软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(7):3043−3063 [doi: 10.13328/j.cnki.jos.006862]

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

目标导向的多线程程序 UAF 漏洞预测方法*

鲁法明[1], 唐梦凡[1], 包云霞[2], 曾庆田[1], 李彦成[3]

[1](山东科技大学 计算机科学与工程学院, 山东 青岛 266590)

[2](山东科技大学 数学与系统科学学院, 山东 青岛 266590)

[3](百度在线网络技术(北京)有限公司, 北京 100193)

通信作者: 包云霞, E-mail: lufaming@sdust.edu.cn; 唐梦凡, E-mail: TangMF0220@163.com

摘 要: Use-after-free (UAF)漏洞是多线程程序的常见并发缺陷. 预测性 UAF 漏洞检测方法因兼顾误报率和漏报率而备受关注. 然而, 已有的预测性 UAF 检测方法未结合待检测目标作针对性优化, 当程序规模大或行为复杂时会导致检测效率低下. 为了解决上述问题, 提出一种目标导向的多线程程序 UAF 漏洞检测方法. 首先, 由程序运行轨迹挖掘程序的 Petri 网模型; 之后, 针对每一个潜在可构成 UAF 漏洞的内存 Free/Use 操作对, 以触发该漏洞为目标导向, 在程序的 Petri 网模型中添加保持操作间因果约束和数据一致性的行为控制结构; 在此基础上, 设计了一种基于 Petri 网反向展开的 UAF 漏洞检测方法. 该方法每次只针对 1 个潜在的 UAF 漏洞, 有针对性地验证其真实性, 从而保证检测的效率. 与此同时, 为了减少待检测的潜在 UAF 漏洞数量, 提出了一种新型向量时钟进行 Free 操作与 Use 操作间的因果关系自动识别, 据此对潜在的 UAF 漏洞进行筛选. 结合多个程序实例对所提方法进行了实验评估. 实验结果表明, 所提方法在检测的效率和准确性方面较主流方法有所提高.

关键词: 软件验证; 并发漏洞; Petri 网; UAF 漏洞; 反向展开

中图法分类号: TP311

中文引用格式: 鲁法明, 唐梦凡, 包云霞, 曾庆田, 李彦成. 目标导向的多线程程序 UAF 漏洞预测方法. 软件学报, 2023, 34(7): 3043–3063. http://www.jos.org.cn/1000-9825/6862.htm

英文引用格式: Lu FM, Tang MF, Bao YX, Zeng QT, Li YC. Target-oriented UAF Vulnerability Prediction Method of Multi-threaded Programs. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3043−3063 (in Chinese). http://www.jos.org.cn/1000-9825/6862.htm

Target-oriented UAF Vulnerability Prediction Method of Multi-threaded Programs

LU Fa-Ming[1], TANG Meng-Fan[1], BAO Yun-Xia[2], ZENG Qing-Tian[1], LI Yan-Cheng[3]

[1](College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China)

[2](College of Mathematics and Systems Science, Shandong University of Science and Technology, Qingdao 266590, China)

[3](Baidu Online Network Technology (Beijing) Co., Ltd., Beijing 100193, China)

Abstract: Use-after-free (UAF) vulnerability is a common concurrency defect in multi-threaded programs. Predictive UAF vulnerability detection methods have attracted much attention for their balance of false positives and misses. However, existing predictive UAF detection methods are not optimized for the target to be detected, which leads to unacceptable detection efficiency when the program is large or has complex behavior. To address the issue, proposes a target-oriented method to detect UAF vulnerabilities in multi-threaded programs. Firstly, the Petri net model of the program is mined from the program traces. Then, for each potential memory Free and Use operation pair that could constitute a UAF vulnerability. To add behavioural control structures that maintains causal constrains and data

* 基金项目: 国家自然科学基金(61602279); 山东省泰山学者工程专项基金(ts20190936); 山东省高等学校青创科技支持计划(2019KJN024); 山东省自然科学基金智慧计算联合基金(ZR2021LZH004); 青岛市西海岸新区 2022 年“揭榜挂帅”技术攻关项目

本文由“形式化方法与应用”专题特约编辑董云卫教授、刘关俊教授、毛晓光教授推荐.

收稿时间: 2022-09-04; 修改时间: 2022-10-08; 采用时间: 2022-12-05; jos 在线出版时间: 2022-12-30

consistency between operations to the Petri net model of the program, with the target of triggering the vulnerability. On this basis, a UAF vulnerability detection method based on Petri net reverse unfolding is designed. This method verifies the authenticity of only one potential UAF vulnerability at a time, thus ensuring the efficiency of detection. This method verifies the validity of one potential UAF vulnerability at a time, thus ensuring the efficiency of detection. At the same time, in order to reduce the number of potential UAF vulnerabilities to be detected, a new vector clock is proposed in this paper to automatically identify the causal relationship between Free and Use operations, and to filter the potential UAF vulnerabilities accordingly. The proposed method is experimentally evaluated with several program examples. The experimental results show that the proposed method improves the efficiency and accuracy of detection compared to the mainstream methods.

Key words: software verification; concurrency vulnerability; Petri net; UAF vulnerability; reverse unfolding

并行与高性能计算[1]对当代科技的发展具有支柱作用. 作为提高计算性能的重要手段, 多线程编程技术已被广泛应用于软件系统的开发. 然而, 由于线程调度顺序的不确定性, 包括死锁、数据竞争、UAF (use-after-free)漏洞在内的多种并发缺陷[2,3]随之而来. 所谓 UAF 漏洞是指程序在运行时通过悬空指针(悬空指针是指仍然指向已被释放内存空间的指针)访问已经被释放的内存[4]. 由于 C/C++语言缺乏对指针的安全性检测与内存垃圾自动回收机制, 谷歌与火狐浏览器以及 windows 操作系统等诸多基于 C/C++语言开发的软件系统易出现 UAF 漏洞,严重情况下, UAF 漏洞会导致计算机被黑客控制, 这使得 UAF 漏洞成为目前最常见的软件攻击媒介之一.

近年来, UAF 漏洞的检测方法主要可以分为静态检测[5−11]、动态检测[12−17]和预测性[18,19]方法这 3 类.

- 静态检测以软件源代码为检测依据, 无须执行程序. 例如, 经典的 GUEB 静态检测工具[7]及 Pinpoint 工具[11]结合数据流分析、指向分析和别名分析等一系列静态分析技术构建数据流图等模型, 检测模型中的 malloc-free-use 路径, 并将其作为 UAF 漏洞予以报告. 此类方法缺少程序运行时的信息, 难以对程序行为进行准确分析和推理, 通常会产生大量误报. 此外, 静态检测方法需要对程序的全部行为进行分析, 检测过程中需穷举所有可能的线程调度序列, 这导致静态检测效率的低下.
- 动态检测方法对程序执行过程中的行为和系统状态进行动态监视, 收集必要的信息来判断哪些操作构成 UAF 漏洞. 这类方法能充分利用程序的运行时信息, 与静态检测相比误报较少. 代表性的动态检测工具包括 AddressSanitizer[15]和 Kernel AddressSanitizer[16]. 它们综合运用动态插桩、污点跟踪和动态符号执行等多种分析技术, 在每次内存访问时先检测对应的影子内存是否可访问, 以此来报告 UAF 漏洞. 动态检测只能针对程序某次运行所采用的调度方案进行漏洞分析, 会导致较高的漏报率.
- 预测性方法介于静态检测和动态检测之间, 它以程序动态运行产生的操作轨迹为输入, 但可以通过操作重排由一条轨迹推导出多条潜在的程序轨迹, 进而从所有这些轨迹中检测 UAF 漏洞. 相比静态方法而言, 预测性方法可有效利用程序轨迹中蕴含的运行时信息, 从而保证较低的误报率; 与此同时, 预测性方法能够对潜在的多条运行轨迹进行分析, 相较传统的动态方法有较低的漏报率. UFO (UAF finder optimal)[19]方法是近年来提出的一种代表性的 UAF 预测方法, 它利用约束求解的方法, 在保证操作间因果约束和数据一致性不变的前提下, 从给定的程序运行轨迹中推导多条潜在的程序轨迹, 并在所有这些轨迹中进行 UAF 检测.

鉴于预测性 UAF 漏洞检测方法能兼顾误报和漏报问题, 本文围绕预测性 UAF 漏洞检测方法展开研究. 然而, 目前已知的以 UFO 方法为代表的预测性 UAF 检测方法存在如下问题: 一方面, 基于约束求解或其他方法进行操作重排时, 现有方法未结合待检测的目标 Free/Use 操作对作针对性的算法优化, 很多不可能触发 UAF 漏洞的操作重排序列和线程程序调度场景也会耗时进行推理; 另一方面, 对于某些行为复杂的多线程程序, UFO 方法产生的约束规则会指数级增长, 这给大规模程序的漏洞检测带来挑战; 最后, 针对较长的程序运行轨迹, UFO 方法会忽略轨迹中的部分事件、通过限制事件数量的方法保证检测效率, 这种做法会使程序行为信息丢失, 进而降低了检测结果的可靠性.

针对上述问题, 本文提出一种目标导向的、预测性的多线程程序 UAF 漏洞检测方法. 首先, 动态执行程序并捕获程序的运行轨迹, 由程序运行轨迹挖掘程序的 Petri 网模型, 该模型虽由单一程序轨迹挖掘得到, 但

可以刻画更多可能的程序运行轨迹; 其次, 针对每一个潜在可构成 UAF 漏洞的内存 Free/Use 操作对, 以触发该漏洞为目标导向, 构建 UAF 伴生 Petri 网(即根据待检测的目标 Free/Use 操作对, 在程序的 Petri 网模型中添加保证数据一致性和操作间因果约束关系的特殊网结构); 最后, 在 UAF 伴生 Petri 网的基础上, 设计一种基于 Petri 网反向展开[20−24]的 UAF 漏洞检测方法. 为了进一步提高检测效率, 本文所提方法每次只针对 1 个潜在的 UAF 漏洞验证其真实性. 与此同时, 为了减少待检测的 UAF 漏洞数量, 本文提出了一种新型向量时钟对 Free 操作和 Use 操作间的因果关系进行自动识别, 以此对需要验证的潜在 UAF 漏洞进行筛选.

1 实例与动机分析

本节首先结合一个多线程程序及其 UAF 漏洞的实例, 说明在 UAF 漏洞检测过程中现有方法存在的一些不足. 之后, 给出本文的研究思路和动机.

1.1 程序与UAF漏洞实例

表 1 给出了一个包含 UAF 漏洞的程序实例, 该实例包含 $thread_0$ 和 $thread_1$ 两个线程, 一个共享变量 x, 一个锁对象 l, 两个指针 p 和 q. 其中, 线程 $thread_0$ 先为指针 p 和 q 分配内存, 后启动线程 $thread_1$, 之后, 将共享变量 x 的值设为 1, 再两次申请和释放锁 l. 第 1 次获取锁 l 后会释放指针 q 引用的内存对象, 第 2 次获取锁 l 后会修改变量 x 的值为 2. 完成两次锁 l 的获取和释放后, 释放指针 p 指向的内存对象并等待线程 $thread_1$ 结束.

表 1 多线程程序 Program 1 的伪代码

$thread_0$	$thread_1$
S_1: p=(int*)$malloc$($sizeof$(int));	S_{13}: $lock$(l);
S_2: q=(int*)$malloc$($sizeof$(int));	S_{14}: *q=0;
S_3: $start$($thread_1$);	S_{15}: **if** (x==1)
S_4: x=1;	S_{16}: *p=0;
S_5: $lock$(l);	S_{17}: $unlock$(l);
S_6: $free$(q);	
S_7: $unlock$(l);	
S_8: $lock$(l);	
S_9: x=2;	
S_{10}: $unlock$(l);	
S_{11}: $free$(p);	
S_{12}: $join$($thread_1$);	

线程 $thread_1$ 被启动后先申请获取锁 l, 获取成功后, 先通过指针 q 向其引用的内存对象写入一个值, 之后, 在 x 的取值为 1 时, 通过指针 p 向其引用的内存对象写入一个值, 最后释放 l, 结束线程.

上述程序存在 3 种不同的执行场景, 其中, 前两类场景会触发 UAF 漏洞, 而最后一个不会, 具体如下.

- 第 1 种场景, $thread_0$ 先第 1 次获取锁 l, 之后释放 q 引用的内存对象, 释放锁 l, 之后 $thread_1$ 获取锁 l, 通过 q 给本已释放的内存对象赋值,由此触发 UAF 漏洞, 该漏洞是第 6 行关于 q 所引用内存对象的释放操作与第 14 行的该对象的写操作构成的.
- 第 2 种场景, $thread_0$ 先获取锁 l, 释放 q 引用的内存对象后释放锁 l. $thread_0$ 再次获取锁 l, 修改 x 的值为 2 后释放锁 l. 此时, $thread_1$ 获取锁 l, 通过 q 给已释放的内存对象赋值, 由此触发一个与第 1 种场景相同的 UAF 漏洞. 然后执行判断语句, 此时, x 的值已经被 $thread_0$ 修改为 2, 所以不满足条件不执行第 16 行的写操作, 之后释放锁 l.
- 第 3 种场景, $thread_1$ 先获取锁 l, 之后通过 q 给内存对象赋值, 然后执行判断语句, 这时, x 的值为 1, 满足条件执行第 16 行的写操作, 后释放锁 l. 此时, $thread_0$ 获取锁 l, 之后按顺序执行直到结束. 此场景下, $thread_0$ 中所有的释放操作均发生的 $thread_1$ 的写操作之后, 所以不会产生 UAF 漏洞.

1.2 动机分析

传统的静态检测工具(如 GUEB[7])基于一系列静态分析技术识别指针的传递, 构建程序的数据流图, 从中标识 malloc-free-use 路径, 并将其作为一个 UAF 漏洞. 例如, 基于上述对程序 Program 1 执行场景的分析, 根据场景一: 首先, 利用别名分析等技术跟踪指针的传递和地址的传输; 其次, 利用 UAF 漏洞的特性验证上述

结果的正确性; 最后, 可得图 1 所示的数据流切片, 该切片中可见一条 malloc-free-use 路径: S_2-S_6-S_{14}, 最终报告一个 UAF 漏洞. 但该检测结果是在场景一所规定的线程调度顺序下得出的, 而且传统的 GUEB 等静态检测工具只能在确定线程调度顺序的情况下才能构建数据流图、分析潜在漏洞. 然而, 多线程程序的调度顺序具有不确定性, 调度方案会随线程数量呈指数级增长, 因此, UAF 的静态检测方法难以保证检测效率.

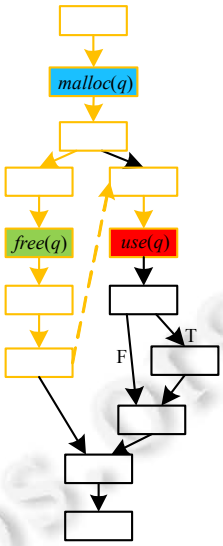


图 1 场景一下程序 Program 1 的数据流切片

以 UFO 为代表的预测性 UAF 漏洞检测方法分析程序的运行轨迹进行 UAF 检测. 表 2 给出了 Program 1 的一条运行轨迹, 其中: *start*(0,1)代表 $thread_0$ 启动 $thread_1$; *join*(0,1)代表 $thread_0$ 同步等待直到 $thread_1$ 终止; *lock*/*unlock*(0,18524392)代表 $thread_0$ 申请/释放 id 为 18524392 的同步锁; *w*/*r*(0,18524364,0)表示 $thread_0$ 向 id 为 18524364 的内存写入/读取值, 而且写或读的值为 0; *free*(0,135257110085648)代表 $thread_0$ 释放 id 为 135257110085648 的内存对象. 轨迹中各操作右上角的编号为该操作对应的程序语句编号, 例如,“*start*(0,1)[3]”表示该操作对应程序的语句 S_3:*start*($thread_1$). 下面结合该轨迹说明 UFO 基于约束求解进行 UAF 检测的原理.

表 2 Program 1 的一条运行轨迹

操作 ID	$thread_0$	$thread_1$
O_1	*start*(0,1)[3]	–
O_2	*w*(0,18524364,1)[4]	–
O_3	*lock*(0,18524392)[5]	–
O_4	*free*(0,135257110085648)[6]	–
O_5	*unlock*(0,18524392)[7]	–
O_6	–	*lock*(1,18524392)[13]
O_7	–	*w*(1,135257110085648,0)[14]
O_8	–	*r*(1,18524364,1)[15]
O_9	–	*w*(1,135257110085632,0)[16]
O_{10}	–	*unlock*(1,18524392)[17]
O_{11}	*lock*(0,18524392)[8]	–
O_{12}	*w*(0,18524364,2)[9]	–
O_{13}	*unlock*(0,18524392)[10]	–
O_{14}	*free*(0,135257110085632)[11]	–
O_{15}	*join*(0,1)[12]	–

基于约束求解的 UFO 方法首先根据轨迹中隐含的操作间因果关系定义一组约束, 包括根据线程内相邻两个操作间的因果依赖、*start* 操作与被启动线程的首个操作间的因果依赖、被汇聚线程的最后一个操作与 *join* 操作之间的因果依赖各添加一个约束规则. 如, 操作 O_1(对应 Program 1 的语句 S_3:*start*($thread_1$))和操作 O_2(对应 Program 1 的语句 S_4:x=1)同属于线程 $thread_0$, 且 O_1 的执行先于 O_2, 故有约束 O_1<O_2. 再如, “启动线程

$thread_1$的操作 O_1”与“$thread_1$的首个操作 O_6(Program 1 的语句 S_{13}:$lock(l)$)”之间也有因果依赖, 故有约束 $O_1<O_6$. 上述约束规则构成的集合记作 Φ_{mhb}. 此外, 受同一个锁对象保护的多个操作之间是互斥的, 它们无法并发且只能以两种不同的次序顺序执行. 如: O_3 与 O_6 都受到锁对象 18524392 的保护, UFO 方法通过约束 $O_5<O_6\vee O_{10}<O_3$ 来表达这两个操作间的互斥关系, 该类约束规则构成的集合记作 Φ_{lock}.

UFO 方法会在上述约束集合的限制下, 对原始轨迹中的操作序列进行重排. 为了保证重排后所得新序列满足数据一致性(即重排前后读操作获取到的值与原始程序轨迹中各读操作获取到的值一致), UFO 方法会进一步添加数据一致性相关的约束. 以表 2 中的轨迹为例, O_8 所读内存 18524364 中的值为 1, 为了保证轨迹重排后该操作仍然读到同样的值, UFO 方法会要求 O_2 必须先于 O_8 执行, 且“O_{12} 先于 O_2 执行”或者“O_8 先于 O_{12} 执行”,因此, UFO 方法会添加约束 $O_2<O_8\wedge(O_{12}<O_2\vee O_8<O_{12})$. 该类保证重排前后数据一致性的约束计入集合 Φ_{rw}.

最后, 为了触发某个 UAF 漏洞, UFO 方法会要求在该漏洞对应的 *free* 操作先于其对应的 *use* 操作执行. 例如, 为了验证 O_4 对应的 *free* 操作与 O_7 对应的 *use* 操作之间是否构成 UAF 漏洞, UFO 方法会添加约束 $O_4<O_7$ 来强制要求 *Use* 操作在 *Free* 操作后执行, 该类约束统一计入集合 Φ_{uaf}.

综合上述各类约束生成规则, UFO 方法由针对表 2 中的程序运行轨迹会构造表 3 所示的约束条件集. 得到该约束集合后, UFO 方法会使用 SMT (satisfiability modulo theories)[25]求解器分别对公式$(\Phi_{mhb}\wedge\Phi_{lock}\wedge\Phi_{rw})\wedge\Phi_{uaf_1}$ 和 $(\Phi_{mhb}\wedge\Phi_{lock}\wedge\Phi_{rw})\wedge\Phi_{uaf_2}$ 进行求解. 对于前者, 求解器将返回一个可行解 O_1-O_2-O_3-O_4-O_5-O_6-O_7-O_8-O_9-O_{10}-O_{11}-O_{12}-O_{13}-O_{14}-O_{15}, 这说明 Φ_{uaf_1} 中操作 O_7 对应的语句 S_{14}:$*q$=0 与操作 O_4 对应的语句 S_6:$free(q)$构成一个 UAF 漏洞; 而对于后者, 求解器没有给出任何可行解, 这说明操作 O_9 与操作 O_{14} 对应的程序语句 S_{16} 与 S_{11} 不会导致 UAF 漏洞. 该检测结果是正确的.

表 3 Program 1 实例的编码约束

约束条件集	约束
Φ_{mhb}	$O_1<O_2<O_3<O_4<O_5<O_{11}<O_{12}<O_{13}<O_{14}<O_{15}$, $O_6<O_7<O_8<O_9<O_{10}$, $O_1<O_6\wedge O_{10}<O_{15}$
Φ_{lock}	$O_5<O_6\vee O_{10}<O_3\vee(O_5<O_6\wedge O_{10}<O_{11})$
Φ_{rw}	$O_2<O_8\wedge(O_{12}<O_2\vee O_8<O_{12})$
Φ_{uaf_1}	$O_4<O_7$
Φ_{uaf_2}	$O_{14}<O_9$

然而, SMT 对约束条件进行求解时是盲目的, 无法以 Φ_{uaf_1} 或 Φ_{uaf_2} 为目标进行针对性的求解算法优化, 当程序规模大、约束规则多时, SMT 的求解检测效率低. 此外, 若将 Program 1 的语句 S_{12}:$join(thread_1)$调整到 S_5:$lock(l)$之前, 则 $thread_0$ 的所有 *free* 操作均在 $thread_1$ 的 *use* 操作后, 此时不可能存在 UAF 漏洞. 然而, UFO 方法仍会构建类似表 3 的约束集合并进行约束求解. 本文将通过逻辑时钟的方法识别 *use* 操作与 *free* 操作之间类似的因果关系, 在此基础上, 可避免类似的、没有必要的约束求解和 UAF 漏洞分析.

综上所述, 本文以每一组针对同一内存对象的 *Free*/*Use* 操作对为检测目标, 设计一种目标导向的 UAF 漏洞检测方法, 通过目标导向提高漏洞检测的效率. 与此同时, 提出一种新型向量时钟识别 *Free* 与 *Use* 操作之间的因果关系, 据此筛除不可能构成 UAF 漏洞的 *Free*/*Use* 操作对, 或者直接确定 UAF 漏洞的真实性, 以此提高 UAF 漏洞检测算法的性能. 下面给出具体介绍.

2 程序的 Petri 网模型构建

程序运行轨迹中, 各操作在控制流上的依赖关系可通过 Petri 网[22,26,27]予以描述. 我们在文献[28]中初步给出了由程序运行轨迹挖掘程序控制流 Petri 网模型的方法. 然而, 文献[28]中挖掘的 Petri 网模型以程序的数据竞争漏洞为检测目标, 未考虑 free 操作的建模, 也未考虑 UAF 漏洞检测过程中的数据一致性问题, 本文将针对上述问题对文献[28]中的工作进行完善.

2.1 多线程程序的运行轨迹

为了进行 UAF 漏洞检测, 约定一个多线程程序的运行轨迹是程序执行过程中各类 UAF 漏洞相关之并发原语形成的事件序列 α.

$\alpha \in Trace ::= Operation^*$

$Operation ::= start(u,v) \mid join(u,v) \mid lock(u,lid) \mid unlock(u,lid) \mid free(u,mid) \mid w(u,mid,x) \mid r(u,mid,x)$

其中:

- u, v 表示线程, lid 表示锁对象对应的 id, mid 表示内存对象的 id, x 表示内存对象中存储的值.
- $start(u,v)$: 表示线程 u 创建并启动一个新线程 v.
- $join(u,v)$: 表示线程 u 阻塞等待, 直至线程 v 终止再继续执行.
- $lock(u,lid)$和 $unlock(u,lid)$: 表示线程 u 获取或释放 id 为 lid 的锁对象.
- $free(u,mid)$: 表示线程 u 释放 id 为 mid 的内存对象.
- $w(u,mid,x)$和 $r(u,mid,x)$: 表示线程 u 向 id 为 mid 的内存对象中写入或读取到值 x.

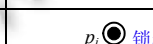
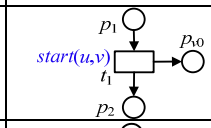
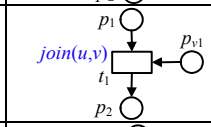
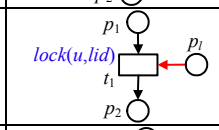
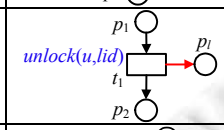
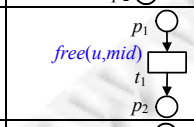
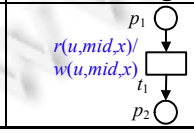
表 2 中给出的就是一个多线程程序运行轨迹的实例.

2.2 程序控制流Petri网模型的挖掘

Petri 网是一个四元组$\Sigma=(P,T;F,M_0)$, 其中, $P=\{p_1,p_2,\ldots,p_n\}$为库所集, $T=\{t_1,t_2,\ldots,t_n\}$为变迁集, $P\cap T=\varnothing$且$P\cup T\neq\varnothing$, $F=(P\times T)\cup(T\times P)$称为网的流关系, $M_0=P\to\{0,1,2,\ldots\}$为Σ的初始标识. 使用 Petri 网对多线程程序的控制流进行建模时, 每个变迁对应程序的一个操作. 库所分控制库所和资源库所两类: 控制库所用于建模线程的控制流状态, 资源库所用于建模程序中的锁对象. Petri 网中的标识表示程序的运行状态. 在程序运行之初, 仅有主线程的初始控制流库所和各个锁对象对应的资源库所各含有一个 token, 其余库所的 token 数为 0.

表 4 给出了由程序的运行轨迹构建程序控制流 Petri 网模型的一般规则. 相比文献[28]的工作而言, 表中进一步给出了 *free* 操作的 Petri 网模型片段, 适当补充了读写操作的相关信息, 并省略了共享变量对应的库所.

表 4 各类程序操作及常见对象的 Petri 网建模规则

程序对象及其操作	对应的 Petri 网模型片段	说明
主线程就绪状态		该库所表示主线程的就绪状态, 初始标识下它包含一个 token, 表示初始状态下主线程处于就绪状态
锁对象		该库所对应一个锁对象, 初始标识它包含一个 token, 表示初始状态下锁对象处于可用状态
$start(u,v)$		变迁 t_1 对应 *start* 操作, 其前驱库所 p_1 表示线程 u 中 *start* 操作的前驱控制流状态, 后继库所 p_2 表示线程 u 中 *start* 操作的后继控制流状态, 后继库所 p_{v0} 表示线程 v 的就绪状态
$join(u,v)$		变迁 t_1 对应 *join* 操作, 其前驱库所 p_1 表示线程 u 中 join 操作的前驱控制流状态, 前驱库所 p_{v1} 表示线程 v 的结束状态; 后继库所 p_2 表示线程 u 中 *join* 操作的后继控制流状态
$lock(u,lid)$		变迁 t_1 对应 id 为 lid 的锁对象的 *lock* 操作, 其前驱库所 p_1 表示线程 u 中 *lock* 操作的前驱控制流状态, 前驱库所 p_l 表示 id 为 lid 的锁对象; 后继库所 p_2 表示线程 u 中 *lock* 操作的后继控制流状态
$unlock(u,lid)$		变迁 t_1 对应 id 为 lid 的锁对象的 *unlock* 操作, 其前驱库所 p_1 表示线程 u 中 *unlock* 操作的前驱控制流状态; 后继库所 p_2 表示线程 u 中 *unlock* 操作的后继控制流状态, 后继库所 p_l 表示 id 为 lid 的锁对象
$free(u,mid)$		变迁 t_1 对应 id 为 mid 的内存对象的 *free* 操作, 其前驱库所 p_1 表示线程 u 中 *free* 操作的前驱控制流状态; 后继库所 p_2 表示线程 u 中 *free* 操作的后继控制流状态
$w(u,mid,x)$/ $r(u,mid,x)$		变迁 t_1 对应对 id 为 mid 的内存对象的读/写操作, 其前驱库所 p_1 表示线程 u 中读/写操作的前驱控制流状态; 后继库所 p_2 表示线程 u 中读/写操作的后继控制流状态

对给定的一个多线程程序运行轨迹, 据上述规则, 首先为主线程就绪状态和各个锁对象构建一个包含 token 的库所; 其次, 针对程序运行轨迹中的每个操作, 根据表 4 所给各种类操作的 Petri 网建模规则, 向程序的控制流 Petri 网模型中添加相应的变迁、库所及流关系即可. 以表 2 中的程序运行轨迹为例, 根据表 4 中的 Petri 模型构建规则可得图 2 所示的程序控制流模型.

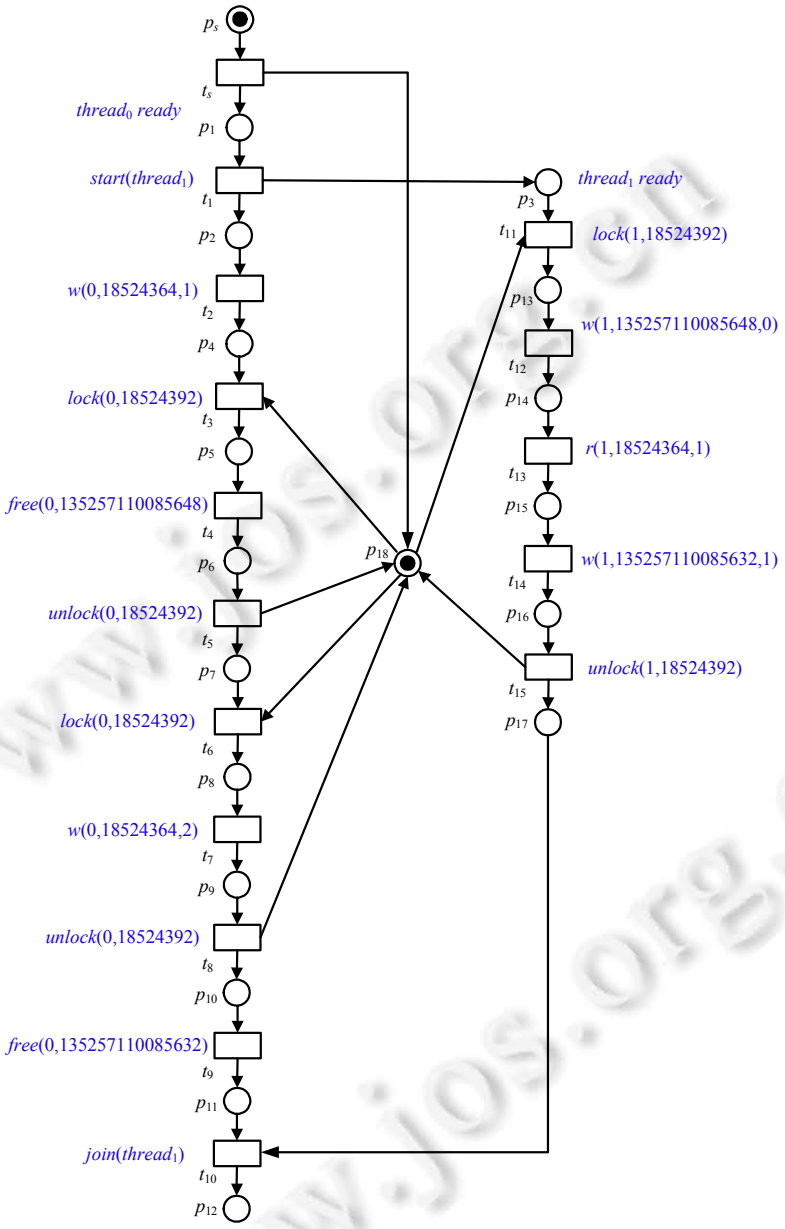


图 2 由表 2 运行轨迹挖掘得到的程序 Program 1 的控制流 Petri 网模型

2.3 数据一致性的Petri网建模

程序 Petri 网模型的各条可执行变迁序列对应原程序的一条可能的运行轨迹. 为了保证这些潜在轨迹满足数据一致性[29], 下面以一个简单场景为例, 说明如何在程序控制流 Petri 网模型的基础上添加相应的 Petri 网行为控制结构.

假设最初给定的程序运行轨迹中有一个读操作 $r(u,mid,x)$, 有且仅有两个向内存对象 mid 写入值 x 的写操作 $w_1(v,mid,x)$与 $w_2(t,mid,x)$. 为了保证 Petri 网可执行序列对应的潜在运行轨迹满足数据一致性, 则或者 $r(u,mid,x)$与 $w_1(v,mid,x)$之间不存在任何其他关于 mid 的写操作, 或者 $r(u,mid,x)$与 $w_2(t,mid,x)$之间不存在任何其他关于 mid 的写操作, 两者有且只有一个成立. 为了在程序 Petri 网模型中添加这种行为约束, 假设 $r(u,mid,x)$, $w_1(v,mid,x)$与 $w_2(t,mid,x)$在程序 Petri 网模型中对应的变迁分别为 t_r, t_w_1 和 t_w_2, 需要将变迁 t_r 拆分为 t_r_1 与 t_r_2 这两个变迁, 拆分前 t_r 的前驱库所 p_pre_r 与后继库所 p_post_r 仍然作 t_r_1 与 t_r_2 的前驱库所和后继库所(称 t_r_1 与 t_r_2 为 t_r 的两个克隆变迁).

除此之外, 在程序 Petri 网模型的基础上, 添加图 3 中绿色和黄色的库所、变迁和流关系, 这些网元素的含义为:

(1) 库所 p_choice 及其后继变迁 t_case_1, t_case_2 分别表示数据一致性的两种不同的场景, 前者意味着读操作 $r(u,mid,x)$观察到的 x 值是写操作 $w_1(v,mid,x)$写入的, 后者意味着读操作 $r(u,mid,x)$观察到的 x 值是写操作 $w_2(t,mid,x)$写入的.

(2) 库所 p_ctrl_1 是一个行为控制库所, 一旦其中含有一个 token, 则意味着读操作 $r(u,mid,x)$观察到的 x 值是 $w_1(v,mid,x)$写入的. 而且在 p_ctrl_1 含有 token 的状态下, $w_1(v,mid,x)$尚未执行, 此时, $w_1(v,mid,x)$之外任何一个对 mid 的写操作可不受限制地执行(这通过库所 p_ctrl_1 与 w_1^c 之间的双向弧表示, 而变迁 w_1^c 表示 $w_1(v,mid,x)$之外任何一个内存空间 mid 关联的写操作).

(3) 库所 $p\_consist\_1$ 是一个保证数据一致性的关键库所, 一旦其中含有一个 token, 则意味着写操作 $w_1(v,mid,x)$已经执行, 而读操作 $r(u,mid,x)$尚待读取这个写入的值. 此时, 任何其他内存空间 mid 关联的写操作都不应该执行. 这一点在图 3 中的网结构中是可以保证的, 因为此时的 p_ctrl_1 中没有 token, 这导致 w_1^c 无法使能. 不过, 待 t_r_1 对应的读操作 $r(u,mid,x)$执行后, p_ctrl_1 会重新获得 token, mid 关联的其他写操作就不再受约束.

(4) 网元素 p_ctrl_2, $p\_consist\_2$, w_2^c 具有与 p_ctrl_1, $p\_consist\_1$, w_1^c 类似的含义与功能, 只不过对应的场景不同, 它们对应着读操作 $r(u,mid,x)$观察到的 x 值由 $w_2(v,mid,x)$写入的场景.

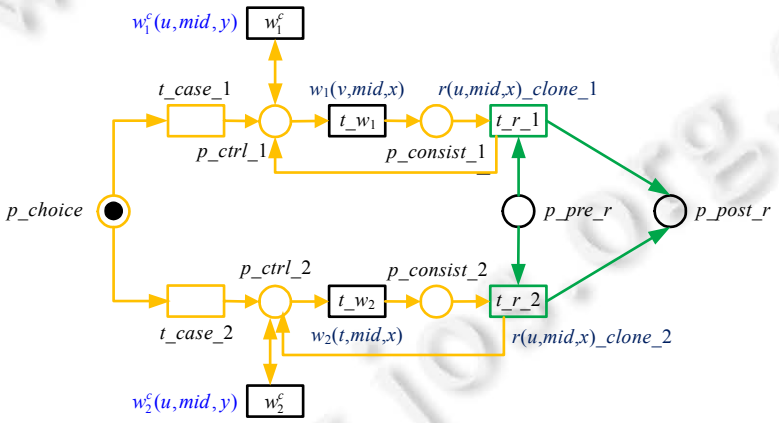


图 3 保持数据一致性的 Petri 网控制结构示意图

需要说明的是, 图 3 给出的场景中, 读操作 $r(u,mid,x)$读到的值既可以是写操作 $w_1(v,mid,x)$写入的, 也可以是写操作 $w_2(t,mid,x)$写入的. 在具体的应用中, 写操作可能有更多的选择, 此时仅需添加更多组 p_choice, p_ctrl, $p\_consist$ 和 $r(u,mid,x)\_clone$ 相关的网络结构即可. 相反地, 若仅有一个可选择的写操作 $w_1(v,mid,x)$, 则只保留一组网络控制结构, 而且读操作 $r(u,mid,x)$对应的变迁无须拆分克隆.

以图 2 中的程序 Petri 网模型为例, 该模型中, 变迁 t_{13} 对应一个读操作 $r(1,18524364,1)$, 读到的值为 1. 模型中只有一个向内存单元 18524364 中写入值 1 的操作(对应变迁 t_2), 因此如图 4 所示, 只需添加黄色显示的

一组 *p_choice*, *p_ctrl*, *p_consist* 相关网络控制结构, 而且读操作 *r*(1,18524364,1)对应的变迁 t_{13} 无须拆分克隆.

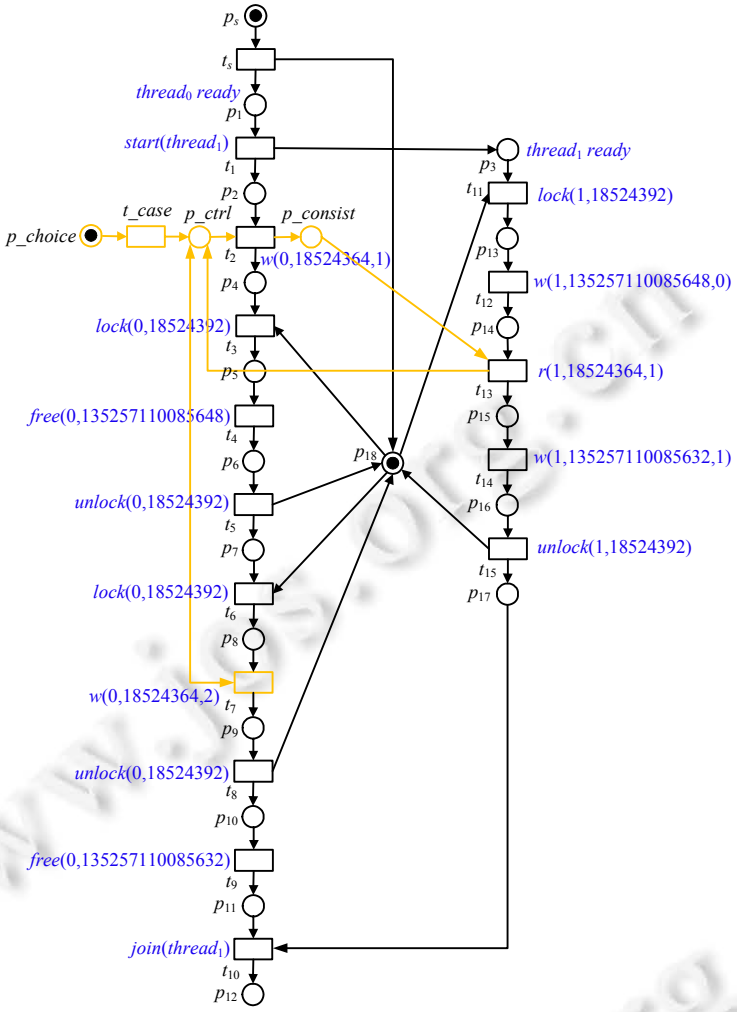


图 4 保持数据一致性的程序 Petri 网模型控制结构实例

根据上述保持数据一致性的基本思想, 可得在程序 Petri 网模型中添加控制结构以保证数据一致性的算法如下(其最坏时间复杂度为 $O(|T|^4)$, 其中, $|T|$为程序 Petri 网模型中变迁的数量, 即程序轨迹中操作的数量).

算法 1. 保持数据一致性的 Petri 网模型生成算法.

输入: 由程序运行轨迹 trace 挖掘到的程序 Petri 网模型$\Sigma=(P,T;F,M_0)$.

输出: 为了保持数据一致性而添加行为控制结构后的 Petri 网 $\Sigma'=(P',T';F',M_0')$.

步骤:

0. 初始化 $(P':=P,T':=T;F':=F,M_0':=M_0)$
1. **FOREACH** ($t\in T$)
2. **IF** (变迁 t 关联的操作是一个读操作, 不妨设其为 t_r, 读操作为 $r(u,mid,x)$)
3. **FOREACH** ($t'\in T$)
4. **IF** (t'关联的操作向内存 mid 中写入值 x, 且 t'对应的操作不是线程 u 的位于 $r(u,mid,x)$后的操作)
5. 将 t'加入集合 $T\_w(mid,x)$;
6. **ELSEIF** (t'关联的操作向 mid 中写入一个非 x 的值, 且 t'对应的操作非线程 u 的位于 $r(u,mid,x)$

后的操作)

7. 将 t'加入集合 $T\_w(mid,*)$;
8. **IF** ($|T\_w(mid,x)|\geqslant 1$)
9. 向 P'中添加一个新库所 *p_choice*, 并令 $M'_0(p\_choice)=1$;
10. **FOREACH** ($t\_w\in T\_w(mid,x)$)
11. 向 T'中添加一个新变迁 *t_case*, 向 P'中添加两个无 token 的新库所 *p_ctrl*, *p_consist*;
12. **IF** (t_w 不是 $T_w(mid,x)$中的第 1 个元素)
13. 向 T'中添加一个 *t_r* 的克隆变迁 *t_r_clone*;
14. 在 T'中添加两个新的流关系, 使得 *t_r_clone* 与 *t_r* 具有相同的前驱库所和后继库所;
15. **ELSE**
16. 设置 *t_r_clone* 为变迁 *t_r* 的别名;
17. $F':=F'\cup\{\langle p\_choice,t\_case\rangle,\langle t\_case,p\_ctrl\rangle,\langle p\_ctrl,t\_w\rangle,\langle t\_w,p\_consist\rangle,\langle p\_consist,t\_r\_clone\rangle,\langle t\_r\_clone,p\_ctrl\rangle\}$
18. **FOREACH** ($t\_w^c\in(T\_w(mid,x)\cup T\_w(mid,*)-\{t\_w\})$)
19. $F':=F'\cup\{\langle p\_ctrl,t\_w^c\rangle,\langle t\_w^c,p\_ctrl\rangle\}$
20. RETURN $\Sigma'=(P',T';F',M'_0)$

2.4 UAF漏洞的伴生Petri网

对每一个潜在的 UAF 漏洞, 假设其对应的内存 *Free* 操作为 O_f, 内存 *Use* 操作为 O_u, 为了触发该漏洞, 需使 O_f 在 O_u 之前发生. 用 Petri 网建模这一依赖关系时, 仅需在 O_f 所对应变迁与 O_u 所对应变迁之间添加一个初始 token 为 0 的中间库所(作为 O_f 的后继、O_u 的前驱), 并为 O_u 所对应的变迁添加一个 token 为 0 的后继库所 *p_UAF*. 如此一来, 如果该潜在 UAF 漏洞可触发, 则存在一个 Petri 网的可达标识, 使得库所 *p_UAF* 中存在一个 token. 由此, 可借助该 Petri 网的可覆盖性[22]对 UAF 漏洞的真实性进行检测. 如此得到的 Petri 网称为当前潜在 UAF 漏洞的伴生 Petri 网, 库所 *p_UAF* 称为当前潜在 UAF 漏洞的标志库所. 显然, 若一个 UAF 漏洞是真实的, 则其伴生 Petri 网中的标志库所在某个可达标识中是可覆盖的.

在图 4 所示的添加数据一致性约束后的程序 Petri 网模型中, 存在两个潜在的 UAF 漏洞, 即变迁 t_4 与变迁 t_{12} 对应的内存 *free* 与 *use* 操作对以及变迁 t_9 与变迁 t_{14} 对应的内存 *free* 与 *use* 操作对. 两者对应的 UAF 伴生 Petri 网模型如图 5 所示. 其中, 红色的库所、流关系是为了验证 UAF 漏洞的真实性而填写的网元素. 库所 *p_UAF*_1, *p_UAF*_2 分别是两个 UAF 潜在漏洞的标志库所. 下节就 UAF 漏洞标志库所的可覆盖性判定方法进行说明.

3 基于 Petri 网反向展开的 UAF 漏洞检测

第 2.4 节已经指出, 得到潜在 UAF 漏洞的伴生 Petri 网后, 通过该漏洞标志库所的可覆盖性能进行漏洞真实性检测. 我们在文献[20]中提出了一种基于反向展开对 Petri 网目标标识进行可覆盖性判定的方法, 本文基于该方法进行 UAF 的漏洞检测. 在此之前, 先对 Petri 网反向展开的相关概念及其用于 UAF 检测的原理进行介绍.

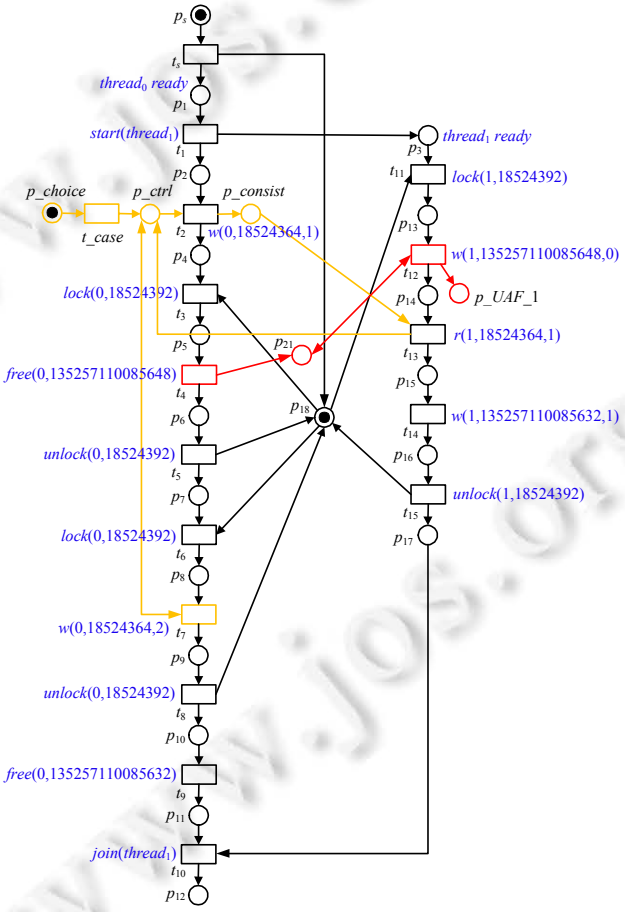
Petri 网的展开技术[24]利用出现网描述网系统的行为. 所谓出现网是一个三元组 $ON=(B,E;G)$, 其中, B 为条件集, E 为事件集, 它们满足: (1) $\forall b\in B:|{}^*b|\leqslant 1$; (2) $G^+\cap(G^{-1})^+=\varnothing$, 即网中不含有向圈; (3) $\forall y\in B\cup E:\neg(y\#y)$, 即网中不存在自冲突; (4) $\forall y\in B\cup E$, 集合 $\{x|x\in B\cup E\wedge(x,y)\in G^+\}$ 是有限的, 即任意节点的前驱节点都是有限的.

基于出现网描述 Petri 网行为的基本思想如下: 用出现网中的一个事件表示 Petri 网系统中某个变迁的一次执行, 用出现网中的一个条件表示网系统运行中涉及的某个 token, 出现网中的流关系用以刻画原始 Petri 网中变迁执行消耗或生成 token 的情况, Petri 网初始标识下的各个 token 分别对应出现网中一个没有前驱的条件. 由于出现网对于并发行为的建模和分析无须穷举所有可能的交错, 故能在一定程度上缓解 Petri 网性质分

析中的状态爆炸问题. 但传统的展开方法仍然包含了系统的所有状态信息, 对所有这些状态的分析仍可能导致无法接受的性能问题. 实际上, 类似本文仅需对 UAF 漏洞标志库所的可覆盖性进行判断, 某些应用问题仅需对系统特定状态的可覆盖性进行判定, 此时, Petri 网的反向展开方法有望缩减状态分析的规模.

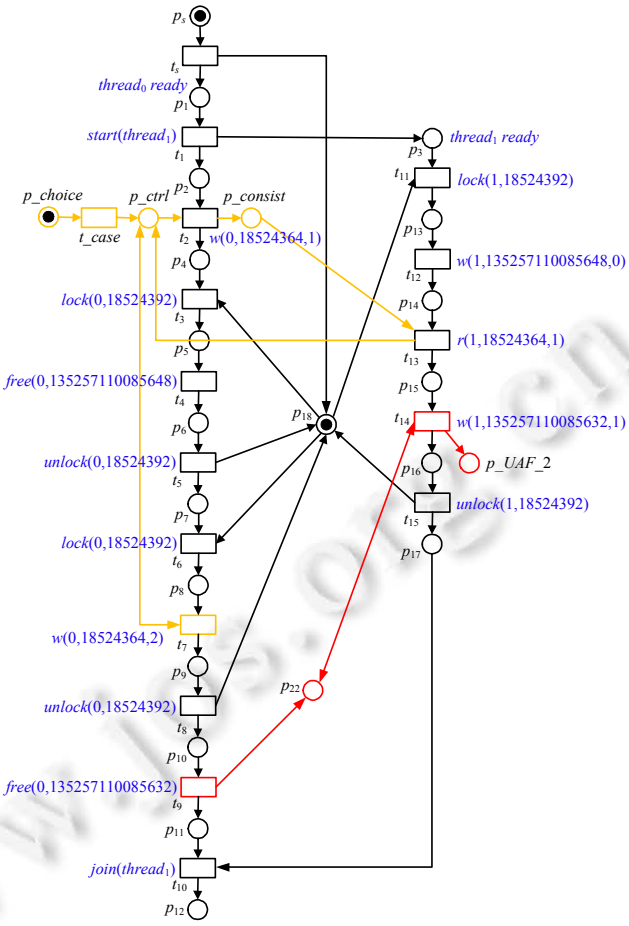
Petri 网的反向展开从需要做出可覆盖性判定的目标标识出发, 仅刻画与可覆盖性判定相关的系统状态. 在UAF漏洞检测的应用场景中, 我们以UAF漏洞标志库所的可覆盖性为目标导向, 仅需从标志库所出发进行反向展开, 在此过程中, 不断检测是否存在一个可达标识覆盖该标志库所即可, 由此可得一种目标导向的、基于 Petri 网反向展开的 UAF 漏洞检测方法.

反向展开进行 UAF 漏洞检测的主要过程如下: 初始时, 反向展开只有一个 UAF 漏洞标志库所对应的条件, 记其对应的标识为 M'_f; 之后, 反向展开算法会基于 M'_f 计算初始的反向扩展集 *RExt*; 随后, 在 *RExt* 不为空的情况下, 算法不断的从中选择一个反向扩展对展开网 *RUnf* 进行扩充. 假设选择的扩展为 *rext*=(*t*,*C*), 则在反向展开 *RUnf* 中创建相应的事件 *e*=(*t*,*C*). 若 *e* 经判定不是反向截断事件, 则根据 UAF 伴生 Petri 网中 *t* 的前驱库所向 *RUnf* 中添加相应的条件. 之后, 根据新的 *Runf* 中更新反向扩展集. 在上述对 Petri 网的反向展开不断扩充的过程中, 若检测到一个事件 *e* 使得其反向配置对应的 Petri 网标识能被 UAF 伴生 Petri 网的初始标识覆盖, 则说明目标标志库所是可覆盖的, 当前 UAF 漏洞可触发, 终止算法; 否则, 若 *RExt* 为空、反向展开结束时仍不存在可被 UAF 伴生 Petri 网初始标识覆盖的反向配置, 则说明目标标志库所不可覆盖的, 当前 UAF 漏洞不可触发.



(a) t_4 与 t_{12} 所对应潜在 UAF 漏洞的伴生 Petri 网

图 5 UAF 伴生 Petri 网模型实例



(b) t_9 与 t_{14} 所对应潜在 UAF 漏洞的伴生 Petri 网

图 5　UAF 伴生 Petri 网模型实例(续)

以图 5(a)中的 UAF 伴生 Petri 网为例, 判断该 UAF 漏洞的真实性需要对目标标识 M_f={*p_UAF*_1}的可覆盖性进行判定, 利用反向展开对此进行判定的过程见表 5. 其中, UAF 漏洞伴生 Petri 网的反向展开 *RUnf* 如图 6 所示. 在第 10 步反向展开的时候, 因反向配置[e_9]对应的标识{p_s,*p_choice*}可被原伴生 Petri 网的初始标识{p_s,*p_choice*}覆盖, 由此可知{*p_UAF*_1}是可覆盖的,这说明漏洞{*p_UAF*_1}是可以被触发的, 即程序 Program 1 的语句 S_{14}:*q=0 与语句 S_6:*free*(q)间存在 UAF 漏洞.

表 5　UAF 漏洞伴生 Petri 网的反向展开与漏洞检测实例

反向展开的各个步骤	每一步对应的反向扩展集 *RExt*
1. 初识的反向展开 *RUnf* 中只包含条件 c_1	{$rext_1$=(t_{12},{c_1})}
2. 选择 $rext_1$ 进行扩展, 在 *RUnf* 中创建事件 e_1=(t_{12},{c_1})及其前驱{c_2,c_3}	{$rext_2$=(t_4,{c_2}),$rext_3$=(t_{12},{c_2}),$rext_4$=(t_{11},{c_3})}
3. 选择 $rext_2$ 进行扩展, 在 *RUnf* 中创建事件 e_2=(t_4,{c_2})及其前驱{c_4}	{$rext_3$=(t_{12},{c_2}),$rext_4$=(t_{11},{c_3}),$rext_5$=(t_3,{c_4})}
4. 选择 $rext_3$ 进行扩展, 在 *RUnf* 中创建事件 e_3=(t_{12},{c_2}), 由反向截断事件定义, e_3 会因 e_1 而截断,此次扩展终止	{$rext_4$=(t_{11},{c_3}),$rext_5$=(t_3,{c_4})}
5. 选择 $rext_4$ 进行扩展, 在 *RUnf* 中创建事件 e_4=(t_{11},{c_3})及其前驱{c_5,c_6}	{$rext_5$=(t_3,{c_4}),$rext_6$=(t_5,{c_5}),$rext_7$=(t_8,{c_5}), $rext_8$=(t_{14},{c_5}),$rext_9$=(t_s,{c_5}),$rext_{10}$=(t_1,{c_6})}
6. 选择 $rext_5$ 进行扩展, 在 *RUnf* 中创建事件 e_5=(t_3,{c_4})及其前驱{c_7,c_8}. 生成新扩展(t_5,{c_8}), (t_5,{c_5}), (t_5,{c_5,c_8}), (t_8,{c_8}), (t_8,{c_5}), (t_8,{c_5,c_8}), (t_{14},{c_8}), (t_{14},{c_5}), (t_{14},{c_5,c_8}), (t_s,{c_8}), (t_s,{c_5}), (t_s,{c_5,c_8}), 由 *RExt* 的限制条件, 只保留(t_5,{c_5,c_8}), (t_8,{c_5,c_8}), (t_{14},{c_5,c_8}), (t_s,{c_5,c_8})	{$rext_{10}$=(t_1,{c_6}),$rext_{11}$=(t_5,{c_5,c_8}), $rext_{12}$=(t_8,{c_5,c_8}),$rext_{13}$=(t_{14},{c_5,c_8}), $rext_{14}$=(t_s,{c_5,c_8}),$rext_{15}$=(t_2,{c_7})}

表 5 UAF 漏洞伴生 Petri 网的反向展开与漏洞检测实例(续)

反向展开的各个步骤	每一步对应的反向扩展集 RExt
7. 选择 $rext_{15}$ 进行扩展, 在 *RUnf* 中创建事件 $e_6=(t_2,\{c_7\})$ 及其前驱$\{c_9,c_{10}\}$. 生成新扩展$(t_1,\{c_9\})$, $(t_1,\{c_6\})$,$(t_1,\{c_6,c_9\})$, 由 *RExt* 的限制条件, 只保留$(t_1,\{c_6,c_9\})$	$\{rext_{11}=(t_5,\{c_5,c_8\}),rext_{12}=(t_8,\{c_5,c_8\}),$ $rext_{13}=(t_{14},\{c_5,c_8\}),rext_{14}=(t_s,\{c_5,c_8\}),$ $rext_{16}=(t_1,\{c_6,c_9\}),rext_{17}=(t_{15},\{c_{10}\}),$ $rext_{18}=(t_7,\{c_{10}\}),rext_{19}=(t_{13},\{c_{10}\})\}$
8. 选择 $rext_{16}$ 进行扩展, 在 *RUnf* 中创建事件 $e_7=(t_1,\{c_6,c_9\})$及其前驱$\{c_{11}\}$. 生成新扩展$(t_s,\{c_{11}\})$, $(t_s,\{c_5\})$, $(t_s,\{c_8\})$, $(t_s,\{c_{11},c_5\})$, $(t_s,\{c_5,c_8\})$, $(t_s,\{c_{11},c_8\})$, $(t_s,\{c_5,c_8,c_{11}\})$, 由 *RExt* 的限制条件, 只保留$(t_s,\{c_5,c_8,c_{11}\})$	$\{rext_{11}=(t_5,\{c_5,c_8\}),rext_{12}=(t_8,\{c_5,c_8\}),$ $rext_{13}=(t_{14},\{c_5,c_8\}),rext_{17}=(t_{15},\{c_{10}\}),$ $rext_{18}=(t_7,\{c_{10}\}),rext_{19}=(t_{13},\{c_{10}\}),$ $rext_{20}=(t_s,\{c_5,c_8,c_{11}\})\}$
9. 选择 $rext_{17}$ 进行扩展, 在 *RUnf* 中创建事件 $e_8=(t_{15},\{c_{10}\})$及其前驱$\{c_{12}\}$	$\{rext_{11}=(t_5,\{c_5,c_8\}),rext_{12}=(t_8,\{c_5,c_8\}),$ $rext_{13}=(t_{14},\{c_5,c_8\}),rext_{18}=(t_7,\{c_{10}\}),$ $rext_{19}=(t_{13},\{c_{10}\}),rext_{20}=(ts,\{c_5,c_8,c_{11}\})\}$
10. 选择 $rext_{20}$ 进行扩展, 在 *RUnf* 中创建事件 $e_9=(t_s,\{c_5,c_8,c_{11}\})$及其前驱$\{c_{13}\}$, 此时, $[e_9]$对应的标识$\{p_s,p_choice\}$可被网系统的初始标识$\{p_s,p_choice\}$覆盖, 由此可知, 存在一个可达标识使得UAF标志库所 *p_UAF*_1 中含有 token, 其对应的 UAF 漏洞可以被触发,算法结束	–

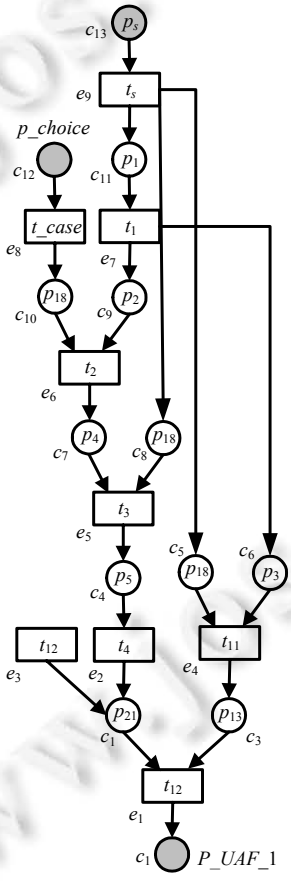


图 6 UAF 漏洞伴生 Petri 网的反向展开 *RUnf*

关于 Petri 网反向展开的具体方法及其进行可覆盖性判定的算法见文献[20].

4 基于向量时钟的 UAF 漏洞筛选与算法优化

按上一节的方法进行 UAF 漏洞检测时, 只要是针对同一内存空间的 *Free* 操作和 *Use* 操作, 均需构造其伴生 Petri 网并检测相应 UAF 标志库所的可覆盖性. 实际上, 若操作 O_f 与 O_u 存在因果依赖关系, 当 O_f 是因而 O_u 是果时, 它们必然会构成 UAF 漏洞; 当 O_u 是因而 O_f 是果时, 它们必然不构成 UAF 漏洞. 基于向量时钟[30]的方法可高效判定程序轨迹中操作间是否具有因果依赖关系. 基于此, 本文拟用向量时钟的方法挖掘 *Free* 和 *Use* 操作对之间的因果关系, 剔除具有因果关系的 *free-use* 操作对, 以此减少通过反向展开进行检测的潜在 UAF 漏洞数量.

4.1 向量时钟及其更新规则

向量时钟是在时间戳[30]的基础上演变而来的一种逻辑时钟方法, 使用向量时钟可以准确地识别和挖掘多线程程序中各操作间的因果关系. 如文献[30]以多线程程序的数据竞争检测为目标, 设计了一套向量时钟及其计算规则, 据此进行读写操作之间的可并发性识别. 然而, 已有方法通常将锁竞争导致的互斥关系借助向量时钟硬性处理为因果约束, 这在提高程序分析效率的同时导致诸多漏报. 本文对传统向量时钟计算规则进行修改, 不考虑操作间互斥关系以因果关系识别为目标, 仅在必要时根据向量时钟添加因果约束. 这既简化了向量时钟的计算方法, 也降低漏报率. 至于操作间互斥关系的处理, 本文交由第 3 节 Petri 网反向展开的方法处理.

为了挖掘运行轨迹中各操作间的因果关系, 本文为每个线程、每步操作都绑定了一组向量时钟, 相关定义如下.

定义 1(线程纪元与向量时钟). 在程序执行的任意时刻, 为每一个线程 t 绑定一个整数形式的逻辑时钟 $c\in N$, 称 $t@c$ 为一个线程纪元, 并约定线程 t 的初始最小纪元为 $t@0$, 记作$\perp_t$; 线程纪元构成的向量称为向量时钟, 记作 V, 最小向量时钟记为$\perp_v$, 其取值为$\lambda t\in Tid$: $t@0$ (其中, $\lambda t\in Tid$ 表示“对每一个线程 t”).

线程纪元和向量时钟满足如下运算规则, 其中, $V(t)$为向量时钟 V 下线程 t 的线程纪元.

(1) 线程纪元比较规则: $t@c_1<t@c_2$ iff $c_1<c_2$.

(2) 线程纪元求最大值: $\max(t@c_1<t@c_2)=t@\max(c_1,c_2)$.

(3) 线程纪元递增规则: $t@c+1=t@(c+1)$.

(4) 向量时钟比较规则: $V_1\subseteq V_2$ iff $(\forall t\in Tid{:}V_1(t)\leqslant V_2(t))\wedge(\exists t\in Tid{:}V_1(t)\leqslant V_2(t))$.

(5) 向量时钟求最大值: $V_1\cup V_2=\lambda t\in Tid$: $\max(V_1(t),V_2(t))$.

(6) 向量时钟递增规则: $inc_t(V)=\lambda u\in Tid$: if $u=t$ then $V(u)+1$ else $V(u)$.

对任意一个线程 t, 为其设置一个线程局部向量时钟, 记作 V^{self}. 在程序未执行任何操作的初始状态下, 初始化其值为$\perp_v$. 以表 2 中 Program 1 的运行轨迹为例, 线程 $thread_0$ 与 $thread_1$ 未执行任何操作时的纪元分别为 $thread_0@0$, $thread_1@0$, 因此, Program 1 对应的最小向量时钟为$\langle thread_0@0,thread_1@0\rangle$ (可简写为$\langle 0,0\rangle$). 接下来, 对于运行轨迹中的各个操作, 均根据其操作种类更新操作执行后各线程的向量时钟. 最后, 根据各个操作执行前后的向量时钟取值进行操作间因果关系的判定. 下面逐一说明.

记一个事件执行前各个线程的向量时钟相关数据对象处于状态 S, 事件执行后的状态记作 S', 并记状态 S 下线程 t 的局部向量时钟为 $S_t.V^{self}$. 图 7 是根据一个程序运行轨迹对线程向量时钟相关数据对象进行更新的一个实例(实线有向弧表示操作间的线程内因果依赖关系, 虚线表示线程间的因果依赖关系). 下面结合该实例对线程局部向量时钟的更新规则及其含义进行说明. 更新后的线程局部向量时钟用来记录操作执行完毕后线程所能处于的最小时间向量值.

向量时钟在线程执行过程中, 为每个线程 t 绑定一个整数类型的逻辑时钟 c, 将 $t@c$ 表示为一个线程纪元, 线程纪元组成的向量即为向量时钟, 初识线程纪元为 $t@0$.

- START 规则

当线程 u 执行 $start(u,v)$操作后, 一方面应将线程 u 的局部向量时钟进行自增, 即令 $S'_u.V^{self}:=inc_u(S_u.V^{self})$;

另一方面, 考虑到线程 v 的操作仅当该操作执行完毕才能开始执行, 故将线程 v 的局部向量时钟更新为 $S_v.V^{self}$ 与 $inc_u(S_u.V^{self})$的最大值, 即令 $S'_v.V^{self} := S_v.V^{self} \cup inc_u(S_u.V^{self})$. 其他线程的局部向量时钟保持不变.

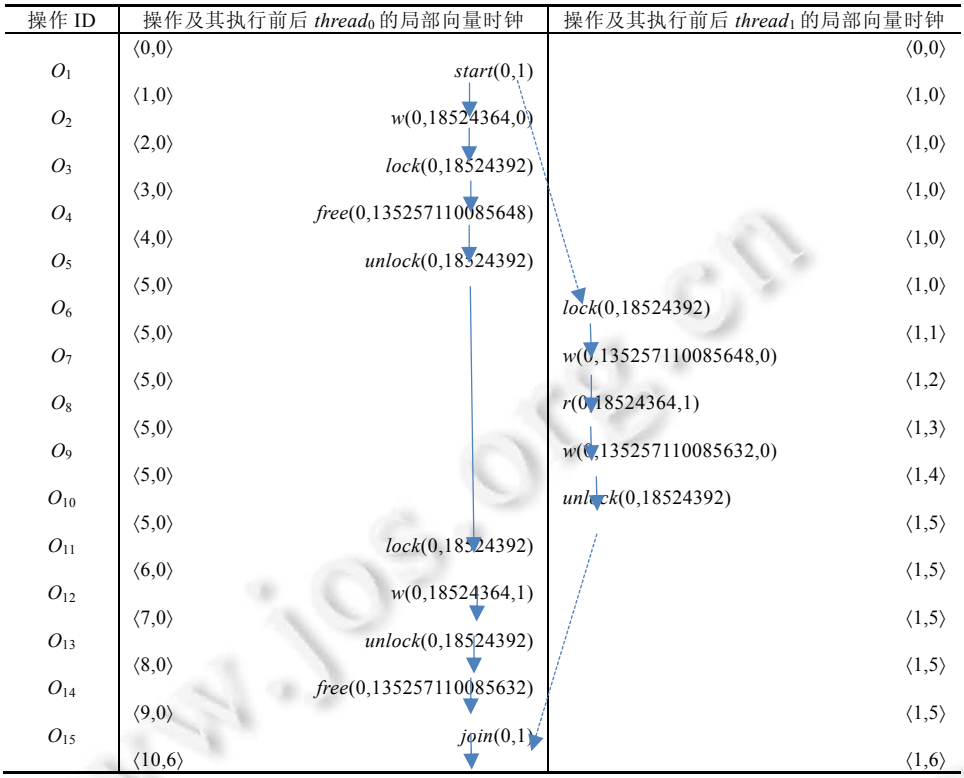


图 7 根据程序运动轨迹进行向量时钟计算的实例

以图 7 中的操作 O_1:*start*(0,1)为例, 线程 $thread_0$ 执行该操作前的局部向量时钟为 $S_0.V^{self}$=⟨0,0⟩, 执行该操作后, 线程 $thread_0$ 的局部向量时钟递增为 $S'_0.V^{self} = inc_0(S_0.V^{self}) = inc_0(\langle 0,0\rangle) = \langle 1,0\rangle$, 线程 $thread_1$ 的局部向量时钟更新为 $S'_1.V^{self} = S_1.V^{self} \cup inc_0(S_0.V^{self}) = \langle 0,0\rangle \cup \langle 1,0\rangle = \langle 1,0\rangle$.

- JOIN 规则

当线程 u 执行 *join*(u,v)操作后, 意味着线程 v 终止; 同时, 线程 t 被从阻塞状态唤醒, 两者具有因果依赖关系. 故除了将线程 v 的局部向量时钟更新为 $S'_v.V^{self} := inc_v(S_v.V^{self})$ 外, 线程 u 的局部向量时钟应更新为 $inc_u(S_u.V^{self})$与 $inc_v(S_v.V^{self})$的最大值, 即令 $S'_u.V^{self} := inc_u(S_u.V^{self}) \cup inc_v(S_v.V^{self})$. 其他线程的向量时钟保持不变.

以图 7 中的操作 O_{15}:*join*(0,1)为例, 该操作执行前, $thread_0$ 和 $thread_1$ 的局部向量时钟分别为⟨9,0⟩和⟨1,5⟩, 后继状态中, $thread_1$ 的向量时钟更新为 $inc_1(\langle 1,5\rangle)$=⟨1,6⟩, $thread_0$ 的向量时钟更新为 $inc_0(\langle 9,0\rangle)\cup\langle 1,6\rangle$=⟨10,6⟩.

- LOCAL 规则

当线程 u 执行 *lock*, *unlock*, *free* 或共享变量的读写操作后, 由于这些操作通常不会引起不同线程内操作之间的因果依赖关系, 因此, 这些操作执行后仅对线程 u 的局部向量时钟执行自增操作即可, 即令 $S'_u.V^{self} := inc_u(S_u.V^{self})$, 其他线程的局部向量时钟保持不变.

以图 7 中的操作 O_2:*w*(0,18524364,0)为例, 该操作执行前, $thread_0$ 的局部向量时钟为⟨1,0⟩, 后继状态中, $thread_0$ 的局部向量时钟更新为 $inc_0(\langle 1,0\rangle)$=⟨2,0⟩. 其余操作更新规则相同.

根据上述向量时钟更新规则, 不难发现, 每个操作执行前, 其所属线程的局部向量时钟是该操作能开始

执行的最早全局时间(全局时间是指既考虑当前线程的纪元, 也考虑其余线程的纪元); 而且, 假设当前操作属于线程 u, 则该向量时钟中与线程 u 相对应的分量是该操作执行时的线程局部时间(即当前线程的纪元). 以 $thread_0$ 的操作 O_{15} 为例, 其执行前的线程局部向量时钟为⟨9,0⟩, 这意味着该操作在 $thread_0$ 处于时刻 9 时开始执行, 最早在线程 $thread_1$ 处于 0 时刻时开始执行. 操作执行后的向量时钟是该操作能结束的最早全局时间, 而且对于线程 u 之外的任意线程 v, 该操作的最早结束时间对应着该向量时钟中与线程 v 关联的分量. 以 O_{15} 为例, 其执行后, $thread_0$ 的局部向量时钟为⟨10,6⟩, 这意味着该操作最早能在线程 $thread_0$ 处于时刻 10 时结束执行, 最早能在线程 $thread_1$ 处于时刻 6 时结束执行.

4.2 基于向量时钟的因果关系挖掘与UAF检测算法优化

根据上节所述向量时钟更新规则及各操作执行前后线程局部向量时钟, 可得操作间因果关系判定准则.

定理 1. 对于程序运行轨迹中的任意两个操作 O_i 与 O_j, 假设 O_i 出现在 O_j 之前(即 $i<j$), O_i 属于线程 u, O_i 执行前, 线程 u 的局部向量时钟关于线程 u 的分量为 $u\_timestmp\_pre\_O_i$. 设 O_j 属于线程 v, O_j 执行后, 线程 v 的局部向量时钟关于线程 u 的分量为 $u\_timestmp\_post\_O_j$. 若 $u\_timestmp\_pre\_O_i<u\_timestmp\_post\_O_j$, 则 O_i 与 O_j 具有因果依赖关系, 即 O_i 需要先于 O_j 执行, 记作 $O_i \rightarrow O_j$.

证明: 根据第 4.1 节末尾所述操作执行前后向量时钟的含义, 操作 O_j 执行后, 线程 v 的局部向量时钟中关于线程 u 的分量若是 $u\_timestmp\_post\_O_j$, 则意味着操作 O_j 最早能在线程 u 处于时刻 $u\_timestmp\_post\_O_j$ 时结束执行; 类似地, 若操作 O_i 执行前线程 u 的局部向量时钟中关于线程 u 的分量是 $u\_timestmp\_pre\_O_i$, 则意味着操作 O_i 在线程 u 处于时刻 $u\_timestmp\_pre\_O_i$ 时开始执行. 因此, 若 $u\_timestmp\_pre\_O_i<u\_timestmp\_post\_O_j$ 成立, 则意味着 O_j 的最早执行时间大于 O_i 的执行时间. 由此可知, O_i 需要先于 O_j 执行, 即 $O_i \rightarrow O_j$ 成立.

以图 7 中的程序运行轨迹与各操作对应的向量时钟为例, 操作 O_4 在 O_7 之前执行, O_4 属于 $thread_0$, 执行前, $thread_0$ 的局部向量时钟为⟨3,0⟩, 这意味着 O_4 在 $thread_0$ 处于 3 时刻时执行; O_7 属于 $thread_1$, 执行后, $thread_1$ 的局部向量时钟为⟨1,2⟩, 这意味着 O_7 在 $thread_0$ 处于 1 时刻时执行. 由此可见, O_4 与 O_7 无必然先后执行顺序.

进一步地, 鉴于 O_4 是内存块 135257110085648 的 *Free* 操作, O_7 是该内存块 135257110085648 的 *Use* 操作, 不能判断这两个操作是否能构成 UAF 漏洞. 从而, 针对这一潜在的 UAF 漏洞, 需再通过前述 Petri 网反向展开的方法验证其真伪. 此外, 若基于向量时钟判定 $O_4 \rightarrow O_7$ 成立, 则意味着该这两个操作构成 UAF 漏洞; 若基于向量时钟判定 $O_7 \rightarrow O_4$ 成立, 则意味着该这两个操作必然不是一个真实的 UAF 漏洞. 以上均无须再通过 Petri 网反向展开的方法验证其真伪. 综上所述, 当给定一个多线程程序运行轨迹时, 首先可根据本节的向量时钟计算各个操作执行前后的局部向量时钟. 之后, 对于每个 UAF 漏洞对应的 *Use* 和 *Free* 操作对, 若能基于向量时钟判定 $O_u \rightarrow O_f$ 成立, 则说明该漏洞是个误报; 若能基于向量时钟判定 $O_f \rightarrow O_u$ 成立, 则说明该漏洞真实可触发. 上述两种情况下, 都无须再通过 Petri 网反向展开进行验证. 仅当 *Use* 和 *Free* 操作不满足这两种因果关系时, 方需要借助 Petri 网反向展开的方法对这个潜在 UAF 漏洞的真实性进行分析, 这无疑可以大大提高 UAF 漏洞检测的效率.

此外, 第 2.3 节为保证数据的一致性添加数据一致性控制结构时, 图 3 中的变迁 w_1^c 与 w_2^c 分别表示 $w_1(v,mid,x)$, $w_2(t,mid,x)$之外的任何一个对内存空间 mid 的写操作. 这是为了防止这些操作发生在 W_1, W_2 与变量的读操作 $r(u,mid,x)$之间. 实际上, 部分对内存空间 mid 的写操作可能只在 $r(u,mid,x)$后发生, 或者只在 W_1, W_2 前发生, 它们是不可能发生在 W_1, W_2 与读操作 $r(u,mid,x)$之间的. 此时, 若通过向量时钟的方法识别上述因果关系, 则各个 UAF 漏洞的伴生 Petri 网模型将大大简化, 从而进一步提高 UAF 漏洞检测的效率.

5 实验评估

本节结合多个程序实例, 分别给出用 UFO 方法、AddressSanitizer[15]工具、Valgrind[31]工具和本文方法进行 UAF 漏洞检测的结果与性能指标, 进而对上述方法和工具的优缺点进行对比和分析.

实验所用的程序实例包括 4 个实用的工具软件(一个并行的文件压缩软件 Pbzip2、一个多线程的 BT 下载

工具 Transmission、一个网站克隆工具 HTTrack、一个大型文件压缩软件 Lrzip), 8 个来自文献[32]的包含 UAF 漏洞程序样本(表 6 中 CWE 开头的程序样本), 另有 5 个自行设计的多线程程序实例. 其中, 前 3 个工具软件来自文献[33], Lrzip 来自文献[11], 程序代码最大规模达到 88 935 行. 上述程序的运行轨迹以及自行设计的 5 个多线程程序实例可通过链接 https://pan.baidu.com/s/1-uK6f1WtKhVimon_khaS_g 下载(提取码: wsua).

表 6 UAF 漏洞检测实验结果与对比分析

多线程程序样本	生成程序轨迹时配置的线程数量	轨迹中隐含的潜在 UAF 漏洞数	报告的 UAF 漏洞数		漏报分析	
			UFO 方法	本文方法	UFO 方法	本文方法
Pbzip2	3	90	1	42	UFO 方法因如下 3 类原因会导致漏报: (1) UFO 方法不检测同一线程内的 UAF 漏洞; (2) 若分析的轨迹中存在内存重复释放错误, 则 UFO 漏洞检测程序会提前终止; (3) 当程序运行轨迹较长时, UFO 方法会忽略轨迹中的部分事件以保证检测的效率	无法像 UFO 方法结合程序分支条件信息对程序轨迹进行派生,由此会产生部分漏报
	4	94	37	42		
	8	104	0	42		
	9	109	0	42		
	10	114	0	42		
Transmission	2	5 717	2	21		
	2	5 717	2	21		
	2	5 717	2	21		
	2	5 717	2	21		
	2	5 717	2	21		
HTTrack	2	1 005	0	541		
Lrzip	4	5 985	3	81		
CWE416-1	2	1	0	1		
CWE416-2	2	1	0	1		
CWE416-3	2	1	0	1		
CWE416-4	2	1	0	1		
CWE416-5	2	1	0	1		
CWE416-6	2	1	0	1		
CWE416-7	2	1	0	1		
CWE416-8	2	1	0	1		
UAF_test1	2	2	1	2		
UAF_test2	2	3	3	1		
UAF_test3	2	2	2	1		
UAF_test4	3	4	2	3		
UAF_test5	2	2	2	2		

表 6 UAF 漏洞检测实验结果与对比分析(续)

多线程程序样本	生成程序轨迹时配置的线程数量	轨迹中隐含的潜在 UAF 漏洞数	误报分析		时间开销(ms)			性能对比分析
			UFO 方法	本文方法	UFO 方法	本文方法		
						无向量时钟优化	有向量时钟优化	
Pbzip2	3	90	无误报	无误报	2 149	110 791	33 703	当 UFO 方法不对同一线程内潜在的 UAF 漏洞进行检测, 或者在轨迹较长时舍弃部分轨迹前缀提升检测效率时, UFO 方法耗时更小, 否则, 本文方法占优
	4	94			23 616	109 611	**33 180**	
	8	104			4 068	109 011	34 689	
	9	109			2 473	108 587	32 681	
	10	114			4 061	108 455	34 230	
Transmission	2	5 717			7 249	925 601	**2 789**	当存在大量潜在 UAF 漏洞时, 本文方法不使用向量时钟优化会耗时多, 使用向量时钟优化耗时更少
	2	5 717			10 767	793 396	**2 777**	
	2	5 717			10 373	735 728	**3 124**	
	2	5 717			8 375	894 659	**2 984**	
	2	5 717			11 265	865 519	**3 084**	
HTTrack	2	1 005			1 811 823	**744 767s**	**48 767**	当潜在 UAF 漏洞不是特别多且 UFO 检测不提前终止和舍弃轨迹前缀时, 本文方法无论是否使用向量时钟均具有更好的性能
Lrzip	4	5 985			1 190 051	**233 415**	**3 159**	
CWE416-1	2	1			154	**23**	**6**	
CWE416-2	2	1			150	**14**	**8**	
CWE416-3	2	1			142	**27**	**8**	
CWE416-4	2	1			152	**8**	**8**	
CWE416-5	2	1			150	**9**	**7**	
CWE416-6	2	1			144	**13**	**7**	
CWE416-7	2	1			148	**13**	**8**	
CWE416-8	2	1			154	**18**	**6**	
UAF_test1	2	2			503	**21**	**26**	
UAF_test2	2	3			586	**354**	**282**	
UAF_test3	2	2			556	**315**	**272**	
UAF_test4	3	4			582	**366**	**341**	
UAF_test5	2	2			549	**42**	**26**	

实验过程中采用的机器配置如下: CPU 为 Intel(R) Core(TM) i5-6200U CPU @ 2.30 GHz, 内存为 9.6 GB, 操作系统为 Ubuntu 18.04. 运行各程序样本生成运行轨迹时配置的线程数量、潜在 UAF 漏洞数量(针对同一内存空间的任意两个 *Free*/*Use* 操作对都看作一个潜在 UAF 漏洞)、UFO 方法与本文方法检测出的 UAF 漏洞数量以及漏报率和时间性能统计分析结果见表 6.

根据实验结果, 下面从 UAF 漏洞的检测结果、漏报和误报分析以及时间开销等方面进行分析.

首先, 本文方法与 UFO 方法类似, 能够确保每一个检测出的 UAF 漏洞都是可复现的, 因为每一个真实的 UAF 漏洞都对应一条能触发该漏洞的路径. 从这个角度看, 两种方法均不存在误报.

其次, 就 UAF 漏洞的检测结果, 针对程序实例 Pbzip2、Transmission、Lrzip、UAF_test1、UAF_test4 以及 CWE416-1 等而言, UFO 方法报告的 UAF 漏洞数量少于本文检测到的漏洞数量. 经分析得知: UFO 方法不会对同一线程内的潜在 UAF 漏洞进行检测(即 *Free* 操作与 *Use* 操作同属于一个线程的情况, 虽然此类 UAF 漏洞是可真实存在的); 而且, 当程序运行轨迹过长时, UFO 方法会舍弃一部分轨迹前缀. 这两种情况都会导致部分漏报. 另外, UFO 方法在检测 HTTrack 时, 会因被检测程序中存在的内存重复释放问题而导致检测过程提前终止, 从而导致 UFO 方法报告的 UAF 漏洞数量为 0. 本文方法不存在上述问题, 然而本文方法也存在一定漏报, 因本文方法无法像 UFO 方法那样结合程序分支的条件信息对程序轨迹进行更广泛的派生.

就 UAF 漏洞检测的时间开销而言, 在不考虑 UFO 方法因被测程序存在内存重复释放错误而提前终止检测或者舍弃部分轨迹前缀的情况下, 本文方法总体上优于 UFO 方法. 一方面, 这是因为本文借助向量时钟的方法提前排除了很多潜在 UAF 漏洞; 另一方面, 这是因为本文基于 Petri 网的反向展开进行目标导向的 UAF 漏洞检测, 而 UFO 方法的约束求解在一定程度上是盲目的.

即使本文不使用向量时钟的方法进行性能优化, 在需要验证的潜在 UAF 漏洞数量不是很多的情况下, 第 3 节所给的、单独基于伴生 Petri 网反向展开的 UAF 漏洞检测算法相比 UFO 方法仍有一定的性能优势, 这印证了目标导向的 UAF 漏洞检测对传统 UAF 预测性检测算法性能的提升. 不过, 当需要验证的潜在 UAF 漏洞较多时(例如 BT 下载工具 Transmission), 需要对每个潜在漏洞对应的伴生 Petri 网进行反向展开, 这其中的部分分析是冗余和重复的, 此时会导致较低的检测性能. 后续将针对这一问题研究具体的改进方法. 实际当中, 本文使用向量时钟对潜在的、需要检测的 UAF 漏洞进行筛选后, 本文方法的效率还是有保证的.

另外, 本文结合上述程序实例, 同时与 AddressSanitizer 和 Valgrind 两个 UAF 检测工具进行实验对比. 结果如表 7 所示: AddressSanitizer 只能检测到本文自行设计的 5 个多线程程序实例(UAF_test 等)中的部分 UAF 漏洞, 而其他 4 个实用工具及 CWE 开头的 7 个程序样本均没有报告 UAF 漏洞. 经分析得知, AddressSanitizer 只能判定输入的程序运行轨迹是否存在 UAF 漏洞, 无法根据给定轨迹做更多潜在程序运行轨迹及潜在漏洞的预测, 从而会存在较多的漏报. 此外, 以 CWE 开头的程序样本中全部包含内存泄漏(内存泄漏(memory leak)是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放, 造成系统内存的浪费, 导致程序运行速度减慢甚至系统崩溃等严重后果. 比如, 当程序中的 *malloc* 和 *free*、*new* 和 *delete* 不成对出现时, 就判定发生内存泄露), AddressSanitizer 在报告内存泄漏问题后会终止程序不再检测, 因此, 以 CWE 开头的程序并不报告 UAF 漏洞. 与 AddressSanitizer 相似, Valgrind 也只能检测到本文自行设计的 5 个程序实例中的部分 UAF 漏洞, 4 个实用工具均没有报告 UAF 漏洞. 同样因为 Valgrind 只能判定输入的程序运行轨迹中是否存在 UAF 漏洞, 而无法根据现有轨迹对更多潜在轨迹及潜在漏洞进行预测, 因此也会存在较多漏报. 与 AddressSanitizer 不同的是, Valgrind 能够检测到以 CWE 开头的 7 个程序样本中的部分 UAF 漏洞, 这是因为 Valgrind 是在程序退出后再将所有的漏洞信息(包括内存泄漏和 UAF 漏洞)报告出来, 不会提前终止程序. 最后, 当同一块内存空间被多次重复释放和分配时, AddressSanitizer 和 Valgrind 都会存在一些漏报. 这是因为当同一内存空间被释放又被重新分配后, 若再次使用指向已被释放内存空间的指针, 此时该内存空间已被重新分配, AddressSanitizer 和 Valgrind 均会报告该内存空间可用, 从而造成漏报.

表 7 UAF 漏洞检测实验结果与对比分析

多线程程序样本	生成程序轨迹时配置的线程数量	轨迹中隐含的潜在 UAF 漏洞数	报告的 UAF 漏洞数		
			AddressSanitizer	Valgrind	本文方法
Pbzip2	3	90	0	0	42
Transmission	2	5 717	0	0	21
HTTrack	2	1 005	0	0	541
Lrzip	4	5 985	0	0	81
CWE416-1	2	1	0	1	1
CWE416-2	2	1	0	1	1
CWE416-3	2	1	0	1	1
CWE416-4	2	1	0	1	1
CWE416-5	2	1	0	1	1
CWE416-6	2	1	0	1	1
CWE416-7	2	1	1	1	1
CWE416-8	2	1	0	1	1
UAF_test1	2	2	2	2	2
UAF_test2	2	3	2	2	1
UAF_test3	2	2	2	2	1
UAF_test4	3	4	2	2	3
UAF_test5	2	2	2	2	2

6 总结与展望

为了提高多线程程序 UAF 漏洞检测的效率, 本文提出了一种目标导向的多线程程序 UAF 漏洞检测方法. 首先, 根据程序的运行轨迹构建程序控制流结构的 Petri 网模型; 其次, 针对每一个潜在 UAF 漏洞, 以触发该漏洞为目标导向, 通过在程序 Petri 网模型中添加数据一致性控制结构和 UAF 漏洞相关的因果约束, 构造了 UAF 漏洞的伴生 Petri 网; 最后, 在 UAF 伴生 Petri 网的基础上, 基于 Petri 网的反向展开算法给出了一种新的 UAF 漏洞检测方法. 与此同时, 为了进一步提高 UAF 漏洞检测效率, 文中提出了一种基于新型向量时钟进行操作间因果关系判定的方法, 据此对潜在的UAF漏洞进行筛选. 实验结果验证了本文方法的可行性和有效性.

然而, 本文方法尚诸多不足.

- 一方面, 本文基于程序的单一运行轨迹检测程序的 UAF 漏洞, 而程序的一条轨迹只能揭示程序的部分行为信息, 因此会导致 UAF 漏洞的漏报. 后续可以考虑将静态检测与动态检测相结合, 通过符号执行方法推导出多条潜在的程序运行估计, 再基于本文方法进行漏洞检测, 据此降低程序漏报率.
- 另一方面, 第 4 节给出的基于向量时钟的操作间因果关系识别仅处理了操作间的线程内因果依赖、线程间 *start* 和 *join* 操作导致的因果依赖, 而如文献[34]所述, 多线程程序的操作之间还可能存在复杂的锁-*start* 耦合因果依赖等关系, 后续将进一步完善本文的向量时钟计算规则, 以更准确地进行因果关系识别.
- 再者, 对于重复释放和分配同一内存地址所造成的 UAF 漏洞, 本文方法尚不能解决, 后续将进一步完善本文方法, 以解决此类问题.
- 最后, 漏洞检测只是保证软件质量的一个步骤, 检测到漏洞后的程序自动修复[35]也值得深入探究.

References:

[1] Robey R, Zamora Y, Wrote; Yin HY, Trans. Parallel and High Performance Computing. Beijing: Tsinghua University Press, 2022 (in Chinese).

[2] Su XH, Yu Z, Wang TT, *et al*. A survey on exposing, detecting and avoiding concurrency bugs. Chinese Journal of Computers, 2015, 395(11): 93−111 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02215]

[3] Zhang J, Zhang C, Xuan JF, *et al*. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[4] Caballero J, Grieco G, Marron M, *et al*. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proc. of the Int'l Symp. on Software Testing and Analysis. 2012. 133−143.

[5] Ayewah N, Hovemeyer D, Morgenthaler J, *et al*. Using static analysis to find bugs. IEEE Software, 2008, 25(5): 22−29.

[6] Cadar C, Sen K, *et al*. Symbolic execution for software testing: Three decades later. Communications of the ACM, 2013, 56(2): 82−90.

[7] Feist J, Mounier L, Potet ML. Statically detecting use after free on binary code. Journal of Computer Virology & Hacking Techniques, 2014, 10(3): 211−217.

[8] Nguyen MD, Bardin S, Bonichon R, *et al*. Binary-level directed fuzzing for use-after-free vulnerabilities. In: Proc. of the 23rd Int'l Symp. on Research in Attacks, Intrusions and Defenses (RAID 2020). 2020. 47−62.

[9] Dewey D, Reaves B, Traynor P. Uncovering use-after-free conditions in compiled code. In: Proc. of the Int'l Conf. on Availability. IEEE Computer Society, 2015. 90−99.

[10] Zhu KL, Lu YL, Huang H. Scalable static detection of use-after-free vulnerabilities in binary code. IEEE Access, 2020, 8: 78713−78725. [doi: 10.1109/ACCESS.2020.2990197]

[11] Shi Q, Xiao X, Wu R, *et al*. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. ACM SIGPLAN Notices, 2018, 53(4): 693−706.

[12] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. In: Proc. of the Winter USENIX Conf. 1991. 125−136.

[13] Lee B, Song C, Jang Y, *et al*. Preventing use-after-free with dangling pointers nullification. In: Proc. of the Network & Distributed System Security Symp. Internet Society, 2015. [doi:10.14722/ndss.2015.23238]

[14] Liu T, Curtsinger C, Berger ED. DoubleTake: Fast and precise error detection via evidence-based dynamic analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering. 2016. 911−922.

[15] Serebryany K, Bruening D, Potapenko A, *et al*. AddressSanitizer: A fast address sanity checker. In: Proc. of the Usenix Conf. on Technical Conf. 2012. 309−318.

[16] The kernel address sanitizer. 2022. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[17] Younan Y. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In: Proc. of the Network & Distributed System Security Symp. 2015. [doi:10.14722/NDSS.2015.23190]

[18] Cai Y, Yun H, Wang J, *et al*. Sound and efficient concurrency bug prediction. In: Proc. of the 29th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE 2021). 2021. 255−267.

[19] Huang J. UFO: Predictive concurrency use-after-free detection. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). 2018. 609−619.

[20] Hao ZY, Lu FM. Reverse unfolding of Petri nets and its application in program data race detection. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1612−1630 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6240.htm [doi: 10.13328/j.cnki.jos.006240]

[21] Mcmillan KL. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Probst DK, von Bochmann G, eds. Proc. of the Int'l Workshop on Computer Aided Verification (CAV'92). LNCS 663, Berlin, Heidelberg: Springer, 1992. 164−177.

[22] Yuan CY. Petri Net Applications. Beijing: Science Press, 2013 (in Chinese).

[23] Pang SC, Jiang CJ, Sun P, *et al*. Property analysis of shared composition Petri nets. Acta Automatica Sinica, 2004, 30(6): 944−948 (in Chinese with English abstract).

[24] Esparza J, Römer S, Vogler W. An improvement of McMillan's unfolding algorithm. Formal Methods in System Design, 2002, 20: 285−310 [doi:10.1023/A:1014746130920]

[25] Moura LD, Bjørner, N. Z3: An Efficient SMT Solver. In: Ramakrishnan CR, Rehof J, eds. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). LNCS Vol.4963, 2008. 337−340. [doi:10.1007/978-3-540-78800-3_24]

[26] Liu GJ. Petri Nets: Theoretical Models and Analysis Methods for Concurrent Systems. Springer, 2022. [doi:10.1007/978-981-19-6309-4]

[27] Xiang D, Liu G, Yan C, *et al*. Detecting data inconsistency based on the unfolding technique of Petri nets. IEEE Trans. on Industrial Informatics, 2017, 13(6): 2995−3005.

[28] Lu FM, Huang Y, Zeng QT, *et al*. Data race detection and replay of multi-threaded program based on Petri net unfolding. Ruan Jian Xue Bao/Journal of Software, online published 2022-01-28 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6618.htm [doi: 10.13328/j.cnki.jos.006618]

[29] Şerbănuţă TF, Chen F, Roşu G. Maximal causal models for sequentially consistent systems. In: Qadeer S, Tasiran S, eds. Runtime Verification. LNCS Vol.7687, Berlin, Heidelberg: Springer, 2023. 136−150. [doi:10.1007/978-3-642-35632-2_16]

[30] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558−565.

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

[32] Gui B, Song W, Huang J. UAFSan: An object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2021). ACM, 2021. 309−321.

[33] Jie Y, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. In: Proc. of the 36th Int'l Symp. on Computer Architecture (ISCA 2009). 2009. 325−336.

[34] Lu FM, Zheng JJ, Bao YX, *et al*. Deadlock detection of multithreaded programs based on lock-augmented segmentation graph. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1682−1700 (in Chinese with English abstract). http://www.jos. org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]

[35] Bayley I, Cai Y, Machado P. Special section on testing and repair for software engineering technologies and applications. Software Quality Journal, 2020, 28(2): 821−822.

附中文参考文献:

[1] Robey R, Zamora Y, 著; 殷海英, 译. 并行计算与高性能计算. 北京: 清华大学出版社, 2022.

[2] 苏小红, 禹振, 王甜甜, 等. 并发缺陷暴露、检测与规避研究综述. 计算机学报, 2015, 395(11): 93−111. [doi: 10.11897/SP.J.1016.2015.02215]

[3] 张健, 张超, 玄跻峰, 等. 程序分析研究进展. 软件学报, 2019, 30(1): 80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[20] 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. 软件学报, 2021, 32(6): 1612−1630. http://www.jos.org.cn/1000-9825/6240.htm [doi: 10.13328/j.cnki.jos.006240]

[22] 袁崇义. Petri 网应用. 北京: 科学出版社, 2013.

[23] 庞善臣, 蒋昌俊, 孙萍, 等. 共享合成 Petri 网的性质分析. 自动化学报, 2004, 30(6): 944−948.

[28] 鲁法明, 黄莹, 曾庆田, 等. 基于 Petri 网展开的多线程程序数据竞争检测与重演. 软件学报, 2022-01-28 在线出版. http://www.jos.org.cn/1000-9825/6618.htm [doi: 10.13328/j.cnki.jos.006618]

[34] 鲁法明, 郑佳静, 包云霞, 等. 基于锁增广分段图的多线程程序死锁检测. 软件学报, 2021, 32(6): 1682−1700. http://www.jos.org.cn/1000-9825/6244.htm [doi: 10.13328/j.cnki.jos.006244]

鲁法明(1981－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为 Petri 网, 并行程序验证, 过程挖掘.

唐梦凡(1998－), 女, 硕士, 主要研究领域为并行程序验证.

包云霞(1979－), 女, 副教授, 主要研究领域为 Petri 网, 并行程序分析与验证.

曾庆田(1973－), 男, 博士, 教授, CCF 高级会员, 主要研究领域为 Petri 网, 人工智能.

李彦成(1989－), 男, 高级工程师, 主要研究领域为软件质量保证.