计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009814] [CSTR: 32024.14.csa.009814]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Linux 配置错误检测①

孙思雨, 翟高寿, 俞朝阳

(北京交通大学 计算机科学与技术学院, 北京 100044)
通信作者: 翟高寿, E-mail: gszhai@bjtu.edu.cn

摘　要: Linux 等大型软件通常利用配置文件来调整系统功能, 当配置项数量较多时, 彼此间依赖关系将变得复杂和容易出错. 若配置依赖约束定义不当, 一定条件下会导致对应配置项即便被选中, 也会由于潜在依赖问题而无法真正生效, 甚至导致系统编译或运行错误. 现有研究主要聚焦于 Kconfig 文件且仅考虑了反向依赖可能引发的配置错误. 本文则立足于 Kconfig 和 Makefile 的全面分析, 并综合研究了前者的直接依赖和反向依赖、二者间依赖不一致以及后者配置项在前者中缺少定义等 4 种情形, 以期发现尽可能多的潜在问题. 在此基础上, 设计配置错误检测工具, 针对 Linux 6.7 版内核源码, 检出并确认存在各类配置错误共计 52 处, 验证了本文方法及原型系统的有效性和实用性.

关键词: Linux; 配置检测; 静态分析; Kconfig; Makefile

引用格式: 孙思雨,翟高寿,俞朝阳.Linux 配置错误检测.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9814.html

Error Detection of Linux Configuration

SUN Si-Yu, ZHAI Gao-Shou, YU Zhao-Yang

(School of Computer Science and Technology, Beijing Jiaotong University, Beijing 100044, China)

Abstract: Linux and other large-scale software usually use configuration files to adjust system functions. When the number of configuration items is large, the dependencies between them will become complex and error-prone. If the configuration dependency constraints are not properly defined, under certain conditions, even if the corresponding configuration item is selected, it will not take effect due to potential dependency problems, or even lead to system compilation or operation errors. Existing studies focus on Kconfig files and only consider configuration errors caused by reverse dependencies. This study comprehensively analyzes Kconfig and Makefile and investigates four scenarios of direct and reverse dependencies of Kconfig, inconsistent dependencies of the two, and the lack of definition of the latter's configuration item in the former, in order to find as many potential problems as possible. On this basis, the study designs a configuration error detection tool for the Linux 6.7 kernel source code and identifies 52 configuration errors, which verifies the effectiveness and practicality of the methodology and prototype system in this study.

Key words: Linux; configuration detection; static analysis; Kconfig; Makefile

大型软件通常需要支持多种复杂功能, 这种改变软件系统以符合特定环境要求的能力也被称作软件可变性[1]. 针对代码量大的软件, 为了维护其可变性, 通常都会建立配置空间, 方便使用者和开发者选择设置相关配置项的取值, 从而动态地选择软件的功能范畴, 确保软件能够按预期工作. 配置空间作为软件代码和软件功能模型之间联通的桥梁, 维持其正确性具有重要的意义. 然而, 在一项针对软件生产线工具覆盖率统计

① 基金项目: CCF-深信服伏羲基金 (CCF-SANGFOR OF 2022001)
收稿时间: 2024-09-25; 修改时间: 2024-10-23; 采用时间: 2024-11-07; csa 在线出版时间: 2025-02-28

的工作[2]中, 发现仅 14% 的开发者实施了软件配置的测试验证, 其原因在于配置项及彼此关系纷繁复杂、难以梳理清楚和正确推导, 一些开发者更愿意以文档说明形式阐述如何进行配置而非对配置空间进行检测.

Linux 作为广泛应用的大型系统软件, 拥有一套成熟的配置及构建系统, 主要由 Kconfig、Kbuild 组成. Kconfig 文件位于 Linux 源码的多级目录下, 内部定义了当前路径下文件实现功能对应的 Linux 内核配置项以及配置项可用条件, Kbuild 系统由内核各目录下的 Makefile 文件组成, 负责依据配置项的值去执行内核源码构建, Linux 内核配置与最终构建的内核功能及大小密切相关[3]. 截至 Linux 内核 5.8, 内核配置项约有 15 000 项[4], 且配置项之间有多种依赖关系, 配置不当可能会导致特定功能不可用、系统崩溃等运行错误. Linux 3.3 版本在配置项 TWL4030_CORE 的直接依赖中添加了配置项 IRQ_DOMAIN, 该变动影响到原有内核对 PMIC 的支持, 导致驱动程序空指针错误[5], 相应解决方案为删除 TWL4030_CORE 对 IRQ_DOMAIN 的依赖并修改部分代码. 若针对配置错误进行检测, 则能够一定程度上避免类似问题出现在发行代码中, 提升 Linux 内核源码的可靠性. Linux 内核拥有着数以百计的发行版以及上万个配置项, 人工对全部内核配置项正确性进行验证将会是一项庞大的工作, 因此, 对 Linux 潜在的配置错误问题进行自动化检测是必要的.

近些年来, 配置分析相关研究领域也受到了中外学术界的大力关注. Sundermann 等人[6]借助可满足性问题 (satisfiability, SAT) 求解器对 Berkeley DB、Linux、BusyBox 等开源工业软件的特征模型进行检测并评估检测效率, 该工作通过读取大型软件的配置空间并将配置项的关系转换为逻辑公式提供给求解器, 统计系统中的有效配置, 该实验偏重于验证配置问题应用可满足性问题求解器的可行性及效率, 并未针对各系统配置错误的情况做检测. Oh 等人[7], 基于静态分析的思想, 实现了工具 kismet 该工具读取 Kconfig 并将配置项间的依赖关系转换为逻辑公式并进行组合, 形成配置错误情况下对应的逻辑表达式, 然后利用 Z3 求解器[8]去判断配置错误表达式能否成立. 该工具能够发现 Linux 内核中 Kconfig 中的一些依赖缺陷, 但其仅考虑了反向依赖引入错误这一种情况. 同时, 其依赖于大型求解器且对每个配置项都会进行表达式求解, 导致安装和运行时间较长. 江梦涛等人[9]基于 Linux 内核源码中的编译选项、文件和函数等之间的依赖关系提出了分层模型, 并设计了解析依赖的算法以及依赖关系的概念模型. 但该研究工作仅给出了理论设计, 且对依赖的分析专注于单个文件的分析, 没有考虑由子路径等引入的依赖关系. 中国科学院软件研究所侯朋朋等人[10], 设计了一种基于多标签的内核配置图用于表示 Linux 内核配置的依赖关系、安全属性以及对性能的影响, 并基于内核配置图实现了内核配置检索框架, 该工作重点在于对内核配置项的解析和检索, 并未实现配置错误的检测.

为此, 本文基于静态分析实现了一种 Linux 配置错误检测工具, 其能够对 Linux 内核源码的 Kconfig 文件中定义的全部类型的配置项及其依赖进行提取并转化为数据模型, 对负责构建内核的 Makefile 文件中配置项及其关联的文件或路径进行检索并推导其依赖, 进而针对 Kconfig 和 Makefile 相关配置的 4 种潜在问题进行自动检测.

1 配置分析与错误检测技术路线

配置分析与错误检测工作总体可划分为 3 个环节: (1) 依据 Linux 配置的应用和生效的原理设计检测工具的基本框架和关键流程; (2) 参照常见代码检测的技术模板设计并完善基本框架中各模块内部的具体流程; (3) 借鉴 SAT 思想构建求解器, 以支持配置错误检测.

1.1 Linux 配置项生效流程

Linux 内核配置生效过程 (即 Linux 可变性实现流程) 如图 1 所示, 具体由 Kconfig 和 Kbuild 两大模块共同完成[11]. 前者内部维护了各种配置项及彼此间的依赖关系, 并会根据用户提供的配置项设置结果及 Kconfig 定义的约束自动调整部分变量的赋值, 然后把配置结果输出到.config 文件, 调用 Linux 脚本工具转换为 Makefile 和预处理器能够读取的格式. 后者读取 Makefile 文件, 依据配置项取值决定是否编译特定源码文件或特定代码段.

本文通过对配置生效流程的研究, 发现配置错误的发生主要有 2 个方面: (1) .config 文件生成阶段, 若 Kconfig 内配置项依赖定义不当, 可能导致某些情况下.config 文件中部分配置因直接依赖未被满足, 无法在 Kbuild 系统中正常构建. (2) Kbuild 系统中的 Makefile 的执行顺序和内部语句可能间接导致配置项存在依赖, 该依赖可能与 Kconfig 系统中定义的不一致. 截至目前的研究工作仅对 (1) 进行配置错误检测, 本文将针对配

置错误的两种情况分别进行检测.

Kconfig 配置项定义示例如图 2, 每行由一个关键字开头, 声明了配置项的数据类型、默认值、依赖关系等. 绝大多数的配置项取值范围为 *y*、*m*、*n*, 其中 *y* 表示有关代码应当编译到内核, *n* 表示不编译对应代码, *m* 表示相关代码应编译为模块. 配置项的依赖关系主要分为 2 种: 直接依赖, 关键字为 depends on, 表示当前定义的配置项在关键字后的条件为真时才有效; 反向依赖, 关键字为 select, 表示当前定义的配置项为 *y* 时, 强制将关键字后的配置项值设置为 *y*.

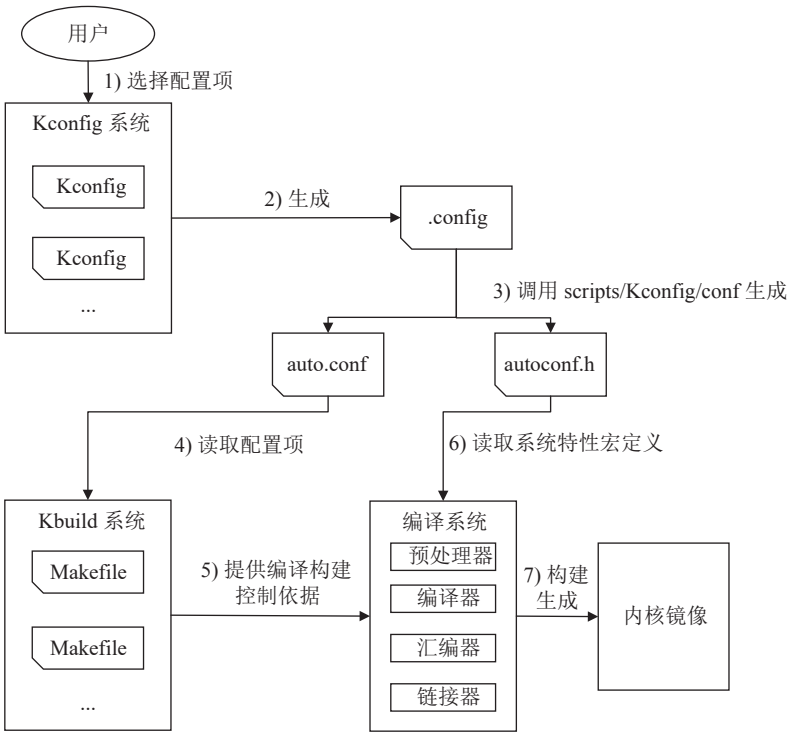

图 1 Linux 配置项生效流程

```
config NETFILTER_XT_TARGET_MARK
        tristate '"MARK" target support'
        depends on NETFILTER_ADVANCED
        select NETFILTER_XT_MARK
        help
        This is a backwards-compat option for the user's convenience
        (e.g. when running oldconfig). It selects
        CONFIG_NETFILTER_XT_MARK (combined mark/MARK module).
```

图 2 Kconfig 文件片段

Makefile 源码片段如图 3 所示. 当遇到含有配置项的语句时, 会依据配置项的值决定是否编译等号后的文件以及是否直接编译到内核.

```
ifdef CONFIG_SWAP
obj-$(CONFIG_MEMCG) += swap_cgroup.o
endif
obj-$(CONFIG_CGROUP_HUGETLB) += hugetlb_cgroup.o
obj-$(CONFIG_GUP_TEST) += gup_test.o
obj-$(CONFIG_DMAPOOL_TEST) += dmapool_test.o
```

图 3 Makefile 文件片段

参照上述 Linux 配置项生效流程, 初步设计检测所需的 3 大功能模块: Kconfig 语法解析模块、Makefile 解析模块、配置检测模块.

1.2 配置检测方法选型

静态分析技术是软件测试领域基本技术之一, 该技术通过读取代码并进行解析, 对程序执行每一种可能性进行建模, 有助于发现程序执行潜在的问题[12], 因此本文选择静态分析的方法来完成对 Kconfig、Makefile 的解析工作. 静态分析过程主要可分为 3 个阶段[13]: 路径探索、变量识别和跟踪分析. 第 1 步, 从根路径开始, 逐层向下探索, 读取需要分析的代码文件, 并建立路径与相关文件的映射; 第 2 步, 依据第 1 步读取的内容, 识别每条路径上的关键变量; 第 3 步, 依据变量的使用语句调用链、边界条件、依赖关系等分析出错的可能性.

代码检测是一种常见的评估代码的方法, 静态分析技术是实现代码检测的一项关键技术[14]. 常见的代码检测工具主要由规则检测模块、检测问题分级模块、静态分析模块构成[15]. 代码检测的一般流程如下: 首先, 调用静态分析模块对代码内容进行初步解析; 然后, 调用规则检测模块识别并记录解析后的代码是否违背规则; 最后, 调用代码分级模块将检测的问题进行

归类和分级. 本文检测工具的架构设计参考了代码检测工具的常见模板, Kconfig 和 Makefile 解析模块基于静态分析技术设计实现, 分析器模块则综合了规则检测和问题分类两大功能.

1.3 配置依赖分析方法与 SAT 求解算法

Chico 等人[6]的工作证明了配置项间的依赖关系合理性检测可以转化为求解 SAT 问题, 即判断给定的表达式是否至少存在一组赋值使得该范式可以为真值[16], 问题的输入通常为若干合取范式 (conjunctive normal form, CNF) 组合成的文件: 由若干行数字子句组成, 每个数字代表一个变量, 行之间的逻辑关系为“与”, 同一行内的数字间代表的逻辑为“或”, 数字为负代表否定, 每行公式以 0 结尾. 在本文的检测工具中, 分析器会将每个配置项转化为变量, 配置项依赖组合成的公式转换为若干子句, 生成 CNF 输入到求解器中, 判断某个配置错误问题是否可能发生.

冲突驱动的子句学习 (conflict-driven clause learning, CDCL) 是求解 SAT 问题的一种主流求解算法, 其前身是 DPLL (Davis-putnam-logemann-loveland) 算法. DPLL 基于深度搜索思想[17], 不断地去选择未被赋值的变量进行赋值直到变量的赋值出现冲突, 冲突发生后, 回溯到最近一个未被修改过值的变量处, 将该变量的值翻转后继续向下搜索. CDCL 在 DPLL 的基础上增加了子句学习机制[18], 在冲突发生时, 会依据冲突发生涉及的子句和变量, 生成一个新的学习子句加入子句集合中, 并依据子句中涉及变量的跨度决定回溯位置. Kconfig 直接依赖和反向依赖语句中配置项数目通常少于 5 个, 因此配置项问题与一些常见的 SAT 问题如电路问题[19]和资源分配问题[20]相比规模相对较小, 每个独立的配置依赖错误问题涉及的变量空间也相对较小, 引入使用了 CDCL 算法的求解器, 能有效提升执行效率且不会产生过多内存开销.

2 配置错误检测工具

2.1 总体设计

配置检测工具的架构见图 4, 主要由 4 个模块构成: Kconfig 解析模块、Makefile 解析模块、分析器、SAT 求解器. 对配置错误问题的检测可划分为 3 个环节: 测试路径探索、关键变量识别、错误分析, 这些环节由检测模块共同完成.

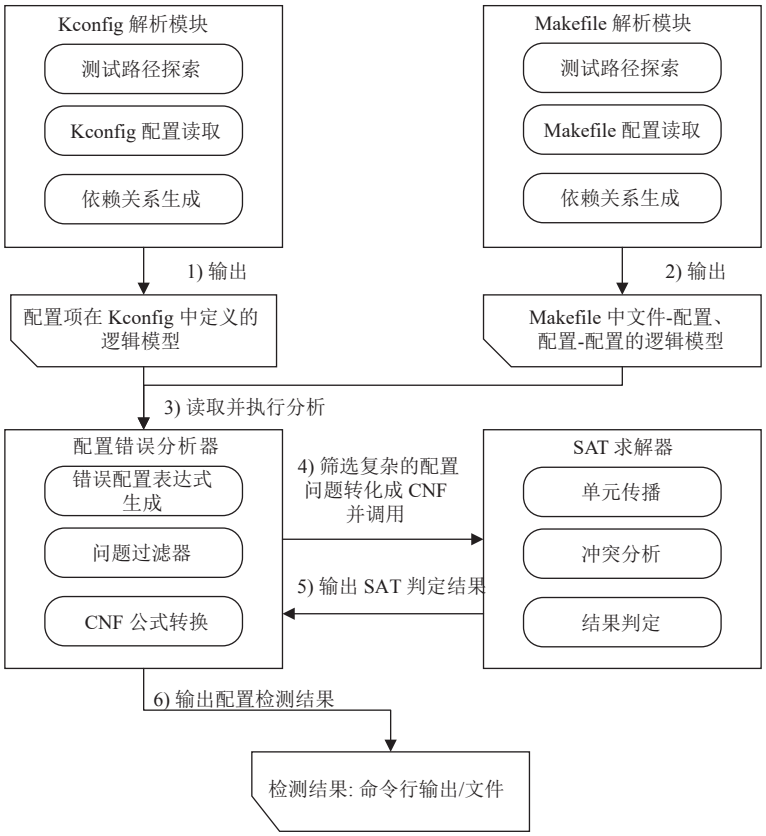

图 4 检测工具框架与执行流程

首先, 测试路径探索阶段, Kconfig 解析模块和 Makefile 解析模块分别从源码的根目录逐层向下搜索 Kconfig 文件和 Makefile 文件. 其次, 在关键变量识别阶段, Kconfig 解析模块从路径探索阶段中收集的 Kconfig 文件内容中识别出配置项定义和配置项之间的多种约束关系, 保存为逻辑模型, Makefile 解析模块从已经收集 Makefile 中识别出配置项与文件的映射, 并推理配置项间的依赖并保存. 在最终分析阶段, 分析器会依据几种不同的配置错误情况, 组装关键变量识别阶段得到的 Kconfig 配置项与 Makefile 配置项的逻辑表达式, 并执行问题过滤, 即初步判断对应配置错误情况是否可能出现, 若不可能出现配置错误, 则会跳过当前配置项, 反之则将当前配置问题转化为 CNF 公式作为 SAT 求解器模块的输入, 检测工具对配置错误问题进行分类后汇总, 输出最终的检测结果.

与过往检测工具解析后直接调用分析程序的流程设计相比, 本文基于配置项的生效流程引入了两个独立的解析模块, 分别解析配置项的定义与配置项对应文件的构建两个环节中涉及的配置项 (配置读取环节完成) 并推理配置项间依赖 (依赖生成环节完成), 有助于发现多种类型的配置错误问题. 分析器在表达式生成环节为每个待检测配置项生成 4 类配置错误问题对应的表达式, 并分别进行运算, 且在生成表达式和进行配置错误检测流程中额外增加了过滤流程, 该流程将会识别依赖约束不会触发配置错误的配置项, 并跳过 SAT 求解器环节, 跳过不必要的运算.

2.2 Kconfig 解析模块

KconfigParser 是 Kconfig 的解析模块, 提供了探索测试路径和解析 Kconfig 文件的功能, 解析模块的执行流程可分为测试路径探索 (finder)、配置读取 (reader) 与依赖生成 (generator) 这 3 个子流程. 算法 1 描述了测试路径探索器 finder 的执行逻辑: 自源码的最顶层目录开始, 遍历当前目录下的文件列表, 对不同的文件进行不同处理.

测试路径探索算法设计如算法 1.

算法 1. 测试路径探索算法

1) 判断当前路径是否为屏蔽路径, 若为屏蔽路径则返回上级路径.
2) 遍历当前路径下的所有文件 (包含子路径).
3) 如果当前文件为子路径, 将将该路径加入当前路径下的待搜索列表中.
4) 若当前文件为 Kconfig 文件 (包含“Kconfig.后缀”等特殊格式的文件), 将该文件的绝对路径加入待检测列表中.
5) 遍历当前路径下待搜索列表中的全部子路径.
6) 递归调用路径探索算法, 将子路径作为当前路径, 执行步骤 1).

接下来, 配置读取器 reader 会读取上一环节收集到的文件并按 Kconfig 语法进行解析, reader 使用栈保存当前配置的前置依赖 (即没有在 depends on 中直接定义, 但因为 if、source 关键字的出现而在前文间接定义的依赖), 当识别到以 if 开头的语句时, 将 if 后的配置项加入依赖栈中. reader 中解析到的配置项信息由 ConfigSymbol 类保存, 该类中有配置名称、配置类型、配置默认值成立条件、配置直接依赖、配置反向依赖字段, 在读取到配置的依赖关系时, 若前置依赖不为空, 则将前置依赖加入配置的直接依赖中.

配置读取算法设计如算法 2.

算法 2. Kconfig 配置读取算法

1) 逐行读取 Kconfig 文件, 并识别每行关键字.
2) 关键字为 config 或 menuconfig, 新建一个 ConfigSymbol 对象, 并保存配置名称, 将定义的配置加入到当前 Kconfig 文件对应的配置集合中.
3) 关键字为 if, 将 if 后的配置项放入前置依赖栈中.
4) 关键字为 endif, 弹出前置依赖栈最顶层的配置.
5) 关键字为 depends on, 获取当前正在解析的配置项的 ConfigSymbol 对象, 将前置依赖栈内容和关键字后的配置项加入直接依赖字段中.
6) 关键字为 select, 获取当前正在解析的配置项的 ConfigSymbol 对象, 将 select 后的变量及其 select 条件加入反向依赖字段中.
7) 关键字为 source, 构建 source 后跟随的 Kconfig 文件路径将当前依赖栈当前内容并保存.

最后, 依赖生成器 generator 会依据 ConfigSymbol 集合中的内容, 拼接每个配置项依赖关系的公式, 并依据算法 3 构建依赖表达式. 对一个配置项, 其直接依赖为 Kconfig 中 depends on 语句的全部配置项以及 if 条件语句、所在 Kconfig 被执行 source 操作的条件语句进行“与”逻辑运算. 配置项的反向依赖由若干个由逻辑“或”连接的子句 (每行 select 语句对应一个子句) 组成. select 语句转换为子句的逻辑为: 当前 select 关键字后跟随的配置项及该配置项后跟随的 if 条件直接进行“与”运算生成的表达式. 默认值成立的条件由配置项的直接依赖条件与默认条件用“与”逻辑连接.

依赖生成算法设计如算法 3.

算法 3. Kconfig 依赖生成算法

1) 遍历配置读取环节识别到的 ConfigSymbol 列表中的每个配置项.
2) 若当前配置项来源 Kconfig 通过 source 关键字生效, 则读取当前 source 生效的对应条件按 AND 逻辑连接到当前配置项对应的直接依赖条件.

3) 遍历当前配置项的反向依赖.
4) 针对单条反向依赖, 将被 select 的配置项及其条件按 AND 逻辑连接, 生成单条子句.
5) 将全部的反向依赖语句使用 OR 连接.
6) 若配置项存在默认值, 将默认值成立条件和直接依赖使用 AND 连接, 若默认值为非布尔值, 新建一个形如“当前配置项_EQ_默认值”的布尔配置项, 并将该配置项的直接依赖条件设置为当前默认值成立条件.

本文提出了一种新型解析非布尔变量及相关语句的方法: (1) 在配置读取阶段, 非布尔变量的默认值保存在其 ConfigSymbol 对象内部的一个 Map 结构中. (2) 在依赖生成阶段, 语句中遇到形如“配置项==值”语句时, 将该表达式替换为一个布尔变量配置项_EQ_值, 若配置项对应的默认值映射中存在该值, 再将这个变量替换为默认值成立条件.

2.3 Makefile 解析模块

MakefileParser 是 Makefile 的解析模块, 与 Kconfig 解析模块类似, 由 Makefile 语法的 finder、reader 和 generator 组成, finder 负责搜索 Makefile 的分布路径, reader 和 generator 负责解析 Makefile 文件的具体内容, 并依据语法分析出配置项间的依赖关系. MakefileParser 的 finder 模块与第 3.1 节中的 finder 实现方法类似, 自顶向下识别全部 Makfile 文件, 遇到屏蔽的文件夹时跳过, 最后返回给 reader 全部需要识别的 Makefile 文件.

MakefileParser 的 reader 算法也设置了一个依赖栈, 这是因为 ifdef 和 ifeq 或者 ifneq 关键字后可能会跟随一个配置项或配置项值判定语句, 因此当碰到 ifdef 等关键字时, 要把关键字后面的条件入栈, 并在遇到 endif 时弹出栈顶. 在 reader 中定义类 MakefileSpace, 每一个 Makefile 文件对应一个该类的对象, 该类关键字段有: 所在路径、涉及的文件集合、涉及的配置项集合、文件与配置项逻辑公式的 Map. 由于 Makefile 中配置项的依赖不是直接定义于文件中而是通过间接推导的, 因此还要维护一个公共的 Map 保存所有文件(包括路径) 关联的配置逻辑表达式, 便于 generator 推导依赖关系.

配置读取算法设计如下算法 4.

算法 4. Makefile 配置读取算法

1) 逐行读取 Makefile 文件, 并识别编译选项语句行.
2) 若文件行以 obj-y 开头, 按:=切分出文件并存储<文件, 依赖集合>.
3) 若文件行以模块名称或 obj-$CONFIG_开头, 按-$切分出配置项, 按:=切分出配置项对应文件, 将当前配置项和当前依赖栈内容加入一个新集合中, 存储<文件, 依赖集合>.
4) 若文件行以 ifdef 开头, 则将 ifdef 后定义的配置项加入依赖栈.
5) 若文件行以 ifeq 开头, 如果 ifeq 后跟随的值是两个配置项则将两个配置项同或条件加入依赖栈, 若跟随的是配置项与 *y*/*n*, 则依据值将配置项的肯定/否定形式加入依赖栈.
6) 若文件行以 ifneq 开头, 如果后面跟随的是两个配置项, 则将两个配置取异或逻辑加入依赖栈, 若跟随的是配置项与 *y*/*n*, 则依据值将配置项的否定/肯定形式加入依赖栈.
7) 若文件以 endif 开头, 弹出依赖栈.
8) 若文件以 obj-$(subst 开头, 保存对应配置项及默认可替换的值, 并记录该语句.

Makefile 配置项的依赖主要是依据路径来推导的, 我们在解析 Makefile 时, 会遇到形似 obj-$CONFIG_A:=文件路径的情况, 而这个文件路径下也存在着 Makefile 文件, 此时由于路径间接地形成了依赖关系. generator 算法识别这种依赖并构成配置项依赖关系, 其核心流程是遍历全部识别到的文件, 并将文件全部上级目录的依赖加入文件对应的依赖集合中.

2.4 配置错误分析器

配置错误分析器接收的是解析到的在 Kconfig 空间和 Makefile 的配置项及其依赖关系, 并将这些依赖条件进行组合, 形成配置错误对应的逻辑表达式, 并判定该表达式是否成立. 配置错误分析器识别的错误类型主要有: 直接依赖缺失错误、直接依赖未满足错误、Kconfig 与 Makefile 依赖不一致错误以及 Kconfig 缺失 Makefile 配置项定义错误.

针对直接依赖缺失类错误, 分析器会收集每个存在依赖的配置项的直接依赖条件, 并读取所依赖配置项的依赖条件, 若配置不存在直接依赖条件, 则会读取反向依赖, 如果均不存在, 则会读取其默认值成立条件, 若不存在默认值则直接判定为隐形配置依赖错误, 若存在相关依赖, 将这些依赖进行“与”操作后生成 CNF 公式, 再进入 SAT 求解器进行求解. 针对直接依赖未满足类错误, 分析器会读取全部含有反向依赖的配置项, 对符合条件的配置项, 先取其直接依赖和配置项本身进行“与”操作 , 然后找到反向依赖选择的配置项的直接依赖, 将其直接依赖取否定逻辑, 然后将逻辑表达式进行“与”操作. 针对 Kconfig 与 Makefile 依赖不一致类错误, 分析器对 Makefile 空间下的配置项逐一去找到其在 Kconfig 中定义的直接依赖和反向依赖, 将 Makefile 分析出的依赖公式与 Kconfig 中的直接依赖

与反向依赖的否定形式分别取“与”逻辑, 进行 SAT 求解的判定. 针对 Kconfig 缺失 Makefile 配置项定义类错误, 分析器会检查 Makefile 空间中的每个配置项, 验证其在 Kconfig 中是否没有定义或是否为存在依赖错误, 满足二者条件之一即判定为配置缺失.

分析器的过滤机制能够提升执行效率, 相当一部分存在直接依赖或反向依赖的配置项间存在着共同的依赖项, 这种情况下二者的依赖关系不可能会引入错误, 分析器在逐个读取依赖中的配置项时, 会将这种情况直接识别出来, 避免进入求解器环节造成不必要的运行开销.

2.5 SAT 求解器

分析器将配置项及依赖转换为 CNF 子句, 一个配置项对应一个变量. 求解器基于 CDCL 算法实现, 通过接收到分析器传递来的 CNF 公式后, 求解 SAT 问题, 求解器内部可再划分为如下关键子模块: 初始化模块、变量选择模块、单元传播模块、冲突处理模块. 求解器算法维护执行过程中的一些数据结构: 最基本的是变量数组和子句数组, 变量数组保存当前全部变量赋值, 为了优化变量的选择和赋值策略, 用两个数组分别来保存变量的正负性 (值为变量在子句中以肯定形式存在次数与以否定形式存在次数之差) 和变量以肯定形式存在的次数, 为了便于学习冲突和快速找到单元变量, 求解器维护变量与其前因 (即限制变量只能为当前值的条件) 的映射表.

本文设计的求解器算法的工作流程是: 读取由配置项及其依赖转化的变量集合和公式并进行初始化后, 循环执行以下流程至求解成功或被判断为无解: (1) 执行单元传播, 即找出当前全部的值能够唯一确定的变量并进行赋值. (2) 重新遍历所有含有未赋值变量的子句, 如果不再存在单元子句, 则优先选择变量出现频率最高的并依据其正负性进行赋值. (3) 冲突判断与处理, 当存在无法同时满足的子句时, 检查内部已经赋值的每个变量, 若变量存在前因, 则将其前因与冲突子句进行“与”逻辑连接, 最终将学习到的新子句加入子句集合中, 撤回所有层级深于当前冲突层的变量.

3 实验结果与分析

本次实验基于 Linux 6.7 版本的内核源码开展, 目的是验证配置检测工具的检测能力, 配置检测工具会自动执行整个测试流程, 首先调用 Kconfig 解析模块、Makefile 解析模块解析配置的相关文件, 并导出配置项的依赖结果, 接下来分析器依据配置的依赖自动执行问题生成流程并调用求解器进行判断.

3.1 Kconfig 与 Makefile 解析器执行结果

Kconfig 解析输出结果示例见图 5、图 6, 主要包含配置项定义、默认值及成立条件、直接依赖、反向依赖.

```
config CONFIG_WATCHDOG_HANDLE_BOOT_ENABLED bool
prompt CONFIG_WATCHDOG_HANDLE_BOOT_ENABLED (CONFIG_WATCHDOG)
def_bool CONFIG_WATCHDOG_HANDLE_BOOT_ENABLED 1|(CONFIG_WATCHDOG)
config CONFIG_DRM_AMDGPU_WERROR bool
prompt CONFIG_DRM_AMDGPU_WERROR (CONFIG_HAS_IOMEM and CONFIG_DRM_AMDGPU and CONFI
def_bool CONFIG_DRM_AMDGPU_WERROR 0|(CONFIG_HAS_IOMEM and CONFIG_DRM_AMDGPU and C
config CONFIG_SBUS bool
config CONFIG_EXTCON_SM5502 tristate
prompt CONFIG_EXTCON_SM5502 (CONFIG_EXTCON and CONFIG_I2C)
config CONFIG_ARM_OMAP2PLUS_CPUFREQ bool
```

图 5 Kconfig 解析结果片段展示: 配置项定义和默认值

```
dep CONFIG_INTERCONNECT_QCOM_SDM670 (CONFIG_INTERCONNECT and CONFIG_INTERCONNECT_QC
select CONFIG_INTERCONNECT_QCOM_RPMH CONFIG_INTERCONNECT_QCOM_SDM670 (CONFIG_INTERC
select CONFIG_INTERCONNECT_QCOM_BCM_VOTER CONFIG_INTERCONNECT_QCOM_SDM670 (CONFIG_I
dep CONFIG_CLK_R8A779G0 (CONFIG_COMMON_CLK and CONFIG_CLK_RENESAS)
select CONFIG_CLK_RCAR_GEN4_CPG CONFIG_CLK_R8A779G0 (CONFIG_COMMON_CLK and CONFIG_C
rev_dep CONFIG_CLK_R8A779G0 (CONFIG_CLK_RENESAS and CONFIG_COMMON_CLK and CONFIG_AR
dep CONFIG_RTW89_8852C (CONFIG_NETDEVICES and CONFIG_WLAN and CONFIG_WLAN_VENDOR_RE
rev_dep CONFIG_RTW89_8852C (CONFIG_RTW89_8852CE and CONFIG_NETDEVICES and CONFIG_WL
dep CONFIG_INTEL_VBTN (CONFIG_X86_PLATFORM_DEVICES and CONFIG_ACPI and CONFIG_INPUT
```

图 6 Kconfig 解析结果片段展示: 直接依赖、反向依赖

Makefile 解析器的执行结果片段图 7, 第 1 行输出为构建内核时可能编译到的文件, 第 2 行是该文件或该文件关联配置项涉及的依赖.

```
文件: linux-6.7/kernel/bpf/reuseport_array.
[CONFIG_BPF_SYSCALL, CONFIG_INET]
文件: linux-6.7/kernel/time/tick-broadcast-hrtimer.
[CONFIG_TICK_ONESHOT, CONFIG_GENERIC_CLOCKEVENTS_BROADCAST]
文件: linux-6.7/drivers/memory/of_memory.
[CONFIG_OF, CONFIG_DDR]
文件: linux-6.7/drivers/sh/clk
[CONFIG_HAVE_CLK, -CONFIG_COMMON_CLK]
文件: linux-6.7/drivers/usb/cdns3/cdns3-trace.
[CONFIG_USB_CDNS3_GADGET, CONFIG_TRACING]
```

图 7 Makefile 解析结果片段

3.2 直接依赖缺失类错误检测结果

直接依赖缺失类错误即对于一个配置项 A, 在其定义语句中 depends on 关键字定义的逻辑公式中的部分配置项不存在定义或无法使得 depends on 的判断条件成立, 该问题的直接后果是: 配置的设置者对该配置项的任何赋值均无效. 直接依赖缺失的检测部分结果如图 8, 在检测中发现了一些由架构问题引起的直接依赖缺失问题, 如: INLINE_SPIN_LOCK_BH 是一个与 Linux 自旋锁中断相关机制的配置项, 该配置定义及依赖位于 kernel 目录下, 其默认值为 *y* 且依赖于配置项 ARCH_INLINE_SPIN_LOCK_BH 和 GENERIC_LOCKBREAK 的否定形式, 而 ARCH_INLINE_SPIN_LOCK_BH 只在 s390、arm64、csky、loongarch 架构

中才会被设置为 *y*, INLINE_SPIN_LOCK_BH 才可能实际生效, 因此该配置项在其他架构下时会因为缺失直接依赖而变成值为 *y* 但不生效的配置错误. 该配置项同时在还存在另一个问题, 在 loongarch 架构下, 不存在其依赖中 GENERIC_LOCKBREAK 的定义, 因此用户无法感知到 loongarch 支持配置项 INLINE_SPIN_LOCK_BH, 与配置设计的预期不符. 对于这种不同架构对配置项支持能力不同的问题的解决方案有 2 种: ① 在配置项的默认值后增加 if 条件关联架构相关配置项; ② 将配置项定义在指定架构下的 Kconfig 文件中.

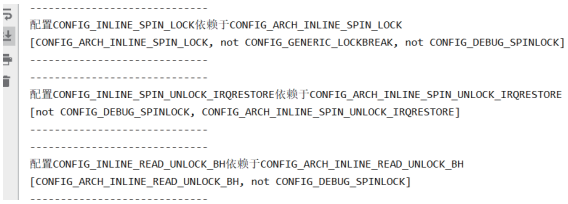

图 8　直接依赖缺失检测结果片段

3.3　直接依赖未满足类错误检测结果

直接依赖未满足类错误由反向依赖引入, 即某个配置项 A 被另一个配置项 B 使用 select 语句强制选中赋值为 *y*/*m* 时, 配置项 A 的直接依赖未被满足, 导致虽然将配置 A 强制赋值为 *y*, 但是 A 并未实际生效, 与第 3.2 节直接依赖缺失问题的区别在于产生的原因, 直接依赖未满足由配置项被 select 强制选中引发, 被选中的配置项 A 的依赖条件在一定情况下仍可以被满足.

检测运行片段如图 9, 输出的第 1 行表示 Kconfig 中存在着 select SND_SOC_IMX_RPMSG 的语句会引发直接依赖未满足问题, 第 2 行为被 select 配置项的直接依赖, 第 3 行为 SND_SOC_IMX_RPMSG 的反向依赖, 若配置在多处 Kconfig 中被 select, 则多条 select 语句的成立子句则会用“or”分隔, 每个子句的第 1 个变量即定义了 select 当前配置的变量. 结果显示, SND_SOC_FSL_RPMSG 对 SND_SOC_IMX_RPMSG 执行 select 操作时可能导致配置问题.

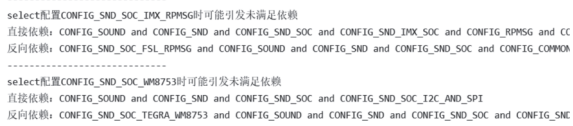

图 9　直接依赖未满足检测结果片段

SND_SOC_FSL_RPMSG 在 sound/soc/fsl/Kconfig 中的定义片段如图 10.

```
config SND_SOC_FSL_RPMSG
        tristate "NXP Audio Base On RPMSG support"
        depends on COMMON_CLK
        depends on RPMSG
        depends on SND_IMX_SOC || SND_IMX_SOC = n
        select SND_SOC_IMX_RPMSG if SND_IMX_SOC != n
        help
```

图 10　SND_SOC_FSL_RPMSG 定义片段

SND_SOC_IMX_RPMSG 在 sound/soc/fsl/Kconfig 中的定义片段如图 11.

```
config SND_SOC_IMX_RPMSG
        tristate "SoC Audio support for i.MX boards with rpmsg"
        depends on RPMSG
        depends on OF && I2C
        select SND_SOC_IMX_PCM_RPMSG
        select SND_SOC_IMX_AUDIO_RPMSG
        help
```

图 11　SND_SOC_IMX_RPMSG 定义片段

SND_SOC_IMX_RPMSG 依赖于配置 OF, 该配置位于 drivers/of/Kconfig 文件中, 是一个与设备树相关的配置. 由 sound/soc/fsl/Kconfig 中的配置定义发现, 并无 select 语句等约束 OF 在 SND_SOC_IMX_RPMSG 和 SND_SOC_FSL_RPMSG 为 *y* 时的赋值, 且 OF 在 make menuconfig 时可以由用户自由选择, 若用户选中了 SND_SOC_FSL_RPMSG 却关闭了 OF, 则会引发直接依赖未满足问题.

3.4　Kconfig 与 Makefile 依赖不一致类错误检测结果

Kconfig 与 Makefile 依赖不一致类错误即针对同一配置项, 在 Kconfig 中定义的依赖关系与在 Kbuild 系统中 Makefile 中依据构建的实际顺序和关键字定义产生的依赖关系不一致的问题, 会导致用户看到的配置项与其配置项实际的构建结果不一致: 用户设置了某配置期望能够编译某段代码, 但实际执行时依赖的配置项未被满足; 或用户以为某个配置项不满足一定条件时不能被设置和使用, 但是在构建过程中并未依赖该条件.

检测的部分运行结果如图 12, 结果输出以虚线分隔为若干小节，譬如第 2 节的第 1 行 (虚线本身) 显示了 XFS_QUOTA 被检测出 Makefile 与 Kconfig 不一致的问题, 第 2 行输出 Kconfig 约束中定义的 XFS_FS 的直接依赖, 第 3 行输出 Makefile 中推导出的直接依赖.

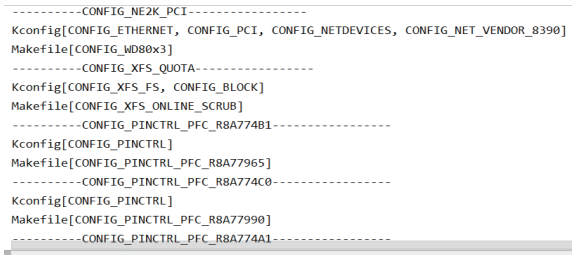

图 12　Kconfig 与 Makefile 依赖不一致检测结果片段

XFS_QUOTA 在 fs/xfs/Kconfig 中的依赖定义如图 13.

```
config XFS_QUOTA
        bool "XFS Quota support"
        depends on XFS_FS
        select QUOTACTL
        help
```

图 13 XFS_QUOTA 定义片段

XFS_FS 与 XFS_QUOTA 位于同一个文件中, 定义片段如图 14.

```
config XFS_FS
        tristate "XFS filesystem support"
        depends on BLOCK
        select EXPORTFS
        select LIBCRC32C
        select FS_IOMAP
        help
```

图 14 XFS_FS 定义片段

在 fs/xfs/Makefile 文件下, 描述了 XFS 相关配置项在构建阶段的使用. 第 142 行为 ifeq ($(CONFIG_XFS_ONLINE_SCRUB), *y*). 第 178 行为 xfs-$(CONFIG_XFS_QUOTA) += scrub/quota.o. 在第 178 行时, 第 142 行的判断还未使用 endif 结束, 因此推导 XFS_QUOTA 依赖 XFS_ONLINE_SCRUB, 而在 Kconfig 的相关定义中, 没有使用直接依赖或反向依赖来约束 XFS_ONLINE_SCRUB 和 XFS_QUOTA 为 *y*.

3.5 Kconfig 缺失 Makefile 配置项定义类错误检测结果

Kconfig 缺失 Makefile 配置项定义类错误, 即 Makefile 中存在配置项作为某文件/文件夹执行编译的条件, 而该配置项却没有在 Kconfig 中被定义, 这种问题会导致配置项对应的代码永远无法被编译, 若该配置项被其他配置项依赖, 则会产生连锁反应, 影响更多配置项的实际编译效果, 这对于 Linux 内核而言是一个较为严重的错误. 检测结果显示, 在所测试的 Linux 6.7 版本下, 不存在配置缺失的问题, 为了验证工具对配置缺失问题检测的能力, 对 Linux 代码进行了改造, 分别在 net、drivers、kernel、mm 目录下的 Makefile 中插入 4 条关联配置项的编译选项, 插入语句为: obj-$(CONFIG_TESTCASE_ONE) += testcaseone.o.

检测结果如图 15, 虽然现有发行版代码中不存在该项问题, 但该检测可以作为 Linux 内核源码定制化开发过程中一条代码质量评估标准.

3.6 实验结果统计分析

本文配置检测原型的检测实验结果见表 1. 就 Linux 6.7 版内核源码而言, 可检出原本存在且有效的直接依赖缺失问题 29 项、直接依赖未满足问题 9 项、依赖不一致问题 14 项. 除此之外, 本文还人为地设置了 4 处配置缺失问题, 并被该工具原型全部检出.

```
配置无定义CONFIG_TESTCASE_TWO文件linux-6.7/mm/testcasetwo.
配置无定义CONFIG_TESTCASE_ONE文件linux-6.7/net/testcaseone.
配置无定义CONFIG_TESTCASE_FOUR文件linux-6.7/kernel/testcasefour.
配置无定义CONFIG_TESTCASE_THREE文件linux-6.7/drivers/testcasethree.
```

图 15 Kconfig 缺失 Makefile 配置项定义检测结果片段

表 1 潜在配置错误检测统计结果

问题类型	检出数	确认问题数	准确率 (%)
直接依赖缺失	45	29	64.44
直接依赖未满足	18	9	50.00
Kconfig与Makefile配置依赖不一致	24	14	58.33
Kconfig缺失Makefile配置项定义	4	4	100.00

现有开源的配置错误检测工具相对缺乏, 且仅覆盖了直接依赖未满足这一配置错误问题, 本文在 Linux 6.7 版本内核源码上执行了现有工具 kismet[7], 该工具聚焦于 Kconfig 中由反向依赖引入的直接依赖未满足错误的检测, kismet 和本文原型的检测范围对比见表 2. 关于直接依赖未满足问题的检测对比见表 3, kismet 发现的 3 项问题中, 经验证确认的 2 项错误 (1 项为误检) 同样被本文原型检出.

表 2 配置错误问题检测范围对比

问题类型	kismet	本文原型
直接依赖缺失	不支持	支持
直接依赖未满足	支持	支持
Kconfig与Makefile配置依赖不一致	不支持	支持
Kconfig缺失Makefile配置项定义	不支持	支持

表 3 直接依赖未满足问题检测结果

工具	检出数	实际错误占比 (%)
kismet	3	66.67
本文原型	18	50.00

4 结论与展望

本文研究 Linux 配置分析与错误检测的方法及技术路线, 设计实现相应的原型系统, 并进行了实验验证. 结果表明, 本文方法及原型系统能够发现比现有其他工具更多类型的配置错误, 特别是 Kconfig 与 Makefile 配置依赖不一致的问题. 对于现有工具也能发现的由反向依赖引发的直接依赖未满足问题, 本文方法及原型系统能够检出更多的并被确认的配置错误. 本文方

法和原型系统可以为用户配置内核和开发者定制内核提供辅助支持. 本文工作的局限性包括: (1) 对配置依赖的解析仅局限于 Linux 中的 Kconfig 和 Makefile, 而 Linux 代码文件中也存在着一些与配置项关联的代码段, 针对此类配置的检测未能覆盖. (2) 在直接依赖未满足的检测中, 误检产生的一个主要原因是多个配置项的间接多次反向依赖约束了被选择变量的直接依赖能够满足, 检测工具目前未能够将这种情况排除. 下一步研究工作重点可放在对内核 C 源码配置项的分析以及工具检测准确率提升等方向.

参考文献

1 Arya A, Gupta P, Singhal S, *et al*. Software systems using variability approaches. Proceedings of the 5th International Conference on Inventive Research in Computing Applications (ICIRCA). Coimbatore: IEEE, 2023. 809–813.

2 Horcas JM, Pinto M, Fuentes L. Empirical analysis of the tool support for software product lines. Software and Systems Modeling, 2023, 22(1): 377–414. [doi: 10.1007/s10270-022-01011-2]

3 Acher M, Martin H, Lesoil L, *et al*. Feature subset selection for learning huge configuration spaces: The case of linux kernel size. Proceedings of the 26th ACM International Systems and Software Product Line Conference—Volume A. Graz: ACM, 2022. 85–96.

4 Martin H, Acher M, Pereira JA, *et al*. Transfer learning across variants and versions: The case of Linux kernel size. IEEE Transactions on Software Engineering, 2022, 48(11): 4274–4290. [doi: 10.1109/TSE.2021.3116768]

5 Abal I, Melo J, Stanciulescu S, *et al*. Variability bugs in highly configurable systems: A qualitative analysis. ACM Transactions on Software Engineering and Methodology, 2017, 26(3): 10.

6 Sundermann C, Heß T, Nieke M, *et al*. Evaluating state-of-the-art # SAT solvers on industrial configuration spaces. Empirical Software Engineering, 2023, 28(2): 29. [doi: 10.1007/s10664-022-10265-9]

7 Oh J, Yıldıran NF, Braha J, *et al*. Finding broken Linux configuration specifications by statically analyzing the Kconfig language. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens: ACM, 2021. 893–905.

8 de Moura L, Bjørner N. Z3: An efficient SMT solver. Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Budapest: Springer, 2008. 338–340. [doi: 10.1007/978-3-540-78800-3_24]

9 江梦涛, 潘朋飞, 宋杨, 等. Linux 内核中编译选项、文件以及函数之间依赖关系的解析方法. 计算机科学, 2014, 41(z1): 445–454.

10 侯朋朋, 张珩, 武延军, 等. 基于多标签的内核配置图及其应用. 计算机研究与发展, 2021, 58(3): 651–667. [doi: 10.7544/issn1000-1239.2021.20200186]

11 Dietrich C, Tartler R, Schröder-Preikshat W, *et al*. Understanding Linux feature distribution. Proceedings of the 2012 Workshop on Modularity in Systems Software. Potsdam: ACM, 2012. 15–20.

12 Mordahl A. Automatic testing and benchmarking for configurable static analysis tools. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. Seattle: ACM, 2023. 1532–1536.

13 Zhan DY, Yu XZ, Zhang HL, *et al*. ErrHunter: Detecting error-handling bugs in the Linux kernel through systematic static analysis. IEEE Transactions on Software Engineering, 2023, 49(2): 684–698. [doi: 10.1109/TSE.2022.3160155]

14 Trautsch A, Erbel J, Herbold S, *et al*. What really changes when developers intend to improve their source code: A commit-level study of static metric value and static analysis warning changes. Empirical Software Engineering, 2023, 28(2): 30. [doi: 10.1007/s10664-022-10257-9]

15 Lenarduzzi V, Pecorelli F, Saarimaki N, *et al*. A critical comparison on six static analysis tools: Detection, agreement, and precision. Journal of Systems and Software, 2023, 198: 111575. [doi: 10.1016/j.jss.2022.111575]

16 庞立超, 王晓峰, 谢志新, 等. 随机正则 3-可满足性问题的解簇结构分析. 计算机应用, 2024, 44(7): 2137–2143.

17 Alouneh S, Abed S, Al Shayeji MH, *et al*. A comprehensive study and analysis on SAT-solvers: Advances, usages and achievements. Artificial Intelligence Review, 2019, 52(4): 2575–2601. [doi: 10.1007/s10462-018-9628-0]

18 Audemard G, Laurent S. On the glucose SAT solver. International Journal on Artificial Intelligence Tools, 2018, 27(1): 1840001. [doi: 10.1142/S0218213018400018]

19 Schneider S, Burgholzer L, Wille R. A SAT encoding for optimal Clifford circuit synthesis. Proceedings of the 28th Asia and South Pacific Design Automation Conference. Tokyo: IEEE, 2023. 190–195.

20 Liu R, Guo KF, Zhou FH, *et al*. Resource allocation for NOMA-enabled cognitive satellite-UAV-terrestrial networks with imperfect CSI. IEEE Transactions on Cognitive Communications and Networking, 2023, 9(4): 963–976. [doi: 10.1109/TCCN.2023.3261311]

(校对责编: 王欣欣)