分析缺乏实用性，而精确的静态分析又需要很好的设计并牺牲可扩展性，如何在精度和可扩展性之间取得平衡是静态分析技术朝着实用性方向迈进必须要考虑的重要问题。

动态分析由于能够获取程序运行时的具体值，可以在分析成本不高的情况下得到准确的分析结果。Sanitizers是谷歌研发的基于动态分析的漏洞检测开源工具集，包括了ASAN、MSAN、TSAN等。ASAN[27]是一个高效的内存错误检测工具，它由一个编译时插桩模块和一个动态运行库组成。ASAN能快速检测诸如释放后使用、缓冲区溢出、空指针解引用等类型的程序错误，但不能检测读取未初始化内存的程序错误。MSAN[31]能自动跟踪内存中未初始化数据的传播，当程序读取未初始化内存时及时报告警告。LSAN专门用于检测程序中的内存泄漏错误，TSAN[51]则是一个检查多线程程序是否包含数据竞争行为的工具。EffectiveSan[52]利用C/C++动态类型实现内存安全漏洞的检测。由于目前很多内存错误都可以看作是类型错误，EffectiveSan[52]将内存错误的检测转化为类型错误的检测问题，通过在运行时使用之前动态检查每个对象的“有效类型”来检测内存错误。尽管动态分析的精确度较高，但它的分析结果往往依赖特定的输入，调度，执行路径等，因此这类方法通常依赖于人工设计测试用例或结合模糊测试方法自动化地生成测试用例。

符号执行也是一种程序分析技术，它使用抽象的符号代替具体值来模拟程序的执行，能通过约束求解器来得到可以执行到目标代码的具体输入值。符号执行技术常用于源代码的正确性与安全性分析。起初受制于早期的计算能力与约束求解工具的能力，符号执行技术并没有在实际中得到广泛的应用。近年来，随着计算能力的提升、SAT/SMT技术和工具的蓬勃发展[53,54]，符号执行技术取得了长足的进步，出现了以动态符号执行(Concolic Testing)[55,56]为代表的一批新的符号执行

技术和工具，以及以SAGE[57]、KLEE[58]、Angr[59]、SPF[60]、S2E[61]为代表的符号执行工具。然而，虽然符号执行是一种精确的程序分析方法，但容易引发路径爆炸问题，因而可扩展性较差。

模糊测试的概念早在1989年就已经被提出[62]，早期以黑盒模糊测试为主，直到2013年底AFL[30]的发布后，模糊测试的实用价值才得到认可。AFL首次采用程序控制流程图中的边覆盖信息反馈和编译插桩实现了代码覆盖导向的灰盒模糊测试技术。最近几年，大量基于AFL的研究工作应运而生，这些工作大致可以归为两类。一类研究工作是代码覆盖导向的模糊测试，以不断提高代码覆盖率为目标，最具代表性的技术包括AFLFast[63]、FairFuzz[64]、Angora[33]等。另一类研究工作是定向模糊测试，能够关注于程序中的一组特定目标语句，并引导模糊测试有针对性地朝向目标所在的代码区域进行探索。例如，AFLGo[65]和Hawkeye[66]根据测试路径到目标位置的距离评估，动态调整种子排序和对种子的变异次数，逐步引导程序执行到目标点。此外，AFL本身也存在一些实现上的缺陷，通过对这些缺陷的修复，可以优化上述方法的效果。例如，CollAFL[67]针对AFL分支覆盖记录存在哈希冲突的问题，设计了一种算法把冲突率降低到接近零，大大增加了覆盖信息的准确率，提升了AFL的效果。在工业界，谷歌公司的开源项目OSS-Fuzz[68]利用最新的模糊测试技术与可拓展的分布式执行，为开源软件提供连续的、大规模执行的模糊测试，并且能够集成到软件项目的开发流程中。截至2021年6月，OSS-Fuzz已经在500个开软软件中发现了超过30,000个漏洞。近期，谷歌公司开放了OSS-Fuzz后端的源代码，随后不久微软公司也开源了基于Microsoft Azure云服务的模糊测试平台OneFuzz[69]。模糊测试的基础设施与工具逐步变得更健全，逐步将测试人员从繁琐的测前准备工作中解脱出来，同时也不断提高测试效率与代码覆盖率。

近年来，综合利用多种研究方法和技术手段来检测内存安全漏洞已经成为研究热点，研究人员不断将各种程序分析技术与模糊测试技术相结合。一方面，通过模糊测试能够生成具体的测试输入以确定漏洞的存在，从而降低漏洞检测方法的误报率；另一方面，利用程序分析技术来指导测试输入的生成，可以增加代码覆盖率，提升漏洞检测效率。Vuzzer[32]是一种集成静态分析和动态污点分析的应用感知的灰盒模糊测试方法，该方法先利用静态分析提取影响控制流的即时值、魔术字符和其它特征字符串，再在模糊测试时对输入进行动态污点分析，通过将污点分析的结果反馈给种子选择和种子变异阶段，以求生成更高质量的测试输入，达到加速覆盖率提升的目的。Gan等人[34]提出了基于数据流的模糊测试技术GreyOne，

设计了模糊测试驱动的污点推断模型，可以精准推断程序路径分析转移与测试用例偏移字节的关系。Beacon[70] 通过对目标点的最弱前置条件分析，有效地修剪了大量不可行路径，从而大幅度节省了定向灰盒模糊测试的时间成本。此外，很多研究工作将动态符号执行与模糊测试相结合，通过遍历程序的执行路径，生成有效的测试输入。Driller[71] 融合了覆盖导向的模糊测试与符号执行技术，使用选择性的混合符号执行技术，当 AFL 被 “卡住” 时，调用 Angr[59] 来生成合法输入，从而有效地分析大规模程序。SAVIOR[36] 还将 AFL 与静态符号执行工具 KLEE[58] 相结合，提出了一种漏洞驱动的混合模糊测试框架。然而，随着程序复杂度提高，混合模糊测试同样面临着路径爆炸的问题。可以看出，程序分析技术不仅能有效辅助测试技术提升测试效率，也能通过与测试技术相结合来缓解误报率高的问题。而模糊测试已从早期完全随机的暴力破解法逐渐演变成了今日的不断与其他领域知识相融合，积极吸收各种相关算法思想，从而使生成的测试用例更为有效。

1.2.1 内存消耗漏洞检测

不少研究工作关注到内存消耗漏洞的检测，主要包括静态分析和动态分析方法。在模糊测试技术方面，虽然已有相关研究针对时间消耗方面的漏洞，但对内存消耗漏洞仍缺少针对性研究。

静态分析：Chin 等人[72] 提出了一种面向对象语言的类型系统，该类型系统描述了数据结构的大小和操作这些数据结构的方法所消耗的堆内存量，可以保守的分析程序运行时堆内存消耗的上界。Chu 等人[73] 提出了一种基于符号执行的内存消耗分析算法，可用于分析最坏情况下程序的内存消耗量，但该方法假设被分析程序的循环和递归都是有界的，对于大规模的复杂程序难以准确反映其真实的内存消耗量。Daniel 等人[74] 提出了基于抽象解释的栈内存消耗的静态分析方法，该方法可以计算出程序最坏情况下的栈内存消耗，并通过形式化证明的方式证明估算出的内存消耗不会被低估。此外，静态分析技术也被广泛应用于内存泄漏的检测。SATURN[75] 是一种基于函数摘要的内存泄漏检测方法。该方法在函数摘要中记录了其是否进行了内存操作和其它相关信息，并结合路径敏感和上下文敏感的静态分析实现了对过程间的内存泄漏检测，但对函数参数的不处理会导致在过程间分析时，内存空间所有权变量集合不够精确。Cherem 等人[76] 将内存泄漏的检测问题转化为一个可达性分析问题，即在目标程序的数据流图上通过分析数值从内存分配点到释放点的传播来检测是否有内存泄露发生。SPARROW[77] 是一种路径不敏感的内存泄漏静态检测工具，该工具将函数抽象建模为参数化表示的函数摘要，并将其用于相应的函数调用语句中，并对返回值进行关联性分析，以检测内

存泄漏漏洞。源伞科技针对内存泄漏，设计了一套新方法 Smoke[78]。Smoke 先使用路径不敏感算法找到整个程序中所有可能发生泄漏的候选程序路径，然后再采用逐步求精的方式，谨慎使用约束求解器来确认每个候选路径是否可行。该方法提出一个名为使用流图的数据格式，并设计了基于使用流图的状态验证算法，将基于稀疏数据结构的内存泄漏验证算法在理论上从 NP-hard 问题降低为 P 问题。

动态分析：Antunes 等人[79] 提出了一种内存消耗漏洞检测方法 PREDATOR，该方法实现在一个全自动黑盒测试工具之上，能自动化地生成测试用例并将它们注入服务器系统，并计算内存资源使用概况，预测每次测试执行的内存资源利用率。通过对内存资源利用率的预测，PREDATOR 可以报告非预期的内存消耗，从而识别内存消耗漏洞。不过 PREDATOR 主要应用于网络服务器，需要提供协议的规范来自动生成测试用例。史立敏等人[80] 提出了面向 Web 服务资源消耗漏洞的检测模型和评测框架，能够检测 Web 服务中的资源消耗漏洞，并且评测 Web 服务资源消耗的脆弱性程度，在 Web 服务受到攻击前预先分析和了解 Web 服务的资源消耗情况。吴民等人[81] 使用运行时动态监测的手段，根据函数调用时的内存堆栈信息来推断是否存在内存消耗漏洞，但是此方法严重依赖测试输入来确保对目标程序的代码覆盖，故常常发生漏报且检测效率较低。MemGuard[82] 在开源的动态二进制插桩框架 Pin 的基础上，采用基于函数族的内存管理方式来管理目标程序中不同类型的函数分配的内存信息块，并实现了对内存泄漏、内存的不匹配释放等漏洞的检测。Radmin[83] 基于目标程序的动态执行轨迹对目标程序的内存消耗进行建模，构建并执行多个概率有限自动机来检测可能的内存资源耗尽问题。由于程序的一条动态执行轨迹仅反映程序的一条执行路径，因此该方法的有效性还取决于测试输入的质量和数量。

模糊测试：SlowFuzz[84] 和 PerfFuzz[85] 以程序执行的指令数量来引导模糊测试，可以指导模糊测试生成病态的测试输入，不断使程序的运行时间增加。SlowFuzz 和 PerfFuzz 可以用于分析程序执行时的最大时间消耗，或检测算法的时间复杂性漏洞。ReScue[86] 基于遗传算法生成异常的正则表达式，自动化地对目标程序进行测试。正则表达式是开发者处理字符串的重要工具，编写不当的正则表达式可能在特定输入下有长达指数级的匹配时间，不断消耗着计算资源。ReScue 旨在检测由于正则表达式匹配造成的计算资源消耗过度的漏洞。以上方法主要关注的是时间消耗方面的漏洞，而不关注内存消耗方面的漏洞，且这些方法无法直接用于内存消耗漏洞的检测。这主要是由于内存消耗有其独特的特点，内存消耗可能在程序执行的过程中突然增加（例如函数调用、内存分配）或突然减少（例如函数返

回、内存释放），而时间消耗是随着程序的执行单调增加的，故检测内存消耗漏洞更为困难。

1.2.2 内存时序漏洞检测

内存时序漏洞的检测一直以来都是研究的热点和难点，典型的内存时序漏洞包括释放后使用和双重释放漏洞。目前大多数研究都聚焦于通用内存安全漏洞的检测，很少有专门针对内存时序漏洞的检测方法。以下从静态分析、动态分析的角度总结现有的内存时序漏洞检测技术。

静态分析：GUEB[87]是个面向二进制程序的释放后使用漏洞检测方法，它首先建立抽象内存模型作为其中间表示，然后执行专用的数值分析和指针别名分析来跟踪堆相关操作和指针传播，最后提取释放后使用序列。UAFChecker[88]以一种简单的自动机的思想，描述每一个内存对象的状态，如果内存对象能够从起始状态正确地转换到终止状态，那么该内存对象的时序就是安全的，否则执行路径上存在释放后使用漏洞。但是由于GUEB和UAFChecker是面向二进制程序的方法，且采用了不精确的反汇编技术和不可靠的指针分析，因此它们既不精确也不可靠。Tac[89]使用支持向量机（SVM）来提高指针别名分析的准确性，从而减少检测结果中的误报。由于机器学习的缺陷，Tac不能保证有效减少真实程序中的误报。CRed[90]基于需求驱动的指针分析和调用上下文深度缩减技术来检测程序中的释放后使用漏洞。它首先使用SVF[91]获取程序中的候选释放后使用漏洞序列，然后执行路径敏感分析以减少误报。为了扩展到真实程序，CRed使用了调用上下文深度缩减技术。由于没有对数组访问别名进行合理的建模以及缺乏对链表的处理，CRed会错过真实程序中的释放后使用漏洞。

动态分析：Undangle[92]是一个检测二进制文件中的释放后使用和双重释放漏洞的动态分析工具，它使用污点分析技术跟踪指针是如何被拷贝的，当内存被释放时，它能够确定是否还有指针指向被释放的内存。Undangle还能够分析在一个指定时间窗口内指向某一个特定内存位置的所有悬挂指针。然而，Undangle需要构造特定的输入以得到包含释放后使用漏洞的路径。ASAN[27]、Valgrind[93]、Doubletake[94]和EffectiveSan[52]等工具能够在程序对指针进行解引用操作时，监测指针指向内存的有效性。当已释放的内存未被重新分配时，悬挂指针指向的内存是无效的，因此这些工具能够检测出这种情况下的释放后使用漏洞。但是如果已释放的内存已经被重新分配，那么此时悬挂指针指向的内存是有效的，而上述动态分析方法无法区分程序执行时在同一内存地址分配的不同的内存对象，因为这涉及到两个不同的内存对象。换句话说，当程序重用已释放内存时，上述动态分析方法将会错

过这种情形下的释放后使用漏洞。UAFSan[95] 使用一个唯一的标识符来标记每个新分配的对象和指针。当指针解引用时，UAFSan 通过检查它们标识符的一致性来判断 UAF 是否发生了，因此可以避免因程序重用已释放内存导致的漏报。此外，类似于 Undangle，使用这些动态分析的工具需要构造特定的输入以得到包含释放后使用漏洞的路径。

1.2.3 内存并发漏洞检测

为了有效发现内存并发漏洞，研究人员已提出了多种针对并发漏洞的检测方法，以提高检测并发漏洞的效果，降低检测成本。尽管目前工业界仍以压力测试作为发现并发漏洞的主要手段，学术界已经相继将静态分析、动态分析、模糊测试、受控并发测试、预测分析等方法用于检测并发漏洞。

静态分析：吴萍等人[96] 提出了一种精确有效的多线程程序静态数据竞争检测框架，该框架应用精确的别名分析并静态模拟访问事件发生序，能够在有效时间内为小规模的多线程程序分析数据竞争问题，但在较大规模的程序上仍面临可扩展性不足的问题。Flanagan 等人[97] 认为检测数据竞争对于验证程序是否存在并发漏洞而言，既不充分也不必要，因此，他们提出了一个基于 Lipton 理论的类型系统，用于检测程序中的原子性违背问题。RELAY[98] 通过静态分析程序上每一个点的锁集来判定是否发生数据竞争，按照自底向上的函数调用方法对每一个函数进行锁集计算，在相对锁集计算完毕的同时也生成了相应的共享变量读写等访存信息。此方法采用非常保守的方式进行别名分析，存在较多误报。Wang 等人[99] 通过模式匹配的方法在 Linux 操作系统的源代码上挖掘内存并发漏洞。然而模式匹配的方法需要人为预定义模式匹配的规则，而这些规则一般难以覆盖到所有的并发漏洞的形式，容易出现漏报。此外预定义的匹配规则往往没有摒弃非并发漏洞的形式，也带来较高的误报。RacerD 工具[25] 在 Meta/Facebook 公司有着成功的实践经验，RacerD 通过分析线程安全类，检测潜在的数据竞争。为了提高可扩展性，该方法进行了模块化设计，对每个函数单独分析，并生成函数摘要，这些函数摘要可以在函数调用时复用，减少重复计算。

动态分析：面向并发程序的动态分析技术通常指基于证据的动态分析技术。Eraser[100] 最早将锁集分析算法引入数据竞争的动态分析中，它使用一种二进制重写技术来监测每一个对共享内存的引用，并在观察到的执行轨迹中验证锁行为的一致性。Djit+[101] 使用时钟向量来记录更新读写操作的先后顺序，通过这些读写操作的偏序关系来检测数据竞争。FastTrack[102] 采用轻量级逻辑时钟代替向量锁机制，只记录每个共享内存最后一次写操作的信息，大大降低了算法时间复杂度，使

得检测算法在时间上（接近于O(1)）得到了很大的提升。RaceChecker[103]结合同步和Happens-before关系修剪不可行的数据竞争，它通过将潜在的数据竞争分组，并同时对每组进行验证，提高了检测效率。尽管动态分析的误报率较低，但由于在程序运行过程中，线程每次的执行顺序具有不确定性，导致每一次执行程序得到的结果不同，因此动态分析不够全面，导致漏报率较高。

预测分析是一种基于预测的动态分析技术，主要通过对程序执行时记录的执行轨迹进行分析，对执行轨迹中的事件在一定约束条件下进行重排序，从而预测可能发生的错误调度。2014年，Huang等人[104]提出了最大因果关系模型，通过将执行序列中的事件编码为一阶逻辑约束，使其可以产生所有可行的数据竞争。2016年，Huang提出了预测分析技术UFO[105]，将最大因果关系模型进行了扩展以预测并发释放后使用漏洞。UFO首先运行插桩后的程序来获取程序的执行路径，然后使用最大因果关系模型来推断其它可能的执行路径。最后UFO将释放后使用的时序约束和可行的执行路径都编码为一阶逻辑约束，并使用约束求解器进行约束求解。理论上UFO可以确保找到所有的并发释放后使用漏洞。但由于约束求解通常非常耗时，UFO实际上使用了宽松的最大因果关系模型。这使得UFO会错过一些并发释放后使用漏洞。2019年，蔡彦等人提出了针对三类内存并发漏洞（即并发释放后使用、并发双重释放、并发空指针解引用）的预测分析方法ConVul[106]，首次基于松弛可交换事件来检测并发漏洞。该工作提出的松弛可交换事件克服了传统检测算法的不足，即使目标事件之前存在复杂的同步约束，也可以通过松弛可交换事件来判断是否可以交换。尽管ConVul可以检测到三类内存并发漏洞，但其检测能力受到可交换事件数量的影响。如果两个可交换事件之间还有许多其它的可交换事件，则ConVul会错过一些内存并发漏洞。

模糊测试：ConAFL[107]是一个结合程序静态分析和模糊测试的轻量级并发漏洞检测框架，它能检测并发释放后使用和并发双重释放等类型的漏洞。ConAFL先利用静态分析识别可能引发并发漏洞的敏感操作，然后在模糊测试的过程中改变这些敏感操作所在线程的优先级，提升了模糊测试过程中触发并发漏洞的概率。Krace[108]提出了一种用于跟踪并发执行交错探索进度的覆盖率追踪标准，即别名覆盖率（alias coverage），并针对内核同步原语进行了全面的建模，实现了精确的动态数据竞争检测。Krace将别名覆盖率组件和动态数据竞争组件在覆盖导向的模糊测试框架中进行了整合，并且成功应用在文件系统的模糊测试中。Muzz[109]是一个面向多线程程序的灰盒模糊测试技术，它采用了三种线程敏感的代码插桩，包括代码覆盖率插桩、线程上下文插桩以及线程调度插桩。在模糊测试的过程中，

这些插桩产生的运行时反馈可以帮助 Muzz 探索线程交错相关的多线程程序执行状态。以上工作侧重于改进面向并发程序的覆盖率追踪标准和生成的测试用例质量，并通过启发式方法在小范围内引入噪音来干扰线程交错，其在线程交错方面的探索是非常不足的。

受控并发测试：受控并发测试技术在程序运行时主动使用线程调度控制手段，迫使程序按照预期的执行调度去执行程序，并监视其执行时是否发现异常。受控并发测试主要可以分为随机测试方法和系统性的测试方法。随机测试方法随机地程序运行时强迫线程切换，从而能够测试到更多的线程交错。Contest[110] 通过在同步语句前随机地引入噪音（例如时延），使得程序每次运行时各个线程之间的交错执行产生差异。Sen 等人[111] 使用已有静态分析技术检测出来的数据竞争指导随机测试，在测试运行时，迫使线程按照可能产生错误的调度执行，从而将误报从潜在的数据竞争中分离出来。PCT[112] 在程序开始运行时随机指定线程的优先级，并且在程序运行期间随机在某些程序点处修改线程优先级。RPro[113] 在 PCT 的基础之上提出了缺陷半径的概念，并结合实际情况，使得每次产生优先级改变的点都只会在特定的程序运行范围内，减小了需要探索的调度空间。从某种意义上来说，PCT 是 RPro 以程序运行中的所有事件数量为缺陷半径时的一个特例。PCT 和 RPro 都属于随机测试，虽然可以产生比传统的压力测试更多的线程交错，但是这些线程交错仍然可能存在重复的情况，并且无法确保在完成一定数量的测试后一定能触发并发漏洞。系统性的测试方法试图系统地探索并发程序的完整调度空间以测试所有可能的线程交错。该技术可以确保在不同的测试执行中每次测试一种不同的线程交错，检测这些线程交错中的并发漏洞。但是由于线程交错空间十分庞大，甚至随着线程数和指令数量的增加是呈指数级增长的，因此这一类测试方法的开销过大。甚至针对较大规模的并发程序，系统性的测试方法实际不可行。为了缓解系统测试过大的线程交错空间，一些研究者提出某些优化方法用于减少相同偏序事件的交织数量，或通过某些启发式方法来降低测试成本，但代价是丢失可能触发其它并发漏洞的线程交错。例如 IPB[114] 通过将抢占点的界限值设置为小于 2 来减少需要测试的线程调度的数量。

总体来说，基于程序分析与测试的内存安全漏洞检测技术在近些年已经取得了很大的进步，但一方面在分析效率、误报、漏报和可扩展性方面依然存在可提升的空间，另一方面对于内存消耗漏洞、内存时序漏洞、内存并发漏洞方面的探索还是不够充分。

1.3 本文的研究问题

从以上国内外研究现状可以看出，尽管研究学者已经对内存安全漏洞检测的相关方法、理论分析与实证研究进行了多角度的深入探索，但其仍存在较大的局限性与不足。

本文在对内存安全漏洞检测相关研究工作进行了系统性的归纳与分析的基础上，按照从易到难、由浅到深的顺序将内存安全漏洞划分为内存消耗漏洞、内存时序漏洞和内存并发漏洞。本文围绕当前内存安全漏洞检测面临的问题与挑战，以提高内存安全漏洞检测技术的有效性和实用性作为目标，着眼于现实世界中主流开源软件的漏洞检测场景，以程序分析和测试技术相结合作为主要途径，对内存安全漏洞对检测问题采取分而治之的策略，分别针对内存消耗漏洞、内存时序漏洞和内存并发漏洞的检测展开深入研究。

内存消耗漏洞检测：内存消耗漏洞是由于软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终可能导致内存被耗尽。该类漏洞包括不受控的递归调用[115]、不受控的内存分配[116]，以及内存泄漏[117]。现有的程序分析与测试方法主要面向一般性的内存安全漏洞，以检查程序的执行路径作为主要检测手段。然而内存消耗漏洞这一类漏洞的触发不仅依赖于程序的执行路径，还依赖于执行路径上的内存消耗，故现有的程序分析与测试方法对于内存消耗漏洞的检测存在一定的盲区。于静态分析而言，如何将内存消耗建模成特殊的程序状态，如何判定参与者是否可以通过输入影响消耗的内存量，以及如何平衡精度度和可扩展性都是难以解决的问题。于动态测试而言，如何生成有效的测试用例去触发内存消耗漏洞也是需要研究问题。本文将研究如何利用轻量级的静态分析技术定位影响内存消耗的操作，以及如何收集和利用内存消耗信息去引导动态测试有效地、高效地、自动化地发现内存消耗漏洞。

内存时序漏洞检测：内存时序漏洞是程序在使用内存时，由于违反了内存使用的安全时序规则[118,119]（Temporal Memory Safety）而产生的漏洞。内存使用的安全时序规则即程序对内存的操作顺序应当遵循先申请，再使用，最后释放的规则，并且在释放内存资源后不可再次使用或释放同一内存资源。典型的内存时序漏洞包括释放后使用[120]和双重释放[121]漏洞。现有的程序分析与测试方法在发现内存时序漏洞方面仍面临很大的挑战，主要由于这类漏洞往往涉及一系列内存操作，这些内存操作可能并不都位于同一代码块中，仅当以某种特定的顺序执行这一系列内存操作时才触发错误。于静态分析而言，要精准地发现内存时序漏洞，必须采用上下文敏感、路径敏感的分析，这势必会引入相当大的成本和开销，难以应用

于较大规模的程序。于动态测试而言，如何生成有效的测试用例去触发内存时序漏洞也是需要研究问题。本文将研究如何利用轻量级的静态分析技术识别潜在的违反了内存使用安全时序规则的操作序列，以及如何利用静态分析识别的操作序列去引导动态测试有效地、高效地、自动化地发现内存时序漏洞。

内存并发漏洞检测：内存并发漏洞是指两个或两个以上的线程因线程交错（交互地作用于一个或多个共享内存）而引起程序崩溃、挂起，或产生与串行执行不同的结果，且这样的结果通常是不确定的。随着并发编程越来越流行，现有的程序分析与测试技术不能很好地满足并发程序的需求，如何有效检测内存并发漏洞仍是亟待研究的难题。内存并发漏洞的检测十分具有挑战性，原因在于并发程序涉及多个线程之间复杂的交互执行，其执行路径具有不确定性，且执行交错的空间十分庞大。一个内存并发漏洞的触发，除了需要特定的程序输入，还往往需要按照特定顺序交错执行不同线程的并发敏感操作。于静态分析而言，如何确定可能并发执行的敏感操作，以及如何检查这些敏感操作按照某个特定的顺序交错执行时的行为是非常有挑战的研究问题。于动态测试而言，要发现内存并发漏洞必须让程序执行到触发漏洞的线程交错，一种直观的检测方法是遍历所有的线程交错，但是会带来状态空间爆炸的问题。如何高效探索并发程序所有可能的线程交错是研究的重点和难点。此外，如何在一定程度上确定性地控制线程调度，以让并发程序中的敏感操作按照某个特定的顺序去交错执行也是一个需要研究的问题。本文将研究如何利用轻量级的静态分析技术识别可能并发执行、有潜在风险的敏感操作，以及如何在动态测试中有效控制和遍历这些敏感操作交错执行的顺序。

1.4 本文的主要工作和贡献

针对以上研究问题，本文提出了一套通用可扩展的内存安全漏洞检测框架，即先采用轻量级程序静态分析技术定位潜在的漏洞，后利用动态测试技术指导和限定检测的范围，从而有方向、有目标地发现各类内存安全漏洞，如图 1-1所示。该框架通过把程序分析与测试技术有机结合，取长补短，提高漏洞检测的效果和效率。

本文的主要工作和贡献概括如下。

（1）本文提出了一种融合程序分析与模糊测试的内存消耗漏洞检测方法。该方法旨在对三类内存消耗漏洞进行高效的自动化检测，这三类内存消耗漏洞为：a. 不受控的递归调用；b. 不受控的堆内存分配；c. 内存泄漏。该方法分为程序静态分析和模糊测试两个阶段。在静态分析阶段，该方法确定目标程序中与内存消耗相关的语句和操作，并基于这些语句和操作进行插桩以在运行时收集内存消耗

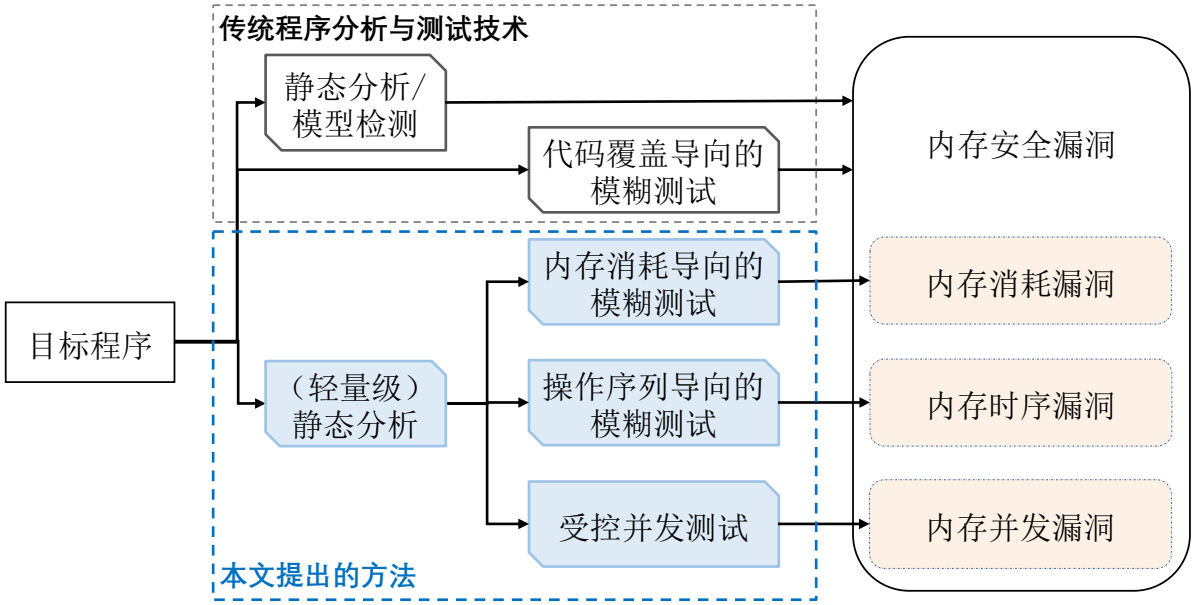


图 1-1　本文技术路线图

信息。在模糊测试阶段，该方法通过内存消耗导向的模糊测试，结合种子动态更新策略，自动化生成能造成大量内存消耗的测试用例，以有效检测内存消耗漏洞。本文实现了该方法的工具原型，并使用了 14 个现实世界中被广泛使用的开源程序对其进行实验评估。实验结果表明，该方法在发现内存消耗漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术，并且能发现其它对比技术都无法发现的内存消耗漏洞。

（2）本文提出了一种融合程序分析与模糊测试的内存时序漏洞检测方法。该方法将内存使用的安全时序规则建模成具有一定类型状态属性的状态机模型，并将内存时序漏洞看作是违反了这一类型状态属性而导致的漏洞。该方法分为程序静态分析和模糊测试两个阶段。在静态分析阶段，该方法基于路径不敏感的可达性分析和指针分析，识别程序中潜在的违反了特定类型状态属性的操作序列。在模糊测试阶段，该方法基于静态分析识别的操作序列，通过操作序列导向的模糊测试，逐步生成能够覆盖完整的操作序列的测试用例，以触发内存时序漏洞。在此过程中，该方法还采用了基于信息流分析的变异优化策略来提升模糊测试的效率。本文实现了该方法的工具原型，并使用了 14 个现实世界中被广泛使用的开源程序对其进行实验评估。实验结果表明，该方法在发现内存时序漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术，并且能发现其它对比技术都无法发现的内存时序漏洞。

（3）本文提出了一种融合程序分析与受控并发测试的内存并发漏洞检测方法。该方法旨在对由线程交错引起的内存安全漏洞进行自动化检测。该方法将并发程

序的执行建模成一个多周期的执行过程，采用了一种基于周期性执行的调度方案，能在一定程度上确定性地控制并发程序的线程交错情况。该方法分为程序静态分析和受控并发测试两个阶段。在静态分析阶段，该方法识别潜在的可能导致并发漏洞的程序语句，包括对共享内存的读写操作和并发同步原语，并将这些程序语句当作调度点。在受控并发测试阶段，该方法采用基于动态切片和调度前缀的调度空间探索方法来针对调度点生成调度决策，基于对线程的控制，该方法测试执行不同的调度决策，能有效探索调度空间并发现内存并发漏洞。本文实现了该方法的工具原型，并在包含46个并发程序的基准数据集上对其进行实验评估。实验结果表明，该方法在检测内存并发漏洞方面要优于当前最先进的受控并发测试技术，并且能发现其它对比技术都无法发现的内存并发漏洞。

（4）本文将提出的内存安全漏洞检测技术实现为一套实际可用的工具集，并将该工具集应用于现实世界的主流开源软件上。本文实现的工具集在主流开源软件中发现了28个内存消耗漏洞、7个内存时序漏洞，5个内存并发漏洞。发现的漏洞也都采取了负责任的披露，其中36个漏洞被国际CVE漏洞数据库收录，这不仅证明了本文方法的有效性和可扩展性，也证明了本文方法的实际应用价值。

本文的创新点总结如下：

• 本文创新性地提出了一套通用可扩展的内存安全漏洞检测框架，不同于传统的程序分析与测试技术，该框架先采用轻量级程序静态分析技术定位潜在的漏洞，后利用自动化测试技术指导和限定检测的范围，从而有方向、有目标地发现各类内存安全漏洞。本文在该框架的理论指导下，针对三类隐藏深、难发现、危害大的内存安全漏洞进一步提出了切实有效的解决方法，

• 本文首次提出增加内存消耗这个新维度来指导模糊测试，使得模糊测试能逐步生成能造成过量内存消耗的测试用例，弥补了当前模糊测试技术在检测内存消耗漏洞方面的盲区。

• 本文首次提出将违反内存使用的安全时序规则的操作序列用于引导模糊测试，提升了模糊测试在发现内存时序漏洞方面的有效性和效率。

• 本文创新性地提出了一种新型的系统性测试方法，使用一种基于周期性执行的调度方案来主动控制线程调度，能根据历史执行信息循序渐进地高效探索调度空间，极大程度地提升了发现并发漏洞的效率。

1.5 论文的组织结构

本文在全面分析国内外内存安全漏洞检测技术研究的基础上，围绕“内存消耗漏洞检测”、“内存时序漏洞检测”，以及“内存并发漏洞检测”这三个关键性技

术问题展开了深入的分析和研究。本文共分为六个章节来介绍相关研究工作，具体研究内容安排如下：

第1章 绪论。首先介绍了本文的研究背景与研究动机；接着分析了内存安全漏洞检测技术研究的国内外研究现状，其中重点从内存消耗漏洞检测、内存时序漏洞检测、内存并发漏洞检测三个方面进行分析，并指出目前研究中存在的主要问题；再者，针对这些问题概述了本文的主要研究内容和贡献；最后对本文各章内容和结构安排进行了说明。

第2章 相关理论及技术基础。详细介绍本文研究工作所涉及的背景知识，包括内存安全漏洞的相关基本概念、程序静态分析和程序动态分析方法、模糊测试的基本框架和算法，以及受控并发测试的相关基础知识。本章内容旨在为后续的研究工作奠定基础。

第3章 融合程序分析与模糊测试的内存消耗漏洞检测技术。首先简要介绍研究背景，以及当前程序分析和模糊测试技术在检测内存消耗漏洞上的局限性；然后详细介绍了本章方法的整体流程、算法设计和关键步骤等；最后设计实验将本章方法与当前最先进的模糊测试技术进行比较，从而验证本章方法的优越性，并对实验结果有效性展开分析与讨论。

第4章 融合程序分析与模糊测试的内存时序漏洞检测技术。首先简要介绍研究背景，以及当前程序分析和模糊测试技术在检测内存时序漏洞上的局限性；然后详细介绍了本章方法的整体流程、算法设计和关键步骤等；最后设计实验将本章方法与当前最先进的模糊测试技术进行比较，从而验证本章方法的优越性，并对实验结果有效性展开分析与讨论。

第5章 融合程序分析与受控并发测试的内存并发漏洞检测技术。首先简要介绍研究背景和受控并发测试背景知识，通过一个示例程序阐述了内存并发漏洞检测的难点和挑战，接着详细介绍了本章方法的整体框架，并基于示例程序介绍了本章的方法流程；然后详细介绍了本章方法的几个关键技术点，包括静态分析和程序插桩、系统性的受控并发测试算法、调度决策生成、周期性执行和反馈信息分析；最后通过大量实验与分析，并将本章方法与其它6种并发测试方法、2种预测分析方法、3种数据竞争检测方法进行对比，从而验证本章方法在发现内存并发漏洞上的有效性和效率。

第6章 总结与展望。总结了本文的主要研究工作，并对未来的研究方向进行展望。

第 2 章　相关理论及技术基础

本章对本文工作涉及的内存安全漏洞、程序分析技术和程序测试技术的主要概念、术语、以及基本思想进行介绍，其内容在各自领域均已存在广泛和深入研究。本章介绍相关背景知识的目的在于：（1）为后续章节提供论述基础，使本文内容更完整；（2）使本文的表述形式（如术语和符号）保持一致。本章仅限于介绍通用背景知识，与特定研究问题相关的概念和技术将在后续章节中详细介绍。

2.1　内存安全漏洞

在所有的安全漏洞中，与内存相关的漏洞产生的危害是巨大的。内存安全漏洞包括很多种，只要和内存相关的漏洞都可以被称为内存安全漏洞。常见的内存安全漏洞包括内存泄漏、释放后使用、双重释放、缓冲区溢出等。因为软件漏洞的特征与形成原因具有一定的复杂性，从单一属性描述软件漏洞几乎是不可能的，现有的研究工作已从多角度对软件漏洞进行描述和分类，例如 PA 计划的分类[44]、COAST 实验室的漏洞的分类[122]、通用缺陷列表 CWE 的分类[123] 等。

本文在研究的过程中采用的就是 CWE 的分类标准，所研究的漏洞类型如表 2-1所示。为了不引起歧义，本文在接下来的内容中，对内存安全漏洞所包含的漏洞类型特指表 2-1中列举的漏洞类型。并且，本文将第 3、4、5章方法针对的漏洞类型分别称作内存消耗漏洞、内存时序漏洞，以及内存并发漏洞。

内存消耗漏洞是指软件没有合理地控制有限内存资源的分配和维护，从而使参与者可以影响消耗的内存量，最终导致可用内存的耗尽。该类漏洞类型包括不受控的递归调用[115]、不受控的内存分配[116]，以及内存泄漏[117]。

表 2-1　本文研究的内存安全漏洞

本文中的分类	漏洞名称	英文名称	相关 CWE 编号
内存消耗漏洞	不受控的递归调用	Uncontrolled Recursion	CWE-400、CWE-674
	不受控的内存分配	Uncontrolled Memory Allocation	CWE-400、CWE-789
	内存泄露	Memory Leak	CWE-400、CVE-401
内存时序漏洞	释放后使用	Use-after-free（UaF）	CWE-416
	双重释放	Double-free（DF）	CWE-415
内存并发漏洞	竞态条件	Race Condition	CWE-362
	并发释放后使用	Concurrency Use-after-free	CWE-362、CWE-416
	并发双重释放	Concurrency Double-free	CWE-362、CWE-415
	并发空指针解引用	Concurrency Null-pointer dereference	CWE-362、CWE-476
	死锁	Deadlock	CWE-833

定义 2.1 (不受控的递归调用). 不受控的递归调用是指程序没有正确地控制发生的递归调用数量，这会导致消耗过多的栈内存空间，最后因栈溢出而导致程序崩溃。

定义 2.2 (不受控的内存分配). 程序基于不可信的大小的值分配内存，但没有确保分配的内存大小在预期限制内，从而允许攻击者分配任意大小的内存，最终导致程序无法正常执行或产生崩溃。

定义 2.3 (内存泄漏). 内存泄漏是指由于编码不当，使得在程序运行过程中一些已分配的堆内存未能释放，导致系统堆内存被不断消耗，最终会引起程序运行速度减慢或计算机系统崩溃等结果。

内存时序漏洞是程序在使用内存时,由于违反了内存使用的安全时序规则[118,119]（Temporal Memory Safety）而产生的漏洞。内存时序的安全使用规则即程序对内存的操作顺序应当遵循先申请，再使用，最后释放的规则，并且在释放内存资源后不可再次使用或释放同一内存资源。在本文中，内存时序漏洞特指释放后使用[120]和双重释放[121]漏洞。

定义 2.4 (释放后使用). 释放后使用（*UaF*）是指在程序运行过程中，某个指针指向的内存被释放后，此时又继续访问已被释放的内存，这会引起程序产生未定义行为或崩溃等结果。

定义 2.5 (双重释放). 双重释放（*DF*）是指在程序运行过程中，对同一块内存区域执行两次内存释放操作，这会引起内存管理数据结构出错，最终会引起程序产生未定义行为或崩溃等结果。

内存并发漏洞是发生在并发程序当中且与内存相关的漏洞，为了不引起歧义，本文将在简要介绍并发程序的相关基础知识后，明确说明本文研究的内存并发漏洞类型。

并发程序（Concurrent Program）与传统的串行程序（Sequential Program）相比，有以下两个特点：

（1）执行结果可能不确定。并发程序动态执行时，会受到操作系统中调度器的支配，调度器的调度是随机的，当它处理并发程序中的事务流时也是不确定的，故即使执行同一个测试输入，多次执行的结果也可能会存在差异。

（2）线程交错空间可能会很庞大。程序执行时有多个线程交错执行，随着线程数量和程序规模的增加，多个线程之间的线程交错情况呈指数级增长。

定义 2.6 (进程). 进程是具有独立功能的程序在某个数据集合上的一次执行过程，是系统进行资源分配和调度的一个独立单位。

定义 2.7 (线程). 线程是进程内的一个执行实体或执行单元，是比进程更小的能独立运行的基本单位。

本文研究的并发程序为单进程多线程的程序，线程间进行信息交换主要通过共享内存机制。

定义 2.8 (共享内存). 共享内存允许两个或多个线程通过读/写同一块内存区域隐式地进行通信。当一个线程改变了这块内存中的内容的时候，这种更改对其它线程来说是可见的。

图 2-1是一个描述了共享内存机制的简单示例图，图中共享变量 *count* 同时允许五个线程对其进行读取/写入操作，因此这五个线程可以通过该共享变量达成线程间通信的目的。但在某一时刻，这五个线程对共享变量 *count* 的访问顺序可能是不确定的。

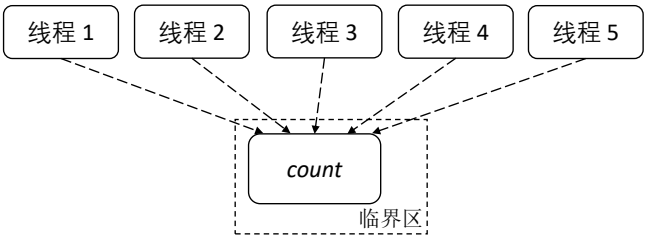


图 2-1　共享内存机制示例图

自 1979 年 Lamport 提出顺序一致性模型后[124]，该模型被广泛应用于研究工作中。后来研究人员放松了对读写访问的约束，提出了更为松散的内存模型[125]，其中最具代表性的为工业界广泛使用的全存储序内存模型（Total Store Order，TSO）[126] 和部分存储序内存模型（Partial Store Order，PSO）[127]。TSO 只需所有的写操作具有全序关系，PSO 甚至都不强制写操作的顺序。本文的所有研究工作都是基于顺序一致性内存模型进行的。

定义 2.9 (顺序一致性内存模型). 顺序一致性内存模型要求在某个时刻每个操作都必须原子地执行，且立即对所有线程可见，此外，每个线程都是严格按照程序顺序执行的方式访问内存。

内存并发漏洞是指两个或两个以上的线程因线程交错（交互地作用于一个或多个共享内存）而引起程序崩溃、挂起，或产生与串行执行不同的结果，且这样的结果通常是不确定的。常见的内存并发漏洞包括竞态条件[128]、由线程交错导致的内存破坏[107]、死锁[129,130] 等。当然，关于内存并发漏洞的分类同样有很多种[37,131,132]，

新的漏洞形式会随着人们对并发漏洞的认识发展而被提出来，例如蔡彦等人[106]将由线程交错导致的内存破坏漏洞进一步分类为并发释放后使用、并发双重释放、并发空指针解引用等。本文在接下来的内容中，内存并发漏洞特指竞态条件、并发释放后使用、并发双重释放、并发空指针解引用和死锁漏洞。

定义 2.10 (数据竞争). 数据竞争（*Data Race*）是指两个或两个以上线程同时访问相同的内存单元，且至少有一个访问操作为写操作，同时这些内存访问操作未使用互斥的同步机制约束。

定义 2.11 (竞态条件漏洞). 竞态条件（*Race Condition*）漏洞是指两个或两个以上线程在读写同一块共享数据时结果依赖于它们执行的相对时间，并且，其内存访问事件的执行时间或顺序影响了程序的正确性。

数据竞争与竞态条件漏洞是两个不同概念，数据竞争是从程序执行的行为上定义的，而竞态条件漏洞是从程序执行产生的结果进行定义的。一方面，只有当数据竞争发生且影响程序正确性的情况才被称为竞态条件漏洞；当数据竞争发生，但是没有对程序正确性造成影响，不能称之为竞态条件漏洞[128]。另一方面，数据竞争这种行为产生的原因是由于内存访问操作不是原子性执行的，可以添加互斥的同步机制约束来消除数据竞争，而竞态条件漏洞产生的原因是由于内存访问事件的执行顺序不符合预期，仅通过添加互斥的同步机制约束不一定能修复竞态条件漏洞。

定义 2.12 (并发释放后使用). 并发释放后使用（*Concurrency UaF*）是指两个或两个以上的线程因线程交错，使得某个指针指向的内存在一个线程中被释放后，此时另一个线程又继续访问已被释放的内存，引起程序产生未定义行为或崩溃等结果，且这样的结果仅在某些线程交错下才会发生。

定义 2.13 (并发双重释放). 并发双重释放（*Concurrency DF*）是指两个或两个以上的线程因线程交错，使得一个线程对一块内存区域执行了一次内存释放操作，而另一个线程又对同一块内存区域再执行了一次内存释放操作，引起程序产生未定义行为或崩溃等结果，且这样的结果仅在某些线程交错下才会发生。

定义 2.14 (并发空指针解引用). 并发空指针解引用（*Concurrency NPD*）是指两个或两个以上的线程因线程交错，使得一个线程将一个指针变量赋值为空值，其后另一个线程立即对该空指针进行解引用，引起程序产生未定义行为或崩溃等结果，且这样的结果仅在某些线程交错下才会发生。

定义 2.15 (死锁). 死锁是指两个或两个以上的线程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称程序产生了死锁。

由于竞争资源产生的死锁称为资源死锁；由于接收不到其他线程发来的消息而陷入无限等待产生的死锁称为通信死锁，在本文中，死锁特指由于竞争内存资源而产生的资源死锁。

与串行程序的内存安全漏洞相比，并发程序的内存安全漏洞[37]难理解、难复现、难调试和难修复。并发漏洞的触发通常涉及多个线程之间复杂的交错执行，因此难以理解。并发程序的线程交错空间十分庞大，且并发漏洞只在特定的线程交错中才会被触发，因此，复现并发漏洞的概率相当小。并发漏洞从引入到暴露需要一定的传播过程[133]。传统的通过迭代运行的方式定位漏洞的调试方法并不适用，增大了并发漏洞的调试难度。调查研究表明，在几种常见的漏洞类型中，并发漏洞是最难正确修复的：大约 39% 的并发漏洞修复是不正确的[134]，并且更为严重的是，对某一漏洞的修复往往会引入新的并发漏洞。

2.2 程序分析技术

程序分析是指对计算机程序进行自动化的处理，以确认或发现其正确性、安全性和性能等方面的性质[19]。以分析过程“是否需要执行程序”为准则，总体上可将程序分析技术分为静态分析技术和动态分析技术两大类。本节旨在对本文后续章节中涉及的相关基础理论知识进行介绍。

2.2.1 静态分析

程序静态分析是指在不运行计算机程序的情况下，通过词法分析、语法分析、语义分析、控制流分析、数据流分析等技术对程序代码进行扫描，验证代码是否满足规范性、安全性等指标的一种代码分析技术。以下进一步介绍静态分析中的一些基本概念，包括控制流分析、数据流分析和敏感性分析。

2.2.1.1 控制流分析

控制流分析是根据程序代码语句结构之间的关系，对程序进行抽象化并确定程序控制结构的程序分析方法。

控制流程图（Control Flow Graph，CFG）是程序的一种基于图的表示形式，是对程序中分支跳转关系的抽象，通过它可以遍历程序执行过程中的所有路径。控制流程图是由程序的基本块（Basic Block），以及基本块和基本块之间的边组成的有向图。一个控制流图可以定义为 $CFG = \langle N, E, entry, exit \rangle$，其中：

• $N = \{b_1, b_2, ..., b_n\}$ 代表程序中的基本块，每个基本块 $b_i \in N$ 是同时满足如下两个条件的最长程序语句序列：(1) 控制流只能从基本块的第一条语句进入；(2) 控制流只能从基本块的最后一条语句出去。由此可知，一个基本块除了最后一个程序语句可为跳转语句外，其它的程序语句均为连续的非跳转语句；并且，对于控制流程图中的每个基本块，其入度和出度至少有一个要大于一。

• $E = \{e_1, e_2, ..., e_n\}$ 代表图中的有向边，每条有向边都表示起始基本块到终止基本块的跳转，对应程序中的一个分支。例如，有向边 $e_i = (b_l, b_m)$ 表示程序可以从该边的起始结点 b_l 的跳转语句跳转到边的终止结点 b_m。

• 为了更方便地对控制流程图进行分析，大多数控制流程图的定义中均要求图中只能有唯一的入口结点 $entry$ 和出口结点 $exit$。

函数调用图（Call Graph，CG）是一个描述程序中各个函数之间的调用关系的有向图。函数调用图中的每一个结点表示一个函数，每一条边表示一个函数调用关系。一个函数调用图可以定义为一个二元组 $CG = \langle F, E \rangle$，其中：

• $F = \{f_1, f_2, ..., f_n\}$ 是一个有穷的结点集合，$f \in F$ 是一个函数定义。

• $E = \{e_1, e_2, ..., e_n\} \subseteq (F \times F)$ 是一个有穷的有向边集合，$(f_l, f_m) \in E$ 表示在函数 f_l 中对函数 f_m 进行了调用。

• 对每个函数 $f \in F$，$caller(f) = \{f_k \mid f_k \in F \wedge (f_k, f) \in E\}$，$callee(f) = \{f_k \mid f_k \in F \wedge (f, f_k) \in E\}$，其中 $caller$ 和 $callee$ 分别表示获取指定函数的调用函数和被调用函数的两个函数。

当前已经有不少控制流程图和函数调用图的自动生成工具，例如本文使用了 LLVM 框架提供的方法生成控制流程图和函数调用图。构造准确的静态函数调用图是不可判定问题，该过程还需要一些辅助支持技术（例如变量的别名分析，C/C++ 的函数指针或虚函数调用分析等），因此静态函数调用图通常只是近似情形。

控制依赖通常用于确定一条程序语句语义的变化是否影响其它程序语句的执行。直观来讲，如果语句 s_1 是一个影响语句 s_2 执行的条件，那么语句 s_2 控制依赖于语句 s_1。例如，位于条件语句两侧分支的语句控制依赖于该条件语句中的谓词。Ferrante 等人[135] 对控制依赖给出了一个基于图论的后必经概念的定义，这被认为是经典的控制依赖的定义。下文在控制流程图和后必经结点概念的基础上，给出控制依赖的定义。

定义 2.16 (后必经结点). 对于控制流程图中的两个连通的基本块结点 b_l 和 b_m，如果每一条从结点 b_l 到出口结点 $exit$（不包括 b_l）的路径均包含 b_m，那么结点 b_m 是结点 b_l 的后必经结点（*Postdominator*）。

后必经是一种传递关系，注意后必经结点的定义中不包括路径的初始结点，所以一个结点不会是它自身的后必经结点。

定义 2.17 (控制依赖). 令 b_l 和 b_m 为控制流程图 CFG 中的两个不同的结点，结点 b_m 是控制依赖于（*Control-dependent on*）结点 b_l 的，当且仅当：

(1) 存在一条从 b_l 到 b_m 的有向路径 p，且 b_t 是 p 中的任意结点（不包括 b_l 和 b_m），b_m 是 b_t 的后必经结点。

(2) b_m 不是 b_l 的后必经结点。

从上面的定义可以看出，如果结点 b_m 控制依赖于结点 b_l，那么 b_l 必须有两个出口。沿着其中的一个出口，总是导致执行 b_m，而另一个出口则导致 b_m 不被执行。

2.2.1.2 数据流分析

数据流分析主要关注程序执行路径上数据流的流向或变量可能的取值，其目的是在程序的一定范围内，确定变量定义和使用之间的关系（def-use 关系）。变量定义指程序中的某条语句对变量进行赋值，除定义之外的其它变量操作称为变量使用。数据依赖表示程序中对某变量进行使用的程序语句对定义该变量的程序语句的依赖，数据依赖的定义如下。

定义 2.18 (数据依赖). 令 s_1 和 s_2 为程序中的两条语句，语句 s_2 关于变量 v 数据依赖于（*Data-dependent on*）语句 s_1，当且仅当：

(1) 变量 v 满足 $v \in DEF(s_1)$，其中 $DEF(s)$ 表示语句 s 定义的变量集合（若变量 v 的值在语句 s 中被赋值，则称 s 定义了 v）。

(2) 变量 v 满足 $v \in USE(s_2)$，其中 $USE(s)$ 表示语句 s 使用的变量集合（若变量 v 的值在语句 s 中被读取或解引用，则称 s 使用了 v）。

(3) 从 s_1 到 s_2 可达，且从 s_1 到 s_2 的执行路径中没有其它语句改变 v 的值。

程序依赖图是以程序的控制流程图为基础，去掉控制流程图中的边，以表达式语句为结点，图中的边由结点之间的控制依赖关系和数据依赖关系确定。

定义 2.19 (程序依赖图). 程序依赖图是一种描述程序中语句间依赖关系的有向图，可以用一个三元组 $PDG = \langle N, E_{dd}, E_{cd} \rangle$ 表示。其中，N 表示 *PDG* 中所有结点的集合，每个结点表示一条语句或一个语句块。E_{dd} 表示 *PDG* 中所有数据依赖边的集合，每一条边表示相邻两个结点的数据依赖关系。E_{cd} 表示 *PDG* 中所有控制依赖边的集合，每一条边表示相邻两个结点的控制依赖关系。

程序依赖图显式描述了程序语句之间的控制依赖和数据依赖关系，同时隐去对程序执行不产生影响的部分，具有更强的表达能力，对提高目标程序的认知有着重要意义[136]。

2.2.1.3 敏感性分析

在静态分析中，经常会涉及一些敏感性分析的相关概念，主要包括流敏感（Flow-sensitive），路径敏感（Path-sensitive）和上下文敏感（Context-sensitive）。

流敏感的分析：在分析的过程中考虑程序中语句的顺序和位置，比如在别名分析中，分析算法需要知道两个指针从哪个语句开始是别名。与流敏感的分析相对应的是流不敏感的分析，分析算法只需要判断两个指针是否是别名，而不需要知道这两个指针从哪个语句开始是别名。因此流敏感的分析相比流不敏感的分析更加精确。

路径敏感的分析：在分析过程中同时考虑程序控制流程图中每一个分支上的条件信息，并记录两个分支路径的不同程序状态。相应的，路径不敏感的分析不考虑分支之间的区别，通常在控制流程图的分支汇合处对数据流状态信息进行合并。例如，给定程序语句 $y = x < 0\,?\,-1:1;$，在路径不敏感的分析中，分析完该语句后，得到的信息为 $\{\{\} \vdash y \in \{-1,1\}\}$，而路径敏感的分析则保留了 $x < 0$ 的约束，即 $\{\{x < 0\} \vdash y \in \{-1\}, \{x \not< 0\} \vdash y \in \{1\}\}$。因此路径敏感的分析相比路径不流敏感的分析更为精确。

上下文敏感的分析：在过程间分析中，分析函数调用时需要考虑调用点的程序信息。一个函数可能会被多个函数调用，那么在不同的函数调用它的时候，在传给它的实际参数或当时的全局变量不同的情况下，可能有不同的行为。例如，在调用函数 $f(a,b)$ 时，a 和 b 的值是相等的，上下文敏感的分析会将这个约束带到被调用函数的分析，而上下文不敏感的分析则忽略 a 和 b 相等的信息。很显然，上下文敏感的分析相对于上下文不敏感的分析更为精确。

在实际应用当中，根据不同的需求可以采用不同的敏感性分析，精确的分析在时间和空间开销上会有更大的损耗，不精确的分析则可能会带来精度的损失（如漏报和误报）。对于精确的静态分析，其目的是尽可能地以低误报的方式找到更多的程序错误，所以一般会采用流敏感、路径敏感和上下文敏感的分析方法。其中路径敏感性对分析时间、分析所消耗的内存、分析效果具有较大影响。对于编译时分析和编译时插桩技术而言，分析过程一般与编译过程相结合，因此引入的时间开销不能太高，所以通常会采用路径不敏感的分析方法。

本文使用的静态分析方法都是轻量级的分析，不以直接发现内存安全漏洞为

目的，而是用于提供信息以指导动态测试。例如第3章只需要识别哪些操作是与内存消耗有关的操作并对不同的操作插入不同的“探针”，故使用的是流不敏感、上下文不敏感和路径不敏感的分析；第4章需要识别潜在的内存时序漏洞，需要更加精确的分析，故使用的是流敏感、上下文敏感和路径不敏感的分析；第5章需要识别出可能导致并发漏洞的调度点，且需要进一步考虑多线程之间的数据流，因而使用的是流敏感、上下文敏感和路径不敏感的分析。

2.2.2 动态分析

动态分析以指定的测试用例运行给定的程序，并确认或发现程序运行时的性质。与静态分析不同，动态分析需要一个真实的或是模拟的运行环境。动态分析需要监控和分析程序运行状态信息，这些信息可能是运行时的提示信息和日志信息，或者是程序运行过程中或结束时的输出结果。分析工具通过得到的信息与事先预定的信息进行对比，可以清晰地检测到程序的异常。动态分析在运行时针对确定的可执行路径进行分析，能更好地处理编程语言中的动态属性，例如指针变量和别名的准确信息，因而是一种精确的检测方法。

2.2.2.1 程序插桩

程序插桩，最早是由J.C. Huang[137]提出的，它在确保被测程序原有逻辑完整性的情况下往程序中插入一些探针（又称为“探测仪”，本质上就是进行信息采集的代码段，可以是赋值语句或采集覆盖信息的函数调用），当探针被执行后，能收集到程序执行时特征数据，通过对这些特征数据的分析，可以获得程序的控制流和数据流信息，进而得到覆盖率等动态信息，从而实现测试目的[138,139]。

程序插桩技术主要分为动态二进制插桩与静态源码插桩，这两种技术已经在程序分析与漏洞挖掘等领域广泛应用。动态二进制插桩可以直接应用于二进制可执行文件，这样就可以在不需要源码插桩的情况下完成插桩。动态二进制插桩的代表工作有Valgrind[93]、DynamoRIO，以及Intel公司开发的Pin[140]。静态源码插桩又叫编译时插桩，它引入的运行时开销更小，还可以对源代码做完整的词法分析和语法分析，故对源代码的插桩能够确保达到很高准确性和针对性。典型的地址消毒剂ASAN[27]就是由一个编译时插桩模块和一个运行时的库组成，支持多种内存错误的检测且执行效率较高（执行时间的延缓倍率低）。在本文后续内容提到的程序插桩均指源码插桩。

大多数针对内存安全漏洞的动态分析（例如ASAN、MSAN、Valgrind等）都是基于程序插桩技术实现的，它们针对编译器级别的对内存的访问操作（例如LLVM

指令集中的 store、load、alloca 等指令），在这些内存访问操作之前插入一段代码将相应的内存对象是否可用的信息映射到影子内存。具体来说，这些动态检测方法将内存对象的可用/不可用状态记录到影子内存中的对应位置。每当程序通过指针访问内存对象时，这些动态检测方法会查询影子内存中记录的内存对象的状态，以此来判断程序所访问的内存对象是否有效。若有效，这些动态检测方法认为程序正在执行合法的内存访问，然后程序继续运行。否则，这些动态检测方法会认为在程序中检测到了内存安全漏洞，然后终止程序执行，并报告内存安全漏洞诊断信息。

2.2.2.2 信息流分析

信息流是指信息的流动和传播，它表示信息之间的一种交互关系。一般而言，如果信息 A 对信息 B 的内容产生影响，则信息从 A 流向 B。所谓信息流分析[141,142]，即对程序中变量值的获取和传播数据流进行分析，它能准确了解变量的特征，也能清楚地知道输入和变量之间的关系。

关于信息流的严格定义是基于熵的形式化定义：程序在某状态 s 下执行某一动作序列 o 后到达新的状态 s'，设 a_s 代表系统在状态 s 下变量 a 的取值，b_s 和 $b_{s'}$ 分别代表变量 b 在状态 s 和 s' 的取值，$H(a_s|b_s)$ 表示条件熵，如果满足 $H(a_s|b_{s'}) < H(a_s|b_s)$，那么称动作序列 o 使信息从变量 a 传递到了变量 b。其中熵的计算方式如下：

$$H(a_s) = -\sum_{x \in a_s} p(x) \log_2 p(x) \tag{2-1}$$

相应的，条件熵 $H(a_s|b_s)$ 可以反映在已知 b_s 取值的概率分布的条件下 $a_{s'}$ 的不确定性有多大，其计算方式如下：

$$H(a_s|b_s) = -\sum_{x \in a_s} \sum_{y \in b_s} p(x,y) \log_2 p(y|x) \tag{2-2}$$

本文以如下程序语句为例来解释如何使用信息流分析来分析变量之间的数据流向关系：

$$if(a)\ \ b = 0;\ else\ \ b = 1;$$

假设 a 取值为 1 和 0 的概率是相等的，若用状态 s 表示该程序语句执行之前的程序状态，状态 s' 表示该程序语句执行之后的程序状态，则有 $H(a_s) = 1$，$H(a_s|b_{s'}) = 0$。由于 $H(a_s|b_{s'}) = 0 < H(a_s|b_s) = H(a_s) = 1$，所以信息从 a 流向了 b。

信息流分析技术同样可分为静态方法和动态方法两类，本文采用的信息流分析技术为动态信息流分析。

2.3 程序测试技术

程序测试是以发现程序中的错误为目的，它需要人们提供输入数据（即测试用例），以便运行程序，观察其输出结果。测试技术多种多样，在实际工程中广泛应用，本节将对模糊测试和受控并发测试技术展开介绍。

2.3.1 模糊测试

模糊测试（Fuzz Testing，也称 Fuzzing）的概念最早是由 Miller 等人[62] 在 1989 年提出的。模糊测试通过随机生成输入来重复执行被测程序，并监控被测程序的执行是否触发异常，根据触发的异常和导致异常的输入定位漏洞。与其它技术相比，模糊测试技术易于部署，具有良好的可扩展性和适用性，并且很容易应用到大规模应用程序上。

依据分类的侧重点，模糊测试技术可以从不同的角度进行分类[143]。根据测试用例的生成方式可以分为基于生成的模糊测试和基于变异的模糊测试；根据被测程序的内部可见性可以分为黑盒模糊测试、白盒模糊测试和灰盒模糊测试；按照探索时的指导策略可以分为覆盖导向的模糊测试、定向模糊测试等。不同的分类方法之间存在交叉重叠，比如模糊测试工具 AFL[30] 是一个覆盖导向的、基于变异的灰盒模糊测试工具。

覆盖导向的灰盒模糊测试技术是当前最有效、最先进的漏洞检测技术之一。它相比黑盒模糊测试的效率更高，能覆盖更多的代码；相比白盒模糊测试的性能开销更小，在同样的时间内能执行（百倍甚至千倍）的输入。覆盖导向的灰盒模糊测试采用轻量级的程序分析方法来获取粗粒度的程序执行覆盖率信息，并使用代码覆盖率作为程序执行反馈信息来指导生成有效的输入；有效的输入即能执行未覆盖代码的输入，基于有效输入进行变异，新生成的输入更容易遍历到程序尽可能多的代码，也更有可能发现这些代码中的潜在漏洞。AFL 自 2014 年由 Google 安全工程师 lcanmtuf 开发，在工业界持续的成功经验证明了该方法的有效性。当前大多数的模糊测试的研究工作都是在 AFL 的基础上发展的，在本节接下来的内容中，以 AFL 为例介绍覆盖导向的灰盒模糊测试技术的基本原理和工作流程。

图 2-2描述了 AFL 的整体工作流程。AFL 持续自动化地生成测试用例对目标程序进行测试，该过程一旦启动运行，即可在预算的时间范围内自动化地检测漏洞而无需人工干预。给定少量的程序输入作为初始的种子池，AFL 每次从种子池中选择一个种子，使用不同的变异算子对该种子进行变异，自动化地生成大量新的测试用例，这些测试用例都会用作目标程序的输入来对目标程序进行测试。待

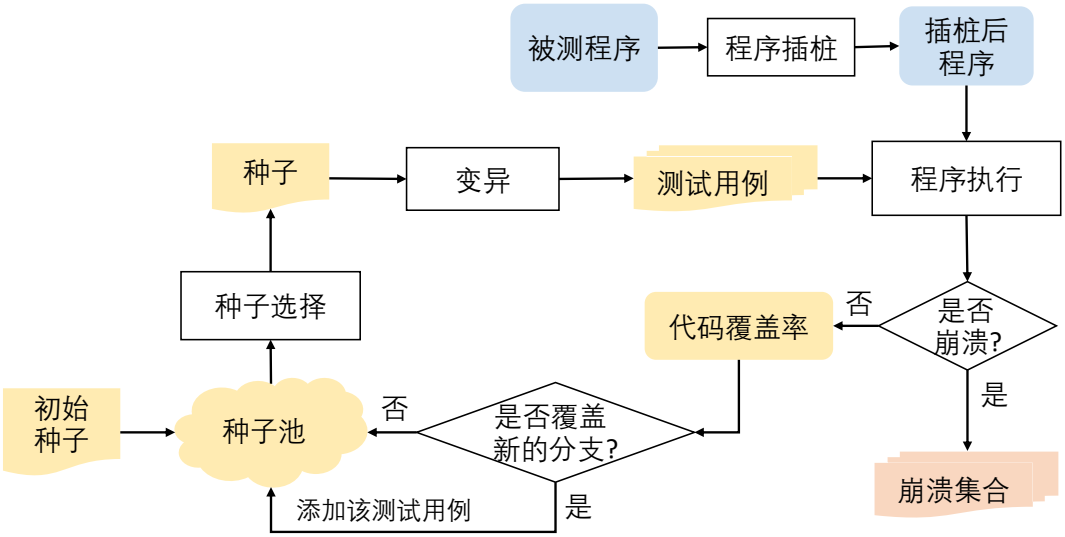


图 2-2　AFL 的整体流程

目标程序执行后，AFL 会收集此次执行的覆盖率信息，并用于指导种子的保留和选择。如果生成的测试用例能覆盖新的分支，那么该测试用例会被当作一个高质量的种子加入到种子池中。AFL 不断重复此过程，直到达到了预设的资源限制。在期间，目标程序动态执行触发的崩溃都会作为发现的程序漏洞报告出来，以待开发人员和测试人员的分析。接下来，本文进一步介绍此过程中的一些重要概念和关键步骤。

首先，被测程序和种子在模糊测试中必不可少。模糊测试是围绕着被测程序进行的，发现被测程序中的异常或漏洞是模糊测试的目标。AFL 在对被测程序进行模糊测试之前，需要通过程序插桩的方式在被测程序的每个基本块头部插入特殊的探针代码，在程序执行时起到收集和记录代码覆盖信息的作用，代码覆盖率信息在模糊测试过程中起着关键的指导作用。种子和测试用例都是指程序的输入数据，种子是生成测试用例的基础，测试用例来自于对种子的变异。初始种子的选择对后续测试的效果影响巨大，高质量的初始种子能够覆盖尽可能多的被测程序的代码区域，并且尽可能少的与其它种子重复覆盖同一代码区域，前者是为了尽可能提升代码覆盖率，后者是为了节约宝贵的计算资源和时间开销，避免资源浪费在重复的种子上。

其次，变异是 AFL 自动生成测试用例的重要手段。AFL 从种子池中选择一个种子对其进行变异来自动化地生成新的测试用例，通过对种子进行位翻转、字节翻转、算数加减、关键值替换、拼接等操作改变种子的部分数据，以生成大量不同的新测试用例。Böhme 等人[63] 提出了能量调度的概念，能量指一个种子被选择后进行变异的次数，而能量调度即根据一个特定的算法对能量进行控制。AFL 会

根据种子的创建时间、执行速度、覆盖的分支等因素动态地调整能量的分配，以将更多的变异机会留给质量更高的种子。

再者，代码覆盖率是决定种子保留和选择的指标。AFL 将生成的测试用例输入到被测程序当中，在动态执行被测程序之后，收集本次执行的覆盖率信息，如果生成的测试用例能覆盖新的代码分支，那么该测试用例会被当作一个高质量的种子加入到种子池中；否则该测试用例被丢弃。高质量的种子即能执行程序还未执行过的代码，后续阶段进一步基于有趣种子进行编译，生成的新的测试用例更容易覆盖到更多未执行过的程序代码，也更有可能发现代码中的潜在漏洞。AFL 利用代码覆盖率指导测试用例生成不断朝着更高的代码覆盖率迈进，故 AFL 被称为一种代码覆盖导向的模糊测试技术。

最后，漏洞的发现需要通过监视崩溃来达成。AFL 在程序动态执行的过程中监视程序是否因内存错误、断言失败等原因发生崩溃，触发崩溃的测试用例会被保存下来，用于后续复现漏洞和确认漏洞。AFL 还会根据代码覆盖情况对发生的崩溃进行分类，若两次崩溃发生时的代码覆盖情况完全一致，则 AFL 将这两次崩溃看作是同一类崩溃。

2.3.2 受控并发测试

受控并发测试技术[144]（Controlled Concurrency Testing）以给定的测试用例对并发程序进行重复性的测试，在程序运行时主动使用线程调度控制手段，迫使程序按照预期的执行交错顺序去执行程序，并监视其执行是否发现异常。受控并发测试技术与传统的压力测试不同，压力测试是模拟实际应用程序的软硬件环境及用户使用过程的系统负荷，在超大负荷的情况下长时间运行被测程序，在极限情况下一些不确定性因素可能会出现平时极少出现的情况（例如并发程序中稀有的线程交错情况），从而可能暴露一些因不确定因素引起的软件漏洞。虽然压力测试也是发现并发漏洞的一种常用方法，但这种方法往往会执行很多无谓的测试，难以覆盖各种线程交错执行的可能性。

受控并发测试技术可以分为非系统性的测试和系统性的测试两类。非系统性的测试通常采用随机调度的方案对程序进行重复性测试[113,145]，从概率上可以保证每一个并发漏洞都有一定的概率被检测到，但是这类方法也需要非常多的测试次数，因为其概率保证是非常小的。系统性的测试早期是指通过控制线程调度，动态执行程序遍历所有可能的线程交错执行序列。理论上，由于系统性的测试完全地探索了调度空间，故不会漏报任何并发漏洞，但该方法对于较大规模的并发程序完全不可行。为了改进测试效率，近年来研究人员提出了使用偏序规约[146]

（Partial Order Reduction）来减少对冗余的线程交错情况的测试，或者使用限定调度技术[147]（Schedule Bounding）在限制引入的调度操作次数的情况下进行测试。如今，系统性的测试通常指使用了限定调度技术的系统性测试技术[114,148–150]。

偏序规约方法是缓解并发程序分析中的状态爆炸问题的一种约简技术，该方法忽略了分析过程中产生的一些无关紧要的线程交错，仅仅需要探索所有可能的线程交错中的一个子集。大多数情况下，没有必要去探索所有可能的线程交错，如果一组连续的指令只对某个线程有作用但对其它线程没有影响的的话，那么可以通过将这些指令序列看作是一个单独的指令，则需要测试的线程交错情况可以明显减少。偏序规约方法通常可以约减 70% 左右的线程交错情况[151]。传统的偏序规约方法的分析代价很大[152–154]，一种在受控并发测试中简单易行的方法是忽略对局部变量的执行交错情况的探索，仅关注对共享数据的读写操作和并发同步原语之间的执行交错情况。因为每个线程中的局部变量对其它线程来说都是不可见的，如果一组连续的指令都是对局部变量的操作，那么这些指令之间没有必要引入多余的线程切换。

限定调度技术限定了引入的调度操作的次数，在只允许进行若干次调度操作的情况下对并发程序进行测试，其每次产生的线程交错也只会在特定的线程切换次数范围内。要完全探索一个并发程序的所有可能的线程交错情况是NP难的，现有研究工作表明现实世界中并发漏洞的漏洞深度都较小[37,112]，因此大多数受控并发测试的研究工作都采取限定调度技术，旨在尽可能发现浅层的并发漏洞。接下来，本文引入漏洞深度的定义，并通过两个简单例子进一步解释漏洞深度的概念。

定义 2.20 (漏洞深度). 一个并发漏洞的深度指的是触发该漏洞需要的最少线程切换次数。

图 2-3描述了漏洞深度分别为 1 和 2 的两种并发空指针解引用漏洞示例。图 2-3(a) 中的并发程序中包含一个漏洞深度为 1 的并发空指针解引用漏洞。由于两个

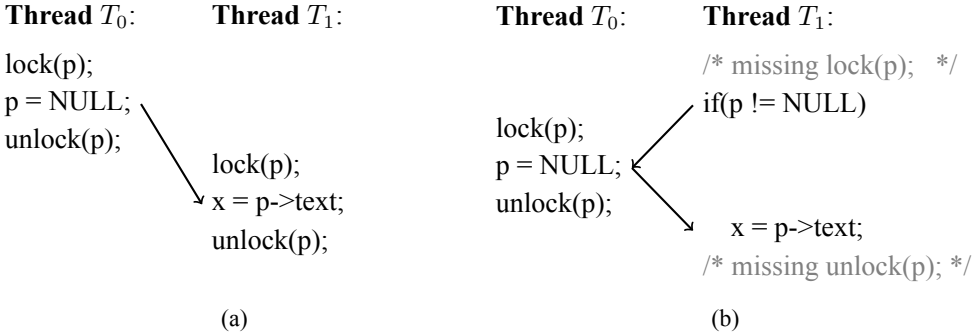


图 2-3　并发空指针解引用漏洞示例

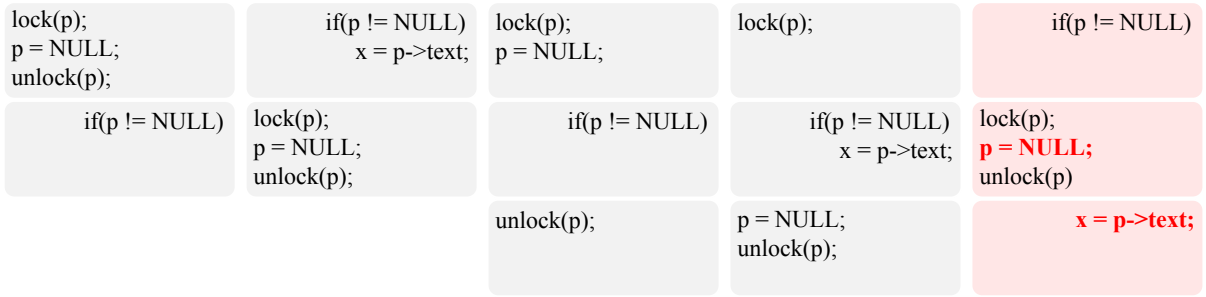


图 2-4　在引入的线程切换不超过 2 次的情况下，对图 2-3(b) 的示例程序做系统性的测试

线程都被互斥锁保护，故不存在数据竞争问题，然而当线程 T_0 先执行，随后线程 T_1 再执行，x = p->next” 语句对一个空指针进行解引用的时候就触发了空指针解引用漏洞。由于该漏洞的触发仅需要 1 次线程切换，故该漏洞的深度为 1。图 2-3(b) 中的并发程序中包含一个漏洞深度为 2 的并发空指针解引用漏洞。由于线程 T_1 中的两个语句没有用与线程 T_0 相同的互斥锁保护起来，线程 T_0 的 “p = NULL;” 语句可能在线程 T_1 的 “if(p != NULL)” 语句执行后立即执行，随后线程 T_0 再执行 “x = p->next” 语句对指针 p 进行解引用的时候就触发了空指针解引用漏洞。由于该漏洞的触发至少需要 2 次线程切换，故该漏洞的漏洞深度为 2。

直观地说，一般情况下漏洞深度较小的漏洞较容易被发现，而漏洞深度越大的漏洞越难被发现，图 2-3中描述的两个漏洞的深度都不大，通常可以通过限定调度技术较快地发现这两个漏洞。例如，对图 2-3(b) 中的程序检测其深度不超过 2 的并发漏洞所测试的线程交错情况如图 2-4所示，一共有 5 种可能的线程交错情况，其中 2-4(e) 描述了触发漏洞时的线程交错情况。尽管限定调度技术能有效发现浅层的并发漏洞，但并发漏洞的检测难度还取决于并发程序的规模和路径的复杂性，当并发程序的规模较大或路径较复杂时，即便是浅层的漏洞也难以触发，故仍然需要有效且实用的并发漏洞检测技术。

2.4　本章小结

本章围绕本文方法涉及的内存安全漏洞、程序分析和测试相关基础知识进行介绍，为后续的研究工作奠定基础。

第 3 章　融合程序分析与模糊测试的内存消耗漏洞检测技术

3.1　引言

3.1.1　研究背景

计算机程序的运行需要占用一定的内存空间，而计算机系统中的内存资源又是十分有限的。因此，程序运行时内存资源的消耗是衡量一个程序性能优劣的重要指标。同一个程序运行时所需的内存空间不是一成不变的，和程序的输入有着很大关系，输入的数据不一样，需要消耗的内存空间也是不同的。如果软件在设计与实现上没有正确控制有限的内存资源的分配和维护，就可能会发生资源耗尽、系统崩溃等非预期的行为，甚至会成为安全攸关的漏洞。例如，如果递归函数的终止条件没有被正确地设计与实现，则递归函数有可能无限调用自身，最终栈内存被耗尽，导致栈溢出崩溃。攻击者可能利用此漏洞精心设计一个能触发栈溢出崩溃的输入来发起拒绝服务（DoS）攻击[155,156]。在现实世界的软件中，已经发现了许多由于未正确控制有限内存资源的分配和维护而引发的安全漏洞[157–160]。

当前针对内存消耗漏洞的自动化检测技术，通常采用基于特定领域或特定实现的启发式规则[73,83,161–163]，例如，Radmin[83] 基于目标程序的动态执行轨迹对目标程序的内存消耗进行建模，构建并执行多个概率有限自动机来检测内存资源耗尽漏洞。一方面它的有效性在很大程度上取决于启发式规则的完整性，另一方面，创建和维护此类规则需要大量的人工工作和专业知识。灰盒模糊测试技术[29,164] 也是一种通用的内存安全漏洞的自动化检测技术，其具有较高的自动化程度和可扩展性，已经在工业界被广泛使用。然而，尽管现有的灰盒模糊技术对于检测一些内存破坏漏洞有着较好的效果，但仍不能有效检测内存消耗漏洞。这是由于当前灰盒模糊测试技术主要是基于分支覆盖率信息反馈引导的，以探索未覆盖的代码分支和路径为目的，而内存消耗漏洞的触发通常不仅取决于程序路径，还取决于该路径上变量的数值（即内存消耗量）。

3.1.2　现存问题

本节通过两个现实世界中的真实漏洞案例来说明当前主流的程序分析技术和模糊测试技术在检测内存消耗漏洞方面的局限性，并引出本章的主要工作。

图 3-1中的代码片段是漏洞 CVE-2018-17985 的简化版本。该漏洞来自 GNU 开源工具集 *Binutils v2.31* 中的 *c++filt* 程序，也是本章在实验过程中新发现并报告的未知漏洞。如图 3-1所示，cplus_demangle_type 函数是一个递归函数，当输入中包含

‘P’字符时，该函数会递归调用自身（第 12 行）。然而，该函数对于递归终止条件并未考虑周全，使得其可能的递归深度未能得到正确控制。具体来说，该函数的递归深度取决于输入字符串中‘P’字符的数量，如果输入字符串包括足够多的字符‘P’，那么该函数执行时会不断地递归调用自身，使得递归调用深度足够大，导致栈溢出崩溃。现有的主流静态分析/扫描工具（例如 Infer[24]、Cppcheck[48]）不具备检测不受控的递归调用漏洞的能力，故未报告该漏洞；而模型检测工具（例如 CBMC[165]）难以在千万行代码规模的程序上部署和应用；动态检测工具（例如 ASAN[27]）仅在栈溢出崩溃发生时才报告漏洞，但关键在于要有特定的测试用例；覆盖导向的灰盒模糊测试工具（例如 AFL[30]、AFLFast[63]）可以对已有测试用例做变异，自动生成大量的测试用例来测试目标程序，但要触发栈溢出崩溃，要求生成的测试用例中包含大量‘P’字符，以让递归函数不断调用自身，这对于覆盖导向的灰盒模糊测试技术来说是非常困难的。

```
1  struct demangle_component *
2  cplus_demangle_type (struct d_info *di) {
3
4    // "peek" is a single character extracted from the input directly
5    char peek = d_peek_char (di);
6
7    switch (peek){
8      ...
9      case 'P':
10       ret = d_make_comp (di,
11         DEMANGLE_COMPONENT_POINTER,
12         cplus_demangle_type (di), NULL);
13       break;
14     case 'C':
15     ...
16   }
17   ...
18 }
```

图 3-1　一个从 CVE-2018-17985 漏洞简化而来的启发性示例（来自 *Binutils v2.31*）

当前基于覆盖引导的灰盒模糊测试技术利用程序控制流程图（CFG）的边覆盖信息来引导模糊测试选择和保留测试用例，它对栈内存消耗和递归深度的变化是不敏感的。以 AFL 采用的模糊测试技术为例，给定一个目标程序，AFL 在为其生成一个测试用例并动态执行后，会收集此次执行覆盖了哪些 CFG 边的信息，包括每条 CFG 边被执行到的次数，但其统计方式较为粗略。具体来说，AFL 将每个 CFG 边的执行次数都用 1 个字节来储存，并采用了循环桶（Loop Bucket）的概念把这个执行次数简单归类到固定的数值上，即 0、1、2、4...64、128。这样的简单归类在某种程度上忽略了循环次数上的微小区别，循环次数接近的两次执行可能被归类为相同的执行情况。例如，如果执行某个测试用例时，分支 A 被执行了 63

次；执行另一个测试用例时，分支 A 被执行了 64 次，那么 AFL 就会认为这两次执行时 A 分支的边覆盖了 64 次，在 A 分支上的覆盖度是相同的。根据循环桶的做法，AFL 无法判断由递归深度造成的细粒度变化，特别是当递归深度大于 128 时，AFL 都认为相关 CFG 边的覆盖度是相同的，因此 AFL 的种子池中无法保留能造成较大递归量的测试用例，难以变异出触发栈溢出崩溃的测试用例（通常需要递归深度达到千万次）。若灰盒模糊测试技术能准确地捕获目标程序运行时的递归调用深度或栈内存消耗信息，并引导测试用例的生成变异向较大的栈内存消耗方向发展，则有望检测出不受控制的递归调用漏洞。

图 3-2中的代码片段是漏洞 CVE-2018-4848 的简化版本。该漏洞是一个不受控的内存分配漏洞，来自开源软件 *Exiv2 v0.26*。DataBuf 是一个包含字符数组的类，它的主要用途是作为需要临时数据缓冲区的函数中的堆栈变量，其构造函数接收一个长整型的变量，并根据变量值的大小分配堆内存空间。当 *Exiv2* 程序调用 Jp2Image::readMetadata() 函数解析 subBox 变量时，将从用户的输入中提取输入的长度并存入 subBox.length 变量中（第 11-12 行）。然后，第 13 行基于 subBox.length 变量的值创建一个 DataBuf 的类对象，该值将作为第 4 行内存分配操作的参数。需要注意的是，程序在分配内存之前并未检查申请的内存大小，允许程序根据用户输入的值分配任意大小的内存空间。因此，攻击者可以精心设计程序输入，使得 subBox.length 的值足够大，以耗费软件系统的堆内存空间。过多的内存分配会因为可用内存数量的减少从而降低计算机的性能，在更糟糕的情况下，过多的可用内存被分配掉可能会导致部分设备停止正常工作，或者应用程序崩溃。事实上，当该 C++ 程序使用 new 操作申请内存时，如果没有足够的内存空间可供分配创建一个 DataBuf 对象，会立即抛出 *std::bad_alloc* 内存分配失败的异常[1]，而程序又未正确处理该异常，使得程序直接崩溃。现有的主流静态分析工具（例如 Infer[24]、Cppcheck[48]）和模型检测工具（例如 CBMC[165]）仅考虑了诸如 *malloc* 函数的返回值是否有做判空检查这种简单情况，并不检测不受控的内存分配漏洞；动态检测工具（例如 ASAN[27]）可以报告程序运行时是否有分配大内存的行为，但需要提供相应的测试用例；灰盒模糊测试工具（例如 AFL[30]、AFLFast[63]）可以对已有的测试用例做变异，自动生成大量新的测试用例来测试目标程序，但要检测出不受控的内存分配漏洞，还要求生成的测试用例能造成过量的内存分配，这对于覆盖导向的灰盒模糊测试技术来说是非常困难的。

另一方面，假设图 3-2的示例程序在第 4 行申请分配的内存（new 操作）在使

[1] *std::bad_alloc* 是 C++ 内存分配函数作为异常抛出的对象类型，以报告存储分配失败；类似于 C 程序内存分配失败时，*malloc* 函数返回空值。

用完之后长时间未释放这块内存（delete 操作），而系统也不能再次将它分配给其它内存申请，那就存在一个可利用的内存泄漏漏洞。从用户使用程序的角度来看，触发一次内存泄漏可能并不会产生危害。但从软件系统的角度来看，内存泄漏的堆积，可能会最终消耗尽系统所有的内存。攻击者可能持续提供能造成内存泄漏的程序输入，或精心设计一个程序输入使得单次运行程序就造成大量的内存泄漏，以此耗尽软件系统的可用内存空间。因此，尽早识别这一类内存泄漏问题对于软件系统的安全性也是至关重要。

```
1  class EXIV2API DataBuf {
2  public:
3    // Constructor with an initial buffer size
4    explicit DataBuf(long size): pData(new byte[size]), size(size) {}
5    ...
6    byte* pData; // Pointer to the buffer
7    size_t size; // The current size of the buffer
8  };
9
10 void Jp2Image::readMetadata() {
11   while (io_->read((byte*)&subBox, sizeof(subBox)) ==
     ↪  sizeof(subBox) && subBox.length ) {
12     subBox.length = getLong((byte*)&subBox.length, bigEndian);
13     DataBuf data(subBox.length); // Allocation without checking
14     ...
15     io_->seek(position - sizeof(box) + box.length, BasicIo::beg);
16   }
17 }
```

图 3-2　一个从 CVE-2018-4868 漏洞简化而来的启发性示例（来自 *Exiv2 v0.26*）

当前基于覆盖引导的灰盒模糊测试技术对于内存分配的值大小是不敏感的，它们主要基于程序控制流程图（CFG）的边覆盖信息来引导模糊测试选择和保留测试用例，虽然较容易生成一些测试用例同时覆盖到图 3-2的第 13 行和第 4 行，但难以生成使得 subBox.length 值更大的测试用例，也难以生成能使得第 4 行内存分配失败的测试用例。以最经典的覆盖导向的模糊测试技术 AFL 为例，假设 AFL 的种子池中包含一个种子用例 a，种子用例 a 可让程序的执行路径同时经过第 11-15 行，并且使得第 12 行定义的 subBox.length 的值为 100。当 AFL 对种子用例 a 进行了一些变异操作后，AFL 可能会生成另一个测试用例 b，测试用例 b 不仅让程序的执行路径同时经过第 11-15 行，还使得第 12 行定义的 subBox.length 的值达到 10000。显然，测试用例 b 相比 a 触发了更大的内存分配，如果将测试用例 b 保留作为高质量的种子用例，基于 b 做变异操作生成的用例相比 a 更有可能造成较大的内存分配，也更容易耗尽堆内存。然而，由于测试用例 b 相较于 a 来说并未覆盖新的分支，AFL 不会将测试用例 b 保留作高质量的种子，而是会将其丢弃。因此，AFL 错失了一个更易发现不受控制的内存分配漏洞的机会。若基于灰盒模糊测试技术在生成

或选择测试用例时不仅能要求同时覆盖第11-15行，还能要求传递给subBox.length变量的值尽可能更大，使得第4行用new申请的内存空间超出可用内存，则有望快速检测出不受控制的内存分配漏洞，也有助于识别由不受控的内存分配而引起的内存泄漏问题。

3.1.3 本章主要工作

针对上述问题，本章提出了一种融合程序分析和模糊测试的内存消耗漏洞检测方法MemLock，该方法可以在不需要特定领域知识的情况下自动化检测不受控的递归调用、不受控的内存分配和内存泄漏这三类内存消耗漏洞。MemLock首先对程序做轻量级的静态分析，确定与内存消耗相关的语句和操作，并基于此进行程序插桩，以在程序运行时收集内存消耗信息。然后，MemLock通过内存消耗导向的模糊测试，引导测试用例的自动生成朝着能造成大内存消耗的的方向迈进。同时，MemLock采用了一种新颖的种子动态更新策略，能为每条程序路径都保留一个能造成最大内存消耗的种子，使得对种子的选择和变异更加有效。最终，MemLock通过有方向、有目的、自动化地生成能造成大量内存消耗的测试用例，有效地发现内存消耗漏洞。本章实现了MemLock的工具原型，并使用现实世界中被广泛使用的开源程序对其进行实验评估。实验结果表明，MemLock在发现内存消耗漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术。MemLock相比AFL、AFLfast、PerfFuzz、FairFuzz、Angora、QSYM方法发现的内存消耗漏洞数量至少要多17.9%，并且MemLock在发现内存消耗漏洞的速度上更快，至少是其它方法的2.07倍。而且，MemLock在现实世界的主流开源应用程序中发现了28个安全攸关的内存消耗漏洞（28 CVEs），也证明了其在实际应用的有效性。

本章的其它章节安排如下：首先，第3.2节详细介绍本章提出的内存消耗漏洞检测方法，包括整体流程、静态分析与程序插桩、内存消耗导向的模糊测试三个部分；接着，第3.3节从崩溃数量、漏洞检测能力、内存泄漏大小、种子造成的内存消耗等方面对MemLock进行实验评估与分析；最后，第3.4节对本章进行总结。

3.2 内存消耗漏洞检测方法

3.2.1 整体流程

本章所提的MemLock方法的整体流程如图3-3所示，主要包括两个阶段：①静态分析与程序插桩；②内存消耗导向的模糊测试。

静态分析与程序插桩阶段以程序的源代码作为输入，通过一系列的程序静态分析技术获得程序代码模型（即控制流程图和函数调用图）以及内存分配/释放相

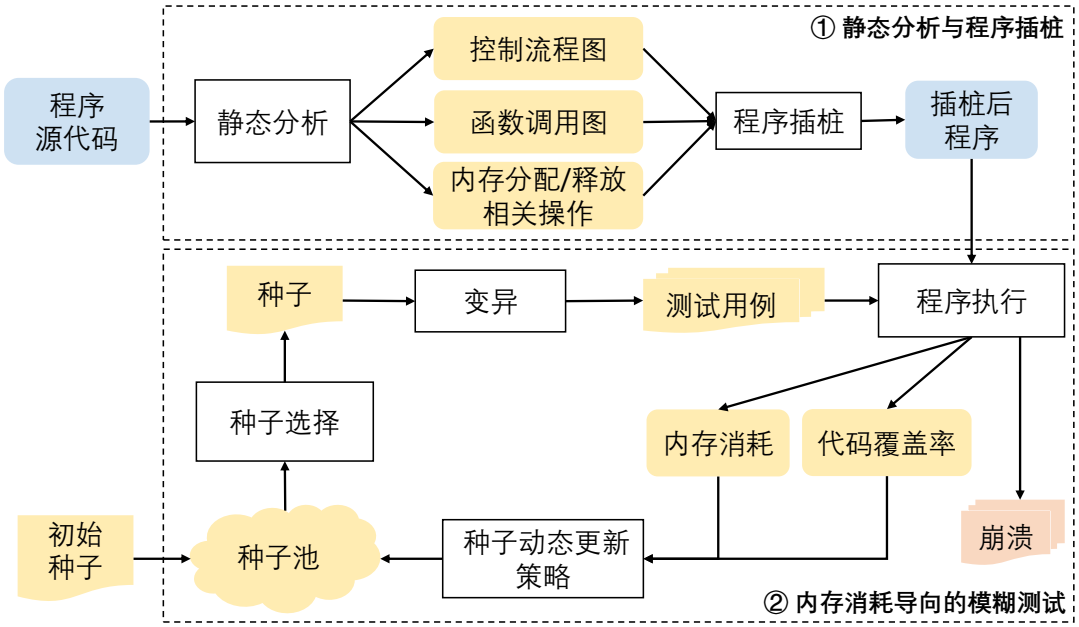


图 3-3　MemLock 的整体流程

关操作（如 *malloc* 和 *free* 等）的位置，再基于静态分析的结果做程序插桩。静态分析的目的和作用主要是为了确定对目标程序进行插桩的位置和内容，以使目标程序在运行时能正确地收集覆盖率信息和内存消耗信息，并反馈给模糊测试阶段。具体来说，程序的控制流程图描述了一个过程内的控制流路径，基于控制流程图插桩可用于收集覆盖率信息；函数调用图描述了过程之间的调用关系，是过程间分析需要用到的程序结构，基于函数调用图插桩有助于跟踪函数调用和返回时的栈帧信息；内存分配/释放相关操作体现了程序运行时内存消耗的变化，基于内存分配/释放相关操作插桩，有助于估算程序运行时内存消耗的峰值。

模糊测试以插桩后的目标程序作为测试对象，自动化地持续生成测试用例对目标程序进行测试，该过程一旦启动运行，即可在预算的时间范围内自动化地检测漏洞而无需人工干预。给定少量的测试用例集合作为初始的种子池，MemLock 每次从种子池中选择一个种子，使用不同的变异算子对该种子进行变异，以自动地生成大量新的测试用例。这些测试用例都会用作目标程序的输入来对目标程序进行测试。待目标程序执行后，MemLock 会收集此次执行的内存消耗信息和覆盖率信息，并用于指导种子的保留和选择。如果生成的测试用例能造成更大的内存消耗或覆盖了新的分支，那么该测试用例会作为高质量的种子加入到种子池中。MemLock 采用了一个新颖的种子动态更新策略，为每条程序路径维护一个能造成最大内存消耗的种子。MemLock 会不断重复此过程，直到达到了预设的资源限制。在期间，目标程序动态执行触发的崩溃都会当作已发现的程序漏洞报告出来，以待进一步的人工分析。

CVE-2018-17985 漏洞的检测过程：图 3-1程序接收的输入主要是字符串，输入字符串中的每一个字符都会在第 5 行被解析并存放到 peek 变量中，假设执行的初始种子仅为一个‘a’字符[2]，它会使得第 5 行 peek 的值也为‘a’。当进行一段时间的模糊测试后，基于覆盖率信息的引导，很容易生成一个新的输入 i_1 包含‘P’字符，使得 peek 的值为‘P’字符。由于该输入覆盖了新的分支，它将作为高质量的种子被添加到种子池当中。当下一次从种子池中选择到 i_1 进行变异时，它能生成包含多个‘P’字符的输入 i_2（例如 “PPPP”），这会使得第 12 行触发四次连续的递归调用。根据 AFL 中循环桶的概念，i_2 和 i_1 对于 10-12 行分支的执行次数是不同的，因此 i_2 会被添加到种子池当中。当后续选择 i_2 进行变异时，有一定概率可以生成包含更多‘P’字符的 i_3（例如 “PPPPP”）。由于 i_3 和 i_2 的执行路径是相同的，即 i_3 不能提供新的分支覆盖，现有的基于覆盖率的灰盒模糊测试技术将会丢弃 i_3，而 MemLock 引入了以内存消耗为指导的反馈机制，由于 i_3 会造成更多次数的递归调用以及消耗更大的栈内存空间，因此 MemLock 会将 i_3 当做一个高质量的种子添加到种子池之中。而且，由于 MemLock 的种子动态更新策略仅会为每条程序路径维护一个能造成最大内存消耗的种子，因此 MemLock 在保留 i_3 的同时会将 i_2 从种子池中剔除，由此可增加 i_3 被选择的机会。当 i_3 在经过多轮变异后，会生成更多数量的‘P’字符，MemLock 的种子动态更新策略也会将包含更多数量的‘P’字符的测试用例不断更新到种子池之中，直到生成了包含足够多‘P’字符的测试用例（上千万个‘P’字符），使得目标程序在测试执行时触发了栈溢出崩溃。至此，MemLock 发现并报告出该程序的不受控的递归调用漏洞。

CVE-2018-4868 漏洞的检测过程：为了便于说明，假设可用堆内存的上限为 10000 字节。覆盖引导的模糊测试技术很容易探索到第一次执行到第 11 行时条件表达式分别为 true 和 false 的情况，假设已经探索到能覆盖第 12-15 行（即为 true 的情况）的测试用例 l_1，并且该测试用例使得第 12 行 subBox.length 的值为 100；该测试用例的一个变异体 l_2 可能仍能覆盖第 12-15 行，同时还为 subBox.length 提供了更大的值（例如 200）。由于以变异体 l_2 作为输入执行时未覆盖新的分支，基于覆盖引导的灰盒模糊技术会将 i_{200} 丢弃，因此错过了一个能造成更大内存分配的优质种子。而 MemLock 引入了以内存消耗为指导的反馈机制，由于 l_2 造成的内存分配比 l_1 更大，虽然 l_2 和 l_1 执行的程序路径是一样，MemLock 会将 l_2 看作一个更高质量的输入，用它替换掉种子池当中的 l_1。当 l_2 经过多轮变异后，会生成能造成更大内存分配的测试用例，MemLock 的种子动态更新策略也会将这些占用内存

[2]假设的值具有一般性，也可以是‘b’或‘c’等字符，对于任何特殊情况应该都是无偏的。

更多的测试用例不断更新到种子池之中，最终当生成了能造成足够大内存分配的测试用例时（例如分配11000字节的堆内存），测试执行时就触发了内存分配失败异常，MemLock检测出该程序有不受控的内存分配漏洞。特别需要注意的是，如果第4行用new操作分配的堆内存空间在程序运行结束前一直未被释放，那么该程序存在内存泄漏漏洞。无论是测试用例l_1还是l_2都能产生内存泄漏报告，l_1造成的内存泄漏总量为100字节，而l_2造成的内存泄漏总量为200字节。于攻击者而言，测试用例l_2所造成内存泄漏危害更严重，这是因为每次造成更大的内存泄漏可以更快地让软件系统瘫痪或崩溃。MemLock能够感知到内存泄漏的大小受到输入的影响，利用内存消耗导向的模糊测试方法生成造成大量内存泄漏的测试用例。因此MemLock不仅检测出不受控的内存分配漏洞，还能暴露内存泄漏漏洞及其严重性。

3.2.2 静态分析与程序插桩

静态分析的目的和作用主要是为了确定与堆和栈上的内存消耗相关的程序语句的位置，实施程序插桩，使得目标程序在运行时能正确地收集覆盖率信息和内存消耗信息并在模糊测试阶段给予反馈。具体来说，MemLock的静态分析主要分析三类静态信息，分别是控制流程图（CFG）、函数调用图（CG）和内存分配/释放相关操作。基于控制流程图插桩可用于收集覆盖率信息，有助于引导模糊测试探索不同的分支和执行路径；基于函数调用图插桩有助于跟踪函数调用和返回时的栈帧信息，可以反映程序运行时的栈空间消耗，有助于引导模糊测试执行的用例消耗更大的栈内存空间；基于内存分配/释放相关操作插桩，用于计算程序运行时内存消耗的峰值，有助于引导模糊测试执行的用例消耗更大的堆内存空间。以下详细介绍本章是如何分析三类静态信息并进行程序插桩的。

控制流程图分析：控制流程图描述了一个过程内所有基本块执行的可能流向，也能反映程序执行过程中经过的路径。MemLock在程序运行时记录和收集程序的分支覆盖情况，即控制流程图中的边覆盖情况，用于引导模糊测试探索未覆盖的分支和路径，这和经典的覆盖引导的模糊测试技术是类似的[166]。MemLock为程序控制流程图中的每个基本块都分配了一个唯一的基本块标记，相应的，也为程序控制流程图中的每条有向边都分配了一个边标记。基本块标记和边标记分别定义如下。

定义3.1 (基本块标记). 对于控制流程图中的每一个基本块$b_i \in N$都有一个唯一的标记$ID_{b_i} \in \mathbb{N}$，其中$N = \{b_1, b_2, ..., b_n\}$表示所有基本块的集合，标记$ID_{b_i}$是一个自然数。对于任意两个基本块$b_l$和$b_m$，若$l \neq m$，则必然有$ID_{b_l} \neq ID_{b_m}$。

定义 3.2 (边标记). 对于控制流程图中的每一条有向边 $e_i \in E$ 都有一个唯一的标记 *Edge-ID*$_{e_i} \in \mathbb{N}$，其中 $E = \{e_1, e_2, ..., e_n\}$ 表示所有向边的集合，标记 *Edge-ID*$_{e_i}$ 是一个自然数。对于控制流程图中的任意一条有向边 $e_i = (b_l, b_m)$，其中 b_l 和 b_m 是两个连通且相邻的基本块，它们的基本块标记分别为 *ID*$_{b_l}$ 和 *ID*$_{b_m}$，则该有向边的边标记可表示为 *Edge-ID*$_{e_i}$ = *ID*$_{b_l} \oplus$ *ID*$_{b_m}$。$\oplus$ 是一个二元运算操作，该操作不满足交换律，即 *ID*$_{b_l} \oplus$ *ID*$_{b_m} \neq$ *ID*$_{b_m} \oplus$ *ID*$_{b_l}$。

对于程序控制流程图中每个不同的基本块，其基本块标记都是不同的，而相邻基本块之间的有向边的边标记则是通过对基本块标记进行 $\oplus$ 运算得到的，$\oplus$ 运算是一种在整数域上能使得整个控制流程图中边标记的重复率尽可能低的二元运算。同时，为了区分有向边的方向，$\oplus$ 运算应不满足交换律[3]。通常模糊测试中的覆盖率指的就是对程序控制流程图中边的覆盖率，通过计算边标记的覆盖率来得到。

MemLock 通过程序插桩将记录边标记的代码插入到目标程序相应的分支当中，当程序运行时就可以通过根据插桩代码被执行的次数来推断出每个分支的执行次数，从而记录边覆盖率信息。为了兼顾覆盖率收集所需的时间和空间，边标记的值域会被控制在一个固定的范围内，然后用一个大小固定的位图（BitMap）来保存程序这一次动态执行路径中边标记到边的执行次数的映射关系，本章将这个位图称作“路径位图”，其定义如下。

定义 3.3 (路径位图). 对于程序的一次动态执行路径，其路径位图可以用一个以 *8-bit* 为单位的长度为 2^K 的数组 $traceBits$ 来表示，记录了该执行路径所覆盖的控制流程图中边的边标记到其执行次数的映射关系。假设控制流程图中的边标记的值域在 $[0, 2^K)$ 范围内，对于控制流程图中的每一条有向边 $e_i \in E$，$traceBits$[*Edge-ID*$_{e_i}$] 表示 e_i 在本次执行路径中的执行次数[4]。

路径位图可以反映一条具体程序路径上的分支覆盖情况（即边标记的覆盖情况），对于不同的路径位图，其反映的程序执行路径也不同（反之不一定成立），因此本章通过将路径位图编码成一个路径标记来表示一条执行路径。

定义 3.4 (路径标记). 对于任意一条已覆盖的程序路径，它的路径标记 $k \in \mathbb{N}$ 是其对应的路径位图 $traceBits$ 的哈希值，记作 $k = Hash(traceBits)$。

[3]在 AFL 中，$\oplus$ 运算被实现为“将左操作数右移一位之后再与右操作数进行异或”。为了便于评估，MemLock 在此处的实现与 AFL 保持一致

[4]在 AFL 中，K 默认被设置为 16；AFL 简单将分支的执行次数归类到 8 个区间上: [0]、[1]、[2]、[3]、[4, 7]、[8, 15]、[16, 31]、[32, 127]、[128, 255]，仅将不同区间的执行次数看作是不同的。在实验时，MemLock 此处采取了与 AFL 相同的配置

函数调用图分析：函数调用图描述了程序中各个函数之间的调用关系，若函数调用图中因函数的相互调用而存在环，则表示这些函数之间存在递归调用。MemLock 不仅利用程序执行的覆盖率信息，还充分利用内存消耗信息。而连续的不断的函数调用，特别是递归函数的调用，是可能导致栈内存被大量消耗的重要因素。当函数调用发生时，系统会为函数分配一段临时的栈帧空间，函数的参数、返回值和局部变量等都会进行压栈操作，在函数调用返回之前它们都在占用着计算机的栈空间。当函数调用返回时，函数占用的栈帧空间会被释放，系统会回收已分配的栈内存以供重复利用。

MemLock 通过跟踪函数调用栈上的函数个数来评估栈空间的消耗，MemLock 在每个函数的入口（即函数中第一条语句）和出口（即函数中所有的 return 语句）前通过程序插桩注入一段更新函数调用深度的代码。本章使用 ft 来表示程序运行时调用栈上的函数个数，该值会随着程序的执行而动态变化。当有函数被调用时，ft 的值增加 1；当执行的函数返回时，ft 的值减少 1。在下文中，本章使用 fm 来表示程序执行期间 ft 的峰值，实际上 MemLock 关注的是 fm 的值，因为 fm 定性地反应了函数调用的最大深度，即从程序执行开始到结束期间调用栈上被调用函数的最大个数。基于函数调用图分析并进行程序插桩，程序动态执行后就可以获得 fm，后续模糊测试阶段将基于 fm 值的反馈引导不断生成新测试用例使得 fm 的值更大。

内存分配/释放相关操作分析：内存分配操作会向系统申请堆内存空间，在程序显式调用内存释放操作之前，已分配的堆内存空间都无法重复利用。因此，程序中大量使用内存分配/释放相关操作也是导致堆内存被大量消耗的重要因素。通常程序执行不同路径时，其占用的堆内存大小是不同的；此外，程序中内存分配操作的参数还受到程序输入的影响，当为程序提供不同的输入时，程序占用的内存空间也是不同的。MemLock 需要获得程序动态运行时堆内存消耗的峰值，为此，需要跟踪每一个内存分配/释放操作引起的内存变化。内存分配操作需要通过参数显式指定其申请的内存空间，系统为其分配的内存空间的起始地址会以返回值的方式返回。MemLock 通过程序插桩获取每一个内存分配操作的参数与返回值。当每个内存分配操作执行时，MemLock 就可以根据其参数来计算内存消耗的增量。对于内存释放操作，同样可通过程序插桩获取参数，但其参数仅包含释放的内存空间的起始地址，而未包含释放的内存大小。为此，MemLock 维护了一个从内存分配操作的返回值到其参数的映射表。当内存释放操作执行时，可根据内存释放操作的参数获取到释放内存的大小。需要注意的是，仅需要对标准库中的内存分

配/释放相关操作进行插桩，获取其参数和返回值即可，因为应用程序的堆内存的分配都是通过一些标准库函数完成的[161,167]，例如 *malloc*、*calloc*、*realloc* 和 *new* 等操作；堆内存的释放操作也是通过相应的标准库函数完成的，例如 *free* 和 *delete* 等操作。即使程序中存在用户自定义的内存分配/释放函数，它们仍然依赖于标准库函数来分配/释放内存[168]。因此，对于实际的应用程序，不需要考虑用户自定义的内存分配/释放函数。

本章使用 ot 表示程序动态执行过程中消耗的堆内存空间的大小，当程序使用了内存分配相关操作申请到了 ot' 字节的堆内存时，值 ot 增加 ot'；而当程序使用内存释放相关操作释放了 ot' 字节内存，则值 ot 将减少 ot'。在下文中，本章使用 om 表示一次程序动态执行的整个过程中 ot 的峰值。MemLock 关注的是通过内存分配/释放操作计算出来的 om 的值，而不是在程序运行时去测算程序实际的内存消耗。一方面是因为程序运行时会受到很多外部因素的影响，实际的内存消耗不是一成不变的，即便以同一输入多次执行程序，其内存消耗也会有一定范围的波动，不利于对比；另一方面，要想准确地测算出真实的堆内存消耗势必要引入一定的运行时开销。om 的值能定性地反应程序运行时堆内存的最大消耗量，不受其它外部因素的影响。基于内存分配/释放相关操作进行程序插桩，程序动态执行后就可以获得 om，后续模糊测试阶段将基于 om 值的反馈引导不断生成新测试用例使得 om 的值更大。

算法 3.1: MemLock 的插桩过程

输入： 原始程序 P
输出： 插桩后的程序 P'

```
1  let CG be a Call Graph, and CFG be a Control flow graph of program P
2  foreach function f ∈ CG do                                  // 遍历每一个函数
3      foreach basicblock bb ∈ CFG_f do                        // 遍历函数中每一个基本块
4          if bb is the first basic block of CFG_f then        // 找到该函数的入口基本块
5              insert an operation that increases ft by 1, and update fm
6          foreach instruction i ∈ bb do                       // 遍历基本块中的每一条指令
7              if i.getOpcode() == Return then                 // 找到函数的出口
8                  insert an operation that decreases ft by 1
9              if i corresponds to the statement "ptr = alloc(size);" then // 内存分配函数调用
10                 insert an operation that calculates ot based on size and ptr, and update om
11             if i corresponds to the statement "free(ptr);" then // 内存释放函数调用
12                 insert an operation that calculates ot based on ptr
13         insert an operation that makes traceBits[bb_pre ⊕ bb]++, where bb_pre is the precursor of bb
```

算法 3.1 描述了本章基于三类静态信息进行程序插桩的过程。该过程以原始程序 P 作为输入，以插桩后的程序 P' 作为输出。首先，基于程序静态分析方法为该程序构造函数调用图 CG 和控制流程图 CFG（第 1 行）。为获取程序执行时函数调用的最大深度，算法依次遍历 CG 中的每一个函数，在函数入口，即函数第一个基本块的开头插入更新 ft 的操作，使得 ft 增加 1（第 3-5 行）；并依次遍历函数中的每一条指令，在函数出口处，即 $return$ 指令前插入更新 ft 的操作，使得 ft 减小 1（第 6-8 行）。为获取程序执行时的分支覆盖情况，算法依次遍历各个函数的控制流程图 CFG_f 中的每一个基本块 bb，并在其中插入一段根据执行到的边标记更新路径位图 $traceBits$ 的操作（第 13 行）。为获取程序执行整个过程中堆内存消耗的的峰值，算法在每个内存分配相关操作之前插入更新 ot 的操作，若 ot 大于 om 则更新 om 的值（第 7-8 行），同样的也在每个内存释放相关操作之前插入更新 ot 的操作（第 9-10 行）。完成静态分析与程序插桩之后，动态执行插桩后程序时获取的 fm、om 和 $traceBits$ 将被用于模糊测试阶段的算法 3.2 中。

3.2.3 内存消耗导向的模糊测试

内存消耗导向的模糊测试方法同时以内存消耗信息和覆盖率信息作为反馈，引导测试用例的自动生成朝着能造成大内存消耗的的方向迈进，同时结合本章提出的种子动态更新策略持续对目标程序进行测试，能有效发现内存消耗漏洞。

算法 3.2 描述了 MemLock 对一个目标程序进行模糊测试的过程。整体而言，MemLock 不断地从种子池中选取种子进行变异，自动化地生成一组新的变异体，并将这些变异体用作测试用例来对目标程序进行测试，同时监视它们的运行状况(第 3-10行)。若发现目标程序触发了内存消耗漏洞，则将相应的测试用例保存起来，以便将来复现和定位漏洞（第 11-12行)。如果测试用例能覆盖新的未覆盖分支或能造成更大的内存消耗，则它将作为一个高质量的种子添加到种子池中（第 14-17行)。MemLock 在预设的时间内持续遍历种子池挑选种子进行变异，不断生成测试用例对目标程序进行测试。该算法与传统的覆盖导向的模糊测试技术的主要区别有两点，一方面，该算法增加了内存消耗引导的反馈机制（详见第 3.2.3.1节），根据测试用例造成的内存消耗量来决定是否将该测试用例作为高质量的种子添加到种子池中，有助于生成的测试用例朝着使目标程序内存消耗增大的方向发展；另一方面，该算法设计了一个种子动态更新策略（详见第 3.2.3.2节)，使得对种子的选择和变异更加有效，提升模糊测试检测内存消耗漏洞的效率。

具体而言，算法 3.2 以插桩后的可执行程序 P' 和一组初始种子 T 作为输入，以一个能触发内存消耗漏洞的测试用例集合 S 作为输出。算法开始时，集合 S 以

算法 3.2: 内存消耗导向的模糊测试算法

输入：一个插桩后的程序 P'，以及一个初始种子的集合 T
输出：触发内存消耗漏洞的测试用例集合 S

```
1  S ← Φ
2  Queue ← T
3  while time and resource budget do not expire do
4      for each input t in Queue do
5          if with probability FuzzProb_t to select t then                    // 种子选择
6              numChildren ← AssignEnergy(t)
7              for 0 ≤ i < numChildren do
8                  child_i ← Mutate(t)                                        // 对种子进行变异
9                  (traceBits_i, fm_i, om_i) ← Run(child_i, P)                // 执行程序并收集反馈信息
10                 k = Hash(traceBits_i)                                      // 计算路径 ID
11                 if the execution triggers memory consumption bugs then
12                     S ← S ∪ child_i
13                 else
14                     if NewCov(traceBits_i) then                            // 代码覆盖引导的反馈机制
15                         Queue ← Queue ∪ child_i
16                     if NewMax(fm_i, om_i) then                             // 内存消耗引导的反馈机制
17                         Queue ← Update(child_i, fmMap[k], omMap[k])        // 种子动态更新
18 return S
```

空集作为初始值。模糊测试过程中维护的种子池实际上是一个队列，算法中用变量 $Queue$ 表示，后文也称其为种子队列。$Queue$ 使用一组初始种子 T 进行初始化（第2行）。算法每次从种子池 $Queue$ 中选择一个种子（第4行），并基于一定的概率决定是否选择对该种子进行变异（第5行）。假设测试用例 t 被选择进行变异，用例 t 会被分配一定的能量值（$numChildren$），即基于测试用例 t 变异生成的变异体的个数。MemLock 使用与 AFL[30] 相同的启发式方法确定 $numChildren$ 的值，它倾向于为覆盖率更高的测试用例赋予更多的能量值。对测试用例 t 进行变异后，会生成 t 的变异体 $child_i$（第8行），算法将 $child_i$ 用作程序 P' 的新测试用例，执行程序并监视其运行情况（第9行），分别收集分支覆盖率信息 $traceBits_i$、最大函数调用深度 fm 和最大堆内存消耗量 om。若测试用例 $child_i$ 执行时发生了崩溃，触发了内存消耗漏洞，则它将被添加到集合 S 中（第12行）。否则，算法进一步分析 $child_i$ 覆盖的分支情况和内存消耗情况（第14和16行）。若 $child_i$ 的执行覆盖了新的未覆盖分支，则将其作为高质量的种子添加到 $Queue$ 中等待下一轮被选择时进行变异（第14-15行）；若 $child_i$ 的执行能造成更大的内存消耗（定义3.8），则通过种子动态更新策略将 $child_i$ 更新到种子队列中（第16-17行）。算法重复以上过程，

直到超出了时间或其它资源的预算范围（第3行）。

3.2.3.1 反馈机制

反馈机制是影响灰盒模糊测试技术效果的重要因素。反馈机制能够评估先前测试用例的测试效果并反馈给模糊测试系统以指导新的测试用例的生成，可提高测试用例生成的质量，并且减少生成不必要的测试用例，从而整体上提高了模糊测试的效率并能达到更加有效的测试效果。反馈机制的设计也决定了模糊测试技术在发现漏洞方面的能力，例如，AFL 使用代码覆盖引导的反馈机制，高效探索不同的程序路径，有利于发现一般性的内存破坏漏洞；SlowFuzz[84] 使用以执行指令数量为引导的反馈机制，有助于发现算法时间复杂性漏洞。而 MemLock 在保留代码覆盖的反馈机制的同时，还引入了内存消耗引导的反馈机制，一方面基于分支覆盖率信息有效探索不同的程序路径，另一方面利用内存消耗信息驱动生成的测试用例不断造成更大的内存消耗，从而能够更有效地发现内存消耗漏洞。为了进一步解释算法 3.2 中的反馈机制（第 14和 16行），本章引入以下定义：

定义 3.5 (最大函数调用深度). 给定一条程序路径 k，以及所有执行了该路径的测试用例的集合 I，该路径上已执行到的最大函数调用深度用 $fmmap[k]$ 表示，其值为 I 中的所有测试用例执行时 fm 的最大值：

$$fmmap[k] \leftarrow \max_{i \in I} fm_i \tag{3-1}$$

其中 fm_i 表示测试用例 i 执行过程中调用栈上函数个数的峰值。

定义 3.6 (最大堆内存分配量). 给定一条程序路径 k，以及所有执行了该路径的测试用例的集合 I，该路径上已造成的最大堆内存分配量用 $ommap[k]$ 表示，其值为 I 中的所有测试用例执行时 om 的最大值：

$$ommap[k] \leftarrow \max_{i \in I} om_i \tag{3-2}$$

其中 om_i 表示测试用例 i 执行过程中消耗的堆内存空间的峰值。

定义 3.7 (新的分支覆盖, *NewCov*). 给定已测试过的测试用例的集合 I，以及一个新的测试用例 t，本章说 t 触发了新的分支覆盖，当且仅当 t 执行到了至少一条未被集合 I 覆盖的分支，或者一个已覆盖的分支触发了新的执行次数（参见定义 *3.3*）。

函数 $NewCov$（第 14行）用于检查新生成的测试用例 $child_i$ 是否执行到了当前 $Queue$ 中的测试用例未覆盖的分支，该检查可通过对比测试用例的路径位图来实现。$NewCov$ 函数的作用主要是充分利用分支覆盖率信息，指导 MemLock 探索不同的程序分支或路径。

定义 3.8 (更大的内存消耗，*NewMax*). 给定一个的测试用例 t，以及该测试用例的执行路径 k，本文说 t 造成了更大的内存消耗，当且仅当 $fm_t > fmmap[k]$，或者 $om_t > ommap[k]$。

函数 $NewMax$（第16行）用于确定新生成的测试用例 $child_i$ 是否能造成当前执行路径上更大的内存消耗量。一方面，它考虑 $child_i$ 的函数调用深度是否超过当前的最大函数调用深度（参见定义3.5）；另一面它还考虑到 $child_i$ 的堆内存消耗是否超过了当前的最大堆内存分配量（参见定义3.6）。如果测试用例 $child_i$ 满足上述两种情况中的任何一种，算法将基于种子动态更新策略，把测试用例 $child_i$ 更新到种子队列中（第17行，详见第3.2.3.2节）。

为了适应本章提出的内存消耗引导的反馈机制，MemLock 优化了种子被选择的概率（算法3.2的第5行）。对于种子池中的每个种子，种子被选择的概率遵循以下定义：

定义 3.9 (偏好用例). 对于一个测试用例 t，若 t 能触发新的分支覆盖（即 $NewCov$），或能导致更大的内存消耗（即 $NewMax$），则 t 被称作偏好用例。

定义 3.10 (种子被选择概率). 一个测试用例 t 是否被选择进行变异，是根据概率 $FuzzProb_t$ 来决定的，$FuzzProb_t$ 的计算方式如下：

$$FuzzProb_t = \begin{cases} 1 & \text{如果 } t \text{ 是偏好用例} \\ a & \text{如果 } t \text{ 不是偏好用例} \end{cases} \tag{3-3}$$

MemLock 倾向于选择偏好用例，并降低对非偏好用例选择的概率。对于一个测试用例 t，若 t 是偏好用例，则总是会被选择进行变异；若 t 不是偏好用例，仅有 a 的概率被选择进行变异。在本章的实验中 $a = 0.01$，这与 AFL[30] 和 PerfFuzz[85] 的设置保持一致。

3.2.3.2 种子动态更新策略

为了有效保留能触发新的分支覆盖或能导致更大的内存消耗的测试用例作为种子，本章提出了一个新颖的种子动态更新策略，旨在为每条程序路径都保留一个能造成最大内存消耗的种子，使得对种子的选择和变异更加有效，提升模糊测试对内存消耗漏洞的检测效率。

如图 3-4所示，模糊测试过程中会维护一个种子池，该种子池实际上是一个队列，队列中的每个节点分别代表一条已发现的程序路径，模糊测试按照“先进先出”的原则从种子池中挑选测试用例种子进行变异操作。由于 MemLock 同时关注分支覆盖率信息和内存消耗信息，而这两类信息在一定程度上是正交的，即执

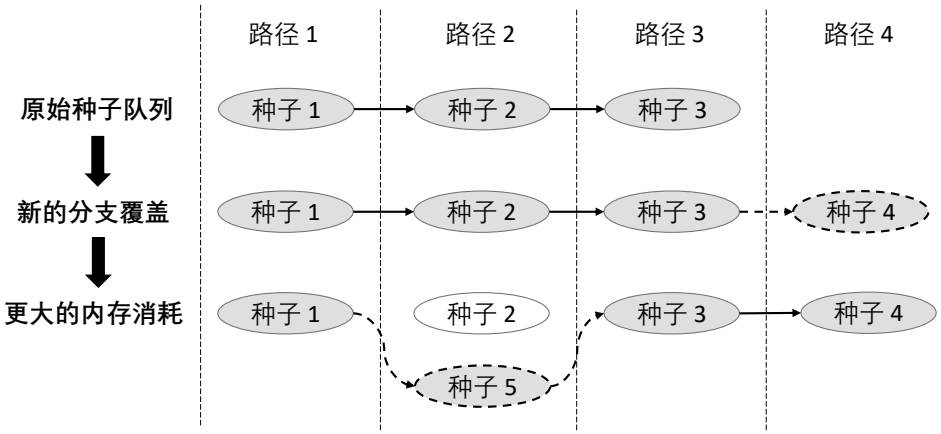


图 3-4　种子动态更新策略示意图

行同一条程序路径的测试用例可能造成不同的内存消耗，而能造成相同内存消耗的测试用例亦可能执行不同的程序路径。触发相同程序路径的测试用例会因内存消耗的增加而不断加入到种子池当中，造成种子池中种子数量的急剧增加，一些已经过时的种子仍然存留在种子池当中，导致高内存消耗的种子被选择的概率降低，遍历一遍种子队列的效率也大大降低。

本章提出的种子动态更新策略同时考虑分支覆盖率信息和内存消耗信息，在以下两种情况更新种子队列：(1) 触发新的分支覆盖 (即 *NewCov*)：如果测试用例 s 覆盖了新的未覆盖分支，则将测试用例 s 作为新的节点添加到种子队列中。如图 3-4第 2 行所示，“种子 4” 的执行路径与原种子队列中所有种子的执行路径都不同，则 “种子 4” 被添加到种子队列的末尾。(2) 造成更大的内存消耗 (即 *NewMax*)：如果测试用例 s 未覆盖新的分支，但造成的内存消耗量比种子池中与 s 的执行路径相同的种子 s' 更大，则 s' 将被 s 替换。如图 3-4第 3 行所示，尽管 “种子 5” 未覆盖新的分支，它与 “种子 2” 的执行路径时相同的，但 “种子 5” 能造成比 “种子 2” 更大的内存消耗，此时将 “种子 2” 替换为 “种子 5”。算法 3.2 的第 17 行通过用新生成的 $child_i$ 替换掉种子池中与其路径相同的种子，为每个路径保留了一个能造成更大内存消耗的种子，在充分考虑分支覆盖率信息和内存消耗信息，且不为种子池增加冗余度的情况下，逐步增加了种子池整体能造成的内存消耗量，带来长期稳定的测试效率提升。

3.3　实验评估

本章实现了一个融合程序分析与模糊测试的内存消耗漏洞检测工具原型 MemLock[5]。该工具原型使用 C/C++ 代码实现，其静态分析和插桩部分是基于 LLVM/-

[5]MemLock 是一个融合程序分析与模糊测试的内存消耗漏洞检测工具，已经通过了 ICSE'20 的 artifacts evaluated available and reusable 的认证，其工具原型与相关数据集可从https://github.com/wcventure/MemLock-Fuzz获取。

表 3-1　实验对象

ID	程序	版本号	代码行数	程序简介
1	mjs	1.20.1	40k	C/C++ 嵌入式 JavaScript 引擎，专为资源受限的微控制器而设计
2	cxxfilt	2.31	1,757k	编程语言工具程序；c++ 名字符号解析工具
3	nm	2.31	1,757k	编程语言工具程序；用于列出目标文件中的符号
4	nasm	2.14.03	105k	NASM 是一款基于 x86 架构的汇编与反汇编软件
5	flex	2.6.4	27k	flex 是一个词法分析器；它生成的程序能够处理结构化输入
6	yaml-cpp	0.6.2	58k	yaml-cpp 是用 c++ 实现的，用于解析和生成 yaml 文件
7	libsass	3.5.4	27k	libsass 是一个用 C 语言实现的 Sass 解析器
8	yara	3.5.0	45k	YARA 是一个帮助恶意软件研究人员识别和分类恶意软件样本的工具
9	readelf	2.28	1,844k	编程语言工具程序；从 ELF 格式的目标文件显示信息
10	exiv2	0.26	84k	Exiv2 是一个用于管理映像元数据的跨平台 c++ 库和命令行实用程序
11	openjpeg	2.3.0	243k	OpenJPEG 是用 C 语言编写的开源 JPEG 2000 编解码器
12	bento4	1.5.1	78k	Bento4 是一个处理 MP4 文件的 C ++ 库，旨在读取和写入 ISO-MP4 文件
13	libming	0.4.8	92k	libming 是一个用来生成 SWF（Flash 动画）文件的 C 程序库
14	jasper	2.0.14	44k	Jasper 是一个开源的语音控制的应用

Clang 框架[169] 实现的，而模糊测试部分是在开源模糊测试工具 AFL-2.52b 的基础之上实现的。本章对实现的 MemLock 工具原型进行了全面的实验评估。

3.3.1　实验对象

本章选取了 14 个不同规模及功能多样的真实应用程序作为实验对象，表3-1列出了详细信息。表中第 1 列表示程序 ID，第 2 列表示程序名，第 3 列表示程序的版本号，第 4 列显示代码行数，第 5 列对程序做简单描述。

本章实验对象涵盖了开发工具（例如 *nm*，*cxxfilt*，*readelf*）、代码处理工具（例如 *nasm*，*flex*，*yaml-cpp*，*mjs*）、图像处理库（例如 *openjpeg*，*jasper*，*exiv2*）、音视频处理工具（例如 *bento4* 和 *libming*）、结构化数据处理库（例如 *libsass* 和 *yara*）等多个方面。这些实验对象都是在实际应用领域经常被使用的开源 C/C++ 程序，规模大小从 27k 到 1844k 行 C/C++ 代码不等，且多次被研究者使用[64,67,70,170,171]。

3.3.2　实验设计

为了对本章方法进行深入研究，本章设计了四大类实验，分别验证以下四个研究问题：

问题一： MemLock 在发现崩溃的数量和效率方面表现如何?

问题二： MemLock 在发现真实的内存消耗漏洞上的能力如何?

问题三： MemLock 是否能触发导致更大内存泄漏的漏洞?

问题四： MemLock 的方法机制和策略是否真的能帮助生成能造成更大内存消耗的测试用例?

为了更好的评估 MemLock，本章选取了六个最具代表性的、业界先进的模糊

测试工具作为MemLock的对比基准，它们分别是AFL[30]、AFLfast[63]、PerfFuzz[85]、FairFuzz[64]、Angora[33]和QSYM[35]。选取这六个模糊测试工具主要出于以下考虑：AFL是谷歌Michal Zalewski开发的一款强大的基于代码覆盖导向的灰盒模糊测试工具，从2013年发布以来，它已经在安全领域被广泛使用，并在学术界作为大多数模糊测试研究工作的对比基准；AFLfast是在AFL基础之上演进的一款变体，它通过更有效的能量调度策略提升了AFL的测试效率；PerfFuzz侧重于发现程序中的的时间复杂性漏洞；FairFuzz利用目标突变策略引导模糊测试探索执行更稀有的分支，从而更高效地提升测试时的覆盖率；Angora基于污点分析跟踪信息流，然后使用梯度下降法高效突破未覆盖分支；QSYM是一个融合了符号执行和模糊技术的混合模糊测试工具，结合两种技术各自的优点从而可以生成具有更高分支覆盖率的测试用例。总之，本章选择的对比基准都是各类具有代表性的、业界先进的模糊测试工具，它们针对不同问题，采取了不同的技术策略，并在实践中广泛用于漏洞发现。本章将这些工具与MemLock一起验证在发现内存消耗漏洞方面的有效性和效率。

在实验的评估标准上，本章遵循了Klees等人在“Evaluating Fuzz Testing”[170]论文中所提出的建议，最直接的评估标准就是发现漏洞的能力。本章评估了各个模糊测试工具在模糊测试过程中直接发现的唯一崩溃（Unique Crashes）的数量，以及发现特定漏洞所消耗的时间。由于MemLock旨在生成能造成更大内存消耗的测试用例，本章统计了各个模糊测试工具触发的内存泄漏大小，并且还统计了种子池中每个种子能造成的内存消耗信息，并观测它们的频率分布。

在配置参数方面，本章参考了PerfFuzz[85]的配置，使用-d选项运行所有模糊测试工具，以跳过模糊测试变异过程中的确定性突变阶段；并且对于所有的模糊测试工具，其配置参数和使用的初始种子是相同的。由于模糊测试严重依赖于随机性的变异，因此每次对模糊测试的评估结果可能会出现一定范围的波动。本章主要采取了三项措施来缓解由于随机性造成的实验偏差。首先，本章会对每个目标程序进行较长时间的测试，直到整个模糊测试达到相对稳定的状态。因此本章每组实验统一运行每个模糊测试工具长达24小时。其次，本章对每个实验对象都会进行10次重复实验，统计平均结果，并基于统计学假设检验评估各个工具的性能。再者，对于同一目标程序，各个模糊测试工具在初始种子的选择上都是相同的。如果目标程序提供了测试用例样本，本章直接将其用作初始种子。否则，根据所需的输入格式从互联网上随机下载一些符合输入格式的文件作为测试用例。

本章以ASAN[27]作为在运行时检测内存消耗漏洞的测试预言（Test Oracle）。对

表 3-2　唯一崩溃数量评估

程序	漏洞类型	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		Angora		QSYM	
		崩溃数	崩溃数	$\hat{A}_{12}$	崩溃数	$\hat{A}_{12}$	崩溃数	$\hat{A}_{12}$	崩溃数	$\hat{A}_{12}$	崩溃数	$\hat{A}_{12}$	崩溃数	$\hat{A}_{12}$
mjs	CWE-674	114	36	**1.00**	31	**1.00**	88	**0.96**	12	**1.00**	0	**1.00**	30	**1.00**
cxxfilt	CWE-674	448	373	**1.00**	304	**1.00**	401	**0.88**	39	**1.00**	0	**1.00**	327	**1.00**
nm	CWE-674	127	12	**1.00**	21	**1.00**	17	**1.00**	0	**1.00**	0	**1.00**	20	**1.00**
nasm	CWE-674	132	6	**1.00**	4	**1.00**	40	**1.00**	0	**1.00**	0	**1.00**	4	**1.00**
flex	CWE-674	61	0	**1.00**	0	**1.00**	0	**1.00**	0	**1.00**	0	**1.00**	0	**1.00**
yaml-cpp	CWE-674	4	0	**1.00**	1	**1.00**	3	0.56	0	**1.00**	0	**1.00**	0	**1.00**
libsass	CWE-674	23	6	**1.00**	4	**1.00**	23	0.53	11	**0.88**	26	**0.25**	7	**1.00**
yara	CWE-674	156	34	**1.00**	33	**1.00**	65	**0.94**	13	**1.00**	0	**1.00**	31	**1.00**
readelf	CWE-789	273	104	**1.00**	110	**1.00**	54	**1.00**	181	**0.88**	0	**1.00**	114	**1.00**
exiv2	CWE-789	10	11	**0.14**	11	0.20	6	**0.90**	15	**0.00**	13	**0.16**	8	0.52
openjpeg	CWE-789	16	8	0.80	5	**1.00**	0	**1.00**	7	0.46	0	**1.00**	5	0.80
bento4	CWE-789	5	2	**1.00**	2	**0.98**	2	**1.00**	1	**1.00**	189	**0.00**	1	**1.00**
	CVE-401	145	78	**1.00**	72	**1.00**	61	**1.00**	125	**1.00**	290	**0.00**	74	**1.00**
libming	CWE-789	18	20	0.40	18	0.60	17	0.62	20	**0.20**	3	**1.00**	16	0.80
	CVE-401	264	336	**0.20**	324	**0.00**	324	**0.00**	371	**0.00**	87	**1.00**	354	**0.00**
jasper	CWE-789	3	2	**0.84**	3	0.56	0	**1.00**	3	0.56	2	**1.00**	2	**0.92**
	CVE-401	210	234	**0.08**	235	**0.08**	35	**1.00**	216	0.40	820	**0.00**	212	0.46
唯一崩溃总数		**2009**	**1262 (+59.2%)**		**1178 (+70.5%)**		**1136 (+76.9%)**		**1014 (+98.1%)**		**1430 (+40.5%)**		**1205 (+66.7%)**	

* CWE-674 代表“不受控的递归调用”漏洞，CWE-789 代表“不受控的内存分配”漏洞，CVE-401 为“内存泄露”漏洞；$\hat{A}_{12}$ 被加粗则代表着相应的 *Mann-Whitney U* 检验结果说明 MemLock 与其对比的工具的发现的 Unique 崩溃数数量差异显著。

于不受控制的递归调用漏洞，其漏洞触发时会导致栈内存空间耗尽，ASAN 会报告栈内存溢出（Stack-overflow）；对于不受控制的内存分配漏洞，其漏洞触发时会分配大量内存空间或导致内存分配失败，可以通过设置“allocator_may_return_null”选项让 ASAN 报告出这类漏洞。此外，集成在 ASAN 之中的 LeakSanitizer 可以检测内存泄漏漏洞及其泄漏的内存大小。

本章所有的实验环境为 Intel (R) Xeon (R) E5-1650 v3 的处理器，主频为 3.40Hz，64 位 Ubuntu LTS 16.04 操作系统，16GB 内存。

3.3.3　实验结果与分析

3.3.3.1　实验一：崩溃数量评估

唯一崩溃（Unique Crashes）是模糊测试工具报告并记录的崩溃数量，AFL 及其衍生物通过检查相同的边（指控制流程图中的边）是否已被执行并导致崩溃来确定崩溃的唯一性。如果两个崩溃所触发的执行路径覆盖的边是相同的，则视为相同的崩溃；否则视作两个的不同的崩溃。评估模糊测试工具发现的唯一崩溃数量是评估模糊测试工具有效性的最直接的手段，发现的唯一崩溃数量越多，通常意味这发现特定漏洞的概率越大。本节后文中提到的崩溃，皆指代模糊测试工具报告的唯一崩溃。

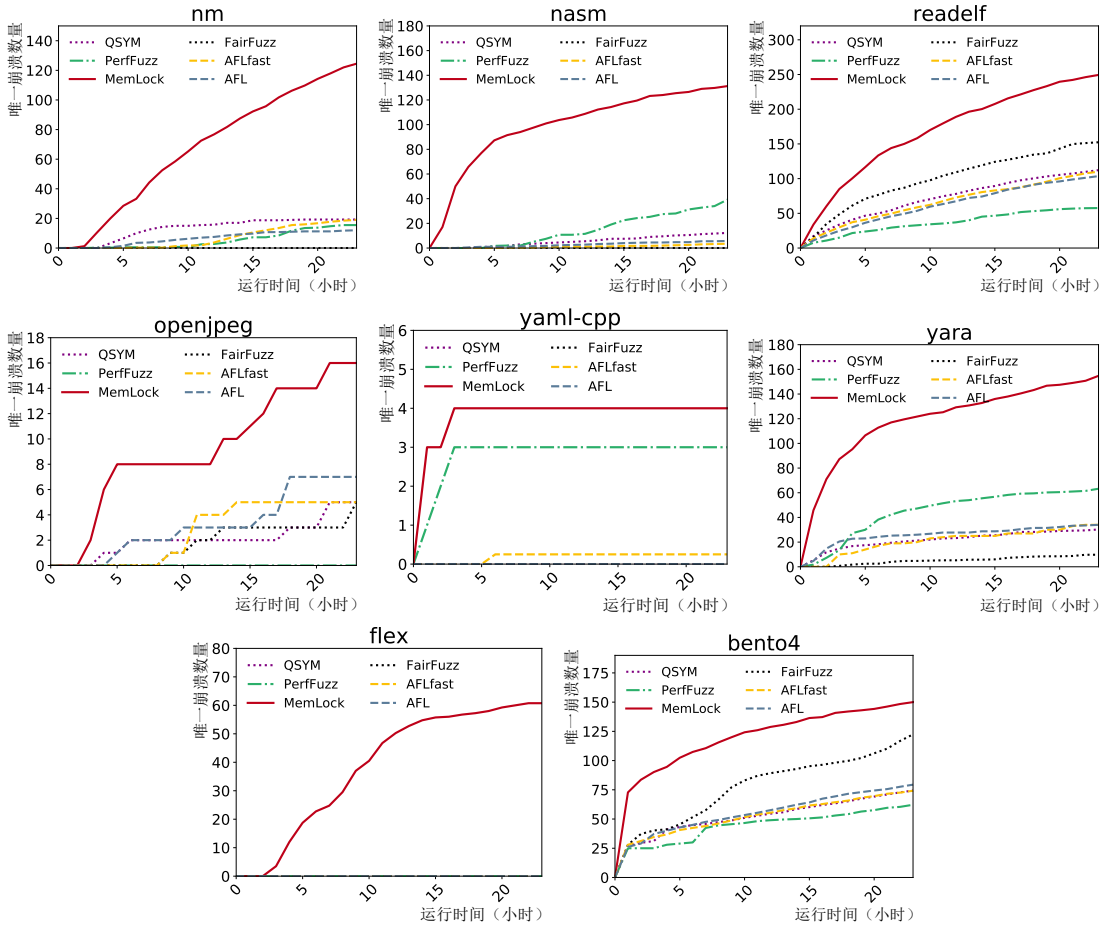


图 3-5　唯一崩溃的增长趋势

表 3-2显示了七个不同的模糊测试工具在 10 次 24 小时的重复实验中发现的与内存消耗漏洞相关的崩溃数量[6]。在 17 组关于崩溃数量的实验数据中，MemLock 在其中 10 组（58.8%）实验中发现的崩溃数量最多。在发现的崩溃总数上，MemLock 共发现 2009 个崩溃，分别相比其它 6 个业界先进的模糊测试工具发现的崩溃总数都要多。与 AFL、AFLfast、PerfFuzz、FairFuzz、Angora 和 QSYM 这些模糊测试工具相比，MemLock 发现的与内存消耗漏洞相关的崩溃数量分别提高了 59.2%、70.5%、76.9%、98.1%、40.5% 和 66.7%。值得一提的是，MemLock 还能够发现其它 6 个模糊测试工具都找不到崩溃。例如，其他 6 个模糊测试工具在 10 次重复实验中都无法在程序 *flex* 中发现任何的崩溃，而 MemLock 却能够在 24 小时内平均找到 61 个关于不受控的递归调用的崩溃，这体现了 MemLock 在发现内存消耗漏洞方面的优越性。

为了更好地比较不同的模糊工具发现崩溃的效率，本章还绘制了各个工具发现崩溃数量的增长趋势图。图 3-5描述各个模糊测试工具在不同的实验对象上发现

[6]本章通过复现崩溃并分析 ASAN 的漏洞报告和崩溃触发时的调用栈来识别与内存消耗漏洞相关的崩溃；对于其它类型的漏洞，本章会在第 3.3.3.5节中另外讨论。

崩溃数量随时间的增长趋势，其中红色线条表示 MemLock。可以观察到 MemLock 发现崩溃数量的增长趋势通常是最快的；在模糊测试工具稳定运行一段时间后，MemLock 发现的崩溃的数量一直处于领先地位（例如从第 3 个小时到第 24 个小时）；并且，MemLock 通常也是第一个报告崩溃的模糊工具。这表明 MemLock 在发现内存消耗漏洞的崩溃方面有着稳定而快速的增长趋势，相比其它模糊测试工具，MemLock 发现内存消耗漏洞的崩溃的效率更高。

本章还对 10 次重复实验的结果进行了统计学假设检验，本章基于 Klees 等人[170] 的建议，使用了 Vargha Delaney 度量[172] 来计算 MemLock 优于每个对比基准的可信度，并采用曼-惠特尼 U 检验[173]（Mann-Whitney U 检验）来检查实验结果的统计显著性差异。为了统计量化 MemLock 比每个对比基准好多少，本章用 $\hat{A}12$ 来表示 Vargha Delaney 度量的值。当应用于 MemLock 和对比基准所发现的崩溃数量时，$\hat{A}12$ 为 0 到 1 之间的值，如果该值正好为 0.5，则两个模糊测试工具的性能相同；如果该值大于 0.5，则表示 MemLock 的性能更好，如果该值小于 0.5，则表示其对比基准的性能更好。换句话说，$\hat{A}12$ 越接近 0.5，代表两个模糊测试工具性能上的差异越小；$\hat{A}12$ 距离 0.5 越远，则表示两个模糊测试工具性能上的差异越大。本章将 MemLock 分别与其它六个业界先进的模糊测试工具进行比较，并在表 3-2的 $\hat{A}12$ 列给出 Vargha Delaney 度量的值。曼-惠特尼 U 检验假设两个样本分别来自除了总体均值以外完全相同的两个总体，目的是检验这两个总体的均值是否有显著的差别。曼-惠特尼 U 检验中根据显著性检验方法所得到的 P 值，一般以 P<0.05 表示存在统计学差异，其含义是样本间的差异由抽样误差所致的概率小于 0.05。在表 3-2中，如果 P 值小于显著性水平（0.05）的值，则对应的 $\hat{A}12$ 值会被用粗体标记。较小的统计显著性差异（P 值）表明 MemLock 和其它对比基准之间的差异更显著。在表中的 102 组 $\hat{A}12$ 值中，有 72 组（70.6%）$\hat{A}12$ 值超过了 0.71[7]，同时其 P 值小于 0.05，这也表明 MemLock 在发现内存消耗漏洞的崩溃数量上能力更强。因此可以得出结论，在大多数实验对象中，MemLock 显著优于其它 6 个业界先进的模糊测试技术。

实验结论：基于对表 3-2和图 3-5的详尽分析，相比其它 6 个先进的模糊测试技术（即 AFL、AFLfast、PerfFuzz、FairFuzz、Angora 和 QSYM），MemLock 在发现与内存消耗漏洞相关的崩溃的数量和效率上都更强。

[7]当 $\hat{A}12$ 在 (0.36,0.44] 和 [0.56,0.64] 的区间内时，两个模糊测试工具之间的差异被认为很小；当 $\hat{A}12$ 在 (0.29,0.36] 和 [0.64,0.71] 的区间内时，两个模糊测试工具之间的差异被认为是中等的；当 $\hat{A}12$ 在 [0,0.29] 和 [0.71,1] 的区间内时，两个模糊测试工具之间的差异被认为很大。

3.3.3.2 实验二：漏洞检测能力评估

本节评估了 MemLock 相较于其它对比基准在现实世界的应用程序中发现内存消耗漏洞的能力。由于模糊测试工具所报告的崩溃是根据边覆盖来判断的，实际上会出现很多触发相同漏洞的崩溃，本实验将发现的所有崩溃归类为具体的真实漏洞，并进行人工检验。由此，本实验可以进一步评估各个模糊测试工具发现特定漏洞所需要消耗的时间。一般来说，发现特定漏洞所需的平均时间越短，则表明模糊测试工具的发现漏洞的能力越强，效率越高。

表 3-3描述了 MemLock 以及其它六个模糊测试工具在 10 次重复实验中发现特定漏洞所消耗的平均时间。在所有的实验对象中共包含 34 个独特的内存消耗漏洞，MemLock 在其中的 25 个（73.5%）漏洞中发现漏洞所需要的时间最短。MemLock 平均需要 5.4 小时就能找到一个特定的漏洞，而 AFL、AFlfast、PerfFuzz、FairFuzz、Angora 和 QSYM 的平均耗时分别是 MemLock 的 2.15、2.15、2.20、2.69、3.76 和 2.07 倍。因此，实验数据表明 MemLock 发现真实的内存消耗漏洞所消耗的时间较其它六个对比基准明显要短。MemLock 在 24 小时内发现了 34 个独特漏洞中的 33 个，而 AFL、AFLfast、PerfFuzz、FairFuzz、Angora 和 QSYM 仅分别发现 26、28、20、17、6 和 25 个特定漏洞，由此可见 MemLock 相比其它对比基准发现的特定漏洞的数量要更多。值得注意的是，MemLock 可以发现其它六个对比基准都未在 24 小时内发现的漏洞，例如 *mjs*、*nm* 和 *flex* 中的三个独特漏洞（即 issue#106、CVE-2018-18701 和 CVE-2019-6293）仅被 MemLock 报告。因此，实验数据表明 MemLock 在发现真实的内存消耗漏洞方面十分有效。

同样的，本节也针对发现特定漏洞所消耗的时间进行了 Vargha Delaney 度量和曼-惠特尼 U 检验，其评价标准与第 3.3.3.1节一致。在表中的 204 组 $\hat{A}12$ 值中，有 139（68.1%）组数据既在 $\hat{A}12$ 上超过了传统的效应标准（0.71），又在 P 值上小于 0.05。因此，可以判定 MemLock 在发现特定的漏洞方面明显优于其他 6 个业界先进的模糊测试技术。

案例分析：为了进一步阐述 MemLock 在发现内存消耗漏洞方面有优越性背后的原因，本文以 CVE-2019-6293 漏洞作为研究案例进行详细说明。CVE-2019-6293 漏洞是词法分析器的生成器 *flex* 中的一个不受控的递归调用漏洞。由于 *flex* 生成的词法分析器必须提供“开始”状态和“结束”状态。mark_start_as_normal 函数将词法分析器状态机中的每个“开始”状态标记为“正常”状态，而“开始”状态是第一个状态的 ε 闭包。如果有一个状态可以从第一个状态通过 ε 路径到达，则 mark_start_as_normal 函数将递归调用自身。本章人工调查了 MemLock 对种子输入

表 3-3　发现内存消耗漏洞所需的时间（单位：小时）

程序	漏洞标识符	漏洞类型	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		Angora		QSYM	
			耗时	耗时	$\hat{A}_{12}$	耗时	$\hat{A}_{12}$	耗时	$\hat{A}_{12}$	耗时	$\hat{A}_{12}$	耗时	$\hat{A}_{12}$	耗时	$\hat{A}_{12}$
mjs	issue#58	CWE-674	0.5	0.3	0.25	0.4	0.25	0.2	**0.13**	0.4	0.25	T/O	**1.00**	0.3	0.22
	CVE-2020-18392	CWE-674	13.7	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
cxxfilt	CVE-2018-9138	CWE-674	0.3	7.2	**1.00**	10.1	**1.00**	0.5	**0.81**	T/O	**1.00**	T/O	**1.00**	3.3	**1.00**
	CVE-2018-9996	CWE-674	T/O	16.5	**0.00**	T/O	0.50	T/O	0.50	T/O	0.50	T/O	0.50	T/O	0.50
	CVE-2018-17985	CWE-674	0.2	1.1	**1.00**	4.5	**1.00**	0.2	0.63	1.9	**1.00**	T/O	**1.00**	1.4	**1.00**
	CVE-2018-18484	CWE-674	0.2	1	**1.00**	4.5	**1.00**	0.2	0.63	8	**1.00**	T/O	**1.00**	1.4	**1.00**
	CVE-2018-18700	CWE-674	0.2	1.2	**1.00**	4.6	**1.00**	0.3	**0.75**	12.6	**1.00**	T/O	**1.00**	1.4	**1.00**
nm	CVE-2018-12641	CWE-674	2.6	19.1	**1.00**	12.6	**1.00**	12.2	**0.88**	T/O	**1.00**	T/O	**1.00**	12.8	**0.88**
	CVE-2018-17985	CWE-674	10.4	18.2	**0.81**	11.9	0.56	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	13.3	0.63
	CVE-2018-18484	CWE-674	9.9	16.4	**0.84**	17.1	**0.84**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	14	0.75
	CVE-2018-18700	CWE-674	9.6	14.9	0.63	17.8	**0.88**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
	CVE-2018-18701	CWE-674	13.9	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
	CVE-2019-9070	CWE-674	18.4	15.6	0.56	13.9	0.44	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	15.8	0.56
	CVE-2019-9071	CWE-674	12.4	T/O	**0.88**	14	0.69	T/O	**0.88**	T/O	**0.88**	T/O	**1.00**	T/O	**0.88**
nasm	CVE-2019-6290	CWE-674	0.9	T/O	**1.00**	19	**1.00**	9	**1.00**	T/O	**1.00**	T/O	**1.00**	17.6	**1.00**
	CVE-2019-6291	CWE-674	1.5	9	**0.94**	14	**1.00**	8.7	**1.00**	T/O	**1.00**	T/O	**1.00**	7.5	**1.00**
flex	CVE-2019-6293	CWE-674	5.4	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
yaml-cpp	CVE-2019-6292	CWE-674	0.4	T/O	**1.00**	18.4	**1.00**	0.9	**0.81**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
	CVE-2018-20573	CWE-674	6.1	T/O	**0.88**	T/O	**0.84**	12.4	**0.84**	T/O	**0.84**	T/O	**1.00**	T/O	**0.84**
libsass	CVE-2018-19837	CWE-674	1.6	13.3	**0.88**	10.5	**0.88**	1.8	0.63	8.5	**0.88**	T/O	**1.00**	5	**0.81**
	CVE-2018-20821	CWE-674	0.1	5.7	**1.00**	6.5	**1.00**	0.1	0.50	9.5	**1.00**	T/O	**1.00**	7.4	**1.00**
	CVE-2018-20822	CWE-674	15.6	14.3	0.50	19.5	0.56	14.6	0.47	11.3	0.56	0.92	**0.00**	10.5	0.44
yara	CVE-2017-9438	CWE-674	0.2	0.9	**1.00**	4.3	**1.00**	0.61	**0.91**	5.3	**1.00**	T/O	**1.00**	0.8	**1.00**
readelf	CVE-2017-15996	CWE-789	0.2	0.3	**0.86**	0.2	0.68	0.5	**0.92**	0.3	0.68	T/O	**1.00**	0.3	**0.96**
exiv2	CVE-2018-4868	CWE-789	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.5	0.1	0.50
bento4	CVE-2018-20186	CWE-789	0.4	0.4	0.50	0.4	0.50	0.4	0.50	0.4	0.50	0.1	**0.00**	0.4	0.50
	CVE-2019-7698	CWE-789	14.6	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	0.5	**0.00**	T/O	**1.00**
libming	CVE-2019-7581	CWE-789	0.6	0.8	0.68	1.4	**0.80**	2	**0.88**	0.4	0.36	T/O	**1.00**	1.6	**0.80**
	CVE-2019-7582	CWE-789	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50
	issue#155	CWE-789	1.4	1	0.30	1.3	0.36	1.4	0.40	1.2	0.42	T/O	**1.00**	1.6	0.64
openjpeg	CVE-2019-6988	CWE-789	7.8	15.1	**0.86**	11.1	**0.84**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	15.3	**0.81**
	CVE-2017-12982	CWE-789	4.5	11.4	**0.72**	10	0.60	T/O	**1.00**	11.9	0.64	T/O	**1.00**	10	0.50
jasper	CVE-2016-8886	CWE-789	4.1	17	**0.88**	22.3	**1.00**	T/O	**1.00**	10.3	0.52	T/O	**1.00**	18.2	**0.88**
	issue#207	CWE-789	1.7	2.2	0.62	3.6	0.68	T/O	**1.00**	2.2	0.68	15.9	**1.00**	4	0.64
平均耗时（单位：小时）			**5.4**	**11.6 (2.15×)**		**11.6 (2.15×)**		**11.9 (2.20×)**		**14.5 (2.69×)**		**20.3 (3.76×)**		**11.2 (2.07×)**	
发现的独特漏洞的数量			**33**	**26 (+26.9%)**		**28 (+17.9%)**		**20 (+65.0%)**		**17 (+94.1%)**		**6 (+450.0%)**		**25 (+32.0%)**	

* CWE-674 代表“不受控的递归调用”漏洞，CWE-789 代表“不受控的内存分配”漏洞，CVE-401 为“内存泄露”漏洞；T/O 代表超时，意味着 10 次 24 小时的重复实验中，模糊测试工具没有找到漏洞；在计算“平均耗时”时，T/O 被替换为 24 小时；$\hat{A}_{12}$ 被加粗则代表着相应的 *Mann-Whitney U* 检验结果说明 MemLock 与其对比的工具的发现漏洞所需的时间差异显著。

的变异记录，并从中确定了几个关键的变异操作。MemLock 通过 *havoc* 变异算子（即对原文件进行大量随机变异）使得测试用例能触发 mark_start_as_normal 函数递归调用自身，然后再多次经过 *splice* 变异算子（即将两个文件拼接起来得到一个新文件）使得递归调用的深度不断增加，最终触发栈内存溢出崩溃。比较有趣的一个现象是，MemLock 平均只需要 5.4 小时就可以发现这个漏洞，而其他模糊测试工具在 24 小时内都未发现该漏洞。在图 3-6中还可以看到各个模糊测试工具能造

成*flex*调用栈上函数个数的峰值。AFL 不会保留能触发超过 5000 次递归调用的种子，因为这些种子输入不会增加覆盖率，而 AFL 仅保留能触发新的边覆盖的种子。与 AFL 相比，MemLock 会有意保留能增加调用栈上函数个数的峰值的种子，从而这些种子经过一系列变异以后更容易生成能触发栈内存溢出崩溃的测试用例。这也解释了为什么 MemLock 可以找到 CVE-2019-6293 漏洞，而其它模糊测试工具无法在全部 10 次重复实验中检测到它。

实验结论：基于对表 3-3和研究案例的详尽分析，可以判定 MemLock 相比其它 6 个业界先进的模糊测试技术在发现真实的内存消耗漏洞方面的能力更强。相比 AFL、AFLfast、PerfFuzz、FairFuzz、Angora、QSYM 方法，MemLock 发现的内存消耗漏洞数量至少要多 17.9%，并且在发现内存消耗漏洞的速度上更快，至少是其它方法的 2.07 倍。

3.3.3.3 实验三：内存泄漏大小评估

内存泄漏漏洞不同于不受控的递归调用和不受控内存分配漏洞，因为内存泄漏可能并不会立即导致程序崩溃，只有当泄漏的内存大小达到足够的量，才会使程序运行速度减缓甚至停止工作，这在长时间运行的服务程序中较为常见。为了评估模糊测试技术在检测内存泄漏漏洞方面的有效性，本章统计了 24 小时内各个模糊工具触发的内存泄漏的大小（泄漏的 Bytes 总数），如表 3-4所示。通常来说，单位时间内造成内存泄漏的大小越大，则表明该程序中的内存泄漏漏洞越严重、危害越大。

MemLock 在 24 小时的执行中发现的内存泄漏的 Bytes 总数远远大于其它模糊测试工具，与其它六个模糊测试工具相比，MemLock 触发的内存泄漏的 Bytes 总数要多于 234% 到 3753163% 不等。这主要是因为 MemLock 采用内存消耗导向的模糊测试，它试图最大化每个内存分配的值，并以提高生成的测试用例造成的内存消耗。当内存泄漏发生时，这些造成高内存消耗的测试用例通常会导致更多字节的内存泄漏。从统计学意义的角度分析，表 3-4中所有 Vargha Delaney 度量的 $\hat{A}12$ 值都超过了传统的效应标准（0.71），并且所有曼-惠特尼 U 检验的 P 值都远远小于 0.05，这也说明了 MemLock 在发现内存泄漏大小的能力上较其它 6 个业界先进的模糊测试技术更强。

实验结论：基于表 3-4，通过分析各个模糊测试工具在 24 小时的运行时间内平均造成内存泄漏的 Bytes 总数，可以判定 MemLock 生成的测试用例能够触发更大的内存泄漏。

表 3-4　内存泄露的 Bytes 总数

程序	工具	内存泄露大小（Bytes）	提升（百分比）	*p*-value	$\hat{A}_{12}$
bento4	MemLock	**52,709,574**	-	-	-
	AFL	151,862	+34609%	0.0061	1.00
	AFLfast	1,233,255	+4174%	0.0061	1.00
	PerfFuzz	105,984	+49633%	0.0061	1.00
	FairFuzz	1,910,466	+2659%	0.0061	1.00
	Angora	141,512	+37147%	0.0060	1.00
	QSYM	15,784,847	+234%	0.0061	1.00
libming	MemLock	**176,320,785**	-	-	-
	AFL	4,869,594	+3521%	0.0061	1.00
	AFLfast	2,535,212	+6855%	0.0061	1.00
	PerfFuzz	47,044,964	+257%	0.0061	1.00
	FairFuzz	828,742	+21176%	0.0061	1.00
	Angora	4,698	+3753163%	0.0060	1.00
	QSYM	1,219,093	+14363%	0.0061	1.00
jsaper	MemLock	**2,372,844,732**	-	-	-
	AFL	56,018,839	+4136%	0.0061	1.00
	AFLfast	48,403,244	+4802%	0.0061	1.00
	PerfFuzz	6,229,898	+37988%	0.0061	1.00
	FairFuzz	56,788,235	+4096%	0.0061	1.00
	Angora	191,907,941	+1136%	0.0105	0.98
	QSYM	38,244,568	+6104%	0.0061	1.00

3.3.3.4　实验四：种子造成的内存消耗评估

由于 MemLock 旨在生成能造成更多内存消耗的测试用例，本实验通过对比 MemLock、AFL、AFLfast、PerfFuzz、FairFuzz、Angora 和 QSYM 的种子池中的种子造成内存消耗的频率分布，来验证 MemLock 的方法机制和策略是否能真的帮助生成能造成更大内存消耗的测试用例。

如图 3-6所示，MemLock 的种子池中包含了大量能造成高内存消耗的测试用例，而且 MemLock 的种子分布主要集中在内存消耗较高的区域（递归调用深度较深或堆内存分配较大），而其它模糊工具的种子分布主要集中在内存消耗较低的区域。例如，对于含不受控制的递归调用漏洞的程序（*nm*、*nasm*、*flex* 和 *yara*），MemLock 生成的大量测试用例都能使调用栈深度达到 20,000 个函数调用以上，而 PerfFuzz 只有极少量的测试用例能使调用栈深度达 20,000 以上，AFL/AFLfast 甚至很难生成使调用栈深度超过 10,000 以上的测试用例。对于含不受控制的内存分配漏洞的程序（*readelf*、*openjpeg*、*jasper* 和 *libming*），MemLock 生成的大量测试用例都能造成较大的堆内存分配，而其它模糊测试工具生成测试用例所造成的内存分配远小于 MemLock 。这主要是因为 MemLock 的内存消耗引导的反馈机制有助于逐渐将造成越来越大的内存消耗的测试用例添加/更新到种子池中。实验结果清楚地验证了 MemLock 的方法机制和策略在生成能造成高内存消耗的测试用例方面的有效性。

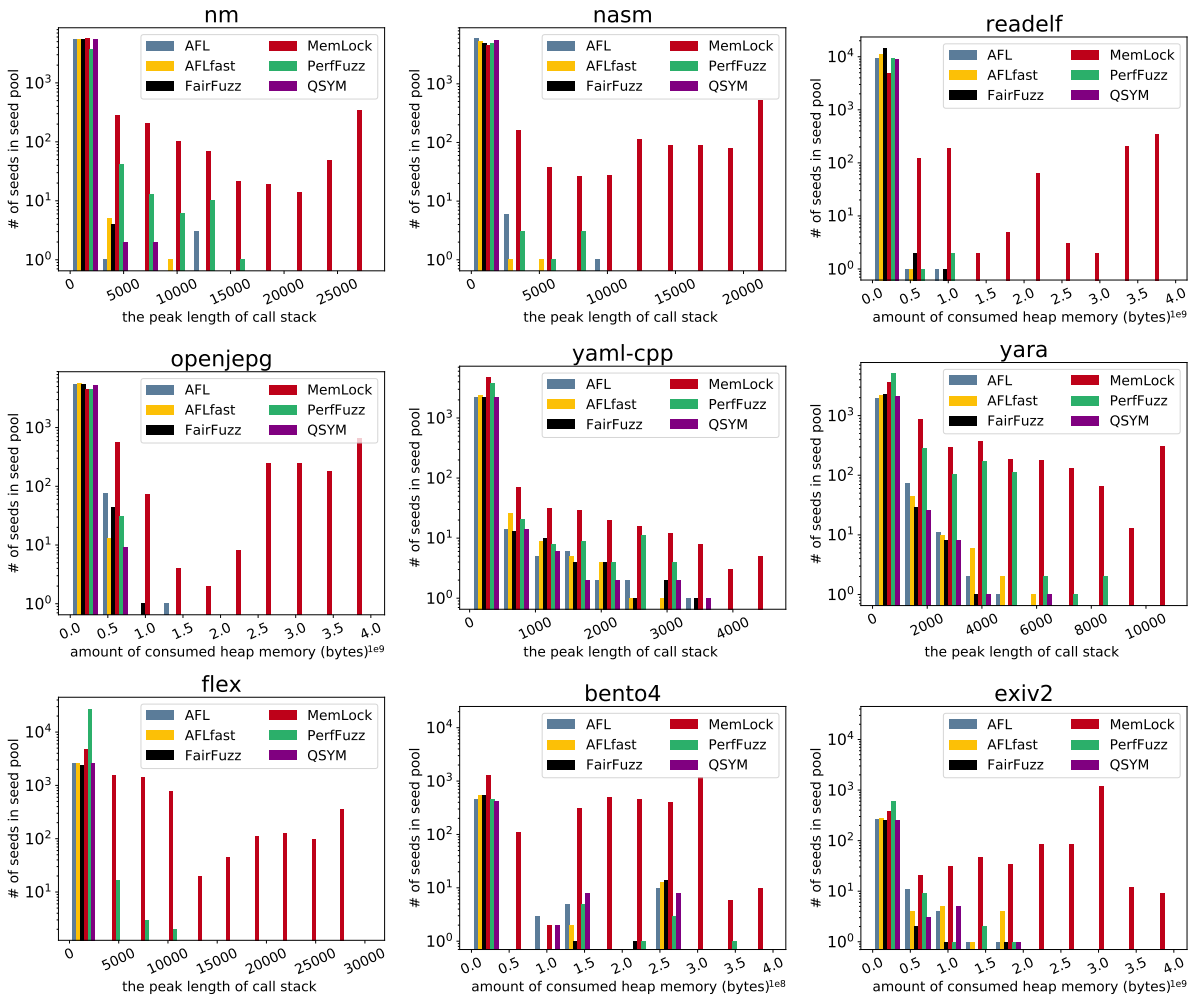


图 3-6　造成不同内存消耗的种子分布

实验结论：基于对图 3-6的详尽分析，本章验证了 MemLock 的内存消耗导向的模糊测试方法的确能指导测试用例的生成，使得造成的内存消耗更大。

3.3.3.5　执行开销与覆盖率评估

实验一到实验四主要表明了 MemLock 在发现内存消耗漏洞方面的有效性，本节的实验旨在评估 MemLock 的其它次要性能指标（例如，执行时性能开销、代码覆盖率等）。由于 MemLock 旨在发现内存消耗漏洞，PerfFuzz 专注于发现时间复杂性漏洞，而 AFL、AFLfast、FairFuzz 等基于代码覆盖检测一般性的内存安全问题。各个模糊测试技术在测试的目标和针对性上有一定的区别，因此尽管 MemLock 在检测内存消耗漏洞方面显著优于其它对比基准，但在其它性能指标方面仍可能会落后于其它对比基准。

MemLock 有意保留能造成大量内存消耗的种子输入，这可能会降低 MemLock 识别其它类型漏洞的能力，故本节还评估了各个模糊测试技术在发现其它类型漏洞方面的能力。在表 3-2的所有实验对象中，MemLock 、AFL、AFLfast、PerfFuzz、

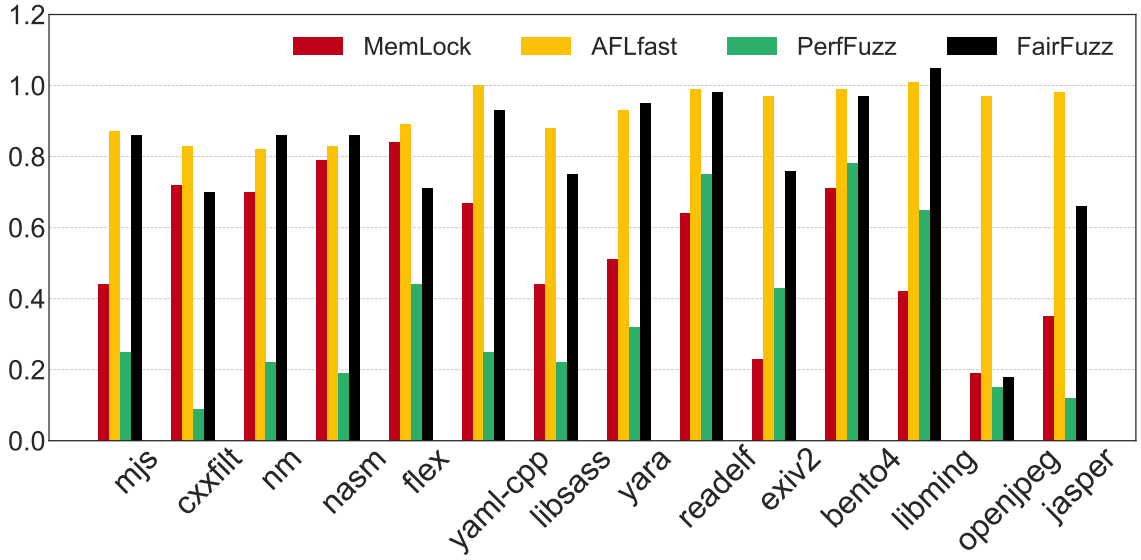


图 3-7　相较于 AFL 的执行速度的比值

FairFuzz、Angora 和 QSYM 分别发现 77、239、228、189、276、343 和 236 种其它类型的崩溃。由此可见，MemLock 在发现非内存消耗漏洞方面的能力的确稍弱于其它代码覆盖导向的模糊测试技术。

另一方面，能造成高内存消耗的测试用例通常需要的执行时间也更久，因此本节也评估了各个模糊测试工具的执行速度（单位时间内执行的测试用例的数量）。图 3-7显示了各个模糊测试工具的执行速度相较于 AFL 的执行速度的比值。总体来说，MemLock 的执行速度相比 AFL、AFLFast、FairFuzz 等工具要慢，但仍然比 PerfFuzz 要快得多，MemLock 的执行速度大概在 AFL 的 20% 到 80% 之间。PerfFuzz 的执行速度是所有模糊测试工具中最慢的，这主要是由于 PerfFuzz 倾向保留执行指令数较多的输入。由此可见，MemLock 的确会产生一定的运行时开销，但该开销在合理的范围内。

考虑到代码覆盖率情况，AFL、AFLfast、FairFuzz 是基于代码覆盖导向的模糊测试技术，QSYM 是一种结合符号执行技术的混合模糊测试技术，它们在实现高代码覆盖率方面有一定的优势，而实现较高的代码覆盖率并不是 MemLock 的主要目标。本节也评估了各个模糊测试工具对不同实验对象的代码覆盖率情况，图 3-8显示了各个模糊测试工具在 10 次 24 小时的重复实验中代码覆盖率随时间的平均增长趋势。可以看到 MemLock 实现的代码覆盖率随时间的增长而增长，但在增长速度和最终实现的代码覆盖率上都不如其它基于代码覆盖导向的模糊测试工具。当运行时间达到 24 小时，MemLock 基本能实现与 AFL 相接近的代码覆盖率情况。PerfFuzz 在大部分情况下（除了 *yara* 和 *bento4*）实现的代码覆盖率情况都不如 MemLock 。由此可见，代码覆盖率与内存消耗漏洞的发现率之间并没有直接的关

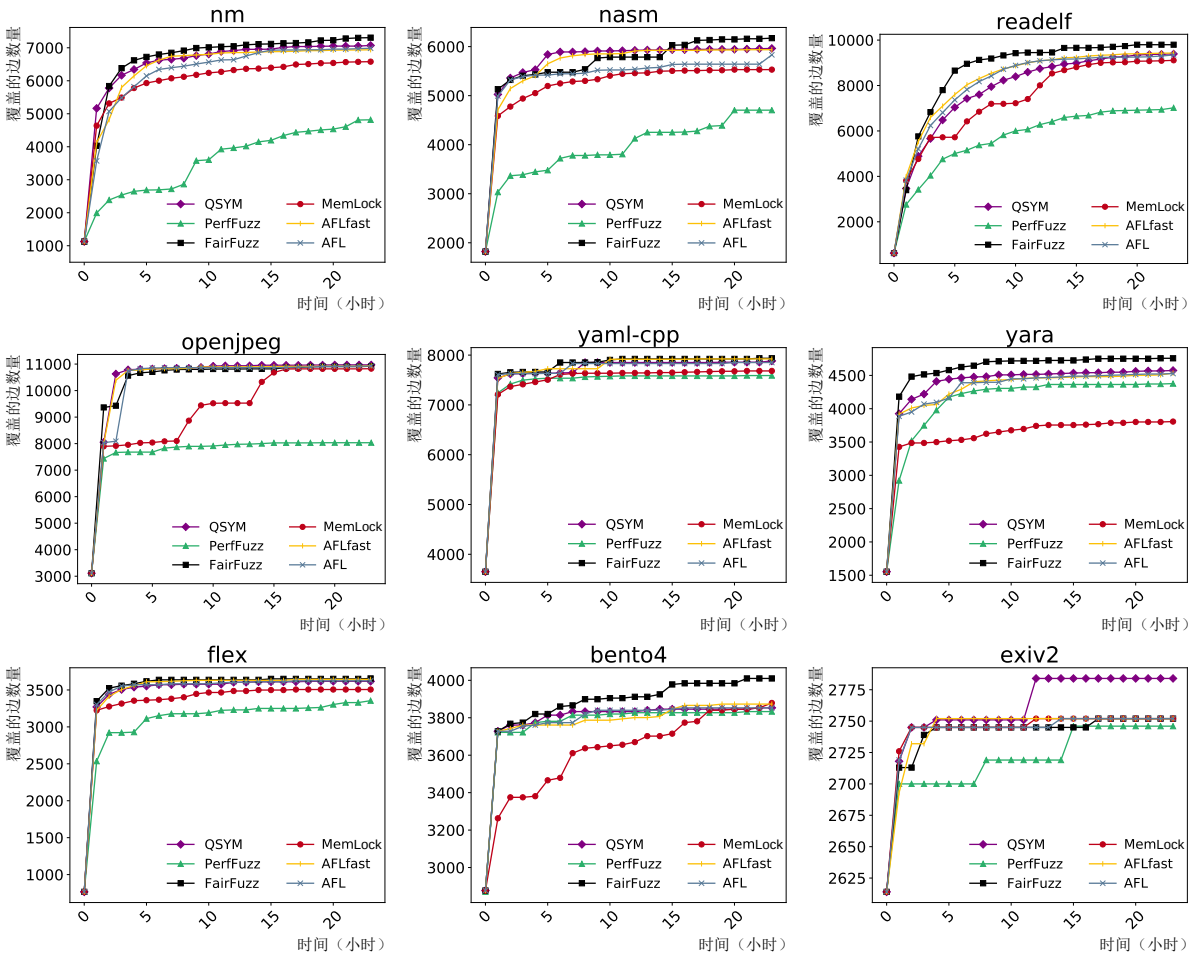


图 3-8　覆盖率评估

系，这也说明了 MemLock 的内存消耗导向的模糊测试在发现内存消耗漏洞方面起着重要的作用。

3.3.3.6　发现的新漏洞

本章还将 MemLock 应用于现实中被广泛使用的真实应用程序（例如 *Mjs*、*Exiv2*、*Flex*、*Yaml-cpp*、*NASM*、*Binutils*、*Bento4*、*Elfutils* 和 *Tinyexr* 等），发现了大量安全攸关的内存消耗漏洞。这些新发现的漏洞都是以前未被披露的未知漏洞，因此本文作者线上提交了漏洞报告，并帮助开发和维护人员一起定位和修复漏洞。MITRE 组织为作者提交的漏洞分配了 28 个 CVE ID，如表 3-5所示，在这 28 个 CVEs 之中有 20 个 CVEs 属于不受控的递归调用漏洞（表中用 CWE-674 表示），有 6 个 CVEs 是不受控的内存分配漏洞（表中用 CWE-789 表示），还有 2 个 CVEs 属于内存泄漏漏洞（表中用 CWE-401 表示）。攻击者可能会提供一些精心设计的输入，通过引起过度的内存消耗来触发这些漏洞，从而发起拒绝服务（DoS）攻击。通过作者的漏洞报告和漏洞披露之后，这些漏洞被开发者积极地修复；在撰写本文时，其中 25 个漏洞已经被修复。值得注意的是，一部分内存消耗漏洞仅能被 MemLock 成功发

表 3-5　MemLock 新发现的内存消耗漏洞（28 CVEs）

CVE ID	程序	漏洞类型	CVE ID	程序	漏洞类型
CVE-2020-36375	*MJS v1.20.1*	CWE-674	**CVE-2019-6291**	*NASM v2.14.03*	CWE-674
CVE-2020-36374	*MJS v1.20.1*	CWE-674	**CVE-2019-6290**	*NASM v2.14.03*	CWE-674
CVE-2020-36373	*MJS v1.20.1*	CWE-674	**CVE-2018-18701**	*Binutils v2.31*	CWE-674
CVE-2020-36372	*MJS v1.20.1*	CWE-674	**CVE-2018-18700**	*Binutils v2.31*	CWE-674
CVE-2020-36371	*MJS v1.20.1*	CWE-674	**CVE-2018-18484**	*Binutils v2.31*	CWE-674
CVE-2020-36370	*MJS v1.20.1*	CWE-674	**CVE-2018-17985**	*Binutils v2.31*	CWE-674
CVE-2020-36369	*MJS v1.20.1*	CWE-674	**CVE-2020-18899**	*Exiv2 v0.27*	CWE-789
CVE-2020-36368	*MJS v1.20.1*	CWE-674	**CVE-2019-7704**	*Binaryen v1.38.22*	CWE-789
CVE-2020-36367	*MJS v1.20.1*	CWE-674	**CVE-2019-7698**	*Bento4 v1.5.1-624*	CWE-789
CVE-2020-36366	*MJS v1.20.1*	CWE-674	**CVE-2019-7148**	*Elfutils v0.175*	CWE-789
CVE-2020-18392	*MJS v1.20.1*	CWE-674	**CVE-2018-20652**	*Tinyexr v0.9.5*	CWE-789
CVE-2020-18898	*Exiv2 v0.27*	CWE-674	**CVE-2018-18483**	*Binutils v2.31*	CWE-789
CVE-2019-6293	*Flex v2.6.4*	CWE-674	**CVE-2018-20657**	*Binutils v2.31*	CWE-401
CVE-2019-6292	*Yaml-cpp v0.6.2*	CWE-674	**CVE-2018-20002**	*Binutils v2.31*	CWE-401

现，而这些漏洞是其它模糊测试技术完全发现不了的，例如CVE-2020-18392、CVE-2019-6293、CVE-2018-18701，关于这几个漏洞的详细实验数据也列举在了表3-3中。本章发现的大量新漏洞也说明了MemLock在实践中是切实有效和可行的。

3.3.4　有效性威胁分析

基于以上实验，本章从以下两个方面来分析影响本章实验结论有效性的因素：

内部因素：众所周知，程序动态分析的结果部分依赖于程序输入，它很大程度会因为错过一些程序路径而造成漏报。与大多数基于代码覆盖导向的模糊测试技术[65–67]类似，MemLock难以覆盖一些较难通过随机变异突破的分支（例如，魔术字符串），对于未覆盖分支中的内存消耗漏洞，MemLock无法保证其有效性。采用一些程序分析技术（例如符号执行）可能有助于缓解这种威胁。此外，模糊测试技术在生成程序输入时具有一定的随机性，尽管本章为了减小这种随机性对实验结论造成的影响进行了长时间的重复实验，并进行了统计学意义上的分析，但这种随机性仍然可能对实验结论存在一定的影响，接下来，可以增加重复实验进行的次数，以提高本章结论的可靠性。

外部因素：影响本章实验结果的外部因素主要是选取的实验对象的样本误差。尽管本章选取了14个不同规模及功能多样的真实应用程序来评估MemLock，其包含34个已知的内存消耗漏洞，并将MemLock与其它六个业界先进的模糊测试技术进行比较。然而，本章选取的实验对象和对比基准可能包括一定的样本偏差，仍不能保证本章结论对所有真实的应用程序都有效。因此，下一步将寻找更多具

有不同特征且来自不同领域的真实应用程序，以提高本章结论的普适性。

3.4 本章小结

本章针对内存消耗漏洞的检测问题，提出了一种融合程序分析与模糊测试的内存消耗漏洞检测方法 MemLock。MemLock 旨在对三类内存消耗漏洞进行高效的自动化检测，这三类内存消耗漏洞为：a. 不受控的递归调用；b. 不受控的堆内存分配；c. 内存泄漏。MemLock 首先使用轻量级的静态分析确定与内存消耗相关的语句和操作，并基于这些语句和操作进行插桩以在运行时收集内存消耗信息，再通过内存消耗导向的模糊测试，结合种子动态更新策略，自动化生成能造成过量内存消耗的测试用例，以有效检测内存消耗漏洞。本章实现了 MemLock 工具原型，并使用了 14 个现实世界中被广泛使用的开源程序对其进行实验评估。实验结果表明，MemLock 在发现内存消耗漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术。相比 AFL、AFLfast、PerfFuzz、FairFuzz、Angora、QSYM 方法，MemLock 发现的内存消耗漏洞数量至少要多 17.9%，并且 MemLock 在发现内存消耗漏洞的速度上更快，至少是其它方法的 2.07 倍。此外，MemLock 也在真实应用程序上新发现了 28 个内存消耗漏洞（28 CVEs）。

本章的主要研究成果撰写的论文《MemLock: Memory Usage Guided Fuzzing》已在软件工程领域的顶级国际会议 ICSE ’20 上发表。本章实现的 MemLock 工具原型和相关实验数据已通过 ICSE’20 的“Artifacts evaluated available and reusable”的认证。

第 4 章　融合程序分析与模糊测试的内存时序漏洞检测技术

4.1　引言

4.1.1　研究背景

程序在使用内存资源时必须要遵守内存使用的安全时序规则[118,119]（Temporal Memory Safety），如果违反了内存使用的安全时序规则，则可能导致数据损坏、信息泄漏，或者遭受拒绝服务和任意代码执行攻击[174,175]。典型的由于违反内存使用的安全时序规则而产生的漏洞包括释放后使用（UaF）和双重释放（DF）。根据近期的报告[90,176]，在 NVD 数据库中，大约 80% 的 UaF 漏洞被评为高严重性或严重性漏洞。相反，只有大约 50% 的堆缓冲区溢出漏洞被视为高严重性漏洞。与其他漏洞（例如堆/栈缓冲区溢出漏洞）相比，UaF 这类由于违反时序规则而产生的漏洞通常更难检测。主要原因是，要成功触发这类漏洞，需要按照特定的顺序执行一系列内存操作，即首先分配内存，然后释放内存，最后再对这块内存地址进行解引用操作。这些操作可能不集中在某一代码块中完成，依序执行的条件十分隐蔽，且需要跟踪较长的操作序列才能发现该漏洞，这使得检测内存时序漏洞相当具有挑战性。

内存使用的安全时序规则本质上可以看作是程序的一种类型状态（Typestate）属性[177]，当程序的内存操作违反这种类型状态属性，就会产生内存时序漏洞。内存使用的安全时序规则可用有限自动机模型来表示，如图 4-1所示。一般来说，对内存的操作通常可以归纳为如下三种：内存分配操作（*malloc*）、内存使用操作（*use*）和内存释放操作（*free*）。相应的内存资源至少包含四个状态：初始状态（*init*）、已分配状态（*live*）、释放状态（*dead*），以及错误状态（*error*）。这四个状态用于表示上述三种操作所引起的内存状态变化。程序可以通过分配操作向操作系统申请一块未分配的内存空间的使用权，成功申请到内存空间后，程序可以通过使用操作使用该内存空间。在使用完内存资源后（即内存处于已分配状态），程序必须释放该内存资源。不正确的内存使用操作可能导致内存时序漏洞，比如使用一块已释放的内存空间，会导致 UaF 漏洞；再比如释放一块已释放的内存空间，会导致 DF 漏洞。

当前主流的内存时序漏洞的检测方法主要包括两种：静态分析和动态测试。使用静态分析技术分析大规模程序时通常面临着路径爆炸、资源消耗激增等问题，且静态分析通常采用上近似分析，倾向于确保完备性，因此误报率较高。根据相

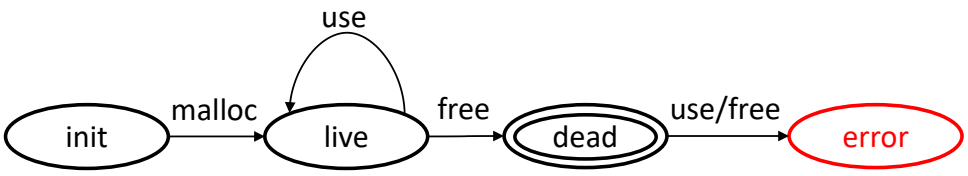


图 4-1　内存使用的安全时序规则的自动机模型表示

关研究工作[178,179]的介绍，检测 UaF 和 DF 这类内存时序漏洞的相关研究工作主要集中在动态测试方面。这主要是因为动态方法易于检测同一指针的副本，也称为别名。换句话说，使用动态方法时，可以直接访问内存中的值，这种能力对于代码分析来说是非常重要的。虽然动态方法能够获得较高的准确性，但同时也因为难以覆盖所有可能的程序路径而出现一些漏报。由于代码覆盖导向的模糊测试技术能有效探索不同的代码路径，该技术通常可以结合传统的程序动态分析方法[27,92]来检测内存时序漏洞。实际上，当前主流的代码覆盖导向的模糊测试技术对于检测内存时序漏洞这类漏洞仍存在一定的局限性。以检测 UaF 漏洞为例，虽然代码覆盖导向的模糊测试技术能有效覆盖程序中不同的分支，但要发现 UaF 漏洞不仅需要覆盖到相应内存使用操作（申请、释放、使用）所在的分支，还需要按照特定的时序（申请→释放→使用）去执行这些操作，这对现有的代码覆盖导向的模糊测试技术来说是相当困难的。

4.1.2　现存问题

本节通过一个现实世界中真实的 UaF 漏洞案例来说明当前主流的程序分析技术与模糊测试技术在检测内存时序漏洞方面的局限性，并引出本章的主要工作。

图 4-2中的代码片段是漏洞 CVE-2018-20623 的简化版本，该漏洞来自 GNU 开源工具集 *Binutils v2.31* 中的 *readelf* 程序。该程序的控制流程图（CFG）如图 4-3(a)所示，CFG 中的每个结点都被用数字标记，对应相应的源代码行号。当按照特定的时序执行某些语句时（即第 4 行→第 7 行→第 10 行→第 14 行），会触发 UaF 漏洞。具体而言，该程序接收一段字符串作为输入（第 3 行），并申请了两块内存空间（第 4-5 行），指针 ptr1 指向第 4 行分配的内存空间，而指针 ptr2 指向第 5 行分配的内存空间，当输入满足一定的约束时，ptr1 和 ptr2 变成了别名（第 7 行），同时指向第 5 行分配的内存空间。同样，当程序的输入满足另一种约束时，ptr1（和 ptr2）指向的内存空间被释放（第 10 行）。当程序执行到第 14 行时，ptr2 被访问，但第 14 行 ptr2 访问的内存空间实际上是在第 10 行被释放的内存空间，因此该操作会触发 UaF 漏洞。

```
1  void main() {
2    char buf[7];
3    read(0, buf, 7)
4    char* ptr1 = malloc(8);
5    char* ptr2 = malloc(8);
6    if(buf[5] == 'e')
7      ptr2 = ptr1;
8    if(buf[3] == 's')
9      if(buf[1] == 'u')
10       free(ptr1);
11   if(buf[4] == 'e')
12     if(buf[2] == 'r')
13       if(buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

图 4-2　一个从 CVE-2018-20623 漏洞简化而来的启发性示例（来自 *Binutils v2.31.1*）

现有的主流静态分析/扫描工具 Infer[24] 和 Cppcheck[48] 都未能成功发现这个 UaF 漏洞。这两个静态分析工具都是面向大规模程序的可扩展静态分析/扫描工具。Infer 工具基于分离逻辑[49] 和 Bi-abduction[50] 实现，擅长发现空指针解引用漏洞，但不做复杂的路径可达性分析，未能成功发现该 UaF 漏洞；而 Cppcheck 主要基于模式匹配发现漏洞，同样难以发现路径约束如此复杂的 UaF 漏洞。知名的模型检测工具 CBMC[165]，通过遍历程序的完整状态空间来检测 UaF 漏洞，但仅能用于规模较小的程序，难以应用于 *Binutils* 这类真实的应用程序。动态运行时检测工具 ASAN[27] 可以在运行时检测 UaF 漏洞，但仅仅适用于提供了能触发 UaF 的测试用例的情况。

当前代码覆盖导向的模糊测试工具（如 AFL[30]、AFLFast[63]）可以对已有的测试用例进行变异，自动化地生成大量新的测试用例以对目标程序进行测试，但其生成的测试用例几乎都无法触发该 UaF 漏洞。本节以 AFL 为例，解释代码覆盖导向的模糊测试技术难以发现内存时序漏洞的原因。图 4-3模拟了使用 AFL 对该程序进行模糊测试的过程。假设初始种子为 “aaaaaaa”，AFL 对初始种子进行变异后生成了三个新的测试用例，分别为 “aaaseen”、“aurseaa” 和 “faraeaa”，这三个测试用例的具体执行路径如图 4-3(b) 所示，它们由于都覆盖了新的分支，故会被当作高质量的种子添加到种子池当中。虽然这四个测试用例都没有触发 UaF 漏洞，但是四条程序路径已经覆盖了 CFG 的所有边，于是在它们之后变异生成的测试用例都不会覆盖新的边，也就将被 AFL 所丢弃。考虑到 AFL 是以边的覆盖率为导向，以及当前种子池中保留的路径情况，AFL 很难有效地生成一个能满足同时覆盖 “第 4 行→第 7 行→第 10 行→第 14 行” 的测试用例。类似的，本文第 3章的方法 MemLock 以内存消耗为导向，若在 AFL 种子池的基础上进行变异，MemLock 新生

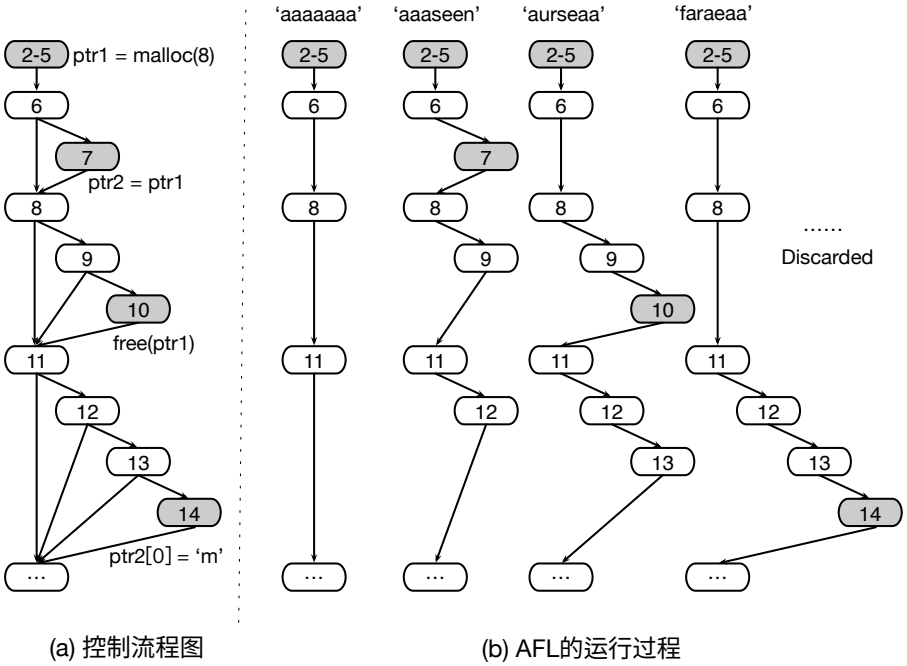


图 4-3　AFL 对图 4-2示例的运行示意图

成的测试用例既不会造成更大的内存消耗，也没有覆盖新的CFG边，故后续也不会再添加新的种子，MemLock 同样难以触发 UaF。

4.1.3　本章主要工作

针对上述问题，本章提出了一种基于程序分析和模糊测试的内存时序漏洞检测方法UAFL。本章将内存使用的安全时序规则建模成具有一定类型状态属性的状态机模型，即内存时序属性，并将内存时序漏洞看作是违反了内存时序属性而导致的漏洞。UAFL 首先对程序做静态分析，找出程序中潜在的违反内存时序属性的操作序列，随后使用找到的操作序列指导模糊测试，逐步生成能够触发违反内存时序属性的测试用例。在此过程中，本章还采用了基于信息流分析的变异优化策略来提升模糊测试的效率。本章实现了UAFL 的工具原型，并使用现实世界中被广泛使用的开源应用程序对UAFL 进行评估。实验结果表明，UAFL 在发现内存时序漏洞方面要显著优于当前业界先进的基于模糊测试的漏洞检测技术。UAFL 在检测内存时序漏洞上相比 AFL、AFLFast、FairFuzz、MOpt、Angora、QSYM 方法发现的漏洞数量上要多于25%，并且实现了至少2.63 倍的提速。此外，UAFL 还在现实世界的主流开源应用程序中发现了7个安全攸关的内存时序漏洞，也证明了其在实际应用的有效性。

本章的其它章节安排如下：首先，第4.2节详细介绍本章提出的内存时序漏洞检测方法，包括整体流程、静态分析与程序插桩、操作序列导向的模糊测试三个部分；接着，第4.3节从静态分析性能、漏洞检测能力、策略有效性、覆盖率等方

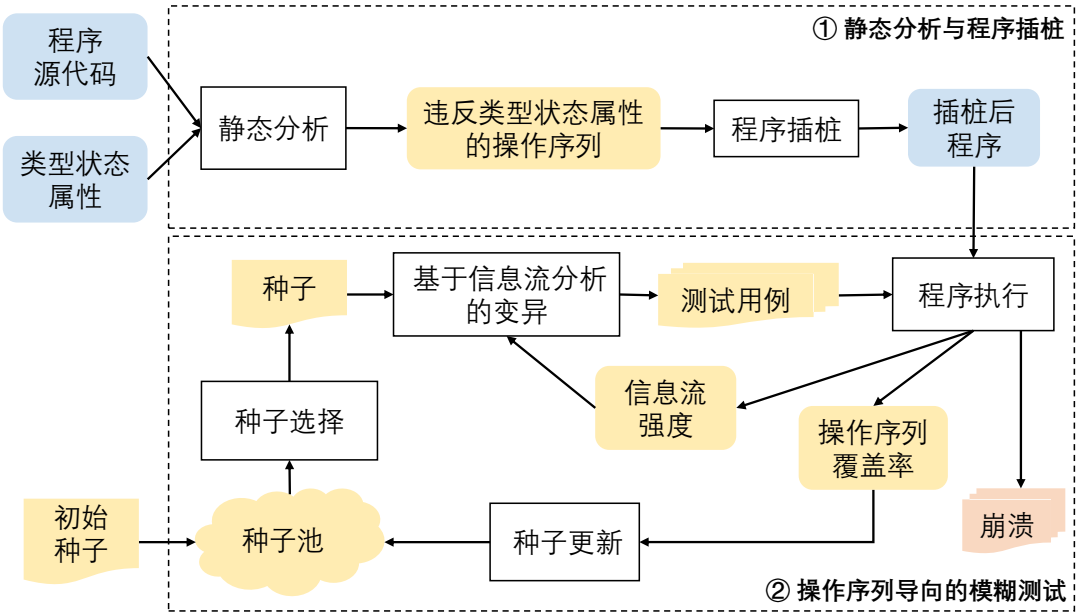


图 4-4　UAFL 的整体流程

面对本章方法进行实验评估与分析；最后，第 4.4节对本章进行总结。

4.2　内存时序漏洞检测方法

4.2.1　整体流程

本章所提的UAFL方法的整体流程如图 4-4所示，主要包括两个阶段：①静态分析与程序插桩；②操作序列导向的模糊测试。

静态分析与程序插桩旨在识别潜在的违反内存时序属性的操作序列，并基于这些操作序列对目标程序进行插桩，以使得程序在运行时能提供操作序列覆盖情况的反馈。该阶段首先通过指针分析识别程序中关于同一内存块的申请、使用、释放操作，再基于路径不敏感的可达性分析找出违反了内存时序属性的操作序列。以图 4-2的示例为例，程序代码第 7 行中，*ptr1* 的值被传递给 *ptr2*，故在其之后的执行过程中，*ptr1* 和 *ptr2* 可能操作同一块内存空间。与这块内存空间相关的操作包括第 4 行的内存申请操作、第 14 行的内存使用操作，以及第 10 行的内存释放操作。然后，UAFL 对示例程序的控制流程图进行搜索，并找到一条违反了内存时序属性的操作序列，该操作序列先执行内存分配操作，然后执行内存释放操作，最后再执行内存使用操作（即第 4 行→第 7 行→第 10 行→第 14 行）。值得注意的是，该操作序列是基于路径不敏感可达性分析被发现的，可能存在误报，即程序实际运行时不一定存在一条可行路径使得该操作序列能完全满足。因此UAFL 将该操作序列视作为一个潜在的 UaF 漏洞的发生条件，通过程序插桩将该信息反馈给阶段②的模糊测试以进一步确认这个潜在的 UaF 漏洞。

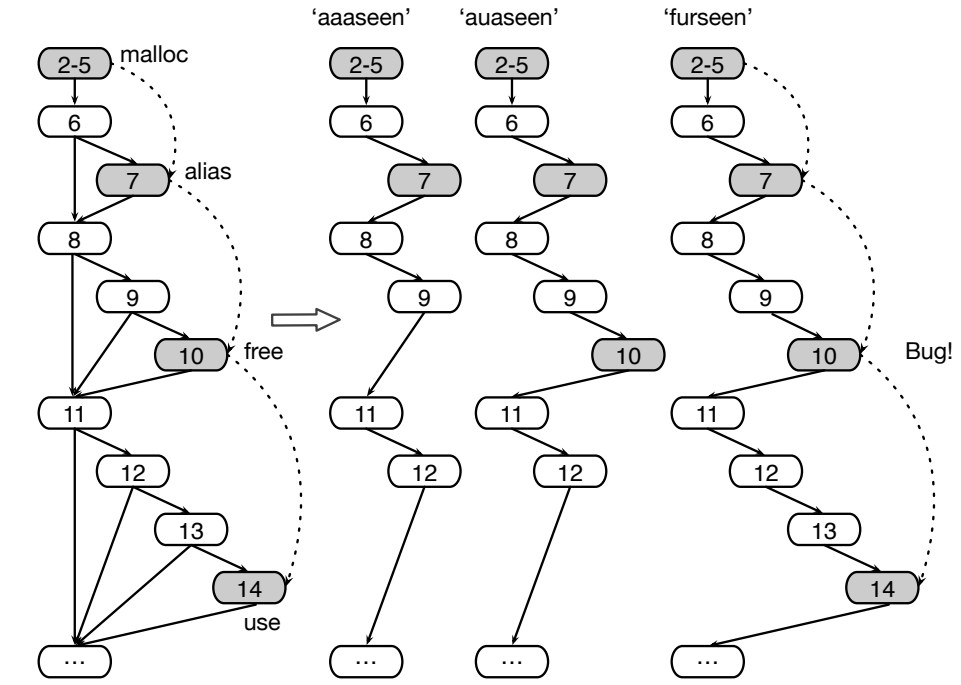


图 4-5　UAFL 对图 4-2示例的运行示意图

模糊测试以插桩后的目标程序作为测试对象，自动化地持续生成测试用例对目标程序进行测试。该过程一旦启动运行，即可在预算的时间范围内自动化地检测漏洞而无需人工干预。UAFL 会充分利用静态分析找出的操作序列，并使用操作序列指导测试用例的生成，使得新生成的测试用例能够逐步覆盖“内存申请→内存释放→内存使用”操作序列。此外，UAFL 还采用了基于信息流分析的变异优化策略来使得变异操作聚焦在与操作序列相关的输入字段上，以提高模糊测试的效率。图 4-5显示了 UAFL 对图 4-2示例进行模糊测试的过程。假设 UAFL 在图 4-3(b)的四个测试用例的基础上继续变异，以操作序列（即“第 4 行→第 7 行→第 10 行→第 14 行”）为指导，UAFL 能够逐渐生成能覆盖整个操作序列的测试用例。例如，基于种子测试用例“aaaseen”进行随机变异，假设第二字符由‘u’变异为‘a’，生成了新测试用例“auaseen”。此测试用例对于 AFL 来说，由于它与种子池中已有的四个测试用例相比（即“aaaaaaa”、“aaaseen”、“aurseaa”和“faraeaa”），没有覆盖新的 CFG 边，因此它会被 AFL 丢弃。然而对于 UAFL 来说，由于该测试用例覆盖了目标操作序列中的一个前缀（即“第 4 行→第 7 行→第 10 行”），UAFL 会将其作为一个高质量的种子添加到种子池当中以待进行进一步的变异。当后续基于种子测试用例“auaseen”进行随机变异时，假设其第一个字符由‘u’变异为‘f’，生成了新测试用例“furseen”，它覆盖了整个目标操作序列（即“第 4 行→第 7 行→第 10 行→第 14 行”），得以触发了 UaF 漏洞。

4.2.2 静态分析与程序插桩

静态分析与程序插桩旨在识别潜在的违反内存时序属性的操作序列（详见第 4.2.2.1节），通过程序插桩使得程序在运行时能提供操作序列覆盖情况的反馈（详见第 4.2.2.2节）。为了更清楚地描述本章方法，本章首先引入一些关于操作序列、类型状态属性、内存时序属性的基本概念：

给定目标程序 P，该程序的控制流程图中的一条路径可用 $P\text{-}path$ 来表示，代表一条从程序 P 的入口到出口所经过的所有语句的有序序列。一条路径 $P\text{-}path$ 中执行的程序语句可能会涉及多个内存对象，对于其中的每一个内存对象，都可以从中提取与该内存对象相关的程序语句，组成一条操作序列[180]。本章只关注与内存的分配、使用、释放操作相关的程序语句，而忽略其它无关的程序语句，操作序列的形式化定义如下。

定义 4.1 (操作序列). 给定一条程序路径 $P\text{-}path$，用 $\mathcal{U}(p)$ 表示该路径涉及的所有内存对象的集合，对于任意一个内存对象 $o \in \mathcal{U}(p)$，路径 $P\text{-}path$ 上该内存对象 o 的操作序列为该路径上所有与内存对象 o 相关的分配、使用、释放操作所组成的有序序列，用 $p[o]$ 表示。

例如，给定程序路径 $P\text{-}path = \langle a.malloc(), a.use(), b.malloc(), b.free(), a.free()\rangle$，其中 a 和 b 分别代表两个不同的内存对象，$malloc()$、$use()$ 和 $free()$ 分别代表内存的分配、使用和释放语句。那么基于定义 4.1，可以得到 $\mathcal{U}(p) = \{a, b\}$，以及 $p[a] = \langle a.malloc(), a.use(), a.free()\rangle$。此外，本文也将操作序列中两个相邻操作的转换关系称为操作序列的边，例如 $p[a]$ 包含两条边，分别为 $a.malloc() \rightarrow a.use()$ 和 $a.use() \rightarrow a.free()$ 的转换关系。

对于一条操作序列，可以检查它是否满足特定的类型状态属性[181]，类型状态属性的形式化定义如下。

定义 4.2 (类型状态属性). 类型状态属性 $\mathcal{P}$ 是一个有限状态自动机 $\mathcal{P} = (\mathcal{Q}, \Sigma, \delta, q_0, F)$，其中 $\mathcal{Q}$ 是一个有限状态的集合；Σ 是关于操作序列的字母表；δ 是状态转换函数；$q_0 \in \mathcal{Q}$ 是该自动机的初始状态；F 表示接受状态。

内存使用的时序安全规则本质上可以看作是程序的一种类型状态属性（如图 4-1），内存使用的时序安全规则的类型状态属性可用自动机 $\mathcal{P}_m$ 表示，简称内存时序属性：

定义 4.3 (内存时序属性). 内存时序属性可用自动机 $\mathcal{P}_m = (\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{Q} \setminus \{q_{err}\})$ 表示，其中 $q_{err} \in \mathcal{Q}$ 是该自动机的错误状态，对于任意的 $\sigma \in \Sigma$, 有 $\delta(q_{err}, \sigma) = q_{err}$；$\mathcal{Q} \setminus \{q_{err}\}$ 表示除了错误状态以外的其它状态都是接受状态。

- $\mathcal{Q} = \{init, live, dead, error\}$
- $\Sigma = \{malloc, use, free\}$
- $\delta = \{init \xrightarrow{malloc} live, live \xrightarrow{use} live, live \xrightarrow{free} dead, dead \xrightarrow{use/free} error, error \longrightarrow error\}$
- $q_0 = init$
- $q_{err} = error$

对于一个给定的程序 P，以及内存时序属性 $\mathcal{P}_m$，如果对于任意一条程序路径 p 都有 $\forall o \in \mathcal{U}(p) : p[o] \in \mathcal{P}_m$，其中，$p[o] \in \mathcal{P}_m$ 表示操作序列 $p[o]$ 可被有限自动机 $\mathcal{P}_m$ 接受，则可以说程序 P 遵守内存时序属性。反之，如果存在一条程序路径 p 使得 $\exists o \in \mathcal{U}(p) : p[o] \notin \mathcal{P}_m$，其中，$p[o] \notin \mathcal{P}_m$ 表示路径 p 上关于内存对象 o 的操作序列不能被有限自动机 $\mathcal{P}_m$ 接受，则可以说发现了一条操作序列 $p[o]$，该操作序列违反了内存时序属性。此外，若程序路径 p 是实际可达的，则可以说程序 P 违反了内存时序属性，即程序 P 存在内存时序漏洞。

4.2.2.1 静态分析

算法 4.1描述了利用静态分析识别潜在的违反内存时序属性的操作序列的过程。该算法以原始程序 P 作为输入，以目标操作序列的集合 S 作为输出。该算法先识别程序 P 中所有内存分配操作（第2行），这些内存分配操作的集合用 S_M 表示，其中 M 表示在 S_M 中分配的所有内存对象的集合。对于每一个 M 中的每一个内存对象 m，以及它对应的内存分配操作 s_m（第3行），该算法通过指针分析技术[26,91] 识别 m 的别名，即所有指向 m 的指针（第4行），然后接着分析这些指向 m 的指针是否存在内存释放操作，若存在内存释放操作则将该内存释放操作添加到集合 S_F 之中（第5行）。同样的，该算法也会寻找所有对 m 及其别名的内存使用操作，并将这些操作添加到集合 S_U 之中（第6行）。最后，该算法基于路径不敏感的可达性分析（$reachable$ 函数），提取违反了内存时序属性的操作序列，并将这些操作序列添加到输出集合 S 中（第7行）。具体来说，函数 $reachable(s_1, \langle a_1, ..., a_n \rangle, s_2)$ 若返回真，则表明从内存操作 s_1 到内存操作 s_2 可达，并且从 s_1 经过一串别名赋值操作 $\langle a_1, ..., a_n \rangle$ 到 s_2 之后，s_1 和 s_2 操作的是同一内存对象。算法识别的操作序列 $\langle s_m, \langle a_1, ..., a_j \rangle, s_f, \langle a_{j+1}, ..., a_n \rangle, s_u \rangle$ 是一个长度至少为 3 的操作序列，其中别名赋值操作 a_i 可为空操作，该操作序列至少包含一个内存分配操作 s_m，一个内存释放操作 s_f，以及一个内存使用/内存释放操作 s_u。

算法 4.1: 利用静态分析识别违反内存时序属性的操作序列

输入 : 原始程序 P
输出 : 一个存放操作序列的集合 S

1 $S \leftarrow \emptyset$
2 $(S_M, M) \leftarrow$ *find_malloc*(P) // 找到内存分配操作
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $A \leftarrow$ *cal_alias*(m) // 找到别名赋值操作
5 $S_F \leftarrow$ *find_free*(A, P) // 找到内存释放操作
6 $S_U \leftarrow$ *find_use*(A, P) // 找到内存使用操作
/* 通过路径不敏感的可达性分析，识别违反内存时序属性的操作序列 */
7 $S \leftarrow S \cup \{\langle s_m, \langle a_1, ..., a_j \rangle, s_f, \langle a_{j+1}, ..., a_n \rangle, s_u \rangle \mid s_f \in S_F \wedge s_u \in S_U \cup S_F \wedge \forall i \in [0, n], a_i \in A \cup \{\varnothing\}$
$\wedge\, reachable(s_m, \langle a_1, ..., a_j \rangle, s_f) \wedge reachable(s_f, \langle a_{j+1}, ..., a_n \rangle, s_u)\}$
8 **return** S

UAFL 采用路径不敏感的可达性分析，其优点是不需要求解复杂的路径约束，成本较低，可扩展性高，故非常适用于复杂性高和规模较大的真实应用程序，但这也会导致较多误报，即识别的操作序列不一定是真实可达的，因此 UAFL 仅将识别出的操作序列视作潜在的内存时序漏洞的发生条件，通过程序插桩注入操作序列导向的信息以在模糊测试阶段进一步确认这个潜在的内存时序漏洞。值得注意到是，UAFL 识别的操作序列还包含了指针的别名赋值操作，当内存释放操作和内存使用操作依赖于特定的别名赋值操作时，UAFL 将该别名赋值操作添加到操作序列当中（算法 4.1的第 7行）。以图 4-2的示例程序为例，由于程序第 7 行是一个指针的别名赋值操作，使得 ptr1 和 ptr2 可能指向同一个内存对象，当程序第 7 行的指针别名赋值操作执行后，程序第 10 行的内存释放操作和第 14 行的内存使用操作实际上操作的是同一内存对象。因此，从图 4-2的示例程序中提取的操作序列为 “第 4 行→第 7 行→第 10 行→第 14 行”。

4.2.2.2 程序插桩

算法 4.2描述了 UAFL 基于静态分析识别出的操作序列进行程序插桩的详细过程。该过程以原始程序 P 和静态分析识别出的操作序列的集合 S 作为输入，以插桩后的程序 P' 作为输出。总体来说，UAFL 的插桩主要包括以下两个步骤：(1) 通过位图 *OPE_mem* 来记录操作序列中边及其控制依赖是否按照既定的时序被执行；(2) 通过位图 *IFA_mem* 为之后的信息流分析记录条件分支语句中条件表达式的值。

UAFL 的插桩是在基本块层面进行的，对于集合 S 中的每条操作序列 $\langle s_0, \ldots, s_n \rangle$，可以获取到它所对应的基本块序列 $\langle b_0, \ldots, b_m \rangle$（第 3 行）。由于可能存在操作序列

算法 4.2: UAFL 的插桩过程

输入：原始程序 P 和一个存放操作序列的集合 S

输出：一个插桩后的程序 P'

```
1  let OP_mem, OPE_mem and IFA_mem be a bitMap in shared memory
2  foreach ⟨s_0, ..., s_n⟩ ∈ S do
3      ⟨b_0, ..., b_m⟩ ← BB_OP(⟨s_0, ..., s_n⟩)
4      foreach b_i ∈ ⟨b_0, ..., b_m⟩ do
5          insert OP_mem[ID_{b_i}] ← 1 into b_i                    // 记录操作序列中操作的执行
6      foreach b_i ∈ ⟨b_1, ..., b_m⟩ do
7          let C_i be a set of conditional statements between b_{i-1} and b_i, where b_i is
             control-dependent on C_i                               // 求控制依赖
8          foreach c_j ∈ C_i do
9              let b_{c_j} be parent basic block of c_j
10             tID ← 0
11             for 0 ≤ k < i do
12                 if OP_mem[ID_{b_k}] == 1 then
13                     tID ← tID ⊕ ID_{b_k}  // 已覆盖的操作序列前缀编码成一个独特的标记
14             insert OPE_mem[tID ⊕ ID_{b_{c_j}}] ++ into b_{c_j}  // 记录操作序列边及控制依赖的执行
15             let c_j be comparison instruction cmp a, b
16             insert IFA_mem[ID_{b_{c_j}}] ← a-b into b_{c_j}      // 记录分支条件语句中变量的值
```

中的多个操作位于同一基本块的情况，基本块序列的长度可能会小于操作序列的长度，即 $m <= n$。对于基本块序列中的每个基本块 b_i，UAFL 在该基本块中插入“$OP\_mem[ID_{b_i}] \leftarrow 1$”语句（第 4-5 行），其中 ID_{b_i} 为基本块 b_i 的一个唯一标记（见定义 3.2），OP_mem 是一个临时的位图，用来记录操作序列中的操作是否被执行。当程序动态执行到该基本块时，$OP\_mem[ID_{b_i}]$ 的值将被设置为 1。

需要注意的是，操作序列中仅包含对单个内存对象的分配、使用、释放操作，这些操作执行可能还依赖于特定的路径约束。要完全覆盖一条操作序列，其相关的路径约束也必须能被满足，对于图 4-2的示例，若要覆盖操作序列中的“第 7 行→第 10 行”，程序在第 8 行和第 9 行都必须走 true 分支，实际上需要先覆盖“第 8 行→第 9 行”和“第 9 行→第 10 行”。为了提供这种更细粒度的引导来使得操作序列更容易被覆盖，UAFL 还额外考虑了与操作序列中的基本块有支配关系的其它基本块的覆盖情况（第 6-13 行）。对于基本块序列中的每一个基本块 $b_i \in \langle b_0, \ldots, b_m \rangle$，UAFL 首先找到控制 b_i 的条件语句的集合 C_i，即对于任意一个 $c_j \in C_i$，b_i 控制依赖于 C_i（第 6-8行），然后获取到 C_i 所在的基本块 b_{c_j}（第 9行），并在该基本块中注入收集基本块覆盖情况的程序语句（第 14行）。例如图 4-2示例的第 8 行和第 9 行所在的基本块在控制流程图上是支配第 10 行的，因此第 8 行和第 9 行的覆盖情况

会被编码进 *tID*，最后被记录到位图 *OPE_mem* 中（算法 4.2的第 14行）。

UAFL还对条件语句进行插桩，为模糊测试阶段的变异优化策略（详见第4.2.3.2节）提供反馈。程序中的每一个条件语句都可以看作成一条比较指令，即 “cmp a, b”，其中 a 和 b 为表达式。UAFL 跟踪并记录 $a - b$ 的值（算法 4.2的第 16 行），该值能有效体现条件语句中变量的变化。例如图 4-2示例的第一个条件语句为 “if (buf[5] == 'e')”，UAFL 跟踪并记录 “buf[5]-'e'” 的值，模糊测试阶段，当对种子测试用例的某个字段进行变异时，若引起的 “buf[5]-'e'” 的值的变化，则说明对该字段的变异操作是有效的，可以增加对该字段的变异力度。

4.2.3 操作序列导向的模糊测试

操作序列导向的模糊测试方法以操作序列的覆盖率作为反馈信息，引导测试用例的自动生成不断地朝着更高的操作序列覆盖率的方向迈进，同时结合本章提出的基于信息流分析的变异优化策略，持续对目标程序进行测试，能有效发现内存时序漏洞。

算法 4.3 描述了UAFL 对一个目标程序进行模糊测试的过程。整体而言，UAFL 不断地从种子池中选取种子进行变异，自动化地生成一组变异体，并将这些变异体用作测试用例来对目标程序进行测试，同时监视它们的运行状况（第 3-9行）。若发现目标程序触发了内存时序漏洞，则将相应的测试用例保存起来，以便将来复现和定位错误（第 10-11行）。如果测试用例能覆盖更多的操作序列的边，则它将作为一个高质量的种子添加到种子池中（第 12-13行）。UAFL 在预设的时间内持续遍历种子池挑选种子进行变异，不断生成测试用例对目标程序进行测试。该算法与传统的覆盖导向的模糊测试技术的主要区别有两点，一方面，该算法使用的是操作序列引导的反馈机制（详见第 4.2.3.1节），根据测试用例对目标操作序列及其控制依赖的覆盖率来决定是否将该测试用例作为高质量的种子添加到种子池中，有助于使得测试用例的生成朝着更高的操作序列覆盖率的方向发展；另一方面，该算法在变异阶段采取了一个基于信息流分析的变异优化策略（详见第 4.2.3.2节），使得变异操作更有效地施加于与操作序列相关的输入字段上，提升了测试用例生成的质量和效率。

具体而言，算法 4.3 以插桩后的可执行程序 P' 和一组初始种子 T 作为输入，以一个能触发内存时序漏洞的测试用例集合 S 作为输出。算法开始时，集合 S 以空集作为初始值。模糊测试过程中维护的种子池实际上是一个队列，算法中用变量 $Queue$ 表示，后文也称其为种子队列。$Queue$ 通过一组初始种子 T 进行初始化（第 2行）。算法持续从种子池 $Queue$ 中选择测试用例（第 4行），并基于一定的概率决

算法 4.3: 操作序列导向的模糊测试算法

输入：一个插桩后的程序 P'，以及一个初始种子的集合 T
输出：触发内存时序漏洞的测试用例集合 S

```
1  S ← Φ
2  Queue ← T
3  while time and resource budget do not expire do
4      for each input t in Queue do
5          if with probability FuzzProb_t to select t then
6              numChildren ← AssignEnergyOSe(t)      // 基于操作序列覆盖度分配能量
7              for 0 ≤ i < numChildren do
8                  child_i ← MutateWithIF(t, IFA_mem)    // 基于信息流分析的变异优化
9                  OPE_mem_i ← Run(child_i, P)           // 执行程序并收集反馈信息
10                 if the execution triggers UaF or DF bugs then
11                     S ← S ∪ child_i
12                 if NewCov(OPE_mem_i) then             // 操作序列引导对反馈机制
13                     Queue ← Queue ∪ child_i
14 return S
```

定是否选择对该测试用例进行进一步变异（第5行）。假设测试用例 t 被选择进行变异，测试用例 t 会被分配一定的能量值（$numChildren$），即基于测试用例 t 变异生成的变异体的个数。UAFL 采取了一个新的启发式方法确定 $numChildren$ 的值，它倾向于为操作序列覆盖率更高的种子测试用例赋予更多的能量值（第6行）。UAFL 基于信息流分析对测试用例 t 进行变异，有策略地生成 t 的变异体 $child_i$（第8行），算法将 $child_i$ 用作程序 P' 的新测试用例，执行程序并监视其运行情况（第9行），收集对于操作序列的覆盖率信息 *OPE_mem*。若测试用例 $child_i$ 执行时发生了崩溃，触发了内存时序漏洞，则它将被添加到集合 S 中（第11行）。否则，算法进一步分析 $child_i$ 对于操作序列的覆盖情况（第12行）。若 $child_i$ 的执行覆盖了操作序列的边，则将其作为高质量的种子添加到 $Queue$ 中等待下一轮被选择时进行变异（第12-13行）。算法重复以上过程，直到超出了时间或其它资源的预算范围（第3行）。

4.2.3.1 操作序列引导的反馈机制

UAFL 采用的是操作序列引导的反馈机制，通过跟踪模糊测试阶段对所有目标操作序列的覆盖率，并使用操作序列的覆盖率指导模糊测试，逐步生成能够覆盖更多、更完整的操作序列的测试用例。

UAFL 与传统的覆盖导向的模糊测试技术在反馈机制的设计上有着很大的差异。覆盖导向的模糊测试技术关注的是分支覆盖信息，即采用 $traceBit$ 来跟踪控

制流程图中的边覆盖情况（算法 3.2的第 14行）；而 UAFL 关注的是操作序列的覆盖率信息，即采用 *OPE_mem* 来跟踪操作序列中的边及其控制依赖的覆盖情况（算法 4.3的第 12行）。另一方面，覆盖导向的模糊测试技术不关心分支的的执行顺序，即 $traceBit$ 仅简单记录了 *Edge-ID* 是否被执行（算法 3.1的第 13行）；而 UAFL 考虑了操作序列中各个操作的执行顺序，即 *OPE_mem* 刻画了操作序列中的边及其控制依赖的执行顺序（算法 4.2的第 7-14行）。具体而言，UAFL 在插桩阶段将已覆盖的操作序列的前缀部分编码成一个独特的标记 *tID*，并将 *tID* 与其后继基本块的时序关系再次编码后记录于 *OPE_mem* 之中（算法 4.2的第 10-13行）。因此，当以不同的执行顺序覆盖相同的内存操作时，*tID* 能体现出差异，进而通过 *OPE_mem* 的值影响种子的保留决策。再次以图 4-5为例，操作序列引导的反馈机制使得 UAFL 成功生成测试用例 “aaaseen”、“auaseen” 和 “fuaseen”，它们对于操作序列的覆盖率是逐步提升的，即从覆盖 “第 4 行→第 7 行” 到 “第 4 行→第 7 行→第 10 行”，最终覆盖了完整的操作序列 “第 4 行→第 7 行→第 10 行→第 14 行”。

UAFL 还采取了一个新的启发式方法来决定给每个种子测试用例分配的能量（第 6行），它倾向于给予操作序列覆盖率更高的种子更多的变异机会。覆盖导向的模糊测试技术（如 AFL）通常根据种子的质量来决定如何分配能量[182]，其计算方式如下：

$$assignEnergy(t) = allocate\_energy(q_t) \tag{4-1}$$

其中 t 表示种子测试用例，q_t 是根据 t 的创建时间、执行速度、覆盖的分支等因素评估得到的种子质量值。UAFL 在对种子的能量分配上除了考虑以上因素之外，还考虑操作序列的覆盖率：

$$AssignEnergyOSe(t) = AssignEnergy(t) * (1 + \frac{\#c\_OSe}{\#t\_OSe}) \tag{4-2}$$

其中 $\#t\_OSe$ 表示所有目标操作序列中边的总数，$\#c\_OSe$ 表示当前种子 t 覆盖的操作序列的边的数量。UAFL 将更多的变异机会分配给操作序列覆盖率更高的种子，从而让这些种子可以有更多的变异机会进一步覆盖更多的操作序列。

4.2.3.2 基于信息流分析的变异优化策略

对已有种子测试用例进行变异是自动生成生成新测试的重要手段，UAFL 采用了基于信息流分析的变异优化策略，其目的是为了使得变异出的测试用例更加有效，从而更快的覆盖操作序列。所谓信息流分析[141,142]，即对程序中变量值的获取和传播数据流进行分析，它能准确了解变量的特征，也能清楚地知道输入和变量之间的关系。UAFL 的操作序列覆盖率信息包含了其中内存操作控制依赖于的

条件语句，故UAFL关注测试输入的哪些字段可以改变特定条件语句中变量的值，随后加大力度对这部分字段进行变异，直到满足条件语句的判断条件。

UAFL在测试执行的过程中动态地推断测试用例中被变异的字节与条件语句中变量之间是否存在关系，该关系可通过信息流强度[183]的大小反映出来。信息流强度的定义及其计算方式如下。

定义 4.4 (信息流强度). 对于给定的两个变量 x 和 y，以及它们的值域分别为 V_x 和 V_y，那么从变量 x 到变量 y 的信息流强度可以表示为：

$$\mathit{IFStrength}(x, y, V_x, V_y) = H(x, V_x) - H(x|y, V_x, V_y) \tag{4-3}$$

其中，$H(x, V_x)$ 代表变量 x 的信息熵：

$$H(x, V_x) = -\Sigma_{x_i \in V_x} P(x = x_i) log_2 P(x = x_i) \tag{4-4}$$

$H(x|y, V_x, V_y)$ 代表在已知变量 y 分布的前提下，变量 x 的条件信息熵：

$$\begin{aligned} H(x|y, V_x, V_y) = &-\Sigma_{y_j \in V_y} P(y = y_j) \\ &*[\Sigma_{x_i \in V_x} P(x = x_i|y = y_j) log_2 P(x = x_i|y = y_j)] \end{aligned} \tag{4-5}$$

UAFL通过变异测试输入的每个字节并记录分支语句判断条件中程序变量的值来进行采样。基于采样得到的信息流强度，UAFL计算得到测试输入每个字节的变异概率。直观的说，从测试输入的某个特定字节到分支语句判断条件中程序变量的信息流强度越高，该字节的变异概率就越高。算法4.4描述了测试输入中每个字节的变异概率的计算过程，包含两个步骤：（1）计算测试输入的每个字节和目标条件语句中变量的信息流强度（第1-8行）。信息流强度越高，则说明该字节对变量值的影响就越大；（2）为这些对变量值有影响的输入字节分配更高的变异概率（第9-12），因为对它们的变异更有可能改变条件分支使得操作序列的覆盖度提高。

算法4.4的输入包括长度为 m 字节的的程序输入 x，以及 n 个cmp指令中的变量（条件语句都可以看作是一条比较指令，即“cmp a, b”或“cmp a-$b, 0$”，a-b 即为本章所指的条件语句中的变量）。该算法的输出为程序输入 x 的每个字节的变异概率，这些概率被存放于数组 $prob[m]$ 中。UAFL对于程序输入 x 的每一个字节（第1行），计算其到变量 b_v 的信息流强度。X 集合被用于临时存储关于程序输入的每一字节的采样值，而 Y 集合被用于临时存储 b_v 中每一个变量的采样值（第2行）。UAFL对 x 的每一个字节（即 $x[i]$）都进行 k 次变异（第3行），相应的，对于 b_v 中的每一个变量也能采样到 k 个值。具体而言，假设每次变异后的新测试用例为 x'（第4行），在执行测试用例 x' 后，b_v 的具体值也可以被获得（来自算法4.2的位图

算法 4.4: 变异概率的计算

输入：一个程序输入 $x[m]$，以及在 n 个‘cmp’指令上的变量 $b_v[n]$
输出：程序输入中每一个字节的变异概率 $prob[m]$

1 **foreach** $i \in \{0,\ldots,m-1\}$ **do**
2 $X = \emptyset, Y = \emptyset$
3 **foreach** $j \in \{0,\ldots,k\}$ **do** //对每一个字节变异 k 次进行采样
4 $x' \leftarrow mutate(x[i])$
5 $b'_v \leftarrow evaluate(x', IFA\_mem)$
6 $X \leftarrow X \cup \{x'[i]\}$
7 $Y \leftarrow Y \cup \{b'_v\}$
8 $E(x[i]) \leftarrow \max_{j\in\{0,\ldots,n-1\}} IFStrength(x[i], b_v[j], X, Y[j])$ //计算信息流强度
9 $minE \leftarrow min(E(x[0]),\ldots E(x[m-1]))$
10 $maxE \leftarrow max(E(x[0]),\ldots E(x[m-1]))$
11 **foreach** $i \in \{0,\ldots,m-1\}$ **do**
12 $prob[i] \leftarrow \dfrac{E(x[i]) - minE}{maxE - minE}$ //计算对该字节的变异概率

IFA_mem）并临时存放于 b'_v 中（第5行）。当 k 个测试用例都被测试完后，UAFL 分别计算被变异的字节 $x[i]$ 和 b_v 中每个变量之间的信息流强度，其中的最大值将被作为 $x[i]$ 的信息流强度（第8行）。最后，UAFL 对于程序输入 x 的每个字节都计算一个信息流强度，通过归一化的方式为程序输入 x 的每个字节计算出一个变异概率（第9-12行）。

4.3 实验评估

本章使用C/C++代码实现了一个融合程序分析与模糊测试的内存时序漏洞检测工具原型UAFL，其中静态分析部分是基于开源的过程间静态数值流（Value-flow）分析工具 SVF[91] 实现的，程序插桩部分是基于 LLVM/Clang 框架[169] 实现的。具体来说，由于 SVF 是一个具有高可扩展性、低耦合度、基于数值流的稀疏分析框架，本章利用 SVF 来执行安德森指针分析[184,185] 以帮助识别分配、释放和使用同一内存块的操作，再通过可达性分析识别违反内存时序属性的操作序列。同时，本章在开源模糊测试工具 AFL-2.52b 的基础之上实现了操作序列导向的模糊测试，并增加了基于信息流分析的变异优化策略的实现。本章对实现的 UAFL 工具原型进行了全面的实验评估。

4.3.1 实验对象

为了进行全面的实验评估，本章选取了 14 个不同规模的真实应用程序作为实验对象。表 4-1列出了详细信息，其中第 1 列表示程序 ID，第 2 列表示程序名，

表 4-1　实验对象

ID	程序	版本号	代码行数	程序简介
1	readelf	2.28	1,844k	编程语言工具程序；从 ELF 格式的目标文件显示信息
2	readelf	2.31	1,758k	编程语言工具程序；从 ELF 格式的目标文件显示信息
3	jpegoptim	1.45	2k	jpegoptim 是一个用来优化 JPEG 文件的工具
4	liblouis	3.2.0	53k	Liblouis 是一个开源的盲文翻译和反向翻译软件
5	lrzip	0.631	19k	lrzip 是一个实用的压缩程序，压缩大文件时能达到较高的压缩率
6	Mini XML	2.12	15k	Mini-XML 是一个用 C 语言开发的轻量级 XML 解析器
7	broingssl	894a4	162k	BoringSSL 是谷歌创建的 OpenSSL 替代品
8	GNU cflow	1.6	50k	cflow 可以生成 C 语言代码的函数之间的依赖关系图
9	Boolector	3.0.0	141k	流行的 SMT 求解器
10	openh264	1.8.0	143k	OpenH264 是思科公司发布的一个开源的 H.264 编码和解码器
11	libpff	20180623	125k	libpff 是一个访问个人文件夹文件（PFF）格式的库
12	mjs	1.20.1	40k	C/C++ 嵌入式 JavaScript 引擎，专为资源受限的微控制器而设计
13	ImageMagick	7.0.8	485k	ImageMagick 是一个免费的创建、编辑、合成图片的软件
14	nasm	2.14	101k	NASM 是一款基于 x86 架构的汇编与反汇编软件

第 3 列表示程序的版本号，第 4 列显示代码行数，第 5 列对程序做简单描述。

本章基于以下因素选取实验对象：在现有工作中被测试的频率；在终端用户中的受欢迎程度；功能的多样性。本章选取的实验对象包括著名的开发工具（例如 *readelf 2.28*，*readelf 2.31*）、代码处理工具（例如 *mjs*，*Mini XML*，*GNU cflow*，*nasm*）、图像处理库（例如 *ImageMagick*，*jpegoptim*）、解压缩工具（例如 *lrzip*，*openh264*）、结构化数据处理库（例如 *libpff*，*liblouis*）、加密密钥管理工具（例如 *boringssl*）和可满足性模理论求解器（例如 *boolector*）等多个方面。本章选取的实验对象都是在实际应用领域被广泛使用的开源 C/C++ 程序，且多次被研究者使用[64,67,70,170,171]。实验对象的规模大小从 2k 到 1844k 行 C/C++ 代码不等。对于每个程序，本章选择其包含已知内存时序漏洞（例如释放后使用和双重释放）的版本（见表 4-1）。

4.3.2　实验设计

为了对本章方法进行深入研究，本章设计了四大类实验，旨在回答以下四个研究问题：

问题一： UAFL 在静态分析阶段的性能表现如何?

问题二： UAFL 在发现真实应用中内存时序漏洞的实际效果如何?

问题三： 基于信息流分析的变异优化策略对于检测内存时序漏洞是否有帮助?

问题四： UAFL 在实现与内存时序漏洞相关代码的覆盖率表现如何?

为了更好的评估 UAFL，本章选取了六个最具代表性的业界先进的模糊测试工具与 UAFL 进行了对比：（1）AFL[30] 是当前最流行的模糊测试工具，大多数覆盖导向的模糊测试技术都是在 AFL 的基础上进行发展的；（2）AFLfast[63] 在 AFL

基础之上增加了能量调度策略，给被选次数较少的种子分配更高的能量，加大对低频路径的探索力度；（3）FairFuzz[64] 利用目标突变策略引导模糊测试探索执行更稀有的分支，从而更高效地提升测试时的覆盖率；（4）MOpt[186] 采用一种定制的粒子群优化算法，对变异算子的效率进行动态评估，并将其选择概率调整到最优分布，从而提高模糊测试的性能。（5）Angora[33] 基于污点分析跟踪信息流，然后使用梯度下降法高效突破未覆盖分支，提高了分支覆盖率；（6）QSYM[35] 是一个混合模糊测试工具，利用模糊测试为程序快速生成大量测试用例，符号执行基于模糊测试的覆盖信息进行搜索，仅为未覆盖到的分支生成测试用例，通过结合两种技术生成具有更高分支覆盖率的测试用例。总之，本章选取的六个模糊测试工具都使用了不同的、业界先进的技术改进了模糊测试的覆盖率以及发现漏洞的有效性，并在实践中被证明是有效的。

在配置参数方面，对于所有的模糊测试工具，其配置参数和使用的初始种子是相同的。由于本章方法是基于模糊测试技术的，而模糊测试技术的有效性一定程度上依赖于随机性的变异，因此在实验测量结果上可能会存在一定的实验误差。根据 Klees 的建议[170]，本章采取了三项措施来缓解随机性带来的实验误差。首先，本章对每个目标程序都进行较长时间的测试，直到整个模糊测试达到相对稳定的状态。因此本章每组实验统一运行每个模糊测试工具长达 24 小时。再者，本章对每个实验对象都会进行 8 次重复实验，统计平均结果，并基于统计学假设检验方法评估各个工具平均性能。第三，对于同一目标程序，各个模糊测试工具在初始种子的选择上都是相同的。如果目标程序提供了测试用例样本，本章直接将其用作初始种子。否则，根据所需的输入格式从互联网上随机下载一些符合输入格式的文件作为测试用例。此外，本章的实验对象主要包含释放后使用（UaF）和双重释放（DF）这两类内存时序漏洞，而 ASAN[27] 是一个适用于 C/C++ 程序的动态内存错误检测器，本章使用 ASAN 作为模糊测试的测试预言（Test Oracle）。

本章所有的实验环境为 Intel (R) Xeon (R) E5-1650 v4 的处理器，主频为 3.60Hz，64 位 Ubuntu LTS 18.04 操作系统，16GB 内存。

4.3.3 实验结果与分析

4.3.3.1 实验一：静态分析性能评估

对于每个实验对象，本章都对其进行了静态分析以辅助程序插桩，统计分析结果如表 4-2所示，*BB* 列给出了各个程序包含的基本块（Basic Block）总数，BB_{UAF} 列表示静态分析得到的可能违反了内存时序属性的操作序列所涉及的所有基本块数量。UAFL 对 BB_{UAF} 中所有的基本块都进行插桩，以提供操作序列导向的反馈

表 4-2　静态类型状态机分析评估

程序	BB	BB_{UAF}	BB_{IF}	BB_{Free}	操作序列	耗时（秒）
readelf 2.28	16,967	2,681 (15.8%)	1,103 (6.5%)	91	41,605	262
readelf 2.31	19,973	3,647 (18.2%)	1,555 (7.8%)	98	130,102	508
jpegoptim	634	36 (5.7%)	28 (4.4%)	5	44	1
liblouis	2,957	486 (16.4%)	190 (6.4%)	8	422	18
lrzip	9,356	1,051 (11.2%)	467 (5.0%)	6	313	150
Mini XML	4,237	890 (21.0%)	788 (18.6%)	10	486	44
boringssl	22,547	3,701 (16.4%)	3,265 (14.4%)	32	84,069	2,005
GNU cflow	5,095	1,402 (27.5%)	751 (14.7%)	33	4330	30
Boolector	26,866	11,511 (42.8%)	9,031 (33.6%)	4	28,586	2,387
openh264	12,735	2,090 (16.4%)	927 (7.3%)	1	1,219	1,127
libpff	18,569	6,371 (34.3%)	6,041 (32.5%)	60	20,865	122
mjs	4,937	546 (11.0%)	343 (6.9%)	16	1,143	24
ImageMagick	31,190	1,573 (5.0%)	1,336 (4.3%)	3	55,877	2,185
nasm	13,965	3,812 (27.2%)	3,390 (24.2%)	2	3,357	2,210
Avg.	13,573	2,842 (19.2%)	2,087 (13.3%)	26	26,601	1,148

机制，这些基本块平均占总基本块的 19.2%。BB_{IF} 列表示基于信息流分析进行插桩的基本块数量。为了计算信息流强度，UAFL 平均需要对 13.3% 的基本块进行插桩。传统的基于覆盖导向的模糊测试技术需要对所有的基本块进行插桩以获取全局分支覆盖情况，而该实验统计结果表明，UAFL 对程序进行更少的插桩（最坏情况下基于操作序列插桩和信息流分析插桩的基本块数量不超过总基本块数量的 32.5%），也意味着它可以集中更多的资源在测试可能违反内存时序属性的操作序列上。

BB_{Free} 列表示包含内存释放操作的基本块数量，“操作序列”列中的数字表示静态分析所识别的可能违反内存时序属性的操作序列数量，例如最小规模的程序 *jpegoptim* 被识别出了 44 条操作序列，而较大规模的程序 *readelf 2.31* 被识别出 130,102 条操作序列，整体来说，平均每个程序被识别出有 26,601 条操作序列。由于 UAFL 采用路径不敏感的可达性分析，会产生较多的误报。因此，后续模糊测试阶段将进一步确认这些操作序列是否真的触发内存时序漏洞。最后一列统计了各个程序在静态分析阶段所用的时间开销，平均每个程序需要花费 1148 秒（0.32 小时）来进行静态分析。静态分析所需要的时间开销远远不到 1 个小时，这与需要长时间运行（例如 24 小时）的模糊测试相比，静态分析的时间开销是可接受的。

实验结论：实验结果表明，UAFL 执行的静态分析所引入的时间开销是可接受的，其平均耗时为 1,148 秒（0.32 小时）。操作序列导向的反馈机制与基于信息流的变异优化这两种策略平均分别需要插桩 19.2% 与 13.3% 的基本块，UAFL 相比覆盖导向的模糊测试技对程序进行更少的插桩，因此可以集中更多的资源测试可

能违反内存时序属性的操作序列。

4.3.3.2 实验二：漏洞检测能力评估

正如 Klees[170] 所建议的，发现真实程序中的漏洞的能力是衡量模糊测试工具有效性最直接的方法。因此，本实验直接使用 UAFL 和其它六个模糊测试工具对 14 个真实的应用程序进行测试，并统计它们发现内存时序漏洞所需的时间。

表 4-3统计了各个模糊测试技术发现内存时序漏洞所需要的时间，其中前三列分别列出了程序名、漏洞标识符和漏洞类型，第四列为 UAFL 发现各个内存时序漏洞所需的时间，第五列为未采用基于信息流分析的变异优化策略的 UAFL 版本 UAFL_{NIF} 的实验结果（本实验主要关注 UAFL 的性能表现，后续实验会具体评价 UAFL_{NIF} 的性能表现），最后六列分别列举了六个业界先进的模糊测试技术的统计结果。对于表 4-3中的 15 个内存时序漏洞，只有 UAFL 发现了全部 15 个漏洞，而 AFL、AFLFast、FairFuzz、MOpt、Angora 和 QSYM 分别漏报了 3、5、3、3、10 和 4 个漏洞。整体而言，UAFL 发现内存时序漏洞的时间平均需要 2.79 小时，而其它模糊测试技术大约在 10 小时左右。与其它六个模糊测试技术相比，即使算上静态分析的时间开销（即平均 0.32 小时），UAFL 发现内存时序漏洞的时间开销仍然要小得多。UAFL 在发现漏洞的速度上，分别是 AFL、AFLFast、FairFuzz、MOpt、Angora 和 QSYM 的 3.25 倍、3.16 倍、2.63 倍、3.35 倍、6.00 倍和 3.80 倍，这充分体现了 UAFL 发现内存时序漏洞的高效性。

值得注意的是，对于 *readelf 2.31*、*lrzip*，几乎所有的模糊测试工具都能在极短的时间内发现其中的 UaF 漏洞，而对于 *liblouis*、*GNU cflow* 和 *ImageMagick* 中更深的漏洞，UAFL 的性能明显要比其它工具好得多。例如 UAFL 仅需要大约 1.80 小时即可发现 *GNU cflow* 中的 UaF 漏洞，而其它模糊测试工具需要 20 小时以上。此外，在这 15 个漏洞中，UAFL 对于其中 13 个（86.7%）漏洞的发现时间都最短。对于程序 *readelf 2.31*，Angora 的性能表现最佳，而 FairFuzz 则对于 *Boolector* 的性能表现最佳。通过人为仔细分析其背后的原因，本章发现这两个漏洞的触发恰好符合这两种模糊测试技术的优化目标。例如，FairFuzz 旨在探索和覆盖稀有的分支，而 *Boolector* 中的与 UaF 漏洞相关的内存释放语句（即 free）正好处于这种稀有的、难覆盖的分支中，故 FairFuzz 在覆盖了稀有分支之后有很高的概率能检测到 *Boolector* 中的 UaF 漏洞。与其他模糊测试技术不同，UAFL 是基于类型状态机属性设计的，因此 UAFL 在发现内存时序漏洞方面的性能表现更稳定。

为了降低模糊测试随机性造成的实验误差，本章对实验结果进行了统计学假设检验，如表 4-4所示。本章用的统计学假设检验方法与第 3.3节保持一致，即采

表 4-3　发现内存使用协议违背漏洞所需要的时间消耗评估

程序	漏洞标识符	漏洞类型	发现漏洞所需要的时间消耗（单位：小时）							
			UAFL	**UAFL$_{NIF}$**	**AFL**	**AFLFast**	**FairFuzz**	**MOpt**	**Angora**	**QSYM**
readelf 2.28	CVE-2017-6966	UaF	0.59	1.32	6.09	1.43	0.68	3.61	T/O	6.20
readelf 2.31	CVE-2018-20623	UaF	0.10	0.10	0.10	0.10	T/O	0.10	**0.02**	0.10
jpegoptim	CVE-2018-11416	DF	0.09	0.10	0.59	0.88	1.08	1.49	T/O	1.95
liblouis	CVE-2017-13741	UaF	1.11	1.81	15.81	T/O	6.96	17.38	T/O	13.42
lrzip	CVE-2018-11496	UaF	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Mini XML	CVE-2018-20592	UaF	0.38	0.93	1.28	2.59	0.54	16.7	T/O	18.99
boringssl	Google Test-suit	UaF	0.33	1.06	T/O	T/O	4.67	7.62	–	T/O
GNU cflow	CVE-2019-16165	UaF	1.80	12.21	23.29	T/O	20.02	T/O	T/O	T/O
Boolector	uaf-issue-1	UaF	0.83	0.97	1.09	0.82	0.39	1.66	–	1.16
openh264	uaf-issue-2	UaF	8.17	13.00	15.80	11.15	8.17	15.39	T/O	18.45
libpff	CVE-2020-18897	UaF	1.39	1.39	4.21	4.11	3.98	4.35	T/O	4.98
mjs	uaf-issue-3	UaF	1.21	1.23	3.10	3.02	1.45	4.6	T/O	6.71
ImageMagick	CVE-2019-15140	UaF	6.29	13.92	T/O	T/O	T/O	T/O	T/O	T/O
nasm	CVE-2018-19216	UaF	2.59	4.69	8.32	3.45	2.86	11.46	2.75	9.64
	CVE-2018-20535	UaF	17.03	T/O	T/O	T/O	T/O	T/O	T/O	T/O
发现的漏洞数量			15	14	12	10	12	12	3	11
平均时间消耗			2.79 + 0.32	5.12 + 0.32	10.11	9.84	8.18	10.42	18.67	11.84
UAFL 发现漏洞速度提升			—	1.75×	3.25×	3.16×	2.63×	3.35×	6.00×	3.80×
UAFL$_{NIF}$ 发现漏洞速度提升			—	—	1.86×	1.81×	1.50×	1.92×	3.43×	2.18×

* UaF 和 DF 分别代表 use-after-free 和 double-free 漏洞的缩写；T/O 代表超时，意味着在 8 次 24 小时的重复实验中，模糊测试工具没有找到漏洞；在计算“平均耗时”时，T/O 被替换为 24 小时；此外，由于 Angora 在对程序 *boringssl* 和 *Boolector* 插桩的过程中发生了异常，无法进行实验评估，其结果用“-”表示。

用 Vargha Delaney 度量[172] 来计算 UAFL 优于其它六个对比工具的可信度，并采用曼-惠特尼 U 检验（Mann-Whitney U 检验）[173] 来检查实验结果的统计显著性差异。Vargha Delaney 度量（$\hat{A}12$）的值衡量了 UAFL 较其它模糊测试工具表现更好的概率，在大多数情况下其 $\hat{A}12$ 的值都超过了 0.71，即 UAFL 优于其它工具的可信度较高。在表 4-4中，如果 P 值小于显著性水平（0.05）的值，则对应的 $\hat{A}12$ 值会被用粗体标记。较小的统计显著性差异（P 值）表明 UAFL 和其它模糊测试工具之间的差异更显著。在一共 90 组假设检验中，有 68 组（75.6%）$\hat{A}12$ 值超过了 0.71，同时其 P 值小于 0.05，因此可以得出结论，在大多数实验对象中，UAFL 在发现内存时序漏洞上显著优于其他 6 个业界先进的模糊测试技术。

实验结论：从表 4-3和表 4-4的实验统计结果可以看出，UAFL 在大多数情况下要明显优于其它六个业界先进的模糊测试技术。UAFL 在检测内存时序漏洞上相比 AFL、AFLFast、FairFuzz、MOpt、Angora、QSYM 方法发现的漏洞数量上要多于 25%，并且实现了至少 2.63 倍的提速。

4.3.3.3　实验三：策略有效性评估

UAFL 主要采用操作序列导向的反馈机制和基于信息流分析的变异优化策略

表 4-4　对发现内存使用协议违背漏洞所需的时间的统计学假设检验

程序	漏洞标识符	$\hat{A}_{12}$ (UAFL)						$\hat{A}_{12}$ (UAFL$_{NIF}$)					
		AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf	CVE-2017-6966	**0.906**	**0.898**	**1.000**	0.609	**1.000**	**0.968**	**0.796**	0.546	**1.000**	0.453	**1.000**	**0.828**
readelf	CVE-2018-20623	0.500	0.500	0.500	0.500	**0.000**	0.500	0.500	0.500	0.500	0.500	**0.000**	0.500
jpegoptim	CVE-2018-11416	**0.995**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.995**	**1.0**	**1.000**	**1.000**	**1.000**	**1.000**
liblouis	CVE-2017-13741	**0.828**	**0.937**	**0.851**	**1.000**	**1.000**	**0.984**	**0.875**	**0.937**	**0.867**	**1.000**	**1.000**	**0.968**
lrzip	CVE-2018-11496	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
Mini XML	CVE-2018-20592	**0.968**	**1.000**	**0.812**	**1.000**	**1.000**	**1.000**	0.617	**0.929**	**0.750**	**0.781**	**1.000**	**0.781**
boringssl	Google Test-suit	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.828**	**1.000**	**1.000**	**1.000**
GNU cflow	uaf-issue-1	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.937**	**0.968**	0.609	**0.968**	**1.000**	**1.000**
Boolector	uaf-issue-2	0.720	**1.000**	**0.030**	**0.880**	**1.000**	**0.780**	0.620	**1.000**	**0.020**	**0.820**	**1.000**	0.720
openh264	uaf-issue-3	**0.937**	**0.781**	**0.150**	**1.000**	**1.000**	**1.000**	0.687	0.359	**0.031**	0.640	**1.000**	**0.875**
libpff	uaf-issue-4	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
mjs	uaf-issue-5	**0.980**	**0.980**	0.590	**1.000**	**1.000**	**1.000**	**0.880**	**0.890**	0.604	0.987	**1.000**	**0.987**
ImageMagick	uaf-issue-6	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
nasm	CVE-2018-19216	0.960	0.600	0.560	**0.920**	0.600	0.800	**0.800**	0.319	0.280	**0.840**	0.280	0.800
	CVE-2018-20535	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.500	0.500	0.500	0.500	0.500	0.500
统计学意义有效		11/15	12/15	9/15	12/15	12/15	12/15	9/15	9/15	7/15	9/15	11/15	10/15

* $\hat{A}_{12}$ 被标记为粗体则代表着相应的 *Mann-Whitney U* 检验结果说明 MemLock 与其对比的工具的发现漏洞所需的时间差异显著；统计学意义有效统计的是 $A_{12} > 0.71$ 且被用粗体标记的数量。

来增强对内存时序漏洞的检测能力。为了分别评估这两个策略的有效性，本章配置了一个未采用基于信息流分析的变异优化策略的 UAFL 的版本 UAFL$_{NIF}$，即 UAFL$_{NIF}$ 只包含操作序列导向的反馈机制，而不包含基于信息流分析的变异优化策略。UAFL$_{NIF}$ 的实验结果分别位于表4-3中的UAFL$_{NIF}$ 列和表4-4中的 $\hat{A}_{12}$ (UAFL$_{NIF}$) 列。实验结果表明，UAFL$_{NIF}$ 整体上要优于其它六个模糊测试工具。UAFL$_{NIF}$ 发现每个内存时序漏洞的平均时间消耗为 5.12 个小时，即使算上静态分析的时间开销（即平均 0.32 小时），其发现漏洞的时间开销仍然比其它六个模糊测试技术要小得多。UAFL$_{NIF}$ 发现内存时序漏洞的速度分别是 AFL、AFLFast、FairFuzz、MOpt、Angora 和 QSYM 的 3.25 倍、3.16 倍、2.63 倍、3.35 倍、6.00 倍和 3.80 倍。同样，统计学假设检验的结果也表明，在一共 90 组假设检验中，UAFL$_{NIF}$ 有 55 组（61.1%）$\hat{A}12$ 值超过了 0.71，同时其 P 值小于 0.05，因此在大多数实验对象中，UAFL$_{NIF}$ 在发现内存时序漏洞上显著优于其他 6 个业界先进的模糊测试技术，这充分体现了操作序列导向的反馈机制的有效性。

本章还将UAFL和UAFL$_{NIF}$ 进行比较。对于表4-1中所有的实验对象，UAFL 发现内存时序漏洞的时间消耗都要明显小于 UAFL$_{NIF}$。整体而言，UAFL 发现漏洞的速度相比 UAFL$_{NIF}$ 平均提高了 1.75 倍。值得注意的是，对于 *nasm* 程序，UAFL$_{NIF}$ 无法在 24 小时内找到其中的 UaF 漏洞（即 CVE-2018-20535），而 UAFL 通过引入基于信息流的变异，可以在 17.03 小时内发现该漏洞；对于 *GNU cflow* 程序，UAFL$_{NIF}$ 发现 UaF 漏洞花费了 12.21 小时，而 UAFL 仅需要 1.80 小时，这也充分体现了基于

信息流的变异优化策略能够提升发现内存时序漏洞的效率。

实验结论：UAFL 中使用的操作序列导向的反馈机制和基于信息流分析的变异优化策略都是有效的。从 UAFL_{NIF} 与其它模糊测试工具的比较结果可以看出操作序列导向的反馈机制有效使得发现内存时序漏洞的能力更强；而从 UAFL 和 UAFL_{NIF} 的比较结果能够说明基于信息流的变异优化策略提升了发现内存时序漏洞的效率。

4.3.3.4 实验四：覆盖率评估

本章统计了各个模糊测试工具对于静态分析阶段识别出的操作序列相关代码的覆盖情况，其主要原因是：（1）与操作序列相关的代码更有助于帮助检测内存时序漏洞；（2）该代码覆盖情况可以评估生成的测试用例到触发漏洞的输入的接近程度，从而揭示生成的测试用例的质量。

图 4-6显示了 UAFL 、UAFL_{NIF} 、AFL、AFLFast、FairFuzz、MOpt、Angora 和 QSYM 达成的操作序列相关代码的覆盖率随时间的变化情况。在前 1 至 3 个小时的阶段，八个模糊测试工具都显示出了相似的代码覆盖增长情况，这是由于此时搜索到的路径不多，基于初始种子进行变异生成大量的测试用例即可覆盖大部分的代码分支。而随着时间的推移，UAFL 和 UAFL_{NIF} 中的代码覆盖率比其它六个模糊测试技术增长得更快。在 24 小时结束时，UAFL 和 UAFL_{NIF} 基本达到了八个模糊测试工具中最佳代码覆盖率的前两名。

代码覆盖率反映了目标程序测试时实际执行到的代码范围，UAFL 和 UAFL_{NIF} 覆盖的与内存时序漏洞操作序列相关的代码更多，这也是它们在检测内存时序漏洞方面更有效的重要原因。此外，注意到表 4-3中其它模糊测试工具的性能表现，FairFuzz 的平均性能较 AFL、AFLFast、MOpt、Angora 和 QSYM 更优。在与操作序列相关的代码的覆盖率表现上，FairFuzz 无论是覆盖率的增长速度，还是最终达成的覆盖率情况，都较其它模糊测试工具要更好，这也说明了内存时序漏洞操作序列相关的代码覆盖率和发现内存时序漏洞的能力是密切相关的。另一方面，内存时序漏洞的触发不仅依赖于覆盖所有相关的内存使用操作，还要求按照一定的时序去覆盖这些操作。为此，本章也人为调查了各个模糊测试工具种子池中的测试用例对操作序列的覆盖情况，例如，尽管 FairFuzz 对内存时序漏洞操作序列相关的代码能达到较高的覆盖率，但仍难以覆盖整个操作序列。又例如，Angora 虽然在程序 *readelf 2.28* 达成了最佳的代码覆盖率，然而，因为 Angora 只分别覆盖了操作序列中的单个内存使用操作，而未覆盖整个操作序列，所以 Angora 并未发现其中的 UaF 漏洞。因此，其它模糊测试技术在发现内存时序漏洞方面不如 UAFL 有效。

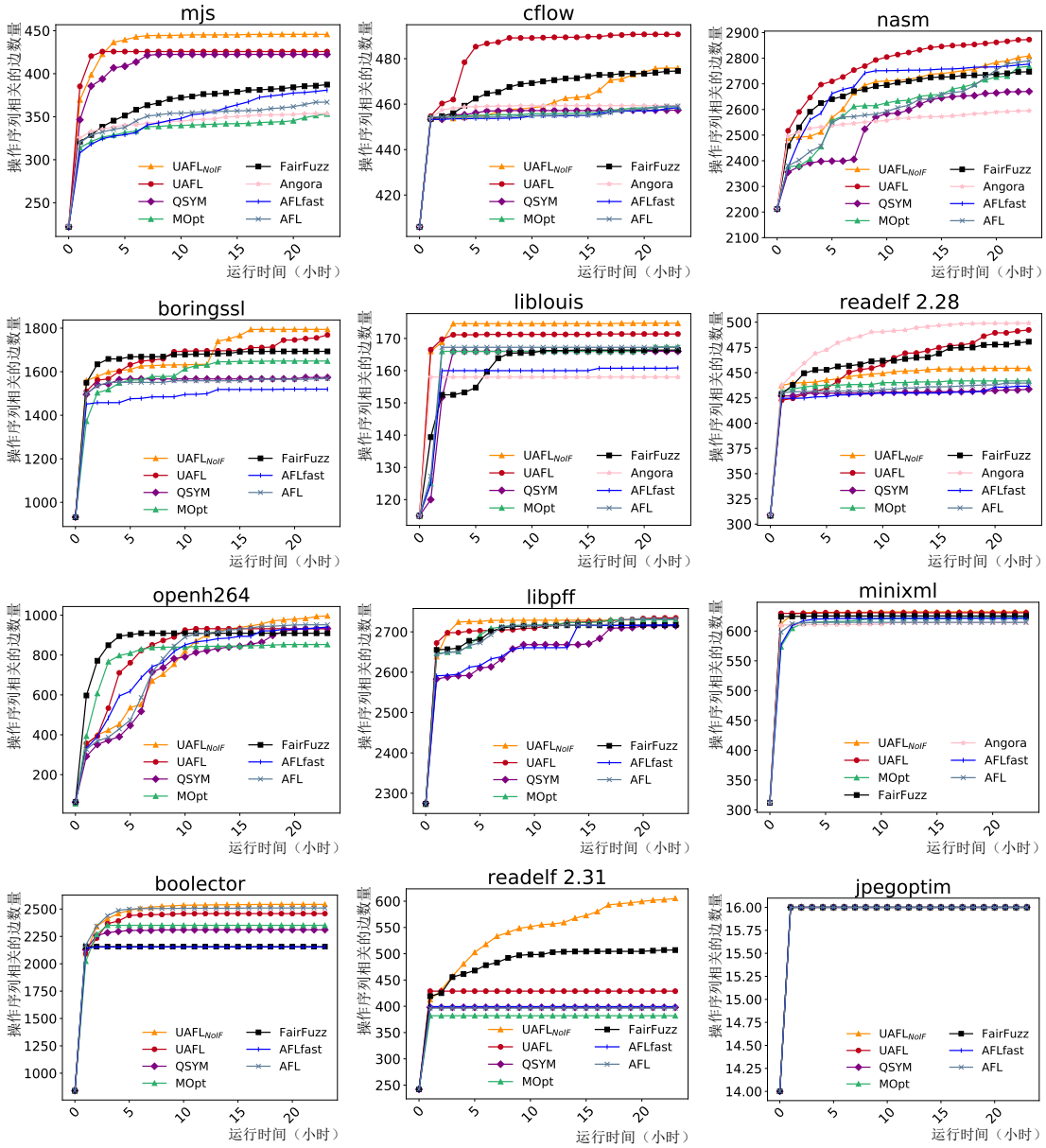


图 4-6　对静态分析阶段识别出的操作序列相关代码的覆盖率

实验结论：与其它六个业界先进的模糊测试技术相比，UAFL 和 UAFL$_{NIF}$ 达成的与内存时序漏洞相关操作序列的代码覆盖率更高，这也是 UAFL 和 UAFL$_{NIF}$ 在检测内存时序漏洞方面更有效的主要原因。

4.3.3.5　发现的新漏洞

本章将 UAFL 应用于现实生活中广泛使用的真实应用程序（例如 *libpff*、*GNU cflow*、*ImageMagick*、*libheif*、*Binaryen*、*Mini-XML* 和 *Elfutils* 等），并发现了一些未知的、安全攸关的内存时序漏洞，包括 6 个释放后使用漏洞和 1 个双重释放漏洞。本文作者线上提交了漏洞报告，并帮助开发和维护人员一起定位和修复漏洞。MITRE 组织为作者提交的漏洞分配了 7 个 CVE ID，如表 4-5所示。攻击者可能会通过提供构

表 4-5　UAFL 新发现的内存时序漏洞（7 CVEs）

CVE ID	程序	漏洞类型
CVE-2020-18897	*libpff 894a4*	CWE-416: Use After Free
CVE-2019-16165	*GNU cflow v1.6*	CWE-416: Use After Free
CVE-2019-15140	*ImageMagick v7.0.8*	CWE-416: Use After Free
CVE-2019-11471	*libheif v1.4.0*	CWE-416: Use After Free
CVE-2019-7703	*Binaryen v1.38.22*	CWE-416: Use After Free
CVE-2018-20592	*Mini-XML v2.12*	CWE-416: Use After Free
CVE-2018-16402	*Elfutils v0.173*	CWE-415: Double Free

思良好的输入使得应用程序违反内存时序属性，并利用漏洞发起拒绝服务（DoS）攻击。通过作者的漏洞报告和漏洞披露之后，这些漏洞被开发者积极地修复；在撰写本文时，这 7 个内存时序漏洞已经全部被修复。本章在真实的应用程序中新发现的这些内存时序漏洞也体现了 UAFL 在实践中是切实有效和可行的。

4.3.4　有效性威胁分析

基于以上实验，本章从以下两个方面来分析影响本章实验结论有效性的因素：

内部因素：本章的方法主要基于静态分析和模糊测试技术，这两个技术本身还面临着一些挑战或存在一些固有的局限性。本章使用的静态分析技术旨在找出程序中潜在的违反内存时序属性的操作序列，理想情况下，该过程应该是上近似的，即所有的违反内存时序属性的操作序列都成功被识别出来。然而该过程仍面临一些挑战，例如别名分析、流敏感和上下文敏感的指向分析等，当操作序列未被正确识别的时候，可能导致后续模糊测试阶段无法发现相应的内存时序漏洞。当前 UAFL 在实现的时候依赖于 SVF 提供的安德森指针分析以及数值流图的构建，今后该部分可以被替换成性能更优的静态分析算法，以进一步提升静态分析阶段的准确性和效率。本章使用的模糊测试技术在生成测试用例时具有一定的随机性，尽管本章对每个实验对象和每个对比工具都进行 8 次长时间的重复实验以减轻随机性带来的影响，并进行了统计学意义上的分析，但这种随机性仍然可能对实验结论存在一定的影响，接下来，可以增加重复实验进行的次数，以提高本章结论的可靠性。

外部因素：影响本章实验结果的外部因素主要在于选取实验对象时引入的样本偏差。本章选择了 14 个现实世界的的应用程序，它们具有不同的功能，在其它研究工作中经常被用作对比基准，并且包含 15 个真实的内存时序漏洞。本章还选取了六个业界先进的模糊测试技术与 UAFL 进行比较。然而，本章选取的实验对象和对比基准可能包括一定的样本偏差，仍不能保证本章结论对所有的应用程序

都有效。此外，本章选取的实验对象所包含的漏洞类型为释放后使用和双重释放漏洞，这是两类最具代表性的内存时序漏洞，本章对于其它类型的时序漏洞（例如重复释放同一个资源或句柄[187]、使用过期的文件句柄[188]等）仍缺乏实验评估。但作者认为本章方法具有通用性，可以通过扩展内存时序属性很容易地扩展到其它类型的时序漏洞上。因此，下一步将寻找更多具有不同特征且来自不同领域的真实应用程序来评估UAFL，以及扩展UAFL以支持检测更多类型的内存时序漏洞，提高本章结论的普适性。

4.4 本章小结

本章针对内存时序漏洞的检测问题，提出了一种融合程序分析与模糊测试的内存时序漏洞检测方法UAFL。UAFL将内存使用的安全时序规则建模成具有一定类型状态属性的状态机模型，使用静态分析识别潜在的违反了该类型状态属性的操作序列。然后，UAFL通过操作序列导向的模糊测试，结合基于信息流分析的变异优化策略，逐步生成能够覆盖完整的操作序列的测试用例，以触发内存时序漏洞。本章实现了UAFL工具原型，并使用了14个现实世界中被广泛使用的开源程序对其进行实验评估。实验结果表明，UAFL在发现内存时序漏洞方面要优于当前业界先进的基于模糊测试的漏洞检测技术。相比AFL、AFLFast、FairFuzz、MOpt、Angora、QSYM方法，UAFL发现的内存时序漏洞数量要多于25%，并且实现了至少2.63倍的提速。此外，UAFL还在真实应用程序上新发现了7个安全攸关的内存时序漏洞（7 CVEs）。

本章的主要研究成果撰写的论文《Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities》已在软件工程领域的顶级国际会议ICSE ’20上发表。

第 5 章　融合程序分析与受控并发测试的内存并发漏洞检测技术

5.1　引言

5.1.1　研究背景

随着多核处理器的普及和高级编程语言对并发编程的支持，并发程序的开发与使用日益广泛。并发程序通常由多个线程组成，通过多个线程之间的相互协作完成复杂任务，能够显著提升程序的资源利用率与计算效率。但是，并发编程的门槛也相对更高，编写正确的并发程序并不容易，因此更容易引入并发漏洞[37]。内存并发漏洞与串行程序中的内存安全漏洞不同，内存并发漏洞往往更难被发现。串行程序的行为不确定性主要来自于输入，而并发程序的行为不确定性不仅与输入有关，还取决于它执行过程中多个线程是如何交错执行的。因此并发程序的状态空间更大，并发漏洞的隐蔽性和潜伏性更大，其检测难度也更大[131,189,190]。

传统的压力测试对于同一个测试输入，在一定的负载下对给定的并发程序持续进行重复测试，期望某次测试能够暴露并发漏洞。压力测试方法非常简单易行，然而在测试的过程中，由于没有提供线程调度控制，很多相同的线程交错情况可能被多次重复执行，而某些能暴露并发漏洞的线程交错可能一次都未执行到，因此该方法发现并发漏洞的概率与效率都非常低下[191,192]。数据竞争给并发程序带来潜在的风险，是很多内存并发漏洞产生的主要原因，因此一些研究人员提出了不少基于程序分析技术的数据竞争检测方法[108,193,194]。然而，数据竞争既不是内存并发漏洞产生的充分条件也不是必要条件。有研究显示[106]，90% 的数据竞争都是良性的、无害的，并且内存并发漏洞也存在于无数据竞争的并发程序中[107]。

为了测试执行更多的线程交错情况，受控并发测试技术（Controlled Concurrency Testing）被研究人员广泛研究[113,144,195–198]。受控并发测试通过控制线程调度，探索并发程序的调度空间，每次测试执行一个不同的线程交错，并监视其执行是否发现异常。受控并发测试会在被测程序的“调度点”之前中插入一段用于控制线程调度的程序代码，这些调度点通常是程序中可能因并发执行交错的顺序而影响程序行为的一些关键语句（例如对共享内存的读写操作，以及锁、信号量等并发编程中的同步原语）。受控并发测试每次会自动生成一个调度决策，并根据调度决策来控制不同线程之间的调度点以何种顺序交错地执行。若在并发程序执行的过程中触发了漏洞，那么将本次执行时的调度决策记录下来，以用于确定性地重现并发漏洞[199–201]。

5.1.2 现存问题

受控并发测试面临的一个关键挑战是如何控制线程调度。只有使线程调度受控制，才能通过调度决策来控制调度点之间的执行顺序。尽管并发程序中的多个线程是并发执行的，但根据每个调度点执行的物理时钟，可将并发程序的每次执行都看作是一个序列化的执行过程，即同一时间只有一个线程在执行。当该线程执行到下一个调度点时，可以根据调度决策来决定是继续执行当前线程，还是选择另一个线程继续执行（这也称“线程切换”）。因此调度决策实际上决定了并发程序在执行到哪个调度点时进行一次线程切换。现有的研究工作通常采用基于时延[108,192]、抢占式调度[114]、优先级调度[107,109,112]等方式来控制线程调度。基于时延的线程调度需要在不同的调度点之前引入一段时延，以使某些调度点的执行顺序发生在时延结束以后，这会显著地降低并发程序的执行速度，使得有限的时间内能探索的线程调度更少。基于抢占式的线程调度通常需要在被测程序中引入抢占式的同步操作（yield 操作、wait-signal 操作、锁、信号量等），新引入的同步操作可能会与被测程序中的同步操作产生病态的交互。例如新引入的 wait-signal 操作可能会与被测程序中的互斥锁产生死锁，该死锁在被测程序中原本是不会发生的，因而是个误报。基于抢占式的线程调度和基于优先级的线程调度通常不能跨核使用，因此一般采用单核调度技术，这类调度技术要求将多个线程强制在一个核上执行，即并发程序在执行时的任意一个时刻仅有一个线程在占用当前的核，会影响程序的执行效率。此外，线程调度方案的选择也影响着调度空间的探索方式，例如对于基于时延的线程调度可以仅考虑在哪些调度点引入一段固定时长的时延；而对于单核调度技术而言，在做调度决策时不仅需要考虑在哪些调度点进行线程切换，还需要考虑切换到哪个线程，故若要尝试分析所有的线程调度顺序也会显得效率低下。因此，仍然需要研究一种更有效的线程调度控制方式。

受控并发测试面临的另一个关键挑战是如何有效地探索线程调度空间。一种直观的探索方法就是枚举所有可能的调度决策，但是需要探索的调度决策的数量随着单个线程中调度点的数量呈指数增长，实际上，一个并发程序的调度空间的探索问题是 NP-hard 的，因此对于现实世界中的并发程序，枚举所有可能的调度决策是不切实际的。近年来，研究人员提出了限定调度技术[147,196]（Schedule Bounding）来缓解这一难题。限定调度技术为一次调度决策中的时延点、抢占点或优先级改变次数（即线程切换次数）设置了一个界限值，将那些线程切换次数大于该界限值的调度决策都忽略，这极大地减小了需要探索的调度空间。由于现实世界中并发漏洞的深度（见定义 2.20）都较小[148,149]，通常不超过 3，因此，限定调度技术

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

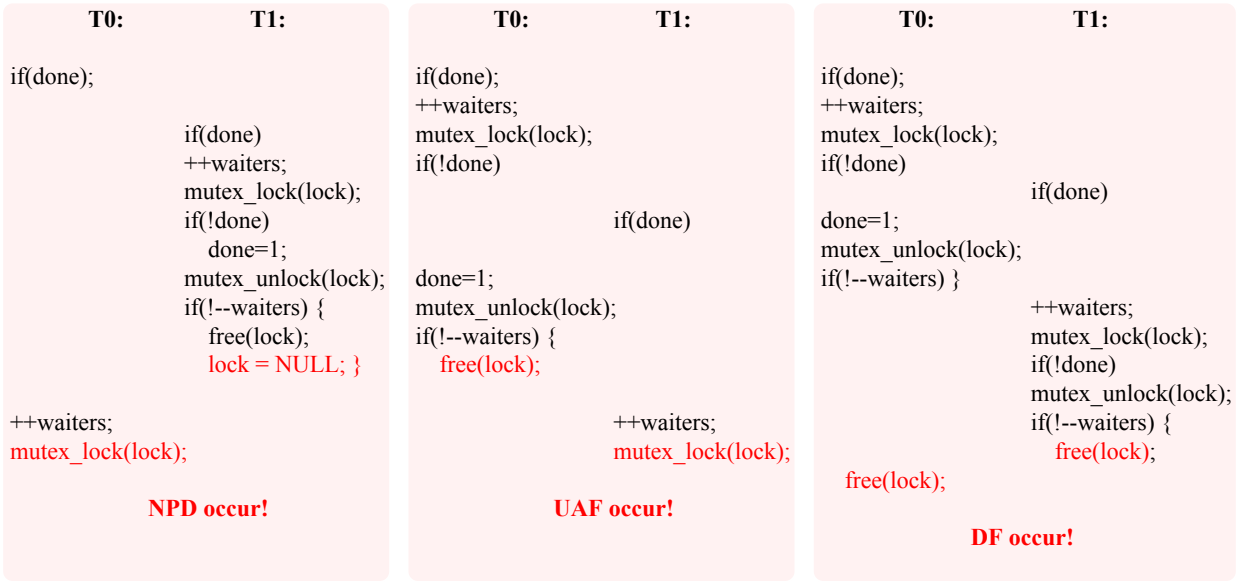
图 5-1　一个从 *CVE-2016-1972* 漏洞简化而来的启发性示例（来自 *Firefox* 浏览器中的库 *libvpx*）

对于现实世界中的并发程序通常是非常有效的。现有的研究工作探索调度空间的方式主要分为随机性或系统性。随机测试（例如 PCT[112] 和 RPro[113]）以随机的方式生成调度决策，每个调度决策都随机地选择调度点进行线程切换。随机性的测试只能为并发漏洞的检测提供概率性的保证，它无法保证某个特定的并发漏洞一定能在某段时间内被发现。系统性的测试（例如 IPB[114,196] 和 IDB[149,196]），也称无状态模型检测[202]，它基于一个预设的规则（例如限定调度技术设定的界限值），尝试测试该规则下所有可能的调度决策。理论上，系统性的测试在测试了一定数量的调度决策后，可以保证小于某个深度的并发漏洞一定能被发现，但它们往往会要求比实际需求更大的调度空间，以致于需要更多开销或漏报一些错误。因此，无论是随机性的测试还是系统性的测试，当前它们在探索线程调度空间方面的有效性仍不够。

图 5-1中的代码片段是漏洞 CVE-2016-1972 的简化版本，该漏洞来自 *Firefox* 浏览器中的库 *libvpx*。该程序在主线程中创建了两个以函数 once 为入口的子线程（第 22-23 行）。函数 once 使用三个共享变量（即 lock、waiters、done）进行同步，旨在确保其中处理的业务流的代码仅在一个线程中被执行（第 10 行）。变量 lock 是一个互斥锁，它在主线程刚开始的时候被创建（第 20 行），并在子线程中被释放（第 15

行），其作用就是为了防止多个线程同时对临界区域（第 9-13 行）中的共享变量进行读写。变量 done 用于配合互斥锁期望确保最多只有一个线程能执行临界区域。一旦临界区域的业务流代码执行完毕，变量 done 将被立即设置为 1（第 11-12 行），在此之后仍未开始执行的线程将直接返回。变量 waiters 表示仍未执行完临界区域代码的线程数（第 8 行），用于限制一些内存释放和置零操作仅对最后一个退出的线程生效（第 14-16 行）。注意，除了第 7 行的“return 0;”语句之外，函数 once 的其他语句要么是对共享变量的读写操作，要么是使用了并发同步原语的同步操作，因此这些语句都是调度点。

尽管函数 once 的同步机制是经过精心设计与实现的，但由于并发编程不太符合人的思维习惯，多线程之间的交互行为很难被考虑周全，函数 once 仍然存在着三种不同类型的内存并发漏洞，分别是并发空指针解引用（并发 NPD）、并发释放后使用（并发 UaF）和并发双重释放（并发 DF）。具体来说，当变量 lock 被一个线程设置为空以后（第 16 行），另一个线程紧接着继续使用这个变量 lock（第 9 行），那么就会触发空指针解引用漏洞（如图 5-2(a)）；当变量 lock 在一个线程中被释放以后（第 16 行），另一个线程紧接着继续使用这个变量 lock（第 9 行），那么就会触发释放后使用漏洞（如图 5-2(b)）；当变量 lock 分别在两个线程中连续被释放（第 15 行），那么就会触发双重释放漏洞（如图 5-2(c)）。这三种并发漏洞都是仅在特定的线程交错下才会被触发，即要求并发程序以特定的顺序去执行对变量 lock、waiters、done 的操作，因此，这三种并发漏洞的检测都是相当困难的。



(a) 并发空指针解引用　　(b) 并发释放后使用　　(c) 并发双重释放

图 5-2　CVE-2016-1972 启发性示例中三种不同类型的内存并发漏洞

本文使用了一些当前主流的程序分析与测试工具来检测该示例程序中的内

存并发漏洞。Infer[24] 和 cppcheck[48] 是用于检测内存安全错误的通用的静态分析工具，它们缺乏对多线程语义的建模，将该程序当作串行程序来分析，故未检测出任何漏洞。AFL[30] 作为目前最前沿，最先进的模糊测试工具之一，本文使用 AFL 对该程序持续测试 24 小时，但并未检测到任何崩溃。由于该程序的行为不受输入的影响，用 AFL 测试该程序实际上相当于持续重复执行该程序，这与第 5.3.3.2节中使用原生执行（Native）对该程序进行测试的结果一致。TSAN[51] 是一款当前主流的并发漏洞检测工具，但它仅报告第 6 行对变量 done 的读操作与第 12 行对变量 done 的写操作之间存在数据竞争，然而变量 done 是开发人员故意设计的用于同步机制的变量之一，故该数据竞争很容易被误认为是良性的数据竞争。而且，示例程序中的内存并发漏洞与变量 lock、waiters 和 done 之间的交互密切相关，仅报告变量 done 存在数据竞争对识别这些内存并发漏洞的帮助十分有限。UFO[105] 是一款基于预测分析的并发 UaF 漏洞动态检测工具，可以在一定约束条件下对程序运行时所记录的执行轨迹中的事件进行重排序以预测其它线程交错中的 UaF 漏洞。然而，UFO 同样未发现示例程序中的并发 UaF 漏洞。一方面由于 UFO 使用了宽松的最大因果关系模型，导致 UFO 会漏报一些 UaF 漏洞。另一方面 UFO 仅能对执行轨迹中出现的事件进行重排序，故只能探索一定范围内的线程交错情况。当示例程序中的一个线程先执行完后，另一个线程再开始执行，即仅允许一次线程切换的情况下，后执行的线程会因为变量 done 已被设置为 1 而在第 7 行直接返回；由于后执行的线程未覆盖到除了对变量 done 以外的任何读写操作，预测分析技术无论如何对当前轨迹中的事件进行重排序，也无法触发并发 UaF 漏洞。

受控并发测试技术对于并发程序的调度空间探索是比较有效的。IPB[114] 和 IDB[149] 是两个具有代表性的系统性的测试技术。IPB 和 IDB 在一万次测试执行之内都成功发现了示例程序中的并发 NPD 和并发 UaF 漏洞，但未发现并发 DF 漏洞。由于并发 DF 漏洞的深度为 5（即该漏洞的触发至少需要 5 次线程切换），相比漏洞深度分别为 2 和 4 的并发 NPD 和并发 UaF 漏洞，并发 DF 漏洞的触发要求对调度空间进行更加充分的探索，这也是说，这个并发 DF 漏洞更难被发现。该示例程序包含两个线程，且每个线程最多可能有 9 个调度点，故该程序的调度空间大小为 $(9+9)!/(9!\times 9!) = 48,620$。IPB 和 IDB 需要迭代地在每个调度点上分别尝试进行线程抢占或线程延迟操作，在如此大的调度空间内搜索可能触发的内存并发漏洞，这个代价是相当大的。而且，IPB 和 IDB 的界限值并不能切实地反映线程切换次数，它们通常需要将调度的界限值设置得比漏洞深度还要大一些才能触发并发漏洞，这也使得它们实际上需要探索更大的调度空间。PCT[112] 在限制线程切

换次数的情况下随机地探索线程调度空间。在本章的实验中，PCT 约有 7% 的概率发现并发 NDP 漏洞，有 0.15% 的概率发现并发 UaF 漏洞，但同样未发现并发 DF 漏洞。由此可见，随机性的测试技术检测到示例程序中内存并发漏洞的概率是非常低的。Maple[203] 基于 6 种预定义的线程交错模式（Interleaving Idiom），启发式地引导线程调度按照其中一种预定义的线程交错模式去执行。这些线程交错模式仅包含该示例程序中并发 NPD 的线程交错模式，而不包括并发 UaF 和并发 DF 的模式，因此 Maple 可以检测到该示例程序中的并发 NPD 漏洞，但漏报了并发 UaF 和并发 DF 漏洞。综上所述，通过这个示例程序可以看出，当前主流的程序分析与测试技术在内存并发漏洞的检测方面仍存在不足。

5.1.3 本章主要工作

本文提出了一种融合程序分析与受控并发测试的内存并发漏洞检测方法 PERIOD，该方法有效地控制线程调度，并系统性地探索所有可能的线程交错情况，旨在对由线程交错引起的内存并发漏洞进行自动化检测。PERIOD 使用一连串执行周期来托管并发程序的执行；当一个执行周期结束后，下一个执行周期才可以开始，其中线程切换可以通过周期间的切换自然地完成。PERIOD 使用了一个高效的、不会与并发同步原语产生病态交互的周期性调度方案，将并发程序中的调度点分配到不同的执行周期中，能一定程度上确定性地控制并发程序的线程交错情况，并且最大程度地保证并发程序的并发执行性。基于对线程的控制，PERIOD 通过自动化地生成调度决策，对目标程序进行多次重复性的测试，且每次测试尽可能地测试一个没有测试过的线程交错，以便发现尽可能多的内存并发漏洞。为了提升测试的效率，PERIOD 首先采用轻量级的静态程序分析来找到潜在的可能导致并发漏洞的调度点，然后针对这些调度点的并发交错情况，采用基于动态切片和调度前缀的调度空间探索方法来指导调度决策的生成，以有效地测试不同的线程交错情况。此外，PERIOD 采用了并发调度决策生成策略，该策略在控制一部分调度点的执行交错的情况下，允许剩余的调度点不受控地并发执行，这不仅提升了测试执行的速度，还极大程度地减小需要探索的调度空间。本章实现了 PERIOD 的工具原型，并使用了 10 个现实世界的 CVE 程序和 36 个基准数据集中的程序对 PERIOD 进行评估。实验结果表明，PERIOD 在发现内存并发漏洞方面要显著优于当前业界先进的受控并发测试技术。具体来说，PERIOD 相比 IPB、IDB、DFS、PCT、Random、Maple 等方法在同样时间内发现的内存并发漏洞的数量要多于 29%。此外，PERIOD 还在现实世界的主流开源应用程序中新发现了 5 个内存并发漏洞，也进一步证明了它的实用价值。

本章的其它章节安排如下：首先，第5.2节详细介绍本章提出的内存并发漏洞检测方法，包括整体流程、静态分析与程序插桩、受控并发测试三个部分；接着，第5.3节从调度决策生成策略、漏洞检测能力、运行时开销等方面对本章方法进行实验评估与分析；最后，第5.4节对本章进行总结。

5.2 内存并发漏洞检测方法

5.2.1 整体流程

本章提出的内存并发漏洞检测方法PERIOD的整体工作流程如图5-3所示。该方法主要包括两个阶段：①静态分析与程序插桩；②受控并发测试。

静态分析与程序插桩旨在识别并发程序中的调度点，并基于这些调度点对目标程序进行插桩。具体来说，PERIOD首先通过静态分析方法遍历每个线程的每条程序语句，并判断该程序语句是否对一个共享数据进行读写操作，或该程序语句是否调用了互斥锁、条件变量、信号量等并发同步原语，若是，则将该语句识别为调度点。接着，PERIOD在被测程序的“调度点”之前插入一段基于周期性执行控制线程调度的程序代码，以便控制被测程序按照调度决策来运行并获取调度点的覆盖情况（和动态调度点切片）。

受控并发测试阶段包括三个关键步骤：I. 调度决策生成、II. 周期性执行和III. 反馈信息分析。调度决策生成步骤旨在系统性地为目标程序的动态调度点切片生成调度决策，并将其反馈给周期性执行步骤。周期性执行步骤通过控制目标程序的线程调度，使得目标程序的线程交错遵循调度决策中预设的线程交错执行顺序。周期性执行步骤还负责收集运行时信息，例如覆盖的调度点的数量、动态调度点切片和调度前缀。反馈信息分析步骤利用收集到的运行时信息来指导生成合法有效的调度决策，以覆盖更多未测试的线程交错。具体来说，PERIOD使用一连串执行周期来托管并发程序的执行，并采用周期限定技术来探索可能的线程交错。PERIOD需要预设一个周期数的上限P，旨在检测漏洞深度小于P的漏洞。PERIOD先从最浅层的并发漏洞（即漏洞深度为1）开始，即从周期数2开始探索。为了使得调度决策能够反映一个实际可行的线程交错情况，PERIOD引入动态调度点切片来表示目标程序在运行时所覆盖的各个线程的调度点的动态切片，且从每个线程只有一个调度点的动态调度点切片开始探索。由于动态调度点切片可以反映每个线程执行到的一条具体可行路径，基于动态调度点切片生成的调度决策更加精确，避免了不必要的探索。对于当前的周期数p，PERIOD为当前每个不同的动态调度点切片创建一个探索任务。对每个探索任务，PERIOD通过步骤I为当前的动

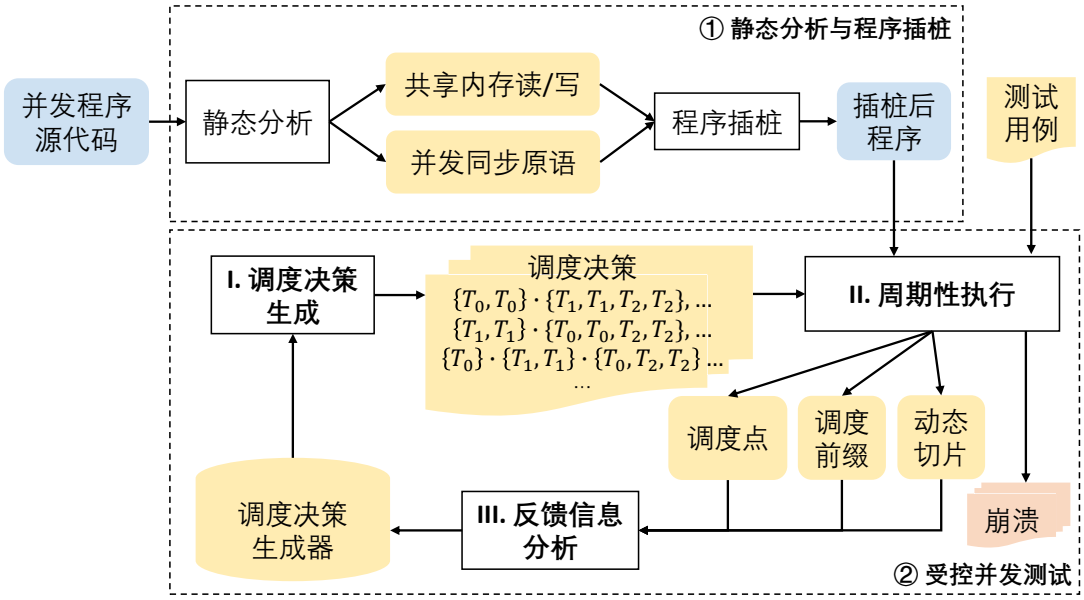


图 5-3　PERIOD 的整体流程

态调度点切片系统性地生成调度决策，然后在每一个调度决策的指导下通过步骤 II 来控制目标程序的执行并收集反馈信息，最后通过步骤 III 对所收集的信息进行分析。如果发现了新的动态调度点切片，那么为其创建一个新的探索任务。此外，为了避免重复探索，PERIOD 会分析和推断一个能够到达新切片的调度前缀（当前调度决策的一部分），并使用其指导后续的调度决策的生成。一旦完成当前周期数 p 的所有探索任务，PERIOD 将当前周期数 p 增加 1（即 $p=p+1$）以继续生成更多的调度决策对目标程序进行测试，直到周期数 p 达到预设的上限 P。PERIOD 由浅入深地、循序渐进地探索调度空间，且有针对性地测试可行的线程交错情况，因此能够高效地发现并发漏洞。

接下来，本文将通过图 5-1中的示例来解释本章方法是如何发现其中的三种内存并发漏洞的。

PERIOD 在第①阶段首先对示例程序进行静态分析，识别程序中的调度点。除了第 7 行的 “return 0;” 语句之外，函数 once 的其它语句都被识别为调度点。PERIOD 第②阶段以起始的动态调度点切片 $s_0=[T_0{:}[6],T_1{:}[6]]$ 开始测试图 5-1中的示例程序，其中 T_0 和 T_1 分别是两个不同线程的标识符，这与源代码第 22-23 行创建的线程名保持一致，$T_0{:}[6]$ 和 $T_1{:}[6]$ 表示每个线程仅第一个调度点（即代码第 6 行）被覆盖。调度决策生成步骤首先为 s_0 生成 2 个周期内的调度决策，调度决策可以形式化地表示为 $\{\ldots\}\cdot\{\ldots\}\ldots\cdot\{\ldots\}$，其中 $\{\ldots\}$ 表示一个包含已分配调度点的执行周期，周期与周期之间通过 $\cdot$ 隔开。PERIOD 分别将两个调度点分配到不同的周期，因而生成两个调度决策：$\{T_0{:}6\}\cdot\{T_1{:}6\}$ 和 $\{T_1{:}6\}\cdot\{T_0{:}6\}$。由于调度决策是基于动态调度

点切片中各个线程的调度点数量生成的，它关注的是各个线程的调度点之间的执行交错的顺序，故调度决策只需说明哪个线程有多少个调度点被分配在哪个周期即可。本文在接下来的内容中省略了调度点对应的代码行号，并按照准字典的方式排序生成的调度决策，对于 s_0 生成的两个调度决策可以表示为：$\{T_0\}\cdot\{T_1\}$ 和 $\{T_1\}\cdot\{T_0\}$。

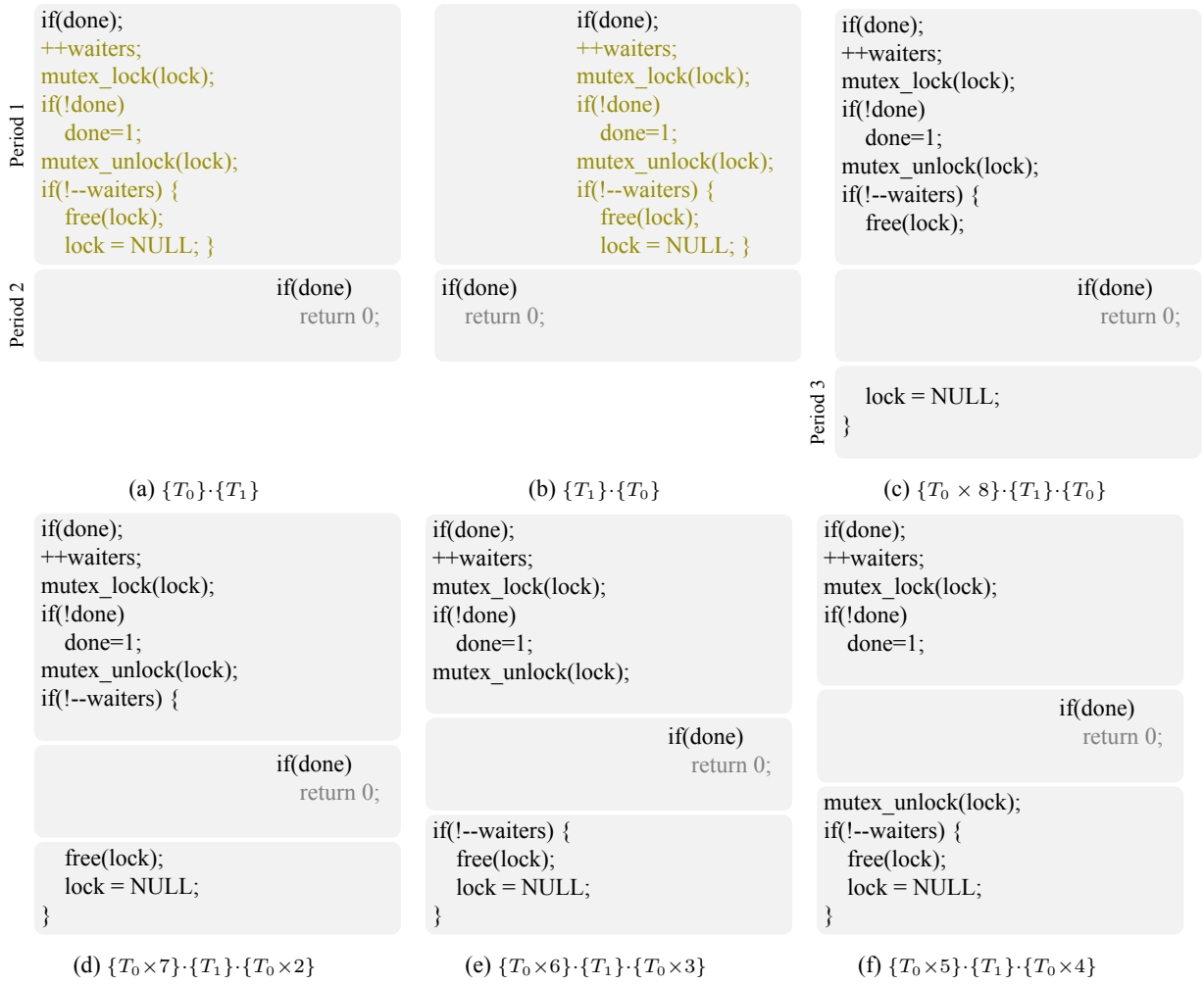


图 5-4　CVE-2016-1972 示例程序的线程交错情况。每个子图分别描述了一个调度决策对应的动态执行轨迹

在 s_0 生成的两个调度决策的分别指导下，PERIOD 通过周期性执行来控制示例程序中调度点的执行顺序。图 5-4(a) 描述了调度决策 $\{T_0\}\cdot\{T_1\}$ 下示例程序的执行轨迹，其中线程 T_0 的第一个调度点 “if(done)” 被安排在第一个执行周期执行，由于其它调度点不受控制，线程 T_0 的其它调度点将与该线程的最后一个受控制的调度点在同一周期内执行，图 5-4(a) 的橄榄色语句描述了新覆盖的调度点；线程 T_1 的第一个调度点 “if(done)” 被安排在第二个执行周期执行，由于线程 T_0 已将变量 done 设置为 1，线程 T_1 将直接通过 “return 0” 返回。在分析了反馈信息之后，可以得到新的动态调度点切片 $s_1 = [T_0{:}[6,8,9,11,12,13,14,15,16],\ T_1{:}[6]]$，其中

新的调度点是图 5-4(a) 中新覆盖的调度点对应的源代码行。同样，如图 5-4(b) 所示，PERIOD 通过执行调度决策 $\{T_1\}\cdot\{T_0\}$，得到了另一个不同的动态调度点切片 $s_2 = [T_0{:}[6],\ T_1{:}[6, 8, 9, 11, 12, 13, 14, 15, 16]]$。

当测试过程中发现了新的动态调度点切片，意味着当前对调度空间的探索还不够充分，仍有许多可行的线程交错未被覆盖到，需要扩大探索的调度空间的范围。PERIOD 通过分析历史执行过的调度决策和覆盖了 s_1 和 s_2 的调度决策，可以推断出 s_1 和 s_2 的调度前缀分别为 $[T_0]$ 和 $[T_1]$，这两个调度前缀包含了到达 s_1 和 s_2 所需的线程切换。调度前缀的一般形式是 $\{\ldots\}\ldots\cdot\{\ldots\}\cdot[T_i]$，其中花括号中的内容要求基于该前缀生成的调度不能改变其内容，而方括号中的内容要求该周期内必须放置 T_i，但 T_i 的个数可以自由调整。PERIOD 分别为新发现的动态调度点切片 s_1 和 s_2 创建两个探索任务 job_1 和 job_2，其中 job_1 基于 s_1 覆盖的调度点使用调度前缀 $[T_0]$ 引导调度决策的生成（即生成的调度决策必须选择 T_0 先开始执行），以覆盖更多的线程交错情况，而 job_2 基于 s_2 覆盖的调度点使用调度前缀 $[T_1]$ 引导调度决策的生成。实际上，由于 s_1 和 s_2 是对称的，job_1 和 job_2 只需探索其中之一即已足够，后文将详细描述 job_1 的探索过程而省略 job_2 的探索过程。

在探索完基于 s_0 生成的两个调度决策后，对于 s_0 的所有可能的线程调度都已被探索完，PERIOD 切换至探索任务 job_1，继续基于 s_1 生成调度决策。基于 s_1 生成的周期数为 2 的调度决策的执行结果与图 5-4(a) 和图 5-4(b) 一致。随后 PERIOD 将周期数增加到 3 并继续基于 s_1 共生成了 8 个调度决策，即 $\{T_0{\times}8\}\cdot\{T_1\}\cdot\{T_0\}$，$\{T_0{\times}7\}\cdot\{T_1\}\cdot\{T_0{\times}2\}$，…，$\{T_0\}\cdot\{T_1\}\cdot\{T_0{\times}8\}$，前 4 个调度决策执行后对应的执行轨迹分别如图 5-4(c)、图 5-4(d)、图 5-4(e) 和图 5-4(f) 所示，它们都未触发内存并发漏洞，也未发现新的动态调度点切片。

在执行了基于 s_1 生成第 5 个周期数为 3 的调度决策 $\{T_0{\times}4\}\cdot\{T_1\}\cdot\{T_0{\times}5\}$ 后，PERIOD 发现了一个新的动态调度点切片 s_3 和能到达 s_3 的调度前缀 $\{T_0{\times}4\}\cdot[T_1]$，如图 5-5(a) 所示。PERIOD 将为新发现的动态调度点切片 s_3 创建一个新的探索任务 job_3，并且继续完成探索任务 job_1。PERIOD 在执行了第 8 个周期数为 3 的调度决策 $\{T_0\}\cdot\{T_1\}\cdot\{T_0{\times}8\}$ 后，触发了空指针解引用错误（如图 5-5(b)）。至此，PERIOD 成功地发现了示例程序中的并发 NPD 漏洞。

接下来，此处将跳过一些无关紧要的调度决策探索过程，直接开始介绍探索任务 job_3 的过程。PERIOD 在调度前缀 $\{T_0{\times}4\}\cdot[T_1]$ 的指导下，基于 s_3 继续生成调度决策对目标程序进行测试。当执行到调度决策 $\{T_0{\times}4\}\cdot\{T_1{\times}2\}\cdot\{T_0{\times}3\}\cdot\{T_1{\times}6\}$ 时，PERIOD 仍未发现新的动态调度点切片（如图 5-5(c)），然而当继续执行下一个调

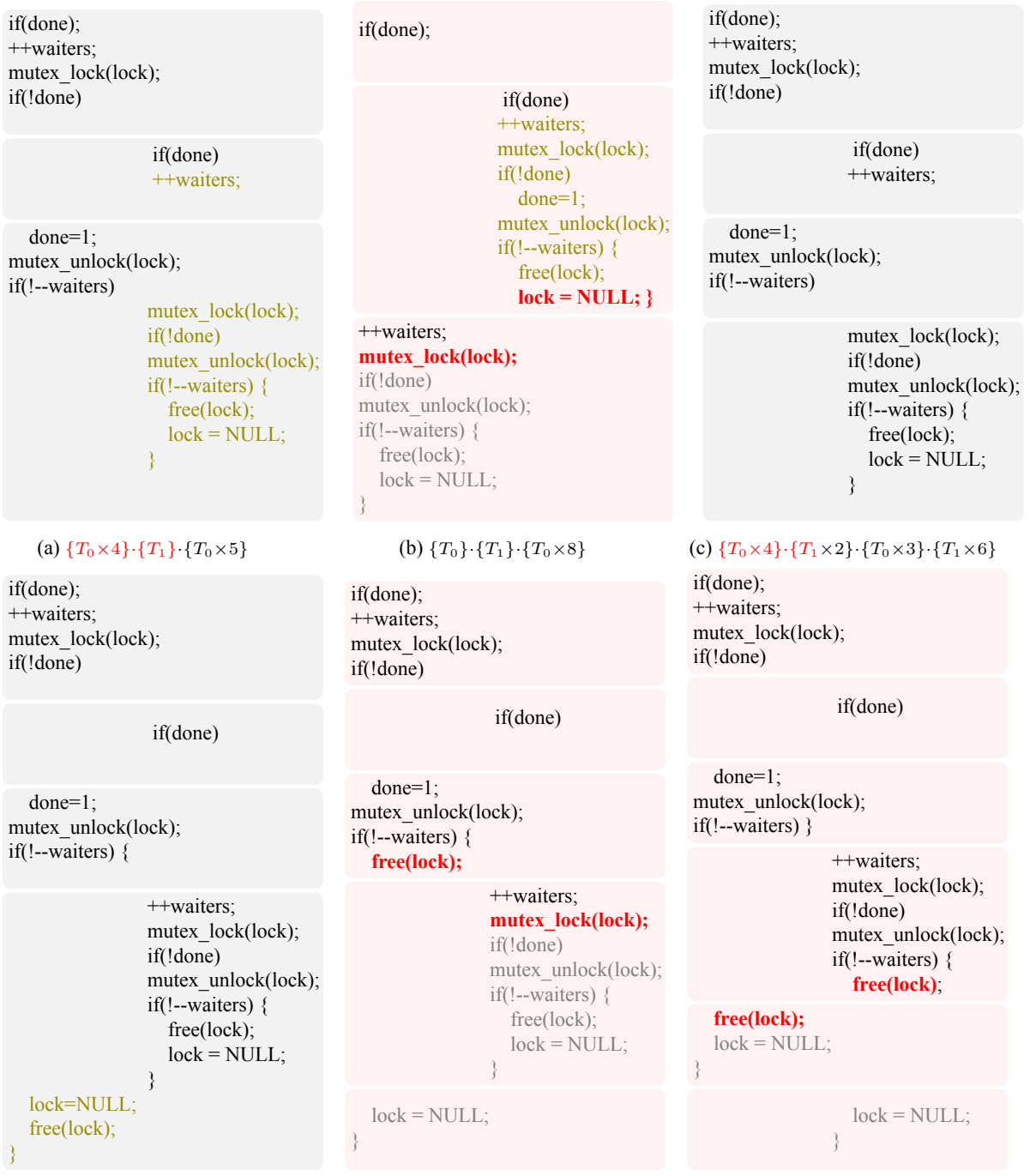


图 5-5　CVE-2016-1972 示例程序的线程交错情况。每个子图分别描述了一个调度决策对应的动态执行轨迹

度决策 $\{T_0{\times}4\}{\cdot}\{T_1\}{\cdot}\{T_0{\times}3\}{\cdot}\{T_1{\times}7\}$ 时，PERIOD 发现了新的动态调度点切片 s_4（如图5-5(d)），在该切片中两个线程同时覆盖了“lock=NULL;”和“free(lock)”语句，这也为发现新的并发漏洞带来了更大的可能性。PERIOD 分析得到 s_4 的调度前缀 $\{T_0{\times}4\}{\cdot}\{T_1\}{\cdot}[T_0]$，并创建一个新的探索任务 job_4。

当进行到探索任务 job_4，且周期数为 5 时，PERIOD 执行到调度决策 $\{T_0{\times}4\}{\cdot}\{T_1\}{\cdot}\{T_0{\times}4\}{\cdot}\{T_1{\times}7\}{\cdot}\{T_0\}$ 后将触发并发 UaF 漏洞（如图5-5(e)）。此处再次省略一

些不必要的探索过程。随后 PERIOD 继续探索周期数为 6 的调度决策，当执行到调度决策 $\{T_0 \times 4\}\cdot\{T_1\}\cdot\{T_0\times 3\}\cdot\{T_1 \times 6\}\cdot\{T_0\times 2\}\cdot\{T_1\}$ 时，触发了并发 DF 漏洞（如图5-5(f)）。当所有的动态调度切片在预设的周期数上限内的所有调度决策都探索完，PERIOD 将停止探索。对于这个例子，当将周期数的上限设置 6（或者比 6 更大的整数），PERIOD 能成功发现并发 NPD、并发 UaF 和并发 DF 漏洞。上文旨在结合示例来解释了本章方法的整体流程，下文将对其中涉及的关键技术点进行详细的说明。

5.2.2 静态分析与程序插桩

静态分析的目的主要是为了识别出能通过改变线程交错而造成并发程序的行为发生变化的程序语句，即调度点，以协助后续的受控并发测试阶段缩小线程调度作用的范围，避免不必要的探索。本文使用的静态分析不仅能够识别并发程序中使用互斥锁、条件变量、信号量等并同步原语进行线程同步的程序语句，而且能定位线程间对共享内存数据进行读写操作的程序语句。有了这些信息之后，程序插桩步骤将静态分析识别的语句当作调度点并在其之前注入基于周期性执行线程调度程序代码，以实现对调度点执行顺序的控制。

在开始介绍 PERIOD 的静态分析流程之前，本文首先引入程序依赖关系定义。并发程序的依赖关系可以分为控制依赖、一般数据依赖和干扰数据依赖[204]（Interference Dependence）。给定并发程序中的两个语句 s_1 和 s_2，若 s_2 能否被执行取决于 s_1 的执行，则称 s_2 控制依赖于 s_1；若 s_1 和 s_2 串行执行，s_2 执行时使用到了 s_1 定义的变量，则称 s_2 一般数据依赖于 s_1；若 s_1 与 s_2 可并发执行，s_2 执行时使用到了 s_1 定义的变量，则称 s_2 干扰数据依赖于 s_1。并发程序中的控制依赖和一般数据依赖的分析分别类似于串行程序中的控制依赖（详见定义 2.17）和数据依赖（详见定义 2.18）分析，干扰数据依赖描述了跨线程的可并发执行的语句之间的依赖关系，有助于识别对共享内存数据的操作。然而，干扰数据依赖的分析也是最具有挑战性的，关键的困难一方面在于如何分析两个程序语句是否可并发执行[205]，另一方面在并发程序中详尽而精确地计算指针的别名信息是极其困难的[206,207]。

定义 5.1 (干扰数据依赖). 令 s_1 和 s_2 为程序中的两条语句，语句 s_2 关于变量 v 干扰数据依赖于 s_1，当且仅当：

(1) 变量 v 满足 $v \in DEF(s_1)$，其中 $DEF(s)$ 表示语句 s 定义的变量集合。

(2) 变量 v 满足 $v \in USE(s_2)$，其中 $USE(s)$ 表示语句 s 使用的变量集合。

(3) 程序语句 s_1 和 s_2 可并发执行。

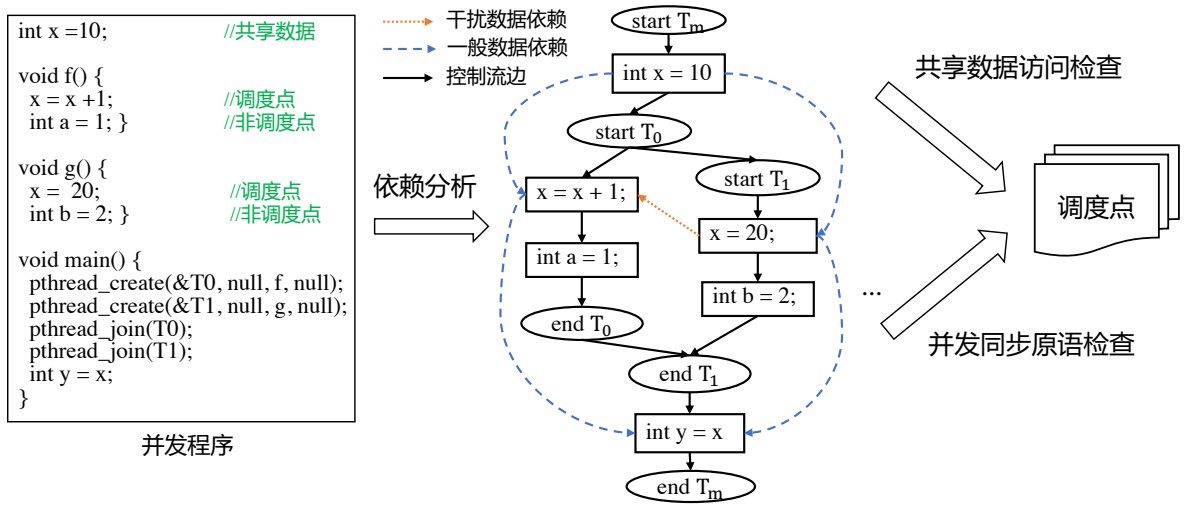


图 5-6　静态分析识别调度点的基本流程

图 5-6描述了 PERIOD 静态分析的基本流程，主要包含三个步骤：依赖分析、数据共享检查和并发同步原语检查。本文不使用详尽而精确的方式分析并发程序中的干扰依赖关系，而是采用一种代价较小而且相对保守的方式尽可能识别并发程序中的干扰数据依赖关系。存在干扰数据依赖关系的两条程序语句即为线程间对共享内存数据进行读写操作的程序语句，并发同步原语则通过检查调用 Linux/Pthread 接口的程序语句来识别。

依赖分析步骤首先根据程序代码结构来构建整个并发程序的线程间控制流程图（tCFG）。并发程序中每个线程都有一个执行流程，因此每个线程的执行过程可以用一个过程间控制流程图（CFG）来描述，若把各个线程的过程间控制流程图连接起来，则构成一个线程间控制流程图。具体来说，本文先通过安德森指针分析[184,185]来解析函数指针，分析间接调用，刻画出函数之间调用与被调用的关系以给每个线程生成一个过程间控制流程图。然后基于对线程入口函数（例如 pthread_create 函数）的建模，把每个线程的过程间控制流图根据父线程与子线程的关系连接起来，以生成 tCFG。在生成了 tCFG 之后，本文基于 tCFG 进一步构造线程程序依赖图（tPDG），tPDG 在程序依赖图（详见定义 2.19）的基础上扩展了干扰数据依赖关系，能更好地体现并发程序多线程之间的交互关系。具体来说，tPDG 显式描述了并发程序语句之间的控制依赖、一般数据依赖和干扰数据依赖关系，本文通过分析条件控制语句及其控制范围来直接获得控制依赖关系，并通过数据流方程进行定义和使用关系（def-use 关系）的分析，计算到达 tCFG 中每个结点的变量定义来挖掘一般数据依赖关系（类似于串行程序中的数据依赖分析)。对于指针变量，本文同样通过安德森指针分析深度解析别名关系，挖掘出指针变量的一般数据依赖。对于函数内的循环和函数间递归，本文通过计算不动点来最终获得

较准确的结果。

定义 5.2 (线程程序依赖图). 线程程序依赖图是一种描述并发程序中语句间依赖关系的有向图，可以用一个四元组 *tPDG* $= \langle N, E_{ndd}, E_{idd}, E_{cd} \rangle$ 表示。其中，N 表示 *tPDG* 中所有结点的集合，每个结点表示一条表达式语句。E_{cd} 表示 *tPDG* 中所有控制依赖边的集合，每一条边表示相邻两个结点的控制依赖关系。E_{ndd} 表示 PDG 中所有一般数据依赖边的集合，每一条边表示可串行执行的两个结点间的数据依赖关系。E_{idd} 表示 *tPDG* 中所有干扰数据依赖边的集合，每一条边表示属于不同线程的、可能并发执行的两个结点间的数据依赖关系。

共享数据检查步骤即分析 tPDG 中的干扰数据依赖关系，将存在干扰数据依赖关系的两条程序语句都识别为调度点。在分析完 tPDG 中的控制依赖和一般数据依赖关系后，本文继续分析干扰数据依赖关系。具体来说，本文从 tPDG 的入口结点开始遍历分析每一条程序语句。若一条程序语句创建了某个内存对象（包括指针），并且该内存对象逃逸出创建它的线程，则本文保守地认为该内存对象是共享的，并且该内存对象的所有数据域也都是共享的。要判断某内存对象是否逃逸出创建它的线程，可以通过追踪一般数据依赖关系（即 tPDG 中的 E_{ndd}）来判定该对象的生命周期是否跨越父线程传递到子线程，若是，则称该对象是线程逃逸的。对于每一个共享数据，本文进一步分析它还被哪些语句访问。给定两个不同的程序语句 s_1 和 s_2，若在 tCFG 上从 s_1 到 s_2 不可达（s_1 和 s_2 不是串行的），并且 s_1 和 s_2 分别位于不同的线程，那么本文说程序语句 s_1 和 s_2 可并发执行。尽管该判断没有精确分析两个语句之间的发生序（即 happends-before 关系[208]），是不精确的，但它也是简单而保守的，足以有效减小后续动态分析的作用范围。如果一个共享数据分别在语句 s_1 中被定义，在语句 s_2 中被使用，并且 s_1 和 s_2 可并发执行，则本文在 tPDG 上添加一条表示 s_2 干扰数据依赖于 s_1 的边 $(s1, s2) \in E_{idd}$。最

表 5-1　常用的 pthread 并发同步原语

并发同步原语类别	相关接口	并发同步原语类别	相关接口
互斥锁	pthread_mutex_lock pthread_mutex_trylock pthread_mutex_unlock	条件变量	pthread_cond_wait pthread_cond_timedwait pthread_cond_signal pthread_cond_broadcast
自旋锁	pthread_spin_lock pthread_spin_trylock pthread_spin_unlock	读写锁	pthread_rwlock_rdlock pthread_rwlock_wrlock pthread_rwlock_tryrdlock pthread_rwlock_trywrlock pthread_rwlock_unlock
信号量	sem_wait sem_trywait sem_post		

后构建了完整的 tPDG 之后，本文跟踪每一条干扰数据依赖的边，将该边所连接的两个程序语句存放于集合 *RW* 中，以便后续用于程序插桩阶段。

并发同步原语检查步骤同样遍历并发程序的每一条语句，检查其是否调用了并发同步原语的相关接口。Linux/Pthread 提供了用于线程同步的一些基本原语，包括常用的互斥锁、读写互斥锁、自旋锁、条件变量、信号量等，如表 5-1所示。也有一些其它线程库（例如 std::thread、OpenMP 等）使用不同的名称的接口，但它们的语义或线程同步功能都是类似的，维护这样一个关于并发同步原语的相关接口是很容易的。尽管设计良好的线程同步可以避免数据竞争问题，但这些原语仍然容易被误用从而引入错误，因此本文也将使用这些原语的程序语句识别为调度点。这就是说，当本文遍历到一条调用了并发同步原语相关接口的程序语句，则将该程序语句标记为线程同步操作，并存放于集合 *SYNC* 中，以便后续用于程序插桩阶段。

算法 5.2描述了 PERIOD 基于静态分析识别出的调度点进行程序插桩的详细过程。该过程以原始程序 $Prog$ 和静态分析识别出的对共享数据进行读写操作语句的集合 RW，以及使用并发同步原语的语句的集合 *SYNC* 作为输入，以插桩后的程序 $Prog'$ 作为输出。首先，基于静态分析方法为目标程序 $Prog$ 构造线程间控制流程图 *tCFG*（第 1行），算法依次遍历 *tCFG* 的每一个基本块 *bb*（第 2行），若该基本块为一个线程的入口（第 3行），则注入一段将当前线程调度方案修改为周期性执行调度方案的程序代码（第 4行）。该算法对于调度点的插桩是在指令层面进行的，算法依次遍历基本块中的每条指令（第 5行），若当前指令对应的程序语句属于共享数据访问点 *RW* 或线程同步操作 *SYNC*（第 6行），则在该指令前注入一段用于控制线程调度的程序代码（第 7行）。总的来说，静态分析识别出的调度点指明了程序插桩的位置，使得后续的受控并发测试与动态分析更加高效，减少了很多不必要的线程交错情况探索。

5.2.3 受控并发测试

给定一个包含 n 个线程的并发程序 $Prog'$ 及其输入，那么并发程序 $Prog'$ 执行过程中的不确定性主要来自线程交错的不确定性[1]，不同的线程交错可能会导致不同的执行路径，即程序的不同部分被执行/激活。为了刻画不同的被执行/激活的部分，本文引入了动态调度点切片（简称切片或 DKPS）。动态调度点切片是程序运行时被执行/激活部分的抽象，也即是程序运行时所覆盖的各个线程的调度点

[1]本文假设程序运行时有 n 个线程可被同时执行，并且线程数 n 不会在不同的执行中发生改变。此外，并发程序执行过程的行为不确定性仅来自输入和线程交错，不考虑由随机数、概率、指令重排序等因素带来的影响。

的一个动态切片，它可以表示为一个包含 n 个元素的列表，其中第 i 个元素为包含第 i 个线程 T_i 被覆盖的调度点的有序列表。动态调度点切片不是一种具体的线程交错情况，它仅反映每个单线程执行到的一条具体路径。当并发程序中的一个单线程的执行路径发生变化，那么其触发的动态调度点切片也必然发生变化。根据不同的调度决策动态执行并发程序后，其触发的动态调度点切片可能是不一样的。PERIOD 旨在发现目标程序中所有可行的动态调度点切片，并系统性地基于每一个切片继续生成调度决策以有效测试目标程序的不同线程交错情况。

算法 5.1详细描述了 PERIOD 对目标程序进行系统性的受控并发测试的过程，该算法由浅入深地、循序渐进地探索调度空间，并有针对性地测试可行的线程交错情况。算法以插桩后的程序 $Prog'$、工作线程的数量 n 和最大周期数的界限 P 作为输入，并输出测试过程中触发的内存并发漏洞及其相应的调度决策的集合 Log。PERIOD 为程序 $Prog'$ 的每一个动态调度点切片都创建一个探索任务 job，其由一个动态调度点切片和一个调度前缀组成。初始的探索任务由一个仅控制每个线程的第一个调度点的动态调度点切片 $[1] \times n$ 和一个空的调度前缀 ϵ 组成（第 2行）。PERIOD 采用了周期限定技术来系统性地探索可能的线程交错，即要求生成的调度决策的周期数不大于预设的周期数上限 P，旨在检测漏洞深度小于 P 的并发漏洞（第 3-6行）。具体来说，周期数 p 刚开始被设置为 2（第 3行），PERIOD 逐步增加周期数直到生成完预设的周期上限 P 的所有调度决策（第 4行），在此过程中，PERIOD 依次对于每一个探索任务根据指定的周期数 p 生成调度决策并动态测试（第 5行）。每个探索任务都将针对其包括的动态调度点切片以指定的调度前缀生成调度决策（第 6行，详见第 5.2.3.1节）。当该探索任务已经基于动态调度点切片生成完所有可能的调度决策，或无法根据指定的周期数 p 继续生成调度决策，本

算法 5.2: PERIOD 的插桩过程

输入： 原始程序 $Prog$、对共享数据进行读写操作语句的集合 RW 和使用并发同步原语的语句的集合 $SYNC$

输出： 一个插桩后的程序 $Prog'$

```
1 let tCFG be a thread Control flow graph of program Prog          // 构造线程间控制流程图
2 foreach basicblock bb ∈ tCFG do                                  // 遍历每一个基本块
3     if bb is the first basic block of a thread then              // 找到线程入口
4         set PERIOD's thread scheduler                            // 设置周期性执行调度方案
5     foreach instruction i ∈ bb do                                // 遍历每一条指令
6         if i ∈ RW || i ∈ SYNC then                               // 找到调度点
7             insert code "while(*)  sched_yield();" to control the thread scheduling   // 注入代码
```

算法 5.1: PERIOD 的系统性受控并发测试算法

输入：一个插桩后的程序 $Prog'$，工作线程的数量 n，以及最大周期数的界限 P
输出：一个记录触发的内存并发漏洞及其相应的调度决策的集合 Log

```
1  Log ← ∅
2  Jobs ← {([1] × n, ϵ)}
3  p = 2                                                    // 起始周期数从 2 开始
4  while p ≤ P do
5      foreach job in Jobs do
6          Schedules ← SchedGen(job.dkps, p, job.prefix)    // 调度决策生成
7          if Schedules = ∅ then
8              Jobs ← Jobs \ {job} continue
9          foreach s ∈ Schedules do
10             dkps, Errors ← Run(P', s)                    // 周期性执行并收集反馈信息
11             Log ← Log ∪ {(s, e) | e ∈ Errors}            // 记录并发漏洞和相应的调度决策
12             prefix = GetPrefix(s)
13             Jobs ← Update(Jobs, dkps, prefix)            // 反馈信息分析
14     if Jobs = ∅ then                                     // 所有可行的调度决策都探索完毕
15         break
16     p ← p + 1                                            // 将当前周期数增加 1 以后继续探索
```

文说基于相应的切片的调度决策生成已经全部完成，即该探索任务已经全部完成，PERIOD 将该探索任务移除（第 7-8行）。接下来，PERIOD 根据每一个调度决策，基于周期性执行的调度方案对目标程序进行一次动态测试，强制目标程序的线程交错遵循调度决策中预设的线程交错执行顺序（第 9-10行）。若程序执行过程中发生了崩溃，触发了内存并发漏洞，PERIOD 则将相应的调度决策和错误日志记录到集合 Log 中（第 11行）。每次程序执行的过程中，PERIOD 都会收集反馈信息，包括覆盖的动态调度点切片 $dkps$ 和调度前缀 $prefix$，并基于对反馈信息的分析更新探索任务的集合 $Jobs$（第 12-13行，详见第 5.2.3.3节）。最后，当完成所有的探索任务或周期数 p 超出了预设的最大周期数界限，算法停止（第 14-15行），PERIOD 报告发现的并发内存漏洞及触发该漏洞所需的调度决策，开发人员和测试人员可以根据报告的调度决策确定性地复现漏洞。

5.2.3.1 调度决策生成

PERIOD 使用一连串执行周期来托管并发程序的执行，仅当一个执行周期结束后，下一个执行周期才开始，其中每个周期都被分配了一定数量的调度点，因此可以通过将不同的调度点分配在不同的周期来控制它们之间的相对执行顺序。“调度决策”是用于决定哪些调度点被分配到哪个周期的一种决策方案，它描述了对多个

调度点的一种预期的执行顺序。本文将调度决策形式化地表示为$\{\ldots\}\cdot\{\ldots\}\cdot\ldots\cdot\{\ldots\}$，其中$\{\ldots\}$表示一个至少包含一个线程标识符的执行周期，周期与周期之间通过$\cdot$隔开。由于PERIOD的调度决策生成步骤是基于特定的动态调度点切片进行的，为了不失一般性，假设一个动态调度点切片包含n个线程。根据这些线程在源代码中创建的顺序，本文分别用线程标识符T_0，T_1，...，T_{n-1}表示它们。为了使得调度决策更通用、简洁，调度决策中不会明确列举具体的调度点，而是根据每个线程的调度点的数量列举线程标识符，例如，一个包含三个执行周期的调度决策$\{T_0,T_0\}\cdot\{T_1\}\cdot\{T_0\}$预期的执行顺序为先在第一个周期执行线程$T_0$的前两个调度点，然后在第二周期执行线程$T_1$的第一个调度点，最后在第三个周期中执行$T_0$的第三个调度点。若还有未在调度决策中分配的其它调度点，这些调度点将跟在它们所在线程的最后一个受控制的调度点之后执行。此外，本文使用$\{T_x\times n\}$来简化描述T_x中的n个调度点被分配在一个时期内，例如调度决策$\{T_0,T_0\}\cdot\{T_1\}\cdot\{T_0\}$也记作$\{T_0\times 2\}\cdot\{T_1\}\cdot\{T_0\}$。

为了更方便解释本文提出的调度决策生成策略，本文首先引入序列化调度决策生成策略的概念，它奠定了PERIOD所采用的调度生成策略的基础。PERIOD所用的调度生成策略又叫并发调度决策生成策略，它针对线程数量较多的并发程序，在序列化调度决策生成策略基础上进行了优化。序列化调度决策生成策略和并发调度决策生成策略具有相同的接口，可配置于算法5.1的第6行中的$SchedGen$函数。

序列化调度决策生成策略：给定一个动态调度点切片$dkps$，对于指定的周期数p，该策略根据五个预定义的规则来生成调度决策，前四个规则如下:

- 规则1：每个周期中只会被分配同一个线程的调度点。
- 规则2：两个连续周期内的调度点应属于不同线程。
- 规则3：一共包含p个周期，且不存在空的周期。
- 规则4：线程标识符T_i出现的次数应当与$dkps$中相应线程的调度点的数量相同。

为了更清楚地阐述如何根据以上规则来生成调度决策，本文引入“调度模式”的概念。一个调度模式表示为$[T_{id_1}]\cdot[T_{id_2}]\ldots\cdot[T_{id_p}]$，其中$[T_{id_i}]$表示只有线程标识符$T_{id_i}$可以被分配到其所在的周期中。给定一个动态调度点切片$dkps$，PERIOD首先按照字典序生成所有可能的调度模式，然后对于每一个调度模式，按照一定的顺序将每个周期中的线程标识符的个数扩展为满足规则3和规则4的情况；对于每一种可能的情况PERIOD都为其生成一个调度决策。通过这种方式，PERIOD可以

有序地、系统性地为 $dkps$ 生成 p 个周期情况下的所有调度决策。假设当前探索的周期数为 $p=4$，一个给定的一个动态调度点切片 $dkps$ 有 3 个线程 T_0、T_1 和 T_2。线程 T_0 包含 3 个调度点，线程 T_1 包含 2 个调度点，而线程 T_2 包含 1 个调度点。首先，本文按照字典序为该探索任务生成所有可能的调度模式。考虑第一种可能的调度模式，本文先将线程 T_0 的调度点分配给第一个周期，接来下根据规则 1 和规则 2，T_1 和 T_2 将不能再分配到第一个周期当中，并且第二个周期中也不能分配 T_0。根据字典序，本文将 T_1 中的调度点分配到第二个周期，并再次将 T_0 中的调度点分配到第三个周期。接下来只剩下最后一个周期未分配调度点，虽然根据字典序应当将 T_1 分配到第四个周期，但如此分配会使得 T_2 中的调度点仍未被分配，这违反了规则 4。因此第四个周期必须分配给 T_2，第一种可能的调度模式为 $[T_0]\cdot[T_1]\cdot[T_0]\cdot[T_2]$。通过将某个周期分配给其它线程，并重新初始化该周期之后的周期，可以有序地生成其它调度模式。例如，由于第四个周期已经无法分配给除 T_2 以外的线程，考虑将第三个周期重新分配给 T_2，其后第四个周期可分配给 T_0 或者 T_1，那么第二种和第三种可能的调度模式分别为 $[T_0]\cdot[T_1]\cdot[T_2]\cdot[T_0]$ 和 $[T_0]\cdot[T_1]\cdot[T_2]\cdot[T_1]$。

一旦确定了调度模式，本文就可以将调度点分配到相应的周期当中。根据规则 3，本文先分别为每个周期分配一个调度点，接下来只需将剩余的调度点分配到同线程的调度点所在的周期中，它们可能有不同的方式，每一种方式分别对应一个不同的调度决策。以调度模式 $[T_0]\cdot[T_1]\cdot[T_0]\cdot[T_2]$ 为例，剩余未分配的线程 T_1 的调度点只能全部分配到第二个周期中，而未分配的线程 T_0 的调度点既可以分配到第一个周期中，也可以分配到第三个周期中，这可以生成两个不同的调度决策，依序分别为 $\{T_0\times 2\}\cdot\{T_1\times 2\}\cdot\{T_0\}\cdot\{T_2\}$ 和 $\{T_0\}\cdot\{T_1\times 2\}\cdot\{T_0\times 2\}\cdot\{T_2\}$。按照这种方式，本文可以通过探索所有可能的分配调度点的方式，为一个给定的调度模式生成调度决策。

实际上，算法 5.1的第 6行中的调度决策生成函数 $SchedGen$ 还接收 $prefix$ 作为参数，因此，序列化调度决策生成策略还要求第五个规则，如下所示：

- 规则 5：生成的调度决策的起始部分应满足调度前缀 $prefix$。

调度决策的生成过程可通过调度前缀来指导（调度前缀的推断见第 5.2.3.3章）。一个调度前缀可以表示为 $prefix=\{T_{id_1}\times n_1\}\ldots\cdot\{T_{id_i}\times n_i\}\cdot[T_{id_{i+1}}]\ldots\cdot[T_{id_j}]$。如果一个调度决策的前 i 周期与 $prefix$ 的前 i 个周期完全相同，并且前 j 个调度模式与 $prefix$ 中相应周期给出的模式也相同，那么本文说该调度决策满足前缀 $prefix$。起始的探索任务中的调度前缀设置为空，任意一个调度决策都是满足空前缀的。注意，若探索任务给定的调度前缀不为空，则 PERIOD 先将前 i 个周期和前 $j-i$

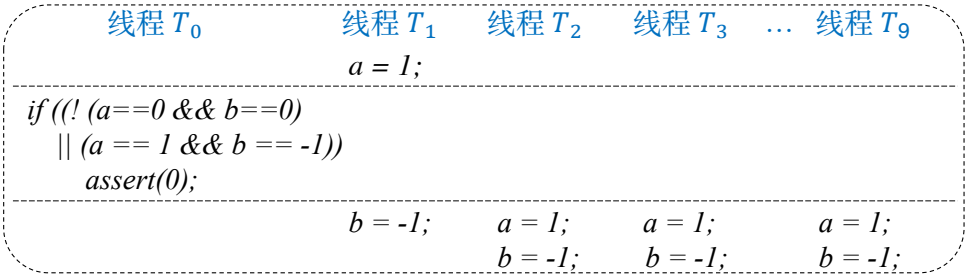


图 5-7　*reorder_10_bad* 程序包含断言失败错误的线程交错情况

个调度模式与调度前缀保持一致，然后将调度前缀中已分配的调度点从切片中去掉，再根据前四个规则扩展剩余的 $p-j$ 个周期的调度模式，并分配剩余的调度点以生成调度决策。

尽管序列化调度决策生成策略可以系统性地探索一个动态调度点切片的调度空间，但对于包含多个线程的动态调度点切片来说，生成调度决策的成本仍然很高，探索的效率很低。在最坏的情况下，要在一个包含 n 个线程的动态调度点切片 $dkps$ 中发现一个漏洞深度为 d 的并发漏洞，序列化调度决策生成策略要求使用至少含 $d+n-1$ 个周期的调度决策才来发现该漏洞。很容易估算基于该策略生成的调度决策的数量约为 $(n\times k)^{d+n-1}$，其中 k 是切片 $dkps$ 中一个线程最多包含的调度点的数量，n 是线程的数目。显然，当线程数 n 较大时，需要探索的调度空间也特别大，并且调度空间的大小随着线程数的增加呈指数级增加。图 5-7描述了一个包含 10 个线程的并发程序可以在线程 T_0 和线程 T_1 之间进行两次线程切换的情况下触发一个竞态漏洞，即由于线程交错导致程序执行 “*assert(0)*” 触发了断言违背错误。但是由于规则 1 的限制，基于序列化调度决策生成策略所生成的调度决策至少要包含 11 个周期才有可能发现该漏洞，需要探索的调度空间过于庞大，因此序列化调度决策生成策略即便在 10,000 个调度决策之内也无法触发此漏洞（详见第 5.3.3.2节和表 5-4）。

针对线程数量较多的情况，本文提出了并发调度决策生成策略，而 PERIOD 实际采用的也正是并发调度决策生成策略，该策略在序列化调度决策生成策略的基础上弱化了规则 1（变为规则 1′），在控制一部分线程的执行交错的情况下，允许另一部分线程不受控地并发执行，可以在线程数较多的情况下有效减小需要探索的调度空间。具体地说，PERIOD 每次仅选择少数线程，先按照序列化调度决策生成策略为被选择的线程生成调度决策，以便探索这些被选择的线程的交错情况。然后，PERIOD 忽略其它未被选择的线程，将它们统一分配到最后一个周期，让它们在最后一个周期不受控地自由执行。以图 5-7的含 10 个线程的并发程序为例，假设 T_0 和 T_1 被选择，一个仅有三个周期的调度决策 $\{T_1\}\cdot\{T_0\}\cdot\{T_1,T_2,\ldots,T_{10}\}$

即可触发其中的断言违背错误。

并发调度决策生成策略：在序列化调度决策生成策略的基础上将规则 1 弱化为规则 1′，其它规则保持不变：

• 规则 1′：每个周期中只会被分配同一个线程的调度点（最后一个周期除外）。

需要注意的是，PERIOD 不会对最后一个周期中的调度点施加任何的控制，它们将不受控的并发执行，但为了维持书写的一致性，本文描述调度决策的时候仍按照字典序排列它们。

并发调度决策生成策略会忽略对一些线程的控制，将它们的调度点统一分配到最后一个周期。虽然那些被忽略的线程中也可能存在并发漏洞，但是因为 PERIOD 系统性地（按照字典顺序）选择线程，所以这些并发漏洞会通过其它的调度决策被发现。具体来说，在周期数 p 的范围内，PERIOD 选择所有从 2 到 $min(p, n)$ 个线程的组合，且对于每个线程组合，PERIOD 使用序列化调度决策生成策略生成周期为 p 的调度决策。由于这些调度决策仅选定该组合中的线程及其调度点，故 PERIOD 在其最后一个周期中添加上被忽略的所有线程的调度点，最终形成完整的调度决策。类似于序列化调度决策生成策略，并行调度决策生成策略能够发现所有被选线程组合中的任意漏洞深度小于 p 的并发漏洞。

并发调度决策生成策略利用更少的周期创建有意义的上下文切换，可以极大程度地减小需要探索的线程交错空间。并发调度决策生成策略的主要思想是在控制一部分调度点的执行交错的情况下，允许剩余的调度点不受控地并发执行，这不仅可以提升测试执行的速度，还极大程度地减小需要探索的调度空间。对于任何一个有 n 个线程的动态调度点切片，其中 $n \geq 2$，无论 n 有多大，并发调度决策生成策略都可以为该切片生成仅有 2 个周期的调度决策，并且可以使用不超过 $d+1$ 个周期来触发一个深度为 d 的并发漏洞；而序列化调度决策生成策略生成的调度决策的周期数至少为 n，至少需要 $d+n-1$ 个周期来触发一个深度为 d 的并发漏洞。

5.2.3.2 周期性执行

周期性执行使用一连串执行周期来托管并发程序的执行，对于给定的调度决策，根据调度决策中对调度点的分配方案控制目标程序在运行时的线程交错，使其遵循调度决策中预设的调度点执行顺序。周期性执行遵守以下规则：（1）仅当一个周期结束后，被分配到下一个周期的调度点才会开始执行；（2）每个周期都一个周期时长，该时长应该足够长以执行完任何周期中托管的调度点（假设它们都是可执行的）；（3）从一个周期开始，到一个周期时长的时间结束，则完成了一

个周期。

PERIOD的周期性执行步骤是基于最早截止时间优先[209](Earliest Deadline First)算法实现的，该算法是实时操作系统中常用的一种调度算法[2]。操作系统周期性地执行任务，当存在多个待执行的任务时，由调度算法实时决定当前哪个任务占用处理器。最早截止时间优先算法允许用户来指定任务的具体需求，从而使操作系统能够做出最好的调度决策。该算法将一个实时任务抽象为三元组，即周期时长($period$)、运行时间($runtime$)和截止时间($deadline$)，其中周期时长指的是该任务被调度一次需要间隔的最短时间；运行时间指的是该任务在每个周期内的运行时间；截止时间指该任务在一个周期内，必须在一个时间期限前完成任务；且要求 $runtime \leq deadline \leq period$。PERIOD 将并发程序中每一个可并发执行的线程视作一个独立的任务，并将这些任务在同一个周期时长下同步使用最早截止时间优先算法进行实时调度，其中运行时间设置为大于并发程序的最坏执行时间，以保证即便将所有线程都分配到一个周期内也能完成任务；截止时间设置为与周期时长相等，并且大于运行时间。sched_yield() 操作是操作系统提供的线程调度操作，当对一个任务施加 sched_yield() 操作后，该任务立即结束在当前周期的执行，让出CPU 占有权，直到下一个周期开始后该任务才会再次被调度执行。PERIOD 通过在被测程序的调度点之前插入可控数量的 sched_yield() 操作来主动将调度点调整到不同的执行周期（算法 5.2的第 7行），而每个调度点之前施加的 sched_yield() 操作的次数是根据给定的调度决策决定的。

图 5-8描述了根据调度决策 $\{T_1\}\cdot\{T_0\}\cdot\{T_1, T_2, \ldots, T_{10}\}$ 进行周期性执行的过程，其中 sched_yield() 在调度点执行之前触发，以将调度点推迟到下一个周期执行。例如在调度点 0.0 执行之前施加一次 sched_yield() 操作将其推迟到第二个周期执行；在调度点 1.1 执行之前施加两次 sched_yield() 操作将其推迟到第三周期执行。本章用 $Run(Prog', s)$ 来表示根据调度决策 s 对目标程序 $Prog'$ 进行周期性执行的过程(算法 5.2的第 10行)。

周期性执行还负责收集一些动态信息，并返回二元组 $(dkps, Errors)$。具体来说，周期性执行一方面收集错误信息 $Errors$，从中可以知道是否触发了错误；另一方面还收集触发的动态调度点切片 $dkps$，并交由反馈信息分析步骤分析是否应创建新的探索任务 job。值得注意的是，基于一个动态调度点切片生成的调度决策可能触发新的动态调度点切片，这是因为不同的线程交错可能导致条件语句中的谓词产生不同的取值，执行了不同的分支。例如，虽然图 5-4(f) 和图 5-5(a) 执行的是

[2]例如，Linux 内核自 3.14 版本以后，提供了一个名为 SCHED_DEADLINE 的调度器，该调度器实现了最早截止时间优先算法。

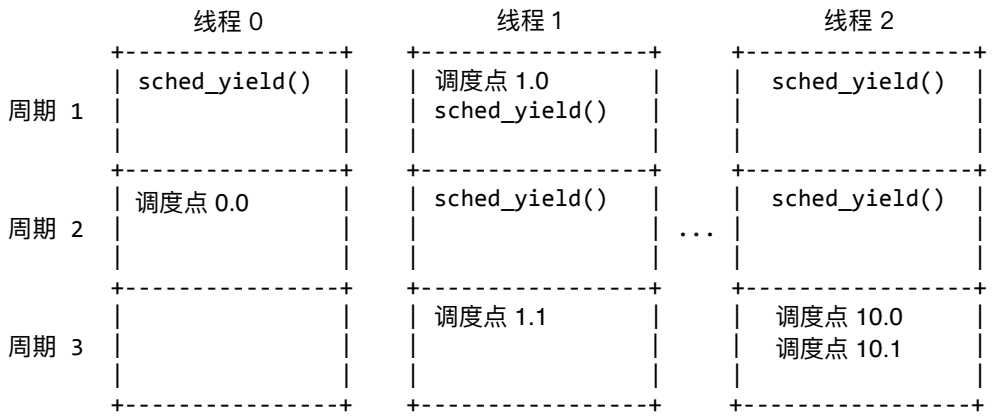


图 5-8　根据调度决策 $\{T_1\}\cdot\{T_0\}\cdot\{T_1,T_2,\ldots,T_{10}\}$ 进行周期性执行

基于同一个动态调度点切片生成的调度决策，但它们分别使得线程 T_1 在“if(done)”语句执行了不同的分支，因此触发了两个不同的动态调度点切片。

5.2.3.3　反馈信息分析

如上节所述，在一个探索任务中基于一个动态调度点切片 $dkps_1$ 生成的调度决策可能触发新的动态调度点切片 $dkps_2$，反馈信息分析处理了这种情况。为此，本文引入了两个动态调度点切片之间的“支持关系”的概念：$dkps_1$ 支持 $dkps_2$ 当且仅当 $dkps_2$ 中每个线程的调度点数量都不大于 $dkps_1$ 中的相应值。从调度决策生成的角度来看，$dkps_1$ 支持 $dkps_2$ 意味着基于 $dkps_1$ 生成的调度决策已经覆盖了基于 $dkps_2$ 生成的所有调度决策，故没有必要进行重复性的探索。

当 PERIOD 执行调度决策 s 时发现了一个新的动态调度点切片 $dkps_2$，反馈信息分析步骤检查该切片是否被当前探索任务面向的切片 $dkps_1$ 支持，若 $dkps_1$ 支持 $dkps_2$，则 PERIOD 忽略 $dkps_2$；若 $dkps_1$ 不支持 $dkps_2$，则 PERIOD 充分利用历史执行过的调度决策信息来分析和推断一个能够到达 $dkps_2$ 的调度前缀（调度决策 s 的一部分），并为动态调度点切片 $dkps_2$ 创建一个新的探索任务 job。探索任务 job 将基于动态调度点切片 $dkps_2$ 生成满足该调度前缀的调度决策。具体来说，PERIOD 将当前执行的调度决策 s 与其上一个执行的调度决策进行对比[3]，定位两个调度决策之间第一个不相同的调度点。假设第一个不相同的调度点来自线程 T_i，经分析可知是由于该调度点进行的线程切换引发了动态调度点切片的改变，因此 PERIOD 保留调度决策 s 中该调度点之前的所有内容，然后添加一个托管线程 T_i 执行的新周期模式 $[T_i]$（即该周期内必须放置 T_i，但 T_i 的个数可以自由调整）来构造调度前缀。例如，对本章的示例程序执行调度决策 $\{T_0\times 4\}\cdot\{T_1\}\cdot\{T_0\times 5\}$ 产生了一个新

[3]如果当前测试执行的调度决策为一个探索任务的第一个调度决策，那么它的上一个调度决策为一个空的调度决策 。

的动态调度点切片（图 5-5(a)），为计算新切片的调度前缀，PERIOD 将该调度决策与上一个执行的调度决策 $\{T_0 \times 5\}\cdot\{T_1\}\cdot\{T_0 \times 4\}$ 进行对比，定位出其中第一个不同的调度点为第 5 个调度点 T_1，那么保留该调度点之前的周期不变（即 $\{T_0 \times 4\}$）再添加 $[T_1]$ 即可得到调度前缀 $\{T_0 \times 4\}\cdot[T_1]$。值得注意的是，可能存在多个不同的调度决策发现了同一个动态调度点切片 $dkps_2$。当动态调度点切片 $dkps_2$ 第一次被发现时，PERIOD 分析和推断 $dkps_2$ 的调度前缀 $prefix_{ori}$，创建一个新的探索任务 job，并将 job 添加到探索任务的列表 $Jobs$ 之中；否则，若任务列表中已存在基于 $dkps_2$ 的探索任务 job 且其调度前缀为 $prefix_{ori}$，PERIOD 根据当前执行的调度决策推断出调度前缀 $prefix$，并将 job 中的调度前缀更新为 $prefix_{ori}$ 和 $prefix$ 的公共前缀。这个过程就是算法 5.2中的 $Update(Jobs, dkps, prefix)$。

从动态调度点切片的支持关系来看，一种简单但低效的探索方式是直接对一个包含了并发程序所有调度点的切片生成调度决策并进行测试，然而该调度空间可能会非常大，而且生成的调度决策可能大多数是不可行的。本文的反馈信息分析将对整个调度空间的探索任务划分为对多个动态调度点切片的探索，并利用调度前缀的指导避免了大量重复和无效的探索。例如在本章的示例程序中，任何满足先执行线程 T_0 的 5 个以上的调度点再执行线程 T_1 的第一个调度点的调度决策都将使得线程 T_1 直接执行 “return 0;” 返回，能触发这种切片的调度决策有很多，但本文的方法对于以调度前缀 $\{T_0 \times 5\}$、$\{T_0 \times 6\}$、$\{T_0 \times 7\}$、$\{T_0 \times 8\}$ 开头的调度决策只需执行一次即已足够，因为当调度决策以这些前缀开头时，无论其后缀如何变化，都不会触发新的动态调度点切片，如图 5-4(c)-5-4(f) 所示。最后，考虑到探索的效率和对并发漏洞检测的保障程度，PERIOD 设定起始周期数为 2 并逐步增加周期数直到上限 P，基于每一个动态调度点切片系统性地探索所有可能的调度决策，故探索过程是由浅入深、循序渐进的，并且能覆盖绝大多数线程切换次数在 P 次以内的线程交错。如图 5-5所示，如果设置的周期数上限 P 足够大，则可以覆盖三个条件语句的所有分支（即每个条件表达式为真或为假的情况都能执行到）；特别地，当将周期数的上限设置为 6 或更大的整数，并发 NPD、并发 UaF 和并发 DF 漏洞都能被成功发现。

5.3 实验评估

本章实现了一个融合程序分析与受控并发测试的内存并发漏洞检测工具原型 PERIOD[4]。该工具原型是基于 LLVM/Clang 框架[169]、SVF[91] 和 SCHED_DEADLINE[210]

[4]PERIOD 是一个融合程序分析与受控并发测试的内存并发漏洞检测工具，已经通过了 ICSE’22 的 artifacts evaluated available and reusable 的认证，其工具原型与相关数据集可从https://doi.org/10.5281/zenodo.5905743获取。

表 5-2　实验对象

基准数据集		程序数	描述
CVE 数据集		10	包含 10 个真实的 CVE 程序（CVE-2009-3547、CVE-2011-2183、CVE-2013-1792、CVE-2015-7550、CVE-2016-1972、CVE-2016-1973、CVE-2016-7911、CVE-2016-9806、CVE-2017-15265 和 CVE-2017-6346）
SCTBench	CS	21	包含大量小型程序和小型的测试用例
	CB	3	真实的应用程序
	Chess	4	包含 4 个 work-stealing queue 的测试用例
	Inspect	2	主要是小型程序
	Miscellaneous	1	包含一个无锁堆栈的测试用例
	Splash2	3	并行 workloads
	RADBench	5	真实的应用程序，包含 Firefox 浏览器和 JavaScript 引擎

实现的。具体来说，周期性执行和反馈信息分析的实现依赖于 LLVM/Clang 框架中对 LLVM 中间语言的插桩。用于识别调度点的静态分析部分是基于开源的过程间静态数值流（value-flow）分析工具 SVF[91] 实现的。而周期性执行的底层实现依赖于 Linux 内核中已有的 CPU 调度器 SCHED_DEADLINE。本章对实现的 PERIOD 工具原型进行了全面的实验评估。

5.3.1　实验对象

为了进行全面的实验评估，本章选取了现有研究工作中的基准数据集作为实验对象（共包含 46 个 Linux/Pthreads 平台的并发程序作为实验对象）。表 5-2列出了详细信息，其中第 1 列给出了基准数据集的名字，第 2 列表示相应的基准数据集中包含的并发程序数量，第 3 列是对程序的简单描述。

本章选取的基准数据集包括 CVE 数据集[211] 和 SCTBench[212]。CVE 数据集是由蔡彦等人[106] 基于对 CVE 漏洞库中收录的内存并发漏洞的分析提取出来的 10 个真实的 CVE 程序。每个程序中都至少包含一个并发 UaF、并发 DF、并发 NPD 漏洞。SCTBench 收录汇总了 7 个已有研究工作中的基准数据集[114,149,213–217]，本章选取了 SCTBench 中的全部 36 个可用的并发程序作为实验对象[5]。

5.3.2　实验设计

为了对本章方法进行深入研究，本章设计了三大类实验，旨在回答以下三个研究问题：

问题一：PERIOD 采用的并发调度决策生成策略相比序列化调度决策生成策略在减小调度空间的有效性上表现如何?

[5]SCTBench 实际上共包含 52 个并发程序。由于其中的 5 个程序无法在 LLVM 平台上编译，且其中有 11 个程序中的错误不受线程交错的影响，可以在任何一次动态执行中被暴露，故本章排除了 SCTBench 中的这 16 个并发程序

问题二：PERIOD 相比其它并发漏洞检测技术，在发现内存并发漏洞方面的实际效果如何?

问题三：PERIOD 相比其它受控并发测试技术在运行时开销方面的表现如何?

本章选取了几类不同的并发漏洞检测技术，包括受控并发测试、预测分析、动态数据竞争检测技术，并在这几类技术中选取公开的工具原型作为对比基准。在受控并发测试技术的类别中，本章选取了三个系统性的测试方法：IPB[148]、IDB[149]和 DFS[196]；以及三个非系统性的测试方法：PCT[112]、Maple[203] 和一个完全随机进行线程切换的公平调度测试方法 Random。为了对比传统的面向串行程序的测试技术和受控并发测试技术的实际效果，本章在实验中还包含了不控制线程调度的测试方法 Native（后文也称原生执行）。在预测分析的类别中，本章选取了两个近期的研究工作的工具原型 UFO[105] 和 ConVul[106] 作为对比基准。在动态数据竞争检测的类别中，本章选取了三个知名的数据竞争检测工具 FastTrack[102]、Helgrind[218] 和 TSAN[51]。此外，本章在实验中还对比了另一个不同版本的 PERIOD 工具，后文称这个版本的工具为 SERIAL。PERIOD 与 SERIAL 的区别在于：PERIOD 配置的是并发调度决策生成策略，而 SERIAL 配置的是序列化调度生成策略（第 5.2.3.1节）。

在实验中，PERIOD、SERIAL、IPB、IDB 和 PCT 都采用限定调度技术，它们的界限值在 CVE 数据集中设置为检测深度不超过 5 的并发漏洞（在 SCTBench 上为 3），原因是 CVE 数据集中已知的最大漏洞深度为 5（在 SCTBench 中为 3）。所有并发测试技术的调度决策总数的都设置为不超过 10,000，即对于任意一个实验对象，一旦测试的次数达到 10,000，则立即停止测试。对于实验用到的预测分析技术和数据竞争检测技术，本章采用它们的默认配置。对于每一种并发漏洞检测技术，本章对每个实验对象都会进行 10 次重复实验，并统计它们的平均结果[6]。

本章所有的实验环境为 Intel (R) Xeon (R) Silver 4212 的处理器，主频为 2.40Hz，64 位 Ubuntu LTS 18.04 操作系统，64GB 内存，安装了 GCC 7.5 和 LLVM 10.0。

5.3.3 实验结果与分析

5.3.3.1 实验一：调度决策生成策略评估

表 5-3和表 5-4分别描述了各个并发漏洞检测技术在 CVE 数据集和 SCTBench 上的统计实验结果。其中有几个关键性的评价指标，分别为“发现该漏洞用的调度决策数”、“调度决策总数”和“触发漏洞的调度决策”，这些评价指标都从某

[6]对于系统性的受控并发测试来说，它们生成的调度决策都是有序的，故 10 次重复实验的结果都是一样的。但对于随机性的并发测试技术而言，它们每次运行的结果存在一定的误差，故本章对它们进行重复实验并统计平均实验结果。对于预测分析技术和数据竞争检测技术，本章同样使用它们运行 10 次目标程序，只要其中一次能发现漏洞，则视作成功发现了并发漏洞。

表 5-3　在 CVE 数据集上的实验统计结果

程序名	漏洞类型	漏洞深度	系统性的并发测试技术												非系统性的并发测试技术								预测分析技术		数据竞争检测技术		
			PERIOD/SERIAL			IPB			IDB			DFS			Native		PCT		Random		Maple		ConVul	UFO / UFO_{NPD}	FastTrack	Helgrind	TSAN
			发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	是否能发现该漏洞？	调度决策总数					
CVE-2009-3547	NPD	1	2	6	3	3	36	5	5	33	4	4	10	1	249	3	5	3333	8	2506	✓	30	✓	✓	✓	✓	✓
CVE-2011-2183	NPD	2	3	906	130	8	98	11	6	85	9	5	31	3	681	10	5	394	8	3745	✓	60	✗	✗	✗	✗	✗
CVE-2013-1792	NPD	2	13	179	6	15	321	18	22	260	14	15	88	5	✗	0	61	124	8	741	✓	165	✓	✗	✗	✗	✗
CVE-2015-7550	NPD	2	3	14	6	8	73	11	6	64	9	5	22	3	✗	0	3	394	1	3745	✓	160	✓	✓	✗	✗	✗
CVE-2016-1972	NPD	2	3		20	16		881	11		472	228		90	✗	0	3	731	55	430	✓		✗	✗	✗	✗	✗
	UAF	4	159	573	11	91	L	918	66	6176	663	229	L	337	✗	0	134	15	1	1539	✗	144	✓	✗	✗	✗	✗
	DF	5	447		1	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✗	✗	✗	✗	✗
CVE-2016-1973	NPD	2	5	31	2	✗	L	0	✗	L	0	✗	L	0	1415	3	✗	0	✗	0	✓	157	✗	✗	✗	✗	✗
	UAF	3	17		5	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✓	✗	✗	✗	✗
CVE-2016-7911	NPD	2	3	19	8	8	204	66	6	170	54	5	58	21	799	15	5	511	5	3733	✓	143	✓	✗	✗	✗	✗
CVE-2016-9806	DF	2	6	42	4	9	226	84	7	193	65	6	71	28	✗	0	3	1135	1	2353	✓	36	✓	-	✗	✗	✗
CVE-2017-15265	UAF	2	11	96	1	✗	88	0	✗	83	0	✗	31	0	✗	0	✗	0	✗	0	✓	73	✓	✓	✗	✗	✓
CVE-2017-6346	NPD	2	5		60	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✓		✗	✗	✗	✗	✗
	UAF	3	47	182	6	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	118	✓	✗	✗	✗	✗
	DF	2	46		14	✗		0	✗		0	✗		0	✗	0	✗	0	20	1625	✓		✓	-	✗	✗	✗
发现的漏洞数(发现漏洞的程序数)			15 (10)			8 (7)			8 (7)			8 (7)			4 (3)		8 (7)		9 (7)		11 (7)		10 (9)	3 (3)	1 (1)	1 (1)	2 (2)

* 所有程序的线程数都为 3（主线程 +2 个子线程）；NPD、UAF 和 DF 分别代表空指针解引用, 释放后使用，和双重释放漏洞；‘L’ 代表着达到了实验设置的最大执行数限制（即 10,000）‘✗’ 表示未发现到相应的漏洞；‘✓’ 表示能发现相应的漏洞；”-” 表示 UFO 工具不支持检测双重释放漏洞。

个方面反映出各个技术在发现并发漏洞方面的能力。“发现该漏洞用的调度决策数” 指的是测试技术在第一次触发相应的并发漏洞时已执行的调度决策数，该数值越小，一般意味着发现漏洞的速度越快。“调度决策总数” 指的是测试技术对于目标程序探索过的调度决策数，该数值在 Native、PCT 和 Random 的评估实验中都被设置为固定 10,000 次，但在系统性的并发测试技术的评估实验中，该数值反映了生成的调度决策的数量。通常，调度决策总数越小，意味着测试阶段需要执行的调度决策数量越少，也意味着探索调度空间的效率更高。“触发漏洞的调度决策” 对于非系统性的测试技术来说意义更大。触发漏洞的调度决策数占调度决策总数的比值越大，意味着发现漏洞的概率越大。

如表 5-3所示，PERIOD 和 SERIAL 在 CVE 数据集上的实验统计结果完全相同。其原因如第 5.2节所述，对于一个包含了 n 个线程的并发程序，且该程序中有一个漏洞深度为 d 的并发漏洞，PERIOD 和 SERIAL 发现该漏洞需要探索的调度空间大

小分别近似于 $(n\times k)^{n+d-1}$ 和 $(n\times k)^{d+1}$，其中 k 表示调度点的数量。由于 CVE 数据集中的全部 10 个并发程序都仅包含 2 个子线程，因此 PERIOD 和 SERIAL 探索的调度空间大小是一致的。而 SCTBench 包含不少线程数更多的并发程序，对于这些程序，PERIOD 和 SERIAL 在各个评价指标上都表现出了差异，这些差异已在表 5-4中被标记为蓝色字体。对于表 5-4中的每一个实验对象，PERIOD 相比 SERIAL 发现漏洞用的调度决策数要更少，并且调度决策总数也更少。事实上，当被测程序包含的线程数大于 2 时，SERIAL 探索的调度空间要比 PERIOD 更大。因此，对于同一个并发漏洞，PERIOD 能在有限的调度决策次数内发现该漏洞，而 SERIAL 发现该漏洞所需的调度决策数远远多于实验中设置的调度决策总数不超过 10,000 的预算，故而 SERIAL 会产生漏报。

从发现漏洞的数量上看，PERIOD 比 SERIAL 发现的并发漏洞要多 8 个（26.67%），这得益于并发调度决策生成策略利用更少的周期创建有意义的上下文切换，极大程度地减小了需要探索的线程交错空间。例如，SERIAL 为 CS.reorder_bad 的 3 线程、4 线程和 5 线程版本的程序（即 CS.reorder_3_bad、CS.reorder_4_bad 和 CS.Reorder_5_bad）分别生成了 30、384 和 5040 个调度决策；当线程数增长到 10（即 CS.reorder_10_bad）时，SERIAL 生成的调度决策数量远远超出了 10,000 的预算限制，并且没有发现任何的并发漏洞。而 PERIOD 为 CS.reorder_bad 的 3 线程、4 线程和 5 线程版本的程序分别生成了 27、100 和 225 个调度决策，发现并发漏洞所用的调度决策数量也相较 SERIAL 要更少，并且对 10 线程版本的程序仅生成 2350 个调度决策，仍然有大量剩余的预算。值得一提的是，PERIOD 在 CS.reorder_bad 的 10 线程版本和 20 线程的版本的程序上发现并发漏洞所需的调度决策数量分别为 27 和 39 次，相比之下 SERIAL 在 10,000 个调度决策内仍无法发现任何的并发漏洞。以上实验结果表明，随着线程数的增加，并发调度决策生成策略在减小调度空间方面的改进越来越显著。此外，并发调度决策生成策略还使得 PERIOD 能够更快地发现并发漏洞。如表 5-4所示，与 SERIAL 相比，PERIOD 第一次发现并发漏洞所用的调度决策数量总是更少。

实验结论：基于对表 5-3和表 5-4的详尽分析，本章提出的并发调度决策生成策略相比序列化调度决策生成策略能更有效地减小调度空间，使得发现的内存并发漏洞的数量更多。

5.3.3.2 实验二：漏洞检测能力评估

在 CVE 数据集上的实验统计结果如表 5-3所示，PERIOD 成功地将 CVE 数据集中的全部 10 个程序都识别为有漏洞的，IPB、IDB、DFS、PCT、Random、Maple

表 5-4　在 SCTBench 上的实验统计结果

程序名	子线程数	漏洞类型	漏洞深度	系统性的并发测试技术												非系统性的并发测试技术							
				PERIOD			SERIAL			IPB			IDB			DFS			Native		PCT			Random		Maple
				发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	调度决策总数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	发现该漏洞用的调度决策数	触发漏洞的调度决策	是否能发现该漏洞？	调度决策总数
CS.account	3	AF	1	2	65	26	4	136	56	3	70	13	4	43	6	3	28	4	2903	1	5	2396	8	1177	✓	80
CS.bluetooth_driver	2	AF	2	9	205	9	9	205	9	7	92	9	7	92	9	36	177	10	✗	0	700	85	8	648	✗	57
CS.carter01	2	DL	2	5	42	11	5	42	11	11	396	38	9	250	18	8	1708	49	555	1	3	608	1	4750	✓	6
CS.circular_buffer	2	AF	2	17	871	207	17	871	207	26	806	363	42	623	219	20	3991	2043	✗	0	11	842	1	9110	✗	58
CS.deadlock01	2	DL	2	3	14	6	3	14	6	11	81	8	8	65	6	10	46	3	4353	2	38	174	1	3745	✗	89
CS.lazy01	3	AF	1	3	39	12	4	60	20	1	208	13	1	87	62	1	118	81	2	6631	1	5128	2	6092	✓	1
CS.queue	2	AF	2	25	998	119	25	998	119	101	L	7275	106	8310	3768	43	L	6405	7993	1	6	984	1	L	✓	64
CS.reorder_10	10	AF	2	27	2350	89	✗	L	0	✗	L	0	✗	7406	0	✗	L	0	✗	0	85	9	✗	0	✗	56
CS.reorder_20	20	AF	2	39	L	2870	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	891	18	✗	0	✗	56
CS.reorder_3	3	AF	2	6	27	6	10	30	10	50	1192	25	33	205	6	126	2494	23	✗	0	192	54	39	237	✗	56
CS.reorder_4	4	AF	2	9	100	12	37	384	90	393	L	31	262	518	7	6409	L	4	✗	0	164	40	68	86	✗	56
CS.reorder_5	5	AF	2	12	225	20	283	5040	816	3587	L	3	✗	996	0	✗	L	0	✗	0	355	28	68	23	✗	56
CS.stack	2	AF	2	3	918	144	3	918	144	25	2429	318	23	1595	273	22	L	512	✗	0	15	6	3	6189	✓	2
CS.token_ring	4	AF	1	2	232	58	7	2424	401	8	503	114	15	113	13	8	280	57	✗	0	11	6	9	1293	✓	45
CS.twostage_100	100	AF	2	690	L	141	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	13453	11	✗	0	✗	56
CS.twostage	2	AF	2	4	33	3	4	33	3	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✓	8
CS.wronglock_3	4	AF	2	6	172	42	26	1320	265	277	L	1227	16	1568	197	3233	L	94	7212	1	7	313	1	3197	✓	19
CS.wronglock	8	AF	2	10	1464	210	✗	L	0	✗	L	0	32	L	710	✗	L	0	✗	0	44	307	1	3286	✓	19
CB.aget-bug	3	AF	2	9	3279	679	12	L	1900	1	4359	2903	1	292	194	1	2847	1814	2	3341	7	3202	4	4853	✓	1
CB.stringbuffer	2	AF	2	12	163	16	12	163	16	13	38	2	13	38	2	8	30	2	✗	0	3555	5	23	673	✓	40
CB.pbzip2	4	BOF	2	41		14	578		27	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗	
		NPD	2	52	1626	41	82	L	427	16	L	36	4	L	1733	12	L	573	✗	0	62	136	1	2545	✓	42
		UAF	1	53		24	83		498	2		4902	3		3053	2		1267	✗	0	7	2193	5	1594	✗	
Chess.WSQ	3	AF	3	105	434	15	574	5175	117	4502	L	306	845	L	270	✗	L	0	✗	0	2	1118	392	10	✓	49
Chess.IWSQ	3	AF	3	108	711	17	836	L	138	✗	L	0	3554	L	192	✗	1503	0	✗	0	5	1173	443	24	✗	10
Chess.SWSQ	3	BOF	3	1628	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	60
		AF	3	1630		13	✗		0	284		1	222		1	✗		0	✗	0	68	257	17	1078	✗	
Chess.IWSQWS	3	BOF	3	1661	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	70
		AF	3	1663		13	✗		0	284		1	222		1	✗		0	✗	0	68	261	3	1918	✗	
Inspect.qsort_mt	3	AF	2	27	8643	135	30	L	96	33	L	365	20	3882	158	✗	L	0	✗	0	44	194	72	100	✗	115
Inspect.boundedbuffer	4	AF	2	20	L	1514	39	L	1382	✗	L	0	608	L	103	✗	L	0	15	294	27	109	8	2808	✗	158
Misc.safestack	3	AF	-	✗	6519	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	59
Splash2.barnes	2	AF	1	2	L	1186	2	L	1186	3	L	1006	3	L	741	2	L	2202	7	1251	2	5013	2	4893	✓	1
Splash2.fft	2	AF	1	2	L	3963	2	L	3963	3	L	9221	3	L	9221	2	L	7210	1	6862	2	5047	2	6241	✓	2
Splash2.lu	2	AF	1	2	6129	2848	2	6129	2848	3	L	6900	3	L	6900	2	L	5560	12	2177	2	5724	2	9714	✓	4
RADBench.bug2	2	AF	3	1985	L	9	1985	L	9	✗	L	0	✗	L	0	✗	L	0	✗	0	1813	10	✗	0	✗	264
RADBench.bug3	2	AF	2	42	L	3478	42	L	3478	✗	L	0	✗	L	0	✗	L	0	✗	0	1	489	✗	0	✗	227
RADBench.bug4	2	AF	3	259	L	6	259	L	6	✗	L	0	✗	L	0	✗	L	0	4	2013	✗	0	1275	13	✓	1
RADBench.bug5	2	DL	2	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	224
RADBench.bug6	2	DL	2	24	2950	340	24	2950	340	30	8327	112	27	4039	70	✗	L	0	236	7	484	34	3	855	✓	14
发现的漏洞数(发现漏洞的程序数)				**38 (34)**			**30 (28)**			**25 (24)**			**27 (26)**			**19 (18)**			**13 (13)**		**33 (32)**		**29 (28)**		**18 (18)**	

* AF、BOF 和 DL 分别代表断言失败错误、缓冲区溢出和死锁；‘L’ 代表着达到了实验设置的最大执行数限制（*CS.twostage_100* 的最大执行数限制为 100,000，其它程序均为 10,000）；‘✗’ 表示未发现到相应的漏洞；‘✓’ 表示能发现相应的漏洞。

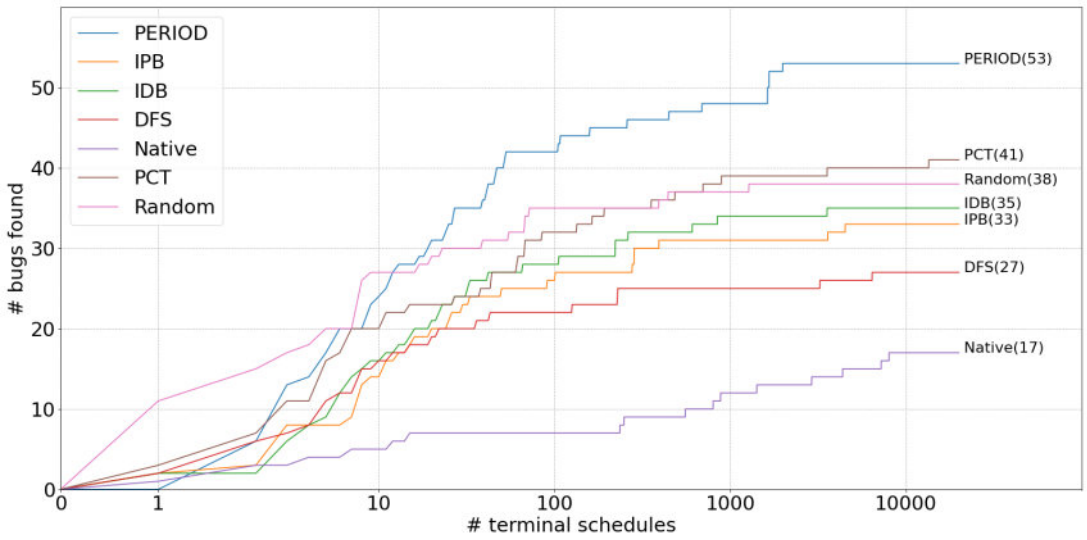


图 5-9　随着执行的调度决策数量的增加，发现漏洞的数量

等并发测试技术仅识别出 7 个程序是有漏洞的，而原生执行仅能识别出 3 个程序是有漏洞的。对于预测分析技术，ConVul 和 UFO 分别识别出 9 个和 3 个程序是包含并发漏洞的，而三种动态数据竞争检测技术最多可以识别 2 个程序包含并发漏洞。从发现的并发漏洞总数来看，PERIOD 和 SERIAL 在 CVE 数据集中总共发现了 15 个漏洞，发现漏洞的数量是最多的。PERIOD 和 SERIAL 可以在同一个程序中发现了多种类型的并发漏洞，而其它对比技术仅能发现少数类型的并发漏洞。例如，在 CVE-2017-6346 的程序中，PERIOD 发现了并发 NPD、并发 UaF 和并发 DF 三种类型的内存并发漏洞，其它系统性的并发测试技术都未发现这三种漏洞，而只有两种非系统性的并发测试技术（即 Random 和 Maple）成功发现并发 NPD 或者并发 DF 漏洞。值得注意到是，PERIOD 可以发现一些所有其它技术都无法发现的内存并发漏洞，CVE-2016-1972 程序中的并发 DF 漏洞只能通过 PERIOD 找到，这也是一个原本在 CVE 数据集中未记录的漏洞。上述实验结果证明了 PERIOD 和 SERIAL 的有效性。

在 SCTBench 上的实验统计结果如表 5-4所示，PERIOD 的性能表现最佳，总共将 30 个程序识别为有漏洞的程序，其中发现了 38 个内存并发漏洞。值得注意的是，PERIOD 在大约一半（总共 40 个并发漏洞中的 20 个）的并发漏洞上发现漏洞所用的调度决策数最少，并且在大约 83.33%（36 个中的 30 个）的程序上相比其它系统性的并发测试技术所生成的调度决策总数更少，这表明 PERIOD 对调度空间的探索效率更高。此外，SERIAL 总共发现了 30 个内存并发漏洞，性能优于其它并发测试技术（PCT 除外）。由于大部分程序的线程数较多，SERIAL 漏报了 10 个漏洞，而其中的 7 个漏洞也被 IPB 和 IDB 漏报。然而，这些在其它技术中被漏报的并发漏

洞大约有 80% 的可以通过 PERIOD 识别出来。PERIOD 和 SERIAL 同样在 SCTBench 中可以发现其它对比技术发现不了的并发漏洞，例如 CS.reorder_10、CS.reorder_20 中的竞态漏洞，以及 CB.pbzip2 中的缓冲区溢出漏洞等。其中 CB.pbzip2 中的缓冲区溢出漏洞也是文本新发现的漏洞，该漏洞未在 SCTBench 的说明文档提及，且所有其它对比技术都未发现该漏洞。此外，对于本文实验中评估的所有技术而言，SCTBench 中仍有两个并发漏洞（即 Misc.SafeStack 和 RADBench.bug5）未被成功发现。本文对这两个并发漏洞进行了人工检验和分析，但即便按照 SCTBench 的说明文档的指示仍然无法成功复现这两个并发漏洞。另一个研究工作[196] 的实验结果同样显示 Misc.SafeStack 是很难被发现的，而该研究工作评估的技术同样都未能发现该漏洞[196]。

为了进一步比较各个受控并发测试技术的优劣，图 5-9描述了各个受控并发测试技术发现的漏洞数量随着调度决策数量增长的累积趋势图，其中调度决策的限制为 10,000；每一条不同颜色的线段代表一种并发测试技术，线段末端写明了技术名称以及该技术总共发现的并发漏洞的数量；线段上的一个坐标点 (x,y) 代表该技术在 x 个调度决策的限制内能发现 y 个并发漏洞。总体来说，图 5-9中 PERIOD 的增长趋势非常显著，PERIOD 可以用同样数量的调度决策发现更多的并发漏洞，或者发现相同数量的并发漏洞所需要的调度决策数量更少。例如，PERIOD 在 100 个调度决策的限制下就能发现 40 个以上的并发漏洞，相比之下其它并发测试技术发现同样数量的并发漏洞需要测试大约 1000 到 10,000 个调度决策。

以上实验结果表明，PERIOD 比其他技术在发现内存并发漏洞方面更有效，这主要有两个重要的原因。第一，PERIOD 采用限定调度技术，并基于动态调度点切片和调度前缀来循序渐进地探索调度空间，同时避免了对大量不可行的线程交错的探索，故在探索调度空间方面是十分有效的。第二，PERIOD 采用的并发调度决策生成策略在控制一部分调度点的执行交错的情况下，允许剩余的调度点不受控地并发执行，这不仅可以提升了测试执行的速度，还极大程度地减小需要探索的调度空间。

案例分析：为了说明 PERIOD 在发现内存并发漏洞方面优越性的原因，本文以两个实验对象作为研究案例，并进行详细阐述。第一个案例来自 CVE 数据集中的 CVE-2016-1972 程序，该程序中存在三种不同类型的并发漏洞，即并发 NPD、并发 UaF 和并发 DF。本文第 5.2.1节已将该程序作为示例程序并简要描述了 PERIOD 是如何发现其中的三种内存并发漏洞的，通过循序渐进地对动态调度点切片进行探索，以及在反馈信息分析的帮助下，PERIOD 为该程序总共仅需要生成 573 个调

度决策，而且能成功发现并发NPD、并发UaF和并发DF这三种类型的并发漏洞。相比之下，IPB和DFS生成的调度决策总数都达到了10,000，并且未能发现并发DF漏洞，这体现了PERIOD在调度空间探索上的优越性。另一个案例来自SCTBench中的CS.reorder_10_bad程序，该程序中存在一个竞态漏洞，漏洞触发时体现在程序中的断言失败。如图5-7所示，尽管该程序有10个线程，但要使得其中的断言“*assert(0)*”被执行到，只需在线程T_0和线程T_1之间进行两次线程切换即可。然而除了PERIOD和PCT，其它方法都无法发现这个竞态漏洞。对于具有较多线程数的并发程序，由于不支持并发调度决策生成，现有的系统性的测试技术需要探索的调度空间过于庞大，导致10,000个调度决策的预算很快被耗尽，这与Serial无法发现该漏洞的原因类似。而对于PCT这类随机性的测试方法而言，尽管它能成功发现该漏洞，但实验结果表明它发现该漏洞到概率极低的，仅0.09%（9/10000＝0.09%）左右。PERIOD得益于采用的并发调度决策生成策略，仅用测试了27个调度决策就发现了并发漏洞，这也说明了并发调度决策生成策略在减小调度空间方面的有效性。

实验结论：基于对表5-3、表5-4和图5-9的详尽分析，PERIOD相比其它并发漏洞检测技术在发现内存并发漏洞方面的能力明显要更强。PERIOD相比IPB、IDB、DFS、PCT、Random、Maple等方法在同样时间内发现的内存并发漏洞的数量要多于29%。

5.3.3.3 实验三：运行时开销评估

受控并发测试由于需要对线程调度进行控制，执行预期的调度决策也可能影响并发程序原本并发执行的效率，故受控并发测试相比原生执行会引入一定的额外运行时开销。本节使用CVE数据集中的全部10个并发程序，以及SCTBench中在线程数方面具有多样性的程序来评估PERIOD的额外运行时开销。其中，SCTBench的程序名内的数字表示该程序的子线程数。

图5-10描述了在不同实验对象上受控并发测试相对于原生执行的平均执行速度的比值。对于所有的实验对象，PERIOD引入的运行时间的减缓程度都要比IPB、IDB和PCT要小。具体而言，PERIOD运行时引入的时间减缓大约是原生执行的2至30倍，而IPB、IDB、PCT方法引入的时间减缓大约是原生执行的10至500倍。而且，随着线程数量的增加，所有受控并发测试技术的执行速度减缓的影响也会随之减弱。PERIOD引入的运行时开销更低的原因主要在两个方面：（1）周期性执行是一种高效的、非抢占式的线程调度控制技术，它不会对被测程序引入原本不会发生的死锁或线程饥饿；（2）IPB、IDB、PCT都采用单核调度技术，破坏了并

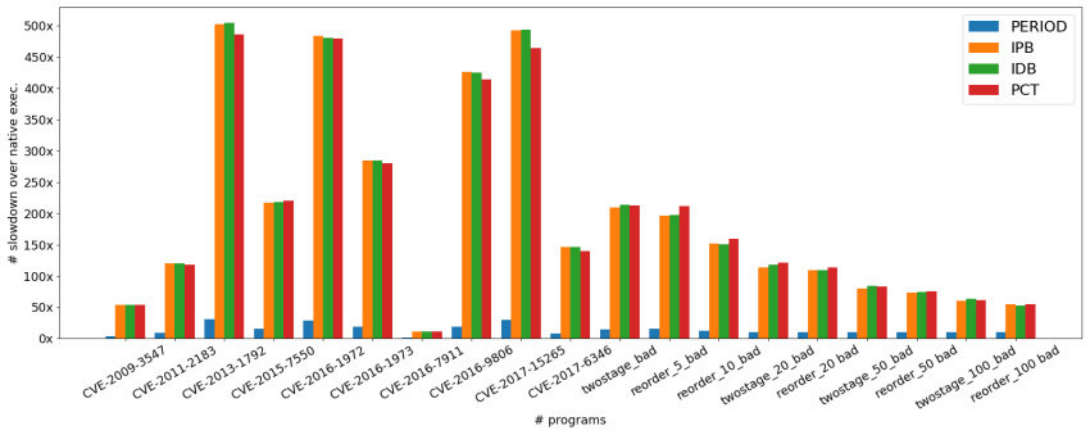


图 5-10　受控并发测试相对于原生执行的平均执行速度的比值

发程序的并行性，会影响并发程序的执行效率，而 PERIOD 的并发调度决策生成策略允许并发程序中的部分线程并发执行。故执行效率更高。

实验结论：基于对图 5-10的详尽分析，PERIOD 相比其它受控并发测试技术引入的运行时开销要低得多。

5.3.3.4　发现的新漏洞

本文还使用 PERIOD 对现实世界中的一些开源并发程序进行了分析与测试。PERIOD 成功在 *Lrzip*、*Pbzip*、*CTrace* 和 *Axel* 程序中发现了 5 个内存并发漏洞[7]，如表 5-5所示。其中 1 个漏洞是并发释放后使用漏洞，1 个漏洞是并发缓冲区溢出漏洞，2 个漏洞属于竞态漏洞。这些新发现的错误都是以前未被披露的未知漏洞，鉴于并发漏洞的稀有性和可利用性，MITRE 组织为本文发现的内存并发漏洞分配了 1 个 CVE ID。本文在 *Pbzip2 v1.0.5* 中发现的并发缓冲区溢出漏洞是在 SCTBench 中发现的未知漏洞，如第 5.3.3.2节所述，该漏洞未在 SCTBench 的说明文档提及，而且所有其它对比技术都未能发现该漏洞。本章在真实的应用程序中成功发现了新的内存并发漏洞，并且新发现的漏洞中存在其它技术都无法发现的，说明了 PERIOD 在实践中是切实有效和可行的。

5.3.4　有效性威胁分析

基于以上实验，本章从以下两个方面来分析影响本章实验有效性的因素：

内部因素：本章使用的静态分析旨在识别多线程程序中的调度点（即对共享内存的读写操作语句和并发同步原语），理想情况下该过程应该是上近似的，即所有的调度点都成功被识别出来。然而该过程仍面临一些挑战，例如别名分析、流敏感和上下文敏感的指向分析等，如果与某个内存并发漏洞相关的调度点未被正确

[7]这 5 个内存并发漏洞的触发条件与详细分析可见 https://sites.google.com/view/period-cct/detected-bugs

表 5-5　PERIOD 新发现的内存并发漏洞

漏洞 ID	程序	漏洞类型
CVE-2022-26291	*Lrzip v0.641*	（并发）释放后使用
#issue-1	*Pbzip2 v1.0.5*	（并发）缓冲区溢出
#issue-2	*CTrace v1.2*	（并发）非法内存解引用
#issue-3	*CTrace v1.2*	竞态漏洞
#issue-4	*Axel v2.17.10*	竞态漏洞

识别，可能导致受控并发测试阶段无法发现相应的内存并发漏洞。当前在 PERIOD 工具中实现的静态分析部分是在 SVF 的基础之上实现的，该部分在未来工作中可以被替换成性能更优的静态分析算法，以进一步提升静态分析阶段的准确性和效率。本章假设并发程序的输入在测试阶段是不变的，并且输入是可获取的，例如可以从已有的测试套件中选取精心设计的测试用例或通过第 4章的方法自动化生成覆盖较高的测试用例，该假设与其它的受控并发测试的研究工作一致。接下来可以尝试在测试的过程中变换并发程序的输入；实际情况下可能程序输入也不是可获取的，可以采用一些测试用例自动生成技术（例如符号执行和模糊测试）来缓解输入缺失带来的影响。此外，本章的方法是基于顺序一致性的假设提出的，对于一些弱内存模型[219] 的处理器架构（Power、ARM 和 RISC-V 等）无法保证其有效性。本文将在未来的研究工作中继续探索面向弱内存模型的内存并发漏洞检测解决方案。

外部因素：影响本章实验结果的外部因素主要是由于实验对象选取方面带来的样本偏差。尽管本章选取了 46 个不同规模及功能多样的程序来评估 PERIOD，其包含 15 个已知的并发 UAF、并发 DF、并发 NPD 漏洞，以及 40 个竞态条件漏洞；而且本章还将 PERIOD 与其它六个业界先进的受控并发测试技术、3 个数据竞争检测技术、2 个预测分析技术进行比较。然而，本章选取的实验对象和对比基准可能包括一定的样本偏差，仍不能保证本章结论对所有真实的应用程序都有效。因此，下一步将寻找更多具有不同特征且来自不同领域的真实应用程序，以提高本章结论的普适性。

5.4　本章小结

本章针对内存并发漏洞的检测问题，提出了一种融合程序分析与受控并发测试的内存并发漏洞检测方法 PERIOD。PERIOD 将并发程序的执行建模成一个多周期的执行过程，并采用了一种基于周期性执行的调度方案，能在一定程度上确定性地控制并发程序的线程交错情况。PERIOD 先通过静态分析识别目标程序中的

调度点，再采用基于动态切片和调度前缀的调度空间探索方法来指导调度决策的生成。基于对线程的控制，PERIOD 通过测试执行不同的调度决策，能有效探索调度空间并发现内存并发漏洞。本文实现了 PERIOD 工具原型，并在共包含 46 个并发程序的基准数据集上对其进行实验评估。实验结果表明，PERIOD 在检测内存并发漏洞方面要优于当前业界先进的受控并发测试技术。相比 IPB、IDB、DFS、PCT、Random、Maple 等方法，PERIOD 在同样时间内发现的内存并发漏洞的数量要多于 29%。此外，PERIOD 在真实应用程序上也发现了 5 个新的内存并发漏洞。

本章的主要研究成果撰写的论文《Controlled Concurrency Testing via Periodical Scheduling》已在软件工程领域的顶级国际会议 ICSE '22 上发表。本章实现的 PERIOD 工具原型和相关实验数据集已通过 ICSE' 22 的 "Artifacts evaluated available and reusable" 的认证。

第 6 章　总结与展望

内存安全漏洞是各类软件和应用程序所面临的最大攻击来源，给软件系统的安全性与可靠性带来了严重的威胁。程序分析与测试技术通过静态分析、动态分析、模糊测试、受控并发测试等方法，快速、有效地发现软件中存在的各类内存安全漏洞，对提升软件质量、保障软件的安全性和可靠性有着广阔的研究前景和深远意义。

6.1　本文工作总结

本文以提高内存安全漏洞检测技术的有效性和实用性作为目标，提出了一套通用可扩展的内存安全漏洞检测框架。在该框架的理论指导下，本文主要针对内存消耗漏洞、内存时序漏洞和内存安全漏洞的检测问题，提出了切实有效的解决方法。本文的主要研究工作包括：

（1）本文论述了国内外研究人员在内存安全漏洞检测领域的研究成果，归纳总结了当前静态分析、动态分析、模糊测试等程序分析与测试技术的优缺点，并指出当前的程序分析与测试技术在检测内存消耗漏洞、内存时序漏洞、内存并发漏洞方面的不足，为本文方法的提出奠定了理论基础。

（2）本文提出了一种融合程序分析与模糊测试的内存消耗漏洞检测方法 MemLock。MemLock 首次提出增加内存消耗这个新维度来指导模糊测试，使得模糊测试能逐步生成能造成过量内存消耗的测试用例，弥补了当前模糊测试技术在检测内存消耗漏洞方面的盲区。具体来说，MemLock 首先使用轻量级的静态分析确定与内存消耗相关的语句和操作，并基于这些语句和操作进行插桩以在运行时收集内存消耗信息，再通过内存消耗导向的模糊测试，结合种子动态更新策略，自动化生成能造成过量内存消耗的测试用例，以有效检测内存消耗漏洞。本文实现了 MemLock 工具原型，并使用了 14 个现实世界中被广泛使用的开源应用程序对其进行实验评估。实验结果表明，MemLock 在发现内存消耗漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术。相比 AFL、AFLfast、PerfFuzz、FairFuzz、Angora、QSYM 方法，MemLock 发现的内存消耗漏洞数量至少要多 17.9%，并且 MemLock 在发现内存消耗漏洞的速度上更快，至少是其它方法的 2.07 倍。

（3）本文提出了一种融合程序分析与模糊测试的内存时序漏洞检测方法 UAFL。UAFL 首次提出将违反内存使用的安全时序规则的操作序列用于引导模糊测试，提升了模糊测试在发现内存时序漏洞方面的有效性和效率。具体来说，UAFL 将内

存使用的安全时序规则建模成具有一定类型状态属性的状态机模型，并使用静态分析识别潜在的违反了该类型状态属性的操作序列。然后，该方法通过操作序列导向的模糊测试，结合基于信息流分析的变异优化策略，逐步生成能够覆盖完整的操作序列的测试用例，以触发内存时序漏洞。本文实现了UAFL工具原型，并使用了14个现实世界中被广泛使用的开源应用程序对其进行实验评估。实验结果表明，UAFL在发现内存时序漏洞方面要优于当前最先进的基于模糊测试的漏洞检测技术。相比AFL、AFLFast、FairFuzz、MOpt、Angora、QSYM方法，UAFL发现的内存时序漏洞数量要多于25%，并且实现了至少2.63倍的提速。

（4）本文提出了一种融合程序分析与受控并发测试的内存并发漏洞检测方法PERIOD。该方法创新性地提出了一种新型的系统性测试方法，采用一种基于周期性执行的调度方案来主动控制线程调度，能根据历史执行信息循序渐进地高效探索调度空间，很大程度提升了发现并发漏洞的效率。具体来说，PERIOD将并发程序的执行建模成一个多周期的执行过程，并采用了一种基于周期性执行的调度方案，能在一定程度上确定性地控制并发程序的线程交错情况。PERIOD先通过静态分析识别目标程序中的调度点，再采用基于动态切片和调度前缀的调度空间探索方法来指导调度决策的生成。基于对线程的控制，PERIOD通过测试执行不同的调度决策，能有效探索调度空间并发现内存并发漏洞。本文实现了PERIOD工具原型，并在包含46个并发程序的基准数据集上对其进行实验评估。实验结果表明，PERIOD在检测内存并发漏洞方面要优于当前最先进的受控并发测试技术。相比IPB、IDB、DFS、PCT、Random、Maple等方法，PERIOD在同样时间内发现的内存并发漏洞的数量要多于29%。

（5）本文在进行理论研究的基础上，实现了一套实际可用的内存安全漏洞检测工具集，该工具集包括内存消耗漏洞检测工具MemLock、内存时序漏洞检测工具UAFL和内存并发漏洞检测工具PERIOD。该工具集已在主流开源软件中发现了28个内存消耗漏洞、7个内存时序漏洞，5个内存并发漏洞。发现的漏洞也都采取了负责任的披露，其中36个漏洞被国际CVE漏洞数据库收录。

6.2 未来工作展望

软件内存安全漏洞的检测是一项十分有意义但又具有一定挑战的工作，融合程序分析与测试的方法已被事实证明是一种能有效发现内存安全漏洞的方法。在本文研究工作的基础上，仍有以下方向值得进一步探索：

（1）软件漏洞的定位与修复技术研究：本文主要针对内存安全漏洞的发现提出了切实有效的解决方案，然而问题根因的定位和漏洞的修复仍然依赖于开发人

员，不仅开发人员需要耗费大量的时间在问题根因的理解上，而且修复补丁的正确性和质量（可读性和可维护性）难以保证，例如大约39%的并发漏洞修复是不正确的[134]。因此，研究有效实用的自动化漏洞根因定位方法和漏洞修复推荐技术能够辅助开发人员更好地理解漏洞产生的根因，为修复漏洞提供指导和建议，有利于提升漏洞修复的效率和质量。

（2）结合预测分析技术进行受控并发测试：受控并发测试可以主动地探索并发程序的调度空间。然而受控并发测试对于每一个未探索的调度，都需要显式地去执行程序，这使得测试效率仍然非常低。近年来，预测分析技术在内存并发漏洞检测方面取得了很大的突破，例如UFO[105]、ConVul[106]、SeqCheck[220]等，本文在实验过程中也评估了这类工具的有效性。预测分析技术可以基于运行时记录的执行轨迹，预测其它可能的线程交错情况，故不需要显式地执行每一个线程交错。然而，预测分析的缺点是只能对已发现的执行轨迹中小范围的并发事件进行重排序，故在调度空间的探索上仍不够充分。因此，一个有前景的解决方案是结合预测分析技术进行受控并发测试，比如使用受控并发测试探索并发关键点的动态切片，再使用预测分析预测该切片对应的执行轨迹中的内存并发漏洞，有望更大幅度地减小需要探索的调度空间。

（3）弱内存模型下的内存安全漏洞检测技术研究：与当前大多数研究一样，本文所有工作都是在顺序一致性内存模型上进行的。然而现实世界中实现顺序一致性模型的代价却十分昂贵。工业架构（Power、ARM和RISC-V等）并未遵循顺序一致性语义，它们引入了存储缓冲区的层次结构，以减少内存访问的时延，也允许程序指令不按照源代码中的语句顺序执行，提高了并发程序的执行效率。这些优化在设计的时候都能够确保串行程序的行为在优化前后保持不变，但是对于并发程序可能会产生非预期的行为。例如基于x86的TSO强内存模型开发的软件，在移植到ARM平台后常出现随机崩溃、重启等后果[221]。因此，研究适用于弱内存模型下的内存安全漏洞检测技术具有非常重要的现实意义和应用价值。

（4）结合人工智能技术进行程序分析与测试：现有的程序分析与测试技术依赖于一定的启发式策略，同时面临着分析误报/漏报率较高、状态空间爆炸、可扩展性不足等一系列在实践中不可避免却难以解决的问题。随着近年来机器学习等人工智能技术得到了广泛的应用，海量的编程语言源代码、程序执行数据和软件漏洞信息被存储和管理。研究者采用统计分析和机器学习等手段提升现有的程序分析与测试技术的能力。例如，Learn&Fuzz[222]使用基于神经网络的统计机器学习方法，学习的输入概率分布来智能地指导模糊测试自动生成符合语法的测试用例。

因此，如何结合人工智能技术提升现有的程序分析与测试技术的效果和效率，是下一步值得研究的方向。

（5）研究面向分布式系统的并发漏洞检测技术：随着云计算和现代应用的发展，分布式软件逐渐比单机软件越来越受欢迎[223]。大规模分布式系统，如存储系统、云计算系统、集群管理服务等应运而生。然而这些以数据为中心的分布式系统的可靠性和安全性同样面临着并发漏洞的威胁[224]。TaxDC[225] 将单机多线程程序中的并发漏洞特征推广到了分布式系统中的分布式并发漏洞，在深入分析了104个开源分布式系统并发漏洞后，提出了一种对分布式系统中并发漏洞的分类方法。DCatch[226] 将分布式系统中的并发漏洞归结于单机节点的内存访问竞争问题，它根据一次动态执行中的节点交互事件，构建分布式系统中各个事件之间的偏序关系，并采用静态分析识别并发内存访问竞争。当前针对单机并发程序的漏洞检测已经具有一定的研究基础，如何这些研究基础合理地转化到分布式系统的并发漏洞的检测研究中，并探索新的检测技术，是未来一个重要的研究方向。

参 考 文 献

[1] V. Van der Veen, L. Cavallaro, H. Bos, et al. Memory errors: The past, the present, and the future[C]. International Workshop on Recent Advances in Intrusion Detection. Springer, 2012, 86–106.

[2] L. Szekeres, M. Payer, T. Wei, D. Song. Sok: Eternal war in memory[C]. Proceedings of the 34th IEEE Symposium on Security and Privacy. IEEE, 2013, 48–62.

[3] M. Dowson. The Ariane 5 software failure[J]. ACM SIGSOFT Software Engineering Notes. 1997. 22(2):84.

[4] 张傲翔. 北美大停电, 信息时代安全面临新挑战 [J]. 信息网络安全. 2003. (9):29–30.

[5] P. Johnston, R. Harris. The Boeing 737 MAX saga: Lessons for software organizations[J]. Software Quality Professional. 2019. 21(3):4–12.

[6] 康昊, 沈学东, 王军. 从永恒之蓝勒索病毒事件浅谈企业网络安全 [J]. 网络安全技术与应用. 2020. (10):141–142.

[7] 陈锦, 王禹. 从数据全生命周期看数据泄漏防护问题 [J]. 中国信息安全. 2018. (3):69–71.

[8] C. Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues[Z]. accessed: 2022-09-15. [online], URL https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues

[9] Google. The Chromium project finds that around 70% of our serious security bugs are memory safety problems[Z]. accessed: 2022-09-15. [online], URL https://www.chromium.org/Home/chromium-security/memory-safety

[10] 杨炀, 张焕国, 王后珍. 一种 C 程序内存访问缺陷自动化检测方法研究 [J]. 计算机科学. 2010. 37(6):155–158.

[11] MITRE. 2022 CWE Top 25 Most Dangerous Software Weaknesses[Z]. Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[12] 李舟军, 张俊贤, 廖湘科, 马金鑫. 软件安全漏洞检测技术 [J]. 计算机学报. 2015. 38(4):717–732.

[13] 张林, 曾庆凯. 软件安全漏洞的静态检测技术 [J]. 计算机工程. 2008. 34(12):157–159.

[14] 曹钦翔, 詹博华, 赵永望. 定理证明理论与应用专题前言 [J]. 软件学报. 2022. 33(6):2113–2114.

[15] 孙小祥, 陈哲. 基于定理证明的内存安全性动态检测算法的正确性研究 [J]. 计算机科学. 2021. 48(1):268–272.

[16] C. Hoare. An axiomatic approach to computer programming[J]. Comm. ACM. 1989. 12(10.1145):363235–363259.

[17] 林惠民, 张文辉. 模型检测: 理论, 方法与应用 [J]. 电子学报. 2002. 30(S1):1907.

[18] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel[C]. Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles. ACM, 2009, 207–220.

[19] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展 [J]. 软件学报. 2018. 30(1):80–109.

[20] F. E. Allen, J. Cocke. A program data flow analysis procedure[J]. Communications of the ACM. 1976. 19(3):137–147.

[21] H. Tang, X. Wang, L. Zhang, B. Xie, L. Zhang, H. Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks[C]. Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2015, 83–95.

[22] R. Baldoni, E. Coppa, D. C. D’ elia, C. Demetrescu, I. Finocchi. A survey of symbolic execution techniques[J]. ACM Computing Surveys. 2018. 51(3):1–39.

[23] 李梦君, 李舟军, 陈火旺. 基于抽象解释理论的程序验证技术 [J]. 软件学报. 2008. (1):17–26.

[24] C. Calcagno, D. Distefano. Infer: An automatic program verifier for memory safety of C programs[C]. NASA Formal Methods Symposium. Springer, 2011, 459–465.

[25] S. Blackshear, N. Gorogiannis, P. W. O’Hearn, I. Sergey. RacerD: compositional static race detection[J]. Proceedings of the ACM on Programming Languages. 2018. 2(OOPSLA):1–28.

[26] Q. Shi, X. Xiao, R. Wu, J. Zhou, C. Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code[J]. ACM SIGPLAN Notices. 2018. 53(4):693–706.

[27] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. AddressSanitizer: A fast address sanity checker[C]. Proceedings of the USENIX Conference on Usenix Annual Technical Conference. USENIX Association, 2012, 309–318.

[28] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, M. Woo. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering. 2019. 47(11):2312–2331.

[29] J. Li, B. Zhao, C. Zhang. Fuzzing: a survey[J]. Cybersecurity. 2018. 1(6):1–13.

[30] M. Zalewski. American Fuzzy Lop 2.52b[Z]. accessed: 2022-09-15. [online], URL http://lcamtuf.coredump.cx/afl

[31] E. Stepanov, K. Serebryany. MemorySanitizer: fast detector of uninitialized memory

use in C++[C]. Proceedings of the 13rd IEEE/ACM International Symposium on Code Generation and Optimization. IEEE, 2015, 46–55.

[32] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, H. Bos. VUzzer: Application-aware Evolutionary Fuzzing[C]. Proceedings of the Symposium on Network and Distributed System Security, vol. 17. IEEE, 2017, 1–14.

[33] P. Chen, H. Chen. Angora: Efficient fuzzing by principled search[C]. Proceedings of the 39th IEEE Symposium on Security and Privacy. IEEE, 2018, 711–725.

[34] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, Z. Chen. GREYONE: Data flow sensitive fuzzing[C]. Proceedings of the 29th USENIX Security Symposium. USENIX Association, 2020, 2577–2594.

[35] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing[C]. Proceedings of the 27th USENIX Security Symposium. USENIX Association, 2018, 745–761.

[36] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, L. Lu. Savior: Towards bug-driven hybrid testing[C]. Proceedings of the 41st IEEE Symposium on Security and Privacy. IEEE, 2020, 1580–1596.

[37] S. Lu, S. Park, E. Seo, Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics[C]. Proceedings of the 13rd International Conference on Architectural support for programming languages and operating systems. ACM, 2008, 329–339.

[38] 蒋炎岩, 许畅, 马晓星, 吕建. 获取访存依赖: 并发程序动态分析基础技术综述 [J]. 软件学报. 2017. 28(4):747–763.

[39] 章隆兵, 张福新, 吴少刚, 陈意云. 基于锁集合的动态数据竞争检测方法 [J]. 计算机学报. 2003. 26(10):1217–1223.

[40] 陈睿, 杨孟飞, 郭向英. 基于变量访问序模式的中断数据竞争检测方法 [J]. 软件学报. 2016. (3):547–561.

[41] 高凤娟, 王豫, 周金果, 徐安孜, 王林章, 吴荣鑫, 张川, 苏振东. 高精度的大规模程序数据竞争检测方法 [J]. 软件学报. 2021. 32(7):2039–2055.

[42] 郝宗寅, 鲁法明. Petri 网的反向展开及其在程序数据竞争检测的应用 [J]. 软件学报. 2021. 32(6):1612–1630.

[43] 赵静文, 付岩, 吴艳霞, 陈俊文, 冯云, 董继斌, 刘嘉琪. 多线程数据竞争检测技术研究综述 [J]. 计算机科学. 2022. 49(06):88–98.

[44] M. Bishop. Vulnerabilities analysis[C]. Chapter Security Policies. 1999, 125–136.

[45] S. Zhang, D. Caragea, X. Ou. An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities[C]. Proceedings of the 22nd International Conference on Database and Expert Systems Applications, vol. 6860. ACM, 2011, 217–

231.

[46] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, A. Cui. HEALER: Relation Learning Guided Kernel Fuzzing[C]. Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. ACM, 2021, 344–358.

[47] 章琳. 内存相关软件漏洞特征分析及漏洞模型构建方法研究 [D]. 学位论文, 江苏大学. 2019

[48] 张仕金, 尚赵伟. Cppcheck 的软件缺陷模式分析与定位 [J]. 计算机工程与应用. 2015. 51(003):69–73.

[49] P. O'Hearn. Separation logic[J]. Communications of the ACM. 2019. 62(2):86–95.

[50] C. Calcagno, D. Distefano, P. O'Hearn, H. Yang. Compositional shape analysis by means of bi-abduction[C]. Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2009, 289–300.

[51] K. Serebryany, T. Iskhodzhanov. ThreadSanitizer: data race detection in practice[C]. Proceedings of the workshop on binary instrumentation and applications. Association for Computing Machinery, 2009, 62–71.

[52] G. J. Duck, R. H. Yap. EffectiveSan: type and memory error detection using dynamically typed C/C++[C]. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2018, 181–195.

[53] L. d. Moura, N. Bjørner. Z3: An efficient SMT solver[C]. Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, 337–340.

[54] B. Dutertre, L. d. Moura. A fast linear-arithmetic solver for DPLL (T)[C]. Proceedings of the 18th International Conference on Computer Aided Verification. Springer, 2006, 81–94.

[55] K. Sen, D. Marinov, G. Agha. CUTE: A concolic unit testing engine for C[J]. ACM SIGSOFT Software Engineering Notes. 2005. 30(5):263–272.

[56] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, Y. Lin. Towards optimal concolic testing[C]. Proceedings of the 40th International Conference on Software Engineering. IEEE, 2018, 291–302.

[57] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing[C]. Proceedings of the Symposium on Network and Distributed System Security, vol. 8. IEEE, 2008, 151–166.

[58] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]. Proceedings of the 8th Symposium on Operating Systems Design and Implementation, vol. 8. USENIX Association, 2008, 209–224.

[59] F. Wang, Y. Shoshitaishvili. Angr-the next generation of binary analysis[C]. Proceedings of the 25th IEEE Cybersecurity Development. IEEE, 2017, 8–9.

[60] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software[C]. Proceedings of the 17th International Symposium on Software Testing and Analysis. ACM, 2008, 15–26.

[61] V. Chipounov, V. Kuznetsov, G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems[J]. Acm Sigplan Notices. 2011. 46(3):265–278.

[62] B. P. Miller, L. Fredriksen, B. So. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM. 1990. 33(12):32–44.

[63] M. Böhme, V.-T. Pham, A. Roychoudhury. Coverage-based greybox fuzzing as markov chain[J]. IEEE Transactions on Software Engineering. 2017. 45(5):489–506.

[64] C. Lemieux, K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage[C]. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018, 475–485.

[65] M. Böhme, V.-T. Pham, M.-D. Nguyen, A. Roychoudhury. Directed greybox fuzzing[C]. Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017, 2329–2344.

[66] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer[C]. Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018, 2095–2108.

[67] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, Z. Chen. Collafl: Path sensitive fuzzing[C]. Proceedings of the 39th IEEE Symposium on Security and Privacy. IEEE, 2018, 679–696.

[68] Z. Y. Ding, C. Le Goues. An Empirical Study of OSS-Fuzz Bugs[C]. Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories. IEEE, 2021, 131–142.

[69] J. Campbell, M. Walker. Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale[Z]. 2020

[70] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, C. Zhang. Beacon: Directed grey-box fuzzing with provable path pruning[C]. Proceedings of the 43rd IEEE Symposium on Security and Privacy. IEEE, 2022, 36–50.

[71] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution[C]. Proceedings of the 23rd Symposium on Network and Distributed System Security, vol. 16. IEEE, 2016, 1–16.

[72] W.-N. Chin, H. H. Nguyen, S. Qin, M. Rinard. Memory usage verification for oo programs[C]. International Static Analysis Symposium. Springer, 2005, 70–86.

[73] D.-H. Chu, J. Jaffar, R. Maghareh. Symbolic execution for memory consumption analysis[J]. ACM SIGPLAN Notices. 2016. 51(5):62–71.

[74] D. Kästner, C. Ferdinand. Proving the absence of stack overflows[C]. International Conference on Computer Safety, Reliability, and Security. Springer, 2014, 202–213.

[75] Y. Xie, A. Aiken. Context-and path-sensitive memory leak detection[C]. Proceedings of the 13rd ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2005, 115–125.

[76] S. Cherem, L. Princehouse, R. Rugina. Practical memory leak detection using guarded value-flow analysis[C]. Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2007, 480–491.

[77] Y. Jung, K. Yi. Practical memory leak detector based on parameterized procedural summaries[C]. Proceedings of the 7th international symposium on Memory Management. ACM, 2008, 131–140.

[78] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, C. Zhang. Smoke: Scalable path-sensitive memory leak detection for millions of lines of code[C]. Proceedings of the 41st IEEE/ACM International Conference on Software Engineering. IEEE, 2019, 72–82.

[79] J. Antunes, N. F. Neves, P. J. Veríssimo. Detection and prediction of resource-exhaustion vulnerabilities[C]. Proceedings of the 19th International Symposium on Software Reliability Engineering. IEEE, 2008, 87–96.

[80] 史立敏, 王晓茜, 张宏斌, 刘心宇, 汪旭童. Web 服务资源消耗脆弱性检测技术研究 [J]. 信息安全研究. 2021. 7(6):527–534.

[81] 吴民, 涂奉生. 内存泄漏的动态跟踪分析 [J]. 计算机工程与应用. 2005. 41(14):18–20.

[82] 曾佳平, 杨秋辉, 汪华龙, 徐保平, 黄蔚. 基于动态插桩的 C/C++ 内存泄漏检测工具的设计与实现 [J]. 计算机应用研究. 2015. 32(6):1737–1741.

[83] M. Elsabagh, D. Barbará, D. Fleck, A. Stavrou. On early detection of application-level resource exhaustion and starvation[J]. Journal of Systems and Software. 2018. 137:430–447.

[84] T. Petsios, J. Zhao, A. D. Keromytis, S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities[C]. Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017, 2155–2168.

[85] C. Lemieux, R. Padhye, K. Sen, D. Song. PerfFuzz: Automatically generating pathological inputs[C]. Proceedings of the 27th ACM SIGSOFT International Symposium on

Software Testing and Analysis. ACM, 2018, 254–265.

[86] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, J. Lu. ReScue: crafting regular expression DoS attacks[C]. Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018, 225–235.

[87] J. Feist, L. Mounier, M.-L. Potet. Statically detecting use after free on binary code[J]. Journal of Computer Virology and Hacking Techniques. 2014. 10(3):211–217.

[88] J. Ye, C. Zhang, X. Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities[C]. Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014, 1529–1531.

[89] H. Yan, Y. Sui, S. Chen, J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection[C]. Proceedings of the 33rd Annual Computer Security Applications Conference. ACM, 2017, 42–54.

[90] H. Yan, Y. Sui, S. Chen, J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities[C]. Proceedings of the 40th IEEE/ACM International Conference on Software Engineering. IEEE, 2018, 327–337.

[91] Y. Sui, J. Xue. SVF: Interprocedural static value-flow analysis in LLVM[C]. Proceedings of the 25th International Conference on Compiler Construction. ACM, 2016, 265–266.

[92] J. Caballero, G. Grieco, M. Marron, A. Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities[C]. Proceedings of the 21st International Symposium on Software Testing and Analysis. ACM, 2012, 133–143.

[93] N. Nethercote, J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation[J]. ACM Sigplan Notices. 2007. 42(6):89–100.

[94] T. Liu, C. Curtsinger, E. D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis[C]. Proceedings of the 38th IEEE/ACM International Conference on Software Engineering. IEEE, 2016, 911–922.

[95] B. Gui, W. Song, J. Huang. UAFSan: An object-identifier-based dynamic approach for detecting use-after-free vulnerabilities[C]. Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2021, 309–321.

[96] 吴萍, 陈意云, 张健. 多线程程序数据竞争的静态检测 [J]. 计算机研究与发展. 2006. 43(2):329–335.

[97] C. Flanagan, S. Qadeer. A type and effect system for atomicity[J]. ACM SIGPLAN Notices. 2003. 38(5):338–349.

[98] J. W. Voung, R. Jhala, S. Lerner. RELAY: Atatic race detection on millions of lines of code[C]. Proceedings of the 15th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2007,

205–214.

[99] P. Wang, J. Krinke, K. Lu, G. Li, S. Dodier-Lazaro. How Double-Fetch Situations turn into Double-Fetch Vulnerabilities: A Study of Double Fetches in the Linux Kernel[C]. Proceedings of the 26th USENIX Security Symposium. USENIX Association, 2017, 1–16.

[100] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson. Eraser: A dynamic data race detector for multithreaded programs[J]. ACM Transactions on Computer Systems. 1997. 15(4):391–411.

[101] E. Pozniansky, A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs[C]. Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2003, 179–190.

[102] C. Flanagan, S. N. Freund. FastTrack: efficient and precise dynamic race detection[C]. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2009, 121–133.

[103] K. Lu, Z. Wu, X. Wang, C. Chen, X. Zhou. RaceChecker: efficient identification of harmful data races[C]. Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2015, 78–85.

[104] J. Huang, P. O. Meredith, G. Rosu. Maximal sound predictive race detection with control flow abstraction[C]. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014, 337–348.

[105] J. Huang. UFO: predictive concurrency use-after-free detection[C]. Proceedings of the 40th IEEE/ACM International Conference on Software Engineering. IEEE, 2018, 609–619.

[106] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, B. Liang. Detecting concurrency memory corruption vulnerabilities[C]. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2019, 706–717.

[107] C. Liu, D. Zou, P. Luo, B. B. Zhu, H. Jin. A heuristic framework to detect concurrency vulnerabilities[C]. Proceedings of the 34th Annual Computer Security Applications Conference. San Juan, PR, USA: ACM, 2018, 529–541.

[108] M. Xu, S. Kashyap, H. Zhao, T. Kim. Krace: Data race fuzzing for kernel file systems[C]. Proceedings of the 41st IEEE Symposium on Security and Privacy. IEEE, 2020, 1643–1660.

[109] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, Y. Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs[C]. Proceedings of the 29th USENIX Security Symposium. USENIX Association, 2020, 2325–2342.

[110] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, S. Ur. Multithreaded Java program test generation[J]. IBM Systems Journal. 2002. 41(1):111–125.

[111] K. Sen. Race directed random testing of concurrent programs[C]. Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2008, 11–21.

[112] S. Burckhardt, P. Kothari, M. Musuvathi, S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs[J]. ACM SIGARCH Computer Architecture News. 2010. 38(1):167–178.

[113] Y. Cai, Z. Yang. Radius aware probabilistic testing of deadlocks with guarantees[C]. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2016, 356–367.

[114] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs[C]. Proceedings of the 8th Symposium on Operating Systems Design and Implementation, vol. 8. USENIX Association, 2008, 267–280.

[115] MITRE. CWE-674: Uncontrolled Recursion[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/674.html

[116] MITRE. CWE-789: Uncontrolled Memory Allocation[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/789.html

[117] MITRE. CWE-401: Missing Release of Memory after Effective Lifetime[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/401.html

[118] R. M. Farkhani, M. Ahmadi, L. Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication[C]. Proceedings of the 30th USENIX Security Symposium. USENIX Association, 2021, 1037–1054.

[119] T. Zhang, D. Lee, C. Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free[C]. Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2019, 631–644.

[120] MITRE. CWE-416: Use After Free[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/416.html

[121] MITRE. CWE-415: Double Free[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/415.html

[122] I. V. Krsul. Software vulnerability analysis[M]. Purdue University, 1998.

[123] R. A. Martin, S. Barnum. Common Weakness Enumeration (CWE) status update[J]. ACM SIGAda Ada Letters. 2008. 28(1):88–91.

[124] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs[J]. IEEE Transactions on Computers. 1979. 9:690–691.

[125] K. Nagar, S. Jagannathan. Automated detection of serializability violations under weak consistency[C]. Proceedings of the 29th International Conference on Concurrency Theory, vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 41:1–41:18.

[126] S. Owens, S. Sarkar, P. Sewell. A better x86 memory model: x86-TSO[C]. Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. Springer, 2009, 391–407.

[127] K. KIMURA, T. KODAKA, M. OBATA, H. KASAHARA. The SPARC Architecture Manual Version The SPARC Architecture Manual Version 9, 1994[J]. IEICE Transactions on Electronics. 2003. 86(4):570–579.

[128] 赵世斌, 周天阳, 朱俊虎, 王清贤. 竞态漏洞检测方法综述 [J]. 计算机工程与应用. 2018. 54(3):1–10.

[129] Y. Cai, R. Meng, J. Palsberg. Low-overhead deadlock prediction[C]. Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering. IEEE, 2020, 1298–1309.

[130] Y. Cai, C. Ye, Q. Shi, C. Zhang. Peahen: Fast and Precise Static Deadlock Detection via Context Reduction[C]. Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2022, 1–13.

[131] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson, W. Afzal. 10 Years of research on debugging concurrent and multicore software: a systematic mapping study[J]. Software quality journal. 2017. 25(1):49–82.

[132] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, H. Hansson. Concurrency bugs in open source software: a case study[J]. Journal of Internet Services and Applications. 2017. 8(1):1–15.

[133] K. Whisnant, Z. Kalbarczyk, R. K. Iyer. Micro-checkpointing: Checkpointing for multithreaded applications[C]. Proceedings 6th IEEE International On-Line Testing Workshop. IEEE, 2000, 3–8.

[134] Y. Cai, W. Chan. Lock trace reduction for multithreaded programs[J]. IEEE Transactions on Parallel and Distributed Systems. 2013. 24(12):2407–2417.

[135] J. Ferrante, K. J. Ottenstein, J. D. Warren. The program dependence graph and its use in optimization[J]. ACM Transactions on Programming Languages and Systems. 1987.

9(3):319–349.

[136] S. Horwitz, J. Prins, T. Reps. On the adequacy of program dependence graphs for representing programs[C]. Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1988, 146–157.

[137] J. Huang. Program instrumentation and software testing[J]. Computer. 1978. 11(4):25–32.

[138] 王克朝, 成坚, 王甜甜, 任向民. 面向程序分析的插桩技术研究 [J]. 计算机应用研究. 2015. 32(2):479–484.

[139] Z. Chen, Q. Zhang, J. Wu, J. Yan, J. Xue. A Source-Level Instrumentation Framework for the Dynamic Analysis of Memory Safety[J]. IEEE Transactions on Software Engineering. 2022:1–21.

[140] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation[J]. Acm Sigplan Notices. 2005. 40(6):190–200.

[141] W. Li, J. Ming, X. Luo, H. Cai. PolyCruise: A Cross-Language Dynamic Information Flow Analysis[C]. Proceedings of the 31st USENIX Security Symposium. USENIX Association, 2022, 2513–2530.

[142] 任玉柱, 张有为, 艾成炜. 污点分析技术研究综述 [J]. 计算机应用. 2019. 39(8):8.

[143] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, M. Woo. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering. 2019. 47(11):2312–2331.

[144] S. Mukherjee, P. Deligiannis, A. Biswas, A. Lal. Learning-based controlled concurrency testing[J]. Proceedings of the ACM on Programming Languages. 2020. 4:1–31.

[145] 岳翰, 吴鹏. 并发软件适应性随机测试方法 [J]. 计算机系统应用. 2015. 24(11):1–6.

[146] C. Flanagan, P. Godefroid. Dynamic partial-order reduction for model checking software[J]. ACM Sigplan Notices. 2005. 40(1):110–121.

[147] P. Thomson, A. F. Donaldson, A. Betts. Concurrency testing using schedule bounding: An empirical study[C]. Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. ACM, 2014, 15–28.

[148] M. Musuvathi, S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs[J]. ACM Sigplan Notices. 2007. 42(6):446–455.

[149] M. Emmi, S. Qadeer, Z. Rakamarić. Delay-bounded scheduling[C]. Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2011, 411–422.

[150] B. Blum, G. Gibson. Stateless model checking with data-race preemption points[C].

Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, 2016, 477–493.

[151] 杨翰文, 龙士工, 谢光颖. 基于偏序规约技术的网络程序 JPF 验证 [J]. 计算机工程与设计. 2014. 35(6):5–10.

[152] P. Abdulla, S. Aronis, B. Jonsson, K. Sagonas. Optimal dynamic partial order reduction[J]. ACM SIGPLAN Notices. 2014. 49(1):373–384.

[153] 戚晓芳, 徐晓晶, 江振亮, 汪鹏. 基于偏序约简程序可达图的并发程序切片方法 [J]. 计算机学报. 2014. 37(3):567–578.

[154] M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, K. Vaidya. Data-centric dynamic partial order reduction[J]. Proceedings of the ACM on Programming Languages. 2017. 2(POPL):1–30.

[155] 陈波, 于泠. DoS 攻击原理与对策的进一步研究 [J]. 计算机工程与应用. 2001. (10):30–33.

[156] 沈宇桔. 正则表达式复杂度攻击自动化检测技术研究 [D]. 学位论文, 南京大学. 2019

[157] MITRE. CVE-2018-17985[Z]. Available from MITRE, Common Vulnerabilities and Exposures (CVE). accessed: 2022-09-15. [online], URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17985

[158] MITRE. CVE-2018-4868[Z]. Available from MITRE, Common Vulnerabilities and Exposures (CVE). accessed: 2022-09-15. [online], URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4868

[159] MITRE. CVE-2019-7704[Z]. Available from MITRE, Common Vulnerabilities and Exposures (CVE). accessed: 2022-09-15. [online], URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7704

[160] MITRE. CVE-2017-9804[Z]. Available from MITRE, Common Vulnerabilities and Exposures (CVE). accessed: 2022-09-15. [online], URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9804

[161] Y. Machigashira, A. Nakata. An Improved LLF Scheduling for Reducing Maximum Heap Memory Consumption by Considering Laxity Time[C]. Proceedings of the 12nd International Symposium on Theoretical Aspects of Software Engineering. IEEE, 2018, 144–149.

[162] Z. Xu, C. Wen, S. Qin. State-taint analysis for detecting resource bugs[J]. Science of Computer Programming. 2018. 162:93–109.

[163] D. Wang, J. Hoffmann. Type-Guided Worst-Case Input Generation[J]. Proceedings of the ACM on Programming Languages. 2019. 3(POPL):13:1–13:30.

[164] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang. Fuzzing: State of the art[J]. IEEE Transac-

tions on Reliability. 2018. 67(3):1199–1218.

[165] D. Kroening, M. Tautschnig. CBMC –C Bounded Model Checker[C]. Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 8413. Springer, 2014, 389–391.

[166] M. Zalewski. American fuzzy lop[Z]. Technical whitepaper for afl-fuzz. accessed: 2022-09-15. [online], URL http://lcamtuf.coredump.cx/afl/technical_details.txt

[167] J. L. Andersen, M. Todberg, A. E. Dalsgaard, R. R. Hansen. Worst-case memory consumption analysis for SCJ[C]. Proceedings of the 11st International Workshop on Java Technologies for Real-time and Embedded Systems. ACM, 2013, 2–10.

[168] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, D. Feng. Towards efficient heap overflow discovery[C]. Proceedings of the 26th USENIX Security Symposium. USENIX Association, 2017, 989–1006.

[169] C. Lattner, V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation[C]. Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization. IEEE, 2004, 75–86.

[170] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks. Evaluating fuzz testing[C]. Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018, 2123–2138.

[171] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, B. Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery[C]. Proceedings of the 40th IEEE symposium on security and privacy. IEEE, 2019, 769–786.

[172] A. Vargha, H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong[J]. Journal of Educational and Behavioral Statistics. 2000. 25(2):101–132.

[173] A. Arcuri, L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering[C]. Proceedings of the 33rd International Conference on Software Engineering. IEEE, 2011, 1–10.

[174] Y. Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers[C]. Proceedings of the 22nd Annual Network and Distributed System Security Symposium. Citeseer, 2015, 1–15.

[175] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, Z. Yang. Dependence guided symbolic execution[J]. IEEE Transactions on Software Engineering. 2016. 43(3):252–271.

[176] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, W. Lee. Preventing Use-after-free with Dangling Pointers Nullification[C]. Proceedings of the 22nd Annual Network and Distributed System Security Symposium. Citeseer, 2015, 16–30.

[177] R. E. Strom, S. Yemini. Typestate: A programming language concept for enhancing

software reliability[J]. IEEE Transactions on Software Engineering. 1986. 12(1):157–171.

[178] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, W. Lee. Preventing Use-after-free with Dangling Pointers Nullification[C]. Proceedings of the Symposium on Network and Distributed System Security. IEEE, 2015, 1–15.

[179] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, M. Lemerre. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities[C]. Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses. USENIX Association, 2020, 47–62.

[180] J. Field, D. Goyal, G. Ramalingam, E. Yahav. Typestate verification: Abstraction techniques and complexity results[C]. Proceedings of the 10th International Static Analysis Symposium. Springer, 2003, 439–462.

[181] S. Fink, E. Yahav, N. Dor, G. Ramalingam, E. Geay. Effective Typestate Verification in the Presence of Aliasing[C]. Proceedings of the 15th International Symposium on Software Testing and Analysis. ACM, 2006, 133–144.

[182] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, Y. Liu. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection[C]. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2019, 533–544.

[183] W. Masri, A. Podgurski. Measuring the strength of information flows in programs[J]. ACM Transactions on Software Engineering and Methodology. 2009. 19(2):5:1–5:33.

[184] M. Shapiro, S. Horwitz. The effects of the precision of pointer analysis[C]. International Static Analysis Symposium. Springer, 1997, 16–34.

[185] N. Heintze, O. Tardieu. Demand-driven pointer analysis[J]. ACM SIGPLAN Notices. 2001. 36(5):24–34.

[186] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, R. Beyah. MOpt: Optimized Mutation Scheduling for Fuzzers[C]. Proceedings of the 28th USENIX Security Symposium. USENIX Association, 2019, 1949–1966.

[187] MITRE. CWE-1341: Multiple Releases of Same Resource or Handle[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/1314.html

[188] MITRE. CWE-910: Use of Expired File Descriptor[Z]. Available from MITRE, Common Weakness Enumeration (CWE). accessed: 2022-09-15. [online], URL https://cwe.mitre.org/data/definitions/910.html

[189] 朱承丞, 董利达. 一种多线程软件并发漏洞检测方法 [J]. 西安电子科技大学学报. 2015. 42(2):167–173.

[190] S. A. Asadollah, D. Sundmark, S. Eldh, H. Hansson. Concurrency bugs in open source software: a case study[J]. Journal of Internet Services and Applications. 2017. 8(1):1–15.

[191] S. Zhao, R. Gu, H. Qiu, T. O. Li, Y. Wang, H. Cui, J. Yang. Owl: Understanding and detecting concurrency attacks[C]. Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Luxembourg City, Luxembourg: IEEE, 2018, 219–230.

[192] G. Li, S. Lu, M. Musuvathi, S. Nath, R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing[C]. Proceedings of the 27th ACM Symposium on Operating Systems Principles. Huntsville, Ontario, Canada: ACM, 2019, 162–180.

[193] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, I. Shin. Razzer: Finding kernel race bugs through fuzzing[C]. Proceedings of the 40th IEEE Symposium on Security and Privacy. Hyatt Regency, San Francisco, CA: IEEE, 2019, 754–768.

[194] H. Feng, L. Yin, W. Lin, X. Zhao, W. Dong. Rchecker: A CBMC-based Data Race Detector for Interrupt-driven Programs[C]. Proceedings of the 20th IEEE International Conference on Software Quality, Reliability and Security Companion. Macau, China: IEEE, 2020, 465–471.

[195] P. Fonseca, R. Rodrigues, B. B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration[C]. Proceedings of the 11st Symposium on Operating Systems Design and Implementation. USENIX Association, 2014, 415–431.

[196] P. Thomson, A. F. Donaldson, A. Betts. Concurrency testing using controlled schedulers: An empirical study[J]. ACM Transactions on Parallel Computing. 2016. 2(4):1–37.

[197] M. Abdelrasoul. Promoting secondary orders of event pairs in randomized scheduling using a randomized stride[C]. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2017, 741–752.

[198] Z. Wang, D. Zhang, S. Liu, J. Sun, Y. Zhao. Adaptive randomized scheduling for concurrency bug detection[C]. Proceedings of the 24th International Conference on Engineering of Complex Computer Systems. Nansha, Guangzhou, China: IEEE, 2019, 124–133.

[199] T. Yu, T. S. Zaman, C. Wang. DESCRY: reproducing system-level concurrency failures[C]. Proceedings of the 11st Joint Meeting on Foundations of Software Engineering. Paderborn, Germany: ACM, 2017, 694–704.

[200] E. Pobee, X. Mei, W. K. Chan. Efficient transaction-based deterministic replay for multi-threaded programs[C]. Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. San Diego, California, USA: IEEE, 2019, 760–771.

[201] E. Pobee, W. K. Chan. Aggreplay: Efficient record and replay of multi-threaded programs[C]. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Paris,

France: ACM, 2019, 567–577.

[202] P. Godefroid. Model checking for programming languages using VeriSoft[C]. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 1997, 174–186.

[203] J. Yu, S. Narayanasamy, C. Pereira, G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs[C]. Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. Tucson, Arizona, USA: ACM, 2012, 485–502.

[204] Y. Cai, P. Yao, C. Zhang. Canary: Practical static detection of inter-thread value-flow bugs[C]. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. ACM, 2021, 1126–1140.

[205] S. Agarwal, R. Barik, V. Sarkar, R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs[C]. Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2007, 183–193.

[206] R. Rugina, M. Rinard. Pointer analysis for multithreaded programs[J]. ACM SIGPLAN Notices. 1999. 34(5):77–90.

[207] Y. Sui, P. Di, J. Xue. Sparse flow-sensitive pointer analysis for multithreaded programs[C]. Proceedings of the 14th International Symposium on Code Generation and Optimization. ACM, 2016, 160–170.

[208] V. Kahlon, C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs[C]. Proceedings of the 22nd Conference on Computer Aided Verification. Springer, 2010, 434–449.

[209] S. Tang, J. H. Anderson, L. Abeni. On the Defectiveness of SCHED_DEADLINE wrt Tardiness and Affinities, and a Partial Fix[C]. Proceedings of the 29th International Conference on Real-Time Networks and Systems. ACM, 2021, 46–56.

[210] A. Balsini. SCHED DEADLINE[R]. Tech. rep., Workshop on Real-Time Scheduling in the Linux Kernel. 2014

[211] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, B. Liang. CVE Benchmark[Z]. accessed: 2022-09-15. [online], URL https://github.com/mryancai/ConVul

[212] P. Thomson, A. F. Donaldson, A. Betts. SCTBench: a set of C/C++ pthread benchmarks for evaluating concurrency testing techniques[Z]. accessed: 2022-09-15. [online], URL https://github.com/mc-imperial/sctbench

[213] C. Bienia. Benchmarking modern multiprocessors[M]. Princeton University, 2011.

[214] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations[J]. ACM SIGARCH Computer Architecture News. 1995. 23(2):24–36.

[215] J. Yu, S. Narayanasamy. A case for an interleaving constrained shared-memory multiprocessor[J]. ACM SIGARCH Computer Architecture News. 2009. 37(3):325–336.

[216] L. Cordeiro, B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking[C]. Proceedings of the 33rd International Conference on Software Engineering. IEEE, 2011, 331–340.

[217] Y. Yang, X. Chen, G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs[R]. Tech. rep., University of Utah's School of Computing. 2008

[218] A. Jannesari, K. Bao, V. Pankratius, W. F. Tichy. Helgrind+: An efficient dynamic race detector[C]. Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009, 1–13.

[219] M. F. Atig, A. Bouajjani, S. Burckhardt, M. Musuvathi. On the verification problem for weak memory models[C]. Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2010, 7–18.

[220] Y. Cai, H. Yun, J. Wang, L. Qiao, J. Palsberg. Sound and efficient concurrency bug prediction[C]. Proceedings of the 39th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2021, 255–267.

[221] Z. Zhang, Z. Zhan, D. Balasubramanian, X. Koutsoukos, G. Karsai. Triggering rowhammer hardware faults on arm: A revisit[C]. Proceedings of the Workshop on Attacks and Solutions in Hardware Security. 2018, 24–33.

[222] P. Godefroid, H. Peleg, R. Singh. Learn&fuzz: Machine learning for input fuzzing[C]. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2017, 50–59.

[223] 周悦芝, 张迪. 近端云计算: 后云计算时代的机遇与挑战 [J]. 计算机学报. 2019. 42(4):677–700.

[224] J. Lu, F. Li, L. Li, X. Feng. Cloudraid: Hunting concurrency bugs in the cloud via log-mining[C]. Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018, 3–14.

[225] T. Leesatapornwongsa, J. F. Lukman, S. Lu, H. S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems[C]. Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2016, 517–530.

[226] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, C. Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems[J]. ACM SIGARCH Computer Architecture News. 2017. 45(1):677–691.

深圳大学

指导教师对研究生学位论文的学术评语

研究生姓名	文成	论文题目	融合程序分析与测试的内存安全漏洞检测技术研究		
学号	1950271008	专业（领域）名称	计算机科学与技术	学位级别	博士

对学位论文的学术评语（评价论文的学术水平和实用价值，论文有无新见解，论据是否充分、可靠，运用基础 和专业理论、知识的实际能力，所体现的科研能力，论文写作是否严谨、科学，是否有学术不端行为，论文存在的主要 缺点和问题等，是否达到学位论文的要求）：

内存安全漏洞是一种非常普遍且危害巨大的漏洞，其分析与检测问题一直是一个非常有意义且有挑战性的课题。该博士学位论文围绕内存安全漏洞检测技术开展研究，具有很好的学术研究意义和应用价值。该论文重点针对三类隐藏深、难发现、危害大的内存安全漏洞提出了一套新颖实用的分析方法与检测技术，并在真实的开源应用程序上进行了较为详细和深入的实验验证。实验结果表明，该论文所提的方法在内存安全漏洞检测方面要优于已有方法，且能够在实际应用中开源软件中发现不少内存安全漏洞。论文结构符合要求，写作思路清晰，实验严谨，结果丰富，同意送审。

指导教师对学位论文的评阅结果： 同意答辩

指导教师（签名）： 日期： 2022-10-12

深圳大学研究生学位（毕业）论文
答辩委员会决议书

研究生姓名	文成		论文题目	融合程序分析与测试的内存安全漏洞检测技术研究		
学号	1950271008		专业名称	081200 计算机科学与技术	学位级别	博士
论文答辩委员会出席名单	委员会成员	姓名	职称	工作单位	本人签名	
	主席	王轩	教授	哈尔滨工业大学（深圳）		
	委员	明仲	教授	计算机与软件学院		
		贾森	教授	深圳大学计算机与软件学院		
		李坚强	教授	深圳大学计算机与软件学院		
		田聪	教授	西安电子科技大学		
		潘微科	副教授	深圳大学计算机与软件学院		
		邹月娴	教授	北京大学深圳研究生院		
	秘书	许智武	副教授	深圳大学计算机与软件学院		
答辩日期：	2022-11-19					
答辩地点：	致腾楼938					

深圳大学研究生学位（毕业）论文

答辩委员会决议书

研究生姓名	文成	论文题目	融合程序分析与测试的内存安全漏洞检测技术研究		
学　号	1950271008	专业名称	081200 计算机科学与技术	学位级别	博士
答辩日期	2022-11-19				

答辩委员会对学位论文及答辩情况的评语及明确的结论意见（850字以内）：

论文针对三类内存安全漏洞的检测问题，开展融合程序分析与测试的内存安全漏洞检测技术研究，选题具有重要的理论意义和应用价值。

论文研究的主要内容及取得的创新性成果如下：
①提出了一种融合程序分析与模糊测试的内存消耗漏洞检测方法；
②提出了一种融合程序分析与模糊测试的内存时序漏洞检测方法；
③提出了一种融合程序分析与受控并发测试的内存并发漏洞检测方法；
④在公开数据集和开源程序上验证了所提方法的有效性。

论文写作规范、逻辑清晰、数据详实、分析深入、结论可信。答辩过程中讲述清楚，回答问题准确，表明该生已在本学科领域掌握了坚实宽广的基础理论和系统深入的专业知识，具有独立从事科学研究工作的能力，论文达到了博士学位论文水平。

答辩委员会应到7人，实到7人，经讨论和无记名投票，以7票赞成，全票通过了文成同学的博士学位论文答辩，同意毕业并建议授予工学博士学位。

答辩委员	表决项目		计票	结论
	毕业论文（准予毕业）	通过	7	通过☑
		未通过	0	未通过□
	学位论文（建议授予学位）	同意	7	通过☑
		不同意	0	未通过□
	学位论文答辩不通过，允许其修改论文后重新申请一次答辩　　票。			

秘书（签名）：　　主席（签名）：

致　　谢

“逝者如斯夫，不舍昼夜”，我要毕业了。只是，山水兼程行而至此，想起往昔悠长岁月，心底涌起无限感慨，无言、惆怅、感恩。一切都变了，好像又没有变。

人的一生是很难的，当时看来小小的种种决定造就了今日的自己。道艰且阻，成长过程有太多缺憾和不甘。博士三年半，疫情不断，天灾人祸，五味杂陈！可是，人生也是充满希望的，想要去旅行，想要多看点书，想要永远保持快乐与思考，想要生活有意义且充满诗意。看到世事万般，也一直感受着温暖与爱意。我多么幸运！

初遇深大，在一个很普通的早上，不曾想过，一年又一年，弱冠少年如今已近而立之年，在此度过了生命中最美好无瑕的日子。晨曦落日、高树百卉，微波粼粼的文山湖、神秘幽深的杜鹃山、慢慢悠悠的校巴、别具一格的南区建筑……每每念及，莫不为之心动。这里有我下课冲刺饭堂、为写论文通宵达旦的身影。见证了这里的点滴变化，我永远为母校骄傲与喝彩。

最先教会我说谢谢的便是父母，长大后我却讷于表达深切心意。哀哀父母，生我劬劳！父母一直辛苦忙碌，为我操劳担忧，竭尽所能给予我所能给予的最大的物质与关爱，包容我的任性与缺点，支持我所有的决定和选择，给了我足够的自由与空间。求学在外，陪伴的时间愈加少，爸妈总是盼望着我回家，最后却只是叮嘱我注意身体，勿多牵挂。昊天罔极之恩，我将用终生去回报。

导吾以狭路，示吾于通途，恩师是启迪智慧的明灯。秦胜潮老师精益求精，严谨慎思，温雅博学，从读博到工作的选择都为我提供了极大帮助。许智武老师兢兢业业，体贴细心，很多个周末把孩子送到游乐园自己就在旁边帮我指导论文，字斟句酌，一丝不苟。他们身上的学术精神和工作态度深深激励着我，感激之情溢于言表，望师珍重！

同学与朋友就像是上天赐予的礼物。大家从四方而来，周末相约周边的餐馆举杯畅饮，在夜深的走廊里互诉生活的烦恼；为新出的一款游戏雀跃不已，也为论文停滞而郁郁不乐。从石楠到云杉，从相识到毕业，日后一别，相见便无定期，只是回忆清晰如斯，一杯淡淡的茶便能氤氲出想念。愿岁并谢，与长友兮！

执子之手，与子偕老。感谢我的女朋友赖漫婷已陪我走过三个多春秋，从第一次牵起你的手起，就从未想过放下。虽不能朝夕相处，却日夜思念，你就像一个小太阳，始终温暖而坚定地陪伴着我。常觉幸运，你来到了我的生命中，一同经历了许许多多的事。以后，我们，还要一起走过长长的，长长的路……

吾生有崖而知无涯，求学二十余载，常怀着好奇和谦虚去探索未知世界。与其他人的比较中会信心受挫，被导师批评敲打、研究毫无头绪时也怀疑自己，一件小事没有做好也会十分自责，也偷懒、茫茫然看不清路在何处，但是，在跌跌撞撞中始终坚持着，生命的伟大不正在于不被痛苦打败吗？我明白，人应该保持热烈地求知的渴望以及积极向上的精神。

前方的路，还很长。珍惜当下，愿这一生少点遗憾。

攻读博士学位期间的研究成果

[1] **Cheng Wen**, Mengda He, Bohao Wu, Zhiwu Xu and Shengchao Qin. Controlled Concurrency Testing via Periodical Scheduling. Proceedings of the ACM/IEEE 44th International Conference on Software Engineering (ICSE). 2022, 474–486. doi:10.1145/3510003.3510178.

[2] **Cheng Wen**, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu and Ting Liu. MemLock: Memory Usage Guided Fuzzing. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE). 2020, 756–777. doi:10.1145/3377811.3380396.

[3] Haijun Wang, Xiaofei Xie, Yi Li, **Cheng Wen**, Yang Liu, Shengchao Qin, Hongxu Chen and Yulei Sui. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE). 2020, 999–1010. doi:10.1145/3377811.3380386.

[4] Zhiwu Xu, **Cheng Wen**, Shengchao Qin and Mengda He. Extracting automata from neural networks using active learning. PeerJ Computer Science. 2021, 7:e436. doi:10.7717/peerj-cs.436.

[5] Zhiwu Xu, **Cheng Wen**, and Shengchao Qin. Type Learning for Binaries and its Applications. IEEE Transactions on Reliability. 2019, 68(3): 893-912. doi:10.1109/TR.2018.2884143.