软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2061−2081 [doi: 10.13328/j.cnki.jos.006569]

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

开源 C/C++静态软件缺陷检测工具实证研究*

李广威[1,2], 袁 挺[1,2], 李 炼[1,2]

[1](计算机体系结构国家重点实验室(中国科学院 计算技术研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 李炼, E-mail: lianli@ict.ac.cn

摘 要: 软件静态缺陷检测是软件安全领域中的一个研究热点. 随着使用 C/C++语言编写的软件规模和复杂度的逐渐提高, 软件迭代速度的逐渐加快, 由于静态软件缺陷检测不需要运行目标代码即可发现其中潜藏的缺陷, 因而在工业界和学术界受到了更为广泛的关注. 近年来涌现出大量使用软件静态分析技术的检测工具, 并在不同领域的软件项目中发挥了不可忽视的作用, 但是开发者仍然对静态缺陷检测工具缺乏信心. 高误报率是 C/C++静态缺陷检测工具难以普及的首要原因. 因此, 选择现有的较为完善的开源 C/C++静态缺陷检测工具, 在 Juliet 基准测试集和 37 个良好维护的开源软件项目上对特定类型缺陷的检测效果进行了深入研究, 结合检测工具的具体实现, 归纳了导致静态缺陷检测工具产生误报的关键原因. 同时, 通过研究静态缺陷检测工具的版本迁移轨迹, 总结出了当下静态分析工具的发展方向和未来趋势, 有助于未来静态分析技术的优化和发展, 从而实现静态缺陷检测工具的普及应用.

关键词: 软件缺陷检测; 静态分析; 软件质量保证; 源码安全

中图法分类号: TP311

中文引用格式: 李广威, 袁挺, 李炼. 开源 C/C++静态软件缺陷检测工具实证研究. 软件学报, 2022, 33(6): 2061–2081. http://www.jos.org.cn/1000-9825/6569.htm

英文引用格式: Li GW, Yuan T, Li L. Study of State-of-the-art Open-source C/C++ Static Analysis Tools. Ruan Jian Xue Bao/ Journal of Software, 2022, 33(6): 2061–2081 (in Chinese). http://www.jos.org.cn/1000-9825/6569.htm

Study of State-of-the-art Open-source C/C++ Static Analysis Tools

LI Guang-Wei[1,2], YUAN Ting[1,2], LI Lian[1,2]

[1](State Key Laboratory of Computer Architecture (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Static software defect detection is an active research topic in the domain of software engineering and software security. Along with the increase of software complexity and size, static software defect detection has been applied in both industry and academy to take the benefit of finding defects in C/C++ programs without execution. A large amount of static analysis tools (SATs) for C/C++ have been developed in recent years, and have played an important role in automatically finding defects in various kinds of C/C++ software projects. In spite of this, developers are still having less confidence on SATs mainly due to the high false positive rate that has been an unsolved problem for a long time. This research dives deep into state-of-the-art static analysis tools for C/C++ and figures out why false positives are raised through the approach of running them on Juliet Test Suite and 37 open-source real-world software projects. With insight of the design and implementation details of the selected open-source SATs, the exact reasons of which result in the high false positive rateare found. Moreover, the effort is also made to trace the tendency of development and the future of state-of-the-art open-source C/C++ SATs.

Key words: software defect detection; static analysis; software quality assurance; source code security

* 基金项目: 国家自然科学基金(61802368, 62090024)

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: 2021-09-05; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

在当下信息化社会中, 大量定制化的 C/C++软件系统被部署并用于取代人工服务. 随着软件规模和复杂性的增长, 传统的软件测试技术已经无法满足日益增长的对软件安全性的需求, 使得这些复杂的 C/C++软件系统中不可避免地存在影响其正常运行的各种缺陷和错误.

据瑞星发布的《2020 年中国网络安全报告》[1]指出, 2020 年攻击频率最高的 10 个 CVE[2]漏洞中有 4 个是系统软件漏洞, 5 个是常用软件(Microsoft Office/Adobe Acrobat)漏洞, 同时, 其中有 5 个是由于 C/C++堆栈溢出(buffer overflow)引起的. 由 C/C++语言编写的基础类库和运行时环境被各式各样的系统使用, 这些类库的缺陷造成的错误传播到各种上层应用从而造成了连锁崩溃, 成为了恶意代码攻击的标靶.

C/C++语言本身的复杂性, 导致了在使用其编写的软件中可能存在大量不同种类的缺陷. 我们在表 1 中列出了 18 种主要和 C/C++相关的 CWE (common weakness enumeration)缺陷编号, 并将其分为 7 类, 涵盖了常见的安全缺陷和编码规范缺陷. 通过查询 NVD[3]缺陷数据库, 我们统计了其中与 C/C++有关的 CWE 缺陷分类自 2010 年起近 10 年来每年的相应数量及其变化趋势. 如图 1 所示, 缓冲区溢出相关缺陷报告(CWE119、CWE125)在 10 年间占比超过 50%, 但是占比有逐渐下降的趋势. 相对而言, 释放后使用(CWE416)和空指针解引用(CWE476)缺陷报告数量逐年提升. 从 2020 年起, NVD 缺陷数据库中 C/C++相关缺陷记录中 80.4%是内存安全缺陷的.

表 1 C/C++相关缺陷 CWEID

CWEID	CWE 名称	缺陷类别
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	Buffer overflow
CWE-121	Stack-based Buffer Overflow	
CWE-122	Heap-based Buffer Overflow	
CWE-124	Buffer Underwrite	
CWE-125	Out-of-bounds Read	
CWE-126	Buffer Over-read	
CWE-127	Buffer Under-read	
CWE-190	Integer Overflow or Wraparound	Integer overflow
CWE-191	Integer Underflow (Wrap or Wraparound)	
CWE-401	Missing Release of Memory after Effective Lifetime	Memory leak
CWE-415	Double Free	Use after free
CWE-416	Use After Free	
CWE-457	Use of Uninitialized Variable	Uninitialized variable
CWE-824	Access of Uninitialized Pointer	
CWE-476	NULL Pointer Dereference	Null pointer dereference
CWE-690	Unchecked Return Value to NULL Pointer Dereference	
CWE-561	Dead Code	Dead code
CWE-563	Unused Variable	

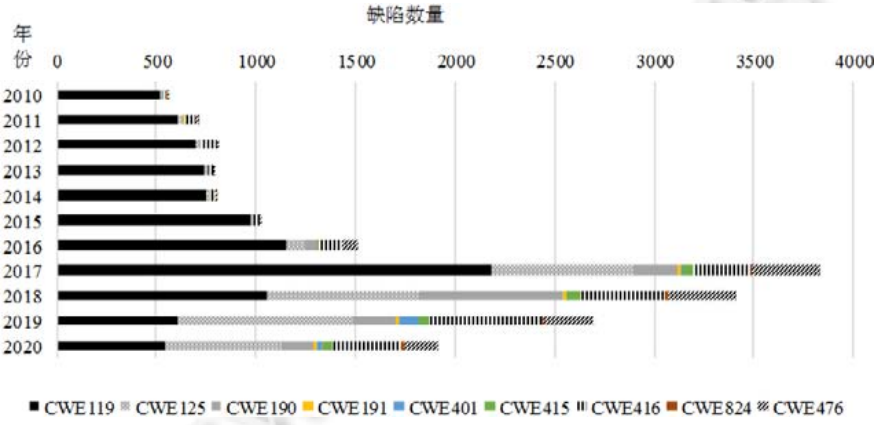

图 1 NVD 数据库中的 C/C++相关缺陷

程序的规模和复杂性进一步使得 C/C++语言中的缺陷深度隐藏于复杂的程序结构和上下文中. 以 C/C++语言的内存安全相关缺陷为例, 诸如 Use-after-free、Double-free、Memory-leak、Null-pointer-dereference 等, 这

些缺陷发生的根本原因是不正确的内存操作. 然而程序的模块化程度和抽象设计的层次越高, 程序越难以理解, 造成在开发和测试过程中, 对于指针的来源和去向也更难以界定. 在有限的程序上下文中, 仅靠人力是很难断定内存操作是否是违规的. 凡此种种, 最终导致了内存安全缺陷难以及时发现和修复, 从而成为软件中潜藏的安全隐患. 即使部分大型库中提供了一系列的内存管理的基础设施, 以此来减轻用户管理内存的复杂度, 但此举同时也增加了检查内存错误的难度.

对软件安全性日益高涨的需求, 使得传统软件测试技术捉襟见肘. 相比于动态测试, 静态缺陷检测由于其可在不实际执行程序的前提下进行分析和检测, 对检测环境要求低, 覆盖率高, 且可以在软件开发的早期发现和定位缺陷, 从而降低了软件缺陷引发的开发和维护成本. 即使如此, 静态缺陷检测工具仍然未能获得开发者足够的重视和认同. 文献[4]针对静态工具的调研中, 66%的项目配置了如何使用静态检测工具, 仅仅37%的项目要求参与者的代码提交必须通过静态工具检测, 相比于 100%的测试通过率要求, 这一比例仍然很小. 无论是开源社区还是工业界开发者, 都对静态检测工具缺乏信心. 2021 年, Linux 内核社区信任危机[5]爆发的导火索也是某静态工具产生的相关补丁. 高误报率是阻碍静态缺陷检测工具广泛应用的主要原因[6], 具体地, 对于使用 C/C++语言编写的软件而言, 语言本身的复杂性、软件实现中的逻辑复杂性、软件规模等, 都是直接制约静态检测工具精度提升的重要影响因素.

本文的主要贡献: 我们通过研究具有代表性的开源 C/C++静态缺陷检测工具在 Juliet 测试套件[7]和 37 个开源项目上的表现, 从易用性、可用性、报告精度、报告可理解性这 4 个方面进行了评价对比, 得到以下关键结论.

(1) 当前流行的静态分析工具提供的都是主流操作系统上的预构建版本或者自动构建脚本, 都存在官方或者第三方提供的预构建 docker 容器镜像, 都提供了足够的文档来说明如何使用工具, 但是检测工具对输入的额外要求, 对工具的使用造成了一定的阻碍;

(2) 在至少 89.2%的项目上, 这些工具都能在 1 小时内给出项目的缺陷报告, 尤其是对于小于 20 万行的开源项目, 它们都能在 1 小时内消耗少于 32 GB 内存的情况下完成检测;

(3) 通过人工检查超过 1 500 个缺陷报告之后, 我们发现 C/C++静态缺陷检测工具给出的缺陷报告往往具有极高的误报率, 比如释放后使用和缓冲区溢出这两类缺陷的误报率都超过 90%;

(4) 开源 C/C++静态缺陷检测工具给出的缺陷报告的可理解性较差, 缺陷报告中包含的具有提示作用的信息极少, 难以帮助开发者确定报告是否是误报, 同时也难以快速定位缺陷发生的原因. 即使部分工具可以给出函数内的一些关键路径, 但仍无法为复杂环境中缺陷的分析和定位提供足够的信息.

同时, 本文实验中通过对开源静态缺陷检测工具进行源码级插桩的方式, 针对静态缺陷检测工具产生误报和漏报的原因进行了详细的分析和归纳.

本文第 1 节讨论现有的程序静态检测方法及工具, 并详细介绍本文所选取的 3 款开源 C/C++静态缺陷检测工具. 第 2 节讨论我们所设计的实验细节, 包括实验流程、实验环境设置、测试集和实验评价方法和指标. 第 3 节对实验结果进行分析, 从性能、精度、报告的可理解性等方面对 3 款静态缺陷检测工具进行评价. 第 4 节对静态检测工具产生误报或者漏报的原因进行进一步的讨论. 第 5 节结合代码的演化历史总结归纳静态检测工具的发展方向和趋势并总结全文.

1 程序缺陷检测

自动化程序缺陷检测技术[8]在 20 世纪 60 年代被应用于检测实际程序中存在的缺陷, 过去的半个世纪中, 这些技术随着计算机软硬件的发展, 从基于语法规则到使用逻辑推断语义的算法, 在不断的融合改进中检测能力逐步提升.

1.1 静态程序分析方法

静态分析[9,10]是一种通过在源代码上静态地对程序状态、数据传播等信息进行计算分析, 而不执行目标代码的分析方式. 静态分析的优点在于能够专注于代码本身, 避免程序运行时所需求的复杂环境, 同时利用对

于程序的抽象建模简化复杂语义, 可以获得较高的分析效率和覆盖率. 静态分析方法依靠其不依赖程序输入和运行环境便能获得程序固有属性信息的优势, 在编译优化、程序验证和程序理解领域发挥着巨大的作用.

数据流分析方法使用数据流分析[11]或者其变种(比如静态污点分析[12])来计算程序的变化状态在程序中的传播, 从而监测特定位置程序的行为和状态. 经典的数据流分析是建立在程序的控制流图及表示所分析抽象信息的格(lattice)上的不动点算法, 通过对控制流图上每个节点计算相应的数据流约束(dataflow constraint), 从而获得相应的数据流事实(dataflow facts); 同时, 算法为了获得唯一最小解要求格是有限高度的, 并且表示数据流约束的数据流方程在格上是单调的.

模型检测(model checking)[13]方法通过对程序进行形式化建模并探索模型的每一个状态, 从而验证一个特定的程序属性是否可以被满足. 常见的建模方法包括有穷状态机、时序迁移系统、异步消息传递模型和分布式共享内存模型等. 然而对于复杂的现实语言, 例如 C/C++、复杂的软件系统, 如浏览器等, 即使通过针对性地对程序局部属性建模、简化特定语言语义等方法进行优化, 模型检测方法仍然面临着建模复杂、搜索状态空间爆炸的问题.

抽象解释(abstract interpretation)[14−16]则是对程序指令进行抽象建模, 通过 Galois 链接用抽象域表达对具体域的可靠近似. 抽象解释提供了严格的理论约束来通过不动点算法求解获得程序的最小不动点, 从而保证其结论的可靠性. 抽象解释将模型检测中的状态穷举问题转变为采用近似优化的抽象域不动点求解, 在一定程度上缓解了状态空间爆炸的问题.

静态分析方法的缺点也非常明确: 由于分析中大量采用的近似(approximatation)处理的技术手段, 因此静态分析技术通常会受到高误报率的困扰.

1.2 静态缺陷检测工具

近年来, 无论是学术界还是工业界, 都涌现出了大量的针对不同语言、不同缺陷类型的静态缺陷检测工具, 例如: 针对性检测 SQL 注入、跨站脚本(XSS)的 PHP 语言静态缺陷检测工具 RIPS[17]、Pixy[18]和 phpSAFE[19]等; 针对传统软件的 Flawfinder[20]、FindBugs[21]、PMD[22]等. 而对于 C/C++编写的软件中存在的缺陷, 存在 CppCheck[23,24]、Frama-C[25]、Oclint[26]、Splint[27]等大量针对性的开源或者免费工具. 同时也有 SonarQube[28]、CheckMarx[29]、Coverity[30]、Fortify SCA[31]等商业源码安全扫描软件, 相比于开源软件, 通常它们会提供丰富的支持文档和更好的用户体验. 为了深入追踪和分析静态缺陷检测工具给出误报或者漏报的具体原因, 我们选择开放源码的静态缺陷检测工具为主要研究目标, 从而允许我们修改工具本身的源码, 通过插桩等技术手段获得更加详细的缺陷相关信息, 例如静态缺陷检测工具分析过程中对程序状态属性的记录信息等.

由于工具繁杂, 我们兼顾下述 3 个标准选择研究目标.

1) 来源: 工业界与学术界;
2) 维护者: 商业公司和开源社区;
3) 被广泛接受且开放的.

我们选取了 FaceBook Infer[32]、Clang Static Analyzer[33]和 SVF Saber[34]这 3 款开源静态缺陷检测工具在 C/C++程序上进行对比, 以评估当前受欢迎的工具的优势及其不足之处. FaceBook Infer 和 SVF Saber 都使用 LLVM[35]的 Bitcode IR (intermeidate representation)作为统一中间表示, 来降低检测方法与特定语言特性的耦合性.

1.2.1 FaceBook Infer

FaceBook Infer (简称为 FbInfer)在 2013 年作为一个学术研究项目被开发出来, 它采用抽象解释(abstract interpretation)技术作为检测多种语言、多种缺陷类型的通用支撑平台. 对于 C/C++缺陷的检测中 FbInfer 使用到了基于分离逻辑(separation logic)[36]的双向假推理(bi-abduction)[37]技术. 该工具在 Facebook 内部广泛使用, 是一款成熟的工业界开源工具.

分离逻辑是一种专门设计用于推断程序内存结构变化的数学逻辑, 通过引入逻辑链接操作符(*)来表示程

序分配的堆栈空间中的分离链接(separating conjuntion)关系. 例如, ($x\mapsto y * y\mapsto x$)可以精确地表示程序分配的两个内存区域及它们之间的指向(points-to)关系: 第 1 个内存区域地址为(x)、内容为(y)的值, 第 2 个内存区域地址为(y)、其内容为(x)的值, 而分离链接操作符(*)表示这两个内存区域是相互分离的.

双向假设推理则是将分离逻辑进行扩展, 以支持自动局部推理. 例如: 当目的是要验证 x 是否为 $NULL$ 时, 双向假设推理提出一个局部假设: ($empty * ?antiframe \vdash x\mapsto NULL * ?frame$), 其中, $antiframe$ 表示分离逻辑验证中需要求解的缺失状态, 而 $frame$ 表示验证目标($x\mapsto NULL$)所在代码区域的不变状态. 在不考虑具体上下文限制的情况下, 对这个例子而言, 通过选取($antiframe=x\mapsto NULL$), ($frame=empty$)可以满足该公式的要求, 从而保证($x\mapsto NULL$)假设被验证为真.

双向假设推理可以采用分治策略将全局推理分解成局部推理, 从而实现跨过程分析. 对于每个函数调用点, 将函数调用表示为一个霍尔逻辑三元组($\{pre\}f(\cdot)\{post\}$), 其中, $f(\cdot)$表示对函数 f 分析获得的规格概要(specification). 最终通过形式化推理选择正确的程序状态, 从而保证对函数 f 的调用是正确的.

在 Facebook 收购了 Infer 之后, 其发展重心转移为移动端程序, 专注于资源泄露和敏感信息泄露等移动应用中常见的性能和安全问题. 同时采用了抽象解释技术, 以提升 Infer 本身作为分析框架的通用性和扩展性.

1.2.2 Clang Static Analyzer

Clang Static Analyzer (简称 ClangSA)是在流行编译工具链 LLVM 上构建并集成于 Clang 中的针对 C/C++和 Objective-C 源代码的静态分析工具, 该工具被大量开源社区和项目所使用, 同时被多款 IDE 集成, 从而为开发者提供快速可用的静态缺陷检测能力. ClangSA 提供在 AST 上对代码进行的语法及语义检查, 涵盖检查内存安全属性、死代码等功能. ClangSA 的主要实现技术是静态符号执行(static symbolic execution)[38,39], 由负责遍历控制流图控制计算路径的 CoreEngine 和负责计算程序状态属性的 Checkers 组成. ClangSA 的 CoreEngine 实现基于 Worklist 的广度优先搜索(breadth-first search)算法, 因此具有较高的时间和空间效率, 从而达到 ClangSA 快速轻量的设计目标, 并提供了良好的扩展性.

ClangSA 是一个随 Clang 和 LLVM 版本迭代更新发布, 并处于持续化开发中的项目. LLVM 提供了 scan-build、scan-view 实用工具, 用以调用 ClangSA 和展示缺陷报告; 同时, ClangSA (ClangTidy)可以作为外部插件被其他工具集成, 例如苹果公司的 XCode IDE 工具.

1.2.3 SVF Saber

SVF 是使用稀疏流图技术的指针分析框架, 为工业界和科研界提供基础的指针分析实现. SVF Saber (简称 Saber)是在其指针分析基础上实现的一个实验性的缺陷检测工具, 最初只支持检测 C 语言中的内存泄漏(memory leak)缺陷, 随着对其指针分析算法的不断完善, 形成了著名的开源指针分析框架 SVF[40]. 稀疏值流分析(sparse value-flow analysis)[34]是当前学术界最新的高效而精确的静态指针分析算法, 为了规避传统指针分析算法对任意代码位置指针的冗余分析产生的巨大代价, 稀疏值流分析通过 Def-Use 链直接传递指针地址, 从而使指针分析的结果表示和计算过程稀疏化. 通常, 稀疏值流分析算法包括两个阶段: (1) 通过不精确的快速指针分析构建 Def-Use 链; (2) 通过上一步获得的 Def-Use 链精化指针分析结果.

Saber 的检测过程由 4 部分组成.

(1) 预分析: 使用流不敏感、上下文不敏感、域敏感的 Anderson 指针分析算法[41]获得程序指针及其别名信息;

(2) 构建完全稀疏的静态单赋值(full-sparse SSA): 将程序转换成静态单赋值形式, 并对调用点、指针的 load/store 操作等对指针的间接访问进行映射, 以连接跨过程间的 Def-Use 传递关系;

(3) 构建稀疏值流图(sparse value-flow graph, SVFG): 将程序中所有的由 Def-Use 链及赋值产生的指针相关值流通过抽象语义构建为一个稀疏值流图, 同时按需记录每条值流边上的分支条件;

(4) 缺陷检测: 将缺陷统一建模为 Source-Sink 模型之后, 通过在稀疏值流图上执行可达性算法, 判断模型是否可被满足.

Saber 的优势在于其基础指针分析算法高效而准确, 可以减少由于指针别名关系不准确造成的冗余检测,

消减误报的数量; 同时, 其高效性让Saber具有了全程序地分析百万行规模项目的能力. Saber当前实现了内存泄漏和内存重复释放(double free)缺陷, 作为SVF平台上的验证性工具实现, 并未针对常见工程的缺陷模式进行优化, 因此仍然存在误报率高的问题, 其优化发展方向是轻量化和可伸缩性, 因此引入了按需的指针分析算法, 从而进一步降低了指针分析的资源消耗.

2 实 验

2.1 实验流程

实验流程如图 2 所示,共分为 4 步.

(1) 构建和配置缺陷检测工具: 本文实验所选择的开源检测工具都提供了多平台上的构建指南, 同时, FbInfer/ClangSA 都有官方提供的预编译版本, Saber 提供了自动构建的脚本. 实验中为了降低对测试集中项目构建过程的干扰, 优先选择使用预构建版本;

(2) 针对实验环境构建测试集: 虽然开源项目一般都提供了配置和构建相关的指导文档和自动化脚本, 但是针对特定系统版本或者依赖库版本, 仍然需要手工地进行部分修改才能使用 Clang 编译成功. Saber 要求项目被编译成 Bitcode 文件作为输入;

(3) 缺陷检测: 使用静态缺陷检测工具检测配置好的软件项目或者项目产生的 Bitcode 文件;

(4) 收集和检查缺陷报告: 每款缺陷检测工具提供的缺陷报告格式和内容相差很大, 因此需要有针对性地处理和过滤. 在此过程中, 我们对缺陷报告进行随机抽样并人工检查报告是否是误报; 然后将统一格式化的数据存入报告数据库中, 以待进一步分析.

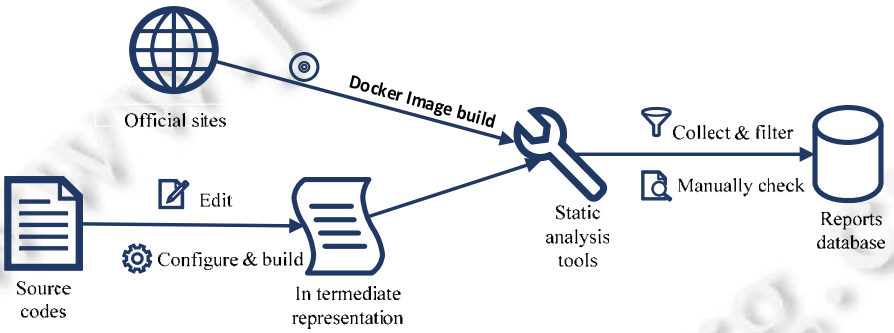

图 2 实验流程

我们编制了一套 Python 脚本, 以自动化地完成上述实验流程中除去人工检查缺陷报告外其他的工作. 对于缺陷报告的确认, 我们通过具有熟练 C/C++开发经验和程序分析背景的数个人员, 独立地通过人工阅读和分析相关源码来确认报告是否是误报, 对于其中怀疑可能是误报的缺陷报告, 通过在静态缺陷检测工具中插桩的方式, 打印出更多的程序属性信息, 以此辅助判断缺陷报告是否是误报. 对于确认的误报, 需要进一步分析和归纳静态缺陷检测工具给出误报的具体原因. 最终, 通过汇总后交叉比对获得交集作为最终的实验结果.

2.2 实验环境

本文中所有实验都在拥有 Intel(R) Xeon(R) Gold 6230@2.10 GHz CPU*2 和 512 GB 内存的 Linux 平台上的 docker 容器里进行. 用于编译和运行静态检测工具的基础容器选用包含最新更新的 Ubuntu 20.04 (GCC 版本 9.3.0), 由于部分开源程序编译需系统特权, 因此 docker 容器的启动命令中加入了--security-opt seccomp=unconfined, 以屏蔽其安全策略.

2.3 配置缺陷检测工具

本文所选取的 3 款工具: FbInfer、ClangSA 和 Saber 都是基于当前流行的 LLVM 编译框架的, 并且要求其检测的项目可以通过 Clang 编译. 由于 GCC 9.3 是 2019 年发布的, 默认情况下, LLVM 使用了 GCC 的头文件. 部分 GNU 来源的项目对 GCC 的部分特性和内置库有版本和兼容性要求, 因此我们选择同样在 2019 年下半年发布的 LLVM 8.0.1 作为基础平台, 以减少因为兼容性差异引发的构建问题. 由于 FbInfer 和 Saber 都提供了 Dockerfile 用于构建镜像或者第三方平台托管的 docker 镜像文件, 而 ClangSA 在常见系统, 如 Ubuntu 的官方软件仓库中提供, 因此我们不需要额外的工作就可以直接获得可用的预构建检测工具.

对于可以通过 Clang 编译的检测项目, 如表 2 中所列, FbInfer 和 ClangSA 都提供了一条可以直接接管项目构建过程的命令. 但是这条命令并不是总能正常工作, 因此 FbInfer 提供了使用编译命令数据库的方式. 在 Github 上也存在一个第三方的 scan-build 工具, 使 ClangSA 支持使用编译命令数据库方式进行分析. 对于 Saber 来说, 情况会变得比较复杂: Saber 要求的输入是程序构建链接后的 Bitcode 文件, 然而当前没有简单易用的开源方案来把整个项目构建链接成 Bitcode 文件, 因此我们只能借助于 WLLVM (whole program LLVM)[42]软件构建目标项目, 并提取 Bitcode 文件.

表 2 静态缺陷检测工具的检测命令

工具	检测命令
FbInfer	infer --[options] run -- project_build_command(make/cmake/…) infer --[options] --compilation-database compile_commands.json
ClangSA	scan-build -[options] project_build_command scan-build -[options] --cdb compile_commands.json
Saber	saber -color=false -dfree/-leak project_bc_files

2.4 测试集

我们选取了 Juliet C/C++测试套件 1.3 版(Juliet Test Suite for C/C++ 1.3)以及 37 个被广泛使用且具有代表性的开源项目作为基准测试集, 以第 1.2 节中所述的 3 款工具在这些测试集上的表现, 来评估 3 款工具在检测实际程序中缺陷上的效果, 并探究误报和漏报发生的原因.

2.4.1 Juliet C/C++测试套件

Juliet C/C++测试套件是一个包括 64 099 小规模测试程序的系统性集合, 涵盖了 118 个类型的软件弱点(weekness), 其中包括缓冲区溢出、空指针引用、未处理异常、命令行注入、死锁等常见的程序缺陷. Juliet 测试套件由 NIST (National Institute of Standards and Technology)于 2010 年首次发布, 并于 2017 年 10 月发布最新修订版本 1.3.

整个 Juliet C/C++测试套件包括 118 个类别, 由于 CWE 编号使用上存在细分和重合, 因此 NVD 数据库中的 CWE 分类数据并不能严格匹配, 比如, CWE119 和 CWE125 通常在 NVD 数据库中涵盖了 CWE121–CWE127 的所有有关缓冲区溢出(buffer overflow)的缺陷报告. 我们根据表 1 中所列的 7 类缺陷选择了 Juliet C/C++测试套件中的对应测试用例进行详细的测试和分析.

表 3 中列出了我们从 Juliet C/C++测试套件中所选择的 16 个分类的测试用例所对应的 CWEID 及其对应名称, 分组栏表示其测试用例的分组数, 不同分组中的测试用例在分支条件、调用函数、数据结构等方面有所不同, 为测试用例提供了多样性的变种. 数量栏列出了某个 CWEID 下所有测试用例中正例(存在对应缺陷)的用例数, 同时, 伴随着每个正例还提供了至少 4 个修复后的测试用例.

表 3 Juliet 测试套件中的 CWE 分类用例

ID	名称	分组	数量
CWE121	Stack_Based_Buffer_Overflow	9	3 100
CWE122	Heap_Based_Buffer_Overflow	11	3 870
CWE124	Buffer_Underwrite	3	1 168
CWE126	Buffer_Overread	2	870
CWE127	Buffer_Underread	3	1 168
CWE190	Integer_Overflow	7	3 960

表 3 Juliet 测试套件中的 CWE 分类用例(续)

ID	名称	分组	数量
CWE191	Integer_Underflow	5	2 952
CWE401	Memory_Leak	3	1 364
CWE415	Double_Free	2	818
CWE416	Use_After_Free	1	393
CWE457	Use_of_Uninitialized_Variable	2	926
CWE476	NULL_Pointer_Dereference	1	288
CWE563	Unused_Variable	1	428
CWE590	Free_Memory_Not_on_Heap	5	2 280
CWE690	NULL_Deref_From_Return	2	768
CWE761	Free_Pointer_Not_at_Start_of_Buffer	1	288

2.4.2 开源程序测试集

为了更加全面地评价静态检测工具的能力, 我们参考了静态缺陷检测相关文献[34,43−45]采用的相关测试项目和一些近年来新兴领域的开源项目, 并兼顾了著名与非著名项目、新老项目、维护规模和迭代速度等维度. 最终, 我们主要选取了以下 3 类开源项目.

(1) 经典开源项目: 通常, 它们具有较长的开发和迭代历史;

(2) 近 10 年来极度活跃的项目: 通常, 这些项目在 Github 中具有较高的星(star)数量;

(3) 长期维护中但缺乏足够部署及测试的项目: 例如 Github 中星数量较少但长期存在, 且 Issue 列表较长时间存在未解决的问题报告的项目等.

我们选取了 37 个现实世界开源应用, 并在表 4 中列出了其版本号、规模大小和来源信息. 表 4 右半部分所列项目均来自 Github, 并在最后一栏标示了项目所拥有的星数量. 这 37 个开源应用中包括 25 个传统 Linux 系统中常用的实用程序(库), 它们在现代活跃 Linux 发行版的各个分支中被广泛使用; 其余 12 个来自近年来发展迅速的机器学习、区块链和下一代互联网标准等新兴领域. 在 19 个以 Github 作为维护主站的项目中, 15 个项目的星数量超过 10 K, 最高达到 50.4 K (bitcoin). 其中, wabt (Web assembly binary toolkits)和 mlpack 是近两年来的新建项目, 而 libssh2 和 nghttp2 是长期维护的项目.

我们在选择项目代码版本时优先选择最近的发行版本, 原因在于: 我们在这些项目的代码贡献规则(contribution requirements)中没有发现任何对使用静态缺陷检测工具的推荐建议或者强制要求. 虽然这个选择策略会导致最终结果中的误报率偏高, 但更利于我们评估静态缺陷检测工具对软件测试中未发现的隐藏缺陷的检测能力.

表 4 开源项目测试集

Project	Version	Lines (K)	Source	Project	Version	Lines (K)	Stars (K)
mujs	1.0.7	15	Aftifex	libssh2	1.9.0	35	0.7
mupdf	1.17.0	339		wabt	1.0.19	52	3.3
httpd	2.4.45	309	Apache	mlpack	3.3.2	166	3.6
bzip2	1.0.6	6	GNU	nghttp2	1.41.0	90	3.7
less	551	20		memcached	1.4.28	12	10.8
sed	4.2.2	33		goaccess	1.4	51	12.9
gzip	1.6	43		shadowsocks	3.3.4	26	14.2
make	4.3	44		swoole	4.5.2	94	16.5
grep	2.25	81		mxnet	1.7.0.rc1	358	19.3
tar	1.32	83		curl	7.71.1	145	19.6
bison	3.7	104		darknet	2020.10	29	20.1
bash	4.4-rc1	113		tmux	3.1b	48	20.5
gnugo	3.8	209		vim	8.2.1328	374	23.0
coreutils	8.32	222		FFmpeg	n4.3.1	1 122	23.8
emacs	26.3	319		wrk	4.1.0	5	27.8
sendmail	8.15.2	95	proofpoint	php	7.4.8	1 171	29.6
wine	5.13	3 339	WineHQ	caffe	1.0	60	31.4
icecast	2.4.3	21	Xiph.Org	git	2.28.0	235	37.0
–				bitcoin	0.20.1	356	50.4

2.4.3 测试集构建

对于 Juliet C/C++测试套件, 我们修改官方提供的构建脚本, 在 Makefile 中使用 Clang 替代 GCC, FbInfer 和 ClangSA 可以自动完成编译命令的替换, 而 Saber 需要我们使用 Clang 的-emit-llvm 参数产生 Bitcode 文件

作为其输入. 由于各种不同 Linux 发行版之间二进制和库无法完全兼容, 即使我们选择的所有开源项目都是在 Linux 系统上开发的, 但在我们的实验环境所选的 Ubuntu 20.04 上仍然需要一些额外选项或者修改才能编译通过. 表 5 中所列举的 20 个项目需要额外的配置步骤, 修改配置文件或者需要在最终构建命令上附加额外宏参数等. 其中, bear 是一个 Linux 实用工具, 通过 LD_PRELOAD 或者 DYLD_INSERT_LIBRARIES 动态获取构建命令, 并记录产生 JSON 格式的编译命令数据库.

表 5 开源项目构建过程中需要做的修改

项目	Configure 步骤	构建修正方法
bitcoin	./autogen.sh && ./configure --with-incompatible-bdb	remove -fstack-reuse=none in configure.ac
curl	./buildconf && ./configure	–
httpd		
php	./buildconf --force && ./configure --disable-phar	replace *HANDLE_EXCEPTION_LEAVE*(·)/*ZEND_VM_LEAVE*(·) with return; in Zend/zend_vm_execute.h
memcached	./configure	remove -Werror in CFLAGS in configure.ac
vim	./configure FORCE_UNSAFE_CONFIGURE=1	–
shadowsocks	./configure --disable-documentation	
wine	./configure --enable-win64	
coreutils	./configure FORCE_UNSAFE_CONFIGURE=1	
gzip	autoreconf && ./configure	make CFLAGS+="-DSLOW_BUT_NO_HACKS -D_IO_ftrylockfile -D_IO_IN_BACKUP=1"
nghttp2	autoreconf -i && ./configure	–
tar	autoreconf -force -I && ./configure	
mlpack	cmake	
wabt	cmake -DCMAKE_BUILD_TYPE= Debug-DBUILD_TESTS=OFF -DCMAKE_EXPORT_COMPILE_COMMANDS=ON	
caffe	cmake -DCMAKE_EXPORT_ COMPILE_COMMANDS=ON	set CPU_ONLY=ON, BUILD_python=OFF, BUILD_docs=OFF, BUILD_python_layer=OFF in CMakeLists.txt
mxnet	cmake -DUSE_MKLDNN=0 -DUSE_CUDA=0 -DCMAKE_EXPORT_COMPILE_COMMANDS=ON	–
swoole	phpize && ./configure --enable-openssl --enable-sockets --enable-mysqlnd --enable-http2	disable HAVE_ASM_GOTO in php_config.h
darknet	–	bear - - make
wrk		repair WITH_LUAJIT/WITH_OPENSSL used in CFLAGS and LDFLAGS to system /usr in Makefile.
mupdf		make USE_SYSTEM_LIBS=yes USE_SYSTEM_LCMS2=yes

2.5 实验分析方法与指标

这 3 款工具并未使用统一的缺陷分类方式且各有所长: FbInfer 专注内存安全问题, 尤其是资源泄露; 而 ClangSA 的设计目的是更快、更早地发现程序局部中存在的缺陷, 其所用技术较为简洁, 专注于统一化的计算框架; Saber 则不同于这两者, 其目的在于验证实验性的新技术和新方案所具有的检测能力和性能水平.

在 JulietC/C++测试套件上, 我们选择下面 3 个统计指标来比较 3 款开源工具的表现.

(1) 召回率(recall rate, RCR): 正确报告数与测试用例中正例总数的比值. 这个指标表示了在测试用例集合上检测工具的检测能力的大小——召回率越高, 检测工具对特定缺陷的检测能力越高;

(2) 误报率(false positive rate, FPR): 错误报告数占工具所有报告总数的比值. 这一指标表示检测工具在给出报告时可能发生错误的几率. 误报率也是衡量缺陷检测工具精度的主要指标;

(3) 共识率(consensus rate, CR): 在同一个测试集合运行多款检测工具时, 两款工具同时报告了同一个缺陷, 那么我们认为这两款工具就这一个缺陷达成了共识. 共识率就是某款工具所有报告中与任意一款其他工具达成共识的报告数与其报告缺陷总数的比值, 该指标是一个极性指标——共识率越高, 表示工具对常见缺陷的检测效果越好; 共识率越低, 表示工具试图检测更复杂或者不常见的缺陷. 使用该指标的原因在于: 实践中, 大多数项目同时采用多款不同的静态缺陷检测工具, 通常开发者会优先处理被多个工具同时报告的缺陷.

在开源测试集上, 我们通过统计不同项目上每个工具的内存和时间消耗来评价其默认检测配置下的通用性能. 而对于缺陷报告, 我们仅通过部分缺陷类别上所产生的报告数量及随机抽样的误报率来估计对应工具所具有的检测能力. 由于本文所选取的 3 款工具都要求测试项目至少可以通过编译阶段, 而数量众多的不同时期的项目历史版本对构建环境的要求较为复杂, 使得我们无法使用已发现的历史缺陷来进行实验, 所以对于开源项目测试集上的结果, 我们只能通过抽样人工检验的方式来确定检测工具的误报率.

3 实验结果分析

经过检索检测工具的相关文档, 我们在实验中对 FbInfer 启用了工具默认的检测器的同时, 对于 FbInfer, 我们额外启用了 pulse 检测器选项, 以有针对性地检测内存安全相关缺陷. 如图 3 所示. 尽管 FbInfer 的文档中列举了多达 139 个缺陷报告类型, 我们在实际中只收集到 22 种缺陷对应的报告. 同样地, 我们对 ClangSA 启用了其所支持的 49 种检测器中的 35 种(包括默认启用的 32 种和我们额外指定的 3 个测试(alpha)检测器), 产生了 32 个不同细分类别的缺陷报告. 对于 Saber, 则运行了它支持的 3 种检测类型内存泄漏、资源泄露和释放后使用(在 Saber 的文档中, 这个选项名称为-dfree, 描述为重复释放检测. 然而令人困惑的是, 它所给出的报告与释放后使用更加匹配). 我们在表 6 中详细列举了实验中对各个检测工具启用的检测器(或选项), 其中, FbInfer 检测器标注了对应缺陷类型的报告细分类型数量; CalngSA 的不同检测器可能给出同一细分类型的缺陷报告, 由于其交叉关系复杂, 因此只列举了检测器; 其中, SVF Saber 的 uaf 选项在实际中是无效的.

表 6 检测工具所支持的检测器分类

工具	FbInfer	ClangSA	SVF Saber
缺陷分类	检测器(类型数)	检测器	检测选项
Buffer Overflow	bufferoverrun (7)	alpha.security.ArrayBoundV2 alpha.unix.cstring.OutOfBounds cplusplus.NewDelete	–
Integer Overflow	bufferoverrun (9)	alpha.security.MallocOverflow	
Memory Leak	pulse (1) biabduction (1)	unix.Malloc cplusplus.NewDeleteLeaks	-leak
Use After Free	pulse (2) biabduction (1)	unix.Malloc cplusplus.NewDelete cplusplus.Move	-uaf -dfree
Uninitialized Variable	pulse (1) uninit (1)	core.uninitialized.*(5) valist.Uninitialized core.CallAndMessage	–
Null Pointer Dereference	pulse (1) biabduction (3)	core.NullDereference core.NonNullParamChecker unix.cstring.NullArg core.CallAndMessage	
Dead Code	bufferoverrun (1) liveness (1)	alpha.deadcode.DeadStores	
Divide Zero	biabduction (1)	core.DivideZero	
Resource Leak	biabduction (1)	–	-fileck
Stack Variable Address Escape	pulse (1)	core.StackAddressEscape	–
Undefined Behavior	–	core.UndefinedBinaryOperatorResult	
Insecure API		security.insecureAPI.*(10)	
API Misuse		unix.API unix.MallocSizeof unix.MismatchedDeallocator unix.Vfork	

图 3 检测工具所支持检测的缺陷类型

3.1 Juliet C/C++测试套件

对于 Juliet C/C++测试套件, 表 7 中列出了我们在 16 个 CWE 缺陷分类上运行 3 款检测工具获得的结果, 其中, TP (true positve)表示正报, FP (false positve)表示误报, RCR 表示召回率(%), FPR 表示误报率(%), CR 表示共识率(%). 特殊符号(–)表示此项缺省, 原因是数据无意义或者不存在.

表 7 在 Juliet 测试套件上的实验结果

工具	FbInfer					ClangSA					SVF Saber				
CWEID	TP	FP	RCR	FPR	CR	TP	FP	RCR	FPR	CR	TP	FP	RCR	FPR	CR
CWE121	802	81	25.9	9.2	9.4	180	0	5.8	0	46.1					
CWE122	1887	176	48.8	8.5	4.5	100	0	2.6	0	92.0					
CWE124	196	110	16.8	35.9	21.9	171	42	14.6	19.7	31.5					
CWE126	300	24	34.5	7.4	7.1	25	0	2.9	0	92.0			–		
CWE127	196	110	16.8	35.9	21.9	171	42	14.6	19.7	31.5					
CWE190	610	101	15.4	14.2	–										
CWE191	618	409	20.9	39.8	–			–							
CWE401	320	250	23.5	43.9	52.6	490	192	35.9	28.2	49.1	520	750	38.1	59.1	32.7
CWE415	125	0	15.3	0	84.0	105	0	12.8	0	95.2	215	547	26.3	71.8	19.0
CWE416	0	0	0	–	–	18	0	4.6	0	100	105	193	26.7	64.8	6.0
CWE457	108	224	11.7	67.5	27.1	333	189	36	36.2	17.2					
CWE476	210	0	72.9	0	64.8	151	0	52.4	0	90.1					
CWE563	150	16	35	9.6	81.9	136	0	31.8	0	100			–		
CWE590			–			46	0	2	0	–					
CWE690	525	0	68.4	0	–			–							
CWE761			–			150	0	52.1	0	92	240	324	83.3	57.4	18.1

由表 7 中数据所示: FbInfer 在 16 类中有两类缺陷不支持, 同时它在 CWE416 (释放后使用缺陷)上并未给出任何报告, 与文档中的描述不符, 推测可能是版本缺陷造成的; ClangSA 在 16 类中有 3 类缺陷不支持, 部分缺陷类别给出的报告数稀少; Saber 文档中记述它支持双重释放和内存泄漏两种缺陷类型的检测, 然后实际上它给出的缺陷报告覆盖了 4 类缺陷.

FbInfer 和 ClangSA 在不同的缺陷分类上的召回率各有优劣, 表明它们各自不同的检测针对性; 同时, ClangSA 在误报率上普遍比 FbInfer 要低, 反映了它们在检测能力和误报率之间所采取的不同折衷策略.

实验结果表明:

- ClangSA 在给出缺陷报告的 13 类缺陷中有 9 类缺陷的误报率为 0, 这表明, 作为一个随编译工具链发布的检测工具, 其主要定位是让开发者更早地识别和修复那些普遍且简单的缺陷, 并极力降低误报带来的开销; 同时, 这一点也体现在它的共识率普遍较高上;
- 而 Saber 得益于更精确的全程序指针分析算法作为基础, 在支持的缺陷类型检测中, 它具有最高的召回率, 表明其检测特定类型缺陷的能力最佳. 同时, 它也有最高的误报率: 一方面在于其实现是实验性的学术项目, 并未做过多的针对性优化; 另一方面, 它采取了较为保守的报告方式以尽可能地报告潜在的缺陷.

3.2 开源项目测试集

我们对每个开源项目上的检测给定了 3 600 s (1 h)的时间限制, 对于 FbInfer 和 ClangSA 的并行度设置为 12, Saber 因为要求输入是 LLVM Bitcode 文件, 而且其并未内置并行运行的机制, 因此我们对每个项目最多允许 12 个检测实例同时运行. 同时, 我们对每个工具尽量使用其选项默认的检测器列表, 相对来说, ClangSA 默认检测的项目更多. 表 8 中所列内存(memory)为检测工具的内存峰值, 时间(time)为每个项目总运行时长. 在汇总行, 内存数据为总体运行最大内存峰值, 时间为所有项目的累积运行时间.

表 8 在开源项目测试集上的性能

Tools		FbInfer		ClangSA		SVF Saber	
Project	Size (Klocs)	Memory (MBs)	Time (s)	Memory (MB)	Time (s)	Memory (MBs)	Time (s)
wrk	5	1 020	15.2	303	6.1	15	12.9
bzip2	6	1 123	19.6	439	11.8	254	13.5
memcached	12	1 331	46.2	925	28.6	158	13.2
mujs	15	772	7.6	162	6.0	1 097	16.4
less	20	2 351	26.8	1 058	12.2	354	14.2
icecast	21	1 753	29.7	1 189	57.9	718	23.8
shadowsocks	26	1 572	21.9	1 156	37.8	1 631	22.1
darknet	29	2 270	123.7	1 211	18.0	3 885	142.2
sed	33	1 128	23.7	352	9.7	14	12.8
libssh2	35	1 643	7.8	1 214	45.4	615	25.4
gzip	43	1 229	13.2	617	12.2	710	62.5
make	44	2 315	37.1	1 278	17.3	650	22.1
tmux	48	5 036	127.3	1 161	54.1	3 238	102.2
goaccess	51	2 776	77.1	1 065	34.9	940	23.5
wabt	52	2 854	184.9	–	–	7 410	29.5
caffe	60	4 925	399.4	4 864	38.1	5 194	46.1
grep	81	1 226	22.4	659	787.0	406	16.1
tar	83	3 950	178.2	1 288	46.7	1 799	32.6
nghttp2	90	5 976	221.7	2 443	113.0	2 886	23.7
swoole	94	3 292	122.1	1	2.0	3 131	33.7
sendmail	95	5 573	466.8	148	287.3	1 141	13.6
bison	104	2 285	49.2	1 335	25.4	2 952	142.2
bash	113	7 748	215.8	3 868	68.4	4 964	157.2
curl	145	1 394	10.7	1 373	64.5	1 925	83.1
mlpack	166	13 634	681.1	19 393	2 452.1	28 581	237.0
gnugo	209	22 969	454.9	943	2 591.8	4 244	115.2
coreutils	222	30 567	612.2	1 234	48.2	3 398	172.6
git	235	18 790	475.4	1 407	138.4	–	>3600
httpd	309	280	15.1	1 096	1 819.6	1 483	102.2
emacs	319	22 586	800.5	1 683	2 547.7	61 159	1 750.4
mupdf	339	1 727	490.6	1 400	80.9	90 619	2 406.3
bitcoin	356	–	>3600	6 045	387.6	15 429	46.2
mxnet	358	9 633	1 573.7	18 986	418.6	497 759	1 308.9
vim	374	41 756	1 457.2	1 414	142.1	–	>3600
ffmpeg	1 122	12 333	1 390.5	1 431	500.3	–	>3600
php	1 171	–	>3600	1 697	131.4	–	>3600
wine	3 339	–	>3600	1 300	2 091.7	7 801	1 233.4
summary	9 824	41 756	5.9 (hours)	19 393	4.2 (hours)	497 759	6.3 (hours)

3.2.1 性能与可用性

如表 8 中所示, 在我们给定 1 h 的时间限制后, FbInfer 的检测过程中有 3 个项目超时的, 而 Saber 有 4 个项目存在超时的情况. 其原因在于软件项目规模(>20 万行)以及项目代码本身的复杂度. 在实验数据中, 我们观察到部分较为反常的性能数据, 例如, FbInfer 在检测 httpd 时, 内存和时间消耗都非常少. 人工检查其检测过程中产生的中间文件时发现, FbInfer 产生了一些内部错误, 导致大部分项目代码并未被分析和检测. 同时, 在 ClangSA 和 Saber 中也存在会导致部分项目代码未能被检测的问题. 比如 ClangSA 在 wabt 上由于内部错误无法给出结果报告, Saber 也在 bitcoin 等项目(的部分检测目标)上发生了崩溃的情况. 除去超时的 3 个项目, FbInfer 有 70.3%的项目耗时超过 ClangSA, 但是总用时仅高 20.4%, 且两者对其中 81.1%的项目都能在 15 分钟内给出项目的缺陷报告. 由此可见, 两款工具的默认检测选项对大部分项目而言具有相近的检测效率(即使它们的检测类别并不一致). 内存峰值上看, FbInfer 相比于 ClangSA 高了 1 倍, 但超过 20 GB 的仅有 3 个项目. Saber 由于进行了全程序分析, 相比于以编译单元(compilation unit)为单位进行检测的 FbInfer 和 ClangSA 来说, 内存峰值高了一个数量级, 达到了 496 GB 的最大内存占用(几乎达到了实验平台的上限 512 GB).

经过多次实验, 我们发现静态检测工具给出的报告数量存在微小的波动(<1.0%); 同时, 实验中我们也发现, FbInfer 的检测过程会造成项目构建目录内出现错误, 导致检测过后的项目无法再次清理和构建. 由于 Saber 是一个实验性工具, 本身存在一些程序缺陷和错误, 检测过程中存在崩溃的情况(8/37).

3.2.2 缺陷报告的精度

我们在表 9 中列举了 ML (memory leak, 内存泄漏)、UAF (use after free、释放后使用)、NPD (NULL pointer dereference)、BOF (buffer overflow, 缓冲区溢出)这 4 类内存安全缺陷, 括号内为其对应误报率(FPR %), 我们通过在各个项目的报告中随机选择 10–30 个不同缺陷报告人工地验证并估计总体的误报率.

表 9 开源项目测试集上的实验结果

Tools	Infer				ClangSA				SVF Saber	
Project	ML (FPR)	UAF (FPR)	NPD (FPR)	BOF (FPR)	ML (FPR)	UAF (FPR)	NPD (FPR)	BOF (FPR)	ML (FPR)	UAF (FPR)
bash	19 (89.5)	0 (–)	16 (20.0)	92 (80.0)	5 (80.0)	0 (–)	55 (60.0)	21 (100)	1 (100)	1 (100)
bison	3 (100)	0 (–)	4 (100)	39 (100)	0 (–)	0 (–)	12 (100)	84 (100)	0 (–)	– (–)
bitcoin	– (–)				6 (–)	3 (100)	24 (100)	4 (100)	– (–)	
bzip2	0 (–)	0 (–)	0 (–)	8 (100)	0 (–)	0 (–)	0 (–)	1 (100)	3 (100)	2 (100)
caffe	0 (–)	0 (–)	2 (100)	43 (100)	0 (–)	0 (–)	1 (0.0)	0 (–)	– (–)	
coreutils	17 (100)	0 (–)	2 (100)	79 (100)	4 (100)	0 (–)	29 (95.0)	52 (100)	557 (100)	316 (86.7)
curl	0 (–)				0 (–)	0 (–)	8 (0.0)	32 (100)	6 (–)	5 (100)
darknet	15 (46.7)	0 (–)	103 (0.0)	82 (100)	40 (5.0)	0 (–)	6 (33.3)	0 (–)	363 (90.1)	53 (100)
emacs	10 (70.0)	0 (–)	9 (33.3)	97 (100)	11 (13.6)	0 (–)	49 (80.0)	80 (100)	87 (93.1)	6 (100)
ffmpeg	0 (–)	0 (–)	180 (21.1)	712 (100)	2 (–)	0 (–)	199 (100)	318 (100)	– (–)	
git	2 (50.0)	0 (–)	33 (80.0)	753 (100)	0 (–)	0 (–)	62 (100)	77 (100)	111 (100)	86 (100)
gnugo	4 (100)	0 (–)	15 (0.0)	113 (100)	0 (–)	0 (–)	5 (100)	16 (100)	30 (100)	23 (100)
goaccess	0 (–)	0 (–)	13 (20.0)	23 (100)	0 (–)	4 (0.0)	15 (100)	4 (100)	1 (100)	1 (100)
grep	9 (100)	0 (–)	8 (37.5)	11 (100)	2 (–)	0 (–)	18 (20.0)	13 (100)	– (–)	
gzip	2 (100)	0 (–)	0 (–)	8 (100)	2 (100)	1 (100)	0 (–)	5 (100)	17 (100)	15 (100)
httpd	6 (66.7)	0 (–)	22 (20.0)	29 (100)	0 (–)	1 (100)	43 (70.0)	39 (100)	7 (100)	4 (100)
icecast	10 (80.0)	0 (–)	52 (10.0)	8 (100)	7 (28.6)	0 (–)	2 (100)	0 (–)	15 (100)	18 (100)
less	0 (–)	0 (–)	17 (81.8)	35 (100)	0 (–)	2 (100)	1 (0.0)	14 (100)	1 (100)	1 (100)
libssh2	0 (–)	0 (–)	1 (100)	14 (100)	0 (–)	0 (–)	4 (100)	2 (100)	14 (100)	12 (100)
make	14 (100)	0 (–)	2 (0.0)	22 (90.0)	0 (–)	1 (100)	11 (100)	31 (90.0)	7 (100)	5 (100)
memcached	6 (83.3)	0 (–)	0 (–)	55 (100)	5 (40.0)	1 (100)	9 (100)	3 (66.7)	4 (50.0)	6 (100)
mlpack	0 (–)	5 (100)	0 (–)	1 (100)	3 (33.3)	1 (100)	1 (100)	2 (100)	– (–)	
mujs	0 (–)				0 (–)	0 (–)	0 (0.0)	1 (100)	1 (100)	1 (100)
mupdf	3 (100)	0 (–)	32 (40.0)	132 (92.3)	0 (–)	0 (–)	64 (90.0)	125 (100)	17 (100)	2 (100)
mxnet	2 (100)	0 (–)	1 (100)	62 (100)	0 (–)	0 (–)	1 (100)	1 (100)	– (–)	

表 9 开源项目测试集上的实验结果(续)

Tools	Infer				ClangSA				SVF Saber	
Project	ML (FPR)	UAF (FPR)	NPD (FPR)	BOF (FPR)	ML (FPR)	UAF (FPR)	NPD (FPR)	BOF (FPR)	ML (FPR)	UAF (FPR)
nghttp2	2 (50.0)	0 (–)	27 (60.0)	39 (100)	0 (–)	0 (–)	6 (100)	8 (100)	16 (100)	5 (100)
php	– (–)				0 (–)	0 (–)	4 (75.0)	82 (100)	– (–)	
sed	2 (100)	0 (–)	7 (100)	7 (100)	2 (–)	4 (0.0)	3 (100)	3 (100)	3 (100)	1 (100)
sendmail	0 (–)	0 (–)	3 (66.7)	247 (100)	2 (0.0)	0 (–)	21 (100)	34 (90.0)	3 (100)	0 (–)
shadowsocks	10 (100)	0 (–)	4 (50.0)	10 (100)	12 (100)	21 (100)	19 (100)	9 (100)	34 (100)	0 (–)
swoole	53 (73.6)	1 (0)	96 (80.0)	77 (100)	2 (100)	2 (50.0)	6 (100)	18 (100)	– (–)	
tar	5 (100)	0 (–)	8 (50.0)	33 (100)	4 (100)	1 (100)	17 (100)	29 (100)	32 (100)	26 (100)
tmux	3 (100)	0 (–)	19 (90.0)	30 (100)	0 (–)	11 (100)	13 (80.0)	2 (100)	15 (100)	20 (100)
vim	1 (100)	0 (–)	39 (90.0)	651 (100)	0 (–)	0 (–)	55 (80.0)	138 (92.3)	– (–)	
wabt	0 (–)	0 (–)	10 (50.0)	47 (100)	0 (–)	0 (–)	2 (100)	0 (–)		
wine	– (–)				87 (90.8)	16 (100)	3007 (93.3)	1461 (100)	149 (90.0)	104 (90.0)
wrk	1 (100)	0 (–)	9 (100)	0 (100)	0 (–)	0 (–)	3 (100)	0 (–)	– (–)	
Summary	199 (82.9)	6 (83.3)	734 (49.3)	3559 (98.5)	196 (66.8)	69 (94.2)	3775 (85.9)	2709 (94.9)	1494 (96.9)	713 (97.2)

在我们所选择的 4 个分类上, FbInfer 的综合误报率为 84.9%, ClangSA 的综合误报率为 85.7%, Saber 的综合误报率为 96.9%. 即使 Fbinfer 和 ClangSA 在误报率上相差不大, 但是结果表明: FbInfer 对误报的绝对数量控制比 ClangSA 更优秀, 其报告的缺陷总数量相对较低. FbInfer 和 ClangSA 在不同缺陷类别上的表现也良莠不齐: 在空指针解引用缺陷上, FbInfer 的误报率只有 49.3%, 是所有分类误报率中最低的; 而其他 3 类均有超过 80%的误报率, 且都高于 ClangSA. 除去无法完成检测的项目后, Saber 有最高的误报率, 但在释放后使用和内存泄漏这两个缺陷上, 相比于 FbInfer 和 ClangSA, 则可以发现更多的真实缺陷.

纵观表 9, 我们发现: FbInfer 在 51 (128)项检测上是 100%的误报率, 而 ClangSA 是 64 (148)项, Saber 是 41 (52)项. 造成这种极高误报率情况的原因之一是: 在我们所选择的 37 个项目中, 36 个是经过大量测试的成熟发行版本. 另一个不可忽略的原因是: 在现实世界的软件项目中, 缺陷发生的上下文非常复杂, 包括在复杂路径条件、深调用链、复杂数据结构等多种因素的影响下, 静态缺陷检测工具的精度急剧下降.

3.2.3 缺陷报告的可理解性

不同于动态检测工具可以产生有效的测试用例来复现缺陷, 静态缺陷检测工具通常只能给出包含潜在异常点的关键信息, 需要开发人员人工分析和定位缺陷, 因此缺陷报告的可理解性至关重要.

在人工检查缺陷报告的过程中, 我们发现 3 款工具给出的缺陷报告的可理解性都不尽如人意. 表 10 中, 我们以 4 类内存安全相关缺陷为例, 列举了不同缺陷报告应当给出的必要信息和实验中 3 款静态检测工具所给出的实际报告, 其中, ClangSA 的报告都附加一个包含参考触发路径的 HTML 文件以提供更丰富和直观的信息, 这些信息可以覆盖表 10 中所列各类型缺陷判别的必要信息, 对定位缺陷是否能发生及发生的原因有很大的帮助作用; 而 Fbinfer 和 Saber 受限于其检测过程中计算和保留的程序信息, 都仅仅给出简短的缺陷描述. 如表 10 中缓冲区溢出缺陷报告样例所示.

- FbInfer 推断出了的缓冲区大小范围的表达式, 但缺少缓冲区大小变量 *G_buffer_size* 和索引偏移量(–1)来源的相关信息, 且缺陷报告中未指明是哪个指针引用的缓冲区发生了溢出. 当报告指向某个 memcpy 调用, 但是并未指出是源缓冲区还是目标缓冲区发生的溢出时就会产生混淆, 这些混淆通常

需要投入更多的人力才能加以区分;

- Saber 则给出了其所计算的稀疏数据流图上关键节点所对应的代码位置, 但是由于其稀疏性的限制, 各个节点稀疏分布在代码中缺乏语义的连贯性, 而且给出的这些关键程序点大多数是令人困惑的分支指令. 同时, Saber 的部分缺陷报告缺少必要的信息, 例如在内存泄露中只给出了泄漏点的位置和指针变量名, 对于被泄露内存的来源和泄露路径在控制流上是否可达都没有提及, 给缺陷定位带来了巨大的困难.

表 10 检测工具缺陷报告样例

缺陷类型	必要信息	工具	报告信息
内存泄露	内存分配点 泄露点 路径可达性	FbInfer	gnu/malloca.c:65: error: Memory Leak memory dynamically allocated at line 52 by call to ‘malloc’, is not freed after the last access at line 65,column 11
		ClangSA	lib/localcharset.c:240 Potential leak of memory pointed to by ‘res_ptr’
		Saber	NeverFree : memory allocation at: (ln: 1153 fl: tar.c)
释放后使用	内存分配点 指针使用点 指针来源信息 内存释放点 相关指针 别名关系 路径可达性	FbInfer	src/mlpack/methods/linear_regression/linear_regression_ main.cpp:204: error: Use After Delete accessing memory that was invalidated by ‘delete’ on line 194
		ClangSA	sed/utils.c:93 Use of memory after it is freed
		Saber	Double Free: memory allocation at: (ln: 1811 fl: wordsplit.c) double free path: →(ln: 1812 fl: wordsplit.c) →(ln: 1817 fl: wordsplit.c)
空指针解引用	指针置空点 解引用指针来源 路径可达性	FbInfer	gnu/obstack.c:204: error: Null Dereference pointer ‘new_chunk’ last assigned on line 200 could be null and is dereferenced at line 204,column 3
		ClangSA	root/benchmarks/coreutils-8.32/src/shuf.c:246 Access to field 'buffer' results in a dereference of a null pointer (loaded from variable ‘*p*’)
缓冲区溢出	缓冲区分配点 缓冲区边界 访问索引变量 索引来源 各点间的可达性	FbInfer	src/tac.c:297: error: Buffer Overrun L1 Offset: -1 Size: [‘*G_buffer_size*’,+oo]
		ClangSA	lib/localcharset.c: 243 Out of bound memory access (accessed memory precedes memory block)

路径可达性判别在缺陷成因分析过程中是不可回避的, 它包含两个部分: (1) 存在一条路径覆盖缺陷发生的关键程序点; (2) 这条路径是可行的. ClangSA 给出的参考路径可以满足前者, 而对于路径是否实际可行则没有任何保证, 通过实验中对其给出缺陷报告的人工分析, 我们发现 ClangSA 给出的路径受到其分析范围(通常是单一编译单元)的影响而可行性存疑. 而 FbInfer 和 Saber 所给出的报告信息对于判断路径可达性没有任何帮助, 甚至 Saber 还会给出一些误导信息, 导致大量的人工浪费在理解程序语义和分析缺陷发生场景上.

4 对于精度的进一步讨论

本文实验中所选取的 3 款静态缺陷检测工具都是开放源代码的, 因此我们可以结合其实现来研究其所给出报告中误报和漏报发生的具体原因, 并对提高静态缺陷检测工具精度的可行性方法进行讨论.

4.1 Juliet C/C++测试集上的误报和漏报

在所选择的 16 类共 24 641 个测试用例中, 我们对 3 款工具的漏报和误报实例进行了分析统计, 对其发生原因进行了分类归纳. 表 11 中列举了各个工具在不同原因下产生误报和漏报的统计信息, 从中我们发现: FbInfer 的误报主要受到对 C/C++的语义建模缺失的影响; 而 ClangSA 是集成到编译器中的检测器对各种语言语义的支持较为全面, 但为了性能, 牺牲了路径条件表达上的准确性; SVF Saber 与 FbInfer 的情况类似, 由于其主要技术采用了稀疏值流分析, 对其他与内存无关的语义关注较少. 对于漏报, 3 款工具都有 80%左右的漏报是由于它们未对缺陷发生的特定场景进行建模检测所致.

表 11 Juliet C/C++测试集上误报/漏报原因

工具 / 误报/漏报原因	FbInfer				ClangSA				SVF Saber			
	误报		漏报		误报		漏报		误报		漏报	
	数量	比例(%)	数量	比例(%)	数量	比例(%)	数量	比例(%)	数量	比例(%)	数量	比例(%)
不精确的路径条件	505	33.6	61	0.4	351	75.5	0	0.0	526	29.0	0	0.0
不完备的语义建模	946	63.0	771	4.8	16	3.4	1 077	7.2	1 288	71.0	339	19.0
过程间信息的缺失	54	3.6	1 830	11.3	98	21.1	2 036	13.7	0	0.0	0	0.0
缺陷场景未建模	–		13 505	83.5	–		11 772	79.1	–		1 444	81.0

(1) 不精确的路径条件

路径敏感性(path sensitivity)分析是近年来提高静态分析精确度的一种有效手段. 在程序关键局部实行路径敏感分析, 是当前静态缺陷检测工具为达到精度和性能的平衡而普遍所采取的折中. 不同于程序的实际执行, 静态分析过程中存在无法精确评估或者表达的路径分支条件, 这也是造成误报或者漏报的首要因素. 例如, 全局变量、静态变量和外部调用等依赖运行时状态的分支条件, 静态分析工具都不能进行精确的表示和评估. 图 4(a)的示例中, 第 59 行和第 68 行对静态变量 staticTrue 和 staticFalse 的相关分支条件在程序运行时是确定的, 由于静态变量对程序的其他部分是可见的(可被改变), 使得静态缺陷检测工具无法在第 59 行或第 68 行得到一个确定的结论, 从而导致本文所选取的 3 款工具都在这个函数内给出了一个内存泄漏的误报.

(2) 不完备的语义建模

由于 C/C++语言本身、程序逻辑和程序运行系统环境的复杂性, 静态分析工具通常只关注常用的核心语言特性和语义, 无法对繁杂的语义进行面面俱到的建模. 这种情况导致了静态缺陷检测工具在使用建模难度较大以及使用频率较低的语言特性的代码上产生了大量的误报和漏报. 对于语言特性, 如字符串(string)、数组(array)、联合(union)、位操作、指针算数运算等; 外部调用, 如 realloc、realpath 等; 复杂数据结构, 如 STL 内提供的各种容器类型, 本文所选 3 款工具对这些语义的建模都是不完备的. 如图 4(b)中的例子所示, 在第 57 行对 data 指针进行 realloc 之后, 在 realloc 的返回值 tmpData 不为 NULL 时, 第 62 行将 data 重新指向 realloc 返回的内存区域, 并在第 67 行将 data 释放, 此函数中并不存在内存泄漏的可能性. 但是由于对 realloc 的语义的建模并不完备, 无法区分 realloc 返回值 tmpData 是否为 NULL 的不同情况下, realloc 调用对 data 指向的内存区域的不同操作对程序正确性的影响, 从而导致 3 款工具都在此函数中报告了一个内存泄漏的误报.

(3) 过程间信息的缺失

本文所选 3 款静态缺陷检测工具采取了 3 种不同形式的过程间分析: ClangSA 只关心当前编译单元内定义的函数, 使用静态符号执行的方式在跨过程的 AST 上进行路径分支遍历; FbInfer 使用数据库存储已分析函数的部分摘要结果, 在调用对应函数时进行映射复合; Saber 则按照调用图自底向上地计算函数的值流摘要, 并在对应调用点以当前上下文的值流摘要代替重新分析被调用函数. ClangSA 和 FbInfer 分析的基本单位都是编译单元, 由于各单元间的复杂依赖关系, 因此它们可能无法获得其他单元内函数的定义或者分析摘要. 进行全程序分析的 Saber 由于性能和计算资源的因素, 分析的调用深度和值流摘要的精度都受到了巨大的限制. 同时, 基于过程间摘要的静态分析算法也受到路径敏感性的困扰: 仍然不存在任何可以完全区分不同路径的摘要生成算法, 现有方法通常只对分析所关注的关键部分进行摘要, 从而降低了摘要的复杂度. FbInfer 和 ClangSA 都未能在图 4(c)所示的代码示例中对可能已经在 helperBad 中被释放的 reversedString 指针的危险使用给出缺陷报告, 通过查看两款工具的实现代码和中间数据, 我们发现 FbInfer 对 helperBad 函数的分析摘要中并未正确地表达程序对 reversedString 的释放并返回的语义. ClangSA 漏报的原因则是它没有遍历执行到程序中所有的路径, 包含 34→35→74 的路径被 ClangSA 忽略, 从而产生了漏报.

(4) 缺陷发生场景未建模

CWE 分类中包含一些开发者通常情况下不关心的缺陷发生场景, 比如程序结束时发生的内存泄漏、指针使用后空指针检查等缺陷普遍被认为不具有危害性, 因此静态检测工具通常不对这些场景下发生的缺陷进行报告. 如图 4(d)中的代码所示, 第 28 行的空指针检查并没有必要, 这揭示了要么空指针检查的位置是错误的,

要么这是一个冗余的检查. 我们观察 ClangSA 的实现后发现, 这些不必要的检查可能会误导分析工具导致漏报, 例如图 4(d)中的第 25 行可能发生的空指针解引用.

```
55. static void goodB2G1(){
59.   if (staticTrue)
60.   {
61.     /* FLAW: Allocate memory on the heap */
62.     data = (int *)malloc(100*sizeof(int));
67.   }
68.   if (staticFalse)
69.   {
71.     printLine("Benign, fixed string");
72.   }
73.   else
74.   {
75.     /* FIX: Deallocate memory */
76.     free(data);
77.   }
78. }
```

(a) CWE401中的误报

```
48. static void good1(){
51.   int* data = (int*)malloc(100*sizeof(int));
      ...
57.   int* tmpData = (int*)realloc(data,
                          (130000)*sizeof(int));
      ...
60.   if (tmpData != NULL)
61.   {
62.     data = tmpData;
        ...
66.   }
67.   free(data);
69. }
```

(b) CWE401中*realloc*相关的误报

```
18. static char * helperBad(char * aString){
22.   char * reversedString = NULL;
23.   if (aString != NULL){
26.     reversedString = (char *) malloc(i+1);
27.     if (reversedString == NULL) {exit(-1);}
33.     /* FLAW: Freeing a memory block and then
        returning a pointer to the freed memory */
34.     free(reversedString);
35.     return reversedString;
37.   }else{
39.     return NULL;
40.   }
41. }
69. void CWE416_..._return_freed_ptr_01_bad(){
73.   char * reversedString = helperBad("BadSink");
      /* a dereference of reversedString */
74.   printLine(reversedString);
80. }
```

(c) CWE416中过程间相关误报

```
20. void CWE476...__null_check_after_deref_01_bad(){
23.   int *intPointer = NULL;
24.   intPointer = (int *)malloc(sizeof(int));
25.   *intPointer = 5;
26.   printIntLine(*intPointer);
27.   /* FLAW: Check for NULL after dereferencing the
      pointer. This NULL check is unnecessary. */
28.   if (intPointer != NULL){
30.     *intPointer = 10;
31.   }
32.   printIntLine(*intPointer);
34. }
```

(d) CWE476中的误报

图 4 Juliet C/C++测试套件上误报漏报举例

4.2 开源程序测试集上的误报

与 Juliet C/C++测试集不同, 在开源项目上, 我们获得的缺陷报告通常会发生在比较复杂的上下文语义中, 这些误报通常是由多种因素共同作用所导致的. 由于缺少已发现缺陷的相关数据, 我们通过人工分析 1 517 个缺陷报告所对应源码, 然后将发生误报的具体原因总结为 4 类: (1) 更加复杂的路径约束; (2) 程序语义的近似折中; (3) 别名混淆; (4) 深调用链和外部调用. 针对 3 款检测工具, 分别在每个开源项目上对特定缺陷类型的缺陷报告中随机选取不超过 30 个误报, 通过对静态检测工具实现中的关键程序点进行插桩等手段, 我们尝试详细分析缺陷检测过程中导致误报的具体原因. 由于无法跟踪缺陷检测过程中的所有信息, 因此一个误报可能定位到多种原因. 表 12 中列举了各个检测工具针对所选定的 4 类缺陷, 在开源程序测试集上误报发生原因占比的相关统计数据. 其中, 复杂路径约束导致了超过 50%的误报. 相对来说, 其他 3 类误报原因对不同的缺陷检测的影响之间存在显著差异.

表 12 开源程序集上误报的原因

工具 / 误报原因(占比%)	Infer					ClangSA					SVF Saber		
	ML	UAF	NPD	BOF	ALL	ML	UAF	NPD	BOF	ALL	ML	UAF	ALL
更加复杂的路径约束	30.0	0.0	86.7	60.0	55.8	38.9	23.3	80.0	66.7	53.7	56.7	60.0	58.3
程序语义的近似折中	23.3	0.0	10.0	63.3	30.5	22.2	56.7	10.0	53.3	37.0	0.0	50.0	25.0
别名混淆	60.0	0.0	13.3	0.0	23.2	55.6	73.3	3.3	0.0	30.6	53.3	53.3	53.3
深调用链和外部调用	23.3	100	56.7	23.3	37.9	33.3	26.7	80.0	40.0	46.3	3.3	0.0	1.7

(1) 更加复杂的路径约束

以缓冲区溢出缺陷为例, 无论是 FbInfer 还是 ClangSA, 我们人工验证的 408 个随机抽取的缓冲区溢出缺陷报告中, 超过 60%的误报都是复杂路径约束导致的. 人工分析对应缺陷报告, 并参考缺陷检测工具的实现后, 我们将路径条件相关的误报发生的情况具体分为两种.

- 一方面是缺陷检测工具对复杂路径条件的简化: 对 FbInfer 来说, 所有跨过程的缓冲区访问中, 对路

径的约束条件大部分都将被简化丢弃; ClangSA 为了性能, 只计算简单分支条件, 甚至完全忽略部分路径条件; Saber 的处理方法是使用二元决策图(binary decision diagrams)[46]编码路径约束, 然而在其实现中, 二元决策图根据稀疏值流图做出了简化以保持最小态, 造成大量约束被合并简化;

- 另一方面则是无法精确处理的路径条件在复杂程序上下文中的传播: 全局变量、静态变量、状态标记量(错误编号、引用计数等)与程序运行时的状态有关, 对其所具有的值或性质无法单纯地通过静态分析予以覆盖. 如图 5 中的 *sentinel_length*, 一方面它表示 *G_buffer* 中预留的长度, 同时在其取非零值时也代表分隔符匹配过程中使用了 non-regexp 模式, 从而导致静态缺陷检测工具无法识别程序在不同分支中行为的差异.

(2) 程序语义的近似折中

静态分析对于程序语句的建模是一种近似抽象, 因此存在部分语义的损失. 例如, 对位操作(bitwise operation)、浮点运算、字符串操作和自然循环等都不能很好地进行抽象. 例如图 5 中的第 255 行, 我们无法确认函数 *safe_read* 的返回值具体是什么, 一般采取近似折中的手段使 *match_start=G_buffer*. 另一个典型的语义近似是对于程序中循环的处理, 我们在 3 款工具的实现中都观察到了对循环的折中处理: FbInfer 对过程内的循环计算了起始、终止等不同条件下的相关循环不变量以表达循环语义; ClangSA 则通过限制节点的访问次数来有限制地对循环加以展开, 但是, 由于受到性能限制, 通常它的检测器在实现中只允许循环节点被遍历执行 1 次; 而 Saber 则会直接忽略循环造成的影响. 例如图 5 中第 297 行所对应的循环, 本文选择的 3 款工具都不能正确地表达循环对 *match_start* 所引用的缓冲区与 separator1 进行比对判断内容是否一致的语义以及它所造成的副作用(对 *mach_start* 所指向位置的偏移操作).

(3) 别名混淆

别名问题是静态分析中的基础问题, 尤其是在 C/C++这类允许用户直接进行内存指针操作的语言. 不同于程序的实际执行, 静态分析方法受到问题规模和分析难度的限制, 无法精确地分析出指针间的关系, 例如运行时动态确定的指针等. 我们在内存泄漏和释放后使用的缺陷报告中观测到了这种由于指针别名的混淆造成的误报.

(4) 深调用链和外部调用

由于语义近似损失的不可避免性, 从缺陷的发生点到触发点在执行路径上越远, 近似所产生的误差就越大; 而且随着调用深度的增长, 分析规模(包括上下文数量、内存对象数量等)会急剧增加, 分析精度逐步下降. 同时, 程序中调用的外部库函数、系统函数等, 在静态分析中由于无法确定调用点的程序状态, 因此而无法确定其效果和返回值. 如果缺陷检测工具为对其进行建模, 那么在分析中就只能忽略其影响. 例如图 5 中第 298 行的 *STREQ_LEN*(*strncmp*)函数的调用, FbInfer、ClangSA 和 Saber 都忽略了它所造成的影响.

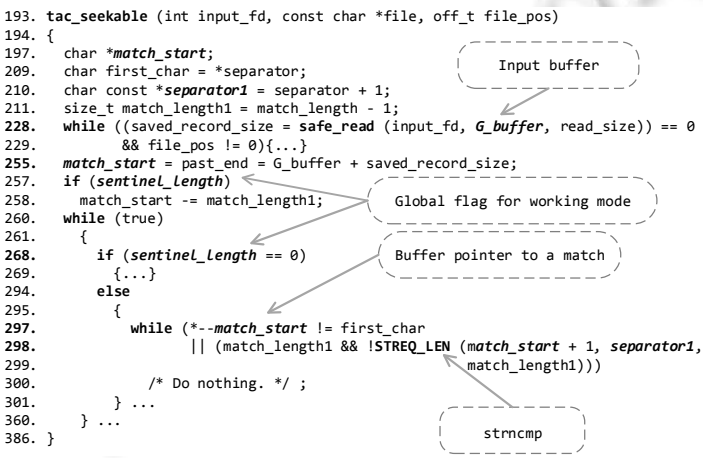

图 5 CoreUtils 中的代码片段

4.3 降低误报率的可行方法

高误报率是当前静态检测工具面临的首要问题, 至今仍然没有高效的通用算法可以有效降低静态分析误报率. 当下比较流行的手段是对静态缺陷检测工具在特定检测目标上进行有针对性的调优, 例如 LGTM[47]等检测平台允许用户自定义检测规则、Fortify SCA 使用机器学习针对特定分析目标消除误报等.

静态分析算法精度的提升, 也是降低静态缺陷检测工具误报率的有效途径之一. 路径敏感的静态分析算法可以在一定程度上解决缺陷定位和成因分析中所必需的路径可达性的判定问题, 近年来, Pinpoint[45]、Fusion[48]等都通过引入部分路径敏感性来提升分析精度. 同样地, 对于上下文敏感性(context-sensitivity)[49]的提高, 也可以提高静态分析的精度. 应用此类方法的难点在于现有缺陷检测工具的算法本身已经很复杂, 对它进行扩展并不容易, 而且精度的提升意味着计算资源消耗的急剧增长, 同时对算法性能优化提出了更高的要求.

5 总结与展望

本文评估了 3 款开源静态缺陷检测工具在 Juliet C/C++测试套件和开源项目测试集上的实际表现, 深入理解了现实世界中静态缺陷检测工具的发展现状和缺陷挖掘的实际效果. Andrei 等人[50]在丰田 ITC 静态分析测试套件上对比了 11 款开源的 C/C++静态缺陷检测工具的效能. 以大规模 C/C++软件项目中的已确认缺陷数据集作为研究对象, 文献[24]评估了 CppCheck 和 FlawFinder 两款开源静态缺陷检测工具的能力. Marcilio、Diego 等人[51]通过追踪研究对 SonarQube 提交给不同开源项目缺陷被修复的情况来考察现实中静态缺陷检测工具所产生的实际贡献. 不同于以上相关工作, 本文中, 我们人工检查了超过 1 500 个缺陷报告, 结合开源静态检测工具的具体实现, 从中归纳获得常见误报发生的具体原因. 对开源静态缺陷检测工具有针对性地在部分缺陷类型上进行了深入研究, 为进一步优化现有的静态缺陷检测技术起到了支撑作用.

通过研究 3 款开源静态检测工具不同版本的发行说明和相关文档, 我们也总结出了当前静态检测工具的发展方向和趋势.

(1) 简单化、平台化、定制化: FbInfer 和 ClangSA 在近年的发展中逐渐框架化、平台化: 一方面来说, 他们面临的是比较广泛的检测目标和检测目的; 另一方面, 工业界对常见缺陷检测以及合作扩展开发检测工具的现实需要. Saber 本身就是 SVF 指针分析平台的验证性项目, 其分析和检测框架都是可扩展的;

(2) 可伸缩性和性能要求提高: 百万行代码级别的程序逐渐增多、软件迭代速度的加快, 都造成了静态检测工具对性能要求的提高, 检测技术性能优化的相关研究也在增加;

(3) 对于复杂缺陷的检测进展较为缓慢: 复杂缺陷的检测语义和上下文较为复杂, 当前工具对于较为复杂的缺陷通常不是检测能力太差就是报告误报率极高. 比如 Saber, 虽然提供了更精确的指针分析能力, 但检测报告通常会报告更多的误报. 由于静态缺陷检测工具逐渐简单化以及对可伸缩性要求的提高, 对复杂缺陷的研究和检测会是一个长期存在的挑战.

本文中, 我们在开源程序测试集上仅针对内存泄漏、释放后使用、空指针解引用和缓冲区溢出这 4 类安全相关缺陷进行了全面的分析研究, 后续我们将扩展到更多的缺陷类型, 如安全编程风格、无效代码等. 同时, 我们将以实验中分析获得的相关结论为基础, 对现有的静态分析算法进行有针对性的优化, 以缓解长期存在于静态检测工具中的高误报率等问题.

References:

[1] Annual Report on China's Cyber Security 2020. http://it.rising.com.cn/d/file/it/dongtai/20210113/2020.pdf

[2] CVE—Common vulnerabilities and exposures. http://cve.mitre.org/

[3] NVD—National vulnerability database. https://nvd.nist.gov/

[4] Vassallo C, Panichella S, Palomba F, Proksch S, Gall HC, Zaidman A. How developers engage with static analysis tools in different contexts. Empirical Software Engineering, 2020, 25(2): 1419−1457.

[5] Revertion of all of the umn.edu commits. https://lore.kernel.org/lkml/20210421130105.1226686-1-gregkh@linuxfoundation.org/

[6] Johnson B, Song Y, Murphy-Hill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? In: Proc. of the 35th Int'l Conf. on Software Engineering (ICSE). IEEE, 2013. 672−681.

[7] Juliet test suite for C/C++. https://samate.nist.gov/SRD/testsuite.php

[8] Shen Z, Chen S. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. In: Proc. of the Security and Communication Networks. 2020.

[9] Li P, Cui B. A comparative study on software vulnerability static analysis techniques and tools. In: Proc. of the 2010 IEEE Int'l Conf. on Information Theory and Information Security. IEEE, 2010. 521−524.

[10] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. 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]

[11] Fosdick LD, Osterweil LJ. Data flow analysis in software reliability. ACM Computing Surveys (CSUR), 1976, 8(3): 305−330.

[12] Yang Z, Yang M. Leakminer: Detect information leakage on Android with static taint analysis. In: Proc. of the 3rd World Congress on Software Engineering. IEEE, 2012. 101−104.

[13] Baier C, Katoen JP. Principles of Model Checking. MIT Press, 2008.

[14] Cousot P, Cousot R. Static determination of dynamic properties of generalized type unions. ACM SIGOPS Operating Systems Review, 1977, 11(2): 77−94.

[15] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1977. 238−252.

[16] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Proc. of the 6th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1979. 269−282.

[17] Dahse J, Schwenk J. RIPS—A static source code analyser for vulnerabilities in PHP scripts. In: Proc. of the Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum, 2010.

[18] Satyanarayana V, Sekhar M. Static analysis tool for detecting Web application vulnerabilities. Int'l Journal of Modern Engineering Research (IJMER), 2011, 1(1): 127−133.

[19] Nunes PJC, Fonseca J, Vieira M. phpSAFE: A security analysis tool for OOP Web application plugins. In: Proc. of the 45th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. IEEE, 2015. 299−306.

[20] Ferschke O, Gurevych I, Rittberger M. FlawFinder: A modular system for predicting quality flaws in Wikipedia. In: Proc. of the CLEF (Online Working Notes/Labs/Workshop). 2012. 1−10.

[21] Hovemeyer D, Pugh W. Finding bugs is easy. ACM SIGPLAN Notices, 2004, 39(12): 92−106.

[22] PMD source code analyzer. https://pmd.github.io/

[23] Cppcheck—A tool for static C/C++ code analysis. http://cppcheck.net/

[24] Pereira JDA, Vieira M. On the use of open-source C/C++ static analysis tools in large projects. In: Proc. of the 16th European Dependable Computing Conf. (EDCC). IEEE, 2020. 97−102.

[25] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. Formal Aspects of Computing, 2015, 27(3): 573−609.

[26] OCLint. https://oclint.org/

[27] Evans D, Larochelle D. Improving security using extensible lightweight static analysis. IEEE Software, 2002, 19(1): 42−51.

[28] SonarQube. https://www.sonarqube.org/

[29] CheckMarx. https://www.checkmarx.com/

[30] Coverity SAST. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html

[31] Fortify static code analyzer. https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer

[32] Facebook infer. https://fbinfer.com/

[33] Clang static analyzer. https://clang-analyzer.llvm.org/

[34] Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. In: Proc. of the 2012 Int'l Symp. on Software Testing and Analysis. 2012. 254−264.

[35] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization (CGO 2004). IEEE, 2004. 75−86.

[36] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science. IEEE, 2002. 55−74.

[37] Calcagno C, Distefano D, O'hearn PW, Yang H. Compositional shape analysis by means of bi-abduction. Journal of the ACM (JACM), 2011, 58(6): 1−66.

[38] King JC. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385−394.

[39] 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.

[40] Sui Y, Xue J. SVF: Interprocedural static value-flow analysis in LLVM. In: Proc. of the 25th Int'l Conf. on Compiler Construction. 2016. 265−266.

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

[42] Whole program LLVM. https://github.com/travitch/whole-program-llvm

[43] Yan H, Sui Y, Chen S, Xue J. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). IEEE, 2018. 327−337.

[44] Yan H, Sui Y, Chen S, Xue J. Machine-learning-guided typestate analysis for static use-after-free detection. In: Proc. of the 33rd Annual Computer Security Applications Conf. 2017. 42−54.

[45] Shi Q, Xiao X, Wu R, Zhou J, Fan G, Zhang C. 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. 2018. 693−706.

[46] Akers SB. Binary decision diagrams. IEEE Trans. on Computers, 1978, 27(6): 509−516.

[47] LGTM—Continuous security analysis. https://lgtm.com/

[48] Shi Q, Yao P, Wu R, Zhang C. Path-sensitive sparse analysis without path conditions. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. 2021. 930−943.

[49] Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for C programs. ACM SIGPLAN Notices, 1995, 30(6): 1−12.

[50] Arusoaie A, Ciobâca S, Craciun V, Gavrilut D, Lucanu D. A comparison of open-source static analysis tools for vulnerability detection in C/C++ code. In: Proc. of the 19th Int'l Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). IEEE, 2017. 161−168.

[51] Marcilio D, Bonifácio R, Monteiro E, Canedo E, Luz W, Pinto G. Are static analysis violations really fixed? A closer look at realistic usage of sonarqube. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). IEEE, 2019. 209−219.

附中文参考文献:

[10] 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. 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]

李广威(1991－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析.

李炼(1977－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为程序分析, 软件安全.

袁挺(1994－), 男, 博士生, 主要研究领域为程序分析.