计算机科学
COMPUTER SCIENCE

流敏感的C/C++程序编程风格检查方法

胡梦泽, 马旭桐, 张豪, 张健

引用本文

胡梦泽, 马旭桐, 张豪, 张健. 流敏感的C/C++程序编程风格检查方法[J]. 计算机科学, 2025, 52(6): 35-43.

HU Mengze, MA Xutong, ZHANG Hao, ZHANG Jian. Flow-sensitive Coding Style Checking for C/C++ Programs [J]. Computer Science, 2025, 52(6): 35-43.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

针对AIGC数字插画设计原则的用户评价指标分析

Analysis of User Evaluation Indicator for AIGC Digital Illustration Design Principles

计算机科学, 2024, 51(11): 47-53. https://doi.org/10.11896/jsjkx.240700085

自动推理技术在求解组合数学难题中的研究进展

Automated Reasoning Techniques for Solving Combinatorial Mathematical Problems:A Survey

计算机科学, 2023, 50(7): 167-175. https://doi.org/10.11896/jsjkx.221000251

基于图神经网络的软件系统中关键类的识别

Identification of Key Classes in Software Systems Based on Graph Neural Networks

计算机科学, 2021, 48(12): 149-158. https://doi.org/10.11896/jsjkx.210100200

语义区域风格约束下的图像合成

Image Synthesis with Semantic Region Style Constraint

计算机科学, 2021, 48(2): 134-141. https://doi.org/10.11896/jsjkx.200800201

一种面向移动机会网络的能效性路由算法

Energy Efficient Routing Algorithm in Mobile Opportunistic Networks

计算机科学, 2019, 46(11A): 387-392.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240300195

流敏感的C/C++程序编程风格检查方法

胡梦泽[1,2,3] 马旭桐[1,3] 张 豪[1,3] 张 健[1,3]
1 中国科学院软件研究所计算机科学国家重点实验室 北京 100190
2 国科大杭州高等研究院 杭州 310012
3 中国科学院大学 北京 100049
(humz@ios.ac.cn)

摘 要 C/C++编程语言应用于众多关键领域的软件系统中,在开发时对编程的规范性和语义的明确性有着极高的要求。为了避免不当使用C/C++语言带来潜在的安全问题,国内提出了面向C/C++语言的《C/C++语言编程安全子集》(后简称为GJB8114)。由于标准中规则较多,程序员在开发过程中难免存在不规范的写法,因此实现用自动化的规则检测工具检测相应的编码规则是必要的。而现有的编程规范检查工具对标准的检查并不全面,在针对需要理解程序上下文的规则的检查时,误报率较高甚至不支持检查。对此,将GJB8114中的规则分类并对复杂规则进行定义。通过调研Testbed工具检查GJB8114中的复杂规则的支持程度,总结得到现有工具存在流敏感分析不完善和无法进行跨文件的全局分析这两个问题。采取了结合语法树匹配的流敏感分析方法和跨文件的全局分析方法来解决这些问题。在此基础上,开发得到了CruletFS工具。实验结果表明,与常见的编程规范检查工具Cppcheck,Testbed等相比,CruletFS在检查复杂规则时有更好的效果。在针对规模较大的项目分析时,CruletFS在开销上也优于Cppcheck。这说明相较于已有的方法和策略,所提方法可以在占用开销较低的基础上有效提高复杂规则检查的准确率。

关键词:C/C++编码风格检查;流敏感静态分析;跨文件代码检查
中图分类号 TP311

Flow-sensitive Coding Style Checking for C/C++ Programs

HU Mengze[1,2,3],MA Xutong[1,3],ZHANG Hao[1,3] and ZHANG Jian[1,3]
1 State Key Laboratory of Computer Science,Institute of Software,Chinese Academy of Science,Beijing 100190,China
2 Hangzhou Institute for Advanced Study,UCAS,Hangzhou 310012,China
3 University of Chinese Academy of Sciences,Beijing 100049,China

Abstract C/C++ programming languages are applied in numerous critical software systems,and there is an extremely high demand for standardization and clarity of semantics during development. To prevent potential security issues arising from improper use of C/C++ languages,a *C/C++ Language Programming Security Subset*(referred to as GJB8114) was proposed domestically. Given the abundance of rules within the standard,it's inevitable that programmers may deviate from these norms,thereby necessitating automated rule detection tools to identify such non-compliant coding practices. However,existing rule checking tools do not provide comprehensive checks against the standards,especially for rules that require understanding of the program's context,leading to high false positive rates or even a lack of support for certain checks. This paper categorizes the rules in GJB8114 and defines what constitutes a complex rule. Through evaluating the Testbed tool's capability to inspect complex rules within GJB8114,it identifies that current tools lack thorough flow-sensitive analysis and are unable to perform cross-file global analysis. To address these issues,this study adops a flow-sensitive analysis method combined with syntax tree matching and a cross-file global analysis approach. Based on this,the CruletFS tool is developed. Experimental results demonstrate that CruletFS performs better in checking complex rules compared to common rule checking tools,such as Cppcheck and Testbed. In analyzing large-scale projects,CruletFS also outperforms Cppcheck in terms of time and memory overhead.

Keywords C/C++ coding style check,Flow-sensitive static analysis,Cross-file code check

到稿日期:2024-03-31 返修日期:2024-08-17
基金项目:国家自然科学基金(62132020)
This work was supported by the National Natural Science Foundation of China(62132020).
通信作者:张健(zj@ios.ac.cn)

1 引言

C/C++语言编写的软件广泛应用于航天工程、汽车工业、医疗设备、武器装备控制系统等各种安全相关领域中[1-2]。为了确保这些领域中的系统能够正常运转，顺利完成既定任务，必须高度重视这类软件的安全性。

C/C++语言是一门灵活而又复杂的语言，开发者可以使用语言的相关特性进行简洁、高效的开发。但与此同时，一些可能不易被察觉的安全缺陷也会随之产生。在程序开发的过程中，部分开发者会更加追求代码编写的效率，而不是代码的正确性和可读性。为了应对这些问题，国内外一些有安全性和可靠性要求的行业推出了契合该行业的C/C++语言的安全编程子集，其目的是通过制定一系列的编程规范并要求开发者在开发过程中遵守这些规范。例如航天科工集团针对航天航空行业特别制订了C语言编程标准，即《航天型号软件C语言安全子集》[3]。这一编程规范是由在航天行业拥有深厚经验的软件工程师与专家针对航天型号软件中所存在的缺陷提出的。《C/C++语言编程安全子集》[4]（后简称GJB8114）则是在《航天型号软件C语言安全子集》基础上，经过进一步的修订而形成的C/C++语言安全编程标准。

由于现有的编程标准大多条目繁多，普遍有100～200条规则进行约束，对此实现一个工具以自动化地分析被测程序是否符合相关标准是有必要的。以GJB8114标准为例，其具有200余条针对C/C++语言的规则，目前也已经有很多工具宣称能够支持对该标准进行检测，如LDRA公司的Testbed[5]、北大软件的CoBOT[6]、上海那一科技的NaiveSystems Analyze[7]等。目前工业界较为主流的工具便是LDRA开发的Testbed工具。

针对Testbed工具是否能够较好地检测GJB8114标准的问题，本文进行了相关调研以及实证研究，发现：1）对于一些需要理解程序上下文的规则（本文将其定义为复杂规则），如“动态分配的内存空间需及时释放”等，Testbed工具分析能力较差，有很多的误报；2）一些GJB8114中如“避免使用无用的多余函数”等需要工具进行全局分析的规则，Testbed工具无法进行分析并给出报告。对此，本文将编程规范检查中常用的语法树匹配与流敏感的分析方法相结合，提出结合语法树匹配的流敏感分析方法，以能够快速和准确地分析被测程序，并提出了跨文件的全局分析方法以综合考虑项目中所有文件的信息。根据这些方法，本文在已有工具Crulet基础上进行二次开发，实现了CruletFS工具。

实验结果表明，CruletFS在误报率更低的前提下比Testbed检查出更多的错误。在工具开销的比较上，实现复杂规则的CruletFS的开销对比Crulet增加不到5%。这说明相较已有的方法和策略，所提方法可以在占用开销较低的基础上提高复杂规则检查的准确率。

本文第2章介绍了静态分析的常用技术与相关工具；第3章介绍了Testbed工具分析GJB8114标准的实证研究；第4章详细介绍了所提出的方法；第5章介绍了使用这些方法实现的工具CruletFS和相关的实验结果；最后总结全文。

2 相关研究

在分析相关研究时，本章将从C/C++规则检查中所用到的相关技术和现有工具两方面进行介绍。

2.1 静态程序分析技术

根据关注的要点不同，静态分析技术会用到抽象语法树分析、数据流与控制流的分析、抽象解释和符号执行等方法。

抽象语法树（Abstract Syntax Tree，AST）分析技术是静态分析中的一种常用技术。它通过解析代码，将代码的源代码结构转换为一种树型表示，即抽象语法树，这种树结构清晰地表示了代码的语法结构，包括变量定义、赋值、控制流语句等。利用AST，分析工具可以高效地遍历代码中的所有元素，对代码进行深入分析，从而发现潜在的代码问题、风格不一致、潜在的性能问题等。抽象语法树分析的优势在于它的准确性和灵活性，能够对代码做出更为精确的理解和分析，有相当一部分静态分析工作便是基于抽象语法树进行的[8-9]。

数据流分析是一项在编译时使用的技术方法。通过在程序代码中收集程序的语义信息结合代数方法，在编译时便能够确定变量的定义和使用。数据流分析关注的是如何在程序的不同部分之间传递信息（变量的定义和使用等）[10]。由于一些规则在判断时需要结合程序语句的上下文信息，因此对这些规则进行检查时需要用到数据流分析。为了在效率和精度之间取得平衡，实际应用的流敏感分析技术往往会采用各种优化方法和启发式算法来减小分析的复杂度[11]。数据流分析往往在LLVM IR上进行，这是因为IR作为一种中间表示相对抽象语法树在结构上更加简单且与语言无关，适合进行数据流分析[12-13]。

基于摘要的过程间分析，其核心在于生成“摘要”。摘要（Summary）是可复用的程序模块分析结果，通过摘要能够快速得到模块的外部行为。创建摘要和利用摘要开展分析的过程称为摘要分析。在实际项目代码中，一个程序模块中往往需要调用其他程序模块。因此，对一个主调模块的分析不可避免地牵涉到对它所调用的模块的分析。这类似于通过复用代码来提高软件开发效率的方法。通过这种基于摘要的程序分析方法，研究者可以将被复用模块的分析结果构建成摘要，并在分析主调模块时对其进行实例化处理，从而加速整个分析过程。

符号执行方法是一种路径敏感的分析方法，它以符号值而不是具体的输入值来执行程序[14]。在符号执行过程中，程序中的变量被表示为符号表达式，通过对符号表达式进行约束求解。符号执行能够发现程序中的潜在错误、漏洞和不变性质，而流敏感的分析方法则不会考虑路径是否可达[15]。

2.2 现有工具

实现编程规则的自动检测，可以大大提高软件开发过程中的编码规范性，保证软件开发的质量。目前，工业界和开源社区已经推出不少的静态分析工具，用于提升代码质量。目前已有工作对静态分析工具进行了比较，但其缺乏对如Testbed之类的编程规范检查工具的比较[16]。

Cppcheck[17]是一款较有影响力的开源静态分析代码检测工具。在分析过程中，Cppcheck并不需要生成完整的抽象

语法树便能进行分析，检查出实际项目中存在的缺陷[18]；但也有研究发现，Cppcheck 工具生成的报告中错误报告的比例较高[19]。通过以词法分析的方式结合一定程度的数据流分析对被测代码进行分析。Cppcheck 支持在较多场景下进行分析，但是 Cppcheck 并不支持对 GJB8114 标准进行分析。

LDRA Testbed 是一款知名的代码分析检测工具，能够支持静态分析大规模程序代码。它是由英国的 LDRA 公司设计的，是一款收费的商业工具。在静态分析代码的同时，Testbed 也能够针对程序代码的圈复杂度、代码行等信息进行分析。LDRA Testbed 主要采用了 DERA 和 MISRA 两种国际标准，同时也支持 GJB5369 和 GJB8114 标准。Testbed 在分析过程中也采用了自建的以词法分析为基础的模型。

Clang Static Analyzer[20]（简称 ClangSA）是由 LLVM 社区 Clang 编译器项目组开发的以静态符号执行方法为基础的缺陷检测工具。工具集成在 Clang 编译器内部，可以通过编译器调用，以获取被测文件的抽象语法树（Abstract Syntax Tree，AST），在语法树上进行后续分析以判断代码中是否存在缺陷。

Astrée[21]工具是一款基于抽象解释的代码分析工具，Astrée 成功应用于空客 A340，A380 等系列飞机飞行控制软件的自动分析并实现了分析的零误报[22]。Astrée 的扩展版本 AstréeA 支持多线程 C 程序中运行时错误、数据竞争、死锁等的检测，并成功应用于 ARINC653 航空电子应用软件（约 220 万行代码）的分析[23]。

Crulet 是一款由 Yang[24]开发的工具，随后经由 Wang[25]重构改进，目前已经能支持 GJB5369 和 GJB8114 这两项标准。Crulet 工具是一款基于 Clang Libtooling[26]的工具，以抽象语法树分析为基础，相较于词法分析能够做到更加精确的分析。实验发现，Crulet 工具总体上有 90%以上的正确率，是一款优秀的分析工具。

在以上 5 款工具中，Testbed 工具和 Crulet 工具都支持 GJB8114 的检查，其中 Testbed 是目前行业内检查 GJB8114 标准较为主流的商业工具。由于商业工具获取难度较大，且并没有针对 GJB8114 规则实现的开源工具，因此仅选用了较为主流的 Testbed 工具进行实证研究。

3 Testbed 工具对 GJB8114 标准支持程度的实证研究

3.1 复杂规则的挑选

GJB8114 标准中的大部分规则适用于具体变量的命名规范或者使用场合，对于这些规则，采用抽象语法树级别[27]的方法已经能准确地进行分析。例如规则“函数参数表为空时，必须使用 void 明确说明”，为实现这条规则的检查器，只需要匹配所有的函数节点，并判断其是否具有函数参数，若没有函数参数，再检查其是否用了 void 进行明确说明即可。很明显，对于这条规则，只需要考虑函数节点本身即可，不需要考虑上下文信息。

GJB8114 标准中仍有一部分规则并不能完全使用这类方法进行解决。例如规则“禁止文件指针在退出时没有关闭文件”，为实现这条规则的检查器，检查工具需要匹配到所有打开文件的文件指针并且分析其在函数结束之前是否关闭了对应的文件。采用抽象语法树的分析方法只能判断是否关闭了对应的文件而无法在考虑程序语句执行顺序的基础上判断是否关闭了该文件，不足以分析该规则。

通过分析单一语法树节点及其子树节点信息并能够判断是否违背的规则称为简单规则，GJB8114 中大部分规则都是简单规则。与此相对的是，将那些需要通过多个语法树节点的上下文关系（需要分析多条语句）才能分析的准则称为复杂规则。GJB8114 中的复杂规则如表 1 所列。

表 1 GJB8114 中的复杂规则

Table 1 Complex rules in GJB8114

规则内容	规则编号
禁止释放没有在堆上分配的变量	3.1.5
指针变量被释放后需赋成 NULL	3.1.6
动态分配的指针在第一次使用前需要判断是否为 NULL	3.1.8
文件指针在退出时需要关闭	3.1.10
禁止试图使用已经被释放的对象	6.1.16
动态分配的内存需要及时释放	6.2.3
禁止不可达语句	8.1.1
禁止变量未赋值就使用	11.1.1
避免无用的多余变量	8.2.2
避免无用的多余函数	8.2.3

由于实际项目存在复杂路径条件、深调用链、复杂数据结构等多种情形，对变量值的分析较为困难，而编程规范检查本身更关注于变量间的关系而非变量的值。因此本文不将与变量值相关的规则列为复杂规则，也不进行实证研究。

3.2 开源程序测试集

为了全面评价 Testbed 工具的分析效果，需要使用 Testbed 对多种不同类型的开源项目进行检测。Github 中的仓库 awesome-cpp[28]收集了大量不同类型的优秀的 C/C++开源项目。本文从中选取了 8 个 C/C++语言编写的开源项目进行分析，这些开源项目来自 7 种不同的应用领域，其具体信息如表 2 所列。

表 2 待测项目

Table 2 Tested projects

项目名称	应用领域	C/C++	行数
inih	Configuration	C	0.808×10^3
cJSON	JSON	C	3.807×10^3
bftpd	FTP	C	4.539×10^3
moogoose	Embedded	C	19.325×10^3
cesanta v7	Embedded	C	25.806×10^3
btsk	AI	C++	1.556×10^3
celero	Benchmark	C++	5.383×10^3
plf_colony	Containers	C++	5.866×10^3

3.3 实验结果分析

本文使用 Testbed 工具对这 8 个 C/C++项目进行了扫描，并对生成的报告进行过滤，仅剩下涉及表 1 中的规则的报告，通过人工逐一排查其报告的准确性，最后的分析结果如表 3 所列（括号内为正确报告的数量，“—”代表未检出报告，N/A 代表不支持这条规则）。Testbed 工具的版本号为 9.7.2。

从分析结果可以看到，Testbed 工具对复杂规则的支持较差，以规则“禁止变量未赋值就使用”为例，Testbed 工具对 8 个项目共生成了 403 个误报。图 1 给出了其中一段简单的误报代码。

表 3 Testbed 工具对开源项目的分析结果

Table 3 Analysis results of open-source projects by the Testbed

GJB8114 规则	bftpd	celero	cJSON	inih	plf_colony	btsk	moogoose	cesanta v7
动态分配的内存需要及时释放	23(0)	—	—	—	—	—	16(0)	15(0)
禁止变量未赋值就使用	15(0)	—	—	1(0)	103(0)	9(0)	93(0)	182(0)
文件指针在退出时没有关闭	5(0)	—	—	—	—	—	—	1(0)
试图使用已经被释放的对象	1(0)	—	—	—	—	—	—	2(0)
避免无用的多余函数	—	—	—	—	—	—	—	—
避免无用的多余变量	—	—	—	—	—	—	—	—
禁止释放没有在堆上分配的变量	39(0)	—	3(0)	—	—	—	14(0)	26(0)
禁止不可达语句	—	8(0)	—	—	—	—	7(0)	367(1)
动态分配的指针在第一次使用前需要判断是否为 NULL	--- N/A---							
指针变量被释放后需赋成 NULL	---N/A---							

```
unsigned char a,b,c,d;
int ori_len=len;
while(len >=4 &&(a=from_b64(s[0])) !=255 &&(b=from_b64(s[1])) !=255 &&(c=from_b64(s[2])) !=255 &&(d=from_b64(s[3]) !=255){
…//使 用 了 a,b,c,d
}
return orig_len-len;
```

图 1 Testbed 的误报举例(变量未赋值就使用)

Fig. 1 Testbed's example of false positive(using a variable before it is assigned)

在该误报代码段中,Testbed 工具认为变量 a,b,c,d 均存在变量未赋值就使用的情况,而事实上在 while 语句内部,变量 a,b,c,d 均被使用,但在 while 语句的 condition 中,实质上已经为这 4 个变量进行赋值,而如果 while 中的 condition 值为 false,则说明这 4 个变量没有被完全赋值,但程序在离开 while 语句后便直接返回,并没有使用到这 4 个变量,因此说并不存在变量未赋值就使用的情况。

经过对 Testbed 的调研研究可以发现,在对以上几条规则进行分析时,Testbed 工具的分析效果较差,产生了大量的误报,严重影响了工具使用者的排查分析。从具体的误报例子中可以发现,Testbed 工具的数据流分析能力较弱,无法对一些较复杂情况进行正确的分析。同时 Testbed 虽然声称能够支持对无用的多余变量以及无用的多余函数这两条规则进行分析,但是在对实际项目的分析时发现,Testbed 工具并没能正确地发现其中没有被使用到的函数出现漏报,这说明 Testbed 无法对这类规则进行分析。

从上述的实证研究中可以发现,虽然 Testbed 能够进行流敏感的分析,但其分析能力较差,主要体现在以下两方面。

1)流敏感的分析不全面

流敏感的分析主要指获取程序中数据的使用、定义及其之间的依赖关系等各方面的信息。在本文中,流敏感的分析主要指对语句中变量使用的分析。Testbed 工具无法完善分析整个语句中变量的信息,如图 1 所示举例便无法准确分析变量 a,b,c,d 在程序中的使用和定义信息,造成了误报。

2)缺乏全局的分析

GJB8114 中的部分规则如避免无用的多余函数,需要工具拥有跨文件的分析能力,同时工具必须在分析完全部的编译单元后才能作出分析。Testbed 工具在分析这些规则时并不能产生任何报告(例如 bftpd 项目中存在多余函数,但工具没有报告),这表明其不具备这项能力。

4 流敏感与全局的分析方法

4.1 结合语法树匹配的流敏感分析方法

4.1.1 过程内的分析方法

常见的规则检查工具大多采取语法树匹配的策略去匹配相应节点,即判断违反一条规则的相关语法树节点结构是通过在抽象语法树上进行匹配,若匹配到相应的结构信息则认为该处违反了规则,并进行报告。而流敏感的分析大多从函数入口或出口开始分析并沿着控制流图收集相关信息。常见的流敏感分析是在 LLVM IR 上而非在抽象语法树上进行的。这是因为 LLVM IR 是静态单赋值(SSA)形式,这种形式中每个变量只被赋值一次,极大简化了数据流分析的复杂性,分析器只需跟踪每个变量的单一定义即可。若要在以语法树匹配为主的风格检查器上完整实现该方法,则必然需要对源代码程序的语句作进一步处理,这会导致与其语法树匹配的方法脱节,这种情况下整体的时间开销也会明显增加。本文提出了一种结合语法树匹配和控制流图语句的遍历相结合的方法,从而能直接在 AST 上进行流敏感的分析。

算法的整体思路如算法 1 所示。对于一条需要流敏感的分析方法的规则,与传统的语法树匹配方法类似,首先需要获取可能违反该规则的语法树结构,并使用 ASTMatch[29] 方法在抽象语法树上进行匹配。匹配到一组语句节点后,生成语句节点所在函数的控制流图,并获取语句节点在控制流图上的位置。以匹配到的第一条语句节点为入口沿着控制流图对途经的每条语句进行判断是否违反了相应规则。这里将会使用 ASTMatch 的方法用对应规则的匹配器对每条语句进行匹配并进行判断。由于抽象语法树已经生成,且每次只在一条语句内匹配,因此整体的时间开销并不大。

算法 1 结合语法树匹配的流敏感分析方法

输入:获取语句节点的匹配器 findNode,对每条语句检查的匹配器 findst

输出:生成报告

```
1. Results ← match(findNode)
2. if Results. empty then
3.     return //若没有匹配到节点则返回
4. for Result in Results do
5.     ST, FD,VD ← Result. getNode
6.     cfg← creatCFG(FD)
7.     theblock, line ← getBlock(cfg, ST)
```

```
8. for stmt ← traversal(cfg, theblock, cfg. exit) do //遍历控制流图
9.     if match(findst, stmt) then
10.         report //节点被匹配则报告
```

算法 1 中 findNode 为获取语句节点的匹配器，其作用是匹配与具体规则对应的语法树节点，Results 为通过匹配式匹配到的所有满足要求的结果，而 Result 属于其中一个满足要求的匹配结果，match 为 Clang 已有的语法树匹配方法。findst 是用于在生成完函数的控制流图后对控制流图上的语句进行遍历时的匹配器，以判断是否有违背规则的情况。第 5 行中 *ST* 指语句节点，*FD* 指函数节点，*VD* 指变量节点。

4.1.2 函数摘要与过程间分析方法

从算法 1 可以看出，该算法的分析依然是过程内的分析，在针对仅需过程内分析的情况时，算法能够正确地进行分析，但如果算法中所涉及的变量会被函数调用，算法便无法分析此时变量的具体情况，如图 2 所示，在这段代码中变量 *a* 被 callfun 函数调用了，若仅采用原算法便无法获知 callfun 函数的内部信息，这会影响分析的结果。为解决这个问题，本文补充了过程间的分析方法：在遇到函数调用节点时对所调用的函数进行分析，生成被调用函数的控制流图，随后递归地使用算法 1 对其进行判断。

```
…
void callfun(int *a){ //需要过程间的分析
…
}
int fr(void)
{
int a;//变量声明
…
callfun(&a);//函数调用
…
}
```

图 2 过程内分析无法解决的情况示例

Fig. 2 Examples of situations that cannot be resolved by intraprocedural analysis

过程间分析的方法的主要时间开销来自于对被调函数的分析。如果每遇到一个被调函数就对其进行分析是比较消耗时间的，通过基于摘要的程序分析技术，将被复用模块的分析结果用摘要进行描述，并在下一次主调模块调用该被调模块时直接获取摘要的值。算法 2 描述了过程间的分析方法，该算法将会在算法 1 遍历到函数调用语句时使用。本文所研究复杂规则并不涉及变量值的计算，因此算法在分析时仅会针对被调用函数中变量是否被释放、是否被关闭等信息生成摘要信息。

算法 2 中 FD 为被调函数，首先算法会生成被调函数的控制流图并将其保存到 cfginfo，cfginfo 中 issummary 方法会判断在该类是否含有函数摘要。若有摘要，算法便直接返回摘要结果；如没有摘要，算法便通过 makesummary 方法对函数进行分析并判断函数模块是否对变量做了相应处理。判断完毕后，算法将判断结果以摘要形式保存到 cfginfo 中并返回值。这样在下一次访问被调函数模块时，算法便可以直接返回值而不进行重复的分析。

算法 2 过程间的分析方法

输入：被调函数模块 FD，调用变量 p

输出：返回摘要结果

```
1. cfgInfo ← creatCFG(FD)
2. if cfgInfo. issummary ≠ UnKnown then
3.    return cfgnfo. issummary
      //分支成立表明建立摘要
4. else
5.    if makesumary(cfgInfo. entry, p) then
6.       cfgInfoset(unprocessed)
7.       return false
         //表明该模块并未对变量做相关处理
8.    else
9.         cfgInfo. set(processed)
10.        return true
```

4.2 跨文件的全局分析方法

现有的规则检查工具在分析源代码时大多采取有序分析每一个编译单元的策略。在每一个编译单元分析完毕后，将当前编译单元的分析结果保存或直接输出，使得工具在分析一个编译单元的某个节点时需要判断该节点是否需要报告，但部分规则需要在每个编译单元结束后才能判断节点是否违背了相关规则(例如判断一个函数是否在项目中被使用)，直接在一个编译单元中进行判断会导致无法准确分析相关结果。GCC 和 Clang 编译器可以在编译过程中对未使用的静态函数进行警告。Cppcheck 只能判断相同函数名是否被使用。显然这些工具所采用的方法无法覆盖全部情况。本文提出了一种全局的分析方法用于分析全部情况，算法 3 是其具体实现过程。

算法 3 跨文件的全局分析方法

输入：获取语句节点的匹配器 findNode，哈希表 allNode 存储所有被匹配到的节点，哈希表 nonComNode 存储目前暂不符合要求的节点

```
1. Results ← match(findNode)
2. if Results. empty then
3.    return //若没有匹配到节点则返回
4. for Result in Results do
5.    Decl ← Result. getNode
6.      usr ← Decl. getUnistring //将相同变量映射成唯一的字符串
7.       if usr Not in allNode then
8.       allNode[usr] ←usr. getMessage
9.    if Not isCompliant(Decl) then
10.      nonComNode[usr] ←usr. getMessage
11.   else if usr in allNode then
12.      if usr in nonComNode AND isCompliant(Decl) then
13.           nonComNode. delete(usr)
```

算法 3 首先针对匹配式 findNode 进行匹配，得到匹配的所有结果 Results。算法 3 遍历其中匹配到的每一个节点并生成其相对应的唯一的字符串 usr，随后在哈希表中判断该 usr 是否被记录，若没有则将其记录在哈希表中并判断是否符合要求。若该 usr 对应节点已被记录为不符合规则要求，则判断其是否在当前编译单元内符合要求，若符合要求则将其从不符合要求的哈希表中移除，这样最后剩下的便是在所

有编译单元上均不符合要求的节点，这些节点便是在全局分析下违反规则的节点。

跨文件的全局分析方法能够分析得到文件中未使用的函数即多余函数，但该方法仍有一定的局限性。例如该方法无法判断文件中虚函数是否被使用。这是因为在 C++语言中，虚函数的使用涉及运行时类型识别(Runtime Type Identification)，在编译过程中，工具无法判断到底哪一个函数被实际使用了，因此这类情况不属于需要被分析的情况。

5　工具实现

5.1　工具框架

由于 Testbed 为商业工具，不适合进行二次开发，因此本文在已有工具 Crulet 的基础上进行了二次开发，实现了 CruletFS 工具。图 3 为 CruletFS 工具的框架图。

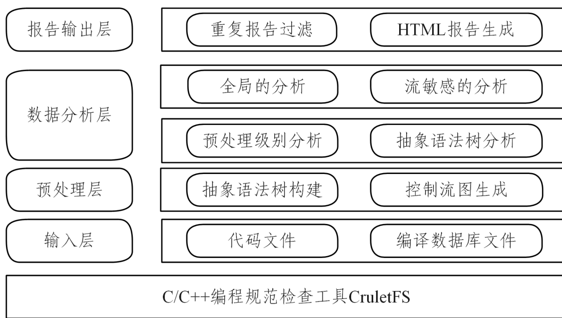

图 3　CruletFS 工具的框架图

Fig. 3　Framework diagram of the CruletFS

在输入层中，CruletFS 将会读取源代码以及对应的编译数据库文件，编译数据库文件中包含了源代码编译的相关信息，如头文件信息、宏的赋值等。

在预处理层中，CruletFS 主要负责语法树和各函数控制流图的构建，在该层中，CruletFS 主要依赖于 Clang 进行分析。Clang 能够编译各种标准的代码，这确保了工具能稳定分析各类复杂的 C/C++项目。

在数据分析层中，CruletFS 采取了 4 种不同的分析方法进行对应的分析，将在 5.2 节中介绍这些方法。

在报告输出层中，CruletFS 将会对重复报告进行过滤并以 HTML 格式输出。

5.2　工具工作流程

图 4 给出了 CruletFS 工具的工作流程。

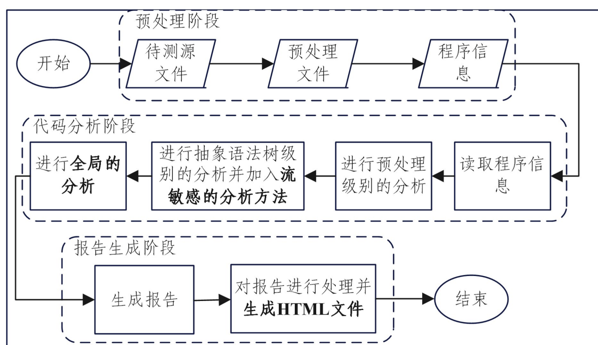

图 4　CruletFS 的工作流程

Fig. 4　Workflow of CruletFS

在预处理阶段中，CruletFS 工具将会构建源代码的抽象语法树，这一步是使用 Clang 进行的，在生成完抽象语法树后，对文件内的各函数，CruletFS 也会生成并存储其相对应的控制流图。通过控制流图，CruletFS 可以获得程序语句的序列信息。

在预处理结束后，CruletFS 将会在代码分析阶段进行每条对应规则的分析，在分析时，CruletFS 采取了 4 种分析方法，分别为预处理级别的分析方法、抽象语法树级别的分析方法、流敏感的分析方法以及全局的分析方法。

预处理级别的方法和抽象语法树级别的方法是源自于 Clang 的 Libtooling，Clang-tidy 工具便是采用这两种方法进行检查的。预处理级别的方法主要处理#define，#undef 等预处理信息，抽象语法树级别的方法则用于获取对应的语法树节点信息并进一步分析其是否违反规则。

流敏感的分析方法以及全局的分析方法是 CruletFS 中所特有的分析方法，这两种方法都是在抽象语法树上获得对应的语法树节点后进行的分析，具体内容在 5.1 节已经说明。

在报告输出阶段中，CruletFS 将会对输出的报告进行整合，将位于同一文件内的报错信息合并至该文件并以 HTML 格式进行输出。HTML 页面包括了被扫描的各个文件，点击文件后便能够在代码文件内获得错误报告信息，方便开发人员进行排查。

6　实验与分析

本章将从实验角度分析 CruletFS 工具检查的有效性和开销。

由第 3 和第 4 章可知，CruletFS 相较于传统的规则检查工具，在流敏感的分析和全局的分析方法上有优势，但仍需要实验判断 CruletFS 工具是否能准确地分析这类规则。同时，一些静态分析工具例如 Fbinfer，ClangSA 等能进行更高精度的分析。从方法上看，CruletFS 的分析能力肯定不如这些工具，但由于 CruletFS 仍是在语法树基础上进行分析的，因此其时间和空间开销应该明显低于传统静态分析工具。

因此，本章将会从以下 3 个方面对 CruletFS 进行评估。

RQ1　CruletFS 是否能够分析表 1 中所提到的规则？

RQ2　CruletFS 分析实际项目的效果如何？

RQ3　CruletFS 的时间开销和内存开销如何？

6.1　实验设置

本文所有实验的实验环境为 Linux 服务器平台，处理器为 Intel© Xeon© E5-2680 v4，内存为 256 GB。本节将介绍相关对比工具和实验所用的测试集。

6.1.1　对比工具的选取

为更好地判断 CruletFS 的有效性，需要选取对比工具进行对比，由于 CruletFS 是静态规则检查工具，因此对比工具也需要为静态工具。其次 CruletFS 不仅要和常见的规则检查工具作对比，而且也需要和主流的静态分析工具比较以判断性能差异。

综上，选择了两款静态分析工具进行结果的比对，两款分析工具名称和版本号如下：

Cppcheck(1.90)

ClangSA（LLVM 18.0.0）

Testbed（9.7.2）

3 款工具的基本信息已在现有工具中进行说明。同时由于本文第 3 章中使用 Testbed 工具对复杂规则进行了实证研究，因此将 Testbed 工具也列为对比工具。

6.1.2 测试集的设计

测试集包含手工构建的测试用例和实际项目的测试集。对于手工构建的测试集，本文对于要分析的 10 条规则，仿照 Juliet[30] 测试集构造测试用例，对于每条规则都构造 60 个测试用例，其中包含 20 个反例和 40 个正例。如图 5 所示的代码片段是其中的一个与规则“变量在被释放后没有赋值为 NULL”相关的测试用例。

```
#ifndef OMITBAD
void free_01_bad()
{
char * data;
data=NULL;
data=(char *)malloc(100 * sizeof(char));
memset(data,'A',100-1);
data[100-1]='\0';
free(data);//释放 data 后未赋值为 NULL
}
```

图 5 变量在被释放后没有赋值为 NULL 的用例

Fig. 5 Testcases where variables are not assigned NULL after free

从图 5 中可以看出，free_01_bad 函数是一个有关变量在被释放后没有赋值为 NULL 的反例，其中 data 变量在被释放后没有赋值为 NULL，违反了 GJB8114 规则。工具在分析时需要能够报告出 free_01_bad 函数中存在的问题。

对于实际项目的测试集，为了便于对比 Testbed 工具，仍选用表 2 中的项目进行实验。

6.2 CruletFS 分析的有效性

使用 CruletFS 工具和相关对比工具对手工构建的测试集进行分析，所得分析结果如表 4 所列，其中 RCR 代表使用指定工具对对应规则集的检测结果的召回率，即若 RCR 为 100 则说明工具能够正确检出对应的 20 个反例，FPR 为对应的误报率，FPR 为 0 则说明工具在对指定规则进行检查后没有出现误报的情况。

表 4 测试集实验结果

Table 4 Test set experiment results

工具	CruletFS		Testbed		Cppcheck		ClangSA	
GJB8114 规则	*RCR*	*FPR*	*RCR*	*FPR*	*RCR*	*FPR*	*RCR*	*FPR*
动态分配的内存需要及时释放	85	34.6	0	—	0	0	80	33.3
禁止变量未赋值就使用	85	51.6	0	—	0	0	100	20
文件指针在退出时没有关闭	100	9.1	95	60.4	70	0	N/A	
试图使用已经被释放的对象	85	0	85	0	0	0	60	0
无用的多余函数	100	0	0	—	20	0	N/A	
无用的多余变量	100	0	0	—	100	0	100	0
禁止释放没有在堆上分配的变量	100	9.1	100	36.4	0	0	100	0
动态分配的指针在第一次使用前需要判断是否为 NULL	100	0	0	—	N/A		N/A	
禁止不可达语句	100	0	100	0	100	0	100	0
指针变量被释放后需赋成 NULL	100	0	0	—	N/A		N/A	

从分析结果上可以看出，对于手工构建的测试用例集，CruletFS 工具在大部分测试代码检查中效果良好，与此同时 Cppcheck 工具在动态分配的内存需要及时释放、禁止变量未赋值就使用等规则中并没有检出，这表明 CruletFS 工具在这些规则的分析上优于 Cppcheck 工具。

同时，将 CruletFS 工具和静态符号执行工具 ClangSA 进行对比发现，ClangSA 在大多数规则的检查结果上有着更好的召回率，同时其误报率显著低于 CruletFS，这是因为 ClangSA 采取了符号执行的方法，能够判断路径是否可行，降低了报告的误报率。

综上可以发现，针对一些小的测试用例的扫描，CruletFS 的效果优于常见的规则检查工具 Cppcheck，但与静态分析工具比较，在召回率和误报率上都有一定差距，这是因为 CruletFS 流敏感分析是较为轻量级的分析。

6.3 实际项目上的分析

CruletFS 工具对开源项目的分析结果如表 5 所列。

表 5 CruletFS 工具对开源项目的分析结果

Table 5 Analysis results of CruletFS tool on open source projects

GJB8114 规则	bftpd	celero	cJSON	inih	plf_colony	btsk	moogoose	cesanta v7
动态分配的内存需要及时释放	—	—	—	—	—	—	—	—
禁止变量未赋值就使用	—	—	1(0)	—	—	—	—	—
禁止文件指针在退出时没有关闭	—	—	—	—	—	—	—	—
试图使用已经被释放的对象	—	—	—	—	—	—	—	—
避免无用的多余函数	12(12)	27(27)	50(50)	—	—	6(6)	82(82)	38(38)
避免无用的多余变量	10(10)	4(4)	—	2(2)	1(1)	—	3(3)	1(1)
禁止释放没有在堆上分配的变量	—	—	—	—	—	—	—	—
禁止不可达语句	—	3(3)	—	—	2(2)	—	4(4)	16(16)
动态分配的指针在第一次使用前需要判断是否为 NULL	4(4)	—	2(2)	—	—	1(1)	—	17(17)
指针变量被释放后需赋成 NULL	52(52)	1(1)	9(9)	—	—	—	22(22)	25(25)

从表5可以看出,CruletFS工具所生成的报告大多为正确的报告,对规则避免使用无用的多余函数的分析中,CruletFS工具在8个项目中共发现了200余个多余函数,而Testbed工具发现了0个。但在另一方面可以看到,CruletFS工具对“动态分配的内存需要及时释放”“文件指针在退出时没有关闭”“试图使用已经被释放的对象”等规则并没能发现一个报告,这是因为所实现的流敏感分析方法为轻量级的分析,无法对所有的情况进行分析,而所选的项目大多是经过大量测试的成熟发行版本。由于CruletFS工具本质上为风格检查工具,因此没有发现报告也是正常的。

在对Crulet分析的有效性进行统计时,本文对结果的召回率进行了分析。但在实际项目的分析上,很难对项目的漏报率进行分析,这是因为很难对实际项目漏报进行较为客观的统计。事实上,目前C/C++语言静态分析的不少工作中也并不涉及对实际项目的召回率[31-32]的分析,因此本文不对实际项目进行召回率的分析。

6.4 CruletFS的开销分析

6.4.1 时间开销的比较

为分析工具的时间开销,本文针对6.2节中所给出的实际项目使用CruletFS进行扫描,对每个项目均扫描100次并计算平均扫描时长以判断工具的时间开销。

为便于对比工具的时间开销,把Cppcheck,ClangSA以及Crulet工具作为对比工具同CruletFS一样扫描对应项目100次并计算平均扫描时长。表6列出了时间开销比较结果。Testbed的实验环境在Windows平台下,由于工具采用前端界面进行分析和其他对比工具均不同,因此本节不将Testbed列为对比工具。从表6中可以看出,CruletFS的时间开销相较于没有引入高精度分析的Crulet,并没有明显增加,这是因为CruletFS在分析过程中仍然以语法树节点匹配为基础,只有在匹配到对应节点的情况下才会生成节点所在函数的控制流图,这较大地减小了分析的时间开销。同时,其时间开销在代码规模较大时显著优于Cppcheck工具。同时,CruletFS的时间开销全面优于ClangSA工具,这是因为符号执行的方法会使时间开销较大。

表6 CruletFS的时间开销比较

Table 6 Time cost comparison of CruletFS

(s)

项目	bftpd	celero	cJSON	inih	Plf	btsk	moogoose	cesanta v7
Crulet	2.982	45.407	1.152	0.122	4.861	8.522	3.125	16.404
CruletFS	3.041	47.604	1.284	0.138	5.061	8.831	3.440	17.196
CSA	17.627	54.266	21.076	4.133	0.891	29.520	39.804	99.617
Cppcheck	19.772	0.161	1.701	0.100	5.488	0.246	7.864	34.019

注:由于Testbed的实验环境与以上工具均不同,因此不进行开销评估。

6.4.2 内存开销的比较

工具的内存开销比较方法和6.3.1节相同,区别为本节侧重于判断工具的内存开销。

实验结果如表7所列。从表7中可以看出,相较于Crulet工具,CruletFS工具的内存开销没有明显提升,与时间开销比较类似,CruletFS工具的内存开销也显著低于ClangSA工具,在代码规模较大的情况下,CruletFS的内存开销也优于Cppcheck。

表7 CruletFS的内存开销比较

Table 7 Memory cost comparison of CruletFS

(MB)

项目	bftpd	celero	cJSON	inih	plf	btsk	moogoose	cesanta v7
Crulet	33.121	118.654	30.897	29.041	87.010	69.930	46.894	65.910
CruletFS	34.463	120.669	31.470	29.968	87.570	70.710	47.454	66.943
CSA	166.365	213.078	180.722	135.563	130.494	195.052	179.969	231.760
Cppcheck	14.321	9.244	6.053	9.151	42.416	12.072	52.575	127.356

注:由于Testbed的实验环境与以上工具均不同,因此不进行开销评估。

结束语 本文以GJB8114规则为基础进行了一系列研究,具体而言本文的主要贡献如下。

1)完成了Testbed工具对GJB8114规则支持程度的调研。本文将GJB8114规则进行了分类,分为了简单的规则和复杂的规则两类,并进行了使用Testbed工具分析项目是否违反GJB8114规则的实证研究。从结果中可以看出,虽然Testbed工具在实践中的检查能力良好,但其针对复杂规则的检测仍有不足。

2)提出了适用于编程规范检查的流敏感和跨文件的分析方法。针对编程规范检查工具的需求,提出了结合语法树的流敏感分析方法以及跨文件的全局分析方法。为在实际项目中发现更多的错误,本文还实现了过程间的分析方法。

3)实现了支持复杂规则检查的CruletFS工具。在已有工具Crulet的基础上进行二次开发,实现了CruletFS工具。与Crulet工具相比,CruletFS可以在不依赖于额外工具的情况下支持复杂规则的检查,并且CruletFS在工具耗时、安全性以及报告生成等部分都进行了优化。

今后可以在所提出方法的基础上,进一步支持其他编程规范(如MISRA C,Cert C/C++等)中复杂的规则并且分析结果的有效性。

参考文献

[1] TIOBE Index [EB/OL]. https://www.tiobe.com/tiobe-index/.

[2] Top Programming Language 2024 [EB/OL]. https://elevatex.de/blog/it-insights/programming-languages-ranking-2024.

[3] safe subset of C language for space armament software [S].

Commission of Science, Technology and Industry for National Defense, 2005.

[4] Safe subset of C/C++ language programming: GJB8114-2013 [S]. General Armaments Department of the People's Liberation Army, 2013.

[5] Liverpool Data Research Associates. LDRA Testbed-static and dynamic code analysis [EB/OL]. http://ldra.com/aerospace-defence/products/ldra-testbed-tbvision/.

[6] GAO Q, MA S, SHAO S, et al. CoBOT: static C/C++ bug detection in the presence of incomplete code [C] // Proceedings of the 26th Conference on Program Comprehension. 2018: 385-388.

[7] Shanghai Nayi Technology Co., Ltd. [EB/OL]. https://naive-systems.com/

[8] MA X, YAN J, LI Y, et al. SPrinter: a static checker for finding smart pointer errors in C++ programs [C] // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019: 1122-1125.

[9] MA X, YAN J, ZHANG H, et al. Detecting Memory Errors in Python Native Code by Tracking Object Lifecycle with Reference Count [C] // 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023: 1429-1440.

[10] WANG X F, ZHAO K J, TIAN Z W. Research on Key Technologies of Data Flow Analysis [J]. Computer Science, 2005, 32(12): 91-93.

[11] SCHUBERT P D, LEER R, HERMANN B, et al. Into the Woods: Experiences from Building a Dataflow Analysis Framework for C/C++ [C] // 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2021: 18-23.

[12] YE S, SUI Y, XUE J. Region-based selective flow-sensitive pointer analysis [C] // International Static Analysis Symposium. Cham: Springer, 2014: 319-336.

[13] SUI Y, YE D, XUE J. Detecting memory leaks statically with full-sparse value-flow analysis [J]. IEEE Transactions on Software Engineering, 2014, 40(2): 107-122.

[14] BALDONI R, COPPA E, D'ELIA D C, et al. A survey of symbolic execution techniques [J]. ACM Computing Surveys, 2018, 51(3): 1-39.

[15] ZHANG J, ZHANG C, XUAN J F, et al. Recent Progress in Program Analysis [J]. Journal of Software, 2019, 30(1): 80-109.

[16] FATIMA A, BIBI S, HANIF R. Comparative study on static code analysis tools for C/C++ [C] // 2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST). IEEE, 2018: 465-469.

[17] Cppcheck-a tool for static C/C++ code analysis [EB/OL]. (2023). https://cppcheck.net.

[18] PEREIRA J D, VIEIRA M. On the use of open-source C/C++ static analysis tools in large projects [C] // 2020 16th European Dependable Computing Conference (EDCC). IEEE, 2020: 97-102.

[19] KAUR A, NAYYAR R. A comparative study of static code analysis tools for vulnerability detectionin C/C++ and Java source code [J]. Procedia Computer Science, 2020, 171: 2023-2029.

[20] LLVM Team. Clang static analyzer [EB/OL]. https://clang.llvm.org/docs/ClangStaticAnalyzer.html.

[21] Astrée runtime error analyzer [EB/OL]. https://www.absint.com/astree/index.htm.

[22] BLANCHET B, COUSOT P, COUSOT R, et al. A static analyzer for large safety-critical software [C] // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. 2003: 196-207.

[23] MINÉ A, DELMAS D. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software [C] // 2015 International Conference on Embedded Software (EMSOFT). IEEE, 2015: 65-74.

[24] YANG L. The Optimization and Improvement of Software Testing for C Programs [D]. Beijing: University of Chinese Academy of Sciences, 2018

[25] WANG W. C/C++ Language Programming Safety Standard Compliance Check [D]. Beijing: University of Chinese Academy of Sciences, 2022

[26] LLVM Team. Libtooling [EB/OL]. https://clang.llvm.org/docs/LibTooling.html.

[27] LLVM Team. Introduction to the clang ast [EB/OL]. https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[28] Fallahi. awesome-cpp [EB/OL]. https://github.com/fffaraz/awesome-cpp.

[29] LLVM Team. ASTMatch [EB/OL]. https://clang.llvm.org/docs/LibASTMatchersReference.html

[30] BLACK P E. Juliet 1.3 test suite: Changes from 1.2 [M]. US Department of Commerce, National Institute of Standards and Technology, 2018.

[31] ZHANG H, LUO J, HU M, et al. Detecting Exception Handling Bugs in C++ Programs [C] // 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023: 1084-1095.

[32] MA X, YAN J, WANG W, et al. Detecting memory-related bugs by tracking heap memory management of C++ smart pointers [C] // 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021: 880-891.

HU Mengze, born in 1999, postgraduate, is a student member of CCF (No. Q7307G). His main research interests include static analysis of C/C++ programs and so on.

ZHANG Jian, born in 1969, Ph.D, professor, Ph.D supervisor. His main research interests include software engineering and automated reasoning.

（责任编辑：喻藜）