C 程序接口误用缺陷静态检测技术研究

（申请清华大学工学博士学位论文）

培 养 单 位：软 件 学 院
学　　科：软 件 工 程
研 究 生：谷 祖 兴
指 导 教 师：顾 明 教 授

二〇一九年六月

Static Analysis Based API-Misuse Defect Detection in C Programs

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Software Engineering

by

Gu Zuxing

Dissertation Supervisor : Professor Gu Ming

June, 2019

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

软件库是一套用于软件开发的数据和编程代码的组合，通过应用编程接口对已实现的功能进行封装，令开发者专注于创新，为现代软件提供基础构造模块。开发人员在使用这些接口时，需要满足各种各样的使用约束，例如调用条件和调用顺序。否则，将会产生接口误用。近年来，很多研究人员致力于接口误用缺陷检测。其中，静态检测技术可以在开发早期应用而获得广泛关注。然而，快速发展的市场需求、复杂的使用模式和大量开源软件库中缺失的说明文档，给接口正确使用带来新的挑战。接口误用依旧普遍存在于软件系统中，是导致软件错误、系统崩溃和漏洞的重要原因之一。为提高接口误用检测的能力，应对项目需求，本文以 C 程序为载体，从面向接口使用约束的领域特定描述方法、规模化静态检测方法和实际项目应用三个方面展开研究，主要内容包括：

1. 提出基于缺陷模式的接口使用约束领域特定语言 IMSpec。首先，为理解 C 程序接口误用缺陷特征，本文对实际开源项目的接口误用缺陷修复报告进行研究，总结常见接口误用缺陷模式。基于缺陷模式的特征，本文提出面向接口使用约束的领域特定语言 IMSpec，以描述实际项目中接口使用约束。
2. 提出基于约束描述的规模化接口误用缺陷检测方法 IMChecker。该方法利用 IMSpec 语言以支持用户自定义的接口，提升检测能力；通过多入口分析策略将大规模代码静态分析任务分解为独立的子任务，以应对实际项目需求；并基于上下文的语义信息以及基于使用情况的统计信息两种策略对检测结果进行过滤与排序。在公开数据集上的实验结果显示，IMChecker 取得 13.21% 的误报率和 16.80% 的漏报率。该结果领先于主流的开源静态分析工具。
3. 整理 C 程序接口误用缺陷数据集 APIMU4C 以及实现 C 程序接口误用缺陷检测工具集 Tsmart-IMChecker。APIMU4C 包含调研中分析的缺陷实例和 C 程序接口误用测试数据集，以帮助研究人员和开发者更好的理解接口误用缺陷、评估检测工具的能力以及展开新的研究工作。Tsmart-IMChecker 包含可视化规约撰写工具、缺陷分析引擎 IMChecker-engine 以及基于差异性对比的结果展示工具，以帮助使用者对接口误用缺陷进行检测。

本文的研究成果均已集成于可信软件工具集 TsmartV3 中，并成功应用于 Linux 内核、OpenSSL 加密库、Ubuntu 系统的应用软件等项目中。在提交的 75 个新的缺陷报告中，61 个已经被开发者确认，32 个被开发者修复。

关键词：API 误用；静态分析；实证研究；缺陷检测

Abstract

A software library is a suite of data and programming code, which provides Application Programming Interfaces (APIs) to encapsulate the internal states. It allows developers to focus on innovation and provide primary building blocks for modern software products. Correct usage is required to satisfy rich constraints, such as call conditions or call orders, and violation of these constraints, called API misuses. Recently, many tools, techniques and methodologies have been proposed to address API-misuse problems. In particular, static analysis techniques have long prevailed as the most promising techniques, since they are typically available early in the development process. In the face of rapidly growing market needs, complex usage patterns and great missing documents of open-source APIs, it becomes a challenging task. As a result, API misuses remain widespread and are a prevalent cause of software bugs, crashes, and vulnerabilities. In this paper, we aim to augment the current analysis abilities for C programs of real-world programs from three perspectives, including a domain-specific language for API usage constraints description, a constraint-directed static analysis technique for large-scale programs, and dataset and GUI clients applied to real-world programs. Our main contributions are summarized as follows,

1. A domain-specific language called IMSpec to specify API-usage constraints. We conduct an empirical study to understand the characteristics of API-misuse bugs in real-world C programs. Leveraging this knowledge, we design a lightweight domain-specific language called IMSpec to specify API-usage constraints of real-world APIs.
2. A constraint-directed static analysis technique IMChecker for large-scale programs. We propose IMChecker, a constraint-directed static analysis technique, which employs IMSpec for project-specific APIs, a multiple-entry analysis strategy for large-scale programs, and context semantic and usage statistics to filter and rank detection results. The experimental results on standard benchmarks show that IMChecker achieves 13.21% false-positive rate and 16.80% false-negative rate, which outperform the state-of-the-art tools.
3. A dataset APIMU4C and a GUI toolkit Tsmart-IMChecker for API-misuse detection. APIMU4C includes all the API-misuse instances in empirical study and a

benchmark for researchers and developers to understand the nature of API misuses, evaluate current detection tools and improve API-misuse detectors. We implement Tsmart-IMChecker, a GUI toolkit to help developers write IMSpec, use IMChecker engine and audit detection results.

Our approaches have integrated to TsmartV3 toolkit and successfully applied to real-world programs, such as Linux kernel, OpenSSL and applications of Ubuntu system. For all 75 submitted bug reports, 61 have been confirmed by relevant development communities, 32 of which have been fixed and merged into master branch.

Key Words: API misuse; Static analysis; Empirical study; Bug detection

目 录

主要符号对照表

API	应用程序编程接口（Application Programming Interface）
SSL	加密套接字协议层（Secure Sockets Layer）
RSSE	软件工程推荐系统（Recommender Systems for Software Engineering）
CWE	常见缺陷类型枚举（Common Weakness Enumeration）
OWASP	开放式 Web 应用程序安全项目（The Open Web Application Security Project）
CVE	公共漏洞与暴露库（Common Vulnerabilities and Exposures）
BISL	接口行为规约描述语言（Behavioral Interface Specification Language）
DSL	领域特定语言（Domain Specific Language）
CPA	可配置程序分析（Configurable Program Analysis）
kLOC	千源代码行（kilometer Lines Of Code）
SSA	静态单赋值形式（Static Single Assignment Form）
GUI	可视化用户接口（Graphical User Interface）
CFA	控制流自动机（Control Flow Automata）

第 1 章　绪论

1.1　研究背景

在过去的二十年里，软件在我们的日常生活中无处不在。每一天，新开发的软件服务为我们带来独特的生活视野，使我们的生活更加便捷与高效。同时，软件开发为经济发展带来了巨大贡献。2018 年软件和信息技术服务业统计公报指出，2018 年我国软件和信息技术服务业从业人员为 643 万人，实现利润总额 8079 亿元，占 GDP 比重 3.6%，已成为经济平稳增长的重要推动力量[1]。

如此庞大的软件产业建立在不断增长的软件产品基础之上。因此需要软件开发团队持续快速地研发，以减少每种新产品的上市时间。例如，通过采用持续部署技术，Facebook 公司每天产生的新版本能够达到 100 次甚至 1000 次[2]。与此同时，随着软件需求增加，软件系统越来越复杂。为能够匹配市场需求的开发速度，至关重要的一点就是对软件库中组件的重用[3-4]。

软件库[5] 是具体语言实现的软件功能模块的集合。这些库为现代软件提供构建的基础，重用它们可以让开发人员站在巨人的肩膀上，专注于创新，而不是重塑基本模块。更具体地说，这些软件库通常提供应用程序编程接口（API）以供开发者来快速构建软件系统。本文将 API 简称为接口。

软件开发者在使用这些 API 时，需要掌握这些接口的功能和约束以正确使用这些编程接口，完成目标任务。一个接口使用（API usage）是一段使用某个或某些 API 来完成特定功能的代码片段，由一些基本程序元素（program element）组成。例如：目标接口的函数调用，条件判断，算数运算等等。这些程序元素的组合需要满足目标 API 的使用约束（usage constraint）。例如：对目标 API 参数的检查、API 调用顺序关系和异常处理等等。这些使用约束则和 API 自身属性相关。当一个 API 使用违反了约束中的一个或多个时，我们称为不正确的 API 使用，即 API 误用（API misuse）[6]。API 误用，已成为导致软件错误、崩溃，甚至漏洞的重要原因之一[7-13]。特别地，C 语言多应用于操作系统、嵌入式系统、数据库等基础软件中。因此，对 C 程序接口误用缺陷研究具有重要价值。本文将 API 误用导致的软件错误、崩溃和漏洞，统称为接口误用缺陷，简称为接口缺陷。

图 1.1展示漏洞 CVE-2015-0288[14] 的代码片段。OpenSSL[15] 是广泛应用的安全通信软件库。该库将加密套接字协议层（SSL）[16] 中的通信协议和加密算法封装在 API 中，以方便客户端开发者使用。开发者在使用 OpenSSL 提供的 API 时，需要对各种证书（certificate）进行验证，以确保信息的安全性和有效性。例如，接

```
1     ========== Incorrect Usage ==========
2  Location: OpenSSL/crypto/x509/x509_req.c: 70
3  X509_REQ *X509_to_X509_REQ(...){
4     [...]
5     pktmp = X509_get_pubkey(x);
6     //缺失对变量pktmp的检测
7     + if (pktmp == NULL)
8     +     goto err;
9     i = X509_REQ_set_pubkey(ret, pktmp);
10    EVP_PKEY_free(pktmp);
11    [...]
12 }
13
14    ========== Correct Usage ==========
15
16 Location: /crypto/x509/x509_cmp.c: 390
17 int X509_chain_check_suiteb(...){
18    [...]
19    pk = X509_get_pubkey(x);
20    rv = check_suite_b(pk, -1, &tflags);
21    [...]
22 }
23
24 static int check_suite_b(EVP_PKEY *pkey,...){
25    [...]
26    // 确保变量pkey不为空
27    if (pkey && ...)
28    [...]// 异常处理代码
29 }
```

图 1.1　缺失参数非空检查导致的漏洞 CVE-2015-0288[14]

口 `X509_get_pubkey()` 用于解码证书，当发生错误时，返回 NULL 作为错误代码。因此当开发者忽略对返回值的检查时，则引入一个空指针解引用错误（图中第 9 行所示）。攻击者可以通过构造非法的证书，利用该漏洞进行拒绝服务攻击（Denial of Service），造成目标系统崩溃。

为帮助使用者理解接口的功能和使用约束，设计人员通过各种方式提供高质量的文档以及应用案例作为参考。然而在现有的开发环境下，这些辅助材料并不能有效地帮助开发者理解这些使用约束[17]。更严重的是，随着开源软件的蓬勃发展，大量的开源软件库没有完整的文档资料，甚至存在错误的使用说明[18-19]。此外，当遇到 API 使用问题时，开发者更喜欢直接在搜索引擎或者问答论坛 StackOverflow[20] 中进行查询，而不是查看官方使用说明。不幸的是，这些问答论坛中同样存在大量的错误[21]。研究表明，如何正确使用 API 是阻碍开发者开发的重要瓶颈之一[22]。

因此，研究人员采用更加主动的方式协助开发者正确使用 API，即软件工程推荐系统（RSSE）[23]。该方法的核心是在软件工程任务上下文中，自动获得重要的信息并提供给开发者，以提高开发者的开发效率。一方面，通过相似代码检索[24-26]提供 API 使用样例；或者通过自动补全技术辅助使用者进行开发[27]。另一方面，研究人员通过自动化的分析方法对已有代码进行缺陷检测，提示潜在的 API 误用缺陷，以提高代码质量。

在过去的十几年中，大量的研究成果被成功应用于自动化 API 缺陷检测[28-30]。特别地，静态分析技术因只需要源代码，可以在开发的早期进行而获得广泛关注[31]。尽管在 API 误用检测方面，研究人员投入大量的工作，API 误用在实际项目中依旧普遍存在[13,32]。一项对 Google 应用商店的调研显示，在 11748 个安卓应用中，10327 个被测对象（超过 88%）存在至少一个加密相关的 API 误用[9]。特别地，富有经验的开发者的代码中[33]，甚至在对缺陷修复的代码中，同样会出现 API 误用。例如，OpenSSL 开发者创建缺陷修复补丁（sha: 1c4221）以 ‘‘修复 crl2pkcs7 app 中存在的内存泄漏缺陷’’。然而该补丁忽略上下文路径关系，导致在某些可达路径上对 ‘‘内存重复释放’’。该缺陷被新的补丁（sha: d285b5）修复，修复的日志为 ‘‘在 crl2pl7 中避免重复释放缺陷’’。

随着软件库接口数量增加、软件规模增长、代码复杂度提高和开源代码广泛应用，现有的 API 误用缺陷检测方法面临巨大挑战。具体来说是检测精度和规模的矛盾关系，规模则体现在缺陷种类与代码规模两个层面。一方面，缺陷检测方法需要具有大规模检测能力，即能够应对实际项目需求，对缺陷进行检测。同时，支持尽可能多的缺陷类型。另一方面，缺陷检测方法需要具有较高精度。如果一个检测工具的结果包含大量误报，即缺陷报告中的缺陷并不是实际缺陷，使用者将会摒弃这些工具[34]。同时，为帮助开发者理解缺陷并提高修复效率，检测结果的有效展示形式也是促进检测工具应用的重要因素之一[35]。因此，对接口误用静态检测技术研究，以解决实际项目中检测精度和规模的矛盾关系具有重要意义。如何设计有效的 API 误用缺陷静态检测技术，为开发人员提供高精度、低漏报的缺陷检测服务，并能够辅助开发者理解缺陷、提高修复效率，是目前工业界的迫切需求，也是学术界的研究热点。

1.2　研究现状

针对接口误用缺陷检测，有三大类技术路线：代码审查、动态检测和静态检测技术。代码审查技术旨在通过软件同行协同走查代码的方式，人工地找出并修正代码中编码规范错误以及缺陷。动态检测技术通过执行程序，利用运行时的状态和语义信息检测缺陷。静态检测技术则不需要执行程序，直接进行缺陷检测。后文中将通过这三个方面对 API 误用缺陷检测相关的现有研究进行总结。

1.2.1　代码审查

代码审查（Code Review）是指开发人员通过系统地阅读程序源代码的 API 使用情况，以找出并修正程序中错误，从而提升软件质量的活动[36]。随着版本控制

管理工具和开源社区的发展，在线软件库（例如 Github[37] 和 Bitbucket[38]）可以允许开发者远程协同审查代码，简化代码审查的代价。一项调研显示，在对于 240 个一线开发者的调研中，超过 90% 的开发者所在的公司在使用代码审查技术以提高代码质量[39]。通过开发者协同走查代码，能够有效找到程序中的缺陷、维持代码统一编程风格、增加可维护性、分享领域知识与协调开发进度[40]。特别地，在 2008 年对超过 12000 个实际项目开发流程的调研中，代码审查的缺陷发现率高达 60-65%[41]。

然而，代码审查活动需要大量的人工操作，是代码开发流程中占用时间最多的活动之一[42]。大量的人力和时间成本使得代码审查受到越来越多的质疑。其中微软公司 2015 年的调研结果显示，代码审查严重阻碍开发的进度。在所有的代码审查结果中，只有 15% 的内容与缺陷有关[43]。与此同时，随着代码量的增大，代码审查的效率降低，并需要开发者具备更好的领域知识。这些因素都制约了代码审查在现代软件开发环境中的效果。

1.2.2　动态检测

动态程序检测技术（Dynamic Program Analysis）是程序分析技术的一种[44]，通过程序的运行时行为检测程序的缺陷。动态程序检测技术利用程序运行时状态或利用运行时捕获的信息，对程序的内部逻辑以及功能进行缺陷检测。因此检测结果的正确率可以达到 100%，即没有误报。针对于接口误用，典型的动态检测技术是集成测试（Integration Testing）[45]。该方法对已经通过测试的底层 API 实现模块组合后的代码进行测试，以检测在组合后 API 使用代码片段中的误用，即通过构造输入和输出动态地执行模块以检测程序中 API 使用的正确性。近年来，动态验证技术（Runtime Verification）获得广泛的应用[46]。动态验证技术从传统的验证技术发展而来，旨在运行时对程序的状态进行监控，以查看程序是否满足预定的属性。从技术方案上来说，通常需要进行两个步骤：插桩与日志分析[47-48]。插桩通过对源代码或者二进制代码进行语句或者指令的改写，以在运行时输出或者获得程序的状态。基于获得的程序状态和日志，通过分析技术对程序的运行状态进行查找。当程序崩溃或者违反预先定义的程序属性时，则找到缺陷。例如：AddressSanitizer[48] 是由谷歌公司开发针对内存缺陷的检测工具。该工具已经集成在 Clang[49] 和 GCC[50] 编译器中，以及 Xcode[51] 开发环境中，找到数百个实际缺陷[52]。

动态检测技术具有高精度、跨函数的特点。能够找到跨越多个函数、复杂的 API 误用缺陷。同时，其复杂度与程序的规模呈线性关系，即运行时间不会随着程序规模变化而产生巨大开销。然而，针对于接口误用缺陷检测，该方法面临以下

不足：（1）程序覆盖率。动态检测技术对程序的运行时状态进行捕获和分析。因此，对程序的检测范围取决于测试输入的质量。模糊测试技术（Fuzzy Testing）[53]通过自动构造随机的输入，能够有效地增加检测覆盖率，并成功应用于工业界[54]。然而，在分析过程中依旧只能覆盖程序执行的部分，难以保证所有的程序路径被覆盖。（2）测试构造与插桩。一方面，为能够尽可能多地覆盖程序分支，测试人员需要构造大量的测试用例。然而，这需要大量人力资本和时间，同时需要对程序的语义具有深刻理解。另一方面插桩技术面对不同的程序结构与编译器的优化策略时，可能失效。同时，修改后的程序运行效率受到影响。（3）测试环境。与针对 API 实现的测试不同，API 使用包含多个程序元素、复杂的逻辑关系以及项目特定的约束条件。同时，部分 API 误用缺陷发生在特定的物理环境下[55]。例如：对于特定硬件配置处理的接口，其异常处理需要在特定的环境下才会触发。这些测试环境的准备需要大量的人力和财力。

1.2.3　静态检测

静态检测技术（Static Program Analysis）[56] 与动态检测技术相对，在不执行程序的情况下对程序中缺陷进行检测。针对于接口误用缺陷检测，静态检测技术有三类常用方法：验证技术、静态分析技术和规约挖掘技术。

验证技术　程序验证技术通过遍历程序的所有可能状态以证明程序的正确性[57]。针对于接口误用缺陷，程序验证技术本身难以直接对缺陷进行检测，需要将检测问题转化为可达性问题[58]。检测工具需要提供或内部集成相应 API 正确使用的约束条件，并将约束条件与程序状态相对应。当发现到达违反约束的程序状态时，则认为检测到 API 误用缺陷。

最具代表的接口误用缺陷验证工具是微软公司的 SLAM 项目[59]。SLAM 项目成立于 2001 年，旨在检测软件接口使用行为是否满足约束条件，以辅助开发者正确地使用接口的功能。开发者通过使用 SLIC 规约描述语言[60] 对程序接口的属性进行描述，并使用 Static Driver Verifier（SDV）[61] 工具对目标 API 进行检查。SDV 对源代码进行解析并根据 SLIC 规约对代码进行插桩，并利用反例引导的抽象解释技术（Counter Example-Guided Abstraction Refinement）[62-63] 对 API 误用缺陷进行查找。至 2010 年，该项目已经支持 200 多个 API 使用规约，成功检测 Windows 操作系统驱动程序中 270 个 API 误用缺陷，有效地提高了接口使用的正确性[64-65]。Avinux[66] 静态验证工具针对操作系统内核，扩展了 SLIC 描述语言。该工具对单个预处理的源代码文件进行缺陷检测。此外，DDVerify[67] 针对操作系统的驱动程序进行场景建模，并对其中的四种场景进行验证。其他通用验证工具也能够支持

部分 API 误用缺陷类型，例如 CPAchecker[68]、CBMC[69]、SMACK[70] 等。

尽管程序验证技术能够支持 API 误用缺陷检测，但是其面临如下不足：（1）规约描述语言（Behavior Specification）能够有效地描述接口使用的约束条件。然而，现有的描述语言多针对接口的功能实现[71-72]。虽然 SLIC 是针对接口使用设计，但是该语言针对于 Windows 驱动程序设计，难以应用于普适性的接口误用缺陷检测。同时该语言使用复杂，需要深刻理解语言以设计相应的规约条件。（2）尽管验证技术能够实现可靠甚至完备的程序正确性分析，然而该技术的可延展性难以满足现实开发者的需求。面对真实项目，状态空间爆炸问题（State Space Explosion Problem）是验证技术难以应用的最重要的挑战之一。

静态分析技术　静态分析技术与验证技术不同，其目标在于通过强大且灵活可控的抽象方法对大规模软件代码高效分析，以完成对源代码的缺陷检测。该技术能够在开发的早期使用，有效地降低缺陷检测和修复的成本[73]。因此，近年来静态分析技术成为代码质量保证的有效途径之一，并被广泛使用。静态分析工具在代码的编译时或基于编译后的中间表达（Intermediate Representation），利用预先定义好的规则基于程序信息对接口误用缺陷进行检测，即在程序中是否能够发现某些代码违反了正确的编程规则。例如，API 的参数是否为空指针[74]，Linux 内核 API 是否使用正确[75] 等等。

近二十年来，研究人员和开发者设计并实现了大量的开源静态分析工具[76]。支持接口误用缺陷检测的代表性工具包括：Clang Static Analyzer（Clang-SA）[77]，Cppcheck[78]，Infer[79]，Sparse[80]，Splint[81]，SSLINT[82] 等等。Clang Static Analyzer（Clang-SA）[77] 是开源编译器框架 LLVM[83] 的重要模块之一，能够独立运行。该工具利用符号执行技术（Symbolic Execution）[84] 推理程序的语义信息，并通过检测插件（checker）的形式对程序中的缺陷进行检测。Cppcheck[78] 工具将代码预处理为符号（token）流，并在符号流中通过模式匹配的方法进行缺陷检测。为支持用户自定义的接口，Cppcheck 提供一套轻量级的规约描述方法，以供使用者撰写项目特定的接口使用约束。与上述两种通用缺陷静态分析工具互补，Facebook 公司的 Infer 工具关注内存安全，Sparse 和 Splint 工具针对 Linux 操作系统设计，SSLINT 则通过对 SSL 协议建模以对安全相关 API 误用进行缺陷检测。

为能够支持大规模、多领域、快速检测的需求，静态分析技术一方面将规则编码在分析引擎或者检测插件中，另一方面采用抽象的方法简化程序结构。因此针对于大规模复杂的 API 误用缺陷检测，其面临如下不足：（1）缺陷模式硬编码，对用户自定义 API 支持不足。例如，Clang-SA 提供大量的检测插件，然而其无法快速支持用户自定义 API，即该工具缺少轻量级的接口以供使用者开发新的检测

器。虽然 Cppcheck 提供的规约描述方法能够有效地缓解用户自定义接口缺陷检测困难。但是其语义简单，难以应对复杂的接口误用缺陷模式。（2）检测结果难以满足用户实际要求。为了能够给使用者提供好的用户体验，静态分析工具往往采取保守策略，即只报告高置信度的缺陷。因此，大量的实际缺陷被忽略，产生漏报。如果提高报告率，则产生大量的误报，即报告的缺陷实际为正确使用。API 使用多包含复杂的上下文语义信息，甚至跨越多层函数调用。同时，针对不同的路径条件，API 使用的约束条件不同。现有的静态分析工具多基于语法结构进行分析，忽略路径信息和过程间（inter-procedural）语义信息，以应对大规模快速分析的需求。因此，会产生大量的漏报以及误报。（3）检测结果展示有效性不足。目前，静态分析工具生成的报告有效信息难以辅助开发者理解缺陷与修复。为帮助开发者对缺陷进行修复，提高缺陷检测结果的展示形式是促进检测工具应用的重要因素之一[35]。

规约挖掘技术　为了弥补传统静态程序分析技术规约撰写困难、难以支持用户自定义接口的不足，近年来研究人员通过基于数据驱动的挖掘技术来自动推理 API 使用约束[85]。这些技术核心思想是在大规模代码库中，出现次数多的使用形式为正确的模式，出现次数少的则为误用。通俗来说，首先通过挖掘技术学习程序中 API 使用的约束。此后，通过这些约束进行异常检测。随着开源代码库（例如 Github）与代码挖掘平台（例如：BOA 平台[86]）的蓬勃发展，大量的工作被投入到规约挖掘技术与工具研究中，并成功应用于实际项目中[30]。

基于数据挖掘规约的推理技术，最早由 Engler 等于 2001 年提出[87]。该方法通过用户自定义的规则模板，在代码中自动推理 API 使用约束。例如，接口 `A` 必须与接口 `B` 成对出现。此后，Li 等基于频繁项目挖掘技术（Frequent Itemset Mining）[88]，设计并开发了全自动的规约挖掘工具 PR-Miner[89]，有效地提高这类方法的精度，增加支持的缺陷模式种类。此后，研究人员不断地设计并实现新的推理技术以支持更多的规约模式。例如，带有条件的因果调用关系（causal relationship）[90]，调用序列[91]，接口调用前置条件[92]，异常处理[93]，以及多种 API 混合使用约束[94]等等。总结来说，这些方法的不同在于如何对源代码进行标准化表示、如何统计 API 使用的次数、如何区别上下文关系和如何通过学习的规约条件进行缺陷检测等等。

尽管规约挖掘技术能够自动化学习 API 使用约束，同时进行误用缺陷检测，这些方法在实际应用中经常产生大量误报与漏报。其主要原因包括：（1）难以获得足够的训练数据。基于数据驱动的学习技术需要大量的有效数据以学习正确的规约条件。然而在单个项目中，难以满足数据量的需求，特别是用户自定义的接

口[30,55]。（2）无关语句影响学习结果。现有的学习技术多基于语法结构进行分析，缺失的语义信息导致在学习过程中无关语句对结果影响巨大[95]。（3）无法学习隐式规约条件。基于学习的方法只能够学习显式的规约条件，而对于 C 程序无法显式表示的条件则无法学习。例如：C 标准库的 `free()` 接口不可以释放非堆内存，不然会导致一个释放非堆内存（CWE-590：free of memory not on the heap）错误。然而 C 程序语法中无法描述该属性。

总结　程序接口误用缺陷检测的三大类技术路线（代码审查、动态检测和静态检测技术）具有各自的特点和不足。其中，动态检测和静态检测能够实现自动化或者半自动化，目前较为受开发者的欢迎。动态检测需要执行目标程序，在运行时或基于运行时信息进行缺陷检测。因此，需要可执行的程序，并且存在覆盖率低和人工构造测试环境困难等不足。特别地，针对部分误用缺陷模式（例如：异常处理、资源泄漏等）需要特殊环境和大量的时间触发缺陷。静态分析则可以在开发早期进行，并能够覆盖全部程序，不需要大量人工参与。但是，存在分析精度、规模等不足。本文旨在发现实际项目中接口误用缺陷，在开发早期尽可能保证接口正确使用。因此，本文选择静态分析技术进行研究。

1.3　研究思路

随着软件规模增长、代码复杂度提高以及开源代码的广泛使用，现有的 API 误用检测方法面临巨大挑战。因此，本文旨在提高接口误用缺陷检测能力，以应对实际项目中接口误用检测精度与规模的矛盾关系。即在大规模实际项目中支持多种接口误用缺陷类型的同时，尽可能地保证检测的精度。特别地，C 语言多应用于操作系统、嵌入式系统、服务器、数据库等基础软件，需要极高的可靠性和安全性。因此，本文以 C 语言作为载体。针对上述已有研究面临的问题，本文将提出基于静态分析的 C 程序接口误用缺陷检测技术和相关的支持工具集合 Tsmart-IMChecker。整体的研究思路如图 1.2所示。本文将从自底向上的 3 个层次展开研究：表示层、分析层和应用层。表示层关注接口使用约束的描述，侧重接口使用约束的描述能力。对 API 使用约束条件进行描述是接口缺陷检测的重要基础。因此，该研究对有效检测接口误用缺陷至关重要。分析层关注接口误用缺陷检测的方法，侧重检测方法对实际程序中规模与准确性的平衡关系。随着开源社区的发展，现代软件呈现出大规模、复杂化的特点。因此，高效、准确的检测方法，是提高代码质量的重要方法。应用层关注接口误用缺陷检测方法在实际项目中的应用效果，侧重接口误用测试数据集整理和工具在实际中的应用效果。本文将总结 C 程序接口误用

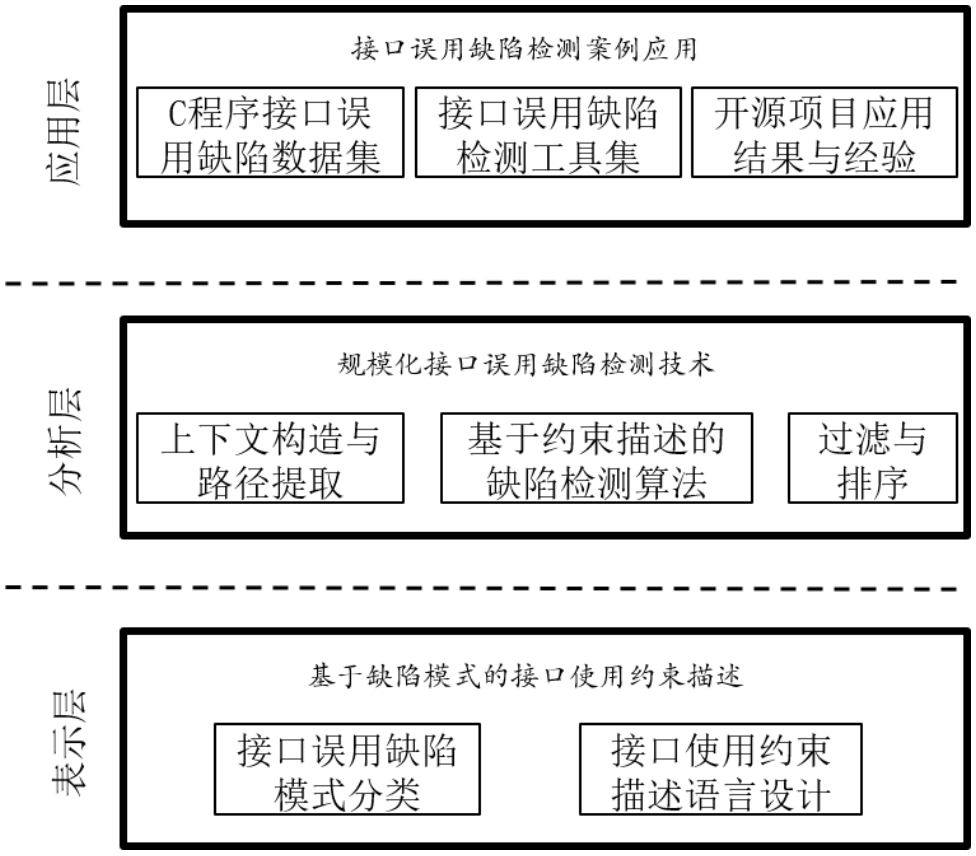

图 1.2　论文研究思路

缺陷数据集，开发可视化支撑接口误用缺陷检测工具集。并将工具在开源项目应用的经验进行总结，以帮助研究人员和开发人员更好地理解接口误用缺陷。

本文的具体研究思路如下：

1. **基于缺陷模式的接口使用约束描述**。规约描述语言能够有效地定义 API 使用的约束条件。然而当前已有规约描述语言多关注于接口的实现或特定应用场景。自动规约挖掘技术则面临有效数据集缺失的问题，难以支持实际项目中接口误用缺陷的特性。因此本文选择接口误用缺陷模式作为切入点，设计一套轻量级的接口使用约束领域特定语言。（1）为理解 C 程序接口误用缺陷的常见模式，本文将对不同领域的开源项目中的实际接口缺陷报告进行调研，总结常见接口误用缺陷模式。（2）针对 C 程序接口误用缺陷的常见模式，设计一套面向 C 程序接口使用约束的领域特定语言。该语言应支持多种接口使用约束模式的描述，同时易扩展到新的接口缺陷类型。本部分重点研究接口使用约束描述的表达能力，是本文研究的重要基础。
2. **规模化接口误用缺陷检测技术**。现代软件具有规模大、结构复杂的特点。因此，静态分析的分析效率（大规模程序）与检测准确性（误报率和漏报率）难以同时满足。针对两者平衡这一关键问题，面向接口误用缺陷的特点，本

文关注如何对静态分析任务进行分而治之，以缓解分析过程中路径、状态爆炸等问题。同时，在支持大规模代码分析的同时，做到对局部代码的精确分析。(1) 针对大规模目标程序，本文将采用多入口分析策略，提取和接口缺陷相关的抽象符号路径语义信息，并设计基于约束描述的接口缺陷检测算法，以支持用户自定义的接口。(2) 为降低多入口分析带来的精度损失，本文将通过基于上下文的语义信息和基于使用情况的的统计信息，对检测结果进行过滤与排序，以提高检测的精度。本部分重点研究检测方法对分析效率和准确性平衡关系，是本文研究的核心技术方法。

3. **接口误用缺陷检测应用**。为帮助研究人员更好地理解 C 程序接口误用缺陷，满足开发者的使用需求，本文将提供 C 程序接口误用缺陷数据集 APIMU4C 与 C 程序接口误用缺陷检测工具集 Tsmart-IMChecker。(1) 为帮助研究人员与开发者更好地理解 C 程序接口误用缺陷，本文将对接口误用缺陷模式调研中的实例进行总结。同时，为更好地评估现有接口缺陷检测工具的能力与引导新的检测技术的开发，本文将结合现有测试数据集与接口误用缺陷模式，构造针对 C 程序的接口误用测试数据集。(2) 如何降低用户的使用代价和提供有效的缺陷展示功能以辅助开发者理解缺陷是使用者对静态分析工具的迫切需求。因此，为提高检测工具的用户友好性以及辅助使用者理解缺陷检测结果，本文将开发一系列可视化的工具来辅助开发者进行接口使用约束的撰写、缺陷检测和结果分析。(3) 最后，本文将在开源项目中进行工具集的案例应用，并总结应用的结果和经验。本部分重点研究针对 C 程序接口误用缺陷的数据集以及工具应用，是本文方法的实际应用。

1.4　论文贡献

本文根据上述研究思路，针对 C 程序接口误用缺陷的静态检测技术进行系统性研究，取得相应的理论效果，实现了对应的工具集合[96-98]。论文的具体贡献如下：

1. 提出用于描述 C 程序接口使用约束的领域特定语言以及规模化接口误用缺陷检测方法。首先，为理解 C 程序接口误用缺陷的特点，本文对不同领域、广泛应用的六个开源项目中 830 个实际 API 误用缺陷修复报告进行调研，共总结出三大类常见接口误用缺陷模式。基于缺陷模式，本文设计 C 程序接口使用约束的领域特定语言 IMSpec，并与传统自然语言描述方式对比。实验结果表明，IMSpec 能够有效描述多种接口使用约束，帮助开发者多找到 12.5% 的误用缺陷实例。同时本文设计并实现基于约束描述的规模化接口误用缺陷

检测方法。该方法利用多入口分析策略、抽象路径提取和基于程序语义和使用情况的过滤排序技术，将规模化代码分析任务分而治之，达到高效率分析的同时，实现局部的精确分析。本文选择 Juliet Test Suite[99] 中 13 个接口缺陷相关 CWE 分类的共 2172 个程序进行评估。结果表明，本文提出的方法误报率和漏报率分别为 13.21% 和 16.80%，优于主流的开源静态分析工具。

2. 总结 C 程序接口误用缺陷数据集并开发基于图形化的检测工具集。基于开源项目和公开测试数据集，本文提供 C 程序接口误用缺陷数据集 APIMU4C。据作者所知，这是第一个针对 C 程序接口误用缺陷的数据集。APIMU4C 数据集能够帮助研究人员和开发者理解 C 程序接口缺陷，以及评估工具的缺陷检测能力。同时本文设计并实现可视化支撑的 C 程序接口误用缺陷检测工具集。在工具集中，本文集成规约撰写工具 IMSpec-writer、接口缺陷分析引擎 IMChecker-engine 和基于差异性对比的结果展示工具 IMDisplayer，以辅助开发者撰写 IMSpec 约束文件和理解缺陷检测结果。本文将 Tsmart-IMChecker 工具集应用于广受关注的开源软件中，在 Linux 内核、OpenSSL 库以及 Ubuntu 操作系统中找到 112 个新的接口误用缺陷。至今，在 75 个提交的缺陷报告中，61 个缺陷被开发者确认，其中 32 个已经被开发者在主分支修复。本文对实际应用结果和经验进行总结，供研究人员和开发人员参考。

1.5　论文结构

本文共包括 5 个章节。第 2 章介绍针对接口使用约束的领域特定语言，包括基于实际案例的接口误用缺陷模式以及领域特定语言的设计思路、语法和语义。第 3 章讨论规模化接口误用缺陷静态检测方法。第 4 章对接口误用缺陷数据集和检测工具集进行介绍，并对工具集在开源项目的应用结果进行总结。最后，第 5 章总结全文并展望延伸本文工作的若干方向。

第 2 章　接口使用约束描述方法

随着软件规模的提升与开源社区的蓬勃发展，开发者经常利用现有的应用编程接口（API）来快速构建系统。在使用 API 完成特定功能时，需要满足对应的约束条件。例如：参数的有效性，正确的调用序列等等。违反这些约束中的一条或多条，则会产生接口误用（API misuse），导致程序错误、系统崩溃，甚至被攻击者利用。对接口使用约束进行准确描述，是避免接口误用的前提条件。接口行为规约描述语言（BISL）提供面向代码层次的形式化描述方法，能够有效地帮助开发者理解 API 的行为以及使用时需要满足的约束条件。因此，研究面向接口使用特点的规约描述语言具有重要意义。

本章首先针对不同领域的开源 C 程序中接口误用缺陷实例进行分析，总结常见缺陷模式以及违反的约束条件。接着基于缺陷模式，提出用于描述 C 程序接口使用约束的领域特定语言 IMSpec，并对语言的设计动机进行介绍，定义该语言语法和语义。本章将 IMSpec 应用于实际项目中的接口实例，以验证语言的有效性。从全文的研究体系上看，本章的工作旨在通过形式化的方法对接口使用约束描述，是接口缺陷检测工作的重要基础。

2.1　引言

开发者在使用 API 构造软件系统时，需要满足特定的使用约束条件以正确地完成相应的功能。例如：API 参数为指针类型时，该指针不可以为空，否则可能会产生一个空指针解引用错误；当通过内存管理接口申请内存资源后，需要使用相应的释放接口以归还资源，否则将产生一个内存泄漏错误。这些由于误用 API 而产生的缺陷是软件错误、系统崩溃的重要原因之一，甚至会被攻击者利用，带来巨大影响。

为保证 API 的正确使用，一方面，API 的设计者提供各种各样的文档、应用案例，以帮助使用者理解 API 的功能和对应的使用约束条件。然而，现在的文档形式难以满足实际需求[17]。更严重的是，随着开源软件的蓬勃发展，大量的软件库没有完整的文档资料，甚至没有或者存在错误的使用说明[18-19]。同时，相对于直接查找官方的 API 使用文档，更多的开发者通过网络搜索来快速地找到相应的使用方法。但是网络资源的可靠性难以保障[32]。另一方面，研究人员通过缺陷检测的方法对 API 误用进行查找，以提高代码质量。然而，现有的检测方法难以满足实际需求。基于静态分析技术的检测工具，多通过预先实现的检测器来进行缺陷

查找[28]。因此，该方法难以找到未预先定义的接口误用缺陷。基于数据挖掘技术的方法，通过推理 API 使用约束，再基于这些约束来进行缺陷检测。然而，现有的数据集质量难以满足学习算法的数据要求[30]。因此，无论是 API 的设计人员还是缺陷检测的研究人员，如何有效地定义 API 使用约束是保证接口正确使用的重要基础。

BISL 提供面向代码级别的形式化描述方法，能够有效地帮助程序员理解 API 的行为以及使用中需要满足的约束条件[100]。通俗来说，这些规约描述为接口的开发者和使用者提供一种形式化的契约模式 (software contract)[101]，即这些规约描述通过形式化的表达形式定义 API 在使用时需要满足的特定约束条件。然而，针对于 API 使用约束的描述，现有的 BISL 具有若干不足：（1）针对普适性程序特征的 BISL 多面向接口实现而设计，有利于描述 API 的内部属性。随着软件的规模和复杂性增加，API 的使用情景复杂化。普适性 BISL 难以描述 API 使用约束条件，即易用性不足。（2）针对接口使用约束的 BISL 通常为某个特定领域的接口而设计，语言表达能力不足，难以应用到普适性 API 使用约束条件，即完备性不足。例如：SLIC[60] 针对于 Windows 驱动程序设计，SSLINT[82] 针对于 SSL 协议中十余个核心接口设计。

为解决上述方法中的不足，本章提出 IMSpec 领域特定语言（DSL），一个基于缺陷模式的 C 程序接口使用约束描述语言。首先，为能够深入理解 C 程序接口误用缺陷的特点从而总结接口使用约束模式，本章以不同领域、广泛使用的六个开源软件为对象，对近五年来 830 个实际接口误用实例进行分析。本章共总结出三大类常见接口误用模式：（1）不正确的参数使用（Improper Parameter Using, IPU），（2）不正确的异常处理（Improper Error Handling, IEH），（3）不正确的因果调用关系（Improper Causal Calling, ICC）。这些缺陷模式一方面可以为描述语言的设计提供基础，简化语言的复杂度。另一方面，有利于辅助研究人员和开发者理解实际项目中的缺陷模式，从而提高接口设计以及避免接口误用。基于接口误用缺陷模式，本章提出面向 C 程序接口使用约束的领域特定语言 IMSpec。本章对 IMSpec 的设计原则进行介绍，并定义语法结构与形式化语义。最后，本章将 IMSpec 应用于调研中发现的典型缺陷实例中，以评估该语言的描述能力和有效性。结果显示，IMSpec 能够支持调研实例中 90.67% 的接口使用约束，对接口使用约束具备描述能力。同时，本章通过调查问卷邀请实际开发者来对误用实例检测的方式来评估 IMSpec 的有效性。实验结果显示，相比于自然语言实际开发者通过 IMSpec 描述的接口使用约束能够多找到 12.50% 的缺陷，即相对于自然语言，IMSpec 能够更加有效地对接口使用约束进行描述。

本章其余部分组织结构如下：2.2节对相关研究进行总结；2.3节给出接口误用缺陷调研的方法并总结调研结果；2.4节给出 IMSpec 的设计思路，定义语言的语法结构与形式化语义；2.5节对 IMSpec 进行实际案例应用与评估；最后在2.6节总结本章工作。

2.2　相关工作

本章相关的研究包括两个方面：接口使用相关的调研工作与规约描述语言的设计工作。本节将分别对这两方面内容进行总结。

接口使用调研　过去的二十年内，研究人员对接口使用从不同角度开展大量的调研工作以保证其被正确使用 [22,102-110]。一方面，研究人员从 API 设计和实现入手，对 API 文档[104-106]、API 演化[107,109] 等内容进行分析；另一方面，则从使用者的角度，对 API 使用中的问题进行分析[22,102-103,108,110]。

随着软件库的更新，新的 API 用来替换过期的 API，以增加稳定性、提升执行时效率。使用这些过期的 API，是 API 误用的一种。Robbes[104-105] 在超过 2600 个系统中对 577 个不同的 API 进行研究，以分析使用过期 API 带来的影响。其结果显示，虽然只有 14% 的 API 与其他的系统模块相关，但是在升级后这些改变会带来极大的影响。其中最多的一个 API 实例涉及到 80 个开发者和 120 个项目。然而，很多开发者对此并没有做出相应的更新措施，极大地降低系统的可靠性。

从使用的角度，Zhong[102] 针对于不同类型 API 的使用情况进行调研分析。从规约挖掘技术的需求，共总结出 9 个与 API 使用相关的发现。特别地，该论文指出在对 API 使用进行规约描述时，需要考虑非顺序调用关系、类型信息和顺序关系三个特点。同时，随着多核处理器的发展，开发人员可以利用并行化技术来加快问题求解。

Okur[103] 对 655 个开源项目中并行计算软件库中的 API 使用情况进行分析。结果显示，超过 10% 的开发者误用这些 API，导致程序并没有并行化运行，而是在串行执行。同时，由于对 API 使用约束的生疏，开发者撰写的代码复杂度高，难以理解与维护。

与 Okur 的工作类似，Nadi[22] 针对加密算法的 API 使用情况进行分析。该研究针对 100 个 StackOverflow 的问题与回答、100 个 Github 上的开源项目和 48 个实际开发者进行调研。其结果显示，尽管开发者认为这些 API 使用难度大，但是他们依旧坚信能够正确使用这些接口。然而，在实际项目中加密算法相关 API 误用却普遍存在[9]。

Robillard[108] 从开发者的角度对 API 使用进行调研。通过多阶段调研问卷与面对面讨论的方式，该研究发现开发者普遍认为 API 学习难度大。其主要原因是现有的文档形式难以有效地帮助使用者理解如何正确使用接口。特别地，缺少足够的使用样例是文档最大的不足。因此，虽然 API 的设计者提供格式良好的文档，但是开发者多利用网络资源以快速地掌握 API 的使用情况。

针对接口误用，现有的接口使用调研工作存在若干不足。首先，针对接口误用缺陷 Okur 对并行化 API 的缺陷模式进行分析，同时 Nadi 对加密算法相关的 API 使用情况进行分析。这些方法都只针对于某一个特殊领域，其结果难以直接适用于普适性接口误用缺陷模式。其次，大部分调研工作多针对面向对象编程语言展开[22,102-104]，其结果难以直接应用于 C 程序。因此，针对 C 程序接口误用缺陷进行调研，总结常见缺陷模式以指导接口约束描述方法设计具有重要意义。

规约描述语言　软件行为规约（behavorial specification）是对软件系统或者组件预期行为的精确描述。独立的代码实现并不能有效描述其功能和使用约束，因此通过规范形式记录下来的信息对于软件维护有着重要作用，能够有效地定义 API 开发者和使用者之间的协议[101]。特别地，形式化规约描述语言能够消除自然语言的歧义。规约能够在软件的整个开发周期使用，一方面开发人员可以根据规约进行内部功能实现；另一方面测试人员能够在调试阶段根据规约去分离错误，划分责任[111]。近三十年来，研究人员针对不同的语言和目标，设计并实现了各种各样的 BISL[100]。例如，针对通用属性检测的 BLAST[71]、ACSL[72]；针对领域特定的 BISL 包括：SSL 协议的 SSLINT[82]、Windows 内核驱动程序的 SLIC[60]、Epex 工具中对异常处理检测的规约[112] 等等。

BLAST[71] 由 Dirk Beyer 教授提出，是面向时序安全属性的规约描述语言，用于 BLAST 自动化验证工具。该语言从两个不同的精度水平对程序属性进行描述。微观来说，该语言能够通过描述监控自动机（monitor automata）的内部转移，在程序的运行时轨迹上对程序时序属性进行分析。宏观来讲，该语言可以通过撰写可达性查询语句，对程序的状态与位置进行查询。通过两个精度水平的描述，BLAST 能够有效地将验证问题转化到多个独立的模型检测引擎中，从而降低验证工作的复杂性。

ACSL（ANSI/ISO C Specification Language）是 Frama-C[113] 代码分析平台用来形式化定义 C 程序属性的规约描述语言。该语言通过注释的方式对程序中属性进行描述，以辅助验证工具对代码实现情况进行检测。ACSL 注重函数合约（function contract），即函数的参数与函数执行后需要满足的性质。其中，前者也被称作前置条件（pre-condition）；后者被称作后置条件（post-condition）。特别地，前置条件

多针对于 API 使用者，即在调用目标 API 之前需要满足的约束。ACSL 以代码注释的形式撰写，因此为利用 ACSL，分析工具需要理解 ACSL 的语法与语义，同时将 ACSL 与目标 C 程序的源代码进行转化。

针对于 C 程序接口误用缺陷检测，微软公司 SLAM 项目[59] 是典型代表之一。开发者通过使用 SLIC 规约描述语言[60] 对程序接口的属性进行描述，并使用基于反例引导抽象解释技术的 SDV 验证工具对目标 API 使用情况进行检查。至 2010 年，该项目已经积累超过 200 个 API 使用规约，成功检测到 Windows 操作系统驱动程序中 270 个 API 误用缺陷，有效提高接口使用的正确性[64-65]。此外，SSLINT 基于程序依赖图（program dependency graph）对 SSL 安全相关的 API 进行建模并通过 Cypher[114] 图查询语言对预定义好的模式进行查询，以检测 API 误用。Epex 对 API 返回值的约束条件进行描述，并利用规约对 C 程序中接口执行失败后不正确的异常处理代码进行检测。

现有规约描述语言对于 C 程序接口使用约束的描述存在若干不足。一方面，通用语言针对多种程序属性设计，通常为接口实现而设计，语法结构多样，语义丰富。然而，接口使用多含有复杂的程序结构，涉及到多个 API 的协同使用。因此通用语言描述使用约束需要复杂的组合。另一方面，现有针对接口使用的描述语言多针对某个特定领域或者某些接口实例设计，难以扩展到普适性的接口使用约束模式。例如 SLIC 语言能够有效地应用于 Windows 操作系统驱动程序，却难以应用于 SSL 安全库中的接口。

2.3　接口误用缺陷分类

为能够在实践中更加有效地解决 API 误用问题，对 API 误用缺陷的特点进行深入研究以总结 API 使用约束模式具有重要意义。一方面，可以通过现有缺陷分类标准进行总结；另一方面则可以通过现有接口误用缺陷检测研究中的缺陷模式进行概括与总结。

现有缺陷模式分类标准　IEEE 组织在 1993 发布 IEEE-1044 软件异常分类，并在 2009 年更新[115]。基于该标准的内容，IBM 公司提出 Orthogonal Defect Classification（ODC）分类标准[116]。ODC 基于缺陷类型（defect type）对缺陷进行分类，即通过代码结构的组成进行分类。例如函数、检测、赋值、文档等等。2016 年 Beller[117] 基于 ODC 提出 General Defect Classification（GCD），从而更加精确地比较静态分析技术的检测能力。然而现有的缺陷分类方式，一方面没有对所有的软件缺陷类型进行描述，因此无法涵盖所有接口误用缺陷域；另一方面缺少对 API 误用缺陷

的详细分类情况。

基于缺陷检测的缺陷分类　过去的十几年内，研究人员对不同的接口误用缺陷模式进行检测。例如，Monperrus[10] 指出，缺失必要的 API 调用普遍存在于缺陷跟踪系统、论文和源代码以及代码的注释中。Thummalapenta[118] 针对 API 调用前需要满足的前置条件进行研究。Wasylkowski[119] 则对接口调用顺序相关错误进行检测。然而，这些研究都只针对接口误用缺陷中某一个特定的种类。Adama[30] 在其研究结果中针对 Java 程序提出 API-Misuse Classification（MUC），从而对 API 误用缺陷进行分类，并比较现有工具的检测能力。该分类基于 API 使用元素与缺陷表现类型进行分类。一个缺陷类型具体可以表现为两种异常类型（violation type），即缺失（missing）或者误用（misuse）。本文针对 C 程序的 API 缺陷进行研究，C 程序与 Java 程序存在巨大设计理念差异。该结果难以直接应用于本文研究内，但是对本文的研究具有重要参考意义。

另一方面，研究人员从 API 使用本身入手，调研并分析面向 API 使用的特点，总结接口误用分类[17,120]。API 使用说明，是 API 文档中的一段自然语言描述，可以用来提醒开发者在使用 API 时需要满足的约束条件。当这些约束条件被使用者忽略或者违反时，认为使用者产生一个 API 误用。然而只有部分说明可以直接对应于接口误用以及产生的缺陷，其他的则更注重 API 的内部功能或者特殊用法。例如：明确地说明 API 的参数可以为空是针对 API 使用中不同方式的引导。不过这些调研结果可以作为本文的重要参考内容。

据本章调研结果显示，目前并没有研究工作对 C 程序接口误用缺陷的问题域进行定义或描述。因此，研究人员难以直接对现有研究进行系统的了解与总结。特别是哪些 API 缺陷种类已经被研究过、哪些没有被研究过、现有的方法能够解决多少问题以及不同方法之间的效果比较等等。因此，为更好的应对 C 程序 API 误用缺陷检测以及接口使用约束描述语言的设计，本章需要一个来源于实际程序、具有接口使用特定领域的缺陷类别。本节剩余部分将对数据收集和分类结果进行详细描述，并对接口误用调研结果进行讨论。

2.3.1　数据收集

本小结将针对数据收集的主要步骤进行详细描述，包括研究对象、缺陷实例收集方法和缺陷分析方法。

研究对象　现代软件开发模式多利用开源社区已有的实现加快开发速度，因此对开源软件的研究具有重要意义。为能够更好地理解实际项目中 C 程序 API 误用缺

表 2.1　API 误用缺陷研究对象。

研究对象	关注数①	提交次数	可用时间②
Linux 内核	69,697	812,391	20050416-至今
OpenSSL	9,510	23,412	19981221-至今
FFmpeg	13,907	93,218	20001220-至今
Curl	24,009	12,130	19991229-至今
FreeRDP	2,959	13,030	20110630-至今
Httpd	2,028	31,341	19960703-至今

① 本文选取 Github 上 Watch 和 Star 中较多者，统计时间为 2019 年 2 月 28 日。
② Github 上可以追溯的代码修改时间。

陷的特点以及开发者如何对这些缺陷进行修复，本章对不同领域、广泛使用的六个开源软件进行分析。如表2.1所示，这六个项目为：

1. Linux 内核[121]：Linux 内核是一种开源的类 Unix 操作系统内核，由芬兰赫尔辛基大学学生 Linus Torvalds 于 1991 年创建。该内核由一系列的程序组成，包括中断服务程序、负责管理多个进程从而分享处理器时间的调度程序、负责管理地址空间的内存管理程序、网络、进程间通信的系统服务程序等。随着开源社区的发展、软件日益复杂的功能和各种硬件的发展，越来越多的驱动程序被集成在 Linux 内核中。Linux 内核在近年来发展迅速，代码量已经超过 13MLOC。
2. OpenSSL[15]：安全套接层协议（SSL）可以在网络上对传输内容进行加密，以提供秘密性传输的功能。OpenSSL 则是实现该协议的开源软件库。该库提供三个主要的功能模块：SSL 协议库、密码算法库以及应用程序。OpenSSL 提供强大和全面的功能，囊括主要的密码算法、常用的密钥和证书封装管理功能以及 SSL 协议，并提供丰富的应用程序供测试或其它目的使用。应用程序可以使用 OpenSSL 进行安全通信，避免窃听。
3. FFmpeg[122]：FFmpeg 是一套针对多媒体处理的开源软件库与应用程序的集合，可以用来记录和转换数字音频、视频，并能将数字媒体转化为流数据。该库由若干子项目组成，包括：多媒体编码解码算法、公用工具函数库、视频场景处理库、后期效果处理库、格式转换、基于 HTTP 多媒体即时广播串流服务器以及多媒体播放器等等。
4. Curl[123]：Curl 是一个利用 URL 语法[124]，在命令行下工作的文件传输工具。该工具支持文件的上传和下载，被称作为综合传输工具。Curl 支持多种通信协议，同时还提供多种安全验证机制。自 1997 年首次发行以来，该工具被开发者广泛应用，吸引上千开发者的贡献。已经成功应用于汽车、电视、交换

机、打印机、手机、平板电脑等多个领域。同时，Curl 还提供 libcurl 接口库供程序员开发使用。

5. FreeRDP[125]：远程桌面协议（Remote Desktop Protocol，RDP）是一个多通道的远程连接协议，能够让客户端连接到提供远程服务的 Windows 服务机。FreeRDP 是一款基于 Apache 协议实现 RDP 的开源软件。目前该软件已经成功应用于多个 Linux 发布版以及 Mac 系统。
6. Httpd[126]：Httpd 是 Apache 开源组织研发面向超文本传输协议（HTTP）服务器的主程序。该程序以独立运行的后台进程存在，并通过子进程或者线程池形式对请求进行处理。Httpd 代码开源，并被广泛应用于各种网络服务中，目前已经成为网络上最受欢迎的服务器之一。

上述研究对象涵盖不同的领域：操作系统（Linux 内核），开源 API 库（OpenSSL 和 FFmpeg），Ubuntu 应用软件（Curl，FreeRDP 和 Httpd）。具有多年的开发历史，并且现在依旧活跃于开源社区中，被应用人员广泛使用。一方面，活跃的开发者和用户社区有利于缺陷的报告和修复；另一方面，被广泛使用使得这些软件的代码质量更具有代表性。特别是针对于代码质量，大量的使用者和开源社区贡献能够快速地对程序中的缺陷进行修复。同时，这些项目都在 Github 网站上开源、具有完整的修改记录和缺陷跟踪系统。这些修改记录和缺陷跟踪系统有利于缺陷实例的收集和总结。因此，本文选择以上项目作为分析对象。

缺陷实例收集　本章通过对软件开发版本修改记录日志分析，以完成 API 误用缺陷实例收集工作。如表2.2所示，本章在六个开源项目中，共对 61096 个修改记录进行分析，在 18476 个缺陷修复记录中收集 3150 个 API 误用缺陷实例。下文将详细描述缺陷实例的收集方法。

首先，通过 Github 上的日志记录追踪系统，对所有的修改记录进行下载和备

表 2.2　API 误用缺陷研究对象

项目名称	Loc	研究时间	修改总数	缺陷数目	API 误用数目
Linux	12.96M	20170901-20171231	24651	6401	868
OpenSSL	454K	20150701-20171231	7564	2391	529
FFmpeg	915K	20160701-20171231	8162	2783	610
Curl	113K	20130101-20170630	7082	2043	499
FreeRDP	259K	20130701-20171231	7565	3535	495
Httpd	203K	20130701-20171231	6072	1323	149
总计	14.90M	-	61096	18476	3150

"hash": "059c98599b1ab1e2e479e1f4617948d4c2a32b84",

"summary": "wl18xx: add checks on wl18xx_top_reg_write() return value",

"description": "wl18xx: add checks on wl18xx_top_reg_write() return value. Check return value from call to wl18xx_top_reg_write(), so in case of error jump to goto label out and return. Also, remove unnecessary value check before goto label out.

Addresses-Coverity-ID: 1226938

Signed-off-by: Gustavo A. R. Silva <garsilva@embeddedor.com>

Signed-off-by: Kalle Valo <kvalo@codeaurora.org>",

"date": "2017-06-28 21:18:40",

"author": "Gustavo A. R. Silva",

"parent_hash": "69551f5f370cc20342fab17ca54716b6ec7e332d"

图 2.1　Linux 内核修改 sha：059c98599 的修改说明

```
drivers/net/wireless/ti/wl18xx/main.c
==========================================================
lhs: 100644 | d1aa3eee0e81f8cc7612eddabc4cf630dbbd1e79
rhs: 100644 | 0cf3b4013dd646104b1f153c3d8a191f12a5dba8
@@ -793,9 +793,13 @@ static int wl18xx_set_clk(struct wl1271 *wl)
ret = wl18xx_top_reg_write(wl, PLLSH_WCS_PLL_P_FACTOR_CFG_2,
   (wl18xx_clk_table[clk_freq].p >> 16) &
   PLLSH_WCS_PLL_P_FACTOR_CFG_2_MASK);
+      if (ret < 0)
+         goto out;
} else {
ret = wl18xx_top_reg_write(wl, PLLSH_WCS_PLL_SWALLOW_EN,
   PLLSH_WCS_PLL_SWALLOW_EN_VAL2);
+      if (ret < 0)
+         goto out;
}

/* choose WCS PLL */
@@ -819,8 +823,6 @@ static int wl18xx_set_clk(struct wl1271 *wl)
/* reset the swallowing logic */
ret = wl18xx_top_reg_write(wl, PLLSH_COEX_PLL_SWALLOW_EN,
   PLLSH_COEX_PLL_SWALLOW_EN_VAL2);
-  if (ret < 0)
-      goto out;

out:
return ret;
```

图 2.2　Linux 内核修改 sha：059c98599 的差异性报告

份。针对于每一次修改记录，抽取修改说明（description）、修改的差异性报告（diff 文件）以及修改前和修改后的源代码文件。图2.1给出 Linux 内核修改 sha：059c98599 的修改说明，包括：修改的哈希值、概要总结、详细说明、作者、时间以及上一版本的哈希值。根据本次修改的哈希值以及上一修改的哈希值，可以抽取（checkout）相应版本的源文件，即修改前和修改后的文件。图2.2给出该修改的差异性报告。其中以“+”开始的绿色行为增加的行，以“-”开始的红色行为删除的行。

接着对修改记录进行分析，抽取本章关心的接口误用缺陷实例。如表2.2所示，在六个项目中，本章共获得 61096 个修改记录实例。由于不同的领域背景、代码的复杂性、接口的熟悉程度等原因，在现阶段情况下，针对每一个实例进行分析难以完成。因此本章通过自然语言处理的方式，在修改说明中对接口误用缺陷进行提取：

1. 针对六个项目，本章随机在每一个项目中选择 100 个修改记录实例进行详细分析。通过 API 文档研究、代码查看、开发者讨论等方式，本章筛选出与缺陷修复（bug fix）相关的实例，并总结这些实例中与缺陷修复相关的关键词。例如："bug", "error", "fix"和"check"等等。
2. 在缺陷修复相关的实例中，进一步筛选和 API 误用缺陷相关的实例。本章利用在已发表论文中的缺陷类型以及其他针对软件缺陷分类中的类型作为基础，对缺陷的原因进行判断。如果缺陷的产生原因是在上述分类中的一种或者多种，或者是违反官方文档中接口使用约束，那么就认为这是一个 API 误用缺陷。本章对这些缺陷修改描述中的关键词进行总结，例如："fix API", "missing check", "null pointer dereference", "add check""memory leak"以及"return value"等等。
3. 基于这些关键词本章在已经提取的 61096 个缺陷实例中基于自然语言处理的方式进行文本匹配，通过筛选缺陷修改说明来获取目标研究对象中的缺陷实例。对于选中的结果，本章过滤掉所有没有修改过 *.c 源文件的修改记录。

图2.1和图2.2是一个缺陷实例。在 Linux 项目中，通过关键词的搜索，本章认为修改 sha：059c98599 的修改与缺陷修复有关。因为修改的描述中包含关键词"check"，即"Check return value from call to wl18xx_top_reg_write()"。此后在 API 误用缺陷实例的搜索过程中，该修改还包括"add checks on wl18xx_top_reg_write() return value"。因此，本章认为该修改与接口缺陷相关。如图2.2所示，该修改用来修复不正确的 API 返回值检查缺陷。该缺陷的原因以及缺陷模式将在后文中进行详细分析。

对于该方法的有效性，本章在抽取的结果中通过随机取样进行验证。在所有提取的 API 误用缺陷修复实例中，本章在每个项目中随机选择 30 个样例。通过对这 180 个实例的详细分析，发现其中 166 个是 API 误用缺陷修复实例，即准确率达到 92.22%。在所有的误报实例中，虽然修改说明中出现相应的关键词，但是修改的内容本身与 API 误用无关。例如：OpenSSL 的一个修改记录 sha：4af389 为"Fix compilation with OPENSSL API compat"。该描述中包含"fix API"关键词，然而，该修改针对于编译错误。因此，本章认为通过以上方法能够有效地在修改记录中

表 2.3　API 误用缺陷实例统计信息

项目名称	缺陷总数	API 误用		研究实例	
		数量	百分比（%）①	数量	百分比（%）②
Linux	6401	868	13.56	283	32.60
OpenSSL	2391	529	22.12	127	24.00
FFmpeg	2783	610	21.92	126	20.66
Curl	2043	499	24.42	134	26.85
FreeRDP	3535	495	14.00	119	24.04
Httpd	1323	149	11.26	41	27.52
总计	18476	3150	17.05	830	26.35

① 与缺陷总数的百分比
② 与 API 误用缺陷总数的百分比

获取 API 误用缺陷实例。

在预实验阶段，由于缺少领域知识、对 API 掌握熟练度不足、以及缺陷的复杂性，平均每个缺陷的理解时间在 3.5 小时。特别地，有的缺陷报告涉及多个文件，包含 API 缺陷修改、重构、文档等多个内容。现阶段情况下难以在有限的时间内，针对每一个实例进行详细分析。因此，本章在 3150 个 API 误用缺陷实例中，在每个项目中抽取 35% 的缺陷实例进行详细分析。针对于这部分缺陷，进一步将修改超过两个文件、修改行数大于 30 行的修改记录进行过滤。如表2.3所示，最终本章在 3150 个 API 缺陷实例中，对随机抽取的 830 个缺陷实例（26.35 %）进行深入分析。

缺陷分析　对于上述步骤收集的缺陷实例，本章人工地对修改说明文件的内容、差异性报告以及源代码文件进行分析，从而对缺陷进行深入理解。包括：接口误用上下文语义中的根本原因、修复的模式以及误用的其他统计信息等。特别地，为能够理解 API 误用缺陷的本质，本章关注 API 误用的约束条件，即在 API 使用的整个代码片段中，哪一部分的错误导致 API 误用。

以图2.2为例。通过缺陷描述可知，该缺陷产生的原因是没有对接口 `wl18xx_top_reg_write()` 的返回值进行检查。同时，差异性报告中显示该缺陷的修复方式是增加返回值 `ret` 的检查，即判断返回值是否小于零。此外本章发现，如果小于零则添加 `goto` 语句，跳转至异常处理部分。总结来说，该缺陷产生的上下文是 `wl18xx_top_reg_write()` 函数调用后，没有进行返回值检查；缺陷的根本原因则是没有对该函数进行异常处理；修复的模式则相对简单，即进行检查并在错误路径上添加跳转语句至异常处理模块。

2.3.2　调研结果

如表2.3所示，在 18476 个缺陷修复实例中，有 3150 个与 API 误用相关的修复实例，平均百分比为 17.05%。其中 Curl 占比例最多，为 24.42%；Httpd 最少，为 11.26%。在上文介绍的预实验方法和结果中，平均 92.22% 的 API 误用缺陷为真正例（True Positive，TP）。因此，本章相信 API 误用缺陷并不是开发阶段发生的偶然事件，其具有普遍性。

> 发现 1：API 误用普遍存在于被分析系统中。调研结果显示，平均 17.05% 缺陷修复相关的修改记录与 API 误用相关。

为能够理解 API 误用缺陷的本质原因，本章对随机选取的 830 个缺陷实例进行分析，包括出错的根本原因、误用缺陷的修复模式以及 API 误用重复发生的统计信息。下文将对调研的结果进行仔细分析。

根本原因　在对所有的目标实例分析后，本章对每个实例的误用原因从 C 程序逻辑结构角度进行分类和整理。调研结果显示虽然每个项目中出错的根本原因、相同原因的百分比不同，但是存在三大类常见 C 程序接口误用缺陷模式：

- 不正确的参数使用（Improper Parameter Using, IPU），即缺失或者错误地对单个参数属性、参数之间以及参数和返回值之间的约束检查。
- 不正确的异常处理（Improper Error Handling, IEH），即缺失或者错误地对错误状态代码地检查，以及缺失或者错误地进行异常处理操作。
- 不正确的因果调用关系（Improper Causal Calling, ICC），即缺失或者重复地调用函数对、函数序列，以及忽略因果函数调用之间的上下文关系。

表 2.4　API 误用缺陷实例类别信息

项目	实例	IPU		IEH		ICC		其他	
名称	个数	数量	百分比	数量	百分比	数量	百分比	数量	百分比
Linux	283	43	15.1	96	33.92	77	27.21	67	23.67
OpenSSL	127	21	16.54	42	33.07	49	38.58	15	11.81
FFmpeg	126	18	14.29	43	34.13	52	41.27	13	10.32
Curl	134	23	17.16	38	28.36	57	42.54	16	11.94
FreeRDP	119	22	18.49	30	25.21	48	40.34	19	15.97
Httpd	41	8	19.51	8	19.51	16	39.02	9	21.95
Total	830	135 (16.27%)		257 (30.96%)		299 (36.02%)		139 (16.75%)	

```
"description": "Fixed API nonnull warning."
"date": "2014-11-17 00:00:09"
"author": "Armin Novak"
libfreerdp/core/gateway/http.c
=====================================================
lhs: 100644 | 4cef378c8cafdc4ad4a237f88a092c511c250ffa
rhs: 100644 | 51c5c04dc257975756a4ee3ded16cae493dea95c
@@ -330,10 +330,12 @@ void http_request_free(HttpRequest* http_request)

BOOL http_response_parse_header_status_line(HttpResponse* http_response, char*
     status_line)
{
-  char* separator;
+  char* separator = NULL;
char* status_code;
char* reason_phrase;
-  separator = strchr(status_line, ' ');
+
+  if (status_line)
+     separator = strchr(status_line, ' ');

if (!separator)
   return FALSE;
@@ -433,7 +435,10 @@ BOOL http_response_parse_header(HttpResponse*
     http_response)
*             |    |
*       colon_pos  value
*/
-     colon_pos = strchr(line, ':');
+     if (line)
+        colon_pos = strchr(line, ':');
+     else
+        colon_pos = NULL;

if ((colon_pos == NULL) || (colon_pos == line))
   return FALSE;
```

图 2.3　FreeRDP 中空指针解引用导致的 IPU-API 误用缺陷（sha：9e5be6f7e8）

本章将统计结果总结于表2.4中，在下文中将通过缺陷实例对这些缺陷类型进行详细讲解。

IPU：软件库的开发者提供 API 以实现对特定功能封装，达到软件复用的作用。然而，开发者在使用这些 API 的时候，需要满足特定的约束条件，以保证 API 使用时上下文环境正确。这些 API 被调用前需要满足的条件，亦被称作前置条件（pre-condition）[92]。

例如，一个 API 的参数中包含指针类型，那么通常情况下在调用该 API 之前，需要保证该指针不为空指针 NULL。然而，本章发现 API 的使用者经常忽略这些约束条件，导致空指针解引用错误。图2.3给出一个 FreeRDP 项目中的缺陷修改记录。接口 `strchr(str, c)` 用来搜索字符串 `str` 中，第一次出现字母 `c` 的位置。其中 `str` 是一个指向字符串的指针。在 FreeRDP 项目 http.c 文件中定义的接口 `http_response_parse_header_status_line()` 中，将来自外部输入的参数 `status_line` 作为实参传递给 `strchr`。然而在调用 `strchr` 之前（16 行与 27 行），并没有对该参数进行非空指针检查。如果该参数为空指针，则会产生一个 API 误用，导致程序崩溃。为避免该缺陷，开发者增加参数非空检查的条件

```
"hash": "f7d183f08472e566a2e6b62a80e200a12670ed0e",
"summary":  "libxvid:  Check return value of write() call",
"date": "2016-11-11 10:17:07",
"author": "Diego Biurrun",
libavcodec/libxvid_rc.c
=====================================================
lhs: 100644 | eddbbe8c651407bdc85652191a7ba964c1776f74
rhs: 100644 | 94301a2ac1ffc10a8b22ba491c70013463107452
@@ -87,7 +87,10 @@ av_cold int ff_xvid_rate_control_init(MpegEncContext *s)
(rce->i_tex_bits + rce->p_tex_bits + rce->misc_bits + 7) / 8,
(rce->header_bits + rce->mv_bits + 7) / 8);

-   write(fd, tmp, strlen(tmp));
+   if (strlen(tmp) > write(fd, tmp, strlen(tmp))) {
+      av_log(s, AV_LOG_ERROR, "Cannot write to temporary pass2 file.\n");
+      return AVERROR(EIO);
+   }
}

close(fd);
```

图 2.4　FFmpeg 中参数、返回值关系的 IPU-API 误用缺陷（sha：f7d183f084）

判断，如图中 18-19 行与 28-31 行所示。

此外，接口参数之间、参数与返回值之间的语义关系也需要考虑。最典型的就是 C 标准库中内存操作相关的接口。例如，接口 `memcpy(d, s, n)` 用来从源内存区间 `s` 中，拷贝 `n` 个字符到目标内存区间 `d`。因此，该函数在使用的时候，目标内存区间 `d` 的大小要大于或者等于拷贝长度 `n`。否则将产生一个内存越界错误。另一方面，参数与返回值之间也可能存在语义关联关系。例如，接口 `size_t write(int fd, void* buf, size t cnt)` 从参数 `buf` 指向的内存区域内，读取 `cnt` 个字符到参数 `fd` 所标志的文件或者网络链接（socket）。该接口的返回值记录实际输出字节个数。特别地，`write()` 的返回值为负数表明该函数调用发生错误。因此如果需要确保 `cnt` 个字节被写入到目标地址时，需要显式地检查该值与函数的返回值之间的数值关系。然而如图2.4中 13 行所示，FFmpeg 项目中 libxvid_rc.c 文件在调用该接口后并没有对返回值与参数进行比对，从而忽略 API 参数与返回值之间的约束关系。为避免该错误，开发者增加返回值与参数之间的约束关系检查，如图中 14-17 行所示。

> 发现 2：在所有被研究的 API 误用缺陷实例中，14.29-19.51% 缺陷产生的原因是不正确的参数使用（Improper Parameter Using，IPU），包括缺失或者错误地对单个参数属性、参数之间以及参数和返回值之间的约束检查。

IEH：安全可靠的软件需要对所有可能出错的条件进行捕获，并对捕获的异常进行对应的处理。因此，很多编程语言提供一套完整的异常处理机制，以保证即使

```
"hash": "086ad79970f3cd0463558bfb69122f6acdc9d2da",
"summary": "ldap: check Curl_client_write() return codes",
"date": "2014-12-10 00:41:32",
"author": "Daniel Stenberg",
ib/ldap.c
============================================================
lhs: 100644 | 14437c3b54715cb5ad161d78ee7b89a412b64a9d
rhs: 100644 | 96521bf21862540e32368e229365ed49d6f43cff
---@@ -384,9 +384,17 @@ static CURLcode Curl_ldap(struct connectdata *conn,
    bool *done)
char *dn = ldap_get_dn(server, entryIterator);
int i;

-   Curl_client_write(conn, CLIENTWRITE_BODY, (char *)"DN: ", 4);
-   Curl_client_write(conn, CLIENTWRITE_BODY, (char *)dn, 0);
-   Curl_client_write(conn, CLIENTWRITE_BODY, (char *)"\n", 1);
+   result = Curl_client_write(conn, CLIENTWRITE_BODY, (char *)"DN: ", 4);
+   if(result)
+     goto quit;
+
+   result = Curl_client_write(conn, CLIENTWRITE_BODY, (char *)dn, 0);
+   if(result)
+     goto quit;
+
+   result = Curl_client_write(conn, CLIENTWRITE_BODY, (char *)"\n", 1);
+   if(result)
+     goto quit;
```

图 2.5　Curl 中忘记对错误状态代码检查的 IEH-API 误用缺陷（sha：086ad79970）

底层的功能出错，系统整体也能够正常处理不会崩溃。不幸的是 C 语言并没有提供原生的异常处理机制。所以，在 C 程序中开发者需要自己定义这样的异常处理机制。由于缺少统一的处理方式，异常处理代码经常和项目的领域特定信息相关，形式多样、重复性高、代码繁琐等等。特别地，当函数调用发生错误后，未能正确进行异常处理则会产生异常处理缺陷，甚至会导致严重的安全漏洞，例如：CVE-2014-0092[127]，CVE-2015-0208[128]，CVE-2015-0285[129]，CVE-2017-3318[130]，CVE-2017-5350[131] 等等。事实上，根据开源网络应用安全项目（The Open Web Application Security Project）指出，不正确的异常处理是导致软件安全漏洞的十大因素之一[132]。

为提高系统的鲁棒性，开发者设计并实现各种领域特定、项目相关的异常处理机制。特别地，定义特定的值来表示异常状态代码（error status code），并记录在函数的返回值中[133]。因此调用可能发生异常的目标接口 `f` 后，需要在调用上下文 `Context` 中对 `f` 的返回值进行检查。并根据检查的情况，针对正确和异常发生的错误状态代码分别进行处理，即是否正常执行后续步骤，或者根据错误状态代码进行对应的异常处理。然而调研结果显示，开发者经常会忘记进行这样的异常检测。如图2.5所示，Curl 项目中接口 `Curl_client_write()` 用来将数据写入到回调函数中，并将运行状态保留在 CURLcode 枚举类型返回值中。当 CURLcode 为 0 时，代表程序正确，其他值则为错误。因此，在开发者调用该函数时，需要对返回值检查以保证该函数运行正确。然而，图中 13-15 行的三次调用都忽略对该返

回值的检查。如果 `Curl_client_write()` 执行错误，程序功能将被破坏，甚至导致程序崩溃。为避免该缺陷，开发人员对返回值进行检查，如图中 17、21 以及 25 行所示。同时，如果发现错误则跳转到异常处理模块，如图中 18、22 以及 26 行所示。

为能够区别异常发生的原因，开发者通常使用不同的缺陷代码表示不同的缺陷原因。因此对错误状态代码检测并不是一件容易的事情。特别地，当无法正确处理边界条件时，则会遗漏错误情况，导致系统鲁棒性被破坏，甚至被攻击者利用。如图2.6所示，Curl 项目 ssluse.c 文件中调用 OpenSSL 库接口 `SSL_read()`，从而在 SSL 链接中读取字符串。该接口如果遇到链接关闭、未知的错误发生以及调用上下文的强制终止命令时，返回一个小于或者等于 0 的错误状态代码，从而通知外部环境。然而，图中 13 行中，虽然开发者对返回值进行检查，却忽略 0 也是错误代码之一。为避免该缺陷，开发者修正错误状态代码检测的条件判断，如图中 16 行所示。

为保障系统鲁棒性，开发者需要对所有可能发生的异常进行捕获，并针对每一种异常分别处理。经过调研后，本章发现开发者多基于两种方式对异常进行处理，即错误状态代码传递（error propagation）与异常日志输出。图2.7给出 Curl 项目与 OpenSSL 异常处理的例子。如图所示，两个项目在对错误状态代码进行检查后，分别调用项目特定的错误日志处理打印接口输出日志信息。即在 12-13 行 Curl 通过 `failf()`，在 28 行 OpenSSL 通过 `SSLerr()` 分别进行异常日志输出。同时，Curl 项目在 14 行返回错误代码 `CURLE_OUT_OF_MEMORY` 从而将错误状态向外传递指明错误的原因是内存空间不足。OpenSSL 则是在 29 行，返回错误代码 0。

```
1  "hash": "520833cbe1601feed1c6473bd28c4c894e7ee63e",
2  "summary": "ossl_recv: SSL_read() returning 0 is an error too",
3  "date": "2013-05-22 23:42:33",
4  "author": "Mike Giancola",
5  lib/ssluse.c
6
7  ============================================================
8  lhs: 100644 | 80fa119574f7940861e3ba855574ae59cd5cf833
9  rhs: 100644 | 36c38042ac964669a6e3b8b21e1dc223dd65ab4b
10 @@ -2595,7 +2595,7 @@ static ssize_t ossl_recv(struct connectdata *conn, /*
       connection data */
11
12 buffsize = (buffersize > (size_t)INT_MAX) ? INT_MAX : (int)buffersize;
13
14 nread = (ssize_t)SSL_read(conn->ssl[num].handle, buf, buffsize);
15 - if(nread < 0) {
16 + if(nread <= 0) {
17
18    /* failed SSL_read */
19    int err = SSL_get_error(conn->ssl[num].handle, (int)nread);
```

图 2.6　Curl 中错误状态代码检查错误的 IEH-API 误用缺陷（sha：520833cbe1）

```
"hash": "1f152a42ae9c2985b8a0cedf90d3b63b2e64a898",
"summary": "sspi: print out InitializeSecurityContext() error message",
"date": "2017-04-07 08:49:20",
"author": "Isaac Boukris",
"parent_hash": "aa2e9e90173bb379ccff800c9019d6626b69c452"
lib/vauth/spnego_sspi.c
============================================================
@@ -224,6 +225,8 @@ CURLcode Curl_auth_decode_spnego_message(struct Curl_easy
    *data,
nego->status = s_pSecFn->InitializeSecurityContext(...)
...
if(GSS_ERROR(nego->status)) {
+   failf(data, "InitializeSecurityContext failed: %s",
+         Curl_sspi_strerror(data->easy_conn, nego->status));
return CURLE_OUT_OF_MEMORY;
}


"hash": "884a790e17a22eed42f1fe41ccaebd8c1fe18902",
"summary": "Fix missing NULL checks in key_share processing",
"date": "2016-11-23 22:39:27",
"author": "Matt Caswell",
ssl/t1_lib.c
============================================================
@@ -1538,6 +1538,10 @@ static int add_client_key_share_ext(SSL *s, WPACKET
    *pkt, int *al)
}
skey = ssl_generate_pkey(ckey);
+   if (skey == NULL) {
+      SSLerr(SSL_F_ADD_CLIENT_KEY_SHARE_EXT, ERR_R_MALLOC_FAILURE);
+      return 0;
+   }
...
```

图 2.7　Curl（sha：1f152a42ae）与 OpenSSL（sha：884a790e17）中异常处理代码示例

> 发现 3：在所有被研究的 API 误用缺陷实例中，19.51-34.13% 缺陷产生的原因是不正确的异常处理（Improper Error Handling，IEH），包括缺失或者错误地对错误状态代码进行检查，以及缺失或者错误地进行异常处理操作。

ICC：因果调用关系以及接口调用的时序关系，都可以表示成 a-b 的模式，即接口 `a` 和 `b` 需要协同完成特定功能。最典型的就是资源管理接口，例如锁机制中的 `lock/unlock`，内存管理中的 `malloc/free` 等等。在使用第一个接口 `a` 后，缺失第二个接口 `b` 则会产生资源泄漏（resource leak）缺陷。

例如，在 OpenSSL 项目中 `OPENSSL_malloc()/OPENSLL_free()` 接口封装 C 标准库中 `malloc/free` 的功能，以协同完成内存的管理。当通过 `OPENSSL_malloc()` 申请的内存没有被 `OPENSLL_free()` 释放时，会产生内存泄漏缺陷。图2.8给出一个这种缺陷实例。如图所示，开发者在第 10 行进行内存申请操作，并在第 11 行确认申请成功。然而在后续的一条路径上，并没有释放该内存对象，导致系统资源泄漏。该缺陷在累积后会导致系统资源消耗完，使得系统崩溃。为修复该缺陷，开发者需要保证该内存对象在所有的分支条件中都被正

```
"hash": "0a618df059d93bf7fe9e3ec92e04db8bc1eeff07",
"summary": "Fix a mem leak on an error path in OBJ_NAME_add()",
"date": "2016-05-24 00:09:56",
"author": "Matt Caswell",
crypto/objects/o_names.c
==========================================================
lhs: 100644 | e43fb30a760464a6e1d45bbd9edfeec4ecedb25a
rhs: 100644 | c655a908ddb8560cf76f73570a9f71d39d624d7d
@@ -191,7 +191,7 @@ int OBJ_NAME_add(const char *name, int type, const char
    *data)
onp = OPENSSL_malloc(sizeof(*onp));
if (onp == NULL) {
/* ERROR */
-       return (0);
+       return 0;
}
@@ -216,10 +216,11 @@ int OBJ_NAME_add(const char *name, int type, const char
    *data)
} else {
if (lh_OBJ_NAME_error(names_lh)) {
/* ERROR */
-           return (0);
+           OPENSSL_free(onp);
+           return 0;
}
```

图 2.8　OpenSSL 中忽略内存释放导致的 ICC-API 误用缺陷（sha：0a618df059）

确的释放。本例中，开发者在第 21 行增加内存释放的操作。

因果调用关系最直接的模式是 a-b，然而正确使用该模式不仅仅包含成对出现的函数调用，还需要考虑目标接口 `a/b` 参数、返回值之间的上下文关系。如图2.8中 11 行所示，只有在内存申请成功时，才需要调用第二个函数。此外，如图2.9所示，FreeRDP 项目中通过 `freerdp_keyboard_get_layouts()` 接口

```
"hash": "1845c0b59098565d656c4703adfa40d7bcfb90cf",
"summary": "Fixed possible memory leak.",
"date": "2014-09-15 08:55:00",
"author": "Armin Novak",
client/common/cmdline.c
==========================================================
lhs: 100644 | 1000cb8af5aab2339cbf0266718b8c32ecb85d6a
rhs: 100644 | 104f910058603ccdef5f8b4f0a5254d5bb182ba7
@@ -1124,12 +1124,14 @@ int freerdp_client_settings_*_print(rdpSettings*
    settings, int

layouts = freerdp_keyboard_get_layouts(RDP_KEYBOARD_LAYOUT_TYPE_STANDARD);
WLog_INFO(TAG, "Keyboard Layouts");
for (i = 0; layouts[i].code; i++)
   WLog_INFO(TAG, "0x%08X\t%s", (int) layouts[i].code, layouts[i].name);
+free(layouts);
layouts = freerdp_keyboard_get_layouts(RDP_KEYBOARD_LAYOUT_TYPE_VARIANT);
WLog_INFO(TAG, "Keyboard Layout Variants");
for (i = 0; layouts[i].code; i++)
   WLog_INFO(TAG, "0x%08X\t%s", (int) layouts[i].code, layouts[i].name);
+free(layouts);
layouts = freerdp_keyboard_get_layouts(RDP_KEYBOARD_LAYOUT_TYPE_IME);
WLog_INFO(TAG, "Keyboard Input Method Editors (IMEs)");
...
free(layouts);
```

图 2.9　FreeRDP 中忽略调用关系中参数和返回值的语义关系导致的 ICC-API 误用缺陷（sha：1845c0b590）

```
1  "hash": "d285b5418ee1ff361f06545e0489ece61bdd1a50",
2  "summary":  "Avoid a double-free in crl2pl7",
3  "date": "2016-06-14 11:27:10",
4  "author": "Matt Caswell",
5  apps/crl2p7.c
6  ============================================================
7  lhs: 100644 | 1631258793ec96ae5d653be3b6ebb7bfa7f3c26d
8  rhs: 100644 | 9c5f79f9f37988d0db709eda6eeaf82d44a1044a
9  @@ -84,10 +84,8 @@ int crl2pkcs7_main(int argc, char **argv)
10
11
12 if ((certflst == NULL) && (certflst = sk_OPENSSL_STRING_new_null()) == NULL)
13    goto end;
14 -if (!sk_OPENSSL_STRING_push(certflst, opt_arg())) {
15 -  sk_OPENSSL_STRING_free(certflst);
16 +if (!sk_OPENSSL_STRING_push(certflst, opt_arg()))
17    goto end;
18 -}
19
20 end:
21    sk_OPENSSL_STRING_free(certflst);
```

图 2.10　OpenSSL 中重复释放内存对象导致的 ICC-API 误用缺陷（sha：d285b5418e）

来申请内存空间，以适配键盘展示格式信息。在内存对象生命周期后，需要通过调用 `free()` 函数对内存进行释放。虽然图中代码片段在 24 行进行内存空间的释放，然而该段代码依旧存在内存泄漏缺陷。如图所示，开发者在第 11、18 以及 21 行分别申请内存空间，并依次赋值给 `layouts` 变量。因此，24 行的 `free()` 函数只释放 21 行的内存申请操作，导致 11 行和 18 行的内存对象泄漏。所以，因果调用关系不仅仅需要考虑 a-b 模式，还需要考虑参数之间、参数与返回值之间的值约束关系。

另一方面，如果在因果调用关系 a-b 模式中，第二个接口 `b` 出现次数比 `a` 多，则会产生重复释放缺陷（double free）。如图2.10所示，开发者在 12 行通过接口 `sk_OPENSSL_STRING_new_null()` 申请内存空间并存储在变量 `certflst` 中。在对接口 `sk_OPENSSL_STRING_push()` 的运行状态检查后，在错误的路径上对 `certflst` 指向的内存进行释放操作。在 15 行通过函数 `sk_OPENSSL_STRING_free()` 后，跳转到 20 行。然而，在 20 行之后的代码中，存在第二个释放函数，这将产生重复释放缺陷。该缺陷会使得程序的行为不可预判（undefined behavior），即程序的行为将完全不可控制。事实上，该类缺陷往往会使得操作系统的内存管理机制失效。

> 发现 4：在所有被研究的 API 误用缺陷实例中，27.21-42.54% 缺陷产生的原因是不正确的因果调用关系（Improper Causal Calling，ICC），包括缺失或者重复地调用函数对、函数序列，以及忽略因果函数调用之间的上下文约束。

其他类型：在所有的被分析实例中，139 个缺陷实例产生的原因与项目特定的语义或者需求相关，难以直接应用于普适性的结果中。例如，有些缺陷产生的本质原因是不恰当的接口设计，所以开发者通过重构接口的定义来修复这些缺陷（Curl-82232bbbaf、FreeRDP-1bcale7820 等等）。再例如，为提供更好的异常处理机制，开发者对日志记录接口（Linux-5b60fc0980）、异常处理中输出的日志信息（OpenSSL-0cb8c9d85e）等等进行修改。此外，还有一些修复用来处理其他接口相关错误，包括：修复笔误（typo）造成的缺陷（Curl-27ac643455）、移除过期函数（FFmpeg-2dafbae994）、代码优化以避免编译警告（Curl-4dae049157）等等。

修复模式　对于所有被分析的 API 误用缺陷实例，本章对修复报告进行详细分析，以总结开发者如何进行修复以及修复的难度。下文将对这部分内容进行总结。

修复策略：如上文图中各例子所示，缺陷修复的模式与缺陷发生的根本原因有直接关系。即：

- 针对 IPU 缺陷模式，需要在 API 调用之前，增加相应参数约束的检查。如果参数与返回值存在语义关系，那么同时还需要对参数和返回值之间的关系进行检查。
- 针对 IEH 缺陷模式，需要在 API 调用之后，显式地对错误状态代码进行检查。特别地，该检查需要考虑所有预定义、领域相关的错误代码。同时，在异常处理的路径上，正确地对异常进行处理。通过调研，本章发现两种常见的异常处理方式：输出错误日志与将错误状态向上传递。
- 针对 ICC 缺陷模式，需要考虑具有因果关系的接口调用对。同时，考虑这些接口对之间的值关联关系，即个数一致、参数一致、返回值与参数一致等等。特别地，接口对之间可能存在因果关系，即需要考虑第一个函数的成功与否，如果成功才需要调用第二个函数。另外，在成功调用第一个函数后，每一条后续路径中都需要对第二个函数进行调用，且不可以重复调用。

修复复杂度：对于所有被分析的缺陷修复报告，本章在差异性报告中统计缺陷修复的行数，从而定量地评估修复的复杂度。本章将统计的信息汇总于表2.5中。如表中结果所示，约 79.40% 的修复在 5 行及以内完成，近 96.15% 的缺陷能够在 10 行以内完成修复，最大修复行数也只有 21 行。

然而，修改行数不能完全代表修复的复杂度。对所有的修复报告总结后，本章发现修复的复杂度与项目领域特定语义、接口行为自身、使用的上下文等多个因素都有密切关系。例如，对于 IPU 错误，最简单的修复策略就是在 API 调用之前显式地添加参数检查。但是，实际项目中很多检查不仅仅是空指针或者常数的整数比较，而是与项目特定需求相关。如图2.11所示，Curl

表 2.5　API 误用缺陷实例修复行数统计信息

项目名称	缺陷数目	缺陷修复相关的行数①					
		1-5	6-10	10+	平均	中位数	最大值
Linux	283	225	47	11	3.88	4	14
OpenSSL	127	108	17	2	3.07	2	13
FFmpeg	126	95	26	5	4.01	3	21
Curl	134	98	26	10	4.65	3	21
FreeRDP	119	103	14	2	2.63	2	11
Httpd	41	30	9	2	4.04	3	11
总结	830	659 (79.40%)	139 (16.75%)	32 (3.85%)	3.73	3	21

① 修复行数为 +/-开始的行数的总和。其中，如果一个修改记录包含多个 API 实例，我们计算平均值。

项目中接口 `SecCertificateCreateWithData()` 即使在遇到不合法或者错误的实参数据传递时，依然不返回空指针。因此，开发者需要显式地调用接口 `SecCertificateCopyPublicKey()`，完成其有效性检查，确保函数 `CFArrayAppendValue()` 参数的正确性。虽然该修复只有 8 行，然而如图2.11中 15-22 行所示，该修复设计复杂的程序语义信息。

此外，针对于图2.9中的缺陷，虽然修复只包含两行代码（即添加内存释放操作于 15 行以及 20 行），但是该修复涉及到数据流关系和上下文语义关系。特别地，如果内存申请后有复杂的路径分支情况，开发者需要保证在每条路径上都正

```
1  "hash": "0426670f0a8ffa69df64a3babfb5caed522feb7f",
2  "summary": "Check CA certificate in curl_darwinssl.c.",
3  "date": "2014-09-01 00:34:37",
4  "author": "Vilmos Nebehaj",
5  lib/vtls/curl_darwinssl.c
6  ======================================================
7  lhs: 100644 | 9ba287d0e91ee470fad880a5aa7d7981a9cfdeb1
8  rhs: 100644 | 3726357472fc3f5d01e6a5959a20c121ba6a0966
9  @@ -1671,6 +1671,16 @@ static int append_cert_to_array(struct SessionHandle
       *data,
10    SecCertificateRef cacert =
11    SecCertificateCreateWithData(kCFAllocatorDefault, certdata);
12    ...
13
14 +  /* Check if cacert is valid. */
15 +  SecKeyRef key;
16 +  OSStatus ret = SecCertificateCopyPublicKey(cacert, &key);
17 +  if(ret != noErr) {
18 +    CFRelease(cacert);
19 +    failf(data, "SSL: invalid CA certificate");
20 +    return CURLE_SSL_CACERT;
21 +  }
22 +  CFRelease(key);
23 +
24 CFArrayAppendValue(array, cacert);
25 CFRelease(cacert);
```

图 2.11　Curl 中对参数添加项目相关的参数检查（sha：0426670f0a）

确地操作内存对象。否则就会导致如图2.9中的内存泄漏或者图2.10中重复释放错误。其中，图2.10的错误是由于开发者在前一个代码修改中，为修复部分可达路径上内存泄漏缺陷（sha：1c4221）所引入的重复释放错误。

> 发现 5：在所有被研究的 API 误用缺陷实例中，大部分缺陷（96.15%）的修复代码行数在 10 行以内。然而，缺陷修复语义复杂，需要深入的理解缺陷发生的上下文信息。

误用重复性　在对 API 误用实例的调研过程中，本章发现即使存在完善的 API 使用手册，一个 API 仍然可能被多次误用。例如，OpenSSL 中接口 `BN_CTX_get()` 用来获取临时结果。如果该过程发生错误，则返回一个空指针 NULL。因此开发者

```
"hash": "5625567f9c7daaa2e2689647e10e4c5d7370718f",
"summary": "Fix another possible crash in rsa_ossl_mod_exp.",
"date": "2017-06-14 09:35:48",
"author": "Bernd Edlinger",
crypto/rsa/rsa_ossl.c
============================================================
lhs: 100644 | 5e0ad92cb1c3c4257f1fd21da0bfad4107d399e5
rhs: 100644 | 92c4be1868a5b1608c1ab7841281014c1a51a692

正确用法片段1
// Line-96
ret = BN_CTX_get(ctx);
num = BN_num_bytes(rsa->n);
buf = OPENSSL_malloc(num);
if (f == NULL || ret == NULL || buf == NULL) {
   RSAerr(RSA_F_RSA_OSSL_PUBLIC_ENCRYPT, ERR_R_MALLOC_FAILURE);
   goto err;
}

正确用法片段2
// Line-256
f = BN_CTX_get(ctx);
ret = BN_CTX_get(ctx);
num = BN_num_bytes(rsa->n);
buf = OPENSSL_malloc(num);
if (f == NULL || ret == NULL || buf == NULL) {
   RSAerr(RSA_F_RSA_OSSL_PRIVATE_ENCRYPT, ERR_R_MALLOC_FAILURE);
   goto err;
}

错误用法以及修复
@@ -608,6 +608,8 @@ static int rsa_ossl_mod_exp(BIGNUM *r0, const BIGNUM *I,
     RSA
vrfy = BN_CTX_get(ctx);
+   if (vrfy == NULL)
+       goto err;

{
BIGNUM *p = BN_new(), *q = BN_new();

相同的错误
7928e-crypto\dh\dh_key.c(2017-01-24, Bernd Edlinger)
1ff74-crypto\dsa\dsa_gen.c(2016-09-21, Matt Caswell)
```

图 2.12　OpenSSL 项目中接口 BN_CTX_get() 被多次误用

需要显式地对该返回值进行检测。然而，在 OpenSSL 的修复实例中，本章发现两个作者，三次误用该接口。特别地，在这些错误代码的同一个文件中，已经存在该 API 的正确用法，如图2.12所示。

在所有的误用实例中，本章共发现 55 个不同的 API 被误用多次。其中 Linux 内核 10 个，OpenSSL10 个，FFmpeg14 个，Curl14 个，FreeRDP5 个以及 Httpd2 个。这些被误用的 API 共发生 178 次，占所有实例的 21.45%。特别地，C 语言标准库接口 `calloc()` 在 FreeRDP 项目中共出现 21 次误用，占该项目的 15.67%。此外，误用第三方库 API 在应用软件中普遍存在。其中，Curl 误用 13 次 OpenSSL 中的 API，FreeRDP 误用 6 次，Httpd 误用 3 次。例如，在 Curl 项目中，修复（sha：0b5efa57ad）通过添加对返回值的检查，来处理被误用的 OpenSSL 接口 `SSL_CTX_load_verify_locations()`。

2.3.3　讨论

由于现有研究中并没有针对 C 程序接口误用缺陷的调研工作或者分类研究，因此本章通过对接口缺陷实例分析以完成对误用模式的总结。这些误用模式能够有效地帮助研究人员和开发者理解接口使用中需要关注的约束条件。下文中，本章将对该调研方法中可能的不足进行讨论。

一方面，本章的缺陷实例在六个开源项目和给定时间段的修改记录中提取。因此，这些缺陷并没有涵盖所有类型的项目和时间，存在完备性不足的可能。此外，本章从修改记录中提取关键词，并基于关键词匹配策略抽取目标修改实例。然而，开发者可能不会在修改记录中撰写相关的关键词。尽管如此，本章所选的六个项目来自不同的领域并且被广泛关注，具有大量的开发人员和良好的日志记录。本章在跨越五年的历史记录中，共分析 830 个缺陷实例。针对于关键词，本章在 600 个实例上进行预实验，以总结相关的关键词。此外，本章参考已有的缺陷分类和针对 API 误用的研究工作，从而帮助对缺陷模式总结。因此，遗漏一大类普遍存在的 API 误用缺陷模式的概率比较低，结果具有较好的典型性和通用性。

另一方面，本章对每个缺陷的分析可能存在主观偏差性。对于每个缺陷实例，本章仔细地研究缺陷描述文件、差异性报告和源代码，并与开发者进行讨论，以理解缺陷的详细信息。并在分类过程中，参考程序逻辑结构。针对于所有的调研结果，本章与至少两个实际开发人员或者软件工程相关从业人员进行核对。因此，本章认为所有的缺陷实例为接口误用实例，并且具有研究价值。为复现该调研工作的结果，本章将调研中的原始数据发布到 Github[①]，供研究人员和开发人员参考。

① https://github.com/imchecker/compsac19

2.4　规约描述语言 IMSpec

如上所述，现代软件多基于程序接口开发。在使用 API 时，需要满足接口使用约束以保证其功能能够正确执行。软件库在提供 API 时，通常会提供基于自然语言描述的使用说明。然而，这些说明由于存在歧义、内容复杂、缺少实例等原因，经常被使用者忽略，导致 API 被误用[106]。形式化规约描述语言能够有效地描述 API 使用时的约束条件，尤其当描述语言基于程序语言的语法结构设计时，能够快速地被使用者接受。

同时，传统的静态分析工具将缺陷模型或需要满足的程序属性编码在分析引擎中。这种策略能够有效地检测预先定义过的接口，比如 C 语言标准库中的接口。然而，对于用户自定义的 API 则支持不足。特别地，很多用户定义的 API 与 C 标准库中的 API 存在一致的使用约束。另一方面，很多项目特定的 API 仅在项目中定义和使用。这些 API 使用的次数有限，有些甚至少于 5 次。这种情况使得基于数据挖掘的技术难以应用。数据挖掘技术基于统计信息来推断正确和错误，算法可行性的基础是存在大量可靠的数据对模型进行训练。然而，有限次的接口使用会导致挖掘技术产生大量的误报和漏报。即学习过程中由于出现次数少，达不到学习模型的最小置信度（support），而忽略这些 API 的使用。所以，基于偏离多数情况的使用实例是错误的缺陷检测方法，在实际项目中存在天然缺陷。因此，这类接口的使用约束只能够通过显式的方式进行描述。

为能够帮助使用者理解接口使用约束，同时支持对用户自定义 API 以及用客户端代码中使用的第三方 API 的误用缺陷检测，本章基于2.3节中总结的 API 误用缺陷模式，设计面向 C 程序接口使用约束的领域特定语言 IMSpec。IMSpec 旨在通过轻量级、类程序语言结构的方式，描述 API 的使用约束条件，以帮助接口开发者和使用者明确使用约束条件、供检测工具对错误使用 API 导致的缺陷进行查找。

2.4.1　设计动机

本文希望通过提供一个面向 C 程序接口使用领域特定需求的轻量级、精确的方式，以支持大多数通用接口使用约束条件。通过该语言对 API 使用约束的描述，从而有效地缩减设计者和使用者间的认知差距。一方面，使得客户端开发者能够快速、准确地使用这些 API；另一方面，能够使研究人员和测试人员有效地对现有的代码进行缺陷检测。因此，基于缺陷调研结果与实际开发者讨论后，本章基于以下原则进行 IMSpec 的设计：

1. 白名单策略：对于一个给定的 API，其误用存在很多种不同的形式，难以枚举。相反，其正确的使用方式有限，能够在有限的资源下做到精确描述。所

以，本章决定通过白名单的方式进行语言设计，即描述正确使用需要满足的约束，并且默认开发者在使用的时候能够显式地对所有的约束进行保证。在检测的过程中，如果遇到约束中的某一条或者几条没有满足，则认为存在一个接口误用。

2. 数据流分析：数据流属性在对接口使用的语义分析中具有重要意义。特别地，分析中需要区分 API 在使用时的上下文，以及与 API 相关的程序对象。接口误用缺陷中，很多模式都涉及到数据流关系。其产生的重要原因是开发者对程度对象操作不一致。例如，一个指向堆内存的指针，在没有进行有效释放就被重新赋值，从而导致内存泄漏缺陷；内存释放函数与内存申请函数个数相对，但是内存对象不相对；没有在所有的路径上都进行有效的内存管理，导致某些异常处理路径中存在内存泄漏等等。
3. API 领域特定元素：为提供一个语法简单、语义丰富的接口使用描述方式，需要重点关注 API 使用的特点，即在保证描述能力的前提下尽可能地精简语法结构。同时，需要考虑接口使用的语义信息。特别地，一些语义信息并不能通过 C 程序的语法进行表示。例如：释放一个指向栈内存的指针，将产生一个释放非堆内存的错误（CWE-590：free memory not on heap）。
4. 工具无关的语义：接口使用约束描述语言的一个重要应用是作为缺陷检测工具的输入。因此，该语言需要有独立于特定工具实现的语义信息，以方便研究人员利用该语言设计不同的缺陷检测工具。同时，可以被研究人员在多个应用场景中使用。例如：用于文档中对接口使用约束进行定义，以解决自然语言存在的不足；用于测试用例生成，供动态检测工具分析；用于代码插桩技术，将规约转化为 C 代码中对应的语句，在运行时进行监控等等。

2.4.2　语法与语义

基于误用缺陷模式的总结以及和开发者讨论的结果，本章提出面向 C 程序接口使用约束的领域特定语言 IMSpec。下文中将对 IMSpec 的语法和形式语义进行定义。

IMSpec 语法元素　本章用 $\mathbb{N}$ 与 $\mathbb{Z}$ 来表示非负整数以及全体自然数，$\mathbb{ID}$ 用来表示所有的标识符。IMSpec 语言的语法结构如图2.13所示，其中 $i \in \mathbb{N}, n \in \mathbb{Z}, id \in \mathbb{ID}$。加粗的字体为保留关键词，例如：**`Spec`** 和 **`Target`** 等。下文中首先概括性介绍每一个元素的含义，接着将通过一个代码案例以及对应的 IMSpec 描述帮助读者理解 IMSpec 语言的细节。

IMSpec 规约集（*Specs*）由独立的 IMSpec 描述实例（*Spec*）组成。一个描述

$$
\begin{aligned}
\mathit{Specs} &::= \mathit{Spec}^{*} \\
\mathit{Spec} &::= \texttt{Spec:}\quad \mathit{Target\ Fib\ Pre\ Post} \\
\mathit{Target} &::= \texttt{Target:}\quad \mathit{FunSig} \\
\mathit{Fib} &::= \texttt{Fib} \\
\mathit{Pre} &::= \texttt{Pre:}\quad \mathit{Cond}^{+} \\
\mathit{Post} &::= \texttt{Post:}\quad (\mathit{Cond}, \mathit{Action}^{*})^{+} \\
\mathit{Cond} &::= \texttt{true} \mid \mathit{Opd\ CmpOp\ Opd} \mid \mathit{Opd\ MemberOp}\ (\mathit{Set}) \\
\mathit{Action} &::= \textbf{return} \mid \mathit{Call} \\
\textbf{return} &::= \texttt{RETURN}\ (\mathit{FunName} : n) \mid \texttt{RETURN}\ (\mathit{FunName} : \texttt{NULL}) \\
\mathit{Call} &::= \texttt{Call}\ (\mathit{FunName} : \mathit{Cond}^{*}) \\
\mathit{Opd} &::= \mathit{Arg} \mid \mathit{UnOp}\ (\mathit{Arg}) \mid \texttt{NULL} \mid n \\
\mathit{FunSig} &::= \mathit{FunName}\ (\mathit{Type}^{*})\ \texttt{->}\ \mathit{Type} \\
\mathit{Arg} &::= \mathit{FunName\_arg\_}i \\
\mathit{UnOp} &::= \texttt{LEN} \mid \texttt{TYPE} \mid \texttt{MEMTYPE} \\
\mathit{CmpOp} &::= \texttt{!=} \mid \texttt{==} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{<} \\
\mathit{MemberOp} &::= \texttt{IN} \mid \texttt{NOTIN} \\
\mathit{Set} &::= id^{+} \mid [n_1 : n_2]^{+} \\
\mathit{FunName} &::= id \\
\mathit{Type} &::= id
\end{aligned}
$$

图 2.13　IMSpec 语法结构

实例旨在对一个目标 API 的使用约束进行描述。一个 IMSpec 的实例主要包含四大部分，

1. *Target*：用来标记目标 API，即这个规约描述的对象。
2. *Fib*：用来定义危险函数，即不推荐使用的函数。包括容易出错的 API 或者已经被弃用的 API。
3. *Pre*：定义目标接口的前置条件，即在调用 API 之前需要满足的约束条件。
4. *Post*：定义目标的后置条件，即在调用 API 之后需要满足的约束条件。目前，IMSpec 主要关注异常处理、因果调用关系两种和 API 使用相关的后置约束条件。

其他元素的语法，概括如下：

- *Cond*：用来描述布尔类型的表达式，可以当作一个谓词关系。用在 *Pre* 中时，用来描述调用目标 API 前需要满足的约束；用在 *Post* 中时，用来描述如果当前约束满足，则需要执行后续的动作；用在 *Call* 中时，用来描述调用关系中需要满足的数据流关系约束。
- *Action*：用来描述在 API 执行之后，当特定的约束条件满足时，需要执行相

应的动作。比如异常处理或者处理因果函数调用关系。

- *Return*：*Action* 的具体表现动作之一，用来描述需要返回特定的值。
- *Call*：*Action* 的具体表现动作之一，用来描述需要调用特定的接口。
- *Opd*：是表达式 *Cond* 中的操作数（operand），包括函数参数 *Arg*、携带内置 *UnOp* 操作符的函数参数、空指针 **NULL** 以及自然数 $\mathbb{Z}$。
- *FunSig*：用来表示接口的声明。
- *Arg*：用来描述接口的参数，其形式为接口名、关键词“arg”以及索引位置 i 的组合。特别地，令 i 为 0 时表示接口的返回值。
- *UnOp*：用来表示内置的操作符，以提供针对接口的语义操作，其中 **LEN** 用来表示指针指向的内存空间的大小、**TYPE** 用来表示参数的类型、**MEMTYPE** 用来表示内存类型。
- *CmpOp*：是表达式 *Cond* 中的运算符（operator），目前 IMSpec 关注关系运算符（relational operators）。
- *MemberOp*：是表达式 *Cond* 中针对集合的运算符，目前 IMSpec 关注于属于和不属于两种。
- *Set*：用来表示集合，目前 IMSpec 关注字符串以及整数的常量集合。
- *FunName*：用来表示函数名。
- *Type*：用来表示参数类型。

IMSpec 案例讲解　为帮助读者更好地理解 IMSpec 的语法结构，作者将通过图2.14中的例子以及图2.15中针对该例子中出错接口的规约描述实例，对 IMSpec 进行详细介绍。例子中共有针对两个 API（`fopen()`, `fgets()`）的四处误用错误：（1）第 10 行中，缺失必要的参数检查；（2）第 15 行中，错误的异常处理；（3）第 20 行中，错误的错误状态代码检测以及（4）第 31 行中，资源泄漏错误。IMSpec 描述中对 `fopen`, `fgets` 的使用约束进行定义。

IMSpec 实例需要明确指出该描述实例的目标对象 *Target*。例如，图2.14中规约描述的对象分别是第 3 行的 `fopen()` 以及第 15 行的 `fgets()`。对于每一个 IMSpec 实例，目标接口的使用约束由三部分组成：不可以使用的特殊标识符 *Fib*，针对于参数使用的前置条件 *Pre*，以及针对于异常处理和因果调用关系的后置条件 *Post*。其中前置条件 *Pre* 与后置条件 *Post* 的组合能够有效描述上文中总结出普适性的三种接口使用约束模式。特别地，本章使用占位符“_”来表示约束中不关心的部分（实现中，这部分将被处理为恒成立）。通俗地讲，可以认为“_”用来表示所有为真的条件。

Fib 标识符用来处理一种特殊的接口使用约束，即被标识的 API 对象不应该

```
1     #define SUCCESS 1
2     #define FILEERR -1
3     #define IOERR -2
4
5     int foo(char *fileName){
6        char buffer[100] = "";
7        FILE *pFile;
8
9     // 1. 缺少参数检查，导致空指针解引用错误
10 +     if(fileName == NULL) return ERROR;
11       pFile = fopen(fileName, "r");
12       if (pFile == NULL){
13
14    // 2.1 错误的异常处理，返回不正确的错误状态代码
15 -        return IOERR
16 +        return FILEERR;
17       }
18
19    // 2.2 错误的错误状态代码检测
20 -     if (fgets(buffer, 100, pFile) < 0){
21 +     if (fgets(buffer, 100, pFile) == NULL){
22          goto err;
23       }
24       ...
25       fclose(pFile);
26       return SUCCESS;
27       err:
28          Log("Error read file");
29
30    // 3. 缺少内存释放操作，导致内存泄漏
31 +     fclose(pFile);
32       return IOERR;
33    }
```

图 2.14　具有多种缺陷模式的接口缺陷样例

```
1  // 接口fopen的IMSpec实例，打开第一个参数指向的文件，如果失败则返回空指针
2  Spec:
3  Target: fopen(char*, char*) -> FILE*
4  Pre:
5  - fopen_arg_1 != NULL,
6  - fopen_arg_2 IN (r, w, a, r+, w+, a+)
7  Post:
8  // 异常检测以及异常处理操作
9  - fopen_arg_0 == NULL, RETURN(foo:FILEERR);
10 // 正确执行，需要关闭文件
11 - fopen_arg_0 != NULL, CALL(fclose:  fopen_arg_0 == fclose_arg_1)
12
13 // 接口fgets的IMSpec实例，从第三个参数中读取第二个参数个字符到第一个参数中，
14 // 如果出错返回空指针
15 Spec:
16 Target: fgets(char*, int, FILE*) -> char*
17 Pre: // omit single parameter validation
18 - LEN(fgets_arg_1) >= fgets_arg_2
19 Post:
20 // 异常检测以及异常处理操作
21 - fgets_arg_0 == NULL, CALL(log:  true), RETURN(_:IOERR)
```

图 2.15　图 2.14中目标接口的 IMSpec 实例

出现在整个项目中。随着软件库的更新，过时 API 被新的 API 替换，以提供更稳定、安全的功能。另一方面，在 C 的标准库或者软件库中有很多的危险函数（容易被误用），调用这些函数会极大地损失系统的可靠性，引入软件缺陷（CWE676: Use of Potentially Dangerous Function）。例如下边这段代码中，

```
void manipulate_string(char * string){
   char buf[24];
   strcpy(buf, string);
   ...
}
```

程序员在没有确保字符指针 `string` 指向的数据大小是否适合本地缓冲区 `buf` 时，使用接口 `strcpy()` 盲目地复制数据。如果 `string` 可以由攻击者来输入或者字符串参数的内容能够被影响，则可能导致缓冲区溢出。现实程序中，多存在用户输入的情况。针对于以上两种情况，一个可行的解决方案就是提供禁止使用的 API 列表。在代码中搜索这些 API 的位置，提示开发者从而使用更安全的替代方案。需要说明的是，*Fib* 不与其他元素一同使用。因此该标志的出现时，代表该目标接口不应该出现在程序，如果出现即程序不正确。

Pre 前置条件用来描述接口使用前需要满足的约束。目前，IMSpec 关注参数值相关的约束条件。一个前置条件 *Pre* 由一组布尔类型的表达式 *Cond* 组成，即每一个 *Cond* 都是一个独立的条件。当这些都满足的时候，才能够确保 API 满足被调用的上下文环境，能够正确完成其内部封装的功能。因此前置条件 *Pre* 中的每个 *Cond* 都必须要为真，才算作正确的接口使用。

在表达式 *Cond* 中，IMSpec 通过 *Arg* 来表示 API 的参数以及返回值。在 *Arg* 中，IMSpec 通过索引位“i”来表示第几个参数。例如，

```
fopen_arg_1 != NULL
```

表示接口 `fopen()` 的第一个参数不为 NULL 的表达式。特别地，令 *Arg* 的索引位置为零时（即形如 *arg*_0）表示该变量为目标接口的返回值。例如，

```
fopen_arg_0 == NULL
```

则表示接口 `fopen()` 的返回值为 NULL 的表达式。因此根据图2.15中第 5 行的描述可知，图2.14中第 11 行对接口 `fopen()` 存在误用，即没有保证第一个参数不为空。

为支持数值关系相关的约束以及 C 语法无法显式表示的语义约束，IMSpec 提供三个内置的运算符 *UnOp*：

- **LEN** 用来表示指针所指向的内存对象的长度。需要说明的是，该运算符与指针具体的类型相关。例如下列代码中，

```
int* ptr = (int*) malloc(sizeof(int)*100);
char* ptr_c = (char*) ptr;
int buffer[100];
```

`LEN(ptr)` 的值为 100，而 `LEN(ptr_c)` 的值为 400 或者 800，具体的值取

决于系统中 int 类型的长度。再比如下面约束，

```
LEN(fgets_arg_1) >= fgets_arg_2
```

则表示 `fgets()` 函数的第一个参数指向的内存区域大小要大于或者等于第二个参数。

- **`TYPE`** 用来表示被作用的操作数的具体类型。该类型信息则在变量声明时就确定。例如，上述例子中 `TYPE(ptr)` 的结果为指向整数类型的指针（int*），而 `TYPE(ptr_c)` 的结果为 char*。
- **`MEMTYPE`** 用来表示指针所指向的内存对象的区域属性，即堆（heap）还是栈（stack）。例如，上述例子中 `MEMTYPE(ptr)` 的结果为 heap，而 `MEMTYPE(buffer)` 的结果 stack。需要说明的是，C 语言语法本身无法表示内存区域的属性。

此外，在实际项目中，某些参数的取值范围需要在特定的值域范围内。例如，一个文件的打开方式只能够是特定模式 $(r, w, a, r+, w+, a+)$ 中的一种。为能够支持这种约束条件，IMSpec 提供集合运算符 *MemberOp*。其中 **`IN`** 表示操作数需要在一个集合当中；**`NOTIN`** 则表示操作数不可以是集合中的元素。目前，IMSpec 的集合元素用于字符串集合以及整数集合。其中，整数集合利用区间 $[n_1 : n_2]$ 的描述方式，即从 n_1 到 n_2 的所有整数。因此，图2.15中第六行的约束

```
fopen_arg_2 IN (r, w, a, r+, w+, a+)
```

的含义是，接口 `fopen()` 的第二个参数需要是这个集合中的某个元素。

Post 后置条件用于对接口异常处理以及因果调用相关的使用约束进行描述。如上文中所述，C 语言没有内置的异常处理机制。因此，开发者多在返回值中携带错误状态代码。所以，为描述接口异常检测相关的使用约束，IMSpec 需要包含两部分：错误代码检测的 *Cond* 表达式，以及表达式成立时后续的异常处理动作 *Action*。基于对实际项目中接口误用缺陷模式的总结，IMSpec 目前支持两种异常处理动作：错误代码向上传递的 *Return* 动作，以及调用特定接口的 *Call* 动作。*Return* 旨在描述当目标 API 执行中发生错误时，需要通知外界调用者。所以，返回一个错误代码 *n*。特别地，针对于不同的上下文，返回值可能不同。IMSpec 通过 *Return* 中的函数名进行区别。同时，如上文所示当函数名为“_”时，则表示对所有的上下文适用。*Call* 旨在描述当目标 API 执行错误时，需要调用其他的接口来协同处理异常。例如资源的关闭、调用日志相关接口输出运行错误信息等。如图2.15中第 9、21 行所示，

```
fopen_arg_0 == NULL, RETURN(foo:FILEERR);
fgets_arg_0 == NULL, CALL(log: true), RETURN(_:IOERR)
```

开发者需要对 `fopen()` 和 `fgets()` 的返回值进行判断。当 `fopen()` 出错时，会返回一个 NULL 指针。因此，该约束表明当发生错误（即返回值为 NULL），开发者需要在 `foo()` 上下文中返回一个错误代码 **FILEERR**。而对于 `fgets()`，发生错误后开发者需要进行两个异常处理：（1）调用异常输出接口 `log()`；（2）返回默认的错误代码 **IOERR**。由于，开发者通常会对异常代码进行宏定义，所以 IMSpec 在解析的过程中，可以提供一个宏定义文件进行替换。

另一方面，*Post* 可以用于对因果调用相关约束进行描述。因果调用关系可以简单描述为 `a-[cond]-b` 形式，即 `a` 被成功执行（`cond` 条件满足），那么就调用 `b` 接口。IMSpec 通过将接口 `a` 作为 *Target*，并将 `b` 在 *Post* 中描述的方式定义这种约束关系。例如，接口 `fopen()` 与 `fclose()` 协同对文件的打开和关闭进行管理。同时，当文件被成功打开时，即 `fopen()` 返回值不为空时，才需要调用 `fclose()` 关闭文件。则 IMSpec 的描述如图2.15中 11 行所示，

```
fopen_arg_0 != NULL, CALL(flose: fopen_arg_0 == fclose_arg_1)
```

即接口使用者需要对 `fopen()` 的返回值进行检查。如果不为 NULL，则需要调用关闭接口 `fclose()`。特别地，需要考虑两者间的数据流关系，即打开和关闭需要针对于同一个文件句柄，即描述中 CALL 后面的 *Cond* 表达式。否则，不一致的资源管理会导致资源泄漏或者重复释放缺陷。

IMSpec 形式语义　静态单侧赋值（SSA）[134]，是程序中间表达（intermediate representation，IR）的特性，即每个变量仅被赋值一次。后文中，本章将基于具有 SSA 的程序形式对 IMSpec 的形式语义进行介绍。

定义 2.1 (程序状态)：　一个程序状态 $s = (l_s, l_e, inst, \Omega)$ 是一个四元组。其中，*inst* 代表程序执行的具体语句（instruction），l_s 和 l_e 分别代表 *inst* 语句执行前和执行后的两个位置（program location），Ω 为语句执行后内存中所有变量的值信息。

直观地，l_s 和 l_e 是程序在执行 *inst* 语句前和执行后到达的位置。本章对三种程序语句进行抽象表示，包括：（1）函数调用语句，即将对接口 `fName()` 的调用表示为 `FCall_fName`；（2）返回语句，即在函数中返回 v 表示为 `Return_v`；（3）条件判断语句（if-condition），即在程序中对值的判断表示为 `Check_cond`。对于其他语句，本章用 `EMPTY` 来表示。在程序执行的过程中，内存变量值信息则根据具体的程序语句进行变化。

定义 2.2 (执行轨迹)：　一条程序执行轨迹 $\tau = s_0, s_1, \ldots, s_n$ 是一个有限的程序状态序列，即从程序位置 $s_0.l_s$ 开始执行，到达 $s_n.l_e$ 结束。其中，前一个状态的结束位置与后一个状态的开始位置相同，即 $s_i, s_{i+1} \in \tau, s_i.l_e = s_{i+1}.l_s$

给定一个针对于目标接口 f 的 IMspec 规约实例 *spec*，以及一条包含目标接口 f 调用的程序执行轨迹 τ，令 s_Δ 来表示在执行轨迹中 f 被调用的动作，即

$$s_\Delta.inst = \texttt{FCall\_f}$$

同时，令 τ_{post} 来表示 s_Δ 后续的程序状态序列。那么，执行轨迹 τ 可以表示为，

$$\tau = s_0, s_1, \ldots, s_{\Delta-1}, s_\Delta, \tau_{post}$$

特别地，针对于包含 *Fib* 的规约，这条轨迹 τ 包含对危险接口的使用，因此必然违反了接口使用约束。所以后文中，本文将不对 *Fib* 进行讨论。

因此，规约实例 $spec = (f, pre, post)$ 的语义可以基于执行轨迹 τ 表示为：在公式2-1中定义的函数 *sat* 是否能够满足，如果不能满足那么这条执行轨迹存在目标接口 f 的误用。其中函数 *sat* 由两部分组成，针对于前置条件 *pre* 的函数 sat_{pre} 与针对于后置条件 *post* 的函数 sat_{post}。

$$sat(spec, \tau) = sat_{pre}(pre, s_{\Delta-1}) \wedge sat_{post}(post, s_\Delta, \tau_{post}) \tag{2-1}$$

其中 sat_{pre} 的定义如公式2-2所示，当所有 *Pre* 中所有定义的 *Cond* 在调用 f 之前满足时，则表明 f 被正确使用。特别地，需要在状态 $s_{\Delta-1}$ 的值信息 $s_{\Delta-1}.\Omega$ 中保证这些约束被满足。否则，该执行轨迹为 f 的误用。其中，对于所有的 $eval(cond, \Omega)$，一方面如果轨迹中包含相应的显式检查 `Check_cond`，则为真。如果没有，则通过值 Ω 进行语义推理。

$$sat_{pre}(pre, s) = \bigwedge_{cond \in pre} eval(cond, s.\Omega) \tag{2-2}$$

$$eval(cond, \Omega) = \begin{cases} True, & cond \text{ is satisfied in } \Omega \\ False, & \text{otherwise} \end{cases} \tag{2-3}$$

针对于后置条件，sat_{post} 的定义如公式2-4所示，即针对于后置条件 *post* 中的每一组 $(c, acts)$，当条件 c 在 f 调用后的程序状态 $s_\Delta.\Omega$ 中满足时，后续的执行轨迹是否能够匹配动作 *acts*。匹配函数 $match(acts, \tau)$ 用来判断后续的动作是否满足约束。公式2-4的值为真则表示使用正确；反之为误用。

$$sat_{post}(post, s, \tau) = \bigwedge_{(c, acts) \in post} \Big(eval(c, s.\Omega) \implies match((acts), \tau) \Big) \tag{2-4}$$

令后置条件中 *Action* 的动作记作为 $as = a_1, a_2, \ldots, a_n$，那么函数 $match(as, \tau)$ 用来检测在程序状态序列 τ_{post} 中是否存在一组状态集合 $\rho = s_{q_0}, s_{q_1}, s_{q_2}, \ldots, s_{q_n}$ 满足如下两个条件：（1）该状态集合中带有执行时间的偏序关系，即 $q_0 < q_1 < \ldots < q_n$ and $s_{q_i} \in \tau$；(2)状态集合中状态的执行语句与规约中的动作一致，即 $\mathbb{ABS}(s_{q_i}.inst) \equiv a_i$。$\mathbb{ABS}$ 操作符用于提取 $s_{q_i}.inst$ 语句的类型以及语义。概括来说，（1）a_i 是函数调用的动作 `Call`(*fName*, *conds*) 时，$s_{q_i}.inst$ 的类型需要是 `FCall_fName`，同时值的关系需要满足，即对于每一个 $c \in conds$ 公式 $eval(c, s_{q_i-1}.\Omega)$ 为真。（2）a_i 是返回值的动作 `Return`(*fName*, *v*) 时，$s_{q_i}.inst$ 的上下文应在 `fName` 中，且类型需要是 `Return_v`。

2.5　应用与评估

基于 C 程序接口误用缺陷实例，本章设计面向接口使用约束的领域特定语言 IMSpec。IMSpec 的设计初衷是对接口使用约束进行描述，以方便开发者和研究人员了解接口使用中需要满足的条件。本节将 IMSpec 应用于实际项目中，同时对 IMSpec 的有效性进行评估。特别地，本节将从两个角度入手：

1. IMSpec 是否可以应用于实际项目中，对接口使用约束进行描述?
2. IMSpec 是否能够有效地帮助开发者理解接口使用约束?

2.5.1　描述能力评估

为回答上述“IMSpec 是否可以应用于实际项目中，对接口使用约束进行描述?”的问题，本章计划从描述能力对 IMSpec 进行应用与评估。

方法　针对于“可描述性”，本章以2.3节调研分析中缺陷实例的误用接口为对象。随机选取 150 个缺陷实例（每个项目 25 个），通过定义接口的使用约束条件，以评估 IMSpec 是否能对这些使用约束进行描述。一个 API 的使用约束可能非常复杂，同时需要深入的领域知识。所以在有限的时间内，难以对所有的约束进行撰写。因此，本章只针对于接口缺陷实例中出错的约束进行描述。

结果　如表2.6所示，在 150 个实例中共有 136 个能够通过现有的 IMSpec 语法进行描述。其中，OpenSSL 能够全部被支持。针对于不能够支持的实例，主要原因如下：

- 项目特定的语义信息。一方面，在随机选择的缺陷实例中，包含非普适性的缺陷类型。例如对异常处理中日志信息打印接口的重构（Linux-5b60fc0980）。

表 2.6　IMSpec 对接口使用约束描述结果

项目名称	缺陷数目	可描述数目	可描述比例
Linux	25	23	92%
OpenSSL	25	25	100%
FFmpeg	25	21	84%
Curl	25	23	92%
FreeRDP	25	22	88%
Httpd	25	22	88%
总计	150	136	90.67%

另一方面，则是具有项目特定的语义信息，例如图2.11(巧合的是该缺陷实例被随机选中)。目标接口 `CFArrayAppendValue()` 的参数在使用时需要满足特定的约束条件，然而该条件与项目特定的语义信息相关，同时表现形式复杂难以描述。

- IMSpec 语法不足。IMSpec 语言针对于接口使用约束的程序元素特点进行设计。同时，为方便开发者使用，本章在设计时并没有将 C 语言的所有语法结构考虑在内。例如：指针操作和取地址操作。因此，IMSpec 语法对接口使用约束的描述存在一定的不足。例如，当接口参数不仅仅存在大小关系，同时需要满足特定的算数关系时，IMSpec 则无法支持。

针对于 IMSpec 的描述能力，评估结果显示在测试用例中 IMSpec 能够支持多数（90.67%）情况接口使用约束，且语法结构简单易于使用（平均每个实例 4.35 行）。但是该语言忽略部分 C 程序属性，难以涵盖所有情况。对于 IMSpec 评估标准和改进将在第 5.2节中介绍。此外上述用例的 IMSpec 规约也将用于“有效性”评估和第4.4节的案例应用中。

2.5.2　有效性评估

为回答上述“IMSpec 是否能够有效地帮助开发者理解接口使用规约?”的问题，本章计划从描述的有效性对 IMSpec 进行评估。

方法　针对于“有效性”，本章以实际开发者作为评估对象，即通过实际开发者评估 IMSpec 是否能够有效地描述接口使用约束。OpenSSL 提供良好的接口使用用户手册 ①，因此本章对 OpenSSL 的缺陷进行随机取样并用于评估。在预实验中，作者对开发者一次能够接受的缺陷审查的数目进行调研。结果显示 5-10 个是开发者能够接受的区间。所以，本章将随机选择的 10 个缺陷实例与这些缺陷的修复版本随

① https://www.openssl.org/docs/manmaster/man3/

机排序，并构造两组测试集：A-doc 组提供 OpenSSL 项目中基于自然语言描述的接口使用说明，B-IMSpec 提供基于 IMSpec 描述的接口使用约束条件。通过随机选择的方式，每组测试集各发放给 20 名研究或者工作领域与测试、程序分析等背景相关的实际开发人员。对于 A-doc 组被测对象，作者对 OpenSSL 的背景进行介绍；对于 B-IMSpec 组被测对象，作者对 OpenSSL 背景以及 IMSpec 进行简单的介绍。特别地，为 B-IMSpec 组提供图2.14和图2.15作为参考资料。本章请这些开发人员对代码中存在的缺陷进行标注，如图2.16中所示。如果代码存在缺陷，请被测对象标注位置和原因；如果没有则打对勾。

$$P = \frac{\text{开发者报告中的真实缺陷数}}{\text{开发者报告缺陷总数}} \tag{2-5}$$

$$R = \frac{\text{开发者报告中的真实缺陷}}{10} \tag{2-6}$$

令包含误用的测试用例为真实缺陷。对于开发人员的标注结果，通过精度 P 和召回率 R 进行评估。其中，精度的定义如公式2-5所示，是被测对象找到缺陷中真实的缺陷个数与缺陷报告总数的比值。较高的精度值表示缺陷检测效果误报率低，即找到的就是缺陷。召回率的定义如公式2-6所示，是真实的缺陷与所有缺陷的商。较高的召回率表示缺陷检测能力强、漏报率低，即能够找到更多缺陷。

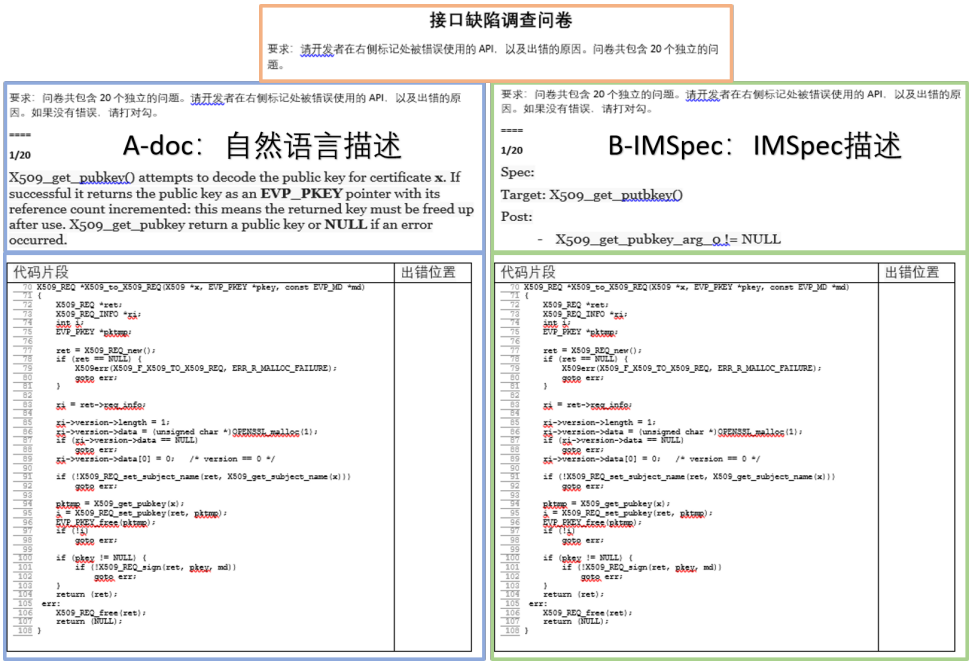

图 2.16　接口误用缺陷检测调查问卷

结果　本章将针对于有效性的评估结果展示在表2.7中。在所有发放的调查问卷中，A-doc 组共收回 13 份（65%），B-IMSpec 组共收回 18 份（90%）。在问卷中，共包含 10 个正确的使用和 10 个错误的使用。本章对收到的调查问卷中所有缺陷报告进行统计。特别地，对于问卷中没有做出标记的用例，本章认为是开发者没有能够找到该缺陷，即认为是正确使用效果等同于对勾。如表中所示，A-doc 组平均报告缺陷总数为 6.92 个，其中 6.69 个为真实缺陷；B-IMSpec 组平均报告缺陷总数为 8.06 个，其中 7.94 个为真实缺陷。在统计后，本章对两组的开发者进行二次回访，对没有回复的原因、缺陷定位的方法、误报和漏报的原因、缺陷分析中遇到的困难、与自然语言相比 IMSpec 是否能够对接口使用约束描述更有效等问题进行调研。在分析和比对后，本章将结果总结如下。

精度与召回率 如表2.7中所示，两组的精度都很高。其中 A-doc 组的平均检测精度为 96.68%，B-IMSpec 的平均检测精度为 98.51%。该结果显示，测试人员对于自然语言和 IMSpec 都能够正确理解，即两者对接口使用约束条件的描述能力相同。然而两组的召回率却存在较大差异，其中 A-doc 组的召回率为 66.9%，比 B-IMSpec 组低 12.5%。在对调查问卷本身和二次回访的结果进行分析和总结后，本章发现 A-doc 组 13 个返回结果的被测对象中有 10 个并没有完全进行，即其后几个缺陷没有做出标记，也没有打对勾或者提供任何正确使用的标记。该现象的一个主要原因是，测试人员对自然语言的描述失去耐心。在二次回访中，作者将基于 IMSpec 描述发送给这些被测对象。结果显示，开发者相比于自然语言，更愿意接受类程序语言的 IMSpec。因此，相对于自然语言，开发者认为 IMSpec 的描述有效性更好。

被测对象遇到的困难 本章对被测对象遇到的困难、误报漏报的原因进行分析和总结。结果显示，开发者在进行调研问卷的过程中，存在如下几个主要困难：

1. 时间问题。在二次回访的过程中，A-doc 组中有 3 位第一次没有回复的开发者解释，他们忽略该调研的主要原因有两个。一方面，20 个测试用例太多，测试者认为需要大量的时间进行分析。另一方面，当看到自然语言描述的规约时，测试者产生放弃进行测试的想法。
2. 结果不一致。在两组测试人员当中，绝大部分被测对象没有 OpenSSL 的开

表 2.7　基于两种描述方式的开发者调查问卷结果

实验组	问卷情况		测试集		平局缺陷报告情况		平均结果	
	发放	收回	缺陷数	正确数	报告总数	真实缺陷	*P*	*R*
A-doc	20	13	10	10	6.92	6.69	96.68%	66.90%
B-IMSpec	20	18	10	10	8.06	7.94	98.51%	79.40%

发经验。但是在提供接口使用约束的条件后，开发者能够对这些领域相关的 API 进行缺陷检测。然而，被测对象的结果存在两方面的不一致。一方面，同一个开发人员对于同种缺陷模式，有的成功找到缺陷位置和原因，有的则没有找到。另一方面，同一个缺陷，有的开发者认为是缺陷有的则认为不是缺陷。特别地，对于参数检查问题，不同的开发者理解不同，即有的认为必须进行检查有的认为可以不检查。

3. 上下文信息不足。为节省篇幅以及对问卷长度的担心，本章每个测试用例只提供目标 API 的单层函数的代码。特别地，对于长度过长的调用，进行程序截取。因此，两组测试人员都表示，很多情况下缺失的上下文信息会影响结果的判断。该影响主要表现在召回率上。

针对于 IMSpec 描述的有效性，评估结果显示实际开发者通该语言能够有效地理解接口使用约束。同时，评估结果显示相对于自然语言，IMSpec 的表现形式更利于开发者理解接口使用约束。

2.6　本章小结

本章提出基于缺陷模式的 C 程序接口使用约束领域特定语言。为有效地进行语言设计，本章首先对 C 程序中接口误用缺陷实例进行研究和总结。以不同领域、广泛使用的六个开源软件作为研究对象，对开发过程中出现的 830 个实际接口误用缺陷实例进行分析和归纳。本章总结出三类常见接口误用缺陷模式，包括：不正确的参数使用、不正确的异常处理以及不正确的因果调用关系。这些缺陷模式有利于研究人员和开发者理解 API 误用缺陷的本质，设计和开发更好的接口误用缺陷检测工具。基于常见缺陷模式，本章提出 IMSpec 领域特定语言，以描述 C 程序中接口使用约束，并给出该语言的设计动机、语法结构和形式语义。本章将 IMSpec 应用于实际项目的缺陷实例中，应用结果显示该语言能够有效地描述实际项目中接口使用约束。

第 3 章　接口误用缺陷静态检测技术

软件库通过应用编程接口（API）来封装已有功能，从而提高软件开发效率。正确的接口使用需要满足特定的约束，否则将引入接口误用，导致接口误用缺陷。静态检测技术是在不运行程序的前提下对程序行为进行分析的技术，能够在开发早期进行应用，极大地降低缺陷修复的成本。现有的静态检测技术可以分为两类，基于程序分析的缺陷检测技术和基于数据挖掘的缺陷检测技术。虽然现有工作能够对实际缺陷进行检测，但是随着软件规模扩大、程序结构复杂以及开源代码的广泛使用，现有工作面临检测精度与规模的矛盾关系。一方面，现有工作支持的缺陷模式和目标接口固定难以扩展，对用户自定义的接口支持不足；另一方面，语义分析不足，大规模程序检测结果不精确。因此，研究精准高效的 C 程序接口误用缺陷检测技术，对于提升软件系统可靠性和安全性具有重要意义。

本章旨在研究规模化接口误用缺陷静态检测技术，以对大规模程序中多种接口误用缺陷进行准确、高效的检测，弥补现有静态检测技术的不足。本章首先基于第3章的缺陷模式对现有检测工具和方法进行分析，总结现有工作的特点和不足。接着提出基于约束描述的 C 程序规模化接口误用缺陷检测方法 IMChecker。IMChecker 通过 IMSpec 语言对接口使用约束描述，从而支持多种缺陷模式和用户自定义的接口；基于多入口分析策略以应对实际项目中大规模代码精确分析的需求；并基于语义信息和统计信息对结果的精度进行提升。从全文的研究体系上看，本章的工作是描述语言 IMSpec 的应用，同时是接口误用缺陷检测实际应用的重要核心。

3.1　引言

开发者在利用 API 快速构建系统的同时，需要满足接口使用的约束条件，从而正确执行接口内部封装的功能。否则，将会产生接口误用导致软件缺陷。对接口误用检测技术研究具有重要意义。一方面，接口误用是导致软件错误、系统崩溃、漏洞产生的重要原因之一。CWE 组织 2011 年发布的最危险 25 种软件错误中，有 40% 和接口误用相关[135]。同时，OWASP 项目在 2017 年发布的最危险 10 类网络漏洞中，有 30% 和接口误用相关[136]。另一方面，随着开源社区的发展，软件库文档缺失、开发人员对 API 理解不足，导致现有的代码中存在大量的接口误用缺陷。

近年来，研究人员设计并实现各种各样的方法来检测接口误用缺陷。特别地，静态检测技术获得广泛的关注和应用。静态检测技术能够在不执行代码的情况下

进行分析，可以应用于开发的各个阶段，有效地提高代码质量。同时，静态分析在使用时可以不借助人工标记、构造测试用例和测试环境搭建，因此能够对所有的程序路径进行模拟执行，适用范围广。针对于接口缺陷检测，从分析策略上来说静态分析包含两种主要的技术路线：程序分析技术[117] 和数据挖掘技术[30]。基于程序分析技术的检测方法和工具需要研究人员和工具实现者具备接口使用的领域知识，通过规约描述或者程序硬编码的方式对目标接口使用的约束进行预先定义。此后，基于程序语义通过可达性分析、程序语义匹配等方式进行缺陷检测。基于数据挖掘技术的方法和工具则通过统计学习的方法，根据算法设计者预定义的模式在项目中进行接口使用约束推理。此后，基于推理的约束从统计意义角度进行缺陷检测。

虽然两者都能够在实际项目中进行应用并找到新的接口误用缺陷，然而在现代软件开发模式下，两者存在两个主要不足：（1）缺陷模式难以扩展，对用户自定义的接口支持不足。前者无法应用于未定义的目标 API，后者则依赖于大量高质量的数据从而学习正确的接口使用约束条件。（2）语义分析不足，分析精度不够，难以应用于实际项目中。一方面，现有的工具多基于语法层分析；另一方面，为支持大规模代码，工具多在过程内分析忽略过程间的语义信息。

为解决上述方法中的不足，平衡分析精度与规模的矛盾关系，本章提出基于约束描述的规模化接口误用缺陷静态检测方法 IMChecker。首先，IMChecker 通过 IMSpec 规约描述以支持多种缺陷模式和用户自定义的接口。精度和效率是静态分析技术需要平衡的重要指标。高精度的分析技术需要昂贵的计算代价，从而限制静态分析在实际项目上应用效果。高效率的分析技术则存在分析精度的损失，产生大量的误报（False Positive，将正确行为报告为缺陷）和漏报（False Negative，实际缺陷没有被检测出）。因此，IMChecker 基于多入口分析策略，将复杂的程序分析问题分而治之，高效率分析的同时实现局部的精确分析。针对多入口分析策略导致的分析精度损失，IMChecker 通过基于上下文的语义摘要信息和基于使用情况的统计信息进行结果过滤，以提高检测的精度。本章基于开源缺陷测试集 Juliet Test Suite 中 13 个接口缺陷相关的 CWE 分类对 IMChecker 方法进行评估。实验结果显示，IMChecker 方法误报率为 13.21%，漏报率为 16.08%。检测能力领先于主流的开源检测工具。

本章其余部分组织结构如下：3.2节对相关研究进行总结，3.3节对规模化接口误用缺陷静态检测算法进行介绍，3.4节给出工具实现和评估结果，最后在3.5节总结本章工作。

3.2　相关工作

静态缺陷检测方法在不执行程序的情况下，对程序中的缺陷进行检测。针对 C 程序接口误用缺陷，近年来研究人员和工具开发者设计并实现各种各样的静态检测算法和工具[30,117]。本节对其中的典型算法和工具进行调研和总结。从技术路线上来说，静态缺陷检测方法可以分为两大类，

- *程序分析技术* 程序分析技术需要显式地提供目标接口的使用约束，因此需要研究人员和开发者具有良好的领域知识。目前，约束的描述方式有两种：规约描述语言和程序硬编码检测插件。前者在分析的过程中，首先将语言进行解析并构造监控自动机的中间表达或进行代码插桩。在程序语义分析阶段，通过可达性分析、程序属性图搜索、语义匹配等方式对缺陷进行检测。基于描述的方法有利于对新的接口进行扩展，即提供对应的缺陷描述。后者则在分析的过程中，利用预先实现的检测插件进行缺陷检测，因此该方法难以扩展，即需要实现新的检测器。基于程序分析技术的工作，最大的优点是可以有效利用积累的领域知识，并利用语义分析获得更加准确的分析结果。
- *数据挖掘技术* 数据挖掘技术则不需要用户提供显式的约束，可以基于统计信息通过学习的方法自动推理约束。其检测的核心在于，通过对程序形式的转化构造中间表达，并基于预先定义的模式学习接口使用的约束。特别地，大多数方法认为：多数使用为正确用法，少数则为错误。虽然基于数据挖掘技术的检测方法不需要人工定义具体的约束，但是需要良好领域知识设计学习模型。因此，其扩展性有限。基于数据挖掘技术的工作，最大的优点是设计好学习模型后可以实现完全自动化，同时可以应用于不同项目。

如表3.1中所示，本章共对 17 个研究工作和工具进行调研和总结。其中前五个为基于程序分析技术的普适性静态缺陷检测工具，包括三个开源软件（Clang-SA、Cppcheck 和 Infer）以及两个商业工具的学术使用版（Pinpoint 和 Coverity）。第 6-7 个为针对接口误用检测基于程序分析技术的静态缺陷检测工具。最后 10 个则为基于数据挖掘技术的程序接口缺陷检测技术和工具。针对提供实际工具的工作，本章对工具进行使用；对于没有工具的工作，本章对论文进行阅读，总结其检测能力。特别地，为减少本章主观臆断带来的影响，对于每一个工作本章或直接与论文作者进行结果核对；或同时阅读 3-5 个引用该论文的其他工作，与这些论文中的描述进行核对。

本章从三个方面对目标研究工作和工具进行调研和总结，即检测能力、扩展性和可用性。如表3.1中所示，第一列为项目名称。第 2-9 列为工具对于第2.3节中总结的接口误用缺陷模式的支持情况。特别地，IPU-s 代表单个参数的检查、IPU-r

表 3.1　静态分析工具对 C 程序接口误用缺陷检测能力调研结果

工具名称	IPU①		IEH			ICC			扩展性②	可用性③
	-s	-r	-c	-p	-l	-s	-c	-r		
Clang-SA[77]	Y	Y	-	-	-	Y	Y	Y	难	✓
Cppcheck[78]	Y	Y	P	-	P	Y	Y	Y	有限	✓
Infer[79]	Y	Y	-	-	-	Y	Y	Y	-	✓
Pinpoint[137]	Y	Y	P	-	-	Y	Y	Y	-	✓
Coverity[138]	Y	Y	Y	P	P	Y	Y	Y	难	✓
SLAM[59]	Y	Y	Y	-	-	Y	Y	Y	有限	✓
SSLINT[82]	P	-	Y	-	-	Y	Y	-	有限	-
PR-Miner[89]	-	-	-	-	-	Y	Y	-	-	-
RGJ07[139]	Y	Y	-	-	-	Y	Y	-	-	-
Chronicler[140]	-	-	-	-	-	Y	P	-	-	-
EDP[133]	-	-	Y	Y	-	-	-	-	-	-
Hector[141]	-	-	-	-	-	Y	P	-	-	-
Chucky[142]	P	P	Y	-	-	-	-	P		✓
NDNR14[92]	P	Y	-	-	-	-	-	-	-	-
APISan[94]	P	P	Y	Y	-	Y	Y	P	-	✓
Antminer[95]	P	P	Y	P	-	P	P	-	-	-
ErrDoc[143]	-	-	Y	Y	P	P	-	-	有限	P

① Y：支持，P：部分支持，-：不支持
② 难：难以扩展，有限：可以扩展，扩展内容受限，-：不支持，✓：方便扩展
③ ✓：可用且基本符合预期，P：可用但与预期相差很大，-：不可用

代表参数之间和参数与返回值之间的检测；IEH-c 代表异常处理中对接口返回值的检测、IEH-p 为异常处理中返回错误代码的检测（error propagation）、IEH-l 为异常处理中对缺陷信息打印支持的检查；ICC-s 为不带上下文关系的接口对、ICC-c 为带有上下文关系的接口对、ICC-r 代表重复调用的检测。第 10 列扩展性关注算法和工具是否预留扩展的接口，能够针对用户需求对项目特定的接口检测进行扩展。其中 ‘‘难’’ 表示可以扩展但是代价较大，比如 Clang-SA 需要重新设计检查器插件；‘‘有限’’ 代表工具提供扩展的方式但是扩展的内容需要满足特定的缺陷模式。最后一列为工具的可用性，即该工具是否可用。特别地，工具的效果是否和发表的论文、工具说明一致。其中，P 代表工具可以下载和使用但是其功能和论文描述不符，即工具难以支持论文中明确给出的代码样例。

检测能力方面，从表中调研结果可知在所有的工作中并没有一个工作能够完全支持所有的接口缺陷模式。特别地，基于数据挖掘技术的工作中部分（40%）工作只能处理某一类缺陷模式，绝大部分（80%）工作不能支持所有类别。检测能力

最好的是商业工具 Coverity 和基于数据挖掘技术的工具 APISan。然而前者实际价格昂贵，难以广泛使用。后者基于数据挖掘技术，本章对工具试用时发现具有极高的误报率（工具作者在论文和工具主页中同样指出高误报率）。此外，在所有缺陷模式中，单个参数检查、异常处理中返回值检查和不带有上下文语义关系的函数调用对检查被大多数工具支持。这些缺陷模式相对简单，能够从语法结构直接进行检查。然而，本章发现绝大部分工具并没有考虑足够的语义信息。例如，对于参数的检查，如果缺少参数或者代码中参数与常量比较则工具支持效果较好；但是如果比较的方式是变量比较，则工具存在大量漏报。

扩展性方面，只有 6 个工具提供扩展的接口。其中 Clang-SA 和 Coverity 需要设计新的检测器插件，扩展难度极大。SLAM、SSLINT 和 Errdoc 可以通过撰写规约的方式对检测目标扩展，然而其只能针对于特定的缺陷模式或者项目特定的接口扩展，即 SLAM 针对于 Windows 操作系统内核的驱动程序设计、SSLINT 针对于 SSL/TLS 安全协议设计和 Errdoc 针对于异常处理设计。Cppcheck 为用户提供最方便的扩展方式，通过提供结构化的接口使用描述方式，用户可以对自定义的接口进行扩展。然而，Cppcheck 提供的扩展语言只能够支持单个参数、参数关系、返回值检测和不带因果关系的函数对四种情况。

应用性方面，基于数据挖掘的工作中可用的工具只有 Chucky 和 APISan。Errdoc 工具在试用时，本章发现其与论文描述差距较大。基于程序分析技术的工具本章选择的都是可以使用的工具，因此实用性较好。然而，在实际项目应用中本章发现这些工具的分析报告存在两种极端的表现，即要么报告数量多但存在大量的误报；要么报告数极少，与其他工具相比存在大量的漏报。

总体来说，针对 C 程序接口误用缺陷检测，现有的工作存在两点主要不足：（1）缺陷模式支持有限扩展性不足，难以支持用户自定义的接口。一方面，基于程序分析的技术需要通过规约撰写或者开发新的检测插件。如第2.2节和本章调研结果所示，现有的约束描述方法或过于复杂，或难以扩展到全部接口缺陷模式。同时，开发新的插件则需要理解工具框架和细节，难以实际使用。另一方面，基于数据挖掘的技术需要大量可靠数据进行约束条件的学习。然而，该需求对于独立的项目难以实现。特别地，用户自定义的接口往往使用次数有限，难以满足学习所需要的数据量。（2）语义分析不足，检测结果存在大量误报和漏报，难以支持实际项目分析。一方面，为提高缺陷检测的效率，现有的工具多基于语法结构进行缺陷分析。因此，难以支持需要语义信息的缺陷类型。另一方面，多数工具基于过程内分析策略，以应对大规模程序。然而，丢失的上下文语义信息会导致误报和漏报。

3.3　接口缺陷静态检测方法

本节提出 IMChecker，基于约束描述的规模化接口误用缺陷检测方法，从而改善现有技术的不足。首先，IMChecker 利用 IMSpec 领域特定语言描述接口使用约束，以支持常见约束类型和用户自定义的接口，扩展检测的应用范围。接着基于多入口分析方法将复杂的程序分析问题分而治之，以提供全局高效分析的同时实现局部的精确分析。最后，通过上下文的语义摘要信息和使用情况的统计信息对检测结果过滤，以提高检测的精度。下文中，本章将通过图1.1中不包含修复片段代码作为案例，介绍 IMChecker 方法的流程和核心思想，并对其中的关键步骤进行详细解释和分析。

3.3.1　IMChecker 工作流

如图 3.1中所示，IMChecker 的工作流程包含三个主要步骤：预处理、静态分析和过滤。

预处理阶段，IMChecker 以用户提供的源代码和 IMSpec 描述文件作为输入，构造分析的上下文环境。一方面，将源代码进行预处理，生成 LLVM-IR[①]中间表达。并基于 IR 指令构造程序的控制流自动机（Control-Flow Automaton，CFA）。另一方面，IMChecker 解析 IMSpec 约束描述实例。对接口使用约束进行划分，将一个目标接口的描述根据语义分解成调研中总结出的三大类约束模式，确定缺陷检

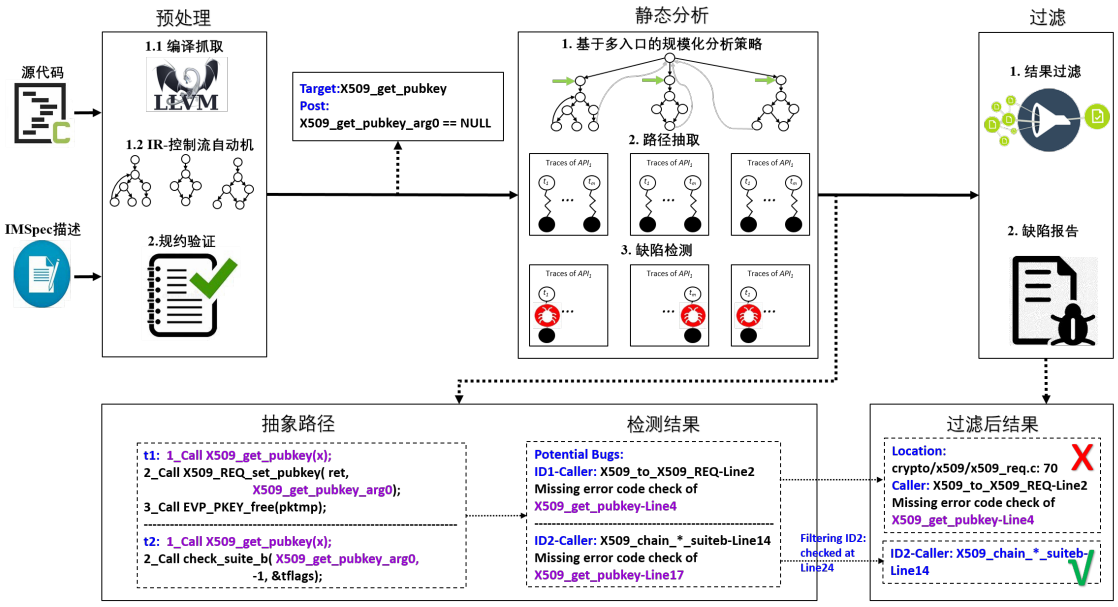

图 3.1　IMChecker 工作流图

① http://releases.llvm.org/3.9.1/docs/LangRef.html

测的目标。

分析阶段，IMChecker 首先根据 CFA 的结构，基于多入口分析策略选择入口，以支持大规模程序分析。针对每个待分析入口，根据接口使用的特点，基于符号执行技术在过程内抽取路径信息，在保留有效语义的情况下简化程序结构。接着利用接口误用缺陷检测算法，通过模式匹配的方式对目标接口的路径信息和使用约束进行分析，并生成初步的检测结果。例如，图3.1中下侧给出图1.1中例子的抽象路径。针对该用例，共存在两条不满足使用约束的路径。

过滤阶段，IMChecker 利用上下文的语义信息和使用情况的统计信息对结果进行过滤与排序。前者面向由于过程内路径提取和多入口分析策略带来的上下文信息丢失所导致的误报。后者则针对于接口使用特殊用法或者 IMSpec 描述中的错误。针对图1.1中例子，基于上下文的语义信息可知第二条路径中缺失的非空指针约束在指针使用之前进行检查。因此，第二条路径被过滤，从而最终的缺陷报告只有一个。

下文中，将对 IMChecker 方法中的关键步骤进行详细介绍。

3.3.2　构造分析上下文

基于静态分析的缺陷检测技术需要对纯文字的 C 程序源代码进行预处理，构造中间表达。目前常用的方式有字节流（Token）、抽象语法树（Abstract Syntax Tree）、自定义的中间表达（Intermediate Representation）等等。同时，基于这些中间表达构造图模型，将缺陷检测问题转化为图遍历问题或者图的可达性问题等等。IMChecker 采用 LLVM-IR 作为中间表达，并基于 CFA 创建图模型，构造分析上下文。

IMChecker 对用户提供的源代码通过预编译的方式生成 LLVM-IR 中间表达。LLVM-IR 旨在提供一个轻量级、低级别、高表达能力的程序语义表示形式，支持类型信息且易扩展。它的目标是成为一种“通用 IR”，通过低层次的语义表达将高层次的语言完全地映射到 LLVM-IR。即类似于微处理器的 IR 方式，允许许多源语言进行映射。通过提供类型信息，LLVM-IR 可以用于各种程序分析任务中。LLVM-IR 具有静态单赋值的属性（SSA），并且每个指令具有原子语义。因此，可以将复杂的 C 语句转化为一系列的原子操作，方便语义的计算。

在预处理生成 LLVM-IR 后，IMChecker 基于 CFA 构造程序图结构。如图3.2中所示，CFA 以指令位置 $l \in L$ 为节点，以控制流操作的指令 $e \in E = L \times I \times L$ 为边。具体来说，每条边 e 存在一个起始指令位置和终止指令位置，边上记录的是具体的指令 $i \in I$。针对 LLVM-IR，I 是所有 IR 的指令集合。特别地，本章将边分

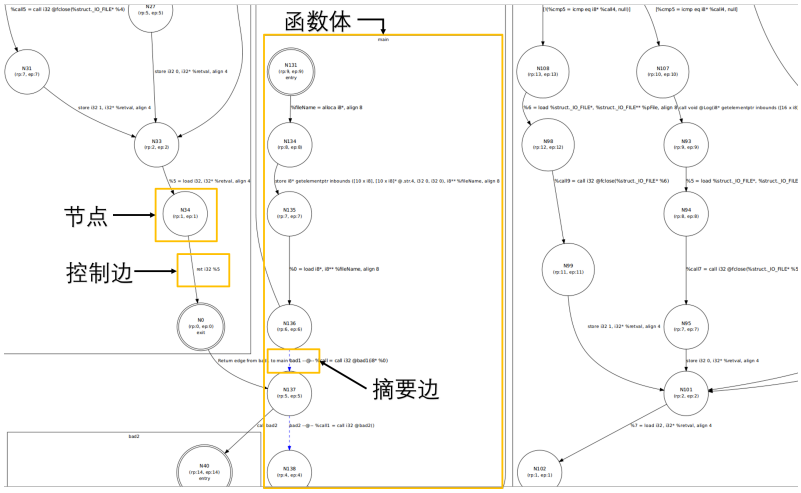

图 3.2　CFA 示意图

为两大类，即控制边（Control Edge）和摘要边（Summary Edge）。前者 i 为具体的语义操作的指令（例如，存储、运算、返回等等），边的起点和终点为 i 的起始位置和终止位置。后者表示函数调用和循环关系。其中，函数调用摘要边上的语句 i 为具体的函数调用指令，循环摘要则为空；边的起始和终止位置则为函数和循环的起始位置和终止位置。循环的起始指令和终止指令通过强连通分量（Strongly Connected Components，SCC）[144] 计算。因此，通过摘要边，在分析中可以利用摘要信息跳过函数展开和循环遍历，在保证大规模代码有效分析（即不展开函数和循环）的同时增加分析精度（即利用预先计算的摘要信息）。

需要说明的是，本章基于 LLVM-IR 实现工具。但 IR 具有复杂的指令集合，需要大量的领域知识。因此，本章后续中使用原生的 C 语法帮助读者理解 IMChecker 的核心方法。

3.3.3　IMSpec 规约分解

难以支持用户自定义的接口是现有检测工作实际应用的重要瓶颈之一。在缺陷实例调研中本文发现接口使用约束具有普适性，最典型的例子就是用户自定义的资源管理接口与 C 标准库内存操作接口 `malloc/free` 具有相同的行为和使用约束。通过用户自定义的接口使用描述，重复利用已实现的检测引擎能够有效地提高缺陷检测能力。因此，IMChecker 工具以 IMSpec 约束描述作为输入，基于描述实例进行缺陷检测。

如第2.3节所述，接口误用缺陷具有三种常见类型：参数相关的 IPU、异常处理相关的 IEH 和调用关系相关的 ICC。每种缺陷模式都有自己的程序行为特点。通过针对性的检测方法，能够有效地简化缺陷检测难度、提高检测准确率。因此，

算法 1: IMSpec 分解算法

输入: 用户输入的 IMSpec 实例集合 Spec

输出: 分解后的规约集合 SpecMap

```
1  for Spec 中的每个实例 spec do
2      s ← Parse (spec);
3      if s 出错 then continue;
4      item ← NewItem ();
5      item.target ← s.target;
6      if s 具有 pre 约束条件 then
7          foreach s.pre 中的 p do
8              item.IPU ← Union (item.IPU, p)
9      if s 具有 post 约束条件 then
10         for s.post 中的 pt do
11             if pt 包含函数调用且和 item.target 有数据依赖关系 then
                   item.ICC ← Union (item.ICC, pt) ;
12             else item.IEH ← Union (item.IEH, pt) ;
13     SpecMap ← Union (SpecMap, item.target, item);
```

IMChecker 在 IMSpec 解析阶段对规约描述实例进行语义分析，将目标 API 完整的接口使用约束进行分解。在后续的检测过程中，利用分解后的原子约束进行检测。

IMSpec 分解算法如算法1所示，本章以图2.15为例，对算法进行解释。分解算法以用户提供的 IMSpec 实例集合作为输入，以分解后的规约集合 **`SpecMap`** 作为输出。对于每一个实例 **`spec`**，首先进行语法解析，并确认目标接口 `target`。此后，基于解析后的结果 *s* 进行分解。因此，例子中的两个目标接口为 `fopen` 和 `fgets`。

在 *s* 的分解过程中，如果 *s* 中具有前置条件 *pre*，那么对于每一个独立的约束条件 *p* 都归为 IPU 类型中。即通过约束

```
fopen_arg_1 != NULL,
fopen_arg_2 IN (r, w, a, r+, w+, a+)
```

可知，`fopen` 拥有两个前置条件需要满足。

如果 *s* 中具有后置条件 *post*，那么对于每一个独立的约束条件 *pt*，判断其中是否含有函数调用动作 *Call*，并且 *Call* 中是否与 `target` 存在数据依赖关系。如

果满足上述两条约束，则归为 ICC 类别；否则归为 IEH 类别。因此，`fopen` 的后置条件中，

```
fopen_arg_0 == NULL, RETURN(foo:FILEERR);
fopen_arg_0 != NULL, CALL(fclose: fopen_arg_0 == fclose_arg_1)
```

第一条没有 *Call* 动作，为 IEH 类型；第二条则是 ICC 类型，因为存在 *Call* 动作，且存在数据依赖关系（即 `fclose` 的参数为 `fopen` 的返回值）。

3.3.4　多入口分析策略

经典的程序分析方法以主函数为入口，自顶向下对程序分析。然而这种方式在面对大规模代码分析时，其分析效率存在巨大挑战。因此，研究人员和工具开发人员选择逐函数的分析策略，从而降低分析复杂度。类似地，IMChecker 采用多入口的分析策略，即将一个复杂的程序分析问题分而治之，在保证全局有效规模化分析的同时，尽可能提高局部分析精度。

如图3.3中示意图所示，IMChecker 在 CFA 上对目标接口的使用情况进行检索，以包含目标接口 f 的函数调用上下文 c 作为分析入口。如图中黄色边为 f 的调用指令，绿色箭头所指向的 c 为分析入口。然而，基于函数入口进行分析，随着函数展开层数增加、循环展开，依旧面临着状态空间爆炸的问题。因此，IMChecker 在分析过程中，采取两个策略以支持大规模分析的同时，尽可能地降低精度损失：

- 限制循环展开。循环展开问题是静态程序分析的难点之一。第2.3节的调研结果显示，绝大部分接口使用错误与循环无关。因此，IMChecker 在分析的过程中，只展开一次循环以提高分析的效率。尽管这种策略会导致路径信息不全，降低分析的精度。但是，这并不会显著地影响分析的结果。
- 限制过程间分析。在 IMChecker 的分析过程中，通过符号执行的方式对每一个 c 进行路径提取。对函数展开会带来极大的分析负担，影响分析效率。同

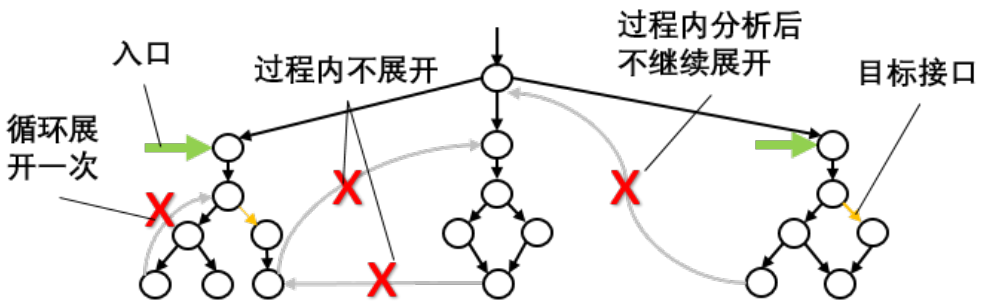

图 3.3　多入口分析策略示意图

时，第2.3节的调研结果显示，大部分接口缺陷可以在过程内进行检测。所以，IMChecker 在执行时对 c 内的其他函数调用不展开，即过程内不展开。同时在 c 执行后，不会继续执行，即过程内分析后不继续展开。函数展开层数会影响接口使用缺陷检测的精度，本章将在过滤阶段通过引入上下文语义摘要信息过滤分析结果，以提高检测精度。

3.3.5　抽象符号路径提取

在缺陷检测之前，IMChecker 通过符号执行（symbolic execution）对程序进行静态模拟执行，并基于符号对程序路径的语义信息进行抽象描述。该方法旨在保留接口相关的程序语义前提下尽可能简化程序结构。

令 $\mathbb{N},\mathbb{Z}$ 表示非负整数和全体整数。在图3.4中，本节给出 IMChecker 路径抽取的抽象语法结构。其中，$id \in \mathbb{N}, n \in \mathbb{N}, z \in \mathbb{Z}$。IMChecker 关注整数和指针两种变量。特别地，利用 Access-path[145] 结构来代表内存地址，简写为 ap。ap 的形式为正则语言 $v(o|d)^*(o|\epsilon)$，即 ap 是首地址 v 与偏移 o 和解引用 d 的组合。每一条路径 t 由一系列的路径动作 a^+ 和动作结束时值映射关系 V 组成。需要说明的是，IMChecker 基于 LLVM-IR 实现，所有的值只会被赋值一次，所以只需要维护一个 V 即可。IMChecker 基于 CFA 图结构，在进行图遍历的时候进行动作提取和值分析。因此，IMChecker 支持流敏感（flow-sensitive）分析。目前 IMChecker 关注三种程序语句：if 语句判断条件 **Assume**、函数调用 **Call** 和返回值 **Return**。特别地，**Assume** 能够有效地捕获路径可达性信息（path-sensitive）。在分析过程中，IMChecker 对每一个动作分配一个唯一的 id，以区别分析的上下文环境（context-sensitive）。V 记录符号变量 sv 到具体值 cv 的映射关系。一个符号值由具体动作产生的 id、产生的接口、以及对应的参数位置 n 构成。例如，$id\_f\_$**arg_**i 表示第 id^{th} 动作为函数调

(Traces) T ::= t

(trace) t ::= $(id\_a)^+; V$

(action) a ::= **Assume**(exp) | **Call** $f(sv^*)$ | **Return**(sv)

(expression) exp ::= $sv1\ cmpop\ sv2$

(value map) V ::= $sv \rightarrow cv$

(symbolic variable) sv ::= ($id\_f\_$**arg_**n)

(compare operator) $cmpop$ ::= ! = | == | >= | > | <= | <

(concrete value) cv ::= z | ap | **NULL**

(function) $f \in \mathbb{F}$

图 3.4　IMChecker 路径信息提取抽象语法

t_1 : 1_**Call** X509_get_pubkey(_);
　　2_**Call** X509_REQ_set_pubkey(_, **1_X509_get_pubkey_arg_0**) ;
t_2 : 1_**Call** X509_get_pubkey(_);
　　2_**Assume**(**1_X509_get_pubkey_arg_0 != NULL**) ;
　　3_**Call** X509_REQ_set_pubkey(_, **1_X509_get_pubkey_arg_0**) ;
t_3 : 1_**Call** X509_get_pubkey(_);
　　2_**Call** check_suite_b(**1_X509_get_pubkey_arg_0**, _, _) ;

图 3.5　图 1.1中抽象路径提取结果

用，其目标接口 f 的第 i^{th} 个参数。特别地，IMChecker 用 0 表示返回值索引，即 f**_arg_**0 表示接口 f 的返回值。在 **Return** 中，用 **arg_**0 表示 f 的调用者 c 的返回值。综上所示，IMChecker 抽取的路径信息能够支持流敏感、路径敏感、上下文敏感的语义信息。

在图3.5中，给出图1.1代码片段的路径信息。其中，“_”为与例子不相关的变量。图1.1代码片段共有三条路径，t_1 和 t_3 为原始代码的路径信息，t_2 为修复后代码的路径信息。所有三条路径起始于 `X_509_get_pubkey()` 函数调用。t_1 直接将接口的返回值 **1_X509_get_pubkey_arg_0** 传递给 `X509_REQ_set_pubkey()`，忽略对该返回值的检查。t_2 中，则对该返回值进行非空检测。t_3 中，同样没有直接对返回值进行检查，并直接传递给 `check_suite_b()` 的第一个参数。

3.3.6　缺陷检测算法

IMChecker 的缺陷检测算法如算法2中所示，以分解后的 SpecMap 和符号路径轨迹映射 Trace_Map 作为输入，以缺陷检测结果 R 为输出。其中, 符号路径轨迹映射是接口到包含该接口路径集合 T:

$$\text{Trace\_Map} : f \rightarrow T$$

缺陷检测结果，是目标接口到缺陷信息（违反的约束 s_{item} 和路径 t）的集合:

$$R : f \rightarrow (s_{item}, t)$$

IMChecker 对每一个约束实例 item 中的目标接口进行相关路径提取获得路径集合 T，再对集合中的每一条独立的子路径 t 进行分析。路径分析包含三个部分，分别对应于约束的三个模式，即 IPU、IEH 和 ICC。

SatisfyIPU 对使用约束中的前置条件进行检测:（1）如果在路径 t 中存在语义等价的 **Assume**，那么该约束 p 则满足；反之如果 **Assume** 与 p 的语义相反，那么

算法 2: IMChecker 缺陷检测算法

输入: 分解后的规约集合 SpecMap，符号路径轨迹映射 Trace_Map

输出: 缺陷报告 R

```
1  for SpecMap 中的每个实例 item do
2      R ← ∅;
3      T ← Extract (item.target);
4      for T 中的每一条路径 t do
5          if item.IPU 存在 then
6              foreach item.IPU 中的 p do
7                  b ← SatisfyIPU (p, t);
8                  if !b then R ← Union (item.target, R, p, t) ;
9          if item.IEH 存在 then
10             foreach item.IEH 中的 eh do
11                 b ← SatisfyIEH (eh, t);
12                 if !b then R ← Union (item.target, R, eh, t) ;
13         if item.ICC 存在 then
14             foreach item.ICC 中的 cc do
15                 b ← SatisfyICC (cc, t);
16                 if !b then R ← Union (item.target, R, cc, t) ;
```

该路径则一定错误。（2）如果在路径 t 中不存在语义等价的 **Assume**，那么则在路径上对目标接口的函数调用动作 **Call** 中的值映射关系 V 中判断 p 是否满足。如果满足则为路径对目标接口使用正确，反之不正确。

SatisfyIEH 对使用约束中后置条件中异常处理相关的约束进行检测。每一个 $eh = (cond, a^+)$ 由两部分组成，错误路径发生条件 $cond$ （即 $cond$ 是包含目标接口的返回值 ret 的布尔表达式，表示该情况下目标接口发生错误）和异常处理动作 a^+（即异常处理路径上需要进行的异常处理操作）。因此，在缺陷检测时首先判断路径是否为缺陷发生路径，再判断异常处理动作是否满足约束：（1）在目标函数调用后的子轨迹 t' 中，判断是否存在 **Assume** 动作，该动作与 ret 相关。如果不存在，则路径缺少了错误代码检查，为错误使用。在相关的 **Assume** 动作中，如果 **Assume** 与 $cond$ 不等价则为非异常处理路径，不需要继续分析。（2）在异常处理

路径上，对 a^+ 中的每个约束进行判断，是否存在等价的路径动作。其中，函数调用约束对应于 **Call** 动作分析，返回值约束对应于 **Return** 动作分析。如果存在不满足的约束，那么这条轨迹为错误使用。

SatisfyICC 对使用约束中后置条件中因果调用关系相关的约束进行检测。每一个 $cc = (cond, g)$ 由两部分组成，因果调用关系上下文语义约束 $cond$（即只有在 $cond$ 满足的情况下才调用后续接口），和因果调用关系的第二个接口 g。因此，在缺陷检测时分为两种情况：（1）$cond$ 为 True（即目标接口和 g 不存在上下文约束关系），那么如果不存在等价于 g 的 **Call** 动作，路径为错误使用。（2）$cond$ 不为 True。一方面，如果存在 **Assume** 动作等价于 $cond$，则需要对 g 进行检查。其中，如果不存在等价的 **Call**，路径为误用；反之为正确使用。另一方面，如果存在 **Assume** 动作不等价于 $cond$，则判断是否存在与 g 等价的 **Call**。如果存在等价的 **Call** 动作，那么路径使用错误。

本章以图3.5中的路径作为例子进行解释。接口 `X509_get_pubkey()` 用于解码证书，当发生错误时其返回 NULL 作为错误代码。所以，其返回值需要进行检查，即 IMSpec 规约为

```
Target: X509_get_pubkey(_) -> _
Post:
  // 需要对错误状态代码进行检查，检查的约束为返回值不是NULL
  - X509_get_pubkey_arg_0 != NULL
```

然而在图3.5中，只有路径 t_2 进行显式地检测，即存在路径动作

```
2_Assume(1_X509_get_pubkey_arg_0 != NULL)
```

因此，IMChecker 认为路径 t_1, t_3 为错误使用路径，被加入到缺陷报告 R 中。

3.3.7　检测结果过滤

为支持规模化接口缺陷检测，IMChecker 采用多入口分析策略，并在分析过程中对函数展开和循环展开进行限制。这种分析策略会忽略跨越函数的上下文语义信息，损失静态分析的精度，导致大量的误报。例如图3.5中的例子，虽然 t_1, t_3 都缺少必要的返回值检测，然而如图1.1中代码 27 行所示，

```
pk = X509_get_pubkey(x);
rv = check_suite_b(pk, -1, &tflags);

static int check_suite_b(EVP_PKEY *pkey,...){
  […]
  // ensure pkey not NULL
  if (pkey && ...)
```

轨迹 t_3 在参数使用的位置确保该指针不为空，即 t_3 为误报。为解决上述精度损失问题，IMChecker 通过两种策略对结果进行过滤，以提升检测精度。

跨函数语义摘要　跨函数的语义摘要过滤方式，旨在通过对跨函数的语义信息计算对检测结果进行过滤。通过对实际项目的预实验后，IMChecker 目前对四种情况进行过滤。令 f 为目标接口，r 为 f 的返回值且携带错误状态信息，c 为调用 f 的函数，g 为需要同 f 协同使用的因果调用函数，H 为 c 中除 f 的函数调用集合。则过滤的模式为：

- r 赋值给 c 的参数。一种常见的情况为，f 为资源申请函数（例如 `malloc`），并将申请后的指针 r 赋值给 c（`foo`）的参数（ptr），并在 c 的上下文中对 r 进行管理。例如下列代码片段，

```
int foo(char **ptr) {
   *ptr = (char *)malloc(100*sizeof(char));
   return 1;
}
int bar(){
   char *p = NULL;
   foo(&p);
   // handle p
}
```

  因此，在 c 的函数体内，并没有对该指针检查或者释放。

- r 赋值给 c 的返回值。与上述情况相似，c 直接返回 r，例如下列代码片段，

```
char *foo(int size) {
   char *data;
   data = (char *)malloc(size*sizeof(char));
   return data;
}
int bar(){
   char *p = NULL;
   p = foo(100);
   // handle p
}
```

- r 作为 $h \in H$ 的参数使用。一种常见的 API 使用方式是在接口的实现内进行参数检测。因此，如果 r 作为参数传递给 h，并且 h 对参数检查，那么 f 的使用则正确。例图1.1例子中 t_3 的情况。
- g 在将 r 作为参数的 $h \in H$ 中调用，即 r 约束满足，$f - g$ 需要协同使用（`malloc/free`）。在 c 中（`foo`），r（data）满足约束，并作为 h（`bar`）的参数，而且 g 在 h 中调用。例如下列代码，

```
void bar(char *p) {
   if (p != NULL) free(p);
}
void foo(){
   char * data = (char *)malloc(100*sizeof(char));
   if (data == NULL) return;
   bar(data);
   ...
}
```

接口误用统计信息　实际项目中的接口使用情况多种多样、部分接口具有特殊用法、用户提供错误的 IMSpec 等等都会导致检测工具产生误报。为过滤这些误报提升检测精度并对结果进行有效排序，IMChecker 通过基于使用情况的统计信息进行结果优化。在分析过程中，IMChecker 分别计算正确使用路径 u_T 和错误路径 u_F 的个数。首先，对于所有使用次数 $u_T + u_F$ 小于最小支持度 α（minimum support) 的目标接口 f 的缺陷报告，IMChecker 归纳为低置信度（confidence)。IMChecker 将该缺陷归纳为警告（Warning)。例如，一个目标接口只有一次使用并且该使用不符合用户提供的约束，很难判断检测结果的正确性。对于超过 α 的目标接口，IMChecker 根据错误置信度 $\mathcal{H}$，

$$\mathcal{H} = (u_T - u_F) \times \frac{u_T}{u_F} \tag{3-1}$$

对结果进行排序。当正确使用与错误使用的比例越大时，则认为越多的正确使用强烈地支持这个误用为错误。当比例相同时，认为正确的次数比错误的次数多时，则可信度更高。例如，$\mathcal{H}(u_T = 2, u_F = 1)$ 与 $H(u_T = 3, u_F = 2)$ 相比，前者的比例更高；$H(u_T = 3, u_F = 1)$ 与 $H(u_T = 6, u_F = 2)$ 相比，则后者更可信。特别地，当 $\mathcal{H} \leq 1$ 时，即错误与正确一样多或者多于正确使用时，IMChecker 将该缺陷归纳为警告类型。

3.4　工具实现与实验评估

本节首先介绍规模化接口误用缺陷检测方法 IMChecker 的实现细节。随后在公开数据集上，对 IMChecker 的分析能力进行评估，并给出 IMChecker 与主流开源工具的比较结果。

3.4.1　工具实现

IMChecker 基于 Java 语言实现，依赖 LLVM3.9 的 IR 作为中间表达，并基于 CPA[68] 算法作为整体方法的实现基础。工具以用户提供的源代码和 IMSpec 规约

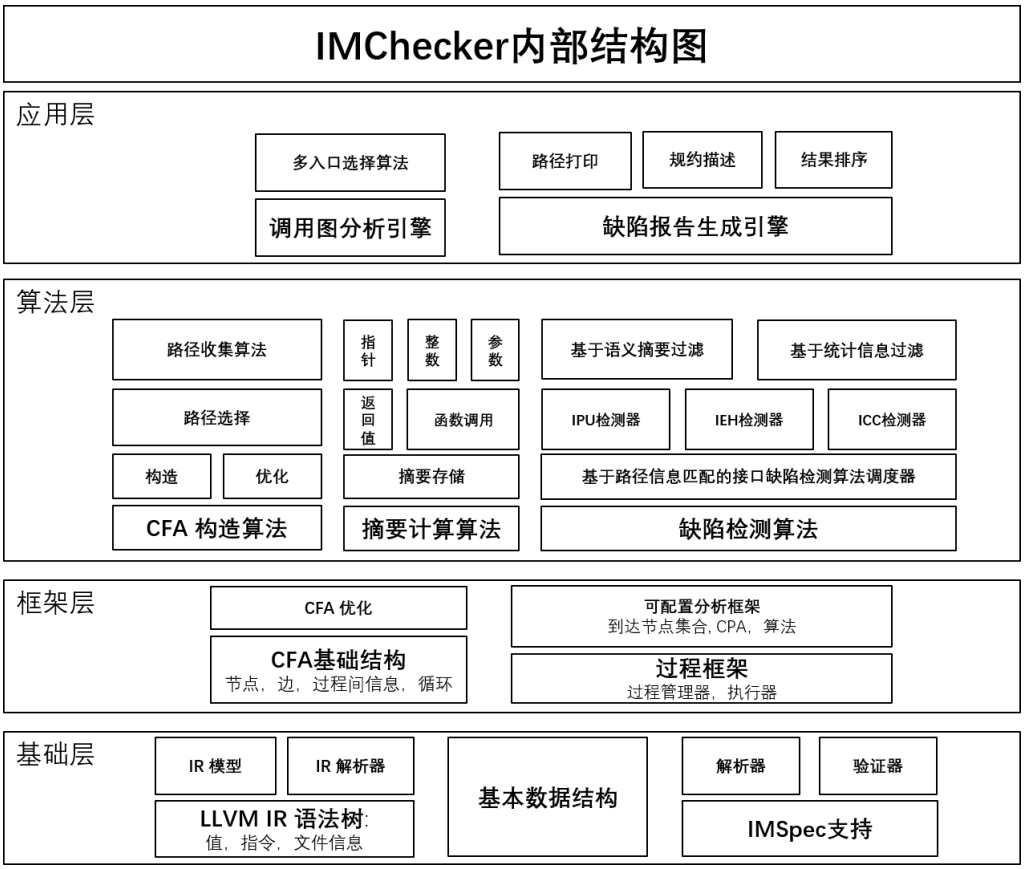

图 3.6　IMChecker 工具实现内部结构图

描述作为输入。其中 IMSpec 实例和缺陷检测结果都基于 Yaml[146] 格式存储。工具实现的内部结构如图3.6所示，共包含四个层次：基础层、架构层、算法层和应用层。首先，基础层通过对源代码进行预处理，生成 LLVM-IR 中间表达；并对 IMSpec 规约描述进行解析、一致性分析和规约化简。框架层构造 CFA，将程序转化为图结构；同时，提供 CPA 算法的分析流程以支持上层语义计算。算法层，则对 CFA 进行优化、计算语义摘要信息以及缺陷检测。最后，应用层通过函数调用图进行分析入口的选择，执行缺陷分析；并将缺陷检测的结果进行过滤、排序和信息打印，并生成最终报告。

基础层　基础层旨在对用户输入的源代码和 IMSpec 规约描述进行预处理，同时提供分析中需要的基本数据结构。首先，对于用户输入的源代码，如果待测程序是单个的 C 文件，IMChecker 直接使用 clang 编译器

```
Ubuntu@ : clang path2source −S −emit−llvm −g
```

进行预处理以生成自包含的 *.ll 文件。该文件相比于 *.c 源文件增加必需的函数和数据结构声明、展开宏定义、消除预处理指令等，并生成对应的 LLVM-IR 中间表

达。如果待测程序是一个 Makefile 工程，那么通过解析 Makefile 中的编译指令并且将所有输出 *.o 文件的指令替换为相应的预处理指令，例如 clang -E。通过预处理生成的 *.i 文件，利用上述指令生成 *.ll 文件。同时，根据 Makefile 的编译目标将项目组织成不同的分析任务。通过 llvm 的链接指令（llvm-link）将给定分析任务下所有的 *.ll 文件进行合并，作为输入进行缺陷检测。在获得 LLVM-IR 中间表达后，利用 javacpp[①]工具解析文件，并构造 IR 模型，包括值、指令和文件信息。另一方面，基础层对用户提供的 IMSpec 语言进行解析。IMSpec 语言面向单个目标接口设计，所以如果规约中存在语法错误则将错误规约忽略并输出给使用者。同时，解析后的 IMSpec 进行语义一致性验证，即是否存在冲突的约束。例如，一个条件是接口的第一个参数大于零，另一个条件是第一个参数小于零等等。

框架层　框架层旨在提供分析框架的实现以及 CFA 基础结构的创建。在对用户提供的源代码解析过后，CFA 基础结构模块对 LLVM-IR 进行包装，创建 CFA 图结构。特别地，CFA 在节点上表示程序的位置，在边上封装程序的具体执行指令。为在 CFA 图上提供更多的语义信息，本章将 CFA 边分为两种，即控制边和摘要边。前者表示具体的语义操作的指令，例如存储、运算、返回等等。后者表示函数调用和循环关系。因此，通过摘要边在分析中可以利用摘要信息跳过函数展开和循环遍历，从而保证大规模代码有效分析的同时提高分析精度。IMChecker 基于过程（Phase）来运行，即每个过程负责不同的预处理或者检测步骤。所以，框架层中实现过程管理的调度器。特别地，IMChecker 基于 CPA 算法来实现具体的路径抽取和缺陷检测。因此 IMChecker 在框架层实现 CPA 算法的核心数据结构和基础调度算法，具体的每一个 CPA 则在算法层中实现。

算法层　算法层旨在实现具体的算法细节。IMChecker 针对 C 程序接口误用缺陷进行检测，具体的实现可以分为三个模块：路径收集、摘要计算和缺陷检测。

- *路径收集* 在路径收集阶段，IMChecker 利用 CFA 构造算法，对框架层的 CFA 原型进行优化。特别地，为支持大规模程序分析，IMChecker 基于入口分析。因此，在这个阶段对入口进行标记。此后，基于优化后的 CFA 图结构，通过第3.3节中的路径抽取方法，收集路径信息，为后续两个步骤做准备。目前 IMChecker 基于 AccessPath[145] 和整数分析对语义信息进行计算。
- *摘要计算* 摘要计算的结果将被应用于缺陷检测当中。在实现层面，IMChecker 则先进行摘要的计算。为支持大规模程序检测，IMChecker 采用基于多入口分析的策略，使得跨函数语义信息丢失导致分析精度下降产生误报。为提供

① https://github.com/bytedeco/javacpp

更加准确的检测结果，IMChecker 利用上下文语义信息对结果进行过滤。具体而言，IMChecker 进行：上下文指针常量计算、整数常量计算、参数约束关系、返回值常量关系和函数调用关系的摘要信息。

- **缺陷检测** 缺陷检测部分，IMChecker 实现基于路径信息匹配的接口误用缺陷检测算法。特别地，为提高检测精度 IMChecker 实现检测算法调度器，即根据使用约束模式的不同调用不同的检测算法实现，以精确分析程序中的接口误用。最后，利用基于语义的摘要信息和基于使用情况的统计信息对结果过滤。

应用层　应用层旨在从工具实际运行的角度，对 IMChecker 进行封装和优化。一方面，应用层实现多入口选择算法和调度引擎。通过该模块，具体执行每个入口的分析。另一方面，对于检测后的缺陷，生成相应的缺陷报告，包括：缺陷产生的路径信息、违反的约束和 IMSpec 与自然语言两种形式的解释信息。特别地，IMChecker 基于使用情况对缺陷检测结果进行排序。

IMChecker 已经集成在 Tsmart 工具集中[147]，工具使用的具体细节将在第4.3节中进行介绍。

3.4.2　实验准备

测试集　由于目前并没有针对 C 程序接口误用缺陷检测的测试集合，因此本章基于美国国家标准技术研究所 (NIST) 整理的 Juliet Test Suite 测试集[99]，对 IMChecker 的分析能力进行评估。该测试集合包含超过 6 万个 C/C++ 用例，涵盖 118 个不同的 CWE 缺陷类型，是目前最为广泛使用的测试集。本章根据接口误用缺陷模式共选取 13 个具体的分类。Juliet 测试集合在每个分类中给出大量重复模式、不同产生原因的测试用例。因此，本章对其中重复的用例进行过滤，保留由于接口误用导致错误的测试用例。同时，对每个分类中的缺陷实例进行过滤和修正，以确保每个测试用例至少含有一个误用和一个修复后的版本。如表3.2所示，针对第2.3节中总结的三大类接口误用模式，本章共选取 2172 个测试用例。为方便研究人员和开发者理解缺陷模式和评估检测工具，本章将这些修订后的测试用例集成到 C 程序接口误用缺陷数据集 APIMU4C 中，并公开在 Github 中①。

比较对象　首先，为检测过滤机制的效果，本章将未添加过滤机制的方法记作为 IMChecker--。通过 IMChecker 与 IMChecker--在测试集上的分析结果评估过滤算法

① https://github.com/imchecker/compsac19/tree/master/APIMU4C

表 3.2　评估测试集组成

缺陷类别	CWEID	缺陷数目	缺陷描述
IPU	121	36	栈内存越界读取
	122	36	堆内存越界读取
	131	42	不正确的计算缓冲区大小
	476	36	空指针引用
	590	360	释放非堆内存
	5 个 CWE 分类，510 个测试用例		
IEH	252	270	未检查错误状态代码
	253	270	错误状态代码检查不正确
	390	72	未进行异常处理
	3 个 CWE 分类，612 个测试用例		
ICC	401	474	内存泄漏
	404	24	不正确的资源释放
	415	120	重复释放
	690	384	未对因果调用中的上下文关系检测
	775	48	未释放文件资源
	5 个 CWE 分类，1050 个测试用例		

的有效性。此外，本章选取广泛使用的代表性开源工具进行检测能力的比较。在对工具描述文档阅读、学术论文研究和工具预实验后，本章共选取如下三个工具：

- APISan[94]（主分支-2018-0601）：该工具是一个开源的学术工具，针对接口误用缺陷设计。通过数据挖掘技术与程序分析技术结合，利用轻量级语义分析方法获得接口使用上下文信息，并基于统计方法推理接口使用约束。工具基于过程内分析方法，考虑函数返回值、参数关系和因果调用关系，并将规约推理的结果用于缺陷检测。不过，推理出的中间结果（接口使用的约束）没有输出，也没有提供扩展规约推理方法或者接收约束的接口。
- Cppcheck[78]（版本 1.83）：Cppcheck 是针对未定义和危险程序结构的缺陷检测工具。该工具开源代码，并支持多种缺陷类型。将代码转化为字节流（token）后，基于上下文敏感（context-sensitive）和流敏感（flow-sensitive）的过程间分析方法，对缺陷进行模式匹配。Cppcheck 在检测算法中集成 C 标准库的接口。同时，为支持用户自定义的接口，提供一套规约描述方法①，以利用实现过的算法对相同的缺陷模式进行检测。例如参数空指针，内存泄漏等等。
- Clang-SA[77]（版本 RELEASE_600）：该工具是 LLVM 开源编译器的组成部分。通过符号执行技术，推理程序语义。该工具采用上下文敏感（context-

① http://cppcheck.sourceforge.net/manual.pdf, chapter10, page 20.

sensitive）和流敏感（flow-sensitive）的过程间分析方法，并实现大量的检测插件，以应对不同的缺陷类型[①]。该工具可以通过开发新的检测插件从而支持用户自定义的接口。然而，需要理解内部数据结构表达和分析流程，开发难度较大。

一方面，这三个工具提供良好的缺陷报告接口，支持多种接口缺陷类型。另一方面，这三个工具基于不同的分析技术（数据挖掘、静态分析-提供规约描述和静态分析-硬编码）。同时，三个工具在预实验中的检测能力稳定，与文档论文中的描述能力一致。

评测指标　在测试集上，本章将采用如下两个指标对工具进行评估：

- 召回率（Recall）：$R = \frac{\text{结果报告中真实的缺陷数}}{\text{总缺陷数}}$。召回率是基于具体实现情况下，工具的检测能力，即工具能够有效地检测多少缺陷。缺陷检测领域普遍认为，一个工具在测试集合上的召回率越高，那么其实际应用中也会有更好的检测效果。
- 精度（Precision）：$P = \frac{\text{结果报告中真实的缺陷数}}{\text{工具报告的缺陷总数}}$。研究表明，阻碍实际用户使用静态分析工具的原因之一是现有工具产生大量的误报，即精度太低[34]。因此，检测工具的精度是重要的指标之一。

运行环境　本章在一台装有 64 位 Ubuntu 16.04 的台式机上进行实验，该机器配有 Intel(R) Core(R) i5-3470@3.20GHz-4 核心 CPU 和 32GB 内存。本章关注接口缺陷检测的能力，因此不设置最长分析时间。IMChecker 中 $\alpha = 2$，最小置信度 $\mathcal{H} = 1$。

表 3.3　IMChecker 方法评测结果

类别	个数	IMChecker				IMChecker- -			
		报告数	TP[①]	P(%)	R(%)	报告数	TP	P(%)	R(%)
IPU	510	490	423	86.33	82.94	745	469	62.95	91.96
IEH	612	580	506	87.24	82.68	677	526	77.70	85.94
ICC	1050	1012	878	86.76	83.62	1320	898	68.03	85.52
总计	2172	2082	1807	86.79	83.20	2742	1893	69.04	87.15

① TP：真实缺陷数目

3.4.3　评测结果

IMChecker 评测结果　评测结果如表3.3中所示，其中报告数为工具缺陷的总报告数，TP 为报告中真实存在的缺陷数目，P 为精度，R 为召回率。表中 3-6 列为 IMChecker 的评估结果，7-10 列为 IMChecker--的评测结果（即不包括过滤机制的结果）。

评测结果表明，在所有的测试用例中，IMChecker--共产生 2742 个缺陷报告，其中 1893 个为真实缺陷，849 个误报，平均精度为 69.04%，平均召回率为 87.15%。在对缺陷报告深入分析后，导致 IMChecker--分析结果不准确的主要原因是跨函数的接口使用。如第3.3节中所示，为能够支持大规模代码有效分析，IMChecker 的检测算法基于过程内分析进行。因此，当接口使用跨函数时，则会导致分析精度下降。

本章通过 `malloc/free` 例子进行分析。在 C 的标准库中，两者组合进行堆内存的申请和释放。其使用约束是，当前者的返回值不为 NULL 时，需要调用后者进行释放。然而 IMChecker--基于过程内的分析方式，对于跨越函数调用的接口，分析精度不足。一方面，语义信息的不足会产生误报，即将正确的用法报告为缺陷。例如下面代码所示，

```
1 void bar(int* p){
2    // do
3    free(p);
4 }
5 void foo(){
6    int* p = (int*) malloc(100*sizeof(int));
7    if (p == NULL) return;
8    bar(p);
9 }
```

IMChecker--在第 6 行发现调用 `malloc` 函数，在第 7 行对返回值进行检测，需要调用 `free` 函数进行内存释放。然而该释放操作在第 3 行的 `bar` 函数内进行，因此 IMChecker--并没有捕获这样的信息，导致将该用例判断为缺陷，产生一个误报。另一方面，同样的语义信息不足则可能导致漏报，即没有检测到缺陷。

```
1 void bar(int* p){
2    // do
3    free(p);
4 }
5 void foo(){
6    int* p = (int*) malloc(100*sizeof(int));
7    if (p == NULL) return;
8    bar(p);
```

① http://clang-analyzer.llvm.org/available checks.html

```
9     free(p);
10 }
```

例如上述代码所示，IMChecker--在第 9 行检测到 `free` 函数。同时该函数的参数与 `malloc` 的返回值指向同一块内存。因此，IMChecker--会将该用例判断为正确的使用。然而，在 `bar` 函数内，已经对该指针所指向的内存释放。所以这是一个重复释放内存错误（CWE-590），即 IMChecker--产生一个漏报。

因此，需要消除跨函数语义带来的影响，解决上述问题以提高检测的精度。如第3.3节介绍，IMChecker 引入基于语义和基于使用情况的过滤机制。在结合过滤算法后，IMChecker 共产生 2082 个缺陷报告，其中 1807 个为真实缺陷。IMChecker 的平均精度为 86.79%，平均召回率为 83.20%。相比较于 IMChecker--，在有限的召回率损失下（3.95%），IMChecker 检测精度有显著提高（17.75%）。损失的召回率由于过滤机制中，对于参数和返回值的处理造成。在实际项目中，本章发现大量的内存申请结果赋值给参数或者返回值并在外界进行检测和维护。因此，IMChecker 将这种情况下的缺陷过滤能够有效减少误报。然而如果外界缺少必要的操作，则会产生漏报。同时，虽然引入过滤机制，IMChecker 依旧存在误报和漏报。目前，IMChecker 基于语义的过滤机制只关注一层函数调用的摘要信息。因此，当调用信息超过两层时，依旧无法准确检测。例如如下代码片段，

```
1 void bar2(int* p){
2    // do
3    free(p);
4 }
5 void bar1(int* p){
6    // do
7    bar2(p);
8 }
9 void foo(){...}
```

由于释放函数跨越两层调用，因此无法被过滤，IMChecker 将产生一个误报。

此外其他一些原因同样导致 IMChecker 分析不够精确，包括：

- 复杂的数据依赖关系。目前 IMChecker 基于 CPA 算法实现，通过 AccessPath 的方式记录指向关系，并维护整数信息。然而，当指针存在偏移操作时，如果偏移的位置不为常量，那么 IMChecker 就会认为这个指针指向该内存对象所有的区间。另一方面，由于目前只维护常量整数，当整数为变量时，IMChecker 会认为该值可能为任何值。例如，`memcpy(d,s,n)` 拷贝内存内容，拷贝的长度 n 应小于目标 d 的大小。然而当程序中，无法对 n 和 d 的关系进行常量推理时，IMChecker 则会认为是错误，从而产生误报。
- 函数指针。C 程序提供函数指针的语言特性。然而在静态分析阶段，无法明确

获取该指针的指向关系。IMChecker 在分析的过程中会忽略这部分内容，导致误报和漏报。

- 循环。循环问题在程序分析中是至今无法良好解决的难题。特别地，如第3.3节缺陷实例调研结果显示，接口误用缺陷很少发生在循环中。因此，IMChecker 只展开一次循环。对于如下代码中的重复释放问题，IMChecker 则会产生漏报。

```
1  char * data;
2  data = NULL;
3  for(i = 0; i < 1; i++){
4     data = (char *)malloc(100*sizeof(char));
5     strcpy(data, "A String");
6  }
7  for(k = 0; k < 2; k++){
8     free(data);
9  }
```

针对于上述总结的不足，一个可行的解决方案是引入更加精确的分析方法。例如，增加跨函数分析、增加循环展开、增加函数指针的映射关系、利用更精确的值分析方法等等。然而这些精确的语义计算会极大地影响分析效率。因此，如何平衡分析效率与精度需要更多的尝试。这部分内容将在第5章中进行描述。

IMChecker 与其他工具对比结果　表3.4中对四个工具的评测结果进行总结。其中 Cppcheck 的检测精度最高，达到 89.95%，其次是 IMChecker 为 86.79% 和 Clang-SA 为 76.75%，APISan 的准确率最低，为 62.66%。在检测能力方面，IMChecker 的召回率最高，为 83.20%，其他三个工具的检测能力则相对较低，在 30-35% 之间。

基于推理方式的 APISan 通过推理的方式学习接口使用的约束，并根据语义的差异性对缺陷进行检测。与基于数据挖掘技术的检测方法一致，工具的检测能力由预先定义的缺陷模式决定。APISan 工具封装 $A \rightarrow B$ 的使用约束，即 A 调用后

表 3.4　IMChecker 与对比工具评测结果

类别	个数	APISan		Cppcheck		Clang-SA		IMChecker	
		报告数	TP	报告数	TP	报告数	TP	报告数	TP
IPU	510	154	48	145	127	127	105	490	423
IEH	612	446	173	298	270	0	0	580	506
ICC	1050	447	435	373	337	746	565	1012	878
总计	2172	1047	656	816	734	873	670	2082	1807
平均-P%		62.66		89.95		76.75		86.79	
平均-R%		30.20		33.79		30.85		83.20	

需要调用 B 接口。然而，并没有对 $A \rightarrow \neg B$ 模式进行定义。因此该工具忽略所有的重复释放（double free）用例，占 ICC 分类的 11.43%。此外，APISan 的学习结果无法保存和跨项目使用。因此，只能针对于单个测试用例或者项目进行分析。当缺少足够的数据对接口使用约束条件进行学习时，APISan 将无法获得正确的使用约束。这导致 APISan 无法支持 11.11% 的缺陷用例。另一方面，基于挖掘技术的方法多基于程序的语法形式分析，无法支持隐含的语义信息。例如，APISan 能够学习显式的语义信息，包括条件语句中的条件判断、函数调用等等。然而，无法推理出隐式的语义约束。例如，不包含显式检查的内存读写接口中参数的关系。另外，有一些语义信息无法通过 C 的语法进行描述，则必然会被忽略。例如，释放非堆内存错误，程序需要对 `free` 的参数所指向的内存区域进行深入的语义分析。

基于程序分析方式的 Clang-SA *和* Cppcheck 工具将接口使用约束编码在独立的检测插件中，在程序分析的过程中，对缺陷直接检测。Clang-SA 提供各种针对于不同缺陷类型的检测插件。例如，security.insecureAPI.UncheckedReturn 插件用来检测忽略对函数返回值检测的接口误用缺陷。然而，并没有一个插件能够支持测试用例中 IEH 缺陷，导致 Clang-SA 漏报所有 IEH 的缺陷实例。

项目特定的接口经常会和 C 标准库中的接口具有相同的使用约束。例如，用户自定义的 `*_new` 接口，与 `malloc` 行为一致，对资源进行管理。在资源生命周期结束后，需要进行释放操作。没有提供可扩展的静态分析工具则无法对这类接口进行分析。因此，Clang-SA 难以支持所有的缺陷用例。Cppcheck 提供面向特定项目的规约描述方法，以支持用户自定义的接口。然而，该语言的描述能力不足，检测能力提升有限。例如，Cppcheck 只支持对常量的返回值进行定义，且不能够描述形如 $a - [cond] - b$ 带有上下文语义约束的函数调用。

本章从表3.4中发现，相对于高检测精度，Clang-SA 和 Cppcheck 的召回率非常低。在对两个工具的原理和测试结果分析后，本章发现两个工具采取一种保守策略，即为提高检测精度增加工具的友好性，只对具有高可能性的缺陷进行报告。因此，相同的缺陷模式在复杂的代码结构中将会被忽视。

总结：对于三个比较的工具，在深入分析实验结果后本章发现，与 IMChecker 相比这些工具对语义分析不足，导致大量的漏报。具体体现在：（1）缺少路径敏感度分析（path-sensitive）。对路径的可达性分析在静态分析中具有重要意义。例如，在 `malloc` 内存申请成功后，当某条异常处理路径上没有释放内存时，将会产生一个内存泄漏错误。然而，当申请失败时，则不需要释放。因此，分析需要对路径信息进行维护和记录，而不是简单的通过调用次数进行缺陷检测。（2）缺少过程间分析（inter-procedural）。当接口使用跨越函数调用时，需要基于过程间信息来进

行程序分析。例如，`malloc/free` 在两个函数中。

综上所述，在测试集上 IMChecker 取得 86.79% 的检测精度和 83.20% 的召回率。检测能力优于比较的三个知名工具。其中，相比较于 Clang-SA 和 APISan，具有更高的检测精度和检测能力。在精度基本一致的情况下（86.79% 与 89.95%），比 Cppcheck 多检测（49.41%）的缺陷。

3.5　本章小结

本章提出基于约束描述的规模化 C 程序接口误用缺陷检测方法 IMChecker。IMChecker 利用 IMSpec 语言对接口使用约束进行描述，以支持用户自己定义的接口以及多种缺陷模式。同时，IMChecker 采用多入口分析策略，将复杂的程序分析问题分而治之，高效率分析的同时实现局部的精确分析。对于多入口分析策略引入的精度损失而导致的误报，IMChecker 通过基于上下文语义的摘要信息和基于使用情况的统计信息对结果进行过滤，以提高检测精度。在 Juliet Test Suite 公开测试集上 13 个接口缺陷相关分类的评测结果显示，IMChecker 方法误报率为 13.21%，漏报率为 16.08%，检测能力领先于主流的开源检测工具。

第 4 章　接口误用缺陷检测工具集与应用

在代码质量保障方法中，规约描述语言能够有效地描述接口使用的约束条件，高效的检测算法能够对大规模程序进行缺陷检测。然而，这些已有成果离不开面向实际应用场景工具集的支持。直观地说，工具决定语言描述、缺陷检测等成果在实际中的应用效果。本章将第2章和第3章的研究成果应用于实际项目中，并将结果总结在本章中。具体来说，本章包含两部分内容：C 程序接口误用缺陷数据集 APIMU4C 和可视化支持的 C 程序接口误用缺陷检测工具集 Tsmart-IMChecker。APIMU4C 包含对开源项目调研中总结的接口误用缺陷案例库，以及针对 C 程序的接口误用测试数据集。Tsmart-IMChecker 工具集包含三个子工具，以帮助使用者方便高效地撰写 IMSpec 规约、执行分析引擎并基于差异性对比的方式分析检测结果。此外，本章将 Tsmart-IMChecker 工具集应用于实际开源项目中，并对应用结果进行总结。从全文的研究体系上看，本章的工作旨在将接口使用约束描述语言 IMSpec 和规模化检测方法 IMChecker 应用于实际项目中，是本文研究工作的应用结果。

4.1　引言

过去的十几年内，研究人员与开发人员面向 C 程序接口误用缺陷检测投入大量的时间和努力进行算法设计和工具实现。例如，微软公司接口缺陷检测项目 SLAM，被广泛使用的开源静态分析工具 Cppcheck 和 Clang Static Analyzer 等等。特别地，为给使用者提供更好的用户体验，这些工具都提供良好的工具接口，即通过命令行的方式直接调用工具进行检测或者提供可视化支撑的 GUI 工具。然而，针对当下的开发环境和程序特点，以上方法存在若干不足。为弥补现有工作的不足，本文在第2章和第3章分别提出基于缺陷模式的接口使用约束领域特定语言 IMSpec，以及基于约束描述的规模化接口误用缺陷检测方法 IMChecker。本章将这两部分研究内容进行整理和封装，应用于实际开源项目中，并对应用结果进行总结。

首先，为帮助研究人员和开发者更好地理解接口误用缺陷，本章将缺陷调研以及算法评估中的原始数据集进行整理和封装，形成 C 程序接口误用缺陷数据集 APIMU4C（API-misuse for C）。APIMU4C 包含三个模块：（1）面向 Git 版本控制库的修改记录挖掘工具 Gitgrabber，（2）开源项目接口误用案例库，（3）基于公开数据集合以及实际项目案例的接口误用测试数据集。APIMU4C 包含本文中所有的

原始数据集，同时研究人员和开发者可以利用 Gitgrabber 对其他领域的缺陷修复进行挖掘和提取。此外，接口误用测试数据集能够帮助研究人员和开发者对现有工具进行评估，从而设计更具针对性的检测算法、选择适用的工具进行特定种类的缺陷检测。

为提高工具的实用性减轻使用者的负担，现有工具多提供良好的用户接口。这些工具面向的领域和对象不同，具有各自的特点。针对 IMSpec 语言和 IMChecker 检测方法，本章设计并开发图形化支撑的工具集 Tsmart-IMChecker，以提供更好的用户体验。该工具集共包含三个主要模块：（1）图形化规约撰写工具 IMSpec-writer，通过点击、选择和填入必要语义信息的方式，辅助开发者撰写接口使用约束；（2）静态分析引擎 IMChecker，通过对 IMChecker 方法的实现和封装，帮助开发者通过命令行的方式直接调用分析引擎并生成对应的缺陷分析报告；（3）图形化结果展示工具 IMDisplayer，通过差异性结果展示的方式帮助使用者快速准确地定位、理解缺陷。

本章将 Tsmart-IMChecker 应用于实际开源项目中，以评估工具的实用性。本章将工具集应用于 Linux 内核、OpenSSL 安全库和 Ubuntu 的应用软件的最新稳定版本中，共发现 112 个新的缺陷。本章将缺陷报告进行整理，共提交 75 个给相应的开发者进行确认。目前，61 个缺陷报告已经被开发者接收，其中 32 个已经在主分支中修复。同时，本章对实际应用中遇到的困难、应用发现进行总结，从而帮助研究人员和开发者设计更好的检测工具。

本章其余部分组织结构如下：4.2节对 APIMU4C 数据集进行介绍，4.3节介绍 Tsmart-IMChecker 工具总体架构和各子工具，4.4节给出工具集在开源项目上的应用结果，最后在4.5节总结本章工作。

4.2　C 程序接口误用数据集

近年来，软件缺陷获得研究人员的广泛关注。特别地，为有效地比较缺陷检测工具的性能，研究人员设计、整理面向不同领域和不同目标的数据集。例如，BugBench[148] 是针对 C 程序的缺陷评估集合，共包含 17 个开源项目中的缺陷实例，其中 13 个与内存、并发和程序特定语义相关。Defects4J[149] 是面向 Java 程序的缺陷集合，共包含来自 5 个开源项目的 357 个缺陷实例。在 2016 年，Amann 等整理 MUBENCH[6] 数据集。该数据集包含 89 个 Java 语言实际项目中的接口误用缺陷。为能够帮助研究人员和开发者更深入地理解接口误用缺陷，本章对全文的原始数据以及获得方法进行整理、修改和工具封装，形成 C 程序接口误用缺陷数据集 APIMU4C。该数据集包含三个部分：面向 Git 版本控制库的修改记录挖掘工

具 Gitgrabber、开源项目接口误用案例库、基于公开数据集合以及实际项目案例的接口误用测试数据集。本节将对每一个部分进行详细介绍。

Gitgrabber　对实际项目中的缺陷实例进行分析有利于理解缺陷的本质，从而帮助定义规约描述语言、设计检测算法。随着软件复杂度的增加和软件规模的提升，现代软件难以独立完成。同时，随着网络的发展越来越多的开发团队进行远程办公。在众多的代码维护方式中，Git 版本控制软件是最为广泛使用和最受欢迎的方式。如图2.1所示，开发者在使用 Git 版本控制软件时，对于每一次修改记录都会提供修改的描述报告。同时，Git 会自动计算修改记录中代码的差异并生成差异报告。因此，本章设计并实现面向 Git 版本控制软件的修改记录挖掘工具 Gitgrabber。该工具基于 Python 语言设计，通过自然语言处理的方式对修改记录进行提取。Gitgrabber 以用户提供的配置文件作为输入，以标记挖掘项目的分支、关键词、时间区间、个数、文件行数等等。Gitgrabber 提供多层挖掘服务，即在第一层挖掘结果的内容中进行二次检索。例如，通过缺陷修复的关键词进行缺陷修复相关修改记录挖掘，再基于某种特定类型的关键词挖掘特定领域的缺陷修复。Gitgrabber 使用方便，只需要提供基于结构化的配置文件并通过如下命令执行，

```
python main.py ——config config.yml
```

挖掘的结果包括：修改记录报告、差异性报告、修改前文件和修改后的文件。本文中缺陷模式调研的缺陷实例均来自于 Gitgrabber 的结果。

接口误用案例库　理解缺陷模式是设计缺陷检测工具的重要基础。对实际项目中的缺陷实例进行分析，有利于研究人员和开发者更深入地理解缺陷的本质。因此，本章将第2章第2.3节中分析的接口缺陷实例进行总结，形成接口误用案例库。目前该案例库共包含来自 6 个项目的 830 个接口误用缺陷实例，其中 Linux 内核 283 个、OpenSSL127 个、FFmpeg126 个、Curl134 个、FreeRDP119 个以及 Httpd41 个。本章对 830 个缺陷进行深入分析和核对，确保这些实例与接口误用相关。同时，针对于每一个缺陷实例，收录缺陷修复报告、差异性报告、修改前错误版本源文件以及修改后正确版本源文件。研究人员和开发者可以对这些案例进行研究，从而设计更具有针对性的接口使用约束描述方法和缺陷检测方法。

接口误用测试数据集　标准化的缺陷测试集合能够有效地评估缺陷检测工具。然而，目前并没有针对 C 程序接口误用缺陷的公开测试集合。因此，本章基于公开数据集以及实际项目中的缺陷实例，构造 C 程序接口误用测试数据集。该数据集构成如表4.1所示，包括 2172 个人工构造的单文件测试用例和 100 个实际项目用

表 4.1　C 程序接口误用缺陷测试集组成

缺陷位置	数目	类型	代码规模
单文件	2172	人工构造	100-200
OpenSSL	50	实际缺陷注入	454k
Curl	30	实际缺陷注入	113k
Httpd	20	实际缺陷注入	203k
总计	2272	包含所有调研中出现的缺陷类型	

例。单文件测试用例来源于广泛使用的公开测试集 Juliet Test Suite 和 ITC[150]。每个测试用例在 100-200 行之间，涵盖不同的 C 语言的语法结构。可以用来对检测工具的语言支持度和检测策略进行分析。实际项目缺陷用例来源于缺陷调研的结果，并注入到项目的最新版本中，可以用来评估检测工具在实际项目上的分析能力和效果。这 2272 个缺陷集合涵盖缺陷调研中常见的三种接口误用模式。C 程序接口误用测试数据集一方面能够帮助研究人员设计更具有针对性的检测算法；另一方面可以对现有工具进行评估，帮助研究人员分析现有工具不足。同时使用者可以通过检测结果对工具的能力进行评估，从而选择更适用的工具。

4.3　工具集组成

软件可信保障工具集 Tsmart (Trustworthy Software System Modelling And Verification Toolkit)，是清华大学软件学院软件系统与工程研究所研发的软件系统建模验证工具集。TsmartV3[147] 工具集在 2019 年 1 月发布第三版，是具有完全自主知识产权，面向软件代码安全可靠性的多维度保障体系。TsmartV3 具有代码合规性分析、缺陷静态检测、缺陷自动修复、并发性属性分析等功能。工具集可用于自动化地检测软件系统中难以发现和调试的重要缺陷，并给出相应的修复建议，从而提高软件系统的可靠性、提高软件开发测试效率、减小因为软件缺陷导致的损失。Tsmart-IMChecker 工具集是 TsmartV3 的重要组成部分，面向 C 程序接口误用缺陷设计。本节将对 Tsmart-IMChecker 的总体架构、模块组成进行介绍。

4.3.1　总体架构

Tsmart-IMChecker 是基于可视化支持的 C 程序接口误用缺陷检测工具集。如图4.1中所示，工具集包含三个主要模块：

1. IMSpec 规约撰写工具：尽管 IMSpec 语言提供一套轻量级、类似程序语言的结构化语法形式，手动撰写规约是一件耗时、容易出错的工作。因此本章设计可视化支撑的 IMSpec 规约撰写工具，帮助用户描述接口使用约束。

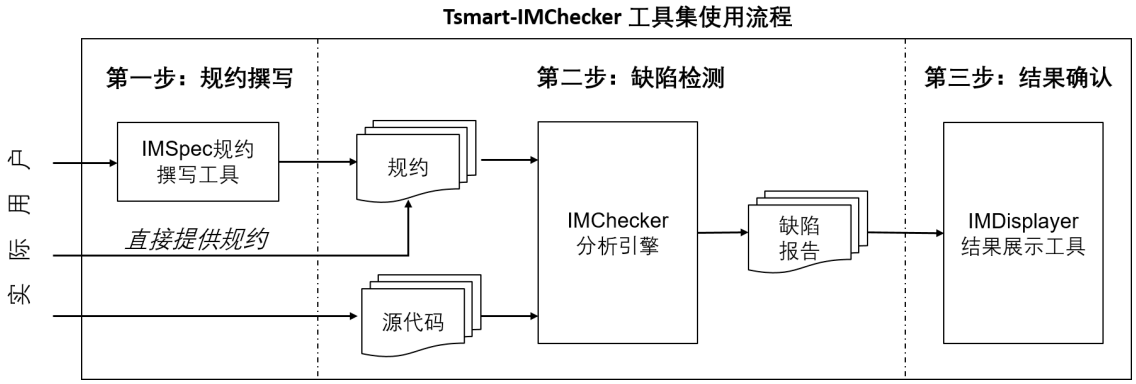

图 4.1　Tsmart-IMChecker 工具集使用流程

2. IMChecker 分析引擎：本章将规模化检测方法封装于 IMChecker 分析引擎中，并提供基于命令行的使用接口。用户可以直接调用该接口进行缺陷检测，也可以将分析引擎嵌入到实际开发环境中，作为第三方插件使用。
3. IMDisplayer 结果展示工具：TsmartV3 提供基于网页版的可视化结果展示工具。为帮助实际用户更好地理解接口误用缺陷发生原因，本章实现基于差异性的结果展示工具。通过在同一个项目中，正确使用实例与缺陷实例的对比，突出接口误用缺陷的原因和上下文差别。

用户需要三个步骤使用 Tsmart-IMChecker 工具进行缺陷检测：撰写规约、缺陷检测和结果确认。（1）首先，开发者通过 IMSpec 规约撰写工具，对目标接口的使用约束进行描述。如果开发者理解 IMSpec 的语法结构，也可以直接撰写文本格式的约束条件。（2）第二步，开发者通过调用缺陷检测引擎，以约束描述实例和源代码作为输入，进行分析。（3）最后，开发者可以直接通过生成的报告核对缺陷检测结果，也可以通过可视化 IMDisplayer 工具对检测结果进行确认。本节将在剩下的内容中对每个模块进行详细介绍。

4.3.2　规约撰写模块

接口使用约束已经被证明能够有效地应用于软件工程领域的不同任务中。特别地，这些约束能够帮助开发者理解如何正确使用 API，以及帮助测试人员对接口误用缺陷进行检测。然而人工撰写这些约束条件需要大量的时间与精力，同时容易在撰写中产生错误。为减轻使用者撰写 IMSpec 约束的负担、提高撰写规约语法的准确性，本章设计并实现图形化支撑的 IMSpec 规约撰写工具。

如图4.2中所示，IMSpec 规约撰写工具包含三个部分：

- *规约列表* 界面的最左侧是规约列表，用于展示使用者已经完成的 IMSpec 接口约束实例。用户可以在列表中直观地了解已经完成的约束描述。目前，

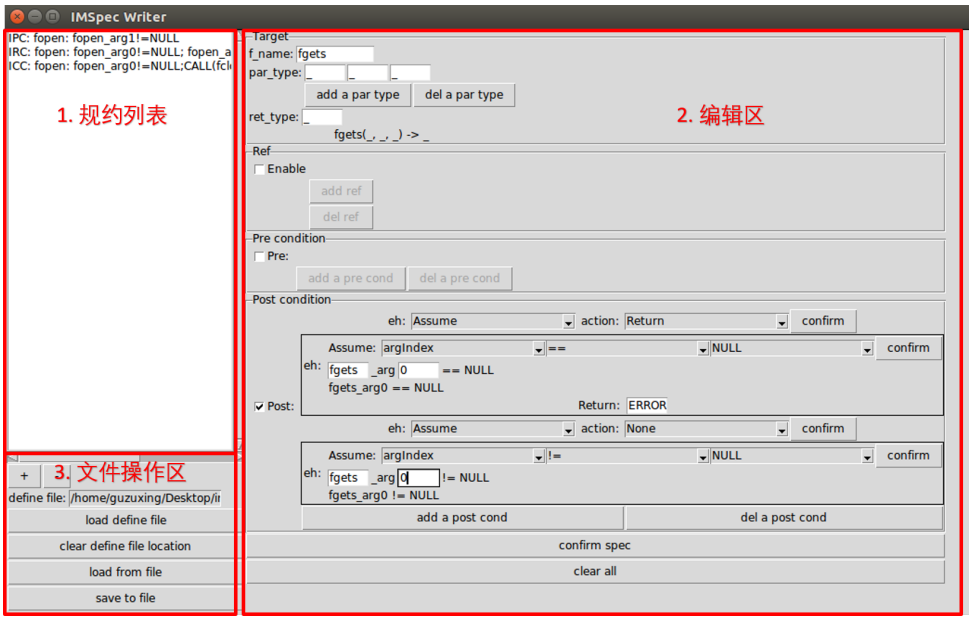

图 4.2　IMSpec 规约撰写工具截图

IMSpec 撰写工具将一个复杂规约分解成多个子约束，供缺陷分析引擎使用从而提高检测精度。因此，显示结果为分解后的约束条件。

- 编辑区 界面的右侧为规约编辑区，用于对 IMSpec 规约进行编辑。编辑区包括三大主要区域：目标对象区域、前置条件编写和后置条件编写区域。用户在确定目标接口的名称和参数以后，可以在下列的 Ref 区域选择性地填写与目标 API 相关的因果调用关系函数。例如：目标接口是 `fopen()` 函数，那么用户可以在 Ref 区域通过填写 `fclose()` 的函数定义进而强化因果调用关系。用户可以根据具体的接口使用约束，对前置条件和后置条件进行编辑。特别地，规约撰写工具在 IMSpec 语法基础之上，在界面显示中进行细微的调整，以提供更直观、准确的功能。在规约撰写中，用户可以使用宏定义以保持程序语义一致性和简洁性。在缺陷检测阶段提供宏定义的头文件即可。
- 文件操作区 界面的左下角是文件操作区域，包括加载宏定义文件、加载文件和导出三个功能。

用户通过命令行，执行 imspec_writer.py 文件运行该工具：

```
Ubuntu@ :python3 imspec_writer.py
```

其运行结果被保存在用户选择目录的 *.yaml 文件中，例如图2.15中 IMSpec 实例。此外，为帮助使用者维持项目特定的语义及特殊定义，IMSpec 规约撰写工具提供了 define.h 文件供使用者定义宏参数。例如，对于图2.15中 IMSpec 实例，用户可以在 define.h 中定义如下宏：

```
#define SUCCESS 1
```

```
#define FILEERR −1
#define IOERR −2
```

在后续解析 IMSpec 语言时，解析器会进行宏定义的展开从而与程序中的语义保持一致。

4.3.3　缺陷检测模块

研究人员和工具开发人员通过简单易用的用户接口提高工具的实用性，减轻使用者的负担。特别地，每个工具针对自己应用对象的不同，在工具接口上都有自己的设计目的和特点。本章对 IMChecker 方法实现和封装，以命令行的方式帮助开发者直接调用分析引擎，并生成对应的缺陷分析报告。

如图4.3所示，IMChecker 分析引擎在命令行中通过直接调用如下指令即可运行，

```
Ubuntu@ :python3 engine.py −−spec=XXX −−specDefine=XXX −−input=XXX
```

其中，参数的具体含义如下：

- spec: 描述目标接口使用约束的 IMSpec 规约文件。该文件可以由 IMSpec 规约撰写工具生成，也可以用户直接根据 IMSpec 语法进行撰写，其内容如图2.15中 IMSpec 实例所示。
- specDefine: 规约实例中宏参数的具体定义。在分析阶段，分析引擎会根据宏定义对 IMSpec 规约进行展开，并应用于缺陷检测，例如在上文中介绍的 define.h 文件。
- input: 待分析文件。目前，分析引擎在单独使用时接受单独可编译的 C 程序

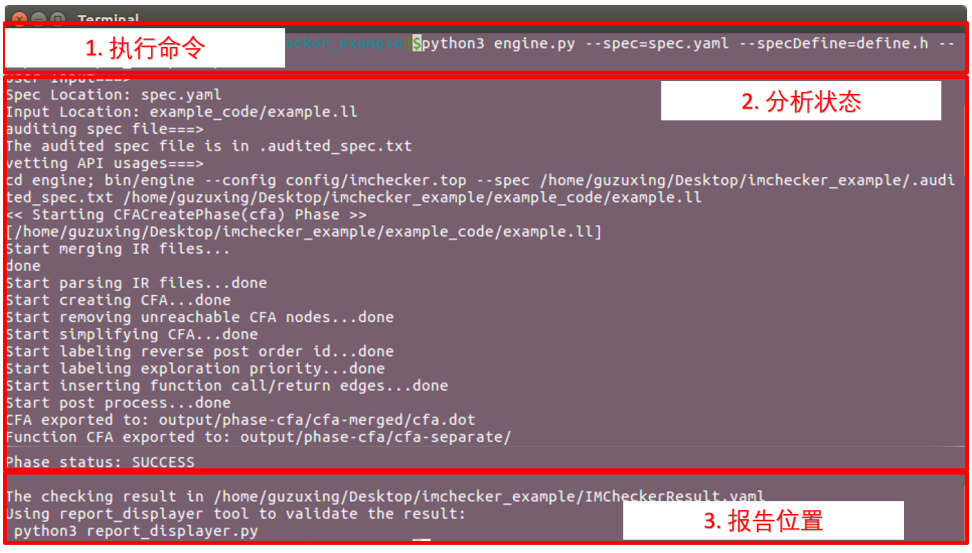

图 4.3　IMChecker 分析引擎运行时截图

源代码，以及预处理后的 LLVM-IR 中间表达。TsmartV3 提供 Build-capture 编译抓取工具，能够自动生成 LLVM-IR 表达。使用者亦可以通过 Clang 对项目进行编译，通过-S 指令生成 LLVM-IR 表达。如果用户通过 TsmartV3 工具对接口缺陷进行检测，则不需要考虑预处理操作，详细使用说明可以参考 TsmartV3 用户手册[147]。

在分析过程中，IMChecker 分析引擎将运行时状态输出到控制台中，以帮助开发者获得分析进展情况。同时，对于分析中遇到的问题，开发者可以获知原因并进行修复。如图4.3所示，分析引擎首先对 IMSpec 规约文件进行解析，接着进入预处理阶段创建 CFA、分析阶段，最后将分析结果输出到缺陷检测结果报告 IMCheckerResult.yaml 中。

IMChecker 分析引擎将结果输出到缺陷检测报告中，针对每一个目标 API 的一种缺陷模式，缺陷检测结果将汇总在一个“Bug”标签中。图4.4中给出图2.14中程序检测的一个缺陷实例，其中：

- Bug 标签为一个目标 API 某一种缺陷的起始标记符号。
- API 标签给出目标 API 的函数名。
- Type 标签给出缺陷的具体类型。在实际应用场景中，本章将缺陷的类型进行进一步扩展从而展示更具体的缺陷原因。例如，IPC 表示不正确的参数使用。具体参数定义可以参见 TsmartV3 用户手册。
- Spec 标签给出导致误用的具体约束条件。图中所描述的约束是第一个参数不可以为 NULL。
- Reason 标签给出基于自然语言的缺陷原因。目前针对于每一个缺陷类型，IMChecker 提供一段自然语言描述的缺陷原因。
- Error 标签给出具体缺陷发生的程序位置。
- Good 标签给出针对目标 API 正确使用的程序位置，即满足约束的函数调用

```
- Bug:
  API: fopen
  Type: IPC
  Spec: fopen==> fopen_arg1!=NULL
  Reason: Missing or incorrect validation of parameter
  Error:
    - IMChecker/tools/example_code/example.c:bad1:14
  Good:
    - IMChecker/tools/example_code/example.c:good1:52
```

图 4.4　IMChecker 分析结果示例

位置。

如果，该类型的错误存在多个实例或者存在多个正确的使用实例，则 Error 和 Good 标签拥有多个实例。使用者可以通过缺陷检测结果报告对缺陷进行核对，也可以通过 IMDisplayer 工具进行分析。此外，该缺陷报告也可以供其他工具的开发者利用。

4.3.4　结果展示模块

为更好地对缺陷结果进行展示，研究人员和开发者设计并实现各种各样的 GUI 界面。针对接口使用和接口误用缺陷的特点，本章设计并实现 IMDisplayer 结果展示工具，通过差异性结果对比的方式，帮助开发者深入理解 API 使用上下文的不同，并为开发者修复缺陷提供参考。

IMDisplayer 结果展示工具在命令行中通过直接调用如下指令即可运行，

```
Ubuntu@ :python3 report_displayer.py
```

如图4.5所示，IMDisplayer 工具包含五个部分：缺陷报告路径选择模块、缺陷信息展示模块、缺陷列表、参考列表以及差异性报告及上下文信息。用户在使用该工具时，首先要选择缺陷报告的路径信息。基于缺陷报告，IMDisplayer 工具会自动统计缺陷报告中的缺陷实例，并将误用 API 数目、缺陷实例数目展示在界面上。用户通过下拉菜单选择目标 API。如图中所示，目标接口为 `fgets()` 函数。在用户选择目标 API 后，IMDisplayer 工具会更新左侧的缺陷列表，以展示该接口所有误用的情况。此后，用户可以通过点击左侧的缺陷列表选择具体要审查的缺陷实例。在用户进行选择后，IMDisplayer 会更新缺陷实例的原因、违反的约束，以及右侧

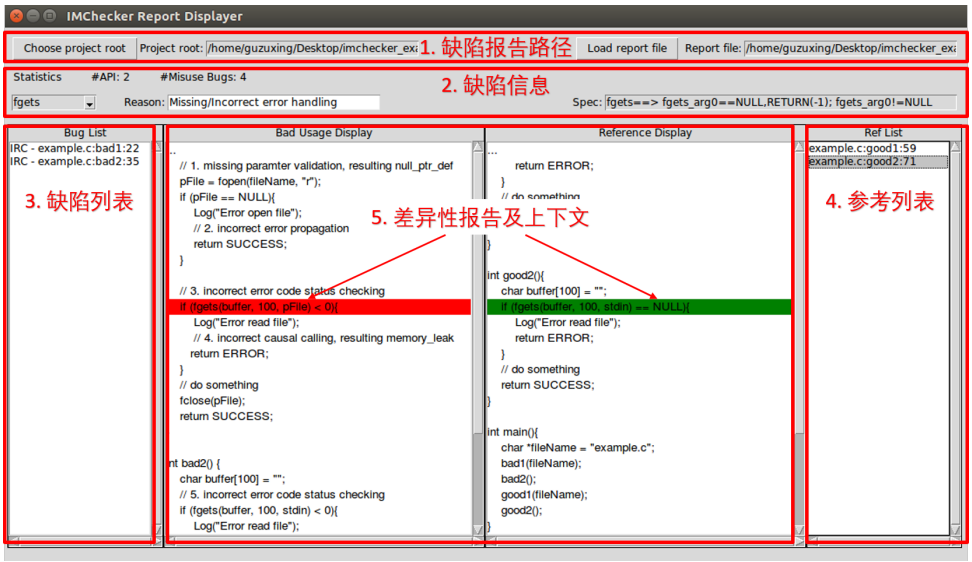

图 4.5　IMDisplayer 结果展示工具运行时截图

参考列表。用户可以通过对参考列表中的使用实例进行选择，在缺陷展示框中对缺陷和正确使用进行差异性对比。例如，图中红色为缺陷发生的位置，绿色为正确使用的位置，以及两者的上下文信息。

IMDisplayer 的设计动机来源于两点：（1）IMSpec 有效性评估结果显示，相对于自然语言 IMSpec 规约描述语言对接口使用约束描述更加有效，即开发者更容易理解使用约束的条件。因此，作者将缺陷具体违反的约束 IMSpec 形式展示在界面中，帮助开发者理解缺陷发生的原因。（2）在应用阶段，开发者提交缺陷报告给对应的开发者。最初只提供缺陷发生的原因和路径信息，开发者则表示如果能提供更多的语义信息将有助于理解缺陷的原因。因此，作者扩展缺陷分析引擎 IMChecker，并设计 IMDisplayer 工具。特别地，在分析过程中将正确使用的路径进行记录。在 IMDisplayer 中，通过差异性比对的方式展现缺陷和正确使用代码片段。不仅能够有效地帮助使用者理解缺陷的原因，同时能够提供缺陷修复的参考代码。（3）IMDisplayer 独立于分析引擎，通过解析缺陷报告中的标签来展示。因此，该工具能够扩展到其他应用场景。

4.4　案例应用

本文工作旨在对实际项目中 C 程序接口误用缺陷进行检测，以提高代码质量、应对现代软件开发的迫切需求。因此，本章将 Tsmar-IMChecker 工具应用于广泛使用的开源项目中评估本章研究的有效性。本小结将对应用对象和方法进行介绍、总结应用的结果、并讨论实际应用中的发现和不足，为研究人员和开发人员提供思路。

4.4.1　实验准备

应用对象　本章选取三个典型领域的开源项目作为应用对象，评估 Tsmart-IMChecker 工具在实际项目中的有效性，包括：操作系统 Linux 内核、广泛使用的第三方库 OpenSSL 安全库和使用第三方库的应用软件。其中，应用软件为 Ubuntu16.04 中使用 OpenSSL 安全库的应用软件。Ubuntu 系统提供多个版本的 OpenSSL 安全库的实现，本章采用 libssl1.0.0[①]作为应用对象。

应用软件的选择步骤如下。首先，本章通过 Ubuntu 系统提供的软件包依赖管理工具搜索所有使用 libssl1.0.0 的应用软件包，即通过如下命令，

```
Ubuntu@ : apt-cache rdepends libssl1.0.0
```

① https://packages.ubuntu.com/xenial/libssl1.0.0

`apt-cache rdepends` 命令[①] 能够查看一个软件包被哪些软件包所依赖。搜索的结果显示，超过 1200 应用软件包依赖于 libssl1.0.0。接着，对这些软件包本章在 Github 中搜索是否存在这些软件包的源代码。目的在于找到这些应用对象中哪些仍然活跃于开源社区中，从而提交的缺陷报告可以及时获得回复。对所有的软件包进行编译、分析需要大量的时间、人力和财力，无法完成。因此本章共选择 15 个广泛使用、不同应用领域、开发者持续维护的应用软件。为有效评估 Tsmart-IMChecker 的检测能力（是否能够发现未知的接口误用缺陷），所有应用对象选择截止至 2018 年 7 月 10-15 日的最新稳定版本，即 Linux 内核 4.18-rc4，OpenSSL-1.1.1-pre8，以及 Ubuntu 操作系统中应用软件：

- dma: 邮件服务，主分支，86 个关注（Star 数目）
- exim: 邮件服务，版本 4.91，393 个关注
- hexchat: 网络实时聊天，版本 2.14.1，1949 个关注
- httping: 针对 HTTP 请求的 Ping 服务，主分支，302 个关注
- ipmitool: 针对智能平台管理接口（Intelligent Platform Management Interface，IPMI）的控制系统，主分支，122 个关注
- open-vm-tools: VMware 软件管理，版本 10.3.0，956 个关注
- irssi: 通信软件，版本 1.1.1，1918 个关注
- keepalive: 负载均衡管理软件，版本 2.0.5，1717 个关注
- thc-ipv6: IPV6 攻击测试软件，主分支，442 个关注
- freeradius-server: 多协议服务器，主分支，943 个关注
- trafficserver: 网络代理服务器，版本 7.1.3，907 个关注
- tinc: VPN 后台服务，版本 1.1-pre16，854 个关注
- sslplit: SSL/TLS 攻击测试工具，主分支，1106 个关注
- rdesktop: 远程桌面服务，主分支，577 个关注
- proxytunnel: 网络代理服务，主分支，155 个关注

目标 API　目标 API 包括两个部分：（1）对于 Linux 内核和 OpenSSL，本章将第2.5节中撰写的 IMSpec 约束作为目标 API，旨在通过这些被误用过的 API 再次对这些项目进行检测，从而观察是否依旧存在误用情况。（2）对于 15 个 Ubuntu 中的应用软件，本章首先通过 GNU cflow[②]抽取这些应用软件的函数调用图。基于函数调用图和 OpenSSL 提供的用户手册，选择调用图中使用 OpenSSL 库中的接口。例如在 dma 项目中，通过 cflow 和 OpenSSL 用户手册比对的结果，本章发现该项

① http://manpages.ubuntu.com/manpages/bionic/man1/apt-rdepends.1.html
② http://www.gnu.org/software/cflow/

目使用 OpenSSL 中包含 `SSL_connect()` 等 17 个不同的 API。接着在 15 个项目中，本章将 API 的使用情况进行总结，选取至少使用过三次的接口。最终共选择 78 个不同的 API 作为检测目标。本章将这些目标 API 的 IMSpec 规约描述文件公开[①]，供研究人员和开发者使用。

实验环境　本章在一台装有 64 位 Ubuntu 16.04 的台式机上进行实验。该机器配有 Intel(R) Core(R) i5-3470@3.20GHz-4 核心 CPU 和 32GB 内存。Tsmart-IMChecker 工具需要通过 Clang 对项目进行预处理，抽取 LLVM-IR 的中间表达。然而，部分项目无法完全被 Clang 支持，例如 Linux 内核。同时，部分项目依赖于旧版本的编译环境或者其他编译库难以满足。针对于这些情况，本章通过人工编译的方式，对这些项目中使用目标 API 的文件进行单独编译，并将这些能够编译的文件作为分析目标提供给分析引擎。为控制分析的时间和输出结果，本章每次分析执行一个目标 API。所有的分析中，本章以 3 小时作为时间界限，即如果三小时无法完成，则将已有分析结果输出。实验中，以公式3-1中定义的置信度 $\mathcal{H} >= 2$ 进行缺陷检测，最小置信度 $\alpha = 3$。对于所有找到的新缺陷，本章进行缺陷理解和归纳，并选

表 4.2　Tsmart-IMChecker 实际项目应用结果

项目名称	缺陷报告总数	确认未修复	已经修复
Linux 内核	30	20	5
OpenSSL	17	5	12
dma	1	0	1
exim	2	0	2
hexchat	2	1	0
httping	1	1	0
ipmitool	1	1	0
open-vm-tools	2	0	2
irssi	2	1	0
keepalive	2	0	2
thc-ipv6	2	0	2
freeradius-server	2	0	2
trafficserver	3	0	0
tinc	2	0	2
sslplit	2	0	2
rdesktop	2	0	0
proxytunnel	2	0	0
总计	75	29	32

① https://github.com/tomgu1991/IMChecker/tree/master/imspec

择 $\mathcal{H}$ >= 3 的缺陷在对应的项目中提交缺陷报告或者提交修改申请（Pull Request）。

4.4.2　缺陷检测结果

如表4.2所示，Tsmart-IMChecker 工具针对于上述 17 个开源项目共检测到 112 个新的缺陷，并对其中 75 个缺陷实例（$\mathcal{H}$ >= 3）进行整理提交缺陷报告。表中第一列为项目的名称；在第二列中，对缺陷提交总数进行展示；第三列和第四列分别给出在提交的缺陷中，开发者已经确认但没有修复的个数，以及开发者确认并且修复的缺陷报告个数。在所有提交的接口误用缺陷报告中，Linux 内核有 30 个，25 个已经被开发者确认，其中 5 个已经在最新的代码中进行修复与集成。OpenSSL 共报告 17 个缺陷，全部被开发者确认，其中 12 个已经被修复并集成到主分支和发布版本中。Ubuntu 系统中的每个应用软件分别报告 1-3 个缺陷，每个项目具体数据如表中所示。综合来看，提交的 75 个缺陷报告中，61 个（81.33%）已经被开发者确认。在确认的报告中，32 个（42.67%）已经被成功修复。对于 14 个未被确认的缺陷报告，开发者并没有给予反馈。下文将对 Linux 内核、OpenSSL 和 Ubuntu 应用软件中的缺陷进行详细讨论。

Linux 内核　如表4.3中所示，在 Linux 内核中 Tsmart-IMChecker 工具共提交 30 个未被报告的接口缺陷。针对于每一个缺陷报告，本章首先在 Linux 的缺陷报告系统[①]中进行查询，是否已经有相关缺陷报告。如果没有，则对缺陷报告进行提交。在提交报告给开发者后，其中 25 个已经被开发者确认并收到开发者回复的邮件。至今，5 个已经被维护者接受。其中 2 个已经集成到 Linux 最新版当中，3 个在进行代码风格修改。

图4.6为作者将误用接口 `alloc_disk()` 的缺陷实例细节提交以及修复的记录截图。根据 Linux 中错误处理机制，接口 `alloc_disk()` 出错时，外层调用者需要返回**-ENOMEM** 来表示内存不足的错误信息。然而在 drivers/block/pktcdvd.c 文件的 `pkt_setup_dev()` 函数中 2718 行调用该接口后，在异常处理路径上没有对返回值进行修改，导致一个不正确的异常处理缺陷（IEH）。在对该缺陷进行仔细核对后，本章将缺陷提交到 Linux 缺陷报告系统中，编号为 200505。该缺陷报告在提交后，获得维护者的回复。在经过三轮的缺陷细节讨论过后，该缺陷被维护者确认，并修改和集成到 Linux 内核的主分支中。缺陷修复的细节如图4.7中所示，即在返回之前对返回值赋值为错误代码**-ENOMEM**。

① https://bugzilla.kernel.org

OpenSSL　如表4.4中所示，在 OpenSSL 中 Tsmart-IMChecker 工具共提交 17 个未被报告的接口缺陷。针对于每一个缺陷报告，本章首先在 Github 中 OpenSSL 代码库中提交 Issue。在提交报告给开发者后，所有的缺陷报告都获得开发者的回复。其中 12 个已经被维护者修复并集成在主分支内，5 个被标记为待评估（Assessed）。

表 4.3　Linux 内核版本 4.18rc4 缺陷检测结果

编号	缺陷编号	误用 API	文件位置 文件名：调用函数	缺陷种类	状态[①]
1	200489	kzalloc	tlb_uv.c: **init_per_cpu**	ICC	✓
2	200505	alloc_disk	pktcdvd.c: **pkt_setup_dev**	IEH	✓✓
3	200511	kzalloc	clk-pxa.c: **clk_pxa_cken_init**	ICC	✓
4	200519	devm_clk_get	hci_bcm.c: **bcm_get_resources**	IPU	✓
5	200521	kzalloc	ip22-gio.c: **ip22_check_gio**	ICC	✓
6	200533	nla_nest_start	ncsi-netlink.c: **ncsi*_nl**	IPU	✓✓
7	200535	nla_nest_start	conntrack.c: **ovs*_get**	IPU	✓✓
8	200537	nla_nest_start	datapath.c: **queue*_packet**	IPU	✓
9	200539	alloc_skb	chtls_cm.c: **chtls*_conn**	ICC	✓
10	200541	__send*alloc_skb	team.c: **team_*_get**	ICC	✓
11	200543	devm_kzalloc	gpio-tegra.c: **tegra_gpio_probe**	IEH	✓
12	200545	devm_kzalloc	core.c: **rsnd_probe**	IEH	✓
13	200547	devm_kzalloc	atmel*_output.c: **atmel_*_endpoint**	ICC	*P*
14	200549	pci*ext_capability	nic_main.c: **pci*_capability**	IPU	✓
15	200551	pci*ext_capability	dpc.c: **dpc_probe**	IPU	✓✓
16	200555	devm*init_i2c	hmc5843_i2c.c: **hmc5843*_probe**	IPU	✓
17	200557	devm_ioremap	pata_pxa.c: **pxa_ata_probe**	IEH	✓
18	200559	alloc_workqueue	fm10k_main.c: **fm10k*_module**	ICC	✓
19	200561	ida_pre_get	namespace.c: **mnt*_id**	IPU	✓✓
20	200563	wm831x*_read	clk-wm831x.c: **wm831x*prepared**	IEH	✓
21	200565	dma_mapping_error	qib_sdma.c: **qib*send**	IEH	✓
22	200567	get_zeroed_page	sysinfo.c: **sysinfo_show**	IEH	✓
23	200569	kzalloc	mach-mx27ads.c: **mx27ads*_init**	ICC	*P*
24	200571	kzalloc	board-v7.c: **i2c_quirk**	ICC	✓
25	200573	kzalloc	coherency.c: **armada_*_init**	ICC	✓
26	200575	kzalloc	octeon-irq.c: **octeon*_map**	ICC	✓
27	200577	kzalloc	gptu.c: **clkdev*_gptu**	ICC	*P*
28	200579	kzalloc	octeon-irq.c: **octeon*_map**	ICC	*P*
29	200581	kzalloc	sysctrl.c: **clkdev_add_pci**	ICC	*P*
30	200583	kzalloc	msi-xlp.c: **xlp*_irqs**	ICC	✓

① ✓✓ 为已经修复的缺陷，✓ 为确认的缺陷，*P* 为未确认的缺陷。

OpenSSL 与 Linux 内核的缺陷报告系统不同，其通过 Github 的 Issue 系统维护，因此缺陷的提交和修复能够获得及时的反馈。

图 4.6　Linux 接口误用缺陷 200505 提交和讨论记录

表 4.4　OpenSSL-1.1.1-pre8 缺陷检测结果

编号	缺陷编号	误用 API	文件位置 文件名：调用函数	缺陷种类	状态①
1	6567	RAND_bytes	speed.c: **RAND*_loop**	IEH	✓✓
2	6568	ASN1_INTEGER_get	tasn_utl.c: **asn1_do_adb**	IPU	✓
3	6569	ASN1_INTEGER_set	p12_init.c: **PK*init**	IPU	✓✓
4	6570	ASN1_object_size	asn1_gen.c: **generate_v3**	IEH	✓
5	6572	BN_set_word	t1_lib.c: **ssl*_dh**	IEH	✓✓
6	6573	HMAC_Init_ex	apps/speed.c: **HMAC_loop**	IEH	✓
7	6574	EVP_PKEY_get0_DH	statem_srvr.c: **tls*_dhe**	IPU	✓✓
8	6575	EC_KEY_generate_key	speed.c: **run_benchmark**	IEH	✓
9	6781	EC*new*_name	ec_ameth.c: **eckey_type2param**	ICC	✓✓
10	6789	ASN1_INTEGER_set	v3_tlsf.c: **v2i*FEATURE**	IEH	✓✓
11	6820	ASN1_INTEGER_to_BN	ts_lib.c: **TS*_bio**	IPU	✓✓
12	6822	BN_sub	rsa_ossl.c: **rsa*_encrypt**	IEH	✓✓
13	6973	EVP_MD_CTX_new	ocsp_srv.c: **OCSP*_sign**	IPU	✓✓
14	6977	ASN1_INTEGER_set	pk7_lib.c: **PKCS7*type**	IEH	✓✓
15	6982	OBJ_nid2obj	asn_moid.c: **do_create**	IEH	✓✓
16	6983	BN_sub	bn_x931p.c: **BN*_Xpq**	IEH	✓✓
17	7235	DH_set0_key	dh_lib.c: **DH*_key**	IEH	✓

① ✓✓ 为已经修复的缺陷，✓ 为确认的缺陷。

```
"summary":  "fix setting of 'ret' error return for a few cases",
"date": "2018-08-16",
"author": "Jens Axboe",
drivers/block/pktcdvd.c
=====================================================
@@ -2740,6 +2740,7 @@ static int pkt_setup_dev(dev_t dev, dev_t* pkt_dev)
pd->write_congestion_on = write_congestion_on;
pd->write_congestion_off = write_congestion_off;

+   ret = -ENOMEM;
    disk = alloc_disk(1);
    if (!disk)
        goto out_mem;
```

图 4.7　Linux 接口误用缺陷 200505 修复细节

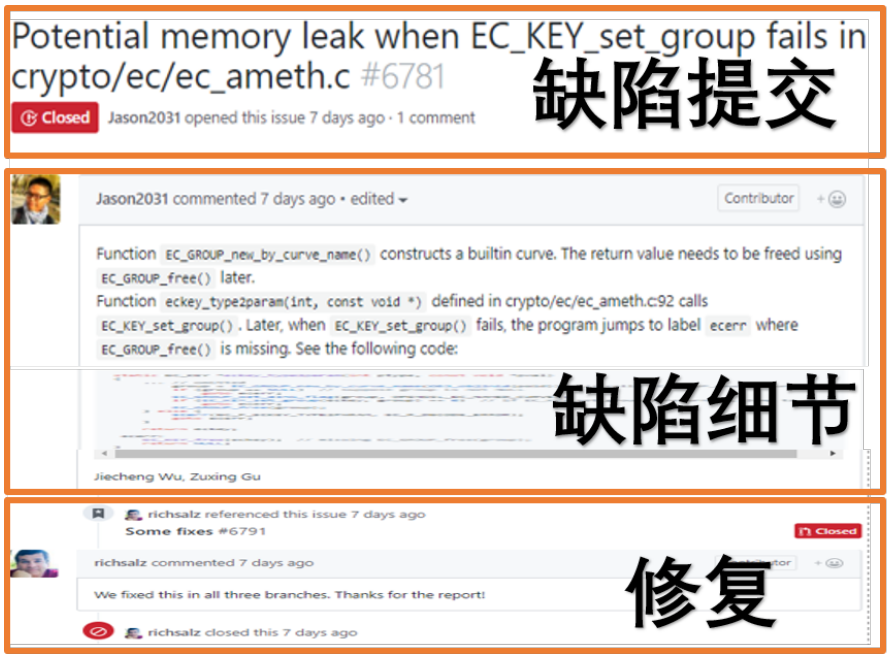

图 4.8　OpenSSL 缺陷 6781 提交和修复记录

图4.8是作者在 OpenSSL 项目检测到的一个内存泄漏缺陷，即不正确的因果调用关系。OpenSSL 提供接口 `EC_GROUP_new_by_curve_name()` 来创建 EC_GROUP 对象。该对象在生命周期结束后需要通过 `EC_GROUP_free()` 进行释放。然而在 ec_ameth.c 文件的 `eckey_type2param()` 的函数体内，在一条异常处理路径上开发者并没有进行释放导致一个内存泄漏错误。如图4.9中所示，开发者在 12 行进行内存对象的申请，然而在 16 行的 if 判断错误处理中直接通过 goto 语句跳转到 21 行继续执行。因此这条异常处理路径上，忽视了对于 12 行申请内存对象 `group` 的释放。在作者提交缺陷报告并说明出错路径后，开发者在 12 小时内完成修复。修复细节如图4.9所示。

OpenSSL 项目设计明确的错误代码机制，并在用户手册中提供基于自然语言

```
"summary":  "Avoid memory leak on failure path.Thanks to Jiecheng Wu, Zuxing Gu
    for the report.",
"date": "2018-07-26",
"author": "richsalz",
==========================================================
@@ -122,13 +122,12 @@ static TLS_FEATURE *v2i_TLS_FEATURE(const
    X509V3_EXT_METHOD *method,
+  EC_GROUP *group = NULL;
   [...]
   else if (ptype == V_ASN1_OBJECT) {
      const ASN1_OBJECT *poid = pval;
-     EC_GROUP *group;
      [...]
      group = EC_GROUP_new_by_curve_name(OBJ_obj2nid(poid));
      if (group == NULL)
         goto ecerr;
      EC_GROUP_set_asn1_flag(group, OPENSSL_EC_NAMED_CURVE);
      if (EC_KEY_set_group(eckey, group) == 0)
         goto ecerr;
      EC_GROUP_free(group);
   } else {
   [...]
ecerr:
   EC_KEY_free(eckey);
+  EC_GROUP_free(group);
   return NULL;
```

图 4.9　OpenSSL 缺陷 6781 修复细节

的描述。然而，项目的开发者却忽略大量的异常处理检测。例如，缺陷 6572、6789 和 6982，均由不正确的异常处理原因导致的软件缺陷。作者在提交缺陷报告后，开发者均进行相应的修复。

应用程序　OpenSSL 软件库是网络通信领域广泛使用的安全库，这些接口误用会极大地损害系统的可靠性。本章在 Ubuntu16.04 的应用软件中，对 OpenSSL 软件库接口的误用进行检查。检测结果显示，这些应用软件存在大量的接口误用缺陷。本章在对结果分析、核对和整理后，将 28 个检测结果提交到对应的 15 个项目中。如表4.5所示，28 个缺陷中 19 个被开发者接受，其中 15 个已经被成功修复。特别地，dma、exim、open-vm-tools、keepalive、thc-ipv6、FreeRADIUS、tinc 和 sslplit 八个项目，对缺陷报告进行及时的确认和修复。虽然其他项目对缺陷报告没有修复，但部分报告已经被开发者确认并标记为下一个版本需要处理的问题。其他报告则处于没有回复的阶段，即开发者并没有否定检测结果为误报。同时，相应的缺陷模式已经在其他的项目中获得认可。因此，本章认为这些缺陷具有极高的可信度。

图4.10是本章在 dma 项目检测到的一个不正确的异常处理错误。OpenSSL 的官方用户手册中指出，接口 `SSL_connect()` 用于初始化 TSL/SSL 服务器握手操作（handshake）。根据 TSL/SSL 协议本身，当握手失败时该函数的实现返回 0 代表链接被成功关闭，返回负数则代表错误发生在协议层、链接等等。因此，该接口的错误状态代码有两种情况，针对该函数的异常处理也需要考虑这两种情况。如

表 4.5　Ubuntu16.04 应用软件缺陷检测结果

项目名称	缺陷编号	误用 API	文件位置 文件名：调用函数	缺陷种类	状态[①]
dma	59	SSL_connect	crypto.c: **smtp*_crypto**	IEH	✓✓
exim	2316	X509_NAME_oneline	tls-openssl.c: **x509*_names**	IEH	✓✓
	2317	SSL_CTX*_list	tls-openssl.c: **tls*_cb**	IEH	✓✓
hexchat	2244	BN_set_word	dh1080.c: **dh1080_init**	IEH	✓
	2245	DH_set0_key	dh1080.c: **dh1080_compute_key**	IEH	*P*
httping	41	SSL_CTX_new	mssl.c: **initialize_ctx**	ICC	✓
ipmitool	37	MD2_Init	auth.c: **ipmi_auth_md2**	IEH	✓
open-vm-tools	291	SSL_*list	sslDirect.c: **SSL_NewContext**	IEH	✓✓
	292	X509*_current_cert	certverify.c: **VerifyCallback**	IPU	✓✓
irssi	943	SSL*_certificate	net*-openssl.c: **irssi*_handshake**	IPU	*P*
	944	BIO_read	certverify.c: **set*_info**	IEH	✓
keepalive	1003	SSL*_new	ssl.c: **build*_ctx**	ICC	✓✓
	1004	SSL_new	ssl.c: **ssl_connect**	ICC	✓✓
thc-ipv6	28	BN_new	thc-ipv6-lib.c: **thc_memstr**	ICC	✓✓
	29	BN_set_word	thc-ipv6-lib.c: **thc_memstr**	IEH	✓✓
FreeRADIUS	2309	BIO_new	session.c: **tls*_cert**	ICC	✓✓
	2310	i2a_ASN1_OBJECT	session.c: **tls*_cert**	IEH	✓✓
trafficserver	4292	SSL_CTX_new	http_load.c: **handle_connect**	ICC	*P*
	4293	SSL_new	http_load.c: **handle_connect**	ICC	*P*
	4294	SSL_write	http_load.c: **handle_connect**	IEH	*P*
tinc	205	BN_hex2bn	tincd.c: **keygen**	IPU	✓✓
	206	RAND_load_file	tincd.c: **main**	IEH	✓✓
sslsplit	224	SSL*_certificate	pxyconn.c: **pxy*_create**	IEH	✓✓
	225	SSL*_PrivateKey	pxyconn.c: **pxy*_create**	IEH	✓✓
rdesktop	280	BN_bin2bn	ssl.c: **rdssl*_encrypt**	IEH	*P*
	281	BN_mod_exp	ssl.c: **rdssl*_encrypt**	IEH	*P*
proxytunnel	36	SSL_connect	ptstream.c: **stream*_ssl**	IEH	*P*
	37	SSL_new	ptstream.c: **stream*_ssl**	ICC	*P*

① ✓✓ 为已经修复的缺陷，✓ 为确认的缺陷，*P* 为未确认的缺陷。

图中代码所示，在 dma 项目中开发者只对负数进行检查，而忽略错误状态代码为 0 的情况。因此，当 `SSL_connect()` 函数在执行中返回 0 时，该项目的异常处理机制将失效，程序的鲁棒性将会被破坏。dma 项目为邮件服务，因此该错误极大可能被攻击者利用从而产生漏洞。本章在提交缺陷报告后，开发者进行了及时修复。如图中所示，开发者对错误状态代码的判断进行完善，从而正确执行异常

图 4.10　dma 缺陷 59 提交和修复记录

处理机制。

4.4.3　应用经验总结

本章将 Tsmart-IMChecker 应用于实际开源项目中，对接口缺陷进行分析并将缺陷报告提交给相应的开发者。本章将缺陷检测、报告提交、开发者讨论和后续跟进工作中遇到的困难、发现和项目应用经验总结在下文中。主要包括如下几个方面：接口误用缺陷、现有文档、缺陷提交和开发者态度。

接口误用缺陷　同第2.3节调研结果相似，接口误用缺陷并不是个例，而是普遍存在于各种软件中。特别地，本章共找到 112 个新缺陷，并提交 75 个高置信度（$\mathcal{H}$ >= 3）的缺陷报告给实际开发者。提交的缺陷报告中，61 个（81.33%）被开发者接受，32 个（42.67%）已经被开发者修复。

在本章分析之前，Linux 内核和 OpenSSL 软件库作为广泛使用的基础软件，两者代码备受各种分析工具关注并检测。例如，商业工具 Coverity[138]、学术工具 Clang Static Analysis 和开源测试工具 AFL 等。然而，实验结果显示，依旧存在接口

误用缺陷。更严重的是这些缺陷的目标 API 均从已有的缺陷修改记录中获得，即这些缺陷已经在过去发生过。此外，分析结果显示被分析的 Ubuntu 应用软件中每个应用都至少误用一个 OpenSSL 的接口。在对缺陷模式、细节分析后，本章发现这些缺陷产生存在两方面的原因：(1) 软件库开发者间缺少缺陷信息分享机制。开发者经常会忘记接口使用的约束。特别地，当新的代码中存在和过去缺陷模式一样的接口使用时，并没有一个能够自动检测的方法，对开发者进行提示。(2) 客户端开发者缺少足够的领域知识，导致误用库函数，即客户端开发者在使用 OpenSSL 软件库时，没有深入理解或者参考文档，导致接口使用错误。

总结来说，开发者在开发过程中难免会产生接口误用，无论是无意忘记约束，还是本身缺少理解。因此，规模化、高效的分析工具对接口误用缺陷检测具有重要的实际意义。特别地，Tsmart-IMChecker 基于规约描述语言进行检测，集成大量的领域知识并容易扩展。同时可以渐进式分析，即可以只应用于新修改的代码中。

现有文档　虽然 OpenSSL 软件库为使用者提供格式优良的自然语言描述的接口使用约束和解释，然而在本章分析的 15 个应用软件中，依旧存在大量的接口误用缺陷。对缺陷报告和文档进行分析后，现有的文档存在如下几个不足：

- 文档自身存在错误。在 OpenSSL 的缺陷检测中，本章发现 OpenSSL 提供的官方文档中存在错误。在进行代码具体使用情况和文档描述的核对后，本章提交文档和接口缺陷报告（Issue-6569）。开发者在收到缺陷报告后，1 天内对文档进行修复，如图4.11中所示。
- 文档利用率不足。一方面，软件库的代码实现中本身存在对库接口的误用；另一方面，客户端开发者对文档参考不足，误用这些接口。在和开发者讨论接口误用原因时，超过一半的开发者表示在遇到使用问题时更喜欢去 Google

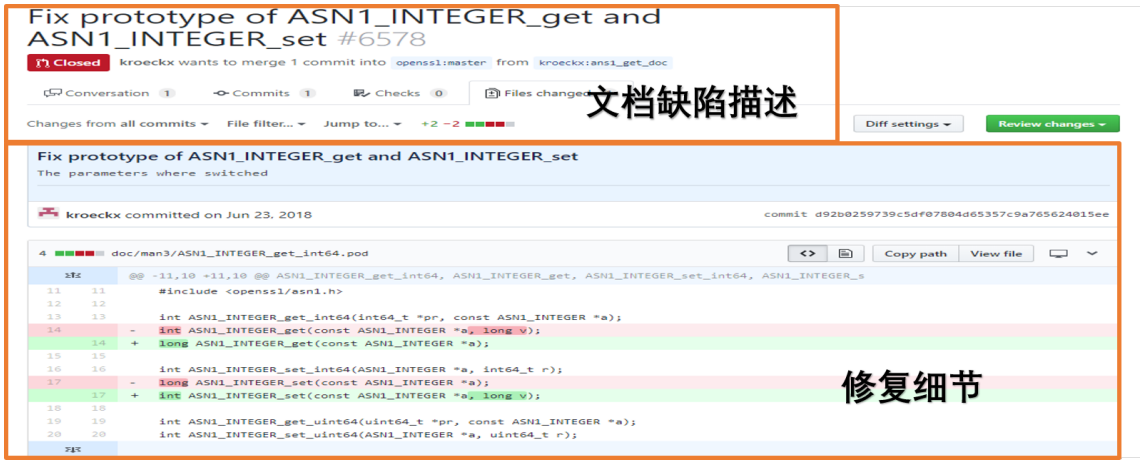

图 4.11　OpenSSL 中接口用户手册缺陷修复记录

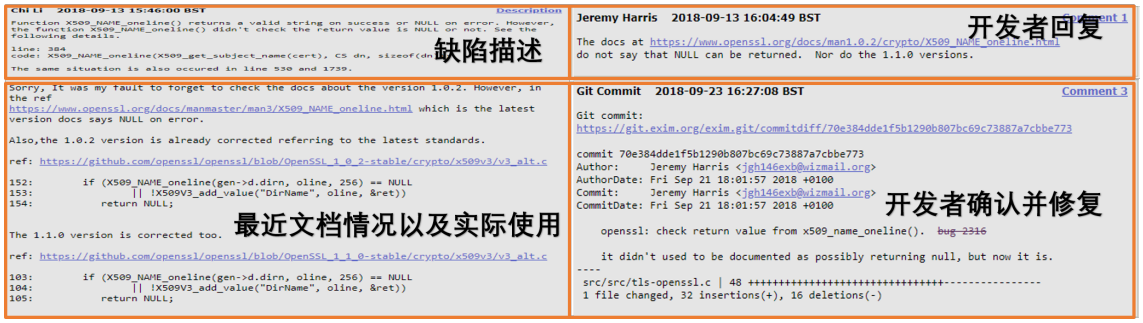

图 4.12　exim 项目中缺陷 2316 修复记录

或者 Stack-overflow 中搜索，而不是参考官方文档。

- 客户端未能感知软件库的更新。如图4.12中所示，在 exim 项目中作者在提交缺陷报告（ID-2316[①]）后，开发者根据旧版本的用户手册认为缺陷不成立。然而，在 OpenSSL 最新的用户手册和最新的代码中，已经对相应的接口进行更新。在提交相应的说明后，开发者对缺陷修复。

总结来说，即使现有的软件库提供格式良好的使用说明和参考用例，开发者在遇到问题时，依旧首选网络平台进行咨询。然而，根据调研结果显示[32]，网络平台中的代码质量并不可靠。特别地，随着开源软件社区的蓬勃发展，大量的软件库缺少文档。所以，现有的文档形式难以满足用户的需求。因此，本章基于缺陷模式设计面向 C 程序接口使用约束的领域特定语言 IMSpec。该语言旨在和现有的自然语言描述形成互补，以提供更好的接口使用约束描述方法。

缺陷提交　在缺陷检测后，本章针对于选择的缺陷实例细节进行深入分析，并提交缺陷报告给相应的开发者。起初，在缺陷报告中只包含错误的位置和原因。开发者很少反馈，或者希望作者提供更多的有效信息。在此基础之上，作者丰富缺陷报告，即在报告中包含：缺陷位置、原因、具体路径信息和对比正确使用片段，供开发者更好地理解缺陷。对于改进后的缺陷报告，提交的多数报告被开发者接受并修复。作者将缺陷报告提交中的经验总结如下：

- 开发者更喜欢提交修复代码（Pull Request）。在缺陷提交的过程中，Linux 内核的维护者表示只接受修复代码。同样，Ubuntu 的应用软件的开发者则更欢迎提交修复代码。OpenSSL 开发者欢迎提交修复代码，同时积极地在帮助作者进行缺陷修复。
- 提交策略。在缺陷报告提交初期，本章连续地提交数十个缺陷报告，获得回复较少。一个可能的原因为开发者认为这样的缺陷报告是批量产生，质量不

① https://bugs.exim.org/show_bug.cgi?id=2316

高。此后，作者降低提交频率，不仅获得较多回复，同时缺陷的认可和修复率都提高。

- 缺陷报告信息要充分。如上文所述，丰富缺陷报告信息，能够有效地帮助开发者理解缺陷细节。特别是缺陷产生的上下文，以及如何正确使用目标接口。

总结来说，缺陷提交需要大量的人力进行信息核对和确认。在提交过程中，作者发现开发者更喜欢接受修复代码。然而，缺陷修复需要大量的领域知识和上下文信息。此外，基于差异性的路径信息能够有效地帮助开发者理解缺陷原因，并提供可能的缺陷修复模式。因此，在 Tsmart-IMChecker 工具集中本章设计并实现基于差异性对比的缺陷结果展示工具 IMDisplayer。

开发者态度　在缺陷报告的过程中，开发者展现出不同的态度。一方面，开发者对缺陷报告进行积极响应，例如 OpenSSL 的开发者多数在 1 天就完成缺陷的修复。另一方面，开发者则更希望作者提供实际的运行轨迹或者提供修复代码，而不仅仅是缺陷报告。本章将与开发者的交流和讨论中总结出的特殊发现描述如下，

- 开发者自身忽视接口使用的约束。在实际开发过程中，开发者难以做到完全正确。即使是简单的参数检查和返回值检查，在大量的代码中普遍存在忽略必要检查的接口误用缺陷。然而，这些缺陷难以通过人工的方式进行一一排查。特别地，当开发者认识到这些缺陷时，会对其他代码中同样的缺陷模式进行检测。如图4.13所示，在 keepalived 项目中，本章提交缺陷报告后，开发者不仅修复对应的缺陷，还对其他的代码进行类似缺陷的检测和修复。
- 特殊目的。虽然有些接口使用违反约束，却是开发者故意为之或者在特殊

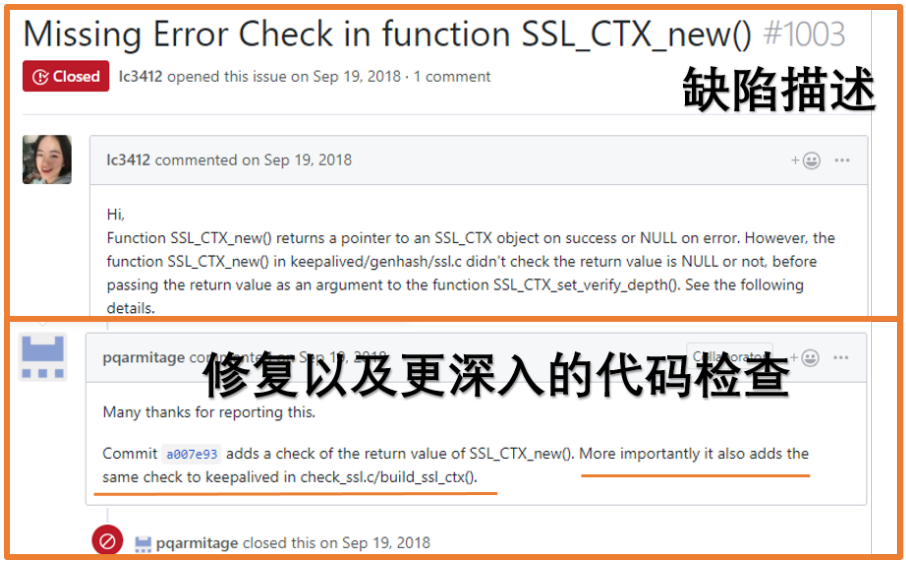

图 4.13　keepalived 开发者根据本章缺陷报告进行深入的代码检查

的应用上下文，无需修复。例如 OpenSSL 项目提供 app/speed.c 文件用于测试算法速度。因此在代码中，存在大量的缺少必要返回值检查的接口误用（IssueID-6575）。OpenSSL 一个开发者认为这些误用在这个上下文中并不需要关注。然而，其他开发者认为这部分代码作为应用中的案例，是其他开发者的重要参考需要重视。开发者进行深入的讨论后，决定在后续的开发过程中进行统一处理（Assessed）。另一方面，由于 C 程序缺少异常处理机制，开发者会故意忽略部分异常检测，因为软件设计时本身就没有考虑这些情况。

总结来说，基于有效信息的缺陷报告能够获得开发者积极的反馈。同时并不是所有的接口误用都会引起开发者的关注。特别地，有一些误用是开发者故意为之。例如，提高运行效率、C 程序无法支持的异常处理情况和接口本身的特殊性等等。

4.5　本章小结

本章将 C 程序接口使用约束领域特定语言 IMSpec 和规模化接口误用缺陷检测方法 IMChecker 在实际项目中进行应用。为帮助研究人员和开发者理解接口误用缺陷，本章整理 C 程序接口误用缺陷数据集 APIMU4C。该数据集包含来源于实际项目的接口误用案例库以及接口误用测试数据集，从而帮助研究人员和开发者对检测工具的性能进行评估、针对性选择检测工具以及设计新的检测算法。同时，本章设计并实现可视化支撑的 C 程序接口误用缺陷检测工具集 Tsmart-IMChecker，并将工具集应用于开源项目缺陷检测中。应用结果显示，本章的方法能够有效地检测到实际项目中的未被发现的接口误用缺陷，并在 Linux 内核、OpenSSL 库和 Ubuntu 系统应用软件的最新稳定版本中提交 75 个缺陷报告。其中 61 个已经被开发者确认，32 个被开发者修复。

第 5 章　结束语

5.1　工作总结

接口误用是导致软件错误、系统崩溃和漏洞的重要原因之一。为提高接口误用检测能力以应对实际项目中检测精度与规模的矛盾关系，本文以 C 程序作为载体，从接口使用约束描述、缺陷检测算法和实际项目应用三个角度展开系统性研究，取得相应理论成果并实现相应的工具集合。本文的工作具体总结如下：

1. 提出基于缺陷模式的接口使用约束领域特定语言 IMSpec。为理解 C 程序接口误用缺陷特性，本文对六个不同领域被广泛使用的开源软件中 830 个接口误用缺陷修复报告进行研究，总结出三大类通用接口缺陷模式（参数、异常处理和函数调用关系）。基于缺陷模式特性，本文提出接口使用约束的领域特定描述语言 IMSpec，并给出 IMSpec 语言的设计思路，定义语法结构与形式化语义。IMSpec 能够有效地描述实际项目中接口误用实例的接口使用约束，为形式化定义接口使用约束条件与接口缺陷检测打下基础。
2. 设计基于约束描述的规模化接口误用缺陷检测方法 IMChecker。该方法通过对目标接口使用情况进行计算，构造分析上下文环境。通过多入口分析策略将大规模代码静态分析任务分解为独立的子任务。针对每一个子任务，即一个目标接口的某个特定调用上下文，通过提取和目标接口缺陷相关的程序语句，并利用抽象符号对路径的语义信息进行记录。最后基于目标接口的使用约束对抽象路径进行缺陷检测。对于多入口分析带来的精度损失，本文通过基于上下文的语义信息以及基于使用情况的统计信息两种策略对检测结果进行过滤与排序。本文在公开数据集 Juliet Test Suite 的 13 个接口缺陷相关的 CWE 分类上取得 13.21% 的误报率和 16.80% 的漏报率。该结果领先于主流的开源静态分析工具。
3. 总结 C 程序接口误用缺陷数据集 APIMU4C 并开发基于图形化的 C 程序接口误用缺陷检测工具集 Tsmart-IMChecker。基于接口误用缺陷实例以及公开测试集合，本文构造 C 程序接口误用缺陷数据集 APIMU4C，以帮助研究人员和开发者更好地理解 C 程序接口缺陷、评估检测工具的能力以及展开新的研究工作。Tsmart-IMChecker 工具集包含可视化规约撰写工具 IMSpec-writer、缺陷分析引擎 IMChecker-engine 以及基于差异性对比的结果展示工具 IMDisplayer。本文将 Tsmart-IMChecker 工具集应用于开源项目中，在最新的 Linux 内核、OpenSSL 安全协议加密库以及 Ubuntu 操作系统应用软件中

找到 112 个新的缺陷。至今，在提交的 75 个缺陷报告中，61 个已经被开发者确认，32 个被开发者修复。本文将实际项目应用中的结果和经验进行总结。

5.2　研究展望

在现有的研究基础上，为进一步保障 C 程序接口的正确使用，拟从如下 3 个方面开展进一步的研究：

1. **IMSpec 描述语言**：目前 IMSpec 规约描述语言由人工撰写。虽然 IMSpec 具有轻量级的语法形式、可视化撰写工具以及一次撰写多次使用的特点，人工撰写规约依旧在实际应用中面临挑战。特别是针对于没有文档的接口以及缺少领域经验的开发者，接口使用规约的正确性难以保证。自动规约挖掘技术能够从大量的代码库中学习接口使用规约。因此可以集成现有基于数据挖掘技术的规约推理工具，以利用自动化学习的结果。IMSpec 能够有效地支持现有挖掘技术的规约模式。一个可能的工作流程可以是，首先利用自动挖掘技术对规约进行学习并生成规约列表，供用户选择和修改。针对缺失的规约，用户可以直接撰写相应的约束条件。此外，一个可靠的规约分享平台能够有效地在不同开发者间分享已有规约描述。同时，IMSpec 目前针对调研结果中的接口使用约束设计，无法支持全部 C 语言语法结构和语义。未来可以逐渐扩展 IMSpec 的语法和语义，并提供语言的评估标准。
2. **工具集合**：为支持大规模代码缺陷检测，本文采取基于多入口的分析策略。因此分析中由于上下文信息丢失，会带来精度损失。可能解决方案包括：（1）增加跨函数的摘要信息计算，以获得更加准确的上下文信息，提升过滤效果；（2）设计基于概率模型的排序算法，以优先展示具有高置信度、高影响域的缺陷检测结果；（3）优化路径提取策略，以针对性地获取更加丰富的语义信息。为能够帮助开发者撰写 IMSpec 约束和对结果进行审查，本文开发可视化工具集合 Tsmart-IMChecker。目前，可视化客户端并没有进行测试，包括：用户友好性、稳定性等等。因此，未来可以邀请实际开发者对客户端试用以及测试，提高工具集的实际效果。
3. **自动修复**：研究自动化接口缺陷修复技术，以辅助开发者修复缺陷。本文在对接口缺陷检测的过程中，记录正确的使用路径以帮助开发者理解缺陷原因、提供缺陷修改意见。后续，可以结合代码综合技术，自动化生成修复补丁，降低开发者的维护成本。另一方面，针对修复后的代码，可以采用回归分析的策略，降低代码分析成本。特别地，对已经检测过的目标接口，如果对应上下文相关代码没有被修改，则不必进行缺陷检测。

参考文献

[1] 中华人民共和国工业和信息化部. 2018 年软件和信息技术服务业统计公报解读[EB/OL]. 2019. http://www.miit.gov.cn/n1146285/n1146352/n3054355/n3057511/n3057518/c6633639/content.html.

[2] Savor T, Douglas M, Gentili M, et al. Continuous deployment at facebook and OANDA[C/OL]// Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume. 2016: 21-30. https://doi.org/10.1145/2889160.2889223.

[3] Savolainen J, Raatikainen M, Männistö T. Eight practical considerations in applying feature modeling for product lines[C/OL]//Top Productivity through Software Reuse - 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings. 2011: 192-206. https://doi.org/10.1007/978-3-642-21347-2_15.

[4] Schwittek W, Eicker S. A study on third party component reuse in java enterprise open source software[C/OL]//CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013. 2013: 75-80. https://doi.org/10.1145/2465449.2465468.

[5] Wikipedia. Software library[EB/OL]. 2019. https://en.wikipedia.org/wiki/Library_(computing).

[6] Amann S, Nadi S, Nguyen H A, et al. Mubench: a benchmark for api-misuse detectors[C/OL]// Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016. 2016: 464-467. https://doi.org/10.1145/2901739.2903506.

[7] Fahl S, Harbach M, Muders T, et al. Why eve and mallory love android: an analysis of android SSL (in)security[C/OL]//the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. 2012: 50-61. https://doi.org/10.1145/2382196.2382205.

[8] Georgiev M, Iyengar S, Jana S, et al. The most dangerous code in the world: validating SSL certificates in non-browser software[C/OL]//the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. 2012: 38-49. https://doi.org/10.1145/2382196.2382204.

[9] Egele M, Brumley D, Fratantonio Y, et al. An empirical study of cryptographic misuse in android applications[C/OL]//2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. 2013: 73-84. https://doi.org/10.1145/2508859.2516693.

[10] Monperrus M, Mezini M. Detecting missing method calls as violations of the majority rule [J/OL]. ACM Trans. Softw. Eng. Methodol., 2013, 22(1): 7:1-7:25. https://doi.org/10.1145/2430536.2430541.

[11] Lazar D, Chen H, Wang X, et al. Why does cryptographic software fail? a case study and open problems[C/OL]//Asia-Pacific Workshop on Systems, APSys'14, Beijing, China, June 25-26, 2014. 2014: 7:1-7:7. https://doi.org/10.1145/2637166.2637237.

[12] Sunshine J, Herbsleb J D, Aldrich J. Searching the state space: a qualitative study of API protocol usability[C/OL]//Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015. 2015: 82-93. https://doi.org/10.1109/ICPC.2015.17.

[13] Legunsen O, Hassan W U, Xu X, et al. How good are the specs? a study of the bug-finding effectiveness of existing java API specifications[C/OL]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. 2016: 602-613. https://doi.org/10.1145/2970276.2970356.

[14] Cve-2015-0288[EB/OL]. https://www.cvedetails.com/cve/CVE-2015-0288/.

[15] Openssl: cryptography and ssl/tls toolkit.[EB/OL]. https://github.com/openssl/openssl.

[16] Weaver A C. Secure sockets layer[J/OL]. IEEE Computer, 2006, 39(4): 88-90. https://doi.org/10.1109/MC.2006.138.

[17] Dekel U, Herbsleb J D. Improving API documentation usability with knowledge pushing [C/OL]//31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. 2009: 320-330. https://doi.org/10.1109/ICSE.2009.5070532.

[18] Uddin G, Robillard M P. How API documentation fails[J/OL]. IEEE Software, 2015, 32(4): 68-75. https://doi.org/10.1109/MS.2014.80.

[19] Zhou Y, Gu R, Chen T, et al. Analyzing apis documentation and code to detect directive defects [C/OL]//Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. 2017: 27-37. https://doi.org/10.1109/ICSE.2017.11.

[20] 程序设计问答网站[EB/OL]. https://stackoverflow.com/.

[21] Acar Y, Backes M, Fahl S, et al. You get where you're looking for: The impact of information sources on code security[C/OL]//IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. 2016: 289-305. https://doi.org/10.1109/SP.2016.25.

[22] Nadi S, Krüger S, Mezini M, et al. Jumping through hoops: why do java developers struggle with cryptography apis?[C/OL]//Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. 2016: 935-946. https://doi.org/10.1145/2884781.2884790.

[23] Robillard M P, Walker R J, Zimmermann T. Recommendation systems for software engineering [J/OL]. IEEE Software, 2010, 27(4): 80-86. https://doi.org/10.1109/MS.2009.161.

[24] Holmes R, Murphy G C. Using structural context to recommend source code examples[C/OL]//27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA. 2005: 117-125. https://doi.org/10.1145/1062455.1062491.

[25] Treude C, Robillard M P. Augmenting API documentation with insights from stack overflow [C/OL]//Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. 2016: 392-403. https://doi.org/10.1145/2884781.2884800.

[26] Ponzanelli L, Bavota G, Penta M D, et al. Mining stackoverflow to turn the IDE into a self-confident programming prompter[C/OL]//11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India. 2014: 102-111. https://doi.org/10.1145/2597073.2597077.

[27] Proksch S, Lerch J, Mezini M. Intelligent code completion with bayesian networks[J/OL]. ACM Trans. Softw. Eng. Methodol., 2015, 25(1): 3:1-3:31. https://doi.org/10.1145/2744200.

[28] Delaitre A, Stivalet B, Fong E, et al. Evaluating bug finders - test and measurement of static code analyzers[C/OL]//1st IEEE/ACM International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015, Florence, Italy, May 23, 2015. 2015: 14-20. https://doi.org/10.1109/COUFLESS.2015.10.

[29] Wang M, Liang J, Chen Y, et al. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing[C/OL]//Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. 2018: 61-64. https://doi.org/10.1145/3183440.3183494.

[30] Amann S, Nguyen H A, Nadi S, et al. A systematic evaluation of static api-misuse detectors[J]. IEEE Transactions on Software Engineering, 2018: 1-1 (Early Access).

[31] Nagappan N, Ball T. Static analysis tools as early indicators of pre-release defect density[C/OL]// 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA. 2005: 580-586. https://doi.org/10.1145/1062455.1062558.

[32] Zhang T, Upadhyaya G, Reinhardt A, et al. Are code examples on an online q&a forum reliable?: a study of API misuse on stack overflow[C/OL]//Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. 2018: 886-896. https://doi.org/10.1145/3180155.3180260.

[33] Oliveira D S, Lin T, Rahman M S, et al. API blindspots: Why experienced developers write vulnerable code[C/OL]//Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018. 2018: 315-328. https://www.usenix.org/conference/soups2018/presentation/oliveira.

[34] Bessey A, Block K, Chelf B, et al. A few billion lines of code later: using static analysis to find bugs in the real world[J/OL]. Commun. ACM, 2010, 53(2): 66-75. https://doi.org/10.1145/1646353.1646374.

[35] Johnson B, Song Y, Murphy-Hill E R, et al. Why don't software developers use static analysis tools to find bugs?[C/OL]//35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. 2013: 672-681. https://doi.org/10.1109/ICSE.2013.6606613.

[36] 代码审查[EB/OL]. https://en.wikipedia.org/wiki/Code_review.

[37] Github[EB/OL]. https://github.com/about.

[38] Bitbucket[EB/OL]. https://bitbucket.org/.

[39] Baum T, Leßmann H, Schneider K. The choice of code review process: A survey on the state of the practice[C/OL]//Product-Focused Software Process Improvement - 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 1, 2017, Proceedings. 2017: 111-127. https://doi.org/10.1007/978-3-319-69926-4_9.

[40] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review[C/OL]// 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. 2013: 712-721. https://doi.org/10.1109/ICSE.2013.6606617.

[41] Jones C. Measuring defect potentials and defect removal efficiency[J]. CrossTalk The Journal of Defense Software Engineering, 2008, 21(6): 11-13.

[42] Bosu A, Carver J C. Impact of peer code review on peer impression formation: A survey [C/OL]//2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013. 2013: 133-142. https://doi.org/10.1109/ESEM.2013.23.

[43] Czerwonka J, Greiler M, Tilford J. Code reviews do not find bugs. how the current code review best practice slows us down[C/OL]//37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2. 2015: 27-28. https://doi.org/10.1109/ICSE.2015.131.

[44] Nielson F, Nielson H R, Hankin C. Principles of program analysis[M/OL]. Springer, 1999. https://doi.org/10.1007/978-3-662-03811-6

[45] Li J, Chen J, Huang M, et al. An integration testing platform for software vulnerability detection method[C/OL]//2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017. 2017: 984-989. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.341.

[46] Bartocci E, Falcone Y, Francalanza A, et al. Introduction to runtime verification[M/OL]// Lectures on Runtime Verification - Introductory and Advanced Topics. 2018: 1-33. https://doi.org/10.1007/978-3-319-75632-5_1

[47] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation [C/OL]//Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. 2007: 89-100. https://doi.org/10.1145/1250734.1250746.

[48] Serebryany K, Bruening D, Potapenko A, et al. Addresssanitizer: A fast address sanity checker[C/OL]//2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012. 2012: 309-318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[49] Clang: a c language family frontend for llvm[EB/OL]. https://clang.llvm.org/.

[50] Gcc: the gnu compiler collection[EB/OL]. https://gcc.gnu.org/.

[51] Xcode: an integrated development environment (ide) for macos[EB/OL]. https://developer.apple.com/xcode/.

[52] Bugs detected by addresssanitizer[EB/OL]. https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs.

[53] Li J, Zhao B, Zhang C. Fuzzing: A survey[J]. Cybersecurity, 2018, 1(1): 6.

[54] Liang J, Wang M, Chen Y, et al. Fuzz testing in practice: Obstacles and solutions[C/OL]//25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018. 2018: 562-566. https://doi.org/10.1109/SANER.2018.8330260.

[55] Zakharov I S, Mandrykin M U, Mutilin V S, et al. Configurable toolset for static verification of operating systems kernel modules[J/OL]. Programming and Computer Software, 2015, 41(1): 49-64. https://doi.org/10.1134/S0361768815010065.

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

[57] D'silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2008, 27(7): 1165-1178.

[58] Reps T. Program analysis via graph reachability[J]. Information and software technology, 1998, 40(11-12): 701-726.

[59] Slam: a project for checking that software satisfies critical behavioral properties of the interfaces. [EB/OL]. https://www.microsoft.com/en-us/research/project/slam/.

[60] Ball T, Rajamani S. Slic: A specification language for interface checking (of c)[R/OL]. 2002: 1-12. https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/.

[61] Ball T, Rajamani S K. The slam project: debugging system software via static analysis[C/OL]// ACM SIGPLAN Notices: volume 37. ACM, 2002: 1-3. https://doi.org/10.1145/503272.503274.

[62] Clarke E, Grumberg O, Jha S, et al. Counterexample-guided abstraction refinement[C/OL]// International Conference on Computer Aided Verification. Springer, 2000: 154-169. https://doi.org/10.1007/10722167_15.

[63] Henzinger T A, Jhala R, Majumdar R, et al. Lazy abstraction[J/OL]. ACM SIGPLAN Notices, 2002, 37(1): 58-70. https://doi.org/10.1145/503272.503279.

[64] Ball T, Bounimova E, Kumar R, et al. Slam2: Static driver verification with under 4% false alarms[C/OL]//Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. FMCAD Inc, 2010: 35-42. http://ieeexplore.ieee.org/document/5770931/.

[65] Ball T, Levin V, Rajamani S K. A decade of software model checking with slam[J/OL]. Communications of the ACM, 2011, 54(7): 68-76. https://doi.org/10.1145/1965724.1965743.

[66] Post H, Sinz C, Küchlin W. Towards automatic software model checking of thousands of linux modules - a case study with avinux[J/OL]. Softw. Test., Verif. Reliab., 2009, 19(2): 155-172. https://doi.org/10.1002/stvr.399.

[67] Witkowski T, Blanc N, Kroening D, et al. Model checking concurrent linux device drivers [C/OL]//Stirewalt R E K, Egyed A, Fischer B. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. ACM, 2007: 501-504. https://doi.org/10.1145/1321631.1321719.

[68] Beyer D, Henzinger T A, Théoduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis[C/OL]//International Conference on Computer Aided Verification. Springer, 2007: 504-518. https://doi.org/10.1007/978-3-540-73368-3_51.

[69] Kroening D, Tautschnig M. CBMC - C bounded model checker - (competition contribution) [C/OL]//Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. 2014: 389-391. https://doi.org/10.1007/978-3-642-54862-8_26.

[70] Rakamaric Z, Emmi M. SMACK: decoupling source language details from verifier implementations[C/OL]//Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. 2014: 106-113. https://doi.org/10.1007/978-3-319-08867-9_7.

[71] Beyer D, Chlipala A, Henzinger T A, et al. The blast query language for software verification. [C/OL]//Giacobazzi R. Lecture Notes in Computer Science: volume 3148 Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. Springer, 2004: 2-18. https://doi.org/10.1007/978-3-540-27864-1_2.

[72] Baudin P, Filliâtre J C, Marché C, et al. Acsl: Ansi c specification language[EB/OL]. 2008. https://frama-c.com/download/acsl_1.4.pdf.

[73] Soni M. Defect prevention: reducing costs and enhancing quality[J/OL]. iSixSigma. com, 2006, 19. https://www.isixsigma.com/tools-templates/software/defect-prevention-reducing-costs-and-enhancing-quality/.

[74] Engler D R, Chelf B, Chou A, et al. Checking system rules using system-specific, programmer-written compiler extensions[C/OL]//Jones M B, Kaashoek M F. 4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000. USENIX Association, 2000: 1-16. http://dl.acm.org/citation.cfm?id=1251230.

[75] Lawall J L, Brunel J, Palix N, et al. WYSIWIB: A declarative approach to finding API protocols and bugs in linux code[C/OL]//Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009. IEEE Computer Society, 2009: 43-52. https://doi.org/10.1109/DSN.2009.5270354.

[76] List of tools for static code analysis.[EB/OL]. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

[77] Clang static analyzer.[EB/OL]. http://clang-analyzer.llvm.org/.

[78] Cppcheck: a tool for static c/c++ code analysis.[EB/OL]. http://cppcheck.sourceforge.net/.

[79] Infer: A tool to detect bugs in java and c/c++/objective-c code before it ships.[EB/OL]. https://fbinfer.com/.

[80] Sparse: a tool for static code analysis that helps kernel developers to detect coding errors. [EB/OL]. https://kernelnewbies.org/Sparse.

[81] Splint: a tool for statically checking c programs for coding errors and security vulnerabilities. [EB/OL]. https://sourceforge.net/projects/splint/.

[82] He B, Rastogi V, Cao Y, et al. Vetting SSL usage in applications with SSLINT[C/OL]//2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. IEEE Computer Society, 2015: 519-534. https://doi.org/10.1109/SP.2015.38.

[83] The llvm project is a collection of modular and reusable compiler and toolchain technologies. [EB/OL]. http://llvm.org/.

[84] Cadar C, Sen K. Symbolic execution for software testing: three decades later.[J]. Commun. ACM, 2013, 56(2): 82-90.

[85] Robillard M P, Bodden E, Kawrykow D, et al. Automated API property inference techniques [J/OL]. IEEE Trans. Software Eng., 2013, 39(5): 613-637. https://doi.org/10.1109/TSE.2012.63.

[86] Dyer R, Nguyen H A, Rajan H, et al. Boa: Ultra-large-scale software repository and source-code mining[J/OL]. ACM Trans. Softw. Eng. Methodol., 2015, 25(1): 7:1-7:34. https://doi.org/10.1145/2803171.

[87] Engler D R, Chen D Y, Chou A. Bugs as inconsistent behavior: A general approach to inferring errors in systems code[C/OL]//Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001. 2001: 57-72. https://doi.org/10.1145/502034.502041.

[88] Grahne G, Zhu J. Efficiently using prefix-trees in mining frequent itemsets[C/OL]//Goethals B, Zaki M J. CEUR Workshop Proceedings: volume 90 FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA. CEUR-WS.org, 2003. http://ceur-ws.org/Vol-90/grahne.pdf.

[89] Li Z, Zhou Y. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code[C/OL]//Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005. 2005: 306-315. https://doi.org/10.1145/1081706.1081755.

[90] Wasylkowski A, Zeller A, Lindig C. Detecting object usage anomalies[C/OL]//Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007. 2007: 35-44. https://doi.org/10.1145/1287624.1287632.

[91] Thummalapenta S, Xie T. Alattin: Mining alternative patterns for detecting neglected conditions[C/OL]//ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009. IEEE Computer Society, 2009: 283-294. https://doi.org/10.1109/ASE.2009.72.

[92] Nguyen H A, Dyer R, Nguyen T N, et al. Mining preconditions of apis in large-scale code corpus [C/OL]//Cheung S, Orso A, Storey M D. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. ACM, 2014: 166-177. https://doi.org/10.1145/2635868.2635924.

[93] Kang Y J, Ray B, Jana S. Apex: automated inference of error specifications for C apis[C/OL]// Lo D, Apel S, Khurshid S. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. ACM, 2016: 472-482. https://doi.org/10.1145/2970276.2970354.

[94] Yun I, Min C, Si X, et al. Apisan: Sanitizing API usages through semantic cross-checking [C/OL]//Holz T, Savage S. 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. USENIX Association, 2016: 363-378. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun.

[95] Liang B, Bian P, Zhang Y, et al. Antminer: mining more bugs by reducing noise interference [C/OL]//Dillon L K, Visser W, Williams L. Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM, 2016: 333-344. https://doi.org/10.1145/2884781.2884870.

[96] Gu Z, Wu J, Li C, et al. Vetting api usages in c programs with imchecker[C/OL]//41th International Conference on Software Engineering, ICSE '19, Montreal, QC, Canada, May 25-31, 2019. 2019: 1-1 (Early Access). http://tomgu1991.github.io/blog/Research/ICSE-Demo-592.pdf.

[97] Gu Z, Zhou M, Wu J, et al. An extensible approach to exploring the incorrect usage of apis [C/OL]//The 13th International Symposium on Theoretical Aspects of Software Engineering, Guilin, China, 29 July - 1 August, 2019. 2019: 1-1 (Early Access). https://tomgu1991.github.io/blog/Research/tase19-paper16.pdf.

[98] Gu Z, Wu J, Liu J, et al. An empirical study on api-misuse bugs in open-source c programs [C/OL]//COMPSAC 2019, Milwaukee, Wisconsin, USA, July 15-19, 2019. 2019: 1-1 (Early Access). https://tomgu1991.github.io/blog/Research/compsac19-paper15.pdf.

[99] Juliet test suite.[EB/OL]. https://samate.nist.gov/SRD/testsuite.php.

[100] Hatcliff J, Leavens G T, Leino K R M, et al. Behavioral interface specification languages[J/OL]. ACM Comput. Surv., 2012, 44(3): 16:1-16:58. https://doi.org/10.1145/2187671.2187678.

[101] Meyer B. Applying "design by contract"[J/OL]. IEEE Computer, 1992, 25(10): 40-51. https://doi.org/10.1109/2.161279.

[102] Zhong H, Mei H. An empirical study on api usages[J/OL]. IEEE Transactions on Software Engineering, 2017. http://ieeexplore.ieee.org/document/8186224/.

[103] Okur S, Dig D. How do developers use parallel libraries?[C/OL]//20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. 2012: 54. https://doi.org/10.1145/2393596.2393660.

[104] Robbes R, Lungu M, Röthlisberger D. How do developers react to API deprecation?: the case of a smalltalk ecosystem[C/OL]//20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. 2012: 56. https://doi.org/10.1145/2393596.2393662.

[105] Hora A C, Robbes R, Valente M T, et al. How do developers react to API evolution? A large-scale empirical study[J/OL]. Software Quality Journal, 2018, 26(1): 161-191. https://doi.org/10.1007/s11219-016-9344-4.

[106] Shi L, Zhong H, Xie T, et al. An empirical study on evolution of API documentation[C/OL]// Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. 2011: 416-431. https://doi.org/10.1007/978-3-642-19811-3_29.

[107] Wu W, Serveaux A, Guéhéneuc Y, et al. The impact of imperfect change rules on framework API evolution identification: an empirical study[J/OL]. Empirical Software Engineering, 2015, 20(4): 1126-1158. https://doi.org/10.1007/s10664-014-9317-9.

[108] Robillard M P, DeLine R. A field study of API learning obstacles[J/OL]. Empirical Software Engineering, 2011, 16(6): 703-732. https://doi.org/10.1007/s10664-010-9150-8.

[109] Bavota G, Vásquez M L, Bernal-Cárdenas C E, et al. The impact of API change- and fault-proneness on the user ratings of android apps[J/OL]. IEEE Trans. Software Eng., 2015, 41(4): 384-407. https://doi.org/10.1109/TSE.2014.2367027.

[110] Zhong H, Thummalapenta S, Xie T. Exposing behavioral differences in cross-language API mapping relations[C/OL]//Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. 2013: 130-145. https://doi.org/10.1007/978-3-642-37057-1_10.

[111] Barnett M, DeLine R, Fähndrich M, et al. The spec# programming system: Challenges and directions[C/OL]//Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. 2005: 144-152. https://doi.org/10.1007/978-3-540-69149-5_16.

[112] Jana S, Kang Y J, Roth S, et al. Automatically detecting error handling bugs using error specifications[C/OL]//25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. 2016: 345-362. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana.

[113] Kosmatov N, Signoles J. Frama-c, A collaborative framework for C code verification: Tutorial synopsis[C/OL]//Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. 2016: 92-115. https://doi.org/10.1007/978-3-319-46982-9_7.

[114] Francis N, Green A, Guagliardo P, et al. Cypher: An evolving query language for property graphs[C/OL]//Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. 2018: 1433-1445. https://doi.org/10.1145/3183713.3190657.

[115] Zubrow D. Ieee standard classification for software anomalies[J]. IEEE Computer Society, 2009.

[116] Chillarege R, Bhandari I S, Chaar J K, et al. Orthogonal defect classification - A concept for in-process measurements[J/OL]. IEEE Trans. Software Eng., 1992, 18(11): 943-956. https://doi.org/10.1109/32.177364.

[117] Beller M, Bholanath R, McIntosh S, et al. Analyzing the state of static analysis: A large-scale evaluation in open source software[C/OL]//IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1. 2016: 470-481. https://doi.org/10.1109/SANER.2016.105.

[118] Thummalapenta S, Xie T. Mining exception-handling rules as sequence association rules [C/OL]//31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. 2009: 496-506. https://doi.org/10.1109/ICSE.2009.5070548.

[119] Wasylkowski A, Zeller A, Lindig C. Detecting object usage anomalies[C/OL]//Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007. 2007: 35-44. https://doi.org/10.1145/1287624.1287632.

[120] Monperrus M, Eichberg M, Tekes E, et al. What should developers be aware of? an empirical study on the directives of API documentation[J/OL]. Empirical Software Engineering, 2012, 17(6): 703-737. https://doi.org/10.1007/s10664-011-9186-4.

[121] Source code of linux kernel v4.18-rc4.[EB/OL]. https://github.com/torvalds/linux/releases/tag/v4.18-rc4.

[122] Ffmpeg: a collection of libraries and tools to process multimedia content.[EB/OL]. https://github.com/FFmpeg/FFmpeg.

[123] Curl: a command line tool and library for transferring data with url syntax.[EB/OL]. https://github.com/curl/curl.

[124] Berners-Lee T, Masinter L, McCahill M. Uniform resource locators (url)[R/OL]. 1994. https://tools.ietf.org/html/rfc1738.

[125] Freerdp: a free remote desktop protocol library and clients.[EB/OL]. https://github.com/FreeRDP/FreeRDP.

[126] Httpd: a powerful and flexible http/1.1 compliant web server.[EB/OL]. https://github.com/apache/httpd.

[127] Cve-2014-0092[EB/OL]. https://www.cvedetails.com/cve/CVE-2014-0092/.

[128] Cve-2015-0208[EB/OL]. https://www.cvedetails.com/cve/CVE-2015-0208/.

[129] Cve-2015-0285[EB/OL]. https://www.cvedetails.com/cve/CVE-2015-0285/.

[130] Cve-2017-3318[EB/OL]. https://www.cvedetails.com/cve/CVE-2017-3318/.

[131] Cve-2017-5350[EB/OL]. https://www.cvedetails.com/cve/CVE-2015-5350/.

[132] Owasp top 10.[EB/OL]. https://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf.

[133] Gunawi H S, Rubio-González C, Arpaci-Dusseau A C, et al. EIO: error handling is occasionally correct[C/OL]//6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA. 2008: 207-222. http://www.usenix.org/events/fast08/tech/gunawi.html.

[134] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form and the control dependence graph[J]. (TOPLAS), 1991, 13(4): 451-490.

[135] CWE, Institute S. 2011 cwe/sans top 25 most dangerous software errors.[EB/OL]. http://cwe.mitre.org/top25/.

[136] Foundation T O. 2017 owasp top 10 application security risks.[EB/OL]. https://www.owasp.org/index.php/Top_10-2017_Top_10.

[137] 源伞科技公司. Poinpoint 静态分析工具[EB/OL]. https://www.sourcebrella.com/pinpoint/.

[138] Inc. S. Coverity scan static analysis tool[EB/OL]. https://scan.coverity.com/.

[139] Ramanathan M K, Grama A, Jagannathan S. Static specification inference using predicate mining [C/OL]//Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. 2007: 123-134. https://doi.org/10.1145/1250734.1250749.

[140] Ramanathan M K, Grama A, Jagannathan S. Path-sensitive inference of function precedence protocols[C/OL]//29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. 2007: 240-250. https://doi.org/10.1109/ICSE.2007.63.

[141] Saha S, Lozi J, Thomas G, et al. Hector: Detecting resource-release omission faults in error-handling code for systems software[C/OL]//2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013. 2013: 1-12. https://doi.org/10.1109/DSN.2013.6575307.

[142] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: exposing missing checks in source code for vulnerability discovery[C/OL]//2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. 2013: 499-510. https://doi.org/10.1145/2508859.2516665.

[143] Tian Y, Ray B. Automatically diagnosing and repairing error handling bugs in C[C/OL]// Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. 2017: 752-762. https://doi.org/10.1145/3106237.3106300.

[144] Henry J, Monniaux D, Moy M. PAGAI: A path sensitive static analyser[J/OL]. Electr. Notes Theor. Comput. Sci., 2012, 289: 15-25. https://doi.org/10.1016/j.entcs.2012.11.003.

[145] Lerch J, Späth J, Bodden E, et al. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T)[C/OL]//30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. 2015: 619-629. https://doi.org/10.1109/ASE.2015.9.

[146] Ben-Kiki O, Evans C, döt Net I. Yaml: a human friendly data serialization standard for all programming languages[EB/OL]. http://yaml.org/.

[147] 清华大学软件学院软件系统与工程研究所. Tsmart 软件可信保障工具集。[EB/OL]. http://tsmart.tech/show.html#/tsmart.

[148] Lu S, Li Z, Qin F, et al. Bugbench: Benchmarks for evaluating bug detection tools[C/OL]// Workshop on the evaluation of software defect detection tools: volume 5. 2005. http://pages.cs.wisc.edu/~shanlu/paper/63-lu.pdf.

[149] Just R, Jalali D, Ernst M D. Defects4j: a database of existing faults to enable controlled testing studies for java programs[C/OL]//International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014. 2014: 437-440. https://doi.org/10.1145/2610384.2628055.

[150] Static analysis benchmarks from toyota itc.[EB/OL]. https://github.com/regehr/itc-benchmarks.

致 谢

首先衷心感谢导师顾明老师和孙家广老师对本人精心的指导。依旧记得在本科保研的时候，顾老师对我提出的要求和希望，是我一辈子的目标。在五年的博士生活中每当遇到困难和疑惑时，与顾老师的讨论总能帮我找到可行的解决方案。特别感谢顾老师肯定我的能力，支持我的研究工作，并提供给我全方面的帮助。孙老师是我最敬仰的老师之一，他严谨、勤奋、谦卑的工作态度让我终生受益。难以忘记每天早上和孙老师在实验室讨论工作，给我分享成功的秘诀、深刻的软件工程思想。特别感谢孙老师派我到美国交换，让我开阔视野。

在美国伊利诺伊斯大学厄巴纳香槟分校进行六个月的交换学习期间，承蒙 Lui Sha 教授热心指导与 Andrew Ou 热心帮助，不胜感激。初到美国，感谢 Andrew 帮助我解决日常生活中的困难，带我领略异国他乡的风采，品尝当地美食。在研究期间，感谢 Lui 教授教我如何快速学习领域知识、如何与非计算机专业人士进行有效沟通，以及为我介绍领域专家帮我开拓研究领域。

感谢实验室的老师和师兄一直以来对我的指导和照顾。感谢周旻老师带领我们静态分析小组。感谢姜宇老师、宋晓宇老师、张华枫师兄、刘嘉祥师兄和刘洽师兄，帮我分析我的研究内容、指导我的论文写作，在平时督催我、在困难时给我鼓励。感谢蔡源、张宇博、高健和郭建敏同学帮我修改学术论文和博士论文。感谢吴捷成和李池，无条件信任我的决策，加入到我的博士论文工作当中。特别感谢我的博士同窗于庆涵，在我住院时对我的照顾，平日里一起吃饭、讨论、相互鼓励，愿我们的友谊地久天长。

感谢我的女友刘畅。我们 2007 年高中相识，至今已经 12 年。六年的恋爱时间里，我们从异地到异国再到异地。感谢你一直以来对我的理解和关怀，陪我一起坚守这份感情。在我情绪低落时为我鼓励加油，在我骄傲满足时提醒我。你最美丽的年华陪我度过，我必不辜负于你。

最后感谢我的父母。你们的包容、支持和关爱是我顺利走过博士求学之路最大保障。我们每年只有过年的 10 天能够团聚，平日里你们也担心打扰我的工作，每次回家我总能感受到家的温馨与快乐。父母对我的为人处世的教导以及社会经验的传授让我受益匪浅。

本课题承蒙多项国家基金的支持，特此致谢。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1991 年 06 月 02 日出生于辽宁省大连市。

2010 年 9 月考入南开大学软件学院软件工程专业，2014 年 7 月本科毕业并获得工学学士学位。

2014 年 9 月免试进入清华大学软件学院攻读博士学位至今。

发表的学术论文

[1] **Gu Z**, Jiang Y, et al. A Cyber-Physical System Framework for Early Detection of Paroxysmal Diseases[J]. IEEE Access, 2018, 6: 34834-34845. (SCI 收录, 检索号: GN6RX，影响因子 4.199)

[2] Wang Y, **Gu Z**, et al. A constraint-pattern based method for reachability determination[C]//Proceedings of the 41st IEEE Annual Computer Software and Application Conference, Turin, Italy, 2017: 85-90. (CCF-C 类会议，EI 检索号: 20174304306998)

[3] **Gu Z**, Song H, et al. An integrated Medical CPS for early detection of paroxysmal sympathetic hyperactivity[C]//2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Shenzhen, China, 2016, pp. 818-822. (CCF-B 类会议，EI 检索号：20170803377769)

[4] **Gu Z**, Wu J C, et al. Vetting API Usages in C Programs with IMChecker. In press. (已被 41th International Conference on Software Engineering 录用，CCF-A 类会议)

[5] **Gu Z**, Zhou M, et al. IMSpec: An Extensible Approach to Exploring the Incorrect Usage of APIs. In press. (已被 13th International Symposium on Theoretical Aspects of Software Engineering 录用，CCF-C 类会议)

[6] **Gu Z**, Wu J C, et al. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In press. (已被 43rd IEEE Annual Computer Software and Application Conference 录用，CCF-C 类会议)

[7] **Gu Z**, Wu J C, et al. SSLDoc: Automatically Diagnosing Incorrect SSL API Usages in C Programs. In press. (已被 The 31st International Conference on Software Engineering & Knowledge Engineering 录用，CCF-C 类会议)

[8] Li Chi, Zhou Min, **Gu Z**, et al. VBSAC: A Value-Based Static Analyzer for C. In press. (已被 28th ACM SIGSOFT International Symposium on Software Testing and Analysis 录用，CCF-A 类会议)

参与的科研项目

[1] 2016.4-2016.12：国家自然科学基金重大集成项目“可信嵌入式软件系统试验环境与示范应用”(No.91218302)

[2] 2016.9 至今：科技部国家重点研发计划“规模化漏洞分析与应用关键技术研究”(No.2016QY07X1402)

研究成果

[1] TsmartV3 工具集：http://tsmart.tech

[2] APIMU4C 数据集：https://github.com/imchecker/compsac19