软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):1961−1977 [doi: 10.13328/j.cnki.jos.006563]

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

反例引导的 C 代码空间流模型检测方法*

于银菠, 刘家佳, 慕德俊

(西北工业大学 网络空间安全学院, 陕西 西安 710072)

通信作者: 刘家佳, E-mail: liujiajia@nwpu.edu.cn

摘　要: 软件验证一直是确保软件正确性和安全性的热点研究问题. 然而, 由于程序语言复杂的语法语义特性, 应用形式化方法验证程序的正确性存在准确度低和效率差的问题. 其中, 由指针操作带来的地址空间的状态变化使得现有模型检测方法的检测准确度难以得到保证. 为此, 通过结合模型检测与稀疏值流分析方法, 设计了一种空间流模型, 实现了对 C 程序在符号变量层面和地址空间层面的状态行为的有效描述, 并提出了一种反例引导的抽象细化和稀疏值流强更新算法(CEGAS), 实现了 C 程序指向信息敏感的形式化验证. 建立了包含多种指针操作的C代码基准库, 并基于该基准库进行了对比实验. 实验结果表明: 所提出的模型检测算法CEGAS在分析含有多种 C 代码特性的任务中, 与现有模型检测工具相比均能取得突出的结果, 其检测准确度为 92.9%, 每行代码的平均检测时间为 2.58 ms, 优于现有检测工具.

关键词: 软件验证; 模型检测; 稀疏值流分析; 指针分析; 漏洞检测

中图法分类号: TP311

中文引用格式: 于银菠, 刘家佳, 慕德俊. 反例引导的 C 代码空间流模型检测方法. 软件学报, 2022, 33(6): 1961–1977. http://www.jos.org.cn/1000-9825/6563.htm

英文引用格式: Yu YB, Liu JJ, Mu DJ. Counterexample-guided Spatial Flow Model Checking Methods for C Codes. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 1961−1977 (in Chinese). http://www.jos.org.cn/1000-9825/6563.htm

Counterexample-guided Spatial Flow Model Checking Methods for C Codes

YU Yin-Bo, LIU Jia-Jia, MU De-Jun

(School of Cybersecurity, Northwestern Polytechnical University, Xi'an 710072, China)

Abstract: Software verification has always been a hot research topic to ensure the correctness and security of software. However, due to the complex semantics and syntax of programming language, applying formal methods to verify the correctness of programs has the problems of low accuracy and low efficiency. Among them, the state change of address space caused by pointer operations makes the verification accuracy of existing model checking methods difficult to be guaranteed. By combining model checking and sparse value-flow analysis, this study designs a spatial flow model to effectively describe the state behavior of C codes at the symbolic variable level and address space level, and proposes a model checking algorithm of counterexample-guided abstraction refinement and sparse value flow strong update (CEGAS), which enables points-to-sensitive formal verification for C codes. This study establishes a C code benchmark containing a variety of pointer operationsand conducts comparative experiments based on this benchmark. These experiments indicate that in the task of analyzing multi-class C code features, the model checking algorithm CEGAS proposed in this study can achieve outstanding results compared with the existing model detection tools. The verification accuracy of CEGAS is 92.9%, and the average verification time of each line of code is 2.58 ms, which are both better than existing testing tools.

Key words: software verification; model checking; spare value flow analysis; pointer analysis; vulnerability detection

* 基金项目: 广东省基础与应用基础研究基金(2021A1515110279); 太仓市基础研究计划(TC2020JC03); 中央高校基本科研业务费专项资金(D5000210588)

本文由"系统软件安全"专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.

收稿时间: 2021-08-29; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

确保软件的正确性是当今软件系统中最为关键且十分艰巨的任务, 它是保障软件系统内生安全的核心因素之一. 手动地检测软件易错, 且开销巨大. 为此, 诸多包括软件分析、测试等方法被提了出来, 用以提升软件的正确性和安全性. 其中, 软件程序的形式化验证一直是软件安全领域中的一个热点问题, 它通过数学方法为软件程序语义逻辑上的正确性提供证明, 从而为软件安全提供了核心保障. 目前, 软件验证主要包含抽象静态分析和模型检测两类方法[1], 前者通过设计抽象域来优化程序分析空间, 后者借助结构化模型来实现一个系统是否满足安全属性或活属性的验证. 现有大多数验证工具结合了这两类方法, 以确保验证的可扩展性和准确度, 如反例引导的抽象细化[2]、符号执行[3]、归纳证明[4]等. 目前, 软件验证已被广泛应用在验证应用软件[5]、通信与安全协议[6,7]、设备驱动与固件、系统内核等一系列软件工程领域, 成为了保障软件安全的主要方法.

通常, 形式化验证首先对程序进行形式化建模并确定检测属性, 之后通过特定策略遍历模型的状态空间, 在理论上暴露出程序所存在的违法检测属性的路径[4]. 然而, 由于程序语言本身具备复杂的语法语义表达能力, 如何准确地对软件行为逻辑进行建模, 确保模型与程序语义的等价性, 一直是软件形式化的关键挑战[1,8]. 为了对程序代码的逻辑正确性进行验证, 现有最直接的方法是人工地将程序代码翻译为形式化模型, 之后进行形式化验证. 这类方法一方面人工成本太大, 同时也无法确保人工翻译出的模型的准确性. 另一种解决方法是设计针对高级编程语言(如 Python, Java 等)的检测算法, 这类语言的抽象程度较高, 易于自动地翻译成形式化验证所需要的模型, 避免了人工翻译带来的错误. 通过一定方法, 可以将其他语言编写的代码转译为这些语言, 进而可以进行形式化验证. 然而, 一些较为底层的编程语言(如 C/C++)具备更为复杂的语义特性, 难以用高级语言来描述. 我们注意到, 现有针对 C 这类语言实现的程序验证工具(如 CPAChecker[9]、SPIN[10]、Gazer[11]等), 大多数是针对程序变量符号层面的状态变化而设计的, 通过基于抽象[12]、插值[13]、反例引导的模型细化[2]等先进技术, 可以实现高效率的软件正确性验证. 然而, 在程序包含指针等内存操作或复杂的数据结构体的情况下, 这些工具将无法或错误地感知到由于内存空间地址上的状态变化而导致的变量符号层面的状态变化, 从而产生虚假的检测结果, 无法保障验证的有效性.

本文以 C 语言程序的形式化验证为研究对象, 旨在解决现有方法由于指针操作带来的地址层面的状态空间变化而无法准确验证的问题. 指针的使用可以使得程序更加简洁, 但在程序动态执行时, 指针也会给程序带来复杂的行为, 不恰当地使用指针成为了代码存在缺陷的主要原因之一. 因此, 程序的静态分析需要进行指针分析, 明确指针变量在不同位置所指向的对象. 然而, 由于指针分析通常需要高开销的数据流分析, 随着程序规模的扩大, 指针分析存在着分析精度和效率难以权衡的挑战. 这也是导致现有大多数模型检测算法不支持指针语法的主要原因. 为了提高模型检测的精度, 现有方法通常利用独立的指针分析算法预先获取程序指针的指向信息, 用以辅助模型检测的进行. 但是, 高精度的指针分析(如流敏感、上下文敏感)计算开销较大, 且可能存在诸多不必要的计算开销, 无法确保分析的可扩展性. 而不敏感的指针分析(流和上下文都不敏感, 如 Anderson[14])虽然可以确保可扩展性, 但其精度则无法保障.

为了解决上述问题, 本文采用先抽象后验证的思路, 结合基于抽象的模型检测算法和稀疏值流分析, 设计了一种反例引导的空间流模型检测算法: 通过指向关系不敏感的方式, 该算法在线性时间内为 C 程序构建出一种融合了其控制流信息和稀疏值流信息的形式化模型, 本文称其为空间流模型; 然后, 利用值抽象的方式抽象空间流模型, 并在抽象模型中检测是否反例, 若存在且该反例在原先模型无效, 则利用该反例, 细化由于变量抽象或者不敏感指向关系导致的错误信息, 以此逐步更新细化模型信息实现程序符号层面和地址空间层面的分析. 通过反例信息来引导模型检测的进行, 可以有效地实现检测精度和效率的权衡.

本文的主要贡献如下:

1) 基于稀疏值流图, 设计了一种我们称之为稀疏空间流的 C 代码模型及其构建方法, 能够自动、高效地描述 C 程序在符号层面和地址空间层面的状态变化过程;
2) 结合形式化方法中的显性状态分析和软件分析中的稀疏值分析方法, 提出了一种反例引导的稀疏空间流模型检测算法, 利用虚假反例引导符号层面和地址空间层面的状态空间分析, 确保验证的准

确性和高效性;

3) 建立了一个具备多种复杂语法语义特性的C程序基准库, 以此基准库, 验证了所设计的模型检测方法较现有方法具备更加准确的验证能力和更加快速的验证效率.

本文第1节介绍相关工作. 第2节概述本文研究的动机和所设计的解决方法. 第3节具体阐述本文所提出的反例引导的空间流模型检测算法. 第4节介绍所提方法与现有方法在检测准确度和效率上的实验对比结果. 最后总结全文, 并对未来值得关注的研究方向进行初步探讨.

1 相关工作

软件验证通常分为状态空间遍历和路径验证两部分问题. 目前, 主流的软件模型检测算法通常是将路径的形式化验证问题转化为公式满足性问题[15], 利用SMT解算器验证路径的可行性. 而根据状态空间的遍历策略, 现有方法通常可以分为显性状态模型检测、归纳证明、谓词抽象这3类方法[15]. 其中, 显性状态模型检测利用状态可达图显性地记录相应状态, 利用深度、广度或启发式算法尽可能地遍历状态空间中初始出发的所有可能路径, 如工具SPIN[10]、CMC[16]和JavaPathfinder[17]. 但是, 随着程序规模的扩大, 状态空间爆炸问题成为了模型检测最关键的挑战. 为此, 研究者通过对程序所有路径作展开的次数设置限定(即限界模型检测), 将限定界限内的路径编码为SMT公式来验证, 从而减缓了状态空间爆炸问题, 如SMACK[18]和CBMC[19]等. 限界模型检测可以实现程序逻辑缺陷的快速检测, 但是无法确保检测的完整性. 归纳证明扩展了限界模型检测的策略, 通过先验证部分状态后, 以此再递归地验证其他状态空间, 一般称为 k 归纳法(k-induction), 如ESBMC[20]. 谓词抽象则是通过设定抽象域, 形成过度近似的抽象模型, 从而缓解状态空间爆炸问题. 最为常用的策略是反例引导的抽象细化(counterexample-guided abstraction refinement, CEGAR)[2], 对抽象模型检测后, 若检测出的反例在原始模型是无效的, 则利用导致该反例无效的信息对抽象域进行使用插值细化, 从而去除虚假反例. CEGAR可以有效地权衡检测的效率和精度, 成为包括SLAM[21]、CPAChecker[9]、Theta[22]等工具在内的最为主要的模型检测策略. 考虑到程序语义语法中的复杂特性和代码规模的扩大, 现有的模型检测工具通常会利用多种状态空间遍历策略混合或利用启发式算法、强化学习等方法, 在确保验证效率的同时, 提高模型检测的全面性.

上述软件验证方法都是基于形式化方法, 大部分只能够分析符号变量上的状态变化, 无法直接支持指针等复杂的程序语义[8]. 现有方法, 如CPAChecker和SMACK, 预先利用指针分析算法计算出指针信息后, 再进行模型检测. 指针分析是程序静态分析的挑战之一, 其目标是计算出指针所指向的对象, 正确的指针信息是检测出软件漏洞(如空指针引用、内存泄漏、释放后重用等)的关键因素之一. 指针分析中存在包括路径、流、上下文、字段等多维度信息, 会影响其分析精度和开销的权衡. 通常, 静态分析会采用不敏感的指针分析, 如基于蕴含约束(又称为Anderson)或基于合并约束(又称为Steensgaard[23]). 这两种方法可以保障分析的可扩展性, 但是分析的精度无法得到保障. 为了计算出更为准确的指向关系, 如流敏感的, 指针分析需要迭代地求解数据流问题, 存在着巨大的计算开销. 为此, 研究者提出了一种稀疏值流分析的方法[24−26], 将原先需要在所有控制流节点上计算数据流的问题, 转移到预先计算出的更为稀疏近似的定义-使用链上, 有效地降低了计算开销, 从而可以应用到大规模的程序分析中. 文献[25]基于稀疏值分析, 为C和C++代码设计了一种需求驱动的指针分析, 通过Anderson算法计算出稀疏值流图后, 按照需要分析的指针, 分析其所在位置的指针信息. 文献[27]基于稀疏值流分析的框架, 通过设计精简局部的指针分析和利用SMT实现全局路径的可行性验证, 实现了百万行级别代码的释放后重用和空指针引用的缺陷检测. 文献[28]基于文献[27]的工作, 提出了一种基于切片的路径敏感分析, 能够为大规模的代码分析提供空指针引用和污点分析.

通过指针分析计算出指向信息后再进行模型检测, 这类方法可以确保检测的准确性. 但其前提是需要进行流、上下文等敏感的指针分析, 计算开销大, 限制了模型检测的效率. 另外, 指针分析的结果对模型检测可能会存在诸多计算冗余, 如与反例无关的指针变量的分析. 为此, 本文通过融合模型检测和稀疏值流分析的方式, 实现了更为高效的C程序模型检测方法.

2 方法概述

2.1 研究动机

现有的模型检测算法大多针对程序变量符号层面的状态变化而设计，在程序包含指针等内存操作或复杂的数据结构体的情况下，这些算法将无法或错误地感知由内存空间地址上的信息变化而导致的变量符号层面的状态变化，因而带来不必要的分析开销和虚假的分析结果.

为了更好地说明这一问题，本节以图 1(a)所示的一段 C 语言程序为例，阐明现有模型检测算法中存在的缺陷. 这里，我们假设该 C 语言代码的模型为$\mathcal{M}$. 需要检测的程序属性为$\varphi\equiv LTL(G!\ \text{call}(error(\cdot)))$，它表示该代码在任何时刻都不会调用函数 $error(\cdot)$，即执行到第 22 行. 模型检测要求验证模型$\mathcal{M}$所有的状态是否会违反φ，可以用公式$\mathcal{M}\models\varphi$表示. 为了求解此问题，如图 1(b)所示，现有的模型检测算法首先将代码转换为控制流图，即为$\mathcal{M}$. 之后，将模型转换为由多个命题(如θ_1,θ_2)组成的形式化公式. 由于现有算法大多都是符号层面的检测算法，无法检测 C 语言中的指针操作，只能够分析到符号变量和相应赋值操作(即 i=3 和 k=8)，因此，命题θ_1，$\neg\theta_2$，$\neg\theta_4$ 和θ_5 为真. 因此，最终检测出问题$\mathcal{M}\models\varphi$不成立，即存在违反属性$\varphi$的执行路径(即反例)：$\theta_1\wedge\neg\theta_2\wedge\neg\theta_4\wedge\theta_5$. 该反例表明：图 1(a)示例代码会按照该路径执行，并能够到达调用函数 $error(\cdot)$的位置(第 22 行). 然而实际上，代码由于存在指针操作，变量 i 和 k 会被指针变量间接赋值，所以该模型检测的验证过程并不全面，无法保证结果准确.

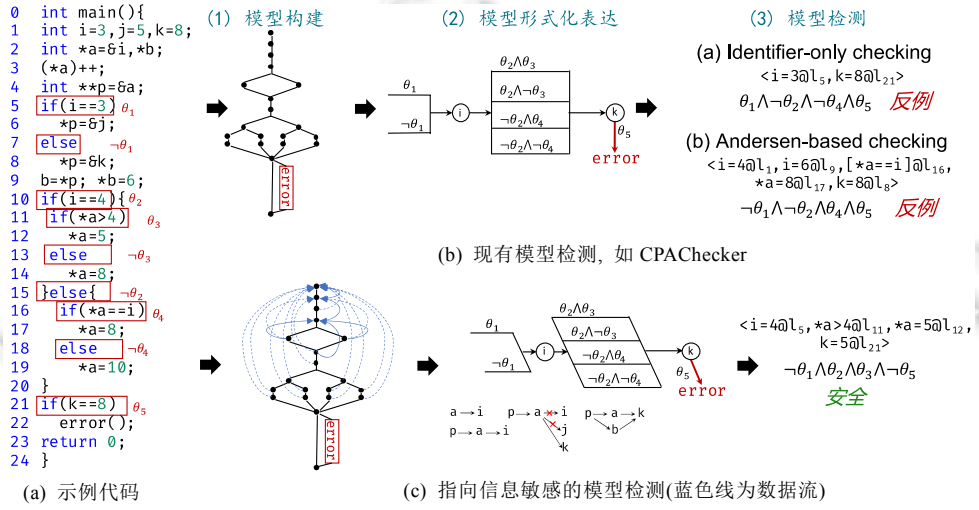

图 1 研究动机示例

为了提高验证的准确度，模型检测算法引入指针分析算法事先或在验证过程中获取指针指向关系，以纠正变量赋值的变化. 指针分析算法具有多种特性，如：

(1) 如果分析对控制流敏感，则是流敏感；否则是流不敏感；

(2) 如果对不同的函数调用上下文进行区分，则是上下文敏感；否则是上下文不敏感；

(3) 如果区分数据结构中的不同字段，则是字段敏感；否则是字段不敏感.

利用针对这些信息敏感的指针，分析可以计算出精确的指向关系. 然而，这种分析随着代码规模的扩大，计算复杂呈现指数增长，很难适用于稍大规模的模型. 因此，现有的算法通常采用信息不敏感的指针算法来减少计算资源和提高计算速度，使其适用于大规模的代码检测任务. 如 Andersen 算法是一种流和上下文都不敏感的指针分析算法，有着线性的计算复杂度，适用于大规模指针分析. 然而，由于算法对流和上下文信息都不敏感，导致计算出的指向关系是过度近似的. 如图 1(b)中基于 Anderson 的模型检测算法在验证模型的同时，

计算指针指向关系, 在代码第 2 行可得到指针变量 a 指向变量 i(即 $a\mapsto i$). 因此在第 3 行, 计算出 i=4, $\neg\theta_1$=true. 由于 Anderson 算法计算出的指向关系是流不敏感的, 算法在第 8 行可计算出指向关系: $p\mapsto\{a\}\mapsto\{i,k\}$. 由此类推, 模型检测最终计算出反例$\neg\theta_1\wedge\neg\theta_2\wedge\theta_4\wedge\theta_5$.

通过实际分析图 1(a)示例代码, 我们可以发现上述两个反例都是不成立的. 由于指针 b 的作用, 使得指针 a 在第 11 行处指向的地址空间(即变量 k 的地址)上的值为 6, 所有θ_3为真命题. 由此可得, 变量 k 在程序第 21 行的实际值为 5, θ_5 为假命题, 程序无法运行到第 22 行. 如图 1(b)所示, 一个准确的模型检测算法应当能够同时感知和检测符号变量和指针代表的内存地址空间上的状态转换过程(即指向信息敏感的模型检测算法), 计算出代码可达路径$\neg\theta_1\wedge\theta_2\wedge\theta_3\wedge\neg\theta_5$, 即证明该代码不会调用函数 $error(\cdot)$.

2.2 研究方法

为了解决上述问题, 在基于反例引导的抽象精化框架的基础上, 本文为 C 程序设计了一种指向信息敏感的模型检测算法, 即反例引导的抽象细化和值流关系强更新算法. 我们称该算法为 CEGAS (counterexample-guided abstraction refinement and strong update), 其框架如图 2 所示. 本节将从建模和验证两个方面, 简要地概述 CEGAS 在确保检测效率和准确度的情况下如何实现指向信息敏感的模型检测.

1) 建模: 现有方法使用 C 程序的控制流信息来执行模型检测. 然而, 控制流信息缺少直接的指针指向关系的信息, 导致不准确的分析. 为解决该问题, 我们提出了一种空间流图(sptial flow graph, SFG)来形式化地刻画 C 程序的逻辑行为, 其中, 我们在保留控制流信息的同时, 采用稀疏值流图来描述指针指向关系. SFG 可以使得 C 程序的形式化模型更加简洁、高效, 并为系统的模型检测提供充分的信息. 如图 2 所示, 构建 SFG 的模块是建立在 LLVM 编译器上, 并在由 C 程序转换而来的定义良好的 LLVM IR 语言上执行;

2) 验证: 为了提高模型检测的准确性, 现有方法是通过独立的数据流分析来为模型检查提供指针指向信息. 然而, 利用数据流分析来获得准确的指向信息的计算成本很高[24,27], 而不准确的指向分析(流/上下文不敏感)则不能保证模型检测的准确性. 为此, 我们通过结合模型检测和稀疏值分析方法, 设计了面向 SFG 的模型检测算法. 具体来说, 我们首先进行一个不敏感但快速(线性)的指向分析, 以获得超近似的指向信息, 并通过显式状态抽象[4,29]获得粗粒度的抽象模型; 然后, 遵循反例引导的模型精化, 即利用虚假反例引导抽象精度的细化和确定指向关系的计算. 通过这种基于 SFG 的反例引导方法, 模型检测的准确性和效率可以同时得到保证.

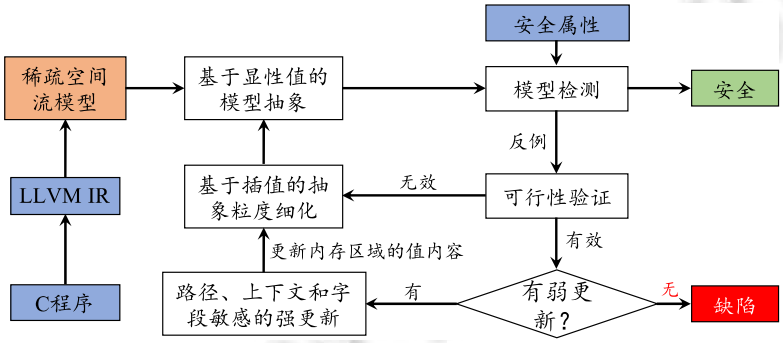

图 2 CEGAS 框架

3 反例引导的空间流模型检测

程序的形式化验证由程序的形式化描述和形式化检测两部分组成. 本节将重点从这两个角度出发, 介绍本文所提出的反例引导的空间流模型检测算法.

3.1 空间流模型定义与构建方法

为了感知程序在符号层面和内存空间上的状态转换信息, 我们设计了一种称为空间流图的高效形式化模型来描述 C 程序中代码的执行行为.

定义 1. 一个空间流图由两个子图组成: $\mathcal{M}=\{G_c,G_s\}$, 分别代表了控制流自动机(control flow automation, CFA)和稀疏值流图(sparse value flow graph, SVFG).

- $G_c=(L,l_0,E)$, 其中, L 是程序位置集合, 代表程序计数器; $l_0\in L$ 为初始程序位置, 即程序的入口位置; $E\subseteq L\times S\times L$ 为控制流边集合, 代表了从一个程序位置迁移到另一个位置中所执行的操作, S 为程序中所有操作语句的集合, 其中所有程序变量的集合为 V;
- $G_s=(N,\mathcal{E})$, 其中, $N\subseteq L\times S$ 代表指针变量的定义节点(def)或者使用节点(use)的集合; $\mathcal{E}\subseteq N\times V\times N$ 代表指针变量所有可能定义-使用链(def-use chain)的集合, 又称为值流边集合. 例如, 一个边 $n_1\xrightarrow{v}n_2$ 表示指针变量 $v\in V$ 由节点 n_1 处被定义, 并于 n_2 处被使用的 def-use 链, 而不是 n_1 和 n_2 之间所有的程序点.

SFG 利用控制流自动机可以准确地描述程序中的状态转移过程. 考虑到软件中复杂的内存地址信息的变化情况(即指针变量的指向关系, 本文使用公式 $p\mapsto o$ 表示指针变量 p 指向对象 o 的内存地址), SFG 进一步采用了稀疏值流图[30]的方式来描述内存地址状态的变化过程. 稀疏值流图只关注指针变量可能的 def-use 链, 而不需要将指针指向关系传播到控制流路径的所有程序点上. 这种稀疏性, 使其能够用于精确地分析大规模程序在内存空间信息的变化[27,28]. SFG 结合 CFA 和 SVFG 两者, 可以高效且精确地描述大规模复杂代码的执行行为信息.

为了构建 C 程序的 SFG, 本文首先采用了 LLVM 开源编译器[31]将 C 程序编译为 LLVM 中间代码, 即 LLVM IR. LLVM IR 具有简洁的语句结构和指令集, 将源代码规范为这一中间语言, 可以简化模型构建过程. LLVM IR 是一种部分静态单一赋值(static single assignment, SSA)形式的语言, 其中包含了两种类型的变量: top-level 和 address-taken. top-level 变量是通过使用标准 SSA 构造算法以 SSA 的形式显式放置, 即该类变量在其生命周期中只会被赋值 1 次, 因此, top-level 的指针变量具有确定的指向关系(即 must point-to); 而 address-taken 变量则不是 SSA 形式的, 需要通过 top-level 变量使用 load(即 $p=*q$)和 store(即 $*p=q$)两个指令间接地访问. 由于 address-taken 变量通过 load 进行的间接使用, 并允许被多个 store 指令间接的定义, 该类指针变量通常具有不确定的指向关系(即 may point-to).

对于 LLVM IR 的指针分析算法是用来明确 address-taken 指针变量或相关表达式中的指向关系. 传统的指针分析方法需要借助于程序的控制流和数据流信息来迭代地计算满足所有可能路径下的指向关系(meet-over-all paths, MOP), 才能保证计算出正确的指向关系. 然而对于大型程序, 这种方法的计算成本高昂且无法扩展. 稀疏程序分析(又称为强更新分析[25])则是通过在程序的数据依赖关系上传播数据流事实(即指针指向关系), 而避免了在程序的控制流图中所有程序点上传播的过程, 从而确保了分析的可扩展性. 稀疏程序分析通常分阶段进行: 首先对程序进行预分析, 利用弱更新来定义指针变量所有的 def-use 链. 弱更新会保守地假设在指向关系不明确的位置保留其旧内容, 因此预分析获得的 def-use 链是过度近似的; 然后, 稀疏程序分析在预分析获得的近似的 def-use 链而不是整个控制流上计算明确的指向关系, 即进行强更新计算, 用新的值覆盖掉指针变量中的先前内容, 它保证了指针分析精度的重要因素.

为了构建空间流图, 我们在 LLVM 编译器的基础上, 借助 Andersen 指针分析算法获取指向信息, 并利用两个弱更新函数($\mu(a)$和 $a=\chi(a)$)来描述不确定的指向关系, 以过程间内存 SSA 的形式来描述包含所有 top-level 和 address-taken 指针变量的 def-use 链, 进而结合控制流信息, 构建出空间流图. 图 3 所示为构建 SFG 的一个示例, $\mu(a)$函数用于表示变量 a 的使用(use). 对于指令 load p, $\mu(a)$表示指针 p 指向的每个变量 a(如 i 和 j)在该指令中可能被间接访问. 函数 $a=\chi(a)$用于表示变量 a 的定义和使用(def & use). 对于 store $*p=\&i$, $a=\chi(a)$表示指针 p 指向的每个变量 a 可能被重新定义和使用. 如图 3(c)所示, 我们将 C 程序的 LLVM IR 转换为 SFG 的方式来描述, 为后端模型检测提供形式化模型.

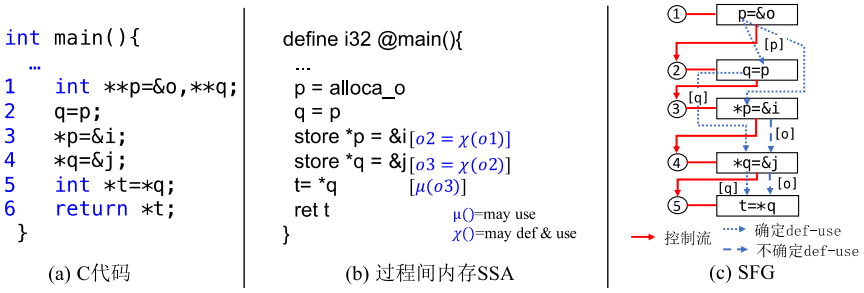

图 3 C 程序的空间流图构建示例

3.2 空间流模型检测

针对空间流模型, 我们提出了反例引导的模型检测算法 CEGAS. 该算法利用发现的虚假反例来同时引导模型抽象精度的细化和值流关系的强更新, 能够在避免不必要的计算开销(如路径约束条件求解、不相关指针分析)的同时, 确保模型检测准确分析变量符号层面和内存地址层面的状态变化过程.

如图 2 所示, CEGAS 检测算法是对 CEGAR 的一种扩展. CEGAS 首先对 SFG 模型控制流信息上的状态变换过程进行抽象, 采用显性值分析的方式验证是否存在反例; 若存在, 则在 SFG 上检测该反例路径上是否存在弱更新而导致不确定的指向关系, 该指向关系可能会导致虚假反例. 因此, 该反例如果存在弱更新, 则沿着该路径执行强更新计算, 以明确的指向关系反馈到抽象模型中, 再进一步验证模型. 本节首先介绍在给定反例路径情况下 SFG 强更新方法, 然后再总体介绍 CEGAS, 具体细节如下.

3.2.1 反例引导的路径敏感强更新

当值分析发现了一条有效的反例路径后, 由于该路径上可能存在弱更新, 而导致错误的流计算, 表明该路径可能是虚假的. 为此, 我们设计了一种反例引导的空间流模型强更新方法, 其核心思想是: 根据值分析检测出的反例路径, 判断该路径上是否存在弱更新, 若存在, 则执行针对路径敏感强更新, 细化了模型中的内存信息变化, 从而进一步分析该路径的有效性.

- 路径敏感: 路径敏感的指针分析是较流敏感指针分析更为准确的分析. 例如针对图 1(a)的示例代码, 利用流敏感指针分析, 在代码第 6 行(l_6)会将指向关系 $a\mapsto i$ 强更新为 $a\mapsto j$. 由于稀疏分析中没有控制流信息, 因此在第 8 行(l_8), 算法会再一次强更新为 $a\mapsto k$. 然而, 当θ_1=true 时, 执行路径无法到达 l_8, 因此 $a\mapsto k$ 的指向关系是错误的. 路径敏感分析则会首先记录路径信息ρ, 通常由一系列谓词组成, 如 $\rho_1=\theta_1\wedge\theta_2\wedge\ldots$和$\rho_2=\neg\theta_1\wedge\theta_2\wedge\ldots$ 由于 $l_6\in\rho_1$, $l_8\in\rho_2$, 因此在第 9 行处, 指针变量 a 的指向关系可以表示为 $a\mapsto\phi(j[\rho_1],k[\rho_2])$. 这里, ϕ 函数表示条件选择函数, 即当前路径为ρ_1时, $a\mapsto j$, 否则 $a\mapsto k$;
- 字段敏感: 字段不敏感的指针分析将结构体变量中不同的字段视为一个单体对象. 而实际上, 不同的字段可能指向不同的对象. 因此, 字段不敏感的指针分析将无法对结构体和数组变量执行有效的强更新, 从而无法保证指向分析结果的正确性. 为此, 我们采用了一种简单、有效的方式来实现字段敏感, 利用 LLVM IR 的 GetElementPtr 指令中多层次的索引信息作为字段的索引, 用以区分同一个结构体变量中不同的字段. GetElementPtr 为 LLVM IR 用于获取结构体或数组元素的指令. 例如, *%tmp= getelementptr inbounds %struct.struct_a* %S, i32 2, i32 1* 表示获取结构体数组 *struct_a* 变量 *S* 第 3 个元素的第 2 个子字段($S_2\cdot e_1$). 首先, 我们为整个结构体或数组变量构建一个对象 *S*, 然后利用 GetElementPtr 中索引信息来构建字段敏感的对象, 进行指向关系更新;
- 约束规则: 强更新是一种在稀疏值流图上的后向可达性分析. 本文采用ρ表示当前分析的路径, 因此我们根据当前ρ, 设计了一系列约束规则来实现该路径上的后向可达性分析. 本文尚未考虑上下文敏感的分析, 而是通过内联的方式将被调用的函数映射到主函数中. 因而, 一个被分析的操作语句包含了ρ, n 的信息, 如$\rho,n:p=\&o$ 表示在当前节点 $n\in N$ 处于路径ρ中. 考虑到弱更新, 一个指针可能存在多

个指向地址, 具有一个指向集. 因此, 本文采用 $p \supseteq q$ 表示将 q 的指向集加入到 p 的指向集中.

我们设计了如表 1 所示的强更新约束规则.

- ADDR 规则表示通过 o 的 def-use 链, 反向获取在节点 n'处声明的 $o[\rho,n']$的内存地址, 来定义 p 的指向关系. [·]表示有效的敏感信息;
- COPY 和 PHI 在 LLVM IR 中是针对 top-level 变量的两个指令, 可以利用确定的 def-use 链获取指向集[7]. 但是由于 PHI 与路径条件相关, 需要判断 n'和 n''是否属于ρ. 当 n'属于ρ时(用 $n' \in \rho$=true 表示), 则 $p[\rho,n] \supseteq q[\rho,n']$, 从而实现了路径敏感;
- GEP 规则用于实现字段访问中的字段敏感分析, 其根据 GetElementPtr 的索引构建针对该字段的对象, 来作为字段指针的指向对象;
- STORE 可能会给 address-taken 变量引入多个间接的 def-use 链, 需要在更新过程中计算出并去除当前虚假的 def-use 链, 以确保指向关系的正确性;
- LOAD 规则表示间接从 address-taken 变量其指向关系集合, 赋值给 top-level 变量使用. 由于多个 STORE 操作的存在, top-level 变量获得的指向关系会存在虚假的 def-use 链;
- SU/WU 规则用于表示 STOREρ,n:*p=_中的强更新和弱更新操作, 在 3 种情况下会执行强更新操作 $kill(p[\rho,n])$: (1) 当指针 p 指向一个路径相关的单体($pcSingletons$)的对象 $o \in \mathcal{O}$时, 需要切除 $n' \xrightarrow{o} n$ 并去掉 o 中原有 n'处的内容, 更新 n 处新的内容; (2) 当 p 指向集为空时, 切除 $n' \xrightarrow{o} n$ 以避免空指针; (3) 其他情况则进行弱更新. 通过 SU/WU 规则可以获得用于处理 LOAD 和 STORE 规则中 address-taken 变量的指向信息;
- COMPO 用于表示 def-use 链的传递性.

表 1 反例引导的路径与字段敏感强更新约束规则

规则	操作语句	约束	作用
ADDR	$\rho,n: p = \&o \; n' \xrightarrow{o} n$	$p[\rho,n] \mapsto o[\rho,n']$	在ρ条件下, p 在 n 处指向 o 的内存地址
COPY	$\rho,n: p = q \; n' \xrightarrow{q} n$	$p[\rho,n] \supseteq q[\rho,n']$	在ρ条件下, p 的指向集包含 q 的指向集
PHI	$\rho,n: p = \phi(q,r)$ $n' \xrightarrow{q} n \; n'' \xrightarrow{r} n$	$p[\rho,n] \supseteq_{n' \in \rho=T} q[\rho,n']$ $p[\rho,n] \supseteq_{n'' \in \rho=T} r[\rho,n'']$	若 n'在ρ中, p 的指向集合则包含 q 的指向集, 否则包含 r 的指向集
GEP	$\rho,n: p = \&(q_i \to e_j)$ $q_i \mapsto o \; n' \xrightarrow{o \cdot e_j} n$	$p[\rho,n] \mapsto o \cdot e_j[\rho,n']$	在ρ条件下, p 指向 $o \cdot e_j$ 的内存地址, 其中, $o \cdot e_j$ 为数组 q 的第 i 个元素中第 j 个字段
STORE	$\rho,n: *p = q \; n' \xrightarrow{p} n$ $n''' \xrightarrow{q} n \; n'' \xrightarrow{r} n$ $p[\rho,n'] \mapsto r[\rho,n'']$	$r[\rho,n''] \supseteq q[\rho,n''']$	在ρ条件下, 获取 p 所指向对象的指向集 r, 使得 r 的指向集包含 q 的指向集
LOAD	$\rho,n: p = *q$ $n' \xrightarrow{q} n \; n'' \xrightarrow{r} n$ $q[\rho,n'] \mapsto r[\rho,n'']$	$p[\rho,n] \supseteq r[\rho,n'']$	在ρ条件下, 获取 p 所指向对象的指向集 r, 使得 q 的指向集包含 r 的指向集
SU/WU	$\rho,n: *p = \_ \; n' \xrightarrow{o} n$ $o \in \mathcal{O} \setminus kill(p[\rho,n])$	$o[\rho,n] \supseteq o[\rho,n']$	$kill(p[\rho,n]) = \begin{cases} o'\rho, & \text{if } p[\rho,n] \mapsto o'[\rho] \wedge o'[\rho] \in pcSingleTons \\ O, & \text{if } p[\rho,n] \mapsto \varnothing \\ \varnothing, & \text{otherwise} \end{cases}$
COMPO	$n \xrightarrow{o} n' \; n' \xrightarrow{o} n''$	$n \xrightarrow{o} n''$	def-use 链的传递

在给定一条反例路径后, CEGAS 首先沿着该路径后向遍历 SFG 中值流信息, 判断该路径上是否存在弱更新. 若存在弱更新, 则表明当前的指向信息存在不明确关系, CEGAS 利用上述约束规则分析 SFG, 执行强更新操作, 去除该路径上弱更新带来的虚假 def-use 链, 进而为接下来的模型检测提供该反例敏感的指向信息.

3.2.2 反例引导的空间流模型检测

模型检测问题可以形式化地描述为$\mathcal{M} \vDash \varphi$, 其中, $\mathcal{M}$为一种 Kripke 结构的模型, φ为检测属性. 本文主要关注于程序的安全属性(safety property), 即程序始终不会到达某个不期望的状态 $state$, 利用线性时态逻辑可以表述为$\varphi \Leftarrow LTL(G!\ state)$. 给定一个 Kripke 结构的模型, 诸多先进的模型检测算法可以用来检测模型正确性[4],

如谓词分析、显性值分析、符号分析、IC3、抽象等. 本文主要采用基于抽象细化的显性值分析[29](也称之为显性状态分析)作为主要的模型检测算法, 结合模型抽象, 这种分析方法可以去除与检测属性不相关的程序变量, 只跟踪那些可以反驳无效的反例路径所必需的变量, 从而抑制由于程序变量及其值过多所导致的状态空间爆炸问题, 使得验证更加简洁、高效. CEGAS 利用抽象细化的模型检测策略, 从空的抽象精度开始, 迭代地从无效路径中细化符号变量和指向信息的精度. 具体的细节如下.

给定一个程序的空间流模型, CEGAS 首先需要对模型进行抽象, 其中分为两种状态.

- 一种是具象状态 s, 它是指一个变量赋值 $s:=cs@l$, 其中, $cs:V\to\mathbb{Z}$表示给一个程序变量被赋予一个整数值, $l\in L$ 为一个程序位置. 类似于现有模型检测算法, 本文的模型检测主要关注于整型变量;
- 另一种是抽象状态 $\hat{s}$, 它表示为 $\hat{s}:=as@l$, 其中, $as:V\to\mathbb{Z}\cup\{\top,\bot\}$为一个抽象的变量赋值, 其中, $\top$表示一个未知值, 如由一个未初始化的变量赋值, 或外部函数调用; $\bot$表示无数值, 即一个矛盾的变量赋值.

抽象状态由具象状态根据精度抽象而来. 本文采用了一种惰性的显性值抽象[12]方法, 它对不同程序路径上的不同抽象状态使用不同精度. 对于一个程序, 精度为一个函数 $\Pi:L\to 2^V$, 它为程序的每个位置提供了一个精度 π, 用于实现变量赋值语句的抽象. 针对一个变量赋值语句, 精度 π定义了一组需要被分析跟踪的程序变量, 用来对变量赋值语句进行抽象. 对于一个给定的精度 π, 一个变量赋值语句的显性值抽象是一个其变量受 π决定. 例如, $\pi=\varnothing$表示所有程序变量都不被跟踪分析, 相应的抽象状态也为空; $\pi=V$ 表示所有程序变量都被跟踪, 相应的抽象状态等于具象状态; $\pi=\{i\}$, 变量赋值集合 $v=\{i\mapsto 1,j\mapsto 2\}$的显性值抽象状态为 $v^{\pi}=\{i\mapsto 1\}$. 总的来说, 在程序位置 $l\in L$, 一个变量赋值的显性值抽象通过精度函数 $\Pi(l)$提出去精度, 进而根据精度跟踪的变量, 计算出抽象状态. 在检测的开始阶段, CEGAS 的精度为 $\Pi_{init}(l)=\varnothing$, 即对于每个 $l\in L$, 所有变量都不被跟踪.

我们定义一条路径为一个由一系列操作语句和程序位置对组成的序列 $\rho:=\langle(op_1@l_1),\ldots,(op_n@l_n)\rangle$, 其中, $\gamma_\rho:=\langle op_1,\ldots,op_n\rangle$为该路径的约束序列. 本文程序位置以 LLVM IR 的 BasicBlock 为基本单位, op 中通常包含多个操作语句. 当 CEGAS 发现一条路径 ρ能够到达违反检测属性 φ的错误位置 l_φ时, 即一个反例 CEX, 它首先在满精度($\Pi(l)=V$)的情况, 利用 SMT 技术分析反例 ρ的可行性(即函数 *isFeasible*(ρ)判断约束序列 γ_ρ是否可满足). 若不满足, 则可能是由于精度不够精细或者存在由弱更新导致的错误指向信息, 需要对精度进行细化. 本文使用了克雷格插值的方法来生成新的插值, 其定义如下.

给定公式 φ^-和公式 φ^+, 且 $\varphi^-\wedge\varphi^+$是不可满足的, 公式 φ^-和公式 φ^+的克雷格插值(Craig interpolant) ι是指一个能够满足如下约束的公式: (1) $\varphi^-\Rightarrow\iota$有效, 即 $\varphi^-\wedge\neg\iota$不可满足; (2) $\iota\wedge\varphi^+\Rightarrow$false, 即 $\iota\wedge\varphi^+$不可满足; (3) ι仅含有公式 φ^-和 φ^+公共的符号, 以及理论本身的符号. 该定义可以扩展到有序的公式序列 $\rho=\varphi_0,\ldots,\varphi_n$, 且 $\wedge_{0\leqslant i\leqslant n}\varphi_i\Rightarrow$ false, 利用上述插值方法可以获得一系列插值 $\iota_0,\ldots,\iota_n$: (1) $\wedge_{0\leqslant k\leqslant i}\varphi_k\Rightarrow\iota_i$有效; (2) $\iota_i\wedge\wedge_{i\leqslant k\leqslant n}\varphi_k\Rightarrow$false; (3) $\forall 1\leqslant i\leqslant n$, $\varphi_{i-1}\wedge\iota_i\Rightarrow\iota_{i+1}$; (4) ι_i仅含有公式 $\wedge_{0\leqslant k\leqslant i}\varphi_k$和 $\wedge_{i\leqslant k\leqslant n}\varphi_k$公共的符号, 以及理论本身的符号.

上述序列插值(用函数 *SeqInterpolant* 表示)可以通过 SMT 技术(如 Z3[32])有效地计算出来. 由于上述只能够利用值空间上的信息来生成插值, 缺少了地址空间信息, 存在精度细化不完整的问题. 为此, 本文采用该序列插值为基础, 提出了指向信息敏感的精度细化算法(Refine), 其伪代码如算法 1 所示. 在给定一条不可行的错误路径 ρ时, 该算法首先计算出 ρ导致的约束序列不可满足的第 1 个节点(第 2 行–第 4 行). 对于路径约束可满足的部分, 我们使用表 1 所示的路径敏感的强更新规则(由函数 *PStrongUpdate* 表示)来分析该路径中是否存在弱更新, 若存在弱更新, 则对该部分进行强更新, 避免错误指向信息的存在(第 5 行、第 6 行). 然后, 算法利用函数 *SeqInterpolant* 计算出插值序列 I(第 7 行–第 10 行); 在获取 I, 将对应程序位置 l_k的插值 ι_k所用的程序变量细化到以 l_k为起点, 沿着路径 ρ后向的所有位置的精度中, 同时, 函数 *BackReachability* 在空间流模型 $\mathcal{M}$找出插值 ι_k所用变量在程序位置 l_k后向的 def-use 链, 进而将 ι_k所用变量被指向的指针变量加入对应位置的精度中. 该算法可以为模型抽象, 从值空间和地址空间两个方面, 迭代地细化精度.

算法 1. 指向信息敏感的抽象精度细化算法(*Refine*(ρ,$\mathcal{M}$)).

输入: 不可行错误路径 $\rho:=\langle(op_1@l_1),\ldots,(op_n@l_n)\rangle$, 空间流模型 $\mathcal{M}$;

输出: 精度Π.

变量: 插值序列 I.

1 设所有程序位置 l 的精度为$\Pi(l):=\varnothing$; $I:=\langle\cdot\rangle$

//找出导致路径ρ上约束不满足的第 1 个节点;

2 **for** $i:=n-1$ to 0 **do**

3 **If** $\wedge_{0\leqslant k\leqslant i}op_k\Rightarrow$true **then**

4 $\rho_I:=\langle(op_0@l_0),\ldots,(op_i@l_i)\rangle$;

5 **If** $hasWeakUpdatePath(\rho_I,\mathcal{M})$:=true **then**

6 $\mathcal{M}:=PStrongUpdate(\rho_I)$; //路径满足, 但存在弱更新, 对模型$\mathcal{M}$进行路径敏感的强更新

7 **for** $j:=i-1$ to 0 **do**

8 $\iota_j:=SeqInterpolant(\wedge_{0\leqslant k\leqslant j}op_k,\wedge_{j\leqslant k\leqslant i}op_k)$;

9 **If** ι_j is empty **then Break**;

10 $I:=\{I\cup\iota_j\}$;

11 **for** $k:=j$ to $i-1$ **do**

12 $\Pi(l_{0\leqslant t\leqslant k}):=\{v|v\in\iota_k,\iota_k\in I\}$; //在路径$\rho$上包括 l_k 后向所有程序位置的精度中加入插值所用变量

13 $\Pi(l_{0\leqslant t\leqslant k}):=\{v^p|v^p\in BackReachability(\mathcal{M},l_{0\leqslant t\leqslant k},\iota_k)\}$; //同上, 加入插值变量被指向的指针变量

结合上述细化方法, CEGAS 借助抽象可达图(abstract reachability graph)以树状形式记录分析过的状态及其之间的可达关系, 使用了两个中间变量来记录分析过程中的信息: 集合 $reached\subseteq E\times\Pi$用于记录当前精度下所有可达的抽象状态; 集合 $waitlist\subseteq E\times\Pi$用于记录当前精度下所有为被分析的抽象状态. E 是指抽象状态集合. CEGAS 算法的伪代码如算法 2 所示, 具体步骤如下.

算法 2. 反例引导的抽象细化与强更新(CEGAS).

输入: 稀疏空间流模型$\mathcal{M}$, 模型检测属性φ;

输出: 检测结果安全或不安全(反例路径).

变量: 精度π_0, 集合 $reached\subseteq E\times\Pi$, 集合 $waitlist\subseteq E\times\Pi$.

1 $\pi:=\pi_0:=\varnothing$;

2 $reached:=\{(b_0,\pi_0)\}$; $waitlist:=\{(b_0,\pi_0)\}$;

3 **While** true **do**

4 **While** $waitlist\neq\varnothing$ **do**

5 choose and remove (b,π) from $waitlist$

6 **For** each b' with $b\rightarrow E(b',\pi)$ **do** //遍历 b 的继承者分支, $b\rightarrow E(b',\pi)$表示在精度π下 b 与 b'分支条件可满足的

7 $\hat{b}:=abstraction(b',\pi)$; //利用精度$\pi$计算 b'抽象的抽象状态

8 **If** $isCovered(reached,\hat{b})\neq$true **then**

9 $waitlist:=(waitlist\cup\{(\hat{b},\pi)\})$; //加入新的抽象状态

10 $reached:=(reached\cup\{(\hat{b},\pi)\})$; //在抽象可达图, 加入新抽象状态, 并记录与之前状态的路径

11 **If** $isTargetState(b',\varphi)$:=true **then** //判断当前 Basicblock 中是否有目标状态

12 **Break While** //退出当前 while 循环

13 **If** $waitlist\neq\varnothing$ **then**

14 $\rho_a:=extractErrorPath(reached)$; //从抽象可达图中提取出抽象路径, 即抽象反例

15 **If** $isFeasible(\rho_a)$:=false **then** //利用 Z3 证明路径ρ_a的满足性

16 $\pi:=\pi\cup Refine(\rho_a,\mathcal{M})$; //如果不满足, 即存在该路径为虚假反例, 对当前精度进行插值

17 **Else if** $hasWeakUpdatePath(\rho_a,\mathcal{M})$:=true **then**

18 $\mathcal{M}$:=*PStrongUpdate*(ρ_a); //路径满足, 但存在弱更新, 对模型$\mathcal{M}$进行路径敏感的强更新

19 π:=$\pi\cup$*Update*(ρ_a,$\mathcal{M}$); //利用强更新信息, 去除精度中弱更新指向信息带来的指针变量

20 **Else return** unsafe; //当前反例路径可行, 违反检测属性φ

21 *reached*:={(b_0,π)}; *waitlist*:={(b_0,π)}; //更新 *reached* 和 *waitlist*, 重新进行抽象分析

22 **Else return** safe;

- 步骤 1: 设置当前精度π为空的精度π_0 (即不记录任何变量), 并初始化 *reached* 和 *waitlist* (第 1 行、第 2 行);
- 步骤 2: 执行显性值分析模型检测算法, 从 *waitlist* 中提取未分析状态 *b*. 在精度π下找出 *b* 可达的继承者(即分支条件在π抽象后是可满足的), 并对继承者进行抽象: 若新的状态未被分析过, 则合并到 *reached* 和 *waitlist*; 若新的状态违反属性φ, 表示发现一个反例路径, 退出当前循环(第 4 行–第 12 行);
- 步骤 3: 判断 *waitlist* 中是否还有剩余未分析状态, 若不存在, 则表明程序对于属性是正确的, 因此停止检测(第 22 行);
- 步骤 4: 若 *waitlist* 不为空, 则从抽象可达图中构建出抽象反例路径ρ_a, 并根据ρ_a 的信息, 利用函数 *isFeasible* 判断路径ρ_a是否在原始模型是否有效(第 13 行–第 15 行);
- 步骤 5: 若ρ_a 无效, 则表明当前路径不可行. 算法使用指向信息敏感的约束差值的精度细化算法(Refine)来更新精度π(第 16 行);
- 步骤 6: 若ρ_a 有效, 则沿着该路径反向遍历模型$\mathcal{M}$, 判断ρ_a 上是否存在弱更新. 若存在弱更新, 则利用表 1 所示约束规则对$\mathcal{M}$进行路径敏感强更新, 去除弱更新带来的指向信息, 然后更新精度中指向关系带来的跟踪变量, 用以去除错误指向信息导致变量赋值抽象状态(第 17 行–第 19 行);
- 步骤 7: 若ρ_a 有效, 且ρ_a 上不存在弱更新, 则表面该路径在值空间和地址空间上都有效, 表明程序存在违反属性φ的反例, 中断模型检测, 并报告发现的反例(第 20 行);
- 步骤 8: 在 *reached* 和 *waitlist* 更新细化后的π, 并跳转至步骤 2 再次进行模型检测(第 21 行).

CEGAS 在第 1 次模型检测时使用空的精度, 这主要考虑两方面原因: 一方面是利用最简单的抽象模型可以快速地检测程序中是否可能存在违反属性的状态, 从而避免了无效的计算分析, 如反例路径上不存在指针操作; 另一方面, 初始的 SFG 使用的是不敏感的指针分析, 因此存在很多虚假的指向信息, 会给模型检测引入错误的判断. 此后, 如果发现检测出的路径上存在弱更新, 则进行强更新后, 给模型检测提供正确的路径敏感的指向信息. 若不存在弱更新, 则表示该路径上指向信息是正确的, 进而检测出的路径认为是有效的.

这里进一步以验证图 1(a)所示的代码为例子, 来说明 CEGAS 的模型验证过程, 其过程结果由图 4 所示.

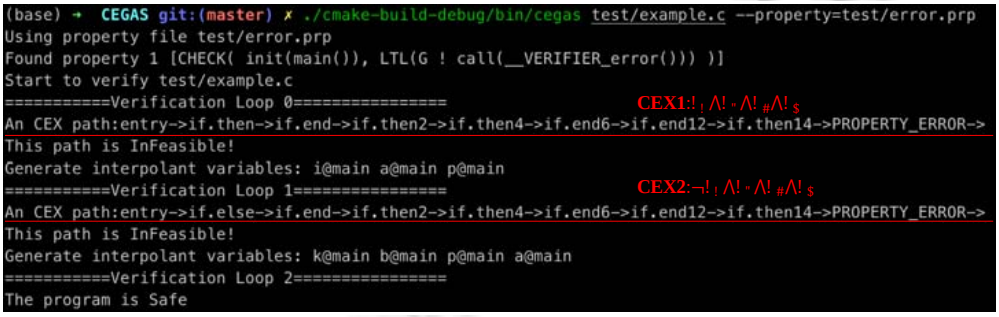

图 4 CEGAS 处理图 1 所示的 C 程序的示例

CEGAS 首先以空的抽象精度遍历代码的控制流图, 根据属性φ, 发现了反例路径 CEX1: $\theta_1\wedge\theta_2\wedge\theta_3\wedge\theta_5$, 但是 CEX1 在原始模型上是无效的, 由此, CEGAS 利用算法 Refine 找出θ_1 是导致 CEX1 开始无效的约束, 进而找出插值变量 *i* 和指针变量 *a* 和 *p*, 增加当前抽象精度. CEGAS 在第 2 次检测时, 又发现无效反例 CEX2: $\neg\theta_1\wedge\theta_2\wedge\theta_3\wedge\theta_5$. 通过分析, 发现导致 CEX2 无效的原因是由于约束θ_5, 亦即缺少对变量 *k* 的精度, CEGAS 计算

出新的插值变量, 包括变量 k 和指针变量 b, p 和 a. 注意到, 这里也计算出了指针 p 和 a. 但是不同的是: 上一步骤计算出的 p 和 a 是用于记录包括约束 θ_1 之前的相关状态, 而这一步骤则是从约束 θ_5 出发反向地在路径 CEX2 上增加 p 和 a 相关的精度. 由于在原先不敏感的空间流模型中, p 所指向的对象 a 在第 9 行前是被视为同时指向 i, j 和 k 这 3 个变量, 但是算法 Refine 中发现这里存在弱更新, 根据该路径信息进行强更新, 即在位置第 9 行之后, 将原始在第 2 行处获得指向关系 $a\mapsto i$ 和第 6 行处 $a\mapsto j$ 去除, 保留 $a\mapsto k$ 的指向关系. CEGAS 在进行第 3 次验证时, 发现不存在能够到达调用 *error*(·)函数的路径, 因此证明图 4(a)所示的代码对于属性 φ 是安全的.

4 实验分析

4.1 实验设计

我们在 LLVM (6.0.0)框架和 Z3 SMT 解算器的基础上建立工具 CEGAS, 用来验证输入程序中的断言. 其中, 我们使用 Andersen 算法来建立程序的不敏感值流信息, 可以确保 CEGAS 在线性时间内构建出程序的空间流模型. 由于本文设计的路径敏感的强更新未实现上下文敏感, CEGAS 通过内联的方式将 LLVM IR 中被调用函数代码映射到主函数中, 因此, CEGAS 只需要分析主函数的空间流模型. 另外, 本文所实现的 CEGAS 主要针对整型变量, 暂不支持浮点变量.

为了验证所设计的 CEGAS 的有效性, 我们结合现有的 C 程序验证基准建立了一系列新的基准代码. 比赛 SV-COMP[33]提供了一个用于程序(C 和 Java 程序)验证的基准库 sv-benchmarks (https://github.com/sosy-lab/sv-benchmarks), 但是其中大部分的基准代码主要用于符号层面的逻辑验证. 为此, 本文结合 sv-benchmarks 和一个用于评估指针分析能力的基准库 Test-Suite (一个用于评估 C/C++程序指针指向关系分析能力的基准库: https://github.com/SVF-tools/Test-Suite), 建立一系列包含多种指针操作的 C 程序基准代码, 用于评估模型检测算法的检测准确度和检测效率(我们将该基准库开源在链接 https://github.com/SFChecker/Benchmark). 通过结合指针操作和其他 C 语言不同的语法语义特性来设计基准代码, 根据所包含的特性, 这些基准代码被分为数组(array)、函数调用(callsite)、全局变量(global)、循环(loop)、路径(path)、结构体(struct)这 6 组基准代码. 其中, 数组这组基准代码操作的变量以数组为主, 每个代码包含数组指针、指针数组、结构体数组指针或指针结构体字段等至少一种复杂类型的变量及其相关指令; 函数调用这组基准代码包含的主要语义特性是局部或全局指针变量在函数调用中的地址传递, 部分代码包含了递归函数和函数指针的使用; 全局变量这类基准代码主要是用于分析全局变量(如整型变量、指针变量、结构体指针)在过程内和过程间传递使用中的逻辑正确性; 循环这类基准代码以使用循环指令(如 for 和 while)为主, 以用于评估软件验证工具针对循环过程中程序状态变化正确性的检测能力, 尤其是针对存在循环的指针操作的语义; 路径这组基准代码类似于图 1(a)的示例代码, 包含了多个嵌套的路径条件分支, 每个分支中包含各类指针操作指令; 结构体这组基准代码操作的变量以结构体为主, 结构体的字段包含整型变量、指针变量、子结构体、数组等不同类型, 用以评估字段敏感的分析能力. 基于这些特性, 我们设计这些基准库代码以验证 SV-COMP 的可达性属性为主要检测属性, 即我们在代码特定位置处设定 *__VERIFIER_error*(·)的函数调用, 该基准库的验证目标是: 在目标代码中, 从 *main* 函数的入口开始, 验证函数 *__VERIFIER_error*(·)是否可达. 该基准库能够为评估模型检测算法的检测能力同时提供符号层面和地址空间的分析案例.

在上述基准库的基础上, 我们选择了几个主流的软件检测工具与 CEGAS 进行对比分析, 包括 CPAChecker[9]、Gazer[11]和 SMACK[18]. 这些工具都在 SV-COMP 比赛中获得过优异成绩. 类似于 CEGAS, 它们都支持面向 C 程序 LLVM IR 代码的模型检测. CPAChecker 是一个能够支持多种检测算法并行检测的 C 程序验证器, 由 Java 语言实现. 由于 CEGAS 是以显性值分析为主要分析方法, 我们将 CPAChecker 分别配置为单独显性值分析模式(CPA-V)、显性值与谓词分析结合模式(CPA-VP)以及 *k*-induction、显性值和谓词结合模式(CPA-KVP, CPAChecker 在 SV-COMP 的默认配置)这 3 种不同模型. Gazer 由一个基于 LLVM 的可以将 C 程序转换为控制流自动机的前端 Gazer (由 C++语言实现)和一个模型检测框架 Theta (由 Java 语言实现)组成, 并

采用基于显性值和谓词分析结合的CEGAR实现C程序的模型检测. SMACK (由C语言实现)是一个模块化的软件验证工具链, 借助于DSA[34]预先计算出LLVM IR中所有指针别名关系, 然后通过将LLVM IR和指针指向信息转换为Boogie[35]中间验证语言后, 利用Boogie验证器进行模型检测. 对比分析均在3.7 GHz Intel i9-10900K CPU和64 G内存的Linux主机上进行.

4.2 实验结果分析

上述的C代码检测工具包括CEGAS在检测C程序时, 如果检测出存在有效的反例路径, 则报告Unsafe; 如果未检测出反例路径, 则报告Safe; 如果检测工具出现无法确认当前反例路径是否具有有效性而无法进行检测时, 则报告Unknown. 同时, 考虑到本文所采用的基准库都较小, 我们设定1分钟为失效时间, 即1分钟内, 检测工具没有检测出结果, 则中断检测, 称该检测时间为Timeout, 并设置检测结果为Unknown. 此外, 部分检测工具存在一些不支持的C语句(如Gazer不支持CallInst指令), 会直接抛出异常, 我们称该检测时间为Failed, 检测结果为Unknown. 由于篇幅问题, 我们在表2中给出了部分检测结果和检测时间, 完整的检测结果见链接https://github.com/SFChecker/Benchmark.

表2 基准代码检测结果与检测时间(ms)

工具 / 基准	GT	CPA-V	CPA-VP	CPA-KVP	SMACK	Gazer	CEGAS
array/array0.c	**Unsafe**	**Unsafe/460**	**Unsafe/132**	**Unsafe/589**	**Unsafe/1165.1**	**Unsafe/24.9**	**Unsafe/10.7**
array/array4.c	**Safe**	Unknown/351	**Safe/141**	**Safe/328**	**Safe/1426.8**	**Safe/16.8**	**Safe/17.1**
array/array6.c	**Safe**	Unsafe/424	Unsafe/156	Unsafe/352	Unsafe/1158.3	Unsafe/77.6	**Safe/19.0**
array/array9.c	**Safe**	Unknown/504	Unsafe/250	Unsafe/367	Unsafe/1583.8	**Safe/19.9**	**Safe/38.5**
callsite/ callsite0.c	**Safe**	Unknown/205	**Safe/95**	**Safe/206**	**Safe/1387.3**	Unknown/ Failed	**Safe/31.9**
callsite/ callsite7.c	**Unsafe**	Unknown/196	**Unsafe/95**	**Unsafe262**	**Unsafe/1582.7**	**Unsafe/60.5**	**Unsafe/24.5**
callsite/ callsite10.c	**Unsafe**	**Unsafe/209**	**Unsafe/79**	**Unsafe/586**	Safe/938.2	Unknown/ Failed	Safe/71.9
callsite/ callsite15.c	**Unsafe**	**Unsafe/244**	**Unsafe/129**	**Unsafe/334**	**Unsafe/1203.9**	**Unsafe/42.7**	**Unsafe/32.2**
global/global1.c	**Unsafe**	**Unsafe/208**	**Unsafe/92**	**Unsafe/230**	**Unsafe/1116.4**	Unknown/21.3	**Unsafe/15.1**
global/global3.c	**Unsafe**	Unknown/201	**Unsafe/95**	**Unsafe/224**	**Unsafe/1126.2**	**Unsafe/35.4**	**Unsafe/17**
global/global4.c	**Unsafe**	**Unsafe/194**	**Unsafe/96**	**Unsafe/223**	**Unsafe/1162**	Unknown/49.2	**Unsafe/20**
global/global7.c	**Unsafe**	**Unsafe/200**	**Unsafe/93**	**Unsafe/238**	**Unsafe/1158.8**	**Unsafe/57.8**	**Unsafe/19.3**
loop/loop1.c	**Unsafe**	Unknown/233	Unknown/ Failed	**Unsafe/446**	Safe/1476.8	**Unsafe/179.8**	**Unsafe/68.1**
loop/loop3.c	**Safe**	Unknown/490	Unknown/ Failed	**Safe/5832**	**Safe/1518.1**	Unknown/159	**Safe/335.1**
loop/loop11.c	**Unsafe**	Unknown/245	Safe/370	Safe/2344	Safe/1542.9	**Unsafe/27.3**	**Unsafe/92.2**
loop/loop15c	**Safe**	Unknown/266	Unknown/ Failed	Unknown/ Timeout	**Safe/1919.5**	Unsafe/404.2	**Safe/506.5**
path/path1.c	**Safe**	Unsafe/518	Unsafe/271	Unsafe/519	Unsafe/1156.7	Unknown/54.3	**Safe/30.9**
path/path6.c	**Safe**	Unknown/507	Unsafe/229	Unknown/Failed	Unsafe/1193.2	Unsafe/70.6	**Safe/29.6**
path/path13.c	**Safe**	Unknown/599	Unsafe/304	Unsafe/858	Unsafe/1938.3	**Safe/17.7**	Unsafe/18.6
path/path25.c	**Safe**	Unknown/570	**Safe/269**	**Safe/524**	**Safe/1770.6**	Unsafe/79.6	**Safe/29.5**
struct/struct1.c	**Unsafe**	**Unsafe/220**	**Unsafe/97**	**Unsafe/229**	**Unsafe/1160.4**	**Unsafe/65.1**	**Unsafe/29.9**
struct/struct5.c	**Safe**	Unsafe/230	Unsafe/110	Unsafe/230	Unsafe/1182.5	Unsafe/78.3	**Safe/22.1**
struct/strcut7.c	**Safe**	Unknown/192	Unsafe/83	**Safe/184**	Unsafe/1179.1	Unsafe/73.3	**Safe/17.9**
struct/struct14.c	**Safe**	Unknown/189	**Safe/74**	**Safe/189**	**Safe/1375.8**	**Safe/19.1**	Unsafe/15

在表2中, GT为Groundtruth, 即基准库存在反例的真实情况. 可以看出: 针对不同的基准代码, 各个检测工具的检测结果和检测时间存在一定的差异. 由于这些基准代码中存在诸多指针操作, 现有的检测工具存在诸多检测不准确或者无法检测的情况. 例如: 各个工具对函数调用的处理方式不同, Gazer对函数调用分析支持较差, 分析该组基准代码的准确度最低. 由于CEGAS采用了内联的方式来处理上下文调用问题, 通过实验分析, 我们发现: 在多数情况下, 这种方法可以有效解决该问题; 但当存在递归函数时, 如callsite10.c、CEGAS会带来错误检测结果. SMACK对递归函数支持能力比较有限, 无法正确验证callsite10.c中的属性. path的基准代码中一般存在多种条件分支语句, 这些分支语句中包含多种指针引用和解引用的指令, CEGAS

通过反例引导的稀疏值流强更新方式, 可以有效地更新指针指向关系, 从而获得比其他工具更加准确的检测结果. 各个工具对结构体的字段敏感程度不同, 导致分析的准确度有所不同, 如 CPA-V 对 arry6.c、array9.c(包含结构体数组指针)和 struct 基准代码的检测准确度较低. CEGAS 采用字段敏感的分析, 能够准确分析大多数结构体相关的基准代码, 但是结构体结合静态变量或函数调用情况时, 也会出现错误判断的情况. 在 6 组基准代码中, loop 的检测时间开销较大, CEGAS 对循环代码展开次数加以限定, 由于 loop 里基准代码的循环次数较小, CEGAS 能够检测出正确结果. 另外, 可以看出: 针对实验用的基准库, 由于缺乏正确的指向信息, CPAChecker 的显性值分析大多数情况下都无法进行有效的模型检测.

图 5 和图 6 说明了各个工具对实验所用基准库的整体效果. 其中, 图 5 说明了各工具针对每类基准的检测准确率. 可以看出, CEGAS 在这 6 类基准代码上都取得了较高的准确度. 在 CPAChecker 的 CPA-V、CPA-VP 和 CPA-KVP 这 3 种检测策略中, 结合了显性值分析、谓词分析和 *k*-induction 的 CPA-KVP 能够获得更加准确的检测结果. 但是, CPAChecker 目前只支持 Anderson 和 Steensgaard 以获得指针关系, 如图 5 所示, CPA-V 在实验所用的基准库上检测准确度较低. 通过加入谓词路径的约束, CPA-VP 可以一定程度地提高 CPA-V (即单一显性值分析)的准确度. SMACK 利用外部指针分析工具 DSA 预先计算出指向信息, 为其进行模型检测提供有效的指向信息, 使得 SMACK 获得较优的准确度. Gazer 类似于 CPA-VP 的方式, 通过结合显性值分析和谓词路径约束来提高分析准确度. 但是我们可以发现, 这些工具在这 6 类基准代码上的准确度差异较大. 而 CEGAS 通过联合显性值分析和指针分析, 除了 callsite 之外都能够获得最高的准确度. 由于 CEGAS 采用内联的方式来解决函数调用问题, 但当 callsite 基准中存在递归函数时, 这种方法就无法去除函数调用, 有可能导致 CEGAS 出现无效结果.

图 6 所示为各工具检测效率的累积分布图(CDF 为累积分布函数), 我们使用检测时间(ms)除以被检测代码行数来评估检测效率, 数值越高, 效率越低. 对于 Timeout 的检测结果, 我们将其效率设置为 250; 对于 Failed 的检测结果, 我们将其效率设置为–10. 其中, 可以看出: SMACK 的检测效率最低, CPAChecker 的 3 个策略其次, CEGAS 和 Gazer 效率接近. 不同于其他几个工具, SMACK 通过将优化后的 LLVM IR 代码转换为 Boogie 的验证语言后, 直接调用 Boogie 验证器进行验证, 验证效率较低. 而其他工具则均采用基于抽象的验证策略, 使用 SMT 解算器作为其中的约束求解器, 验证效率得到提升. 通过联合显性值分析和谓词分析的方式, CPA-VP 和 Gazer 都取得了较优的检测效率. CPA-KVP 通过在 CPA-VP 上加入 *k*-induction 的额外检测, 能够提高检测准确度, 但也会导致其检测效率有所下降. CPAChecker 的 3 种检测策略和 Gazer 均存在多种检测超时或者检测不支持的情况, 其中, Gazer 的失败率最高(20%), CPA-KVP 的超时情况最多(4 次), 而 SMACK 和 CEGAS 都能够在一定时间内给出检测结果. CEGAS 联合显性值分析和稀疏值流分析, 能够在模型检测过程中提供准确符号变量层面和地址空间层面的状态变化信息, 降低了由于错误指向信息导致的分析开销, 使其获得显著的检测效率.

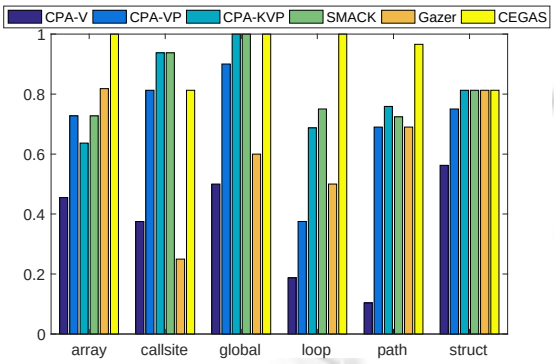

图 5 CEGAS 与现有工具的验证准确度

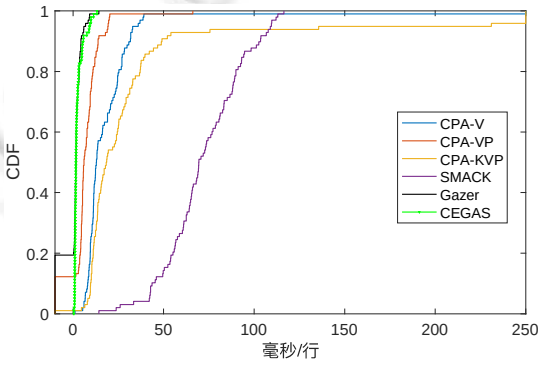

图 6 CEGAS 与现有工具的验证效率

最后, 我们去除检测时间为 Timeout 和 Failed 的结构来计算各个检测工具对基准库的总体检测准确度和

平均检测效率, 见表3. 可以看到: CEGAS在所用基准库上总体的检测准确度为92.9%, 平均检测效率为2.58 ms/行, 均优于其他对比工具. 综上所述, 相比于现有 C 代码模型检测算法, 本文通过联合显性值分析和稀疏值流分析的方法, 设计的反例引导的空间流模型检测方法CEGAS针对包含各类指针操作的C代码, 均能取得更加准确和更高效率的检测.

表3 基准代码总体检测准确度与平均检测效率

指标	CPA-V	CPA-VP	CPA-KVP	SMACK	Gazer	CEGAS
准确度(%)	31.6	69.4	79.6	80.6	61.2	**92.9**
平均效率(ms/行)	16.6	9.0	25.52	72.3	2.64	**2.58**

5 总 结

本文针对现有 C 代码模型检测算法无法有效分析地址空间层面的状态变化而导致的检测准确度低的问题, 设计了一种反例引导的空间流模型检测算法. 本文首先设计了一种空间流模型, 通过结合控制流信息和稀疏值流信息, 能够有效地描述程序在变量符号层面和地址空间层面的状态变化; 提出了反例引导的抽象细化与强更新算法, 通过不敏感的指针分析方法实现快速的空间流模型构建, 利用变量抽象的方式抽象模型, 然后通过发现的无效反例引导模型抽象粒度的细化和不敏感指向关系的强更新操作, 实现检测效率和准确度的有效权衡. 通过设计一系列 C 代码基准库, 本文将提出的方法与现有前沿检测工具进行比较, 在多种 C 语言特性的基准代码中, 本文所提出的方法在检测准确度和检测效率上均取得了突出结果.

诚然, 本文所提出的方法还存在诸多缺陷, 需要在未来的工作中加以改进: (1) 本文所提出方法通过内联的方式解决上下文问题, 存在一定的缺陷, 在后期工作中需要改进路径敏感的强更新, 以支持上下文敏感; (2) 在实验中, 我们发现CPAChecker和Gazer结合显性值分析和谓词分析的方法较纯粹的显性值分析方法能够获得更准确的结果, 未来, 可以改进CEGAS以支持谓词分析; (3) CEGAS在包括状态抽象、路径可行性分析、分支条件判断等多个地方均需调用 SMT 解算器, 开销较大, 且部分约束求解(如分支条件判断)可以采取本地计算的方式来提高效率, 如文献[27]中所采用的方法.

References:

[1] D'silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2008, 27(7): 1165−1178. [doi: 10.1109/TCAD.2008.923410]

[2] Clarke E, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM), 2003, 50(5): 752−794. [doi: 10.1145/876638.876643]

[3] Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 2018, 51(3): 1−39. [doi: 10.1145/3182657]

[4] Clarke EM, Henzinger TA, Veith H, Bloem R. Handbook of Model Checking. Cham: Springer, 2018.

[5] Kalra S, Goel S, Dhawan M, Sharma S. Zeus: Analyzing safety of smart contracts. In: Proc. of the Symp. on Network and Distributed Systems Security (NDSS). 2018. 1−12. [doi: 10.14722/ndss.2018.23082]

[6] Yu Y, Li Y, Hou K, Chen Y, Zhou H, Yang J. CellScope: Automatically specifying and verifying cellular network protocols. In: Proc. of the ACM SIGCOMM 2019 Conf. on Posters and Demos. 2019. 21−23. [doi: 10.1145/3342280.3342294]

[7] Chaki S, Datta A. ASPIER: An automated framework for verifying security protocol implementations. In: Proc. of the 22nd IEEE Computer Security Foundations Symp. IEEE, 2009. 172−185. [doi: 10.1109/CSF.2009.20]

[8] Zhang XL, Zhu YF, Gu CX, Chen X. C2P: Formal abstraction method and tool for C protocol code based on Pi caculus. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1581−1596 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6238.htm [doi: 10.13328/j.cnki.jos.006238]

[9] Beyer D, Keremoglu ME. CPAchecker: A tool for configurable software verification. In: Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Berlin, Heidelberg: Springer, 2011. 184−190. [doi: 10.1007/978-3-642-22110-1_16]

[10] Holzmann GJ. Software model checking with SPIN. Advances in Computers, 2005, 65: 77−108. [doi: 10.1016/S0065-2458(05)65002-4]

[11] Ádám Z, Sallai G, Hajdu Á. Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2021. 433−437.

[12] Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In: Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 2022. 58−70. [doi: 10.1145/565816.503279]

[13] Henzinger TA, Jhala R, Majumdar R, McMillan KL. Abstractions from proofs. In: Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 2004. 232−244. [doi: 10.1145/982962.964021]

[14] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. DIKU: University of Copenhagen, 1994.

[15] Beyer D, Dangl M, Wendler P. A unifying view on SMT-based software verification. Journal of Automed Reasoning, 2018, 60(3): 299−335. [doi: 10.1007/s10817-017-9432-6]

[16] Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. In: Proc. of the Operating Systems Design and Implementation (OSDI). 2002. 75−88. [doi: 10.1145/844128.844136]

[17] Havelund VK, Brat G, Park S, Lerda F. Model checking programs. Automted Software Engineering (ASE), 2003, 10(2): 203−232. [doi: 10.1023/A:1022920129859]

[18] Rakamarić Z, Emmi M. SMACK: Decoupling source language details from verifier implementations. In: Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Cham: Springer, 2014. 106−113. [doi: 10.1007/978-3-319-08867-9_7]

[19] Clarke EM, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2988, Springer, 2004. 168−176. [doi: 10.1007/978-3-540-24730-2_15]

[20] Gadelha MR, Monteiro F, Cordeiro L, *et al*. ESBMC v6.0: Verifying C programs using *k*-induction and invariant inference. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2019. 209−213.

[21] Ball T, Levin V, Rajamani SK. A decade of software model checking with SLAM. Communication of ACM, 2011, 54(7): 68−76.

[22] Tóth T, Hajdu Á, Vörös A, *et al*. Theta: A framework for abstraction refinement-based model checking. In: Proc. of the 2017 Formal Methods in Computer Aided Design (FMCAD). IEEE, 2017. 176−179. [doi: 10.23919/FMCAD.2017.8102257]

[23] Steensgaard B. Points-to analysis in almost linear time. In: Proc. of the Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 1996. 32−41. [doi: 10.1145/237721.237727]

[24] Oh H, Heo K, Lee W, Lee W, Yi K. Design and implementation of sparse global analyses for C-like languages. In: Proc. of the 33th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2012. 229−238. [doi: 10.1145/1273442.1250789]

[25] Sui Y, Xue J. Value-flow-based demand-driven pointer analysis for C and C++. IEEE Trans. on Software Engineering, 2018, 46(8): 812−835. [doi: 10.1109/TSE.2018.2869336]

[26] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. In: Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM, 2007. 480−491.

[27] Shi Q, Xiao X, Wu R, *et al*. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2018. 693−706. [doi: 10.1145/3192366.3192418]

[28] Shi Q, Yao P, Wu R, *et al*. Path-sensitive sparse analysis without path conditions. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI). 2021. 930−943. [doi: 10.1145/3453483.3454086]

[29] Beyer D, Löwe S. Explicit-state software model checking based on CEGAR and interpolation. In: Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE). Berlin, Heidelberg: Springer, 2013. 146−162. [doi:10.1007/978-3-642-37057-1_11]

[30] Sui Y, Xue J. SVF: Interprocedural static value-flow analysis in LLVM. In: Proc. of the 25th ACM Int'l Conf. on Compiler Construction (CC). 2016. 265−266. [doi: 10.1145/2892208.2892235]

[31] Lattner C, Adve V. LLVM: A compilation framework Forlifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). 2004. 75−86. [doi: 10.1109/CGO.2004.1281665]

[32] de Moura LM, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2008. 337−340. [doi: 10.1007/978-3-540-78800-3_24]

[33] Int'l competition on software verification (SV-COMP). http://sv-comp.sosy-lab.org

[34] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2007, 42(6): 278–289. [doi: 10.1145/1273442.1250766]

[35] De Line R, Leino KRM. BoogiePL: A typed procedural language for check-ing object-oriented programs. Technical Report, MSR-TR-2005-70, Microsoft Research, 2005.

附中文参考文献:

[8] 张协力, 祝跃飞, 顾纯祥, 陈熹. C2P: 基于Pi演算的协议C代码形式化抽象方法和工具. 软件学报, 2021, 32(6): 1581–1596. http://www.jos.org.cn/1000-9825/6238.htm [doi: 10.13328/j.cnki.jos.006238]

于银菠(1991－), 男, 博士, 副教授, CCF专业会员, 主要研究领域为软件与系统安全, 物联网安全, 深度学习安全.

刘家佳(1984－), 男, 博士, 教授, 博士生导师, 主要研究领域为网络安全, 智能通信.

慕德俊(1963－), 男, 博士, 教授, 博士生导师, 主要研究领域为硬件安全, 机器学习, 数据挖掘.