计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(7): 1884−1903, 2026

基于跨版本测试的编译器引入型漏洞检测方法

徐坚皓[1,2] 李林克[1] 张昆博[1] 唐 晨[1] 茅 兵[1]
[1](计算机软件新技术全国重点实验室(南京大学) 南京 210023)
[2](东南大学网络空间安全学院 南京 211189)
(jianhao.xu@seu.edu.cn)

Compiler-Introduced Vulnerability Detection Methods Based on Cross-Version Test

Xu Jianhao[1,2], Li Linke[1], Zhang Kunbo[1], Tang Chen[1], and Mao Bing[1]
[1] (*National Key Laboratory for Novel Software Technology* (*Nanjing University*), *Nanjing* 210023)
[2] (*School of Cyber Science and Engineering*, *Southeast University*, *Nanjing* 211189)

Abstract Compilers can introduce security vulnerabilities during the compilation process. These compiler-introduced vulnerabilities are often subtle and can impact critical software such as operating system kernels, databases, and browsers, posing significant security risks. Nowadays, various softwares, including compilers, are rapidly iterated and updated, which means that new vulnerabilities can be introduced into important software binary code by compilers at any time. Our research indicates that such vulnerabilities often persist in software for a considerable period before being addressed. Therefore, timely detection of newly introduced compiler vulnerabilities is crucial for ensuring software security. There has been extensive research on detecting security vulnerabilities introduced by compilers. However, existing detection methods often focus on specific known vulnerability types and struggle to efficiently handle the frequent updates of compilers and the programs being tested. Consequently, we propose a novel detection technique based on general security code modeling and differential testing, which can efficiently identify security vulnerabilities introduced by compilers in the target program before and after version updates (either in the compiler or source code). This method identifies security code through error handling functions and determines compiler-introduced vulnerabilities by examining the differences in the removal of security checks across different versions. Evaluation experiments conducted on the Linux kernel demonstrate that our approach achieves a low false negative rate, maintains a practical false positive rate, and efficiently detects security vulnerabilities newly introduced by compilers during the compilation process of binary code updates.

Key words undefined behavior; compiler introduced vulnerabilities; differential testing; vulnerability detection; static analysis

摘 要 编译器可能在编译过程中引入安全漏洞。这种编译器引入型漏洞比较隐蔽,会影响绝大多数重要的软件如操作系统内核、数据库、浏览器,有严重的安全风险。如今各类软件包括编译器都在快速地迭代更新,这意味着随时可能有新的漏洞被编译器引入重要的软件二进制代码。实证调研表明,这类漏洞往往在软件中存在较长时间才能被修复。因此,及时地检测到编译器新引入的漏洞对保障软件安全十分重要。学术界已经有不少检测编译器引入的安全漏洞的研究工作。但是现有的检测方法往往只能针对特定的已知的漏洞类型,并且难以高效地应对编译器和待测源程序的高频次更新。因此,提出了一种新的基

收稿日期: 2025-03-24; **修回日期**: 2026-01-19
基金项目: 国家自然科学基金项目(62402107, 62572231, 62172201, 62032010)
This work was supported by the National Natural Science Foundation of China (62402107, 62572231, 62172201, 62032010).
通信作者: 茅兵(maobing@nju.edu.cn)

于通用安全代码建模和差分测试的检测技术，可以高效地检测待测程序在版本（编译器或源代码）更新前后被编译器引入的安全漏洞。该方法基于错误处理函数来定位安全代码，并通过不同版本中安全代码被去除的差异来确定编译器引入的安全问题。基于 Linux 内核的评估实验表明，该方法具有较低的漏报率和满足实际可用性的误报率，可以高效检测到二进制代码版本更新时被编译器新引入的安全漏洞。

关键词 未定义行为；编译器引入型漏洞；差分测试；漏洞检测；静态分析

中图法分类号 TP314

DOI: 10.7544/issn1000-1239.202550226 **CSTR:** 32373.14.issn1000-1239.202550226

计算机科学和技术的快速发展及各类计算机软件系统给人们的工作和生活提供便利的同时，也存在着各种安全隐患。

当前计算机软件的发展呈现出规模化和复杂化的趋势。规模化和复杂化给软件安全带来了许多挑战。大规模的软件系统涉及的代码量庞大，存在更多的潜在漏洞和错误。复杂的业务逻辑和交互可能导致隐藏的安全问题。软件系统的不断变更和更新可能引入新的错误和漏洞。因此，对于当前日益规模化和复杂化的计算机软件，软件安全需要得到更多的重视。

软件漏洞（software vulnerability）是指计算机系统安全方面的缺陷，使得系统或其应用数据的保密性、完整性、可用性、访问控制等面临威胁[1]。软件漏洞通常源于源程序自身的问题，可以定位到软件源代码的相关错误，编译器在这类漏洞的形成过程中是透明的（transparent）。安全专家们对这类软件漏洞防御技术进行了长期且大量的研究，产出了诸多有影响力的成果。然而，近些年来，越来越多的软件安全漏洞，其漏洞点并不在源代码中直接存在，而是由编译器在编译阶段引入。

图 1 展示了编译器如何在安全的源程序中引入安全漏洞并影响程序的正常运行：步骤①表示开发者开发出“安全”的源代码，并且将其交给编译器进行编译；步骤②表示编译器编译安全的源代码，然而在编译的过程中引入了隐蔽的安全漏洞；步骤③表示这些隐蔽的安全漏洞会影响程序正常的运行。

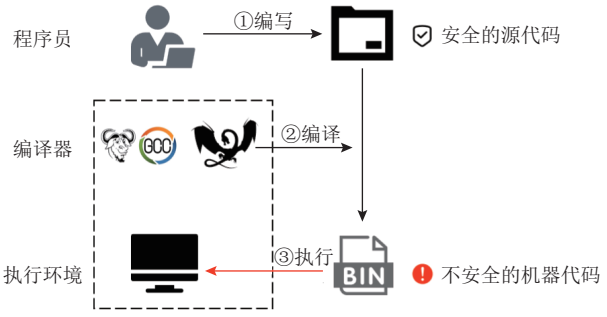


Fig. 1 Process of security vulnerabilities introduced by compilers

图 1 编译器引入安全漏洞的过程

编译器是重要的软件基础设施，主要承担着把高级语言编写的源代码翻译成机器可执行的机器代码的任务。由于编译器在软件生态系统中的特殊地位，编译器引入型漏洞存在 3 个特点：1）隐蔽性。这类漏洞在源代码级别是不可见的，是在编译阶段被安静地引入的。2）广泛性。几乎所有的重要系统程序，如操作系统、浏览器、数据库、虚拟机等，最终实际运行的代码都需要经过编译器的处理，因此都可能存在编译器引入型漏洞。单个编译器的安全问题也可能会影响很多应用软件。3）复杂性。漏洞的形成原因很复杂，跟编译器行为和具体源代码的语义结构都有关。漏洞的具体后果也很多样，这是因为这些漏洞大多表现为编译器对源程序的语义修改，直接引入的是逻辑错误，再间接地导致各种各样的安全后果。

编译器引入的安全漏洞日益突出，越来越受到关注。同时，由于上文提到的区别于传统漏洞的特点，使得编译器引入型漏洞更加难以避免。除了编译器实现错误引入的漏洞，正确的编译器也可能引入安全问题。编译器的正确性并不等同于安全性。最常见的情况是未定义行为（undefined behavior）相关的漏洞。当编译器基于程序中不存在未定义行为这一假设进行优化时，如果源代码中存在未定义行为，编译器的优化就可能引入安全漏洞。如图 2 所示，程序员在第 4 行可能引入了一个空指针解引用，但在某些环境下（如嵌入式 ARM 设备），这样的操作不会导致段错误。但是编译器假设程序中不会出现空指针解引用，因此在第 5 行中发现指针 *b* 被解引用后，推断第 6 行的空指针检查是不必要的，于是优化将其移除。然而，这一优化可能导致一个问题：如果空指针传播到其他部分，系统可能因此变得不安全。例如，攻击者可以利用这个逃逸的空指针修改中断

```
1    /* GCC Bugzilla id: 33629 */
2    void nullpointer_code (void *a)
3    {
4        int *b = a;
5        int *c = b;
6        if (b) { /* this check may be removed */
7            ...     /* 'b' is dereferendec here */
8        }
9        ...
10   }
```

Fig. 2 Security vulnerability introduced by a compiler

图 2 一个编译器引入的安全漏洞

表, 进而获取执行控制权。

此外, 编译器和其用户程序都面临着高速迭代的开发过程, 由于编译器引入漏洞的特点, 很难通过测试提前在发布前排除这类漏洞, 这意味着随时可能有新的漏洞被引入各种重要的系统软件, 甚至曾经被修复的漏洞也可能因为编译器的发展而被重新引入。

已有的相关工作都很难针对性地检测编译器引入型漏洞, 主要问题体现在下面 3 个方面。

1)现有的静态检测方法往往只建模所以也只能检测特定的某几类问题而无法针对更通用的安全问题, 并且往往因为依赖旧版本的编译器分析框架而普遍无法应用来测试最新的编译器。

2)尽管动态方法例如模糊测试搭配 Sanitizer 已经成为检测编译器引入型漏洞最主要的工具。然而因为需要运行程序, 动态检测方法针对这类问题存在天生缺陷: 一方面编译器及源程序版本更新就意味着需要重新运行被改变的目标程序, 而另一方面动态方法又存在普遍的代码覆盖率的问题。这说明动态检测方法难以有效追踪编译器优化引入的语义变化, 也无法保证高效的漏洞检测能力。

3)由于编译器和源程序的更新速度快, 现有办法很难在版本更新后迅速检测新引入的问题。即如何确保检测方法在不同编译器版本和源代码版本变更下仍然有效。

总的来说, 对于编译器引入漏洞的检测, 主要挑战在于: 1)如何针对通用的安全问题; 2)如何追踪编译器引入的语义变化; 3)如何判定编译器优化的安全性; 4)如何应对编译器和各类程序的快速迭代。在这 4 个挑战中, 第 2 个是本文的核心技术挑战。

针对以上 4 个挑战, 我们设计并实现了一种新的跨版本检测编译器引入型漏洞的方法。具体而言: 1)基于错误处理代码来建模普遍的安全问题; 2)构建了一套跨编译流程的语义差异分析方法来定位编译器对安全代码的去除; 3)基于跨版本的差分测试来判定编译行为的安全性; 4)定位版本更新引入的新的安全问题。其中跨编译流程的语义差异分析方法是我们的核心创新点。评估实验表明, 我们的方法可以自动检测由于编译器/源程序更新被编译器引入的漏洞, 拥有很低的漏报率。同时, 在不遗漏可疑编译器引入的代码变化的前提下, 能够保证较低的误报率, 满足在大规模的代码集和较大的编译器优化变化的条件下的实际可用性。

1 相关工作

1.1 软件漏洞检测

传统的软件漏洞检测方法主要依赖人工定义的规则集, 通过分析源代码与预定义规则的匹配情况来识别潜在漏洞[2]。近年来, 随着深度学习技术的发展, 漏洞检测范式逐步转向自动化特征学习。研究者尝试利用深度学习模型自动从代码中提取漏洞特征, 以提升检测能力。例如, Li 等人[3]通过提取与库函数调用相关的语义切片并加以拼接, 采用双向 LSTM 模型以捕捉上下文特征。Lin 等人[4]则系统性地结合了多种词嵌入模型与主流神经网络架构(如 TextCNN, LSTM), 对漏洞检测方法进行全面研究, 并评估了不同技术组合的性能表现。胡菘等人[5]提出了一种基于代码图表征的检测方法, 通过引入可学习的读出函数和基于对比学习的跨粒度损失函数, 增强了模型对代码结构特征的建模能力。

然而, 此类方法均建立在一个关键的前提假设之上, 即由源代码到目标代码的编译过程是语义无损且安全的。因此, 它们难以处理源代码本身安全、但编译过程中被引入安全风险的场景, 而这正是编译器引入型漏洞的核心问题所在。

1.2 编译器引入安全问题检测

D'Sliva 等人[6]定义了 correctness-security gap 的概念, 即编译器的正确性和安全性存在距离的问题。动态检测方法主要依赖运行时分析, 以发现编译器引入的安全漏洞。其中, 最具代表性的方法是模糊测试(fuzzing)[7]。模糊测试通过随机生成大量的测试样例并作为输入运行来测试程序, 并常常利用 Sanitizer 帮助在运行过程中捕捉安全漏洞, 这种动态测试的局限性主要是其代码覆盖率。同时面对编译器或者程序的高频次更新, 由于需要重新运行待测程序, 动态方法很难测试新增加的代码或者新部署的编译器优化。此外, 这些动态方法常常会受到 sanitizer 的限制, 现有的 sanitizer 往往只能针对特定

种类的编译器引入型漏洞，并且会有很严重的运行时开销[8]。

静态检测方法主要通过总结编译器引入型漏洞的语义特征来进行漏洞检测。例如，STACK[9]检测了编译器由于某几种“未定义行为(undefined behavior)”引发的安全代码去除。Unisan[10]调研并检测了由于编译器在结构体数据中插入填充字节引发的信息泄露漏洞。Yang 等人[11]和 van Sprundel[12]通过识别语义模板来检测死存储去除(dead store elimination)引起的敏感数据刮除。此外，一些研究专注于特定类型的安全问题，如加密密钥误用[13]、信息泄露漏洞[10]以及未定义行为[14]。这类方法的局限性主要体现在：1)漏洞类型依赖性强。只能针对已发现的问题定制化地进行复杂建模，不能自动化地迁移到未知种类的问题上。针对未知原因造成的漏洞的静态检测，仍然缺乏自动化的检测方法。2)时效性弱。相关漏洞检测往往依赖特定版本的编译器，难以迁移到新版本的编译器。一旦编译器升级，检测方法往往需要调整，无法有效扩展到新版本。

1.3 未定义行为检测

C/C++程序中的未定义行为是编译器引入型漏洞的重要诱发因素，已有很多用于检测未定义行为的研究，这些工作也能在一定程度上预防部分编译器引入的安全漏洞。动态工具如 UBSanitizer[15]可以检测常见未定义行为如整型移除等，ThreadSanitizer[16]可以检测数据竞争，TypeSanitizer[17]可以检测违反了严格别名规则的程序。除了 Sanitizer 以外，Dr. Memory[18]和 Valgrind[19]无需编译时插桩就可以在二进制级别进行检测。COMPDIFF[7]差分比较了不同编译器对同一程序结果来寻找程序中的未定义行为。静态工具例如 Infer[20]以及 CppCheck[21]都可以检测常见的未定义行为。这类工作只针对特定的未定义行为引发的安全问题，其中动态方法面临覆盖率低的问题，难以应对编译器和源程序的高频次迭代更新。

1.4 编译器测试

编译器测试技术是用于验证和评估编译器的正确性、稳定性和性能的一系列方法和工具。编译器作为将源代码转换为可执行代码的关键组件，其正确性对于软件系统的功能和性能至关重要。Csmith[22]可以基于 C 语言语法随机生成程序来进行编译器测试，在此基础上，Lidbury 等人[23]提出了 CLsmith，用于生成 OpenCL 编译器测试输入程序。Chen 等人[24]提出了 HiCOND，使用了历史数据辅助有效测试程序生成。Le 等人[25]提出了 EMI，该方法生成了一个与给定测试输入下的原始测试程序等价的变体程序变体，并使用这些程序对编译器进行测试。基于这个思想，研究人员进一步提出了 Athena[26]和 Hermes[27]。这类工作可以找到触发某种编译器正确性问题的测试用例(如一些随机生成的 C 代码)，但无法找到真实世界源程序中的安全问题，对于找到的编译器正确性问题也难以确认其真实的安全影响。

1.5 其他的编译器漏洞测试

已有大量工作围绕编译器漏洞展开研究，包括对编译器缺陷特征与成因的实证分析[28-29]，以及用于检测编译器编译正确性缺陷的测试工具与方法[30-32]。此外，在 Chen 等人[33]的研究中可以找到更多的编译器测试工作。Lee 等人[34]研究了一些 LLVM 中关于未定义行为的互相冲突的编译器假设，并提出了一个保证编译器生成正确代码的解决方案。除此之外，许多工作[35-40]形式化地验证了编译器功能的正确性，但这些工作无法处理编译器引入的安全问题。

1.6 安全编译

安全编译是指在编译过程中采取一系列的安全保护措施，使编译器在优化过程中能够保证安全属性的不变。一些研究开发了具有特定安全属性的编译器，例如机密性[41]、值完整性、恒定执行时间[42-46]、微体系结构副作用[47]。一些工作用形式化验证器来保证机密性[48]和恒定时间安全[49]。除此之外，完全抽象编译[50-53]致力于在编译过程中确保观察等价性，因此是安全编译的一种通用解决方案。然而，Patrignani 等人[54]的研究显示，性能和复杂的使用限制了它们的广泛应用。传统编程语言如 C/C++仍然存在编译器的安全问题。

2 背景知识

本文所提方法主要基于 LLVM 编译器框架实现，下面首先会对编译器的结构和 LLVM 的设计细节进行介绍，随后会介绍编译器优化类型和功能，最后会介绍编译器优化引入的安全问题。

2.1 编译器

编译器是一种将高级编程语言转化为机器可执行代码的软件工具，它能够将程序员编写的人类可读写的高级语言代码，如 C, C++, Java 等，转换为计算机能够解读、运行的低级机器语言的程序。编译器在软件开发中扮演着重要的角色，它可以将程序员编写的抽象、易于理解的代码转化为计算机可以直接执行的指令序列。

2.1.1 编译器流程

编译器的基本结构分为前端、中端和后端。其中前端是编译器的第 1 个阶段,它与源代码的解析和分析相关。前端负责处理源代码,并将其转换为内部表示形式,通常是抽象语法树(AST)或类似的数据结构。中端是编译器的优化阶段,它对前端生成的内部表示形式进行分析和优化,以改进程序的性能和效率。经过以上 2 步处理后,编译器进入后端部分,它将经过优化的中间表示代码转换为目标机器代码,以便在特定的硬件平台上执行。

2.1.2 LLVM 概况

LLVM 是一个开源的编译器基础设施项目,主要采用 C++语言编写,包含一系列模块化的编译器组件和工具链。目前 LLVM 支持 C, C++, Objective-C, Swift, Rust, go 等语言的编译,支持包括 x86, ARM, MIPS, PowerPC, AMDGPU, SPARC, RISC-V 等多种后端指令集。

如图 3 所示,LLVM 编译器采用 3 段式编译器设计,分为前端、中端和后端。LLVM IR(intermediate representation)是 LLVM 在中端优化过程中的代码表示。而 LLVM MIR(machine IR)是一种 LLVM 在后端优化过程中的代码表示。前端对输入的高级语言进行解析,会生成语言无关的中间表示(IR)。随后中端会对前端生成的 IR 进行分析和体系结构无关的优化,生成优化后的 IR 传递给后端进行进一步处理。后端会对传入的 IR 代码进行体系结构相关的优化和处理,生成和汇编指令一一对应的 MIR,最终生成可执行的目标机器指令序列。

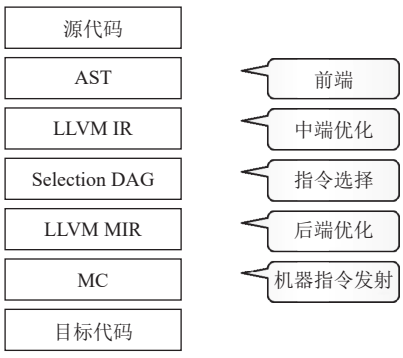


Fig. 3 LLVM compilation process

图 3 LLVM 编译流程

LLVM 指令(instructions)是 LLVM IR 中的基本操作单位,它们描述了程序的计算和控制流操作。LLVM 指令可以执行算术运算、逻辑运算、内存访问、控制流转换等操作。调试信息(debug information)用于保存每条指令和程序源代码的关联信息,通常包括变量名称、类型信息、行号等,可用于调试器、性能分析器和其他开发工具。在 LLVM 中也提供了丰富的调试信息支持,可以在编译期生成和处理调试信息。

LLVM IR 中的调试指令(debug instructions)用于描述源代码级别的信息,如变量范围、调试表达式、源代码位置等。这些调试指令通过调试元数据(debug metadata)与源代码中的特定位置和实体相关联。

2.2 编译器优化

编译器提供了大量的编译优化功能,以提高生成目标代码的性能和效率。它利用各种技术和算法,如常量传播、复写消除、循环优化、内联等,对代码进行重组和改进,以产生更高效的目标代码。通过优化,编译器可以降低程序的运行时间,减少内存消耗并提高系统的资源利用率,对提高软件质量和性能有重要的作用。

2.2.1 指令去除

有一类优化算法在优化时导致程序的一部分指令被去除。例如常量传播(constant propagation)通过识别程序中的常量表达式,将其结果直接替换为常量值,从而减少不必要的计算。死代码消除(dead code elimination)可以删除程序中不会执行的代码,包括未使用的变量、无效的分支和无效的函数调用,以减少目标代码的大小和提高程序的运行效率。冗余表达式消除(redundant expression elimination)识别和消除程序中重复计算的表达式,通过使用中间变量或将表达式的结果保存起来以避免重复计算,提高代码执行效率。

2.2.2 重排序

重排序是指在优化中改变部分指令执行的顺序进行优化。循环不变代码外提(loop invariant code motion)是一种应用在循环中的优化,用户的循环代码可能存在不变量,编译器可以识别出来,在不改变程序语义的前提下,将循环不变的语句或表达式移动到循环体之外,避免了每次循环都进行重复计算。循环交换(loop interchange)针对多重循环通过重新排列循环的嵌套顺序以改变循环迭代的顺序和访问数据的方式,从而提高程序的性能。

2.2.3 重新生成

重新生成是指在优化中用新的指令替换原有的指令进行优化。强度削弱(strength reduction)通过将高成本的操作替换为等效但低成本的操作来改进程序的性能。常见的强度削弱技术包括乘法替换为移位操作,除法和求余替换为移位操作或乘法操作,幂运算替换为乘法操作,根号运算替换为乘法操作等。

2.3 未定义行为

在编程语言规范中，未定义行为（undefined behavior, UB）是指语言标准未明确定义其执行结果的行为。未定义行为可以分为2类：一类是确定性的编程错误，比如缓冲区溢出、释放后使用，可能导致程序出现不可预测的执行结果并危害程序安全性；另一类是在特定的环境下有明确执行结果且其本身不会破坏程序安全性，但编译器根据“代码中不存在未定义行为”的假设可能会执行优化破坏的代码行为。C语言规范中有200种以上未定义行为，90%以上属于第2类[8]，如：空指针解引用、有符号整数溢出、违反严格别名规则等。

图4的例子展示了1个空指针解引用的例子，当输入*ptr*为NULL时，第3行存在未定义行为。图2的例子则展示了这种未定义行为可能在特定的场景下被编译器引入安全问题。

```
1    void bad_dereference (int *ptr)
2    {
3        int value = *ptr;
4        printf("Value: %d/n",value);
5    }
```

Fig. 4 An example of UB

图4 未定义行为示例

2.4 编译器引入的安全漏洞

编译器的功能是将一个高级语言程序翻译为机器能直接理解和执行的目标代码，编译器的正确性保证了编译前后的代码具有相同的功能。然而，作为一个大型的复杂软件，编译器的正确性往往无法完全保证。即使在编译器做到了完全正确的编译后，也仍然有可能在编译器优化中引入安全问题。

编译器漏洞是指在编译器软件中存在的错误或缺陷，可能导致生成的目标代码产生意外的行为或不正确的结果。这些漏洞可能对软件开发和整个系统产生严重的危害，并且很难察觉。它可能导致程序行为出现错误或崩溃，使软件无法正常运行。更严重的是，编译器漏洞可能引发安全漏洞，使恶意攻击者能够执行恶意代码或绕过安全措施。此外，编译器漏洞还会影响目标代码的性能和降低代码质量，影响程序运行速度和导致资源占用增加。同时错误的调试信息也将导致开发人员在追踪和修复问题时遇到困难。在使用含有漏洞的编译器编译得到的软件时，用户可能面临程序崩溃、数据丢失或不可预测行为，从而导致用户体验受损。

在编译器提供了正确的编译优化后，也不能保证编译结果的安全性。一个原因是程序安全特性可能超出了基本的语法功能范围。如图5所示，这段代码试图通过*memset*函数将存有敏感数据的变量赋值为0来防止数据泄露。然而在编译器优化的过程中，分析得到这个变量在之后的代码中没有被使用，在死代码删除优化中去除了这条语句相关的赋值操作，导致了编译器引入的信息泄露。另一个原因是，编译器只保证了语言规范中定义行为编译的正确性，而真实代码中存在一些不包含在语言规范中的未定义行为。当编译器基于程序中不存在未定义行为这一假设进行优化时，如果程序中存在未定义行为，就可能导致异常的情况。如程序员可能会使用if(*x*+10<*x*)来检查是否存在整数溢出，但编译器假设程序中不存在整数溢出这一未定义行为并因此删除这一行代码相关的指令。

```
1  /*  Linux commit: d4c5efdb97773f59a2b711  */
2  /*  Code path:drivers/char/random.c  */
3  static void extract_buf(struct entropy_store  *r,__u8  *out)
4  {
5      ...
6      memset(&hash, 0, sizeof(hash));
7      memzero_explicit(&hash, sizeof(hash));
8  }//  this patch tells the compiler to keep the deat store
```

Fig. 5 Information leaks introduced by compiler optimizations

图5 编译器优化引入的信息泄露

3 研究问题与动机

编译器是重要的软件基础设施。为了保证其正确性，其设计的正确性经过了大量的验证。然而，如同其他的软件，编译器的设计和实现都还存在一些错误或者局限，使得编译器可能在编译生成机器代码过程中引入安全漏洞。这类漏洞被引入的过程可以参照图1。

编译器引入这类漏洞的原因可以分为2类：1）编译器设计或者实现的局限。编译器依照语言规范的正确性标准，然而正确性并不等价于安全性[6]。如同Xu等人[8]对于这类问题的总结，这类问题可能是源自编译器实现的局限，例如编译器会假设一部分特殊的程序语义比如VB在源程序中不存在；还可能源自编译器设计的局限，例如很多安全属性比如程序的执行时间、数据的敏感性等并不在编译器正确性的范畴之内，编译器优化在设计上就不能维护这些安全属性。2）编译器实现的错误。编译器也是人类开发的程序，就像所有其他的程序，也可能存在实现的缺陷导致生成有安全问题的代码。

编译器引入型漏洞相较于一般的漏洞更加隐蔽，因为漏洞在源代码级别是不可见的，所以更难被检

测发现。而这类漏洞影响的范围很广,几乎所有的重要系统程序都需要经过编译器得到机器代码,因此这类漏洞有着比较大的安全影响。

由于当代程序快速迭代的需要,更新的编译器和源程序都可以使得新的漏洞被引入。这类漏洞可能会潜伏在代码中很久,因此在更新时及时发现新引入的漏洞有很重要的意义。根据我们的调研,历史上编译器引入型漏洞都需要比较长的时间才能被发现。具体来说,基于 Xu 等人[8]的数据集,我们分析了数据集中 Linux 内核中的编译器引入型漏洞。我们判定漏洞引入时间的方法是通过分析代码修改的 commit 记录。具体来说,我们在该数据集上寻找漏洞相关的代码变更,并采用以下策略来确定漏洞的引入时间:1)定位漏洞对应的代码修改,通过分析漏洞的影响范围和补丁,找到修复该漏洞的 commit,并进一步向前回溯代码历史,确定首次引入该漏洞的 commit;2)代码模式匹配,在引入漏洞的 commit 之前的代码版本中,未发现相同的代码模式,在该 commit 之后,直到漏洞修复前,代码模式持续存在,表明漏洞确实是在该 commit 引入的。

我们的分析结果如表 1 所示,发现漏洞从被引入到实际修复之间平均经过了 3.83 年。漏洞被引入的时间是风险代码和编译器发布时间的较大值。

Table 1 Residence Time of Compiler-Introduced Security Issues in Linux Kernel Code

表 1 Linux 内核代码中编译器引入的安全问题的滞留时间

Linux commit ID	代码发布年份	漏洞类型	编译器发布年份	修复年份	漏洞持续时间/年
809850b7a5fcc	2013	No-UB	2006	2017	4
503358ae01b70	2009	No-UB	2011	2012	1
1da177e4c3f41	2005	No-UB	2006	2006	0
0f22aab897159	2008	No-UB	2006	2009	1
1da177e4c3f41	2005	No-UB	2020	2018	0
cca2e6f87e385	2008	No-UB	2006	2009	1
1da177e4c3f41	2005	DSE	2006	2021	15
fb34a08c3469b	2009	Default	2010	2010	0
9b1be4dc02bb6	2013	Env	2014	2021	7
b9c5d6a643589	2012	Env	2012	2016	4
7ddfe625cbc14	2015	Env	2007	2015	0
0a488a53d7ca4	2008	Env	2007	2008	0
9d0fd1eb8a3c1	2005	DSE	2006	2005	0
3c49b52b155d0	2014	DSE	2006	2020	6
00f496c416122	2016	DSE	2006	2016	0
ffd8d3fa58134	2008	DSE	2006	2014	6
76cb9521795a1	2008	DSE	2006	2014	6
1da177e4c3f41	2005	DSE	2006	2020	14
265a5b7ee3eb2	2011	DSE	2006	2016	5
541af946fe136	2014	DSE	2006	2014	0
d4c5efdb97773	2014	DSE	2006	2015	1
9b1be4dc02bb6	2013	DSE	2006	2015	2
1da177e4c3f41	2005	DSE	2006	2021	15
fbf9482911825	2009	Padding	2007	2009	0
78f821b648267	2014	Padding	2007	2015	1
4262a727621ce	2012	Padding	2007	2017	5
55dd00a73a518	2017	Padding	2007	2018	1
9fb0f14e31b61	2011	Padding	2007	2019	8
1f5e31d7e55ac	2009	Padding	2007	2020	11
1da177e4c3f41	2005	Reorder	2006	2007	1
平均					3.83

4 系统设计与实现

为了跨版本(编译器版本或者源代码版本)地检测编译器引入的安全漏洞,本文设计静态分析方法,作出了以下改进。

1)针对挑战 1(如何针对通用的安全问题),本文基于错误处理代码的安全性建模,针对广泛存在的错误检查,建立通用的安全代码模式。此方法的通用性体现在:各种类型的安全问题都需要“错误处理代码”,不限于特定的安全场景和漏洞类型。规范的错误处理逻辑(如输入验证、资源释放、权限检查等)能够覆盖大多数安全问题的防御场景,能够针对各种类型的软件漏洞(如缓冲区溢出、释放后使用、整形溢出等)。这表明通过错误处理代码建模能够捕捉到跨漏洞和场景的通用缺陷模式。这些模式在高质量代码(如 Linux 内核)中表现尤为明显,其错误处理代码的完整性与安全性密切相关。基于这一认识,我们通过识别这些通用的安全代码模式,构建了一套不依赖于特定漏洞特征的安全分析框架。

2)针对核心技术挑战 2(如何追踪编译器引入的语义变化),本文对识别到的安全代码进行标记,并构建跨编译流程的语义差异分析系统,判定当前测试版本(源代码/编译器)中间表示代码与目标代码是否存在差异,即安全代码在编译后是否被去除,将被去除的安全代码报告为疑似的安全问题,这构成了本文的核心创新点。

3)针对挑战 3(如何判定编译器优化的安全性),本文引入差分测试系统。在对单个版本找到所有疑似的安全问题后,利用差分测试分析 2 个版本之间

的差异，并定位潜在的安全漏洞。此设计的核心思路是：编译器对单个版本中安全代码的处理（如对安全代码的去除）是可能的安全问题；2 个版本间如果存在某个版本专有的这种可能的安全问题，则很可能为实际的安全漏洞。理由是：①不同版本的编译器对安全代码的处理（比如对安全代码的去除）应该是一致的，因为安全代码的语义存在性是代码的关键属性，编译器不应该改变这种属性，且实证研究表明[34]，主流编译器在版本迭代中会保持对安全关键代码的稳定处理。因此，如果大部分版本没有去除的安全代码被某个新版本的编译器去除了，则很有可能发生了安全问题。②更新后的源程序中新出现的编译器对安全代码的激进处理（比如对安全检查的去除）往往比代码中既有的系统问题更有安全风险，因为新引入的代码更缺少测试。

4）针对挑战 4（如何应对编译器和各类程序的快速迭代），本文的方法建立在前 3 点之上，通过结合安全代码建模、跨编译流程的分析系统以及差分测试，形成一套能够适应编译器和程序版本快速迭代的检测方案。

基于此，我们设计了如图 6 的检测流程：1）基于对安全检查的建模，实现安全代码的定位并进行中间表示（IR）语言的级别标记，以应对通用的安全问题；2）将 IR 语言的代码编译到目标代码，并构建跨编译流程的语义差异分析系统，最终判定当前测试版本（源代码/编译器）的安全代码在编译后是否被去除，将被去除的安全代码片段报告为疑似的安全问题；3）基于差分测试的测试准则，通过比对不同版本的疑似安全问题，识别新版本独有的潜在安全漏洞，并报告给安全维护人员。

我们基于 LLVM 实现了这样的跨版本的编译器引入型漏洞的检测系统。其中安全代码定位与标记部分在 CheQ[55] 的基础上新增 982 行 C++代码，跨编译流程分析系统在 LLVM 14 的基础上新增 2 096 行 C++代码，差分测试实现为 2 888 行 Python 代码。

4.1 安全代码定位与标记

本文的方法首先识别错误处理代码，以此为基础定义通用安全代码的模式。具体来说，我们的安全代码是基于错误处理代码的扩展，在 CheQ[55] 的错误代码识别和安全检查的基础上，另外定位对被检查变量的第一次读取操作，将安全检查错误、错误处理跳转语句、检查后首次读共同构建为通用安全代码。并在后续分析追踪它们在编译过程中的变化，以检测编译器引入的安全漏洞。

4.1.1 安全检查的定义

本文采用 CheQ[55] 的安全检查定义，该定义基于安全检查的内在特性，而非特定程序的实现方式。安全检查的主要作用是捕获错误，因此其条件判断（如 if 或 switch 语句）通常用于指示异常状态。一旦错误被捕获，至少有 1 个分支用于错误处理，另一个分支则继续正常执行。

形式化地，设 B_c 为包含条件语句的基本块，并且其具有多个后继基本块 B_s, B_{si}（其中 i 是索引）。设 B_r 为当前函数内包含 return 语句的基本块，定义从 B_c 开始，经过 B_{si} 并最终到达 B_r 的代码路径序列为 [B_c, B_{si}, B_r]。由于 B_c 和 B_r 之间可能存在多条子路径，因此 [B_c, B_{si}, B_r] 可能涵盖多个包含 B_c, B_i, B_r 的代码路径。在此基础上，我们用 S_i 表示 B_c 的第 i 个分支，并定义错误处理（FH）性质 P 如下：

如果 [B_c, B_{si}, B_r] 至少包含 1 个必须处理错误（MUST-FH）的分支，同时至少包含 1 个可能处理错误的分支，同时至少包含 1 个可能处理错误（MAY-FH）或不处理错误（NO-FH）的分支，则认为 B_c 处的条件语句是一个安全检查。换言之，满足上述条件的条

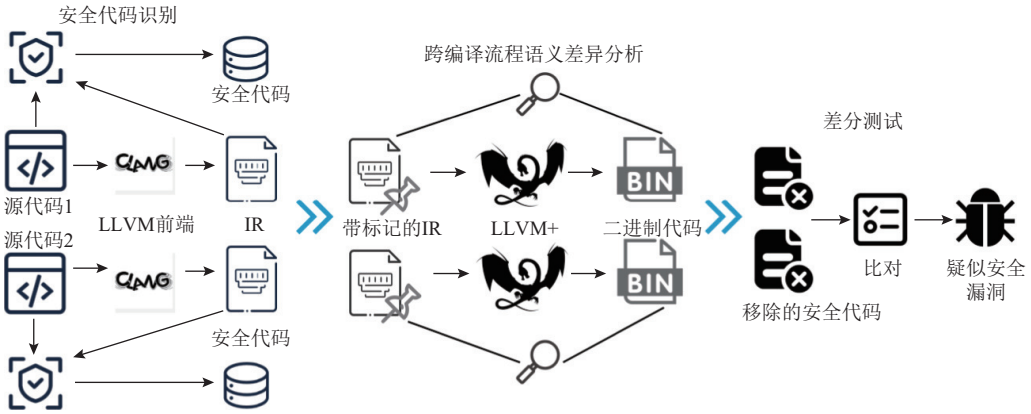


Fig. 6 Flow chart for detection

图 6 检测流程图

件语句可被视为安全检查。

4.1.2 安全代码的定位

我们首先基于 CheQ[55] 实现对编译器中端优化前的 IR 中的安全检查的定位，具体如算法 1 所示，通过分析函数的基本块和控制流，标识出可能用于错误处理和安全检查的指令。该部分实现为基于 LLVM 的 IR 语言的分析工具。该方法为操作系统内核设计，但是同时也适用于所有显式地定义了错误处理函数的 C 语言编写的大型软件。该分析工具的输入为源代码以及一份基本错误处理函数列表。以操作系统内核为例，错误处理函数包括用于中止执行和发出错误消息的函数。该方法的具体步骤为：1)全面识别错误处理函数和错误代码。基于输入的基本错误处理函数，在待测程序的优化前 IR 代码(O0)中迭代地分析这些函数的包装(warpper)函数并纳入错误处理函数列表。2)识别基本块的错误处理原语，包括“错误处理块”“错误返回块”“错误修复块”。3)构建错误处理关系图，并根据安全模型查询该图识别出安全检查。定位到安全检查后，我们可以进一步跟踪并确定与每个安全检查相关的变量，获取基本块中对该安全检查所检查的变量的第 1 次读取操作，为此，我们使用广度优先搜索(BFS)算法，从安全检查所在的基本块开始，沿着控制流图的路径查找，直到发现目标变量的第 1 次读操作。由此，得到了我们所关心的安全代码，即所有相关的安全检查和它们所检查的变量的第 1 次读取操作。并将安全代码相关信息序列化至 protobuf 文件中，以供后续步骤使用。

算法 1. 在函数中识别安全相关代码算法。

输入：函数 *F*，基本块错误映射 *bbErrMap*；

输出：安全相关代码集合 *SCSet*。

① 标记控制流边的错误传播状态；
② for *inst* in {*F*}
③ 初始化条件变量 *cond* 为空；
④ if *inst* 是分支或 switch 指令
⑤ 获取 *inst* 所在的基本块 *bb*；
⑥ 初始化计数器 *must_return_err*=0, *may_return_err*=0, *must_handle_err*=0, *may_handle_err*=0；
⑦ for *bb* 的每个后继块 *succ*
⑧ 从 *edgeErrMap* 中获取边(*inst*, *succ*)的错误标志；
⑨ 根据标志更新计数器；
⑩ end for
⑪ if(*must_return_err* > 0 and *may_return_err* > 0)or(*must_handle_err* > 0 and *may_handle_err* > 0)
⑫ *cond*←*inst*.*getCondition*(); /*提取分支条件作为安全检查条件*/
⑬ end if
⑭ if *cond*
⑮ 获取 *cond* 对应的关键变量映射；
⑯ if *inst* 是分支或 switch 指令
⑰ 在 *bb* 中拓展关键变量集；
⑱ for *bb* 的每个条件后继 *succ*
⑲ if not *succ* 为出错路径
⑳ 沿 *succ* 寻找关键变量的首次读操作；
㉑ end if
㉒ end for
㉓ end if
㉔ 将安全检查和读操作信息加入 *SCset*；
㉕ end if
㉖ end if
㉗ end for
㉘ return *SCset*。/*返回安全相关代码*/

同时，为了追踪编译流程中安全检查的变化，我们需要对定位到的安全检查进行标记并作为后续跨编译流程分析的基础。

4.1.3 安全代码的标记

识别到安全代码后，我们进行安全代码的标记。该部分实现为 LLVM 的 Utility Pass 被整合到 LLVM 的编译流程中。在进入中端优化前进行安全代码的识别与指令标签的标记。该 Pass 会遍历待测 IR 代码的每条指令，从上一步输出的对应安全代码 Protobuf 序列化文件中查询当前指令是否是安全代码，并根据上述安全标记的格式标记好安全指令。安全标记被实现为 LLVM DebugLoc 类的一部分，它会贯穿 LLVM 编译流程始终，将指令标签的信息在编译期间进行传递。

此外，除了基于安全检查的自动化安全代码标记，我们还实现了一套源代码级别的手动安全代码标记方案。测试者可以在源代码通过代码注释插入标记“no-remove”“no-reorder”以标记不希望相应代码被删除或重排序，我们在 LLVM 前端搜寻并处理这种代码标记，并将被标记的代码相关指令的信息传到 LLVM 中端 IR 代码中，然后如同以上的安全代码标记过程一同标记。具体来说，我们在 LLVM 前

端的注释处理函数中检查约定的安全代码标记，并将有标记的源代码的行号信息存在一个前端的全局对象中（如 ASTContext）。此后，在完成 AST 分析、IR 生成后，删除 AST 和发射 IR 前，遍历 IR 指令并给源代码行号匹配的 IR 指令打上标记到 Metadata。之后的安全代码标记阶段会检查这里的 Metadata 并进行最终的安全代码标记。

4.2 跨编译流程分析

识别到安全代码后，我们需要检测是否在经过编译器编译后对安全代码的处理发生了改变，比如安全代码被去除或重排序。然而，在编译过程中，安全代码的变化跟踪是一个复杂且具有挑战性的问题。主要的挑战之一是如何将 IR 与最终生成的二进制代码建立精确的对应关系。具体而言，LLVM IR 与汇编指令并非一一映射，例如 cmp 在 IR 级别用于比较 2 个数，但在 binary 级别可能被优化为条件跳转指令（cmp+jmp）、条件设置指令（setcc）、位运算优化（如 and, test 代替 cmp）；由于编译器后端优化的影响，会发生基本块级别的合并以及指令跨基本块的变动，因此无法简单地直接使用 Basic Block 级别的对应，且 LLVM 提供的 IR 到基本块的映射信息也会变得不完善；由于编译器优化引入的控制流合并，来自不同执行路径的安全检查可能会合并至同一处，使得原有的安全代码关系难以还原；调试信息本可以辅助对应的过程，但是调试信息本身可能有错误和遗漏，比如函数内联可能会导致原来的指令出现在不同的位置，甚至被展开到多个位置而调试信息仍然指向原函数的行号，以及优化阶段去除的指令可能缺乏调试信息。

为了应对这些挑战，本文设计了跨编译流程分析系统，旨在准确追踪编译流程中安全代码的变化。具体来说，首先在编译优化前对安全代码进行指令级别的标记，并集成到代码指令的编译时数据结构中。其次，考虑到编译器优化对指令的影响，我们需要修复一些中断的标记传递。最后，在目标代码级别，我们可以根据传递到最终目标代码的标记信息以及有关指令的类型，判断我们关注的安全代码是否发生了变化：安全检查是否被去除。

然而，IR 与二进制代码的映射存在许多困难，为了更好地处理这些问题，本文提出以下解决方案：1）基于测试集总结特殊指令与 IR 的对应关系。我们首先构建一个测试集，收集特定安全代码模式在 IR 与二进制代码中的转换对应关系。通过该测试集，我们归纳这些模式的转换规则，以提高匹配的准确性。2）上下文整体匹配。选择性使用 LLVM 提供的 IR 到基本块的对应关系，将上下文看作一个整体。如果一个基本块在 IR 中被拆分成为多个基本块（或者多个 IR 基本块被合并为一个），可以尝试以多个基本块为单位进行匹配，而不仅仅单独分析一个基本块。结合指令类型信息来进一步提高匹配的精确度。3）控制流引导匹配。利用上下文中的指令结构控制流图（CFG）找到对应指令的对应位置。例如，如果 cmp 指令丢入，可以使用其前后指令作为特征，结合变量的活跃范围以及 SSA 形式推测原始 IR 所在的位置。4）增强调试信息的利用。使用多层次的调试信息，尽可能保留原始 IR 的语义，并结合控制流位置与上下文指令特征建立对应关系，以应对优化带来的信息缺失。

为了确保我们能够准确追踪编译过程中安全代码的变化，并进行编译前后的比对，我们将待测试的编译流程限定到中端优化前和后端优化后。这样的设计几乎不会减少我们可检测的安全漏洞的范围，理由是：中端之前和后端之后主要是源代码到 IR 代码以及机器中间表示代码（MIR）到二进制代码之间的转化，规则比较固定，没有优化发生，所以出现安全问题的几率很低。同时与从二进制代码中提取信息相比，这样做不需要进行反汇编分析，在效率和准确性上都有优势。

首先，对安全代码进行标记。在 4.1 节，我们已经有了某些 IR 指令是否是安全相关代码的标记，然而这种标记还不满足编译前后的比对的需求。为此我们设计了一种新的指令标记。这种指令标记分为 2 部分：IR 指令的唯一标识和安全标识。新采用的 IR 指令的唯一标识可以被集成到编译流程中并最终被传递到目标指令代码。由此可以帮助建立从 IR 代码到目标指令代码的关系。具体来说，我们使用一个包含了数个 IR 指令的属性的元组{文件名，函数名，基本块编号，基本块内的指令索引}作为指令的唯一标识。我们没有复用编译器分析系统中调试信息系统的源代码和行号列号信息，因为源代码级别的信息不能唯一地表示 IR 级别的指令。指令标识的安全标识即为 IR 指令是否是安全相关代码的标记。

其次，修复中断的标记传递。现代编译器都存在调试信息模组，每条 IR 语言级别的指令都有相应的调试信息结构。这样的数据结构会跟随编译流程并抵达最终的目标代码。然而，编译器的调试信息模组存在底层的缺陷，因而在面对复杂的编译器优化时，标记的传递会被中断，如图 7 所示。引起标记

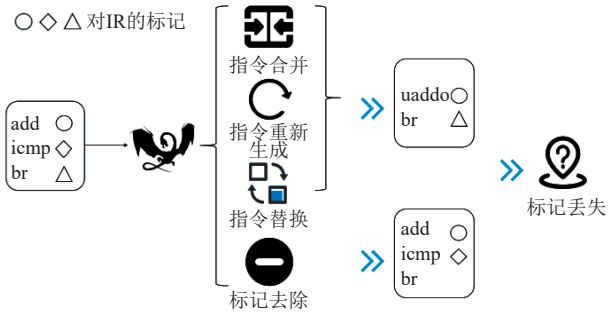


Fig. 7 Tags lost due to compiler optimization

图 7 编译器优化导致的标记丢失

传递中断的编译器优化行为可以分为如下 3 类：1)指令的合并。由于每条指令只能存储 1 条调试信息，指令的合并必然导致其中一条指令标记的丢失。2)指令的重新生成。编译器优化很多时候在进行指令转换或者拷贝时会重新生成新的指令，这些新生成指令许多并没有继承相应的标记。3)其他伴随调试信息缺失的优化行为。编译器调试信息系统自身的一些缺陷或者保守措施也会导致信息的缺失。为了修复这些中断的标记传递，我们采取了如图 8 所示的 2 项措施：1)支持单条指令用于多个调试信息标记，以确保机器指令在继承 IR 信息时不会有优化过程而丢失关键的安全标记；2)在标记发生中断的编译器优化过程补上标记的传播。此外，我们对标记的传播采取了比较宽松的传播规则：所有由已标记指令拆分、拷贝、合并得到的指令都会继承标记。

最后，判断实际的安全代码的变动。为了保证标记的传播，我们采用了比较宽松的传播规则，因此持有相关标记的目标代码指令中可能会包含无关的指令。为了解决这个问题，我们在机器代码级别基于指令类型筛选目标指令。由于 IR 代码在编译到机器代码时，通常只会映射到少数几类语义相似的指令，我们可以利用指令对应关系，精准识别到最终的目标指令。换句话说，仅依赖安全标记可能会导致误识别，而结合指令类型匹配能有效排除无关指令。原因是，持有标记的指令必然来源于原指令的拆分、拷贝或合并，不会凭空产生，并且如果某条指令同时满足持有安全标记且类型匹配 IR 级别的安全代码，那么它很难是完全无关的指令。通过这种方式，我们可以去除绝大多数误标记的指令，确保最终识别到的安全代码变动具有高准确性。

基于此，我们修改了 LLVM 的中端和后端优化，接收自动化方法得到的安全代码的 Protobuf 信息文件和(被标记的)源代码文件作为输入，并在编译阶段的 MIR 优化结束、代码发射之前输出 MIR 指令的标记信息到 Protobuf 信息文件。如前文所述，由于存在编译器优化导致标记断传的情况，我们通过迭代式的开发方式，根据标记断传的实例调试 LLVM 来判定需要修复的标记传递。我们以 Linux 内核代码作为测试代码，测试安全检查的标记的传递。我们最终修复的标记传递详见表 2。

4.3 差分测试

如何判定编译器优化的安全性是检测编译器引入漏洞的核心挑战之一，尤其是对于静态分析方法。本文的核心思路是首先分别定位 2 个相近版本的编译器和源代码中的疑似的编译器引入的安全问题(例如去除了源代码中某个安全检查)，再选择 2 个版本差分测试的结果，也就是只在单独某个版本(尤其是新版本)被报告的安全问题，作为最终报告的安全漏洞。

这个思路基于的核心观察是：1)不同版本的编

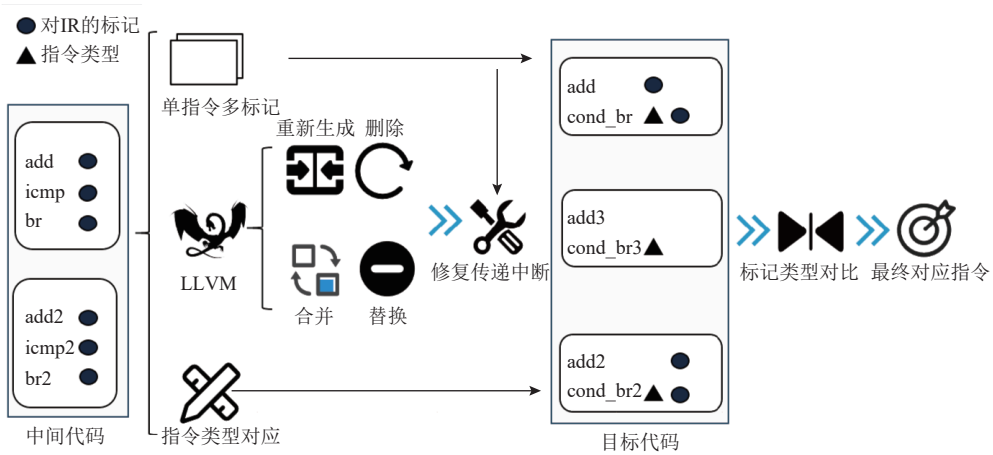


Fig. 8 Fix the broken tag passing

图 8 修复中断的标记传递

Table 2 Fixed LLVM Mark Pass

表 2 修复的 LLVM 标记 Pass

标记丢失类型	修改 Pass	修改文件名	函数名
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*performBranchToCommonDestFolding*
指令替换	GVN	llvm/lib/Transforms/Scalar/GVN.cpp	*processInstruction*
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*SwitchToLookupTable*
指令重新生成	MachineBlockPlacement	llvm/lib/CodeGen/MachineBlockPlacement.cpp	*optimizeBranches*
指令重新生成	JumpThreading	llvm/lib/Transforms/Scalar/JumpThreading.cpp	*unfoldSelectInstr*
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*HoistThenElseCodeToIf*
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*TurnSwitchRangeIntoICmp*
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*SimplifyBranchOnICmpChain*
指令重新生成	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*CloneInstructionsIntoPredecessorBlockAndUpdateSSAUses*
指令合并	SimplifyCFG	llvm/lib/Transforms/Utils/SimplifyCFG.cpp	*SimplifyCondBranchToCondBranch*
指令合并	JumpThreading	llvm/lib/Transforms/Utils/Local.cpp	*TryToSimplifyUncondBranchFromEmptyBlock*
指令重新生成	JumpThreading	llvm/lib/Transforms/Utils/Local.cpp	*ConstantFoldTerminator*
标记去除	JumpThreading	llvm/lib/Transforms/Utils/Local.cpp	*hoistAllInstructionsInto*
指令合并	DAGCombiner	llvm/lib/CodeGen/SelectionDAG/DAGCombiner.cpp	*foldSelectOfConstants*
指令重新生成	IfConversion	llvm/lib/CodeGen/IfConversion.cpp	*reverseBranchCondition*
指令合并	BranchFolding	llvm/lib/CodeGen/BranchFolding.cpp	*FixTail*
指令重新生成	Inliner	llvm/lib/Transforms/Utils/InlineFunction.cpp	*inlineDebugLoc*

译器对安全代码的处理应该是一致的；2)新的代码中的疑似编译器引入的安全问题有着更高的处理优先级和长久的危害等级。之所以还需要差分测试确认安全漏洞的原因是源代码层面可能存在一些冗余的安全代码且编译器的优化也会拷贝或者重新组合一些安全代码。以上 2 种情况下，编译器对这些冗余安全代码的去除和简化都是合法和安全的，但是仍然会被单版本的跨编译前后的分析报告为疑似的安全问题，而且难以与真正的问题区分开。

修复安全标记需要首先确认相关的编译器优化没有去除安全代码，然后为缺失标记的指令添加对应的标记。图 9 展示了一个被修复的 LLVM 标记缺失，这段代码属于 SimplifyCFG，是一段被多次迭代的对控制流的优化过程。第 5 行对条件分支指令进行了合并，因为每条指令的初始设计是只能包含 1 条调试信息，所以这必然会导致其中一个分支指令丢失其安全标记。我们在这里获取被合并指令(BI)的安全标记并加入到保留的合并指令(PBI)中。

```
1  llvm/lib/Transforms/Utils/SimplifyCFG.cpp
2  static bool SimplifyCondBranchToCondBranch(BranchInst *PBI, BranchInst *BI)
3  {
4      // Merge the conditions
5      Value *Cond=createLogicalOp(Builder, Instruction::Or, PBICond, BICond,
6                                  "brmerge");
7      PBI->appendInstIndexSet(BI->getInstIndex( ));
8
9      // Modify PBI to branch on the new condition to the new dests
10     PBI->setCondition(Cond);
11     ...
12 }
```

Fig. 9 An example of fixed LLVM marker pass

图 9 修复的 LLVM 标记传递示例

本文在差分测试方法中，严格控制 2 个核心变量：编译器版本和被编译程序代码版本。其余变量在差分测试中均需保持一致。为了增加差分测试的鲁棒性，每次分析只测试 1 个变量，另一个保持一致。

值得注意的是，尽管本方法需要修改编译器的数据结构甚至一些优化的调试信息传递，但是由于修改的数据结构都是比较通用的数据结构，修改代码的位置也位于相对确定的编译器优化逻辑的位置，因此我们的方法具有一定的通用性，可以直接适用或稍加修改适用于 LLVM 14 及之后的版本。

此处我们仅支持同编译器不同源代码版本或者不同编译器版本同源代码版本的差分测试。针对同编译器不同源代码版本的情况，我们只分析有变化的源代码来简化分析。我们可以以 2 个版本的源代码作为输入，首先通过 diff 得到 2 个版本的差异文件。我们只关注新版本源码中被新加入或者修改的代码，只对这部分代码进行安全代码定位和标记，并对这部分代码进行单次的编译流程比对分析，视被报告的疑似安全问题为差分比较得到的可能的编译器引入型漏洞。

针对不同编译器相同源代码版本的情况，我们分别先得到 2 个版本经过跨编译流程分析的疑似安

全问题,再以安全代码的IR级别的指令信息作为安全代码的唯一标识,进行2个版本的交叉比对,将输出只在单个版本被报告的安全代码变动并报告为可能的编译器引入型漏洞。

5 实验验证

5.1 实验目标选择

为了评估本文系统的有效性,我们以Linux kernel为主要目标待测源程序,原因是我们已知的目前最新的开源的真实世界中编译器引入型漏洞数据集中,最具代表性的是Xu等人[8]构建的数据集,且该数据集的真实代码部分是完全基于Linux kernel的。并且Linux kernel是一个复杂且广泛使用的开源项目,其代码库的庞大和多样性使其成为评估漏洞检测技术的理想候选目标。Linux kernel的重要性和广泛应用意味着对其安全性的分析具有实际意义。基于此,我们设计了2个评估实验,分别测试跨源代码版本和跨编译器版本2种情况下的漏洞检测效果。

5.2 实验设置

实验1. 跨源代码版本——重新引入风险代码。实验1旨在评估本文方法检测编译器引入型安全漏洞的有效性和通用性。具体包含2部分:

1)真实漏洞重现实验(有效性验证)。我们在Linux内核代码中重新引入了6个历史上曾经出现过的风险代码,通过对比漏洞引入前后的代码版本,检测编译器优化可能引发的漏洞。Linux内核的版本是5.17.4, LLVM的版本是14.0.3。

如表3所示,这些风险代码选自Xu等人[8]的漏洞数据集中的相关编译器引入的安全漏洞。我们在Linux内核x86的默认配置选项的基础上修改编译器配置,确保这些风险代码以尽可能重新触发编译器引入型漏洞。在该数据集中所有LLVM相关的8个漏洞中,我们成功引入了6个,剩余的2个无法被当前的LLVM编译器触发。

Table 3 List of Reintroduced Kernel Risk Codes

表3 重新引入的内核风险代码列表

数据集漏洞编号	被修复的commit ID	修改文件名
b-8	a3ca86aea507	drivers/staging/agnx/pci.c
b-21	d4c5efdb9777	drivers/char/random.c; include/linux/string.h; include/linux/string.h
l-2	12051b318bc3	arch/mips/include/asm/io.h; arch/mips/kernel/setup.c
l-8	d50f2ab6f050	fs/ext4/super.c
l-13	beaec533fc27	include/linux/llist.h
l-30	1186e4e90483	arch/mips/kernel/traps.c
l-37	8690ba446def	drivers/video/aty/mach64_ct.c
l-41	94f6bac1058f	arch/x86/kernel/cpu/common.c

2)全漏洞模拟实验(通用性验证)。为全面论证本文所提方法针对各漏洞类型的通用性,我们进一步扩展了测试范围,使用了Xu等人[8]数据集中所有提供了测试用例的漏洞进行测试(总共34个)。对于其中无法迁移至Linux内核环境的用例,我们直接使用其原始代码进行测试,并应用4.1.3节中的方法手动标记安全代码。对于无法在当前待测LLVM版本重现的错误行为,我们通过模拟出错的编译器行为(即修改中端优化阶段的LLVM IR代码)来进行测试。

实验2. 跨编译器版本——UB优化前后。为了模拟可能引入安全漏洞的编译器优化更新,本实验将关闭特定UB优化选项的编译器作为初始版本,将开启相关优化的编译器作为更新版本。我们选择通过编译选项切换(而非直接切换编译器版本)来构造此类对比,主要基于以下理由:跨UB相关选项的测试能够覆盖更广泛的编译器行为变化,且这些变化与编译器引入的漏洞直接相关。这种确定性的代码变换能够更稳定地检验本文方法的有效性。相比之下,不同编译器版本之间与UB相关的变化往往具有随机性,并且需要源代码中已存在风险代码才能被触发,导致实际可检测到的问题范围有限。这种低频次、随机性的事件虽然不影响编译器侧的漏洞检测,但不利于系统性地验证本文方法的有效性。

此外,通过调整UB相关的选项,能够更充分地暴露内核中潜在的风险代码。对于特定类型的风险代码,不安全的编译配置能够确保编译器以高概率触发不安全行为。全面检测操作系统内核中的风险代码具有独立的意义,因为从根本上预防编译器引入型漏洞依赖于用更稳健的代码替代这些风险片段。即便在受到编译选项保护的情况下,这些风险代码仍可能因编译器随版本更新而引入新问题。Linux内核默认关闭了一批UB优化来避免编译器引入的安全漏洞。本实验通过修改内核的构建文件Makefile来重新激活这些优化,从而在固定编译器版本的前提下模拟不同优化配置下的行为差异,以此有效评估编译器优化对漏洞引入的影响。

具体来说,我们通过修改Makefile去除以下选项-fno- strict-aliasing, -fno-delete-null-pointer-checks, -fno-strict-overflow来激活编译器优化,以模拟一个更容易引入安全漏洞的编译器。待测的Linux内核的版本是5.17.4, LLVM的版本是14.0.3。

5.3 实验结果

在真实漏洞重现的实验 1 中，对于重新引入的 Linux 内核的 6 个历史漏洞，本文的检测方法成功找到了 6 个；在全漏洞模拟实验中，如果仅基于错误处理代码标记安全代码，本文方法可以覆盖 44.1% 的全部编译器引入型漏洞。如果手动标记所有安全代码，本文方法可以覆盖 73.5% 的全部编译器引入型漏洞。此外，根据 Xu 等人[8]对编译器引入型漏洞的分类，我们对比了如表 4 所示的防御编译器引入型漏洞工作，展示了本文方法面向编译器引入型漏洞具有优于相关工作的通用性。

Table 4 Coverage Capability Comparison of Different Automated Methods in Different Compiler-Introduced Vulnerabilities

表 4 不同自动化方法在各类编译器引入型漏洞中的覆盖能力比对

方法类型	方法	年份	目标漏洞类型	覆盖率/%
Runtime sanitizer	UBSan[15]	2015	ISPec1,2,3	30
	ThreadSan[16]	2009	ISPec1,2,3	14
	TySan[17]	2017	ISPec1,2,3	12
Dedicated security analysis	STACK[9]	2013	ISPec1	66
	Unisan[10]	2016	OSPec1	40
	Yang 等人[11]	2017	OSPec1	60
	van K-Hunt[13]	2018	OSPec2	60
	van Sprundel[12]	2018	OSPec1	60
	Wu 等人[56]	2020	ISPec1,2,3	38
	SpecFuzz[57]	2020	OSPec3	100
	SpecTaint[58]	2021	OSPec3	100
	KUBO[14]	2021	ISPec1,2,3	63
本文			ISPec1,2,3	79(89,0,75)*
			OSPec1	50
			OSPec2	100
			OSPec3	0

注：“*”中“79”表示覆盖了所有漏洞中的 79%，“89,0,75”分别表示对 ISPec1,2,3 的覆盖率为 89%,0,75%。

实验 2 最终基于跨编译器版本的差分测试比对 2 个编译器版本产生的检查报告，检测出了 287 处不一致，即潜在的安全漏洞。其中 Linux 内核默认设置（也就是没有相关 UB 优化的编译器）版本上检出了 6 248 个安全代码变动；Linux“更新”设置（重新激活了 UB 优化的编译器）版本上检出了 6 481 个安全代码变动。对以上变动进行差分测试得到 287 处不一致，报告为疑似的安全漏洞。经过最终人工确认，287 个疑似的安全漏洞中有 16 个真实的编译器引入的安全漏洞。此差分测试的精确率（precision）为 5.57%（16/287）、误报率（FPR）为 4.19%（271/6 465）。

5.4 漏报与误报分析

5.4.1 漏报分析

由于难以获取真实的漏洞 ground truth，确认本系统在真实世界的漏报率是不现实的。实验 1 的模拟引入漏洞的结果表明，本文方法具有比较低的漏报率。如果以所有编译器引入型漏洞作为目标，本方法漏报的主要来源是安全模型的局限性：1）错误处理代码不能覆盖所有的安全问题；此外，一些安全问题可以通过确定性的安全操作来解决，不需要设定额外的安全检查和错误处理分支。2）编译器可能通过代码去除之外的动作（比如改变代码顺序）引入安全问题。除此之外，本文方法还存在 2 种理论的漏报情况：1）修复的安全标记的传播可能是错误的。我们在修复安全标记的传播时的默认假设是，被涉及的编译器优化在被修复的位置不存在指令被实际去除的情况。我们人工确认了被修改的编译器优化是满足这种假设的，但是无法确认编译器会一直满足这种假设。未来的编译器如果在这个位置发生了实现的错误并且导致相关的指令被去除，则我们会遗漏这种情况。2）安全代码被复制多份，其中 1 份被错误地去除了。编译器在做控制流相关的优化时，可能会把 1 份代码复制到不同的控制流路径，如果该 IR 级别的安全代码的所有拷贝只有部分被去除，本文方法无法分析其中可能的错误去除。这种拷贝往往发生在 IR 标记安全代码之前，此时不会影响我们的分析，但这种代码拷贝或许也可能发生在我们标记安全代码之后。

5.4.2 误报分析

实验 2 的结果（精确率为 5.57%、误报率为 4.19%）可以帮助分析本文系统引入误报的情况。本文系统的误报分为 2 种：1）安全代码没有被实际去除，但是我们报告为去除；2）安全代码被去除了，但是这样的去除是正确的，不会引起安全问题。在实验 2 中，第 1 种误报占总体报告的 23.0%，第 2 种误报占总体报告的 71.4%。

第 1 种误报的来源主要是我们的系统没有处理一些复杂的编译器优化相关的调试信息缺失。如图 10 的例子，源代码中第 5 行的检查和第 11 行的安全检查是重复的，编译器会重新排布控制流并合并 2 个检查。这里的第 11 行的检查语义还存在，但是指令信息却丢失了。

第 2 种误报的来源主要是源代码中一些冗余的

```
1  // net/wireless/nl80211.c
2  static int nl80211_connect(struct sk_buff *skb, struct genl_info *info)
3  {
4    ...
5    if (info->attrs[NL80211_ATTR_HT_CAPABILITY_MASK]) // 安全检查1
6      memcpy(&connect.ht_capa_mask,
7             nla_data(info->attrs[NL80211_ATTR_HT_CAPABILITY_MASK]),
8             sizeof(connect.ht_capa_mask));
9
10   if (info->attrs[NL80211_ATTR_HT_CAPABILITY]) {
11     if (!info->attrs[NL80211_ATTR_HT_CAPABILITY_MASK]){ // 安全检查2
12       kfree_sensitive(connkeys);
13       return -EINVAL;
14     }
15     memcpy(&connect.ht_capa,
16            nla_data(info->attrs[NL80211_ATTR_HT_CAPABILITY]),
17            sizeof(connect.ht_capa));
18   }
19   ...
20 }
```

Fig. 10 Linux example of flag loss caused by compiler merge optimization

图 10 编译器合并优化导致的标记丢失的 Linux 案例

安全检查或者编译器优化过程中产生的冗余安全代码。如图 11 所示，由于第 15 行的函数 *alloc_mark*(定义在第 2 行)被 inline，第 6 行检查的 2 个分支各自会有 1 份 16 行的检查。在 *alloc_mark* 本该返回 NULL 的分支，第 16 行的检查是可以被推断并且被去除的。这里的冗余安全检查是因为 inline 优化产生的。

```
1  // kernel/audit_tree.c
2  static struct fsnotify_mark *alloc_mark(void)
3  {
4    struct audit_tree_mark *amark;
5    amark = kmem_cache_zalloc(audit_tree_mark_cachep, GFP_KERNEL);
6    if (!amark)
7      return NULL;
8    fsnotify_init_mark(&amark->mark, audit_tree_group);
9    amark->mark.mask = FS_IN_IGNORED;
10   return &amark->mark;
11 }
12 static int create_chunk(struct inode *inode, struct audit_tree *tree)
13 {
14   ...
15   mark = alloc_mark();
16   if (!mark) {
17     mutex_unlock(&audit_tree_group->mark_mutex);
18     kfree(chunk);
19     return -ENOMEM;
20   }
21   ...
22 }
```

Fig. 11 Redundant safety checks removed by the compiler

图 11 被编译器去除的冗余的安全检查

实验 2 针对 Linux 内核最终报告的疑似漏洞数量可以满足实际的测试需求，我们考虑到 Linux 内核的代码量，以及去除的编译选项影响到的优化的体量，认为本系统在面对大型软件系统和复杂的编译器变化仍然可以维持一定的精度，满足实际的可用性(详见 5.7 节)。例如，实验 2 中，Linux 总体的代码量达到 2 700 万行，实验分析的条件检查共有 264 966 个，其中安全检查有 51 463 个，说明该实验分析的代码基数很大；报告了 4.6% 的安全检查去除，说明 2 个版本间存在很大的编译优化的变化；而面对这种体量的代码和编译器变化，本系统最终报告的疑似漏洞数量是 287，可以通过手工分析较快确认实际问题，根据实际人工确认过程的统计，平均每个问题由单个安全分析人员完成确认所需时间小于 4 min；若按单人顺序处理全部问题折算，总耗时小于 20 h。

5.5 新报告的漏洞分析

对于以上评估实验报告确认的安全漏洞，除了我们重新引入的风险代码，通过人工分析最终确认了 16 处可能会被编译器引入漏洞的风险代码，这些风险代码中，目前已有 16 处被 Linux 内核开发者确认。目前被确认的漏洞都是因为 Linux 内核中还存在一些包含了 UB 的风险代码。由于编译器的编译选项设置，这些风险代码目前并不会被编译器引入实际的安全问题。但随着编译器的迭代发展，这些代码还是存在被编译器影响的可能。因此，编译器开发者会推荐修复程序中的这种风险代码。

图 12 是一个本系统检测到的目前 Linux 内核代码中仍然存在的风险代码的例子。第 7 行的 *len* 是一个 loff_t(long long)类型的变量，第 9 行检查 *len* 的计算过程有没有发生溢出并作处理。但是第 7 行存在 UB，所以不采取防御性编译选项的编译器会把第 9 行的检查去除，导致错误的 *len* 值会在第 11 行后被继续使用造成问题。

```
1  // fs/hugetlbfs/inode.c
2  static int hugetlbfs_file_mmap(struct file *file, struct
3  vm_area_struct *vma)
4  {
5        ...
6        vma_len = (loff_t)(vma->vm_end - vma->vm_start);
7        len = vma_len + ((loff_t)vma->vm_pgoff << PAGE_SHIFT);
8        /* check for overflow */
9        if (len < vma_len)
10         return -EINVAL;
11       ...
12 }
```

Fig. 12 Risky code in the Linux kernel

图 12 Linux 内核的风险代码

5.6 性能分析

本系统在编译阶段检测漏洞，因此对待测程序不存在运行时开销。我们在 Intel® Xeon® CPU E5-2620 v4@2.10 GHz, 64 GB 内存的环境对本文系统的分析开销进行了评估。作为基线，LLVM 14.0.3 直接编译 Linux-5.17.4 花费的时间为 67 min 19 s。应用了本系统后，编译同样的内核需要 148 min 30 s。此外，定位和标记安全代码需要 5 min 36 s，差分测试需要 10 min 6 s。由于单个版本只需要编译 1 次，每次版本更新需要进行 1 次差分测试，因此本系统并不会带来显著的编译时开销和分析开销，可以适用于大型的软件系统。

5.7 可用性分析

影响本文方法可用性的主要因素是误报的数量。首先，本方法的实际的误报规模满足工业界的可用

性要求。静态分析方法在工业界的可用性标准是：面对每日的程序更新，反应时间为 8~12h[59]。本文的检测工具的反应时间分为系统分析时间、额外的编译开销、误报处理时间以及潜在的跨编译器版本迁移开销 4 部分。

对于 Linux 内核（被编译的源程序）的每日代码更新，本文工具的预计反应时间约为 1.82 h，满足可用性的要求。具体来说，实验 1 引入的 8 个漏洞带来的代码变化近似为 8 次代码提交；此外统计得到近 1 年内 Linux 内核的每日代码提交量为 212.8 个。为了模拟 Linux 的每日更新的量级，实验 1 的与代码提交数量线性相关的系统分析时间和误报处理时间（40 s）随比例近似为 1 064 s。再加上 2 次分析安全代码的时间开销（5.5 min×2）和 1 次额外的编译开销（81 min），本文工具针对 Linux 内核的每日代码更新的反应时间约为 1.82 h，满足 8~12 h 的可用性需求。

对于 LLVM（编译器）的每日代码更新，本文工具的预计反应时间约为 9.07 h，满足可用性的要求。具体来说，实验 2 中重新引入的 3 个编译选项相关的代码量，可以保守地预估为 3 日的代码工程量。实验 2 需要 1.4 h 的额外编译时间和 0.3 h 一共 2 次分析安全代码的时间开销，以及 20 h 的人工分析和与代码量线性相关的系统分析时间。测试不同编译器版本本身并不存在显著困难，但需要额外的工程投入，将“跨编译器流程分析”的实现迁移至目标编译器版本。由于所修改的部分主要涉及扩展编译器的基础数据结构和插桩基本优化行为，这些部分在不同版本间通常保持稳定，迁移工作大多可借助 git merge 半自动完成。本质上，代码迁移需要解决的是编译器代码语义的相似度匹配问题，即定位更新后原有插桩点代码逻辑的新位置。小版本更新通常不会改变编译器 API 或引入新功能，所需处理的合并冲突较少，单人处理时间一般低于 30 min。而大版本更新（约每 6 个月 1 次）可能涉及编译器 API 变更及新功能的引入，需要更多的迁移时间，根据过往版本迁移过程中的实际工作量统计，单次迁移的保守估计为 15～40 人时。综合大小版本更新，LLVM 的日均代码更新量所带来的额外迁移负担平均不超过 0.7 人时。此外，模拟到 Linux 的每日更新的量级，本文工具的预计反应时间为 9.07（1.4+0.3+20/3+0.7）h，满足 8~12 h 的可用性需求。

其次，当前误报的成因相对集中，因而具有进一步自动化处理的空间。例如，部分误报来源于特定优化导致的安全代码合并，可通过插桩相关编译过程辅助判定合并行为，从而在控制漏报风险的前提下降低误报数量。冗余安全代码被去除导致的误报（占全部报告的 71.4%）大部分可归到此类。尽管当前误报率没有造成显著的人工分析负担，但是如果有更高的自动化需求，可以使用这种方法降低误报量。

6 局限性分析

6.1 通用性分析

本文提出的基于错误处理代码建模安全问题的方法，其核心优势在于不依赖针对特定漏洞的建模，跟相关工作相比具备更多的通用性（见表 4）。然而，该方法也存在局限性，无法覆盖所有的安全问题。具体来说，分为 2 种情况：

1）不包含错误处理逻辑的安全代码。若目标代码中未显示包含错误处理逻辑，则本文方法无法覆盖此安全代码，导致相关漏洞无法被检测。

2）编译器通过代码去除之外的动作引入的其他安全问题。本文方法主要针对编译器优化过程中因代码去除引发的安全问题，对于编译器其他行为引入的安全问题，尚无法有效覆盖。

6.2 应用场景分析

由于本文提出的方法基于“跨版本检测”，所以具有一定的应用前提，但我们认为该方法在实际工程中仍具备重要价值，理由如下：

1）对于重要的系统程序，旧版本问题已基本稳定，新问题集中在新增或变更代码中。随着大型软件系统的迭代发展，旧版本中已暴露的关键安全问题大多已被社区或专家修复，当前的风险点更可能出现在新引入或修改的代码片段中。因此，聚焦版本差异进行漏洞检测不仅现实可行，而且更具成本效益。

2）版本迭代频繁，手工审计难以跟上节奏。在许多开源或商用系统中，版本更新频繁且变更量大，依赖人工审计不仅成本高，而且容易遗漏隐藏风险。我们的跨版本检测方法可自动聚焦变更点，辅助开发者第一时间发现由编译器引入的新型缺陷，大幅提升审计效率。

综上所述，跨版本检测不仅契合当前软件开发与维护的实际需求，也为自动化漏洞挖掘提供了一种高效且具有实践价值的技术路径。

6.3 迁移性分析

本文的研究对象目前只包括 LLVM 编译器框架，但是本文的核心思想具有普适性，可以迁移到其他

的主流编译器,但需要额外的工程性工作量。本方法需要待测的编译器满足2个条件:支持IR语言且编译器优化基于IR语言,编译过程中支持记录指令级别的DWARF调试信息。主流的C编译器如GCC,ICC均支持这些条件。迁移所需的工作量分3部分:1)“安全代码识别”不需要迁移。安全代码是程序的固有属性,不随编译器转移。基于LLVM框架定位的安全代码可以基于调试信息中的指令元数据标记到待测编译器的中间语言中。2)“跨编译流程标记传递”需要额外的迁移工程量。需要扩展目标编译器的IR语言的指令数据结构来记录额外的标记信息(以调试信息的形式),并在此基础上用和LLVM一样的方法修复中断的标记传递。3)差分测试部分不需要迁移,与具体的编译器无关。

6.4 检测发现的局限性

尽管本实验结果表明,本文所提出的方法能够有效识别出因编译器优化差异而暴露的潜在漏洞及其对应的风险代码,但本研究仍存在一些局限性,值得进一步讨论与完善。

首先,如5.2节所述,实验2中用于触发漏洞的编译环境是通过主动修改内核构建配置(如开启通常被默认关闭的UB优化)而构造的。这种做法虽然能系统性、高效地验证方法的检测能力,但其发现的问题在现实开发环境中可能因项目始终采用一致的保守编译选项而不会实际触发。然而,需要强调的是,这些被检出的风险代码本身是真实存在的,它们构成了潜在的安全隐患。修复这些代码中的UB,是从根本上提升代码健壮性、防御未来编译器未知变更所引入风险的核心手段。本研究的意义正在于为开发人员提供了定位此类深层代码缺陷的有效工具。

其次,在本文的实地部署与测试中,我们尚未能在当前主流的x86-64体系结构下捕获到由真实编译器版本迭代所直接引入的漏洞实例。这可能是由于:1)现代稳定版编译器在保守优化选项下的行为已相对成熟,此类问题本身具有偶发性和低概率性;2)大型开源项目(如Linux内核)对编译器版本更新持审慎态度,通常会采用经过充分测试的稳定版本,这在一定程度上规避了前沿编译器变更带来的立即风险。

这一事实表明,要全面验证本文方法对“在野”(in-the-wild)编译器缺陷的检测能力,需要在更广阔的项目源和更长期的编译器版本演进中进行持续监测。这也是我们未来工作的重点方向之一。

7 未来研究方向

1)高精度的编译器引入型漏洞的检测。本文提供了一种跨版本的编译器引入型漏洞的检测方法。然而,本文方法的精度难以用于对大型程序进行全面的检测,即在单个版本的二进制中发现漏洞。这方面的主要挑战在于编译器的优化干扰和安全问题判断准则的缺失。例如,源程序中本身可能存在冗余的安全代码,而编译器优化也可能进一步使部分安全代码变得冗余。因此,需要新的判定准则来判断这些安全代码的去除是否正确。

2)其他的安全代码建模方法。本文采用的安全代码的建模方法需要待测的程序有确定的错误处理代码,这适用于大型程序如操作系统内核,但仍有很多不适用的程序同样也受到编译器引入型漏洞的影响。

8 总 结

本文提出了一种能够跨版本检测编译器引入安全漏洞的新方法。根据观察发现,编译器或者待测源程序中被编译器新引入的安全代码的去除更容易是安全漏洞。基于这样的观察,本文系统的核心思路是差分比较不同版本二进制代码被去除的安全检查并根据跨版本差分测试的结果来确定安全漏洞。评估结果表明,本系统有较低的漏报率,面向编译器引入型漏洞具有优于相关工作的通用性,并且有满足实际可用性的报告精度,面对大型程序和复杂的编译器变化仍可以将可能的漏洞限定在比较小的范围内。实证研究表明,本方法可以在大型程序中高效地检测到跨版本的编译器引入型漏洞,并最终在Linux内核中发现并确认多处可能被编译器引入漏洞的风险代码。

作者贡献声明:徐坚皓负责研究方法设计、代码实现、论文写作;李林克负责代码实现、实验执行和分析;张昆博负责代码实现和实验分析;唐晨负责实验分析、论文修改和方案讨论;茅兵负责方案设计和方案讨论。

参 考 文 献

[1] National Information Security Vulnerability Database. Vulnerability classification specification[EB/OL]. 2021[2025-10-27]. http://www.

cnnvd.org.cn/web/wz/bzxqById.tag?id=2&mkid=2 (in Chinese)
（国家信息安全漏洞库. 漏洞分级规范[EB/OL]. 2021[2025-10-27]. http://www.cnnvd.org.cn/web/wz/bzxqById.tag?id=2&mkid=2）

[2] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis[C]//Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 480−491

[3] Li Zhen, Zou Deqing, Xu Shouhuai, et al. VulDeePecker: A deep learning-based system for vulnerability detection[C]//Proc of the 25th Network and Distributed System Security Symp (NDSS 2018). Reston: Internet Society, 2018: 1−15

[4] Lin Guanjun, Xiao Wei, Zhang Jun, et al. Deep learning-based vulnerable function detection: A benchmark[C]//Proc of the 21st Int Conf on Information and Communications Security. Berlin: Springer, 2020: 219−232

[5] Hu Song, Luo Jiachi, Wan Wenkai, et al. LHG-VD: A learnable hierarchical graph representation vulnerability detection method[J]. Journal of Computer Research and Development, 2025, 62(9): 2348−2361 (in Chinese)
（胡菘, 罗嘉驰, 万文凯, 等. LHG-VD: 一种可学习的层次化图表示漏洞检测方法[J]. 计算机研究与发展, 2025, 62(9): 2348−2361）

[6] D'Silva V, Payer M, Song D. The correctness-security gap in compiler optimization[C]//Proc of the 2015 IEEE Security and Privacy Workshops. Piscataway, NJ: IEEE, 2015: 73−87

[7] Li Shaohua, Su Zhendong. Finding unstable code via compiler-driven differential testing[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2023: 238−251

[8] Xu Jianhao, Lu Kangjie, Du Zhengjie, et al. Silent bugs matter: A study of compiler-introduced security bugs[C/OL]//Proc of the 32nd USENIX Security Symp. Berkeley, CA: USENIX Association, 2023[2025-09-09].https://www.usenix.org/system/files/usenixsecurity23-xu-jianhao.pdf

[9] Wang Xi, Zeldovich N, Kaashoek M F, et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior[C]//Proc of the 24th ACM Symp on Operating Systems Principles. New York: ACM, 2013: 260−275

[10] Lu Kangjie, Song C, Kim T, et al. Unisan: Proactive kernel memory initialization to eliminate data leakages[C]//Proc of the 23rd ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2016: 920−932

[11] Yang Zhaomo, Johannesmeyer B, Olesen A T, et al. Dead store elimination (still) considered harmful[C]//Proc of the 26th USENIX Security Symp. Berkeley, CA: USENIX Association, 2017: 1025−1040

[12] van Sprundel I. Memsad: Why clearing memory is hard[EB/OL]. 2018[2025-10-27]. https://media.ccc.de/v/35c3-9788-memsad

[13] Li Juanru, Lin Zhiqiang, Caballero J, et al. K-Hunt: Pinpointing insecure cryptographic keys from execution traces[C]//Proc of the 25th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 412−425

[14] Liu Changming, Chen Yaohui, Lu Long. KUBO: Precise and scalable detection of user-triggerable undefined behavior bugs in OS kernel[C/OL]//Proc of the 2021 NDSS Symp. Reston: Internet Society,2021[2025-12-04].https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-5_24461_paper.pdf

[15] Conti M, Crane S, Davi L, et al. Losing control: On the effectiveness of control-flow integrity under stack attacks[C]//Proc of the 22nd ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2015: 952−963

[16] Serebryany K, Iskhodzhanov T. ThreadSanitizer: Data race detection in practice[C]//Proc of the Workshop on Binary Instrumentation and Applications. New York: ACM, 2009: 62−71

[17] Finkel H. The Type Sanitizer: Free yourself from -fno-strict-aliasing[EB/OL]. 2017[2025-12-04]. https://llvm.org/devmtg/2017-10/slides/Finkel-The%20Type%20Sanitizer.pdf

[18] Bruening D, Zhao Qin. Practical memory checking with Dr. Memory[C]//Proc of the 9th Int Symp on Code Generation and Optimization (CGO 2011). Piscataway, NJ: IEEE, 2011: 213−223

[19] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation[J]. ACM SIGPLAN Notices, 2007, 42(6): 89−100

[20] Meta. A tool to detect bugs in Java and C/C++/objective-C code[CP/OL]. 2022[2025-10-27]. https://fbinfer.com/

[21] CppCheck developers. A tool for static C/C++ code analysis[CP/OL]. 2022[2025-10-27]. http://cppcheck.sourceforge.net/

[22] Yang Xuejun, Chen Yang, Eide E, et al. Finding and understanding bugs in C compilers[C]//Proc of the 32nd ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2011: 283−294

[23] Lidbury C, Lascu A, Chong N, et al. Many-core compiler fuzzing[J]. ACM SIGPLAN Notices, 2015, 50(6): 65−76

[24] Chen Junjie, Wang Guancheng, Hao Dan, et al. History-guided configuration diversification for compiler test-program generation [C]//Proc of the 34th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2019: 305−316

[25] Le V, Afshari M, Su Zhendong. Compiler validation via equivalence modulo inputs[J]. ACM SIGPLAN Notices, 2014, 49(6): 216−226

[26] Le V, Sun Chengnian, Su Zhendong. Finding deep compiler bugs via guided stochastic program mutation[J]. ACM SIGPLAN Notices, 2015, 50(10): 386−399

[27] Sun Chengnian, Le V, Su Zhendong. Finding compiler bugs via live code mutation[C]//Proc of the 31st ACM SIGPLAN Int Conf on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM, 2016: 849−863

[28] Romano A, Liu Xinyue, Kwon Y, et al. An empirical study of bugs in webassembly compilers[C]//Proc of the 36th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2021: 42−54

[29] Sun Chengnian, Le V, Zhang Qirun, et al. Toward understanding compiler bugs in GCC and LLVM[C]//Proc of the 25th Int Symp on Software Testing and Analysis. New York: ACM, 2016: 294−305

[30] Chen Yang, Groce A, Zhang Chaoqiang, et al. Taming compiler fuzzers[C]//Proc of the 34th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2013: 197−208

[31] Chen Yuting, Su Ting, Sun Chengnian, et al. Coverage-directed differential testing of JVM implementations[C]//Proc of the 37th

ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2016: 85−99

[32] Le V, Sun Chengnian, Su Zhendong. Randomized stress-testing of link-time optimizers[C]//Proc of the 25th Int Symp on Software Testing and Analysis. New York: ACM, 2015: 327−337

[33] Chen Junjie, Patra J, Pradel M, et al. A survey of compiler testing[J]. ACM Computing Surveys, 2020, 53(1): 1−36

[34] Lee J, Kim Y, Song Y, et al. Taming undefined behavior in LLVM[J]. ACM SIGPLAN Notices, 2017, 52(6): 633−647

[35] Dave M A. Compiler verification: A bibliography[J]. ACM SIGSOFT Software Engineering Notes, 2003, 28(6): 2−2

[36] Fedyukovich G, Gurfinkel A, Sharygina N. Incremental verification of compiler optimizations[C]//Proc of the 6th Int Sym on NASA Formal Methods. Berlin: Springer, 2014: 300−306

[37] Leroy X. Formal verification of an optimizing compiler[C/OL]//Proc of the 5th ACM & IEEE Int Conf on Formal Methods and Models for Co-Design. Berlin: Springer, 2007[2025-12-05]. https://memocode.irisa.fr/2007/leroy-compcert-memocode07.pdf

[38] Leroy X. Formal verification of a realistic compiler[J]. Communications of the ACM, 2009, 52(7): 107−115

[39] Lopes N P, Lee J, Hur C K, et al. Alive2: Bounded translation validation for LLVM[C]//Proc of the 42nd ACM SIGPLAN Int Conf on Programming Language Design and Implementation. New York: ACM, 2021: 65−79

[40] Mansky W, Gunter E. A framework for formal verification of compiler optimizations[C]//Proc of the 1st Int Conf on Interactive Theorem Proving. Berlin: Springer, 2010: 371−386

[41] Besson F, Dang A, Jensen T. Securing compilation against memory probing[C]//Proc of the 13th Workshop on Programming Languages and Analysis for Security. New York: ACM, 2018: 29−40

[42] Almeida J B, Barbosa M, Barthe G, et al. Jasmin: High-assurance and high-speed cryptography[C]//Proc of the 24th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 1807−1823

[43] Barthe G, Blazy S, Grégoire B, et al. Formal verification of a constant-time preserving C compiler[J]. Proceedings of the ACM on Programming Languages, 2019, 4(POPL): 1−30

[44] Cauligi S, Soeller G, Brown F, et al. Fact: A flexible, constant-time programming language[C]//Proc of the 2nd IEEE Cybersecurity Development (SecDev). Piscataway, NJ: IEEE, 2017: 69−76

[45] Simon L, Chisnall D, Anderson R. What you get is what you C: Controlling side effects in mainstream C compilers[C]//Proc of the 3rd IEEE European Symp on Security and Privacy (EuroS&P). Piscataway, NJ: IEEE, 2018: 1−15

[46] Watt C, Renner J, Popescu N, et al. Ct-wasm: Type-driven secure cryptography for the Web ecosystem[J]. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 1−29

[47] Patrignani M, Guarnieri M. Exorcising spectres with secure compilers[C]//Proc of the 28th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2021: 445−461

[48] Sinha R, Rajamani S, Seshia S A. A compiler and verifier for page access oblivious computation[C]//Proc of the 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017: 649−660

[49] Almeida J B, Barbosa M, Barthe G, et al. Verifying constant-time implementations[C]//Proc of the 25th USENIX Security Symp. Berkeley, CA: USENIX Association, 2016: 53−70

[50] Agten P, Strackx R, Jacobs B, et al. Secure compilation to modern processors[C]//Proc of the 25th IEEE Computer Security Foundations Symp. Piscataway, NJ: IEEE, 2012: 171−185

[51] Patrignani M, Garg D. Secure compilation and hyperproperty preservation[C]//Proc of the 30th IEEE Computer Security Foundations Symp (CSF). Piscataway, NJ: IEEE, 2017: 392−404

[52] Vu S T, Heydemann K, de Grandmaison A, et al. Secure optimization through opaque observations[C/OL]//Proc of the 8th Principles of Security and Trust. New York: ACM, 2014[2025-12-05]. https://dl.acm.org/doi/pdf/10.1145/3062341.3062343

[53] Vu S T, Heydemann K, de Grandmaison A, et al. Secure delivery of program properties through optimizing compilation[C]//Proc of the 29th Int Conf on Compiler Construction. New York: ACM, 2020: 14−26

[54] Patrignani M, Ahmed A, Clarke D. Formal approaches to secure compilation: A survey of fully abstract compilation and related work[J]. ACM Computing Surveys, 2019, 51(6): 1−36

[55] Lu Kangjie, Pakki A, Wu Qiushi. Automatically identifying security checks for detecting kernel semantic bugs[C]//Proc of the 24th European Symp on Research in Computer Security. Berlin: Springer, 2019: 3−25

[56] Wu Zekai, Liu Wei, Liang Mingyue, et al. Finding bugs compiler knows but doesn't tell you: Dissecting undefined behavior optimizations in LLVM[EB/OL]. 2020[2025-10-27]. https://i.black-hat.com/eu20/Wednesday/eu-20-Wu-Finding-Bugs-CompilerKnows-But-Does-Not-Tell-You-Dissecting-UndefinedBehavior-Optimizations-In-LLVM.pdf

[57] Oleksii O, Bohdan T, Mark S, et al. SpecFuzz: Bringing spectre-type vulnerabilities to the surface[C]//Proc of the 29th USENIX Security Symp (USENIX Security 20). Berkeley, CA: USENIX Association, 2020: 1481−1498

[58] Qi Zhenxiao, Feng Qian, Cheng Yueqiang, et al. SpecTaint: Speculative taint analysis for discovering spectre gadgets[C]//Proc of the 30th USENIX Security Symp (USENIX Security 21). Berkeley, CA: USENIX Association, 2021: 1059−1076

[59] Madsen M, Livshits B, Fanning M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries[C]//Proc of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013: 499−509



Xu Jianhao, born in 1995. PhD, associate professor. His main research interests include security testing of software security, systematical security, and software infrastructure (compilers, operating systems).

徐坚皓, 1995 年生。博士，副研究员。主要研究方向为软件安全、系统安全、软件基础设施(包括编译器、操作系统)的安全性测试。



Li Linke, born in 1999. PhD candidate. His main research interests include software security and compiler security.

李林克, 1999 年生。博士研究生。主要研究方向为软件安全、编译器安全。



Zhang Kunbo, born in 1999. Master. His main research interests include software security and compiler security.

张昆博, 1999 年生。硕士。主要研究方向为软件安全、编译器安全。



Tang Chen, born in 2001. Master candidate. His main research interests include software security and reverse engineering.

唐　晨, 2001 年生。硕士研究生。主要研究方向为软件安全、逆向工程。



Mao Bing, born in 1967. PhD, professor, PhD supervisor. His main research interests include software security and system security.

茅　兵, 1967 年生。博士, 教授, 博士生导师。主要研究方向为软件安全、系统安全。