https://crad.ict.ac.cn
63(5): 1350−1375, 2026

LASM: 面向全程序分析的 x86 内嵌汇编代码自动提升

赖远明[1,2] 王 喆[1,2] 武成岗[1,2] 于 丁[3] 侯承轩[1,2]

[1]（处理器芯片全国重点实验室（中国科学院计算技术研究所） 北京 100190）

[2]（中国科学院大学 北京 100049）

[3]（北京地平线机器人技术研发有限公司 北京 100089）

（laiyuanming@ict.ac.cn）

LASM: Automatically Lift x86 Inline Assembly for Whole Program Analysis

Lai Yuanming[1,2], Wang Zhe[1,2], Wu Chenggang[1,2], Yu Ding[3], and Hou Chengxuan[1,2]

[1]（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）

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

[3]（*Beijing Horizon Robotics Technology Research and Development Co.*, *Ltd*, *Beijing* 100089）

Abstract Integrating C/C++ with assembly language is a common practice in software development, particularly in operating system-level software. However, programming with inline assembly can be more error-prone due to its complex architecture-related details and the lack of compiler checks. Modern software engineering employs static analysis to detect bugs and security vulnerabilities. Nevertheless, analyzing inline assembly poses challenges beyond the capabilities of LLVM IR-based static analysis tools. If the effects of inline assembly on data flow and control flow are not addressed effectively, the result can be a rise in false positives and a decline in precision. We present LASM, an innovative tool that can automatically lift x86 inline assembly code into LLVM IR and integrate it with the IR generated from compiling the C/C++ code. LASM enables downstream static analysis tools to perform whole program analysis on applications containing inline assembly code fragments, thereby reducing false positives and improving precision. It emulates hardware and the semantics of assembly instructions using LLVM IR. By providing optional optimizations that simplify the semantics of atomic instructions and enhance pointer arithmetic semantics, LASM supports static analysis tools more effectively. By completely and accurately emulating assembly instructions, the LLVM IR lifted by LASM can be compiled into an executable program. The strength of LASM lies in its ease of integration; LLVM IR-based static analysis tools can analyze inline assembly code by incorporating it seamlessly into their existing workflows without requiring additional modifications or adaptations to their implementations. LASM facilitates the reuse of many pre-existing tools in the ecosystem. Our experimental results indicate that LASM effectively supports SVF-based static analysis on the Linux kernel. This analysis systematically categorizes global variables based on whether their access occurs during the initialization or post-initialization phase. The results show an impressive 62.03% reduction in misclassifications and an 8.06 percentage point increase in precision. LASM helps to identify 14.80% more protectable variables without introducing new misclassifications. These results highlight LASM's potential to enhance the effectiveness of static analysis in the presence of inline assembly code.

Key words inline assembly; mixed C/C++ and assembly programming; assembly code lifting; semantics merging; static analysis; whole program analysis; LLVM IR

摘 要 C/C++和汇编语言混合编程在软件开发中被广泛使用。汇编语言的复杂性让混合编程容易出错。静态程序分析工具能减少软件漏洞,提高其鲁棒性。但基于 LLVM IR 开发的静态程序分析工具无法分析程序中的内嵌汇编代码。内嵌汇编代码影响程序中的数据流和控制流。静态分析工具忽略汇编代码导

收稿日期: 2024-11-11; **修回日期**: 2026-03-04

致程序中部分数据流未被纳入到分析中，使得误报率增加，准确率降低。设计并实现了LASM系统，该系统能将C/C++代码中的x86内嵌汇编代码自动提升成LLVM IR，并与编译C/C++得到的中间表示（IR）相融合，让下游的静态程序分析工具能对程序的所有代码进行全程序分析，降低误报率，提高准确率。LASM通过模拟硬件执行环境并使用LLVM IR的指令模拟汇编指令的语义，实现了汇编代码到LLVM IR的提升，并针对静态分析提供了原子指令语义简化和指针运算指令语义提升的优化。LASM能正确完整地模拟汇编指令的语义，提升汇编代码得到的LLVM IR代码在编译链接成可执行文件后可以正确运行。同时，LASM能和下游基于LLVM IR的静态程序分析工具无缝衔接，不需要任何适配即可增强工具的分析效果，达到复用和增强大量已有分析工具的目标。实验表明，LASM能支持针对Linux kernel的静态程序分析。在基于SVF分析Linux kernel对其全局变量的访问，并基于访问发生在初始化阶段还是初始化后阶段对变量进行分类的实验中，LASM的支持使分析精度提高了8.06个百分点，错误分类的变量数量下降了62.03%，在没有引入新分类错误的情况下多找到了14.80%的可保护变量。

关键词 内嵌汇编；C/C++与汇编混合编程；汇编代码提升；语义融合；静态分析；全程序分析；LLVM中间表示

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202440881 **CSTR:** 32373.14.issn1000-1239.202440881

C/C++等高层级程序设计语言为软件开发提供了更高层级的抽象，让开发者从内存布局、硬件特性等繁杂的细节中解脱出来，把精力聚焦在实现软件的功能逻辑上，提高了软件开发效率。汇编语言则赋予了开发者直接操作硬件，精确选择指令和精细排布指令及数据的能力，有助于实现更高效的代码和操作特权硬件。C/C++语言和汇编语言在软件开发上各有优势。因此，这2种语言混合编程被广泛应用于操作系统、嵌入式软件和高性能科学计算库等底层系统软件的开发上。例如，Linux内核中存在大量汇编代码，大约占内核总代码量的2%[1]。在大型科学计算库OpenBLAS中，汇编代码的占比接近20%[2]。在使用defconfig配置编译的Linux kernel 5.10.112中，一共有48 694个函数，其中多达18 327个函数含有内嵌汇编代码，占总函数数量的37.64%。使用C/C++实现的Debian的包中，大约有11%直接或间接依赖于内嵌汇编，在Github上排名最靠前的那些C语言工程里，有28.1%包含内嵌汇编代码[3-4]。

开发者使用汇编语言编程时，由于缺失了编译器的大量检查，非常容易编写出有漏洞的代码。即使是最资深的Linux内核开发者，在编写内嵌汇编代码时也容易出错，并且这类错误难以被审查和人工定位[5-7]。静态程序分析技术能帮助开发者发现软件中潜在的漏洞和安全缺陷。这项技术在发现和定位程序漏洞上的应用已经有很长的历史。早在1979年，能检测类型转换错误、未使用变量和未初始化变量的lint就与Unix7一起被发布。NASA要求所有软件都必须经过静态分析工具的分析后才能投入使用[8-9]。软件开发行业的实践证明了该技术的应用能降低软件研发成本，减少软件安全缺陷，提高软件安全性和健壮性[10]。

LLVM IR是LLVM编译框架的中间表示，它是SSA格式的，具有语言无关的类型系统，可提供程序代码的高层级信息，能很好地支持各种复杂精细的代码分析和变换[11]。目前已涌现出大量基于LLVM IR的静态程序分析工具。SVF[12]是其中最受欢迎、性能最好的开源工具之一[13]。但这些工具都只能分析程序中使用高级语言编写的那部分代码，无法分析程序中的汇编代码。在分析混合编程实现的软件时，要么人工处理汇编代码，对其进行建模，从而辅助工具进行分析，要么不分析汇编代码。例如，在分析Linux kernel时，CRIX[14]和LRSan[15]需要对汇编进行手动处理，IPPO[16]，HERO[17]则没有处理和分析汇编代码。

在混合编程的场景中，汇编代码被嵌入C/C++代码中，或被存储在单独的汇编文件中。Clang将软件源代码编译成LLVM IR时，不将汇编文件作为它的输入，并将嵌于C/C++语言代码内部的内嵌汇编代码视作黑盒，把内嵌汇编指令作为字符串嵌入中间表示（IR）指令中，不做额外处理。这导致以LLVM IR作为输入的静态程序分析工具普遍忽略程序IR中的内嵌汇编，跳过对它的分析。这种处理方式有2个方面的负面影响：一是汇编中的漏洞或安全缺陷无法被静态分析工具发现；二是忽略了汇编对程序数据流和控制流的影响，导致分析高级语言代码的准确性下降，产生误报率更高、准确率更低的分析结果。而误报率高和准确率低是静态分析在实际应用中的

最大障碍之一[8,18]。过往的研究表明，误报率超过30%的静态分析工具就会被弃用[19]。有的静态分析工具[20]选择牺牲一定的可靠性(soundness)来提高精度(precision)，避免过高的误报率。

为解决上述问题，让基于LLVM IR的静态程序分析工具能同时分析程序中的C/C++代码和内嵌汇编代码，实现全程序分析(whole program analysis)的目标，本文设计并实现了LASM系统。该系统能自动地将程序的x86汇编代码提升为LLVM IR表示，并将提升汇编代码得到的IR与编译高级语言代码得到的IR进行融合，生成对静态程序分析友好的LLVM IR。LASM能无缝接入下游的静态程序分析工具的工作流中。

实验表明，LASM能正确提升汇编代码，并为下游基于LLVM IR的静态程序分析工具检测程序内嵌汇编中的数据流提供有效支持。经LASM提升得到的IR被编译成可执行文件后依然能正确执行。软件中的内嵌汇编代码被提升成LLVM IR后，对静态分析工具而言不再是黑盒，能让分析工具识别和跟踪到更完整的数据流和控制流，从而得到准确率更高的分析结果。本文在SVF Test Suite[21]测试集和Linux kernel上分别验证了LASM对静态程序分析工具Saber[22]和CRIX[14]检测内嵌汇编中的数据流的支持。在使用静态分析工具DataProt分析Linux内核对其全局变量的读写情况，并根据读写操作发生的阶段对变量进行分类的实验中，LASM让错误分类的变量数量下降了62.03%，精度提高了8.06个百分点，多找到了14.80%的可保护变量，并且不会引入新的分析错误。这些实验证明LASM对内嵌汇编的提升能帮助下游基于LLVM IR的静态程序分析器获得更好的分析效果。

本文的贡献如下：

1)以支持静态程序分析为目标，提出了基于语义模拟的x86-64内嵌汇编提升方法，可将汇编代码正确地提升成LLVM IR，避免错误的代码提升干扰下游的静态分析工具。经LASM提升得到的IR再编译后可正确执行。

2)采用语义融合技术将提升汇编得到的IR和编译高级语言得到的IR融合在一起，实现了高级语义与汇编语义的融合，在保留了高级语义的同时，为下游的静态分析工具提供汇编代码的语义信息，以实现全程序分析的目标。

3)针对静态程序分析设计了原子指令语义简化和指针运算指令语义提升的优化机制，让LASM生成对静态分析更加友好的IR。

4)通过实验证明LASM能帮助静态程序分析工具识别内嵌汇编中的数据流和控制流，降低静态分析的误报率，提高分析精度。

1 相关背景

1.1 LLVM编译框架及其IR

LLVM[23]编译框架源于伊利诺伊大学的一个研究项目。它采用编译器经典的3段式设计，即前端(frontend)、中端(optimizer)和后端(backend)。高级语言代码先被前端编译成平台无关的LLVM IR中间表示，以便中端对代码实施各种编译优化。经过多年的发展，LLVM已经包含了许多项目，形成了非常完整的工具链，并在学术研究和商业领域被广泛使用。LLVM IR是SSA格式的，并且是平台无关的，非常适合用于静态程序分析。因此有大量的静态分析工具是基于LLVM IR开发的。使用任何高级语言开发的软件，只要能被编译成LLVM IR，就能使用这类工具对它进行分析。LLVM还有其他的中间代码表示，例如用于后端的MachineInstr和MCInst。LASM在提升汇编代码前，先将其转化成MCInst表示。

1.2 代码提升

LLVM将LLVM IR编译成汇编代码的过程被称为lowering。代码提升则是相反的过程，它将二进制代码或汇编代码转换成LLVM IR等高级语言表示。低级语言的代码被提升成LLVM IR后，就能使用基于LLVM IR的静态程序分析工具对它们进行安全检查和漏洞扫描。我们称通过代码提升得到的LLVM IR为Lifted-IR，将编译高级语言得到的LLVM IR称为High-IR。LASM融合Lifted-IR和High-IR后得到的IR被称为Mixed-IR。

1.3 全程序分析

静态程序分析通过在源码上静态地分析程序状态和数据流的传播，无需程序的输入和动态执行就能发现潜在的安全缺陷和漏洞，可以获得很高的分析效率和代码覆盖率，被广泛用于大型系统软件的程序缺陷检测中[13]。全程序分析是指使用静态分析技术对程序的所有代码进行统一的、一致性的分析，这可以提升静态分析的效果[24-25]。

2 研究动机

在软件开发中，为了同时获得更高层级的抽象描述能力，高性能、细粒度的指令选择能力和直接操

作硬件的能力，汇编语言和 C/C++语言混合编程被广泛使用[3-4]。然而，混合编程会引入 2 个方面的问题。一方面，使用汇编语言编程更容易出错，在程序中引入难以被人工发现和定位的漏洞或安全缺陷[26-28]。另一方面，虽然现在涌现出很多出色的基于 LLVM IR 的静态程序分析工具，它们能分析 C/C++代码，但对汇编代码无能为力，无法用来查找汇编中潜在的漏洞和安全缺陷。程序中的汇编代码还影响数据流和控制流，静态分析工具忽略了这种影响，导致其对 C/C++代码的分析精度下降。目前缺乏能对混合编程实现的软件进行全程序分析的静态程序分析工具。

解决上述问题的一种可行方案是使用不同分析工具分别对高级语言代码和汇编代码进行分析，然后合并分析结果。但研究表明分别分析多个独立模块，然后合并分析结果，会降低分析的准确率[24]。

另一种方案是将高级语言和汇编语言转成同一种表示，再对其进行分析。比如先将 C/C++代码编译成汇编代码，再使用面向汇编代码的静态程序分析工具对其进行分析。这种方法不需要合并分析结果。但 C/C++代码在编译成汇编代码后，丢失了高级语义信息，比如复杂的数据结构信息，这对静态分析非常不利。在汇编代码上进行静态分析远不如在 LLVM IR 上高效。也可以通过手动编写与汇编代码等价的 LLVM IR，实现基于 LLVM IR 的全程序分析。但这种方法非常低效，且容易出错，难以大规模使用。

本文将程序中的汇编代码自动地提升成 LLVM IR，并将其与编译 C/C++代码得到的 IR 进行融合，让下游大量基于 LLVM IR 的静态程序分析工具能对含有汇编代码的 C/C++程序进行全程序分析，如图 1 所示。待分析程序经 LASM 的汇编代码提升处理后，可直接作为下游静态程序分析工具的输入，无需对静态分析工具进行任何额外的适配，达到复用现有分析工具和提升分析精度的目的。这种方法的优势在于能将 C/C++代码的高级语义信息和汇编代码的语义信息整合在一起，让基于 LLVM IR 的静态程序分析工具能分析程序完整的数据流和控制流，从而提升分析精度。

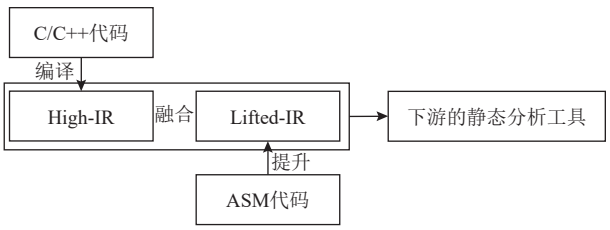


Fig. 1 Assembly code lifting assisted staic analysis

图 1 汇编代码提升辅助静态分析

3 技术路线与技术挑战

LASM 的设计目标是保留 High-IR 中的高层级语义信息，保证汇编代码提升的正确性，并生成对静态程序分析友好的 LLVM IR。核心思路是保留所有的 High-IR，将程序中的内嵌汇编代码提升为 Lifted-IR，并将 2 种 IR 融合成一个整体。如图 2 所示，混合编程的程序被 Clang 编译成 LLVM IR 后，内嵌汇编代码和 IR 是交织在一起的。首先，LASM 从 IR 中抽取出内嵌汇编代码，实现汇编代码和 High-IR 的完全分离。随后，LASM 采用模拟执行的方式，将汇编指令逐条转换成 LLVM IR 指令序列。最后 LASM 通过自动生成胶水函数，完成 Lifted-IR 和 High-IR 之间的语义融合，并对 Lifted-IR 实施面向静态分析的代码优化，得到对静态程序分析友好的 Mixed-IR。

LASM 采用的技术路线对静态程序分析更友好，但面临 3 个技术挑战：1）如何语义等价地将汇编代码提升为 LLVM IR，保证正确性；2）High-IR 和 Lifted-IR 存在语义鸿沟；3）如何让提升得到的代码对静态程序分析更友好。为解决上述挑战，LASM 使用 IR 指令等价模拟汇编指令的语义，使用胶水函数融合 High-IR 和 Lifted-IR 的语义，并针对静态分析场景做了相应的优化。

4 系统设计方案

4.1 系统架构

LASM 系统包含 6 个核心模块，如图 3 所示。汇编抽取模块将每一段内嵌汇编代码抽取到一个新的 LLVM IR 函数中。该函数被称为内嵌汇编函数，它将被编译成汇编文件。在 High-IR 中对内嵌汇编的调用则被修改成了对内嵌汇编函数的调用。经过汇编抽取后，High-IR 中就没有内嵌汇编代码了。硬件环境模拟模块将各类寄存器和栈模拟成 LLVM IR 中的 Env 结构体。汇编数据提升模块则是将汇编代码中定义的全局变量提升为 LLVM IR 中的全局结构体变量。

汇编指令提升模块先将汇编代码解析为 MCInst，并将其切分成 LiftBlock。每一个 LiftBlock 都会被提升成一个 LLVM IR 函数，并用 LLVM IR 的指令序列逐条模拟 MCInst 指令的语义。

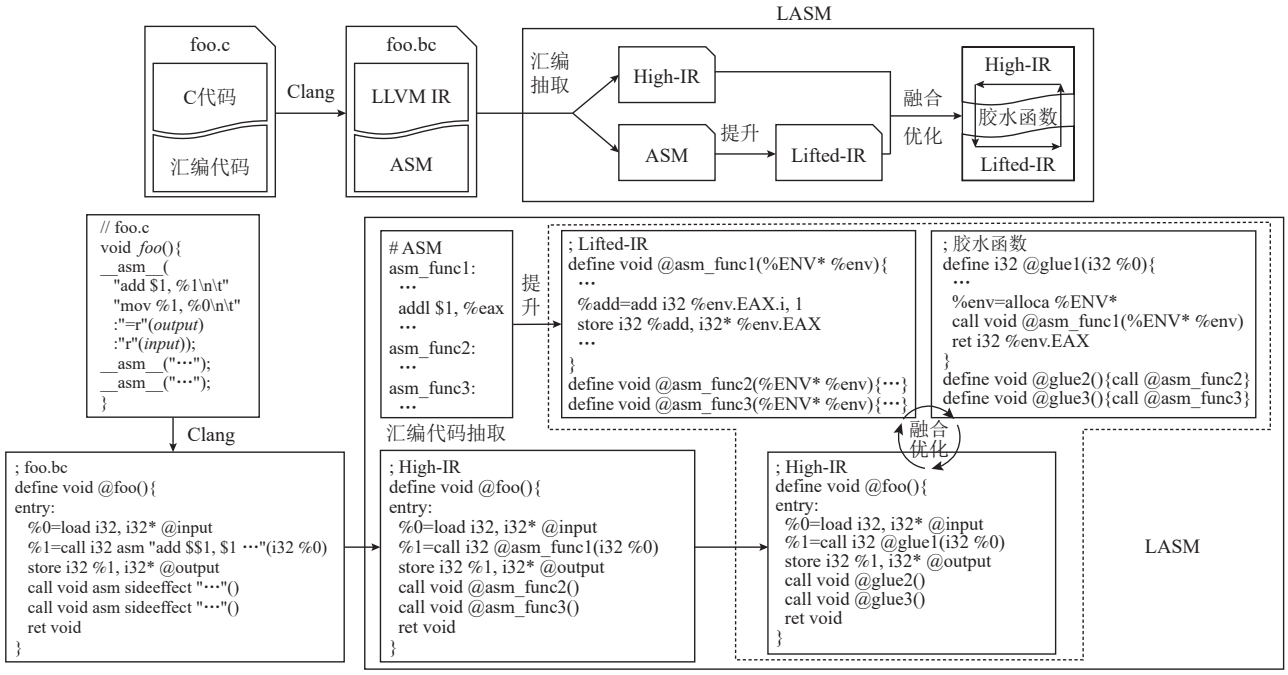


Fig. 2 Overview of LASM

图 2 LASM 系统概览

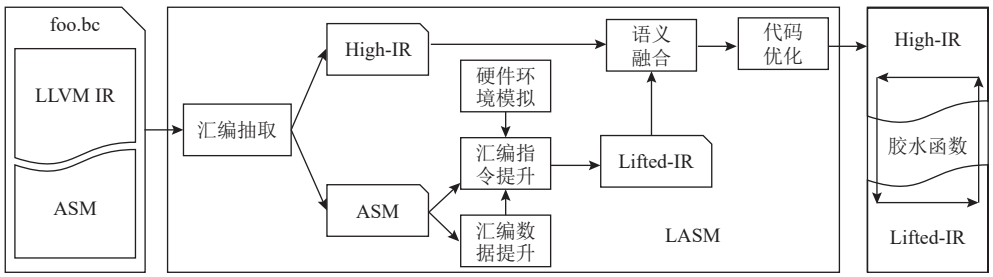


Fig. 3 The framework of LASM

图 3 LASM 系统框架

语义融合模块生成胶水函数，用于衔接 Lifted-IR 和 High-IR。Lifted-IR 和 High-IR 具有不同的函数调用约定和内存模型，在胶水函数的帮助下才能实现二者之间数据流的联通，完成跨代码边界的函数调用。

代码优化模块先使用 LLVM 的内联优化等编译优化机制对 Lifted-IR 进行优化，并针对 Lifted-IR 中的原子指令和指针运算指令进行优化，生成对静态分析器更加友好的 IR。

4.2 内嵌汇编抽取

内嵌汇编抽取是为了实现 IR 和内嵌汇编代码的分离。图 4(a)所示的 C 语言代码是含有内嵌汇编的函数 *foo*，它被 Clang 编译后得到图 4(c)所示的 IR。从图 4(c)中可以看出，LLVM IR 将内嵌汇编代码当作黑盒，通过 call 指令调用它。编译器在将 IR 编译成汇编代码时，将按照约定好的接口创建它和内嵌汇编之间的衔接指令，并将内嵌汇编中的占位符转换成指令的合法操作数，这个过程非常复杂[29]，难以对其进行建模。LASM 先将调用内嵌汇编代码的语句提取到新建函数@foo_inline0 中，如图 4(d)所示。函数@foo 对内嵌汇编的调用语句则被修改成对函数@foo_inline0 的调用，如图 4(b)所示。随后，函数@foo_inline0 及其依赖的数据定义被抽取到一个新建的文件 foo_ASMFunc.bc 中。将该文件编译成汇编文件后再对它进行汇编代码提升处理。完成内嵌汇编抽取后，IR 和汇编之间的边界变得清晰，数据交互则遵循 API 调用约定，方便 LASM 的后续处理。

4.3 硬件环境模拟

为了精确和完整地提升汇编代码的语义，LASM 在 LLVM IR 层模拟了一套硬件执行环境，即 Env。Env 是一个结构体，如图 5 所示。汇编指令操作的硬件资源包括寄存器和内存。其中内存又可以分为全

```
/* foo.c */
void foo(char* ch){
__asm__(
        "mov %1, %0"
        :"=r"(s) :"r"(ch));
}
```

(a) 含内嵌汇编的C代码

```
/* foo_High-IR.bc */
define dso_local void @foo(i8* %ch){
entry:
   %0=tail call i8* @foo_inline0(i8* %ch)
   store i8* %0, i8** @s
   ret void
}
```

(b) 被抽走内嵌汇编后的IR

```
/* foo.bc */
define dso_local void @foo(i8* %ch){
entry:
   %0=tail call i8* asm "mov $1, $0", "=r, r, ~{dirflag}, ~{fpsr}, ~{flags}"(i8* %ch)
   store i8* %0, i8** @s
   ret void
}
```

(c) 含有内嵌汇编的IR

```
/* foo_ASMFunc.bc */
define i8* @foo_inline0(i8* %0) {
   %2=call i8* asm "mov $1, $0", "=r, r, ~{dirflag}, ~{fpsr}, ~{flags}"(i8* %0)
   ret i8* %2
}
```

(d) 内嵌汇编函数

Fig. 4 Inline assembly extraction

图 4 内嵌汇编抽取

局数据区(.data 节, .rodata 节, .bss 节等)、堆和栈。LASM 模拟了所有寄存器。全局数据区中存放的是程序的全局数据, LASM 将这些全局数据提升为 LLVM IR 的结构体类型的数据。汇编代码对堆上数据的操作均始于调用 *malloc* 等动态内存分配的 API。提升汇编时, 只需要使用 LLVM IR 调用对应的 API 函数即可。LASM 使用数组来模拟栈, 并在初始化 Env 时, 将 Env.Rsp 指向它。模拟的栈被称为软件栈。

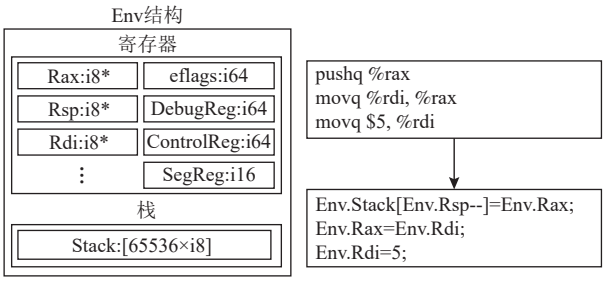


Fig. 5 Env structure and assembly instruction emulation

图 5 Env 结构体与汇编指令模拟

对物理寄存器的模拟需要选择与其位宽相等的数据类型来模拟。一个直观的方案是使用不同位宽的整型来模拟对应位宽的寄存器。比如使用 i64 类型模拟 Rax 等 64b 寄存器。但这样做会导致将一个指针类型的数据存入模拟寄存器前, 必须先将指针转换成整型。对保存在模拟寄存器中的指针进行解引用前, 必需先以整数的形式将它从 Env 中取出, 然后将其转换成目标类型的指针, 伪代码如图 6 所示。使用 inttoptr 和 ptrtoint 指令在指针和整型数据之间进行转换将影响静态程序分析工具对指针传播的跟踪, 导致不精确的指针别名分析结果。LASM 使用 LLVM IR 中的指针类型 i8*来模拟 64b 寄存器, 其他位宽的寄存器则使用对应位宽的整型模拟。如图 6 所示, 在这种设计选择下, 当指针通过虚拟寄存器进行传播和使用时, 发生的是不同指针类型之间的转换, 避免了将内存中的整型数据转换成指针类型的操作。当 64b 整数类型存入虚拟寄存器时, 也会发生整型和指针类型之间的类型转换, 但被操作数据实际上是整型, 对指针分析没有影响。

```
# %ptr的类型为Type*类型的指针
# %Env.rdi为指向Env.Rdi的指针

# 使用i64类型模拟寄存器
# 将指针类型的数据存入i64类型的模拟寄存器
%1=ptrtoint Type* %ptr to i64
store i64 %1, i64* %Env.rdi

# 将指针类型的数据从i64类型的模拟寄存器中取出
%2=load i64, i64* %Env.rdi
%3=inttoptr i64 %2 to Type*

# 使用i8*类型模拟寄存器
# 将指针类型的数据存入i8*类型模拟寄存器
%4=bitcast Type* %ptr to i8*
store i8* %4, i8** %Env.rdi

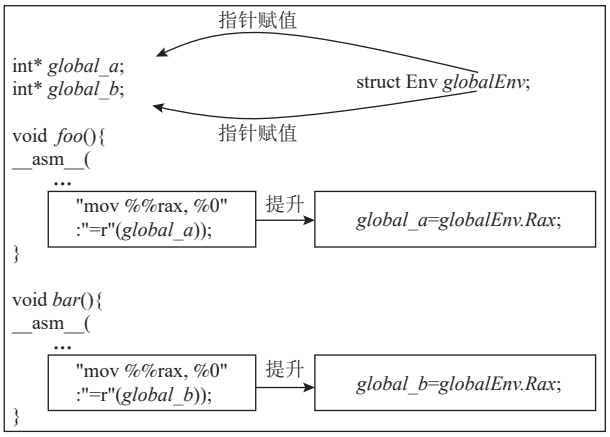
# 将指针类型的数据从模拟寄存器中取出
%5=load i8*, i8** %Env.rdi
%6=bitcast i8* %5 to Type*
```

Fig. 6 Propagate and access pointers via emulated registers

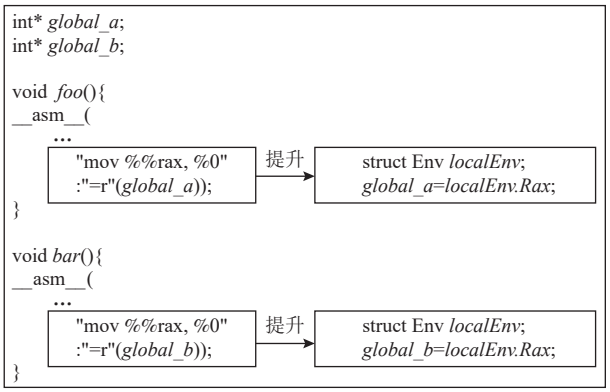
图 6 通过模拟寄存器传播和访问指针

在汇编语言中, 寄存器和栈都是全局可见的。将 Env 设置成全局数据, 能让任何 Lifted-IR 对 Env 的修改都是全局可见的, 可以非常直观地模拟汇编指令对机器状态的访问。但这样会使指针分析的精度下降。如图 7(a)所示, 程序中 2 段完全独立的内嵌汇编将 Rax 寄存器中的值分别写入了指针类型的全局数据 *global_a* 和 *global_b* 中。当这 2 段内嵌汇编被提升后, 就变成了分别将全局数据 *globalEnv.Rax* 的值写入 *global_a* 和 *global_b* 中, 为了便于理解, 图中使用 C 语言代替 LLVM IR 来描述。这将导致静态程序分析工具误认为 *global_a* 和 *global_b* 具有相同的指向集, 存在指针别名关系。

LASM 将模拟通用寄存器的 Env 设置成局部数据, 即局部 Env(local_Env)。Lifted-IR 中函数的原型则被设计成为无返回值, 且有一个局部 Env 指针参数, 即 void @lifted_func(%local_Env* %local_env)。同时, 为了支持 Linux 内核, 将段寄存器、DR 寄存器和 CR 寄存器等设置为全局数据, 即全局 Env(global_Env)。引入局部 Env 后, 每个内嵌汇编片段都使用各自私有的 local_Env, 将 local_Env 中的值写入了指针类型



(a) 使用全局Env



(b) 使用局部Env

Fig. 7 Difference between using global Env and local Env

图 7 使用局部 Env 和使用全局 Env 的区别

的变量不会导致指针传播的范围被扩大, 如图 7(b)所示。局部 Env 的设计也便于 LASM 在将来支持多线程程序。

4.4 汇编数据提升

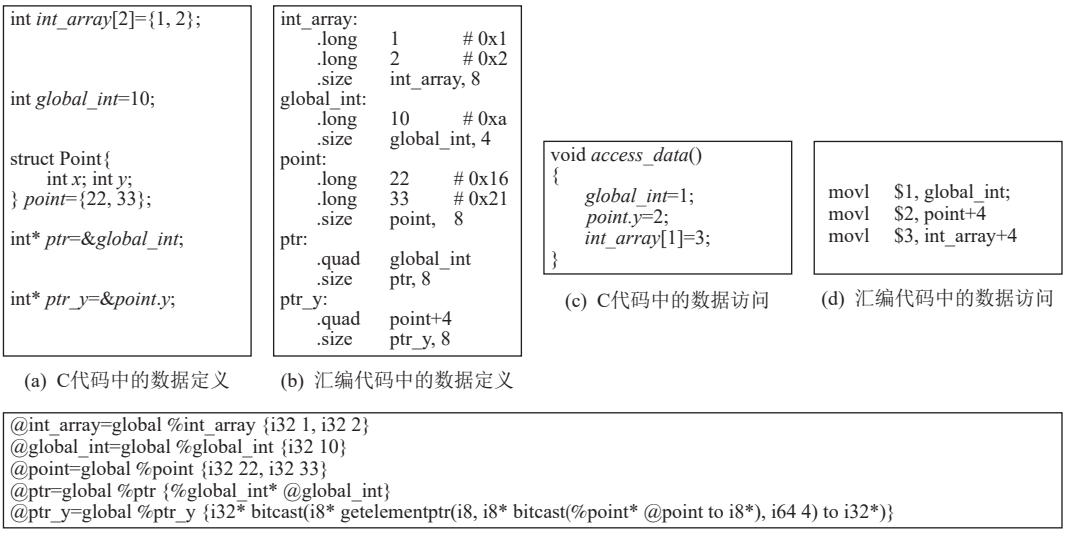
LASM 在对汇编文件进行解析时, 将数据段切分成一个个小单元。汇编使用标签绑定数据, 标签名可以看作是 C 语言中的变量名。数据在汇编文件中的排列顺序和它在内存中的布局是对应的。汇编中有 2 种常见的定义和初始化数据的方式: 一种是使用数据类型和常量; 另一种是使用数据类型和标签名, 如图 8(b)所示。后一种对应 C 代码中指针类型的数据。汇编代码通过标签访问数据, 当被访问的数据是数组的元素或者结构体的域时, 则通过标签加偏移量的方式访问, 如图 8(d)所示。把汇编进一步编译成二进制文件, 并将其数据段在 LLVM IR 上表示为全局数组是一种相对简单的数据提升方法。二进制文件数据段中的数据表示形式是单一的, 代码通过偏移或绝对地址访问数据。在 LLVM IR 上将绝对地址或偏移映射成对上述全局数组的访问就能模拟汇编访问全局数据的语义。但这种提升方法将所有全局数据视为一个整体, 对细粒度的静态程序分析不利。

为了支持汇编数据的定义和访问方式, 同时更好地支持静态程序分析工具, LASM 将汇编中的数据分块提升为结构体类型。汇编中的标签是其数据定义的开始, 因此, LASM 以标签为分割点, 将数据区中定义的数据切分成数据块, 每一个数据块都将被提升成 LLVM IR 中的结构体类型。汇编中使用常量初始化的数据被提升成 IR 后, 也使用对应的常量值进行初始化。使用标签名初始化的数据, 提升后则引用该标签对应的 IR 结构体数据进行初始化。图 8(b)所示的数据会被分割成 5 个数据块。标签 int_array 和标签 global_int 之间的数据构成其中的一个数据块, 该数据块被提升后得到如图 8(e)所示的全局结构体@int_array。该结构体有 2 个域, 分别被初始化为 1 和 2。汇编中的数据 ptr 则被提升成全局结构体@ptr, 并引用了@global_int 进行初始化。

4.5 汇编指令提升

汇编指令的操作数有 3 类: 立即数、寄存器、和内存。LASM 在模拟了硬件执行环境和提升了全局数据后, 就能使用 LLVM IR 模拟汇编指令的语义。汇编指令对硬件资源的操作被模拟成对 Env 的操作, 对全局数据的操作被模拟成操作提升得到的结构体数据。汇编代码的程序状态可以建模为

$$S=(R,flags,mem),$$



(a) C代码中的数据定义 (b) 汇编代码中的数据定义 (c) C代码中的数据访问 (d) 汇编代码中的数据访问

(e) 提升后的数据表示

Fig. 8 Lifting global data in assembly code

图 8 提升汇编代码中的全局数据

其中 R 为寄存器状态，为了方便描述，把标志位寄存器单独建模为 *flags*，*mem* 为内存状态。提升为 LLVM IR 后，程序状态可以建模为

$$T(S)=(E.R,E.flags,mem),$$

其中 E 为 Env 类型的对象，用于模拟硬件环境状态，$E.R$ 用于模拟寄存器状态 R，$E.flags$ 用于模拟标志位寄存器状态 *flags*。在 2 个程序状态模型中，*mem* 均表示堆、栈和全局数据区的内存。LASM 在 Lifted-IR 上使用软件栈 *E.Stack* 模拟汇编层的硬件栈。全局数据区的数据则被提升为 Lifted-IR 上的全局变量。显然状态 $T(S)$ 是汇编程序状态 S 在 LLVM IR 上的映射。若 1 条汇编指令执行后，程序状态从 S 转移至 S'，LASM 要做的则是使用 LLVM IR 指令实现程序状态 $T(S)$ 到 $T(S')$ 的转移。

汇编中整型的二元算术运算可以抽象为取源操作数和目的操作数、运算求值、运算结果存入目的操作数和更新标志位等步骤，使用通用的大步语义框架将其描述为

$$\frac{\begin{array}{c}val_dest=read(S,dest)\\ val_src=read(S,src)\\ result=OP(val_dest,val_src)\\ flags'=update_flags(OP,val_dest,val_src,result)\end{array}}{\langle OP\ dest,src,S\rangle \Downarrow S'[dest\mapsto result,flags\mapsto flags']}。$$

在 Lifted-IR 上被模拟为

$$\frac{\begin{array}{c}val_dest=read(T(S),dest)\\ val_src=read(T(S),src)\\ result=OP(val_dest,val_src)\\ flags'=update_flags(OP,val_dest,val_src,result)\end{array}}{\langle OP\ dest,src,T(S)\rangle \Downarrow T(S')[dest\mapsto result,E.flags\mapsto flags']}。$$

其中函数 *read* 表示读取操作数的值。操作符 *OP* 表示 add(加)、sub(减)、mul(乘)、div(除)等二元算术运算操作。不同的运算可能需要更新不同的标志位，使用函数 *update_flags* 进行描述，该函数的行为与指令手册保持一致。在汇编上，读取立即数、寄存器和内存操作数的语义可分别描述为

$$read(S,imm)=imm,$$

$$read(S,reg)=R(reg),$$

$$read(S,mem,addr)=mem_load(S,addr,size)。$$

在 Lifted-IR 上被模拟为

$$read(T(S),imm)=imm,$$

$$read(T(S),reg)=E.R(reg),$$

$$read(T(S),mem,addr)=mem_load(T(S),addr,size)。$$

操作符 *OP* 在汇编和 Lifted-IR 上的语义是完全等价的。以 64 位算术运算为例，加、减、乘、除 4 种二元算术运算操作的语义可分别描述为

$$\frac{dest=v1\ \ src=v2}{\langle \text{add}\ dest,src,\delta\rangle \Downarrow \delta'[dest\mapsto (v1+v2)\bmod 2^{64}]},$$

$$\frac{dest=v1\ \ src=v2}{\langle \text{sub}\ dest,src,\delta\rangle \Downarrow \delta'[dest\mapsto (v1-v2)\bmod 2^{64}]},$$

$$\frac{dest=v1\ \ src=v2}{\langle \text{mul}\ dest,src,\delta\rangle \Downarrow \delta'[dest\mapsto (v1\times v2)\bmod 2^{64}]},$$

$$\frac{dest=v1\ \ src=v2\ \ src\neq 0}{\langle \text{div}\ dest,src,\delta\rangle \Downarrow \delta'[dest\mapsto (v1/v2)\bmod 2^{64}]}。$$

当运算发生在汇编代码中时，δ 和 δ' 分别表示 S 和 S'，发生在 Lifted-IR 上时则分别表示 $T(S)$ 和 $T(S')$。

函数 *update_flags* 根据 *OP* 操作的运算结果以及

该操作更新标志位寄存器的规则更新 $S[flags]$ 和 $T(S)[E.flags]$。在汇编和 Lifted-IR 上，相同的操作将使用完全一样的标志位置位规则。不同之处在于更新的对象不同，汇编指令更新的是标志位寄存器，而 Lifted-IR 更新的是 E 中模拟的标志位寄存器。显然，当汇编和 Lifted-IR 分别在 S 和 $T(S)$ 状态下对相同的操作数执行相同的算术运算 OP 后，将得到相同的运算结果和相同的标志位置位结果，即分别得到新的状态 S'和 $T(S')$。对于其他的汇编算术运算指令，LASM 也能通过类似的方法进行等价模拟。

数据传送的操作可以分为 2 类：一类是向寄存器传送数据；另一类是向内存传送数据。汇编代码中内存访问的 SIB 寻址语义可以描述为

$$\frac{addr = R(base) + R(index) \times scale + disp}{S \xrightarrow{calc_addr} S[temp_addr \mapsto addr]}。$$

在 Lifted-IR 上则被等价模拟为

$$\frac{addr = E.R(base) + E.R(index) \times scale + disp}{T(S) \xrightarrow{calc_addr} T(S)[temp_addr \mapsto addr]}。$$

汇编上向寄存器和内存中传送数据的语义可以描述为

$$\frac{v = read(S, src)\ src \text{ is } imm, reg, \text{or } mem}{\langle \text{mov } src, R(reg), S\rangle \Downarrow S'[R(reg) \mapsto v]},$$

$$\frac{v = read(S, src)\ addr \text{ is valid}, src \text{ is } imm \text{ or } reg}{\langle \text{mov } src, addr, S\rangle \Downarrow S'[mem(addr) \mapsto v]}。$$

在 Lifted-IR 上则被等价模拟为

$$\frac{v = read(T(S), src)\ src \text{ is } imm, reg, \text{or } mem}{\langle \text{store } src, E.R(reg), T(S)\rangle \Downarrow T(S')[E.R(reg) \mapsto v]},$$

$$\frac{v = read(T(S), src)\ addr \text{ is valid}, src \text{ is } imm \text{ or } reg}{\langle \text{store } src, addr, T(S)\rangle \Downarrow T(S')[mem(addr) \mapsto v]}。$$

可以看到，通过 LLVM IR 算术运算指令和访存指令的组合，可以实现对汇编的内存寻址和数据传送指令的等价模拟。

以图 9 所示的代码为例，汇编代码的第 3 行将立即数 6 写入 Edi 寄存器，被提升成 Lifted-IR 的第 2 行，将立即数 6 写入 env.edi 中。同时，汇编指令的隐式操作也会被 LASM 模拟，图 9 中第 3 行 Lifted-IR 模拟汇编指令 callq 将返回地址压到栈上的隐式操作。

汇编代码		Lifted-IR伪代码	
1	asm_func:	define void @asm_func(%ENV* %env)	1
2	…	%env.edi=6	2
3	movl $6, %edi	%env.Stack[%env.Rsp--]=@asm_func_split_1	3
4	callq bar	call void @bar(%ENV* %env)	4
		ret void	5
		define void @asm_func_split_1(%ENV* %env)	6
5	testb $1, flag	if flag==1	7
6	je .LBB1_2	call void @asm_func_LBB1_2(%ENV* %env)	8
		else	9
		call void @asm_func_bb1(%ENV* %env)	10
		ret void	11
7	# %bb.1:	define void @asm_func_bb1(%ENV* %env)	12
8	movl $1, a	call void @asm_func_LBB1_3(%ENV* %env)	13
9	jmp .LBB1_3	ret void	14
10	.LBB1_2:	define void @asm_func_LBB1_2(%ENV* %env)	15
11	movl $2, a	call void @asm_func_LBB1_3(%ENV* %env)	16
		ret void	17
12	.LBB1_3:	define void @asm_func_LBB1_3(%ENV* %env)	18
13	popq %rbp	%ret=%env.Stack[%env.Rsp++]	19
14	retq	call void %ret(%ENV* %env)	20
		ret void	21

Fig. 9 Assembly code segmentation and pseudocode of instruction emulation

图 9 汇编代码切分和指令模拟伪代码

LLVM IR 以函数的形式组织指令。LASM 将汇编代码切分成 LiftBlock，并将每个 LiftBlock 都提升成 LLVM IR 中的函数。跳转指令、函数调用指令和函数返回指令是一个 LiftBlock 的结束指令，标志着当前 LiftBlock 的结束。这些指令的下一条指令则是另一个 LiftBlock 的起始指令。标签是跳转指令和函数调用指令的目标，它标志着前一个 LiftBlock 的结束和新 LiftBlock 的开始。标签后的第 1 条指令是新 LiftBlock 的起始指令。如图 9 所示，左边的 14 行汇编代码被切分成 5 个 LiftBlock，以标签 asm_func 开头的第 1 个 LiftBlock 被提升成 LLVM IR 函数 void @asm_func(%ENV* %env)。将汇编代码切分成 LiftBlock 之后，LASM 使用 LLVM IR 的 call 指令模拟汇编中跳转指令、函数调用指令和函数返回指令的语义，实现控制流转移。

汇编指令 jmp 和 jcc 的语义可描述为

$$\frac{addr \text{ is valid}}{\langle \text{jmp } addr, S\rangle \Downarrow S'[R(pc) \mapsto addr]},$$

$$\frac{cond(R[flags]) = \text{true } addr \text{ is valid}}{\langle \text{jcc } addr, S\rangle \Downarrow S'[R(pc) \mapsto addr]},$$

$$\frac{cond(R[flags]) = \text{false } addr_{\text{next}} \text{ is valid}}{\langle \text{jcc } addr, S\rangle \Downarrow S'[R(pc) \mapsto addr_{\text{next}}]}。$$

跳转指令的核心语义是修改 *pc* 的值。LASM 使用 LLVM IR 的 call 指令等价模拟 jmp 指令，使用 br 和 call 指令的组合等价模拟 jcc 指令：

$$\frac{addr \text{ is valid}}{\langle \text{call } addr, T(S)\rangle \Downarrow T(S')[E.R(pc) \mapsto addr]},$$

$$\frac{cond(E.R[flags]) = \text{true } \%\text{t: call } addr}{\langle \text{br i1 } cond, \%\text{t}, \%\text{f}; \text{call } addr, T(S)\rangle \Downarrow T(S')[E.R(pc) \mapsto addr]},$$

$$\frac{cond(E.R[flags]) = \text{false } \%\text{f: call } addr_{\text{next}}}{\langle \text{br i1 } cond, \%\text{t}, \%\text{f}; \text{call } addr_{\text{next}}, T(S)\rangle \Downarrow T(S')[E.R(pc) \mapsto addr_{\text{next}}]}。$$

汇编指令 jmp 和 jcc 的目标都是 label，它是 LiftBlock 的起始标志。这 2 条指令还会让 LASM 将其和其下一条指令切分到 2 个不同的 LiftBlock 中。所以汇编指令 jcc 的目标地址 *addr* 和其下一条指令的地址 $addr_{\text{next}}$ 在 Lifted-IR 上都对应着一个函数的起始地址。LASM 使用 LLVM IR 的 call 指令将控制流转移到预期的目标地址上。汇编指令 jcc 本身的语义就包含了对标志位的判断。但 LLVM IR 的 call 指令不具备这一语义，LASM 需要使用 br 指令来模拟 jcc 指令对标志位的判断。LLVM IR 的 call 指令对 *E.R*(*pc*) 的修改与汇编跳转指令对 *R*(*pc*) 的修改是等价的，同时 LASM 维持 *E.Stack* 不变，因此，*S* 和 *T*(*S*) 上发生的状态转移是等价的。

汇编指令 call 和 ret 的语义可描述为

$$\frac{\begin{array}{c} tar_addr \text{ is valid and } L \text{ is } len(\text{call inst}) \\ ret_addr = R(pc) + L \\ R' = R\left[rsp \mapsto R(rsp) - 8,\ pc \mapsto tar_addr\right] \\ Mem' = Mem\left[R(rsp) - 8 \mapsto ret_addr\right] \end{array}}{\langle \text{call } tar_addr, S\rangle \Downarrow S'[R \mapsto R', Mem \mapsto Mem']},$$

$$\frac{\begin{array}{c} ret_addr = Mem(R(rsp)) \\ R' = R[rsp \mapsto R(rsp) + 8, pc \mapsto ret_addr] \end{array}}{\langle \text{ret}, S\rangle \Downarrow S'[R \mapsto R']},$$

其核心语义是修改 *pc* 和函数调用栈的栈桢。LASM 将其模拟为

$$\frac{\begin{array}{c} tar_addr \text{ is the address of the first liftblock in target func} \\ ret_addr \text{ is the address of succ_liftblock} \\ E.R' = E.R\left[rsp \mapsto E.R(rsp) - 8, pc \mapsto tar_addr\right] \\ Mem' = Mem\left[E.R(rsp) - 8 \mapsto ret_addr\right] \end{array}}{\langle \text{spush } ret_addr; \text{call } tar_addr, T(S)\rangle \Downarrow T(S')[E.R \mapsto E.R', Mem \mapsto Mem']},$$

$$\frac{\begin{array}{c} ret_addr = Mem(E.R(rsp)) \\ E.R' = E.R\left[rsp \mapsto E.R(rsp) + 8, pc \mapsto ret_addr\right] \end{array}}{\langle \text{spop } ret_addr; \text{call } ret_addr, T(S)\rangle \Downarrow T(S')[E.R \mapsto E.R']},$$

其中 spush 操作表示向软件栈 *E.Stack* 中压入数据，在模拟汇编的 call 指令时，通过该操作完成返回地址入栈的模拟。spop 为从软件栈中弹出数据，在模拟汇编的 ret 指令时需要通过该操作取出返回地址。通过模拟返回地址的入栈和出栈操作，保证模拟执行后 *T*(*S*′) 中的 *E.Stack* 与 *S*′中硬件栈的状态是等价的。LLVM IR 的 call 指令则等价模拟了控制流转移的语义。

为了支持 Linux 内核，LASM 还对 3 类特殊的汇编指令做了处理，如表 1 所示。这 3 类汇编指令没有对应的 LLVM IR 指令。第 1 类是不修改内存和寄存器的汇编指令，它们不会影响静态程序分析。比如特权指令 INVLPG 只是无效内存页对应的 TLB 项。第 2 类是写特定寄存器的汇编指令，LASM 模拟其语义，操作相应的模拟寄存器，比如 CPUID 指令执行后将结果写入 Eax, Ebx, Ecx, Edx 这 4 个寄存器中，LASM 则将外部数据写入它们对应的模拟寄存器中。第 3 类是切换执行上下文的指令，LASM 将重新初始化 Env，并切断数据流的传播。

Table 1 Handling Special Instructions

表 1 处理特殊指令

类型	类型描述	指令示例	处理方式
1	不影响程序分析	INVLPG：无效 tlb entry	不模拟，仅标识该指令内容
2	写特定寄存器	CPUID：赋值 Eax，Ebx，Ecx，Edx	写模拟寄存器
3	切换上下文环境	VMRESUME：切换虚拟机状态	重新初始化 Env，切断数据流传播

4.6 语义融合

Lifted-IR 和 High-IR 表达的是不同层级的语义。High-IR 中的函数调用遵循 API 调用约定，有形参和实参的概念，参数和返回值通过符号传递。而汇编语言中的函数调用遵循的是 ABI 约定，参数和返回值通过寄存器、栈来传递。LASM 通过模拟汇编指令的语义来完成汇编代码到 LLVM IR 的提升。因此 High-IR 和 Lifted-IR 并没有因为都使用了 LLVM IR 的指令集而表达同一层级的语义，如图 10 所示。

LASM 使用胶水函数衔接 Lifted-IR 和 High-IR 之间跨代码边界的函数调用，如图 11 所示。当 High-IR 调用 Lifted-IR 的函数时，需要 High_To_Low 类型胶水函数的辅助，当 Lifted-IR 调用 High-IR 中的函数时，则需要 Low_To_High 类型胶水函数的辅助，以完成参数和返回值的传递。LASM 需要为每一个被跨代码边界调用的函数自动生成函数名唯一的胶水函数。生成胶水函数依赖于目标函数的函数原型。LASM 通过扫描 High-IR 发现和搜集函数原型的定义。如果程序中的函数被实现在汇编语言中并被高级语言调用了，那么在 High-IR 中有该函数的原型声明。反过来，一个函数被实现在高级语言中，被汇编代码调用了，High-IR 中有该函数的实现。如果汇编代码调用了库函数，且该库函数未被高级语言调用，那么 High-IR 中没有该函数的原型声明，需要用户提供该库函数的原型。LASM 内置了部分库函数的原型定义。

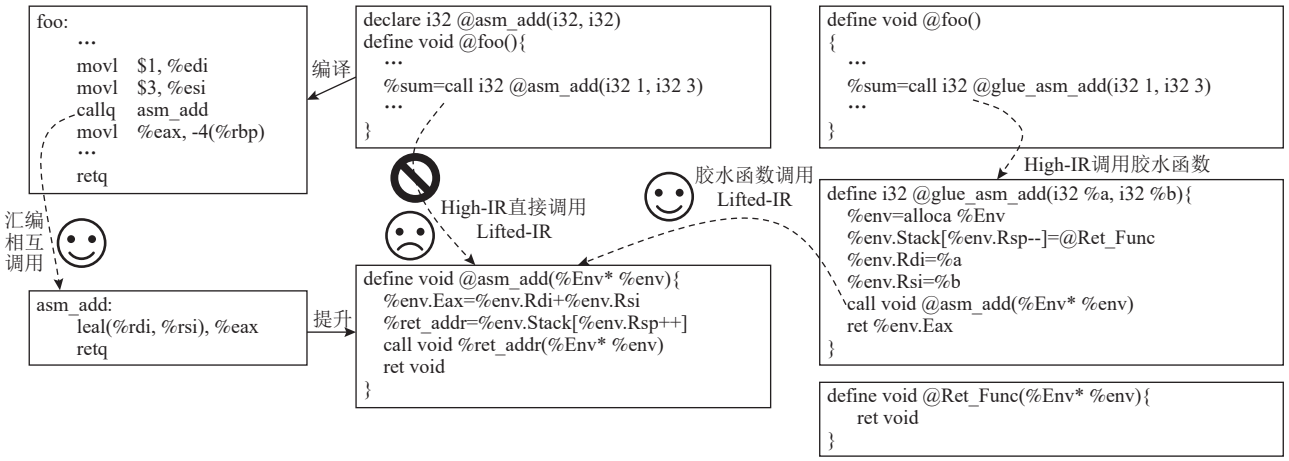


Fig. 10 Function call across code boundaries

图 10 跨代码边界的函数调用

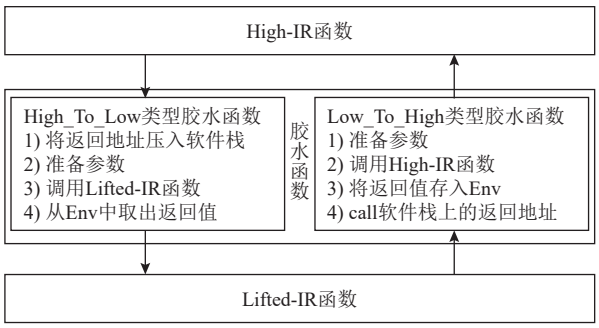


Fig. 11 Functionality of glue functions

图 11 胶水函数功能

在获取了函数原型后，LASM 根据 System-V 的 ABI 调用约定和 API 调用约定，为调用目标函数设置实参。在生成 High_To_Low 胶水函数时，首先生成传递参数的代码，将胶水函数的形参的值加载到 Env 结构体中。假设参数个数为 N，当 N<6 时，将第 1~N 个参数按 ABI 约定的顺序加载到 Env 模拟的参数寄存器中。当 N>6 时，则将第 1~6 个参数分别加载到 Env 中的 RDI, RSI, RDX, RCX, R8, R9 中，第 7~N 个参数加载到软件栈上。胶水函数在软件栈上压入模拟的返回地址后，即可调用 Lifted-IR 上的目标函数 F，调用结束后将返回值从 Env.Rax 中取出，在恢复软件栈栈顶指针后，生成返回到 High-IR 中的指令。

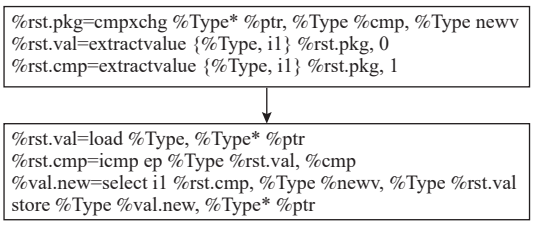
在生成 Low_To_High 胶水函数时，也进行类似的操作。不同之处在于，参数传递过程是从 Env 结构体中取出参数值，并构造对应的实参变量存储参数值，然后调用 High-IR 中的目标函数 F。当函数调用结束后，将返回值存入 Env.Rax 中。最后模拟汇编指令 ret 的语义，从软件栈上取出返回地址并调用，以返回到 Lifted-IR 中。

4.7 面向静态分析的优化

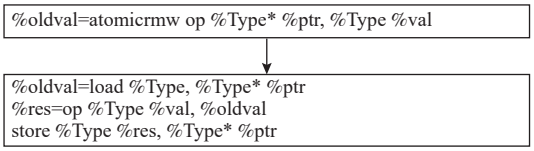
使用 LLVM IR 模拟汇编指令语义的代码提升方法能准确完整地表达汇编指令的语义，保证了正确

性，但表达的是低级语义，比如通过指针运算实现数组元素的访问。为了更好地支持下游的静态程序分析工具，LASM 提供了面向静态程序分析的优化，包括原子指令语义简化和指针运算指令语义提升。这 2 项优化能让程序中的数据流关系更直观地呈现给静态程序分析工具。优化机制会打破汇编语义的完整性，是可选的。

原子指令语义简化通过使用非原子指令替代原子指令来实现。cmpxchg 可以实现对内存的原子性写入。LASM 使用非原子的 load, cmp, store 指令序列来模拟 cmpxchg 的功能，如图 12 所示。这种模拟在写内存这一操作上和 cmpxchg 是一致的，但不具备原子性，对于 atomicrmw 指令也做了类似的处理。原子操作的指令序列具有图 12 所示的指令序列模式，LASM 采用指令模版匹配的方法识别定位 IR 上的原子操作序列，并将其替换成非原子操作指令序列。在对数据竞争不敏感的静态程序分析场景下，这项优化能让分析工具获得更准确的值流分析结果。



(a) 替换cmpxchg指令



(b) 替换atomicrmw指令

Fig. 12 Atomic instructions optimization

图 12 原子指令优化

指针运算指令语义提升通过使用 LLVM IR 的 GetElementPtr 指令(简称 GEP 指令)替换指针运算指令序列来实现。LLVM IR 使用 GEP 指令表达高级语言中访问数组元素和结构体域的语义。但在汇编代码中，往往是使用数组或结构体的基址和偏移量相加的方式来计算访存地址。Lifted-IR 则使用 IR 指令模拟了汇编的这种指针运算。指针运算对静态程序分析工具不友好。LASM 使用启发式的方法，将 base 加 offset 的指针运算替换成 GEP 指令。对于 Lifted-IR 中的 add 指令，我们先找到其非立即数操作数的类型来源，然后生成 GEP 指令来替换该指令。如果该操作数源于结构体或者是数组类型，我们通过 add 指令的立即数操作数计算出指针算术运算是要访问结构体的哪个域，或者是数组的哪个元素，并创建对应的 GEP 指令。生成 GEP 指令后，add 指令将会被替换。非立即数操作数通常需要经过数层类型转换。它可能是变量或者表达式。如果是变量，则获取其类型；如果是表达式，在获取其类型后还需递归地获取其操作数的类型。

如图 13 所示，使用 GEP 替代指针运算后，指令变得更加简洁，没有了指针和整型之间的类型转换，可为静态程序分析工具的指针分析提供数组元素访问的高级语义信息。

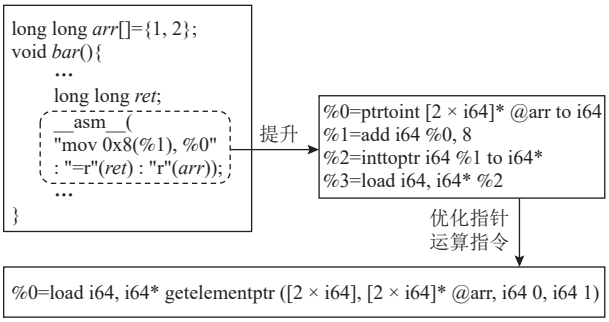


Fig. 13 Pointer arithmetic instructions optimization

图 13 指针运算指令优化

5 系统实现

本文基于 LLVM 10.0.1 实现了 LASM 系统。该系统包含约 3.29 万行 C++代码，可提升 x86-64 内嵌汇编和汇编文件，现已支持 356 种汇编指令。为了减小开发量，我们使用 C++语言实现了对汇编指令的语义模拟。模拟汇编指令语义的函数被称为语义模拟函数。然后使用 Clang 将指令的语义模拟函数编译成 LLVM IR。在提升汇编指令时，调用语义模拟函数的 IR 即可。所有语义模拟函数都被赋予了 alwaysinline 的属性。当一个汇编函数的所有指令都被提升之后，LASM 对提升后的代码施加 LLVM 的内联优化和 O2 优化选项的其他编译优化机制，将各条指令对应的语义模拟函数内联到一起，起到简化控制流的效果。同时，编译优化也消除了相当一部分模拟 eflags 寄存器更新的指令。

单个内嵌汇编片段含有的指令数一般都不多，经过内联优化后，提升内嵌汇编得到的 Lifted-IR 全都会被内联到 High-IR 中。同时，内嵌汇编中很少有控制流转移指令。因此，内嵌汇编代码被提升后，通常是 High-IR 调用胶水函数，胶水函数调用 Lifted-IR

的函数, 不会有更深层的函数调用。如图 10 所示, 在胶水函数中 LASM 会先将函数@Ret_Func 的地址压入软件栈中, 而 Lifted-IR 后续从软件栈中取出该函数地址并调用。间接函数调用让控制流变得复杂, 在实现时, LASM 引入了一个影子栈, 专门用于存储函数调用的返回地址, 从 Lifted-IR 返回时变成从影子栈中取返回地址。在提升内嵌汇编的场景中, 经过内联优化后, 对影子栈的操作都在同一函数内, 数据流非常清晰。LASM 把从影子栈上取返回地址再进行间接调用的操作改为直接调用函数@Ret_Func, 简化了控制流。

6 实验评估

LASM 能够准确提升汇编代码的语义, 并支持下游的静态程序分析工具, 增强其在漏洞发现、安全检查等方面的能力。我们设计了 2 组实验来评估 LASM 语义提升的正确性和对下游静态分析工具的支持。所有实验均在搭载 Intel® Xeon® Gold 6148 2.40 GHz 160 核 x86 CPU, 1.4TB 内存, 安装了 Ubuntu 22.04 64 位操作系统的服务器上完成, 其他软件环境包括 Clang 10.0.1, SVF, CRIX, RetDec, mctoll 等。

6.1 实验设置与测试集介绍

6.1.1 正确性测试

我们使用提升后再编译执行的方法来评估 LASM 代码提升的正确性, 包含 2 个实验, 测试集信息见表 2。一个实验是提升程序的所有代码, 测试集为 LLVM Test Suite[30]。该实验先将 C 语言代码编译成汇编代码, 再使用 LASM 将汇编代码提升成 Lifted-IR, 然后将 Lifted-IR 编译成可执行文件并执行, 通过执行结果是否正确来判断 LASM 是否正确提升了汇编代码的语义。另一个实验是提升程序的部分代码, 用于评估语义融合的正确性, 测试集为 SPEC CPU 2006[31] 中的 C 语言测试用例。该实验先将一个 C 语言程序的部分.C 文件编译成汇编文件, 将其他所有.C 文件编译成 High-IR, 然后使用 LASM 将汇编文件提升成 Lifted-IR, 再将 High-IR 和 Lifted-IR 融合成 Mixed-IR, 进而编译成可执行文件并执行。如果执行结果正确, 则表明 LASM 的语义融合是正确的。

Table 2 Test Datasets for LASM Correctness Testing

表 2 用于 LASM 正确性测试的测试集

测试集	测试目的	测试用例数量
LLVM Test Suite	测试语义提升正确性	119
SPEC CPU 2006	测试语义融合正确性	10

LLVM Test Suite 是 LLVM 官方提供的测试套件, 用于对 LLVM 编译器进行测试。该测试套件中的 SingleSource 测试集包含了众多测试用例。LASM 暂不支持浮点指令、向量指令以及弱符号。测试集中包含这些程序特性的测试用例将被排除。经统计, 测试集中有 175 个 C 语言测试用例能在 x86 平台上编译运行, 且在控制台产生输出, 这其中有 55 个使用了浮点指令或者向量指令, 还有 1 个使用了弱符号。因此, LLVM Test Suite 中一共有 119 个测试用例适合用于 LASM 的正确性测试。

SPEC CPU 2006 测试集由现实世界的复杂应用程序组成, 包含了丰富的语法现象, 常被用于编译器的正确性测试和编译优化测试。我们选取了 SPEC CPU 2006 中 10 个使用 C 语言实现的测试用例用于测试 LASM 的语义融合是否正确。

6.1.2 支持静态分析工具的测试

我们通过提升被分析程序中的内嵌汇编代码来评估 LASM 支持下游静态分析工具的能力。先使用静态分析工具对含有内嵌汇编的测试用例进行分析, 得到提升内嵌汇编前的分析结果; 然后使用 LASM 提升测试用例中的内嵌汇编代码; 再使用静态分析工具分析对其进行分析, 得到提升内嵌汇编后的分析结果。检查对比 2 份分析结果, 分析 LASM 能否支持下游的静态程序分析工具。

我们还将与二进制代码提升工具进行横向对比, 以评估 LASM 在支持静态程序分析工具分析内嵌汇编上相对它们的优势。这类工具能将二进制文件提升为 LLVM IR, 在功能上与 LASM 相似。在该类工具中, RetDec[32] 和 mctoll[33] 是目前对静态程序分析支持最好的 2 款工具[34]。本文选择这 2 款工具作为 LASM 的横向比较对象。我们使用-O0 优化选项将测试集的源码编译成二进制文件, 然后用 RetDec 和 mctoll 将其提升为 LLVM IR 并交给静态程序分析工具分析, 得到分析结果后与 LASM 的支持情况进行对比。

我们选择 SVF Saber[22], CRIX[14] 两个开源的静态分析工具来评估 LASM 对静态分析工具的支持。同时, 我们还基于 SVF 实现了 DataProt, 并评估 LASM 对它的支持。DataProt 是对 Linux kernel 全局数据进行分类的静态分析工具, 分类的策略类似于文献 [35]。SVF Saber 的分析对象是 SVF Test-Suite[21], 而 CRIX 和 DataProt 的分析对象是 Linux kernel 5.10.112。实验中使用的静态分析工具和被分析程序如表 3 所示。

Table 3 Static Analysis Tools and Programs to Be Analyzed

表 3 静态分析工具和被分析程序

静态分析工具	静态分析工具的功能	被分析的程序	被分析程序数量
SVF Saber	检测内存泄漏和重复释放程序缺陷	SVF Test-Suite	139
CRIX	检测内核中关键数据的安全检查缺失	Linux kernel 5.10.112	1
DataProt	对内核的全局数据进行分类保护	Linux kernel 5.10.112	1

SVF Saber 简称 Saber, 是 SVF 官方提供的基于 SVF 的静态程序缺陷检测工具, 它能分析 C 语言程序中内存泄漏(memory leak)和重复释放(double free)缺陷。与此同时, SVF 还提供了基准测试套件 SVF Test-Suite, 测试套件中含有 47 个重复释放测试用例和 92 个内存泄漏测试用例。测试用例的源码中带有人工标注, 提示代码中何处存在缺陷, 为测试静态分析工具提供了校验基准。Saber 对这些测试用例进行分析后, 会根据测试用例中的标记判断分析结果是否正确。跟踪数据流和指针传播是程序缺陷检测流程中的重要环节。Saber 无法分析内嵌汇编代码, 程序中的内嵌汇编代码会影响它对数据流和指针传播的跟踪。SVF 基准测试套件中的测试用例都不包含汇编代码, 因此我们将在 SVF Test-Suite 的内存泄漏和重复释放测试用例中插入内嵌汇编片段。这些内嵌汇编代码具有传播指针的功能。插入内嵌汇编后, 分别使用 Saber 分析 SVF Test-Suite 的原始测试用例、带内嵌汇编的测试用例, 以及经过 LASM 提升处理的带内嵌汇编的测试用例, 通过比较 3 次分析的结果来评估 LASM 对 Saber 的支持情况。如果向测试用例中插入内嵌汇编后 Saber 无法得到正确的分析结果, 但经过 LASM 的提升处理后, 能得到和分析原始测试用例一致的分析结果, 则证明 LASM 能支持 Saber 分析汇编代码中的指针传播。

CRIX 是一个静态分析工具, 用于检测 Linux kernel 对关键变量缺少安全检查的程序缺陷。它只对常见的汇编函数进行了手动建模。我们将在 Linux kernel 中插入影响数据流的内嵌汇编, 这些内嵌汇编无法被 CRIX 分析到。如果在这些内嵌汇编的影响下, CRIX 无法检出原本能检测到的程序缺陷, 而使用 LASM 提升内嵌汇编后, CRIX 的检测能力恢复了, 就证明 LASM 能支持 CRIX 对 Linux kernel 的分析。

DataProt 是我们基于 SVF 开发的静态分析工具, 它能分析 Linux kernel 对全局数据的读写, 并根据 kernel 对数据的读写发生在 kernel 运行时的哪个阶段对数据进行分类。Linux kernel 的运行分为初始化阶段和初始化后阶段。那些仅在初始化阶段被调用的函数被开发者打上了__init 标签。__init 是 Linux kernel 提供的一个宏, 被这个宏标记的函数代码会被编译器放置在.init.text 段。这个段所占据的内存会在 Linux kernel 初始化完成后被释放回收。相应地, 如果一个全局数据仅在初始化阶段被读, 开发者可以给它打上__initconst 标签, 如果仅在初始化阶段被读写, 则可以给它打上__initdata 标签。带有__initconst 和__initdata 标签的数据也会在 Linux 完成初始化后被释放回收。从 Linux kernel 4.6 开始, __ro_after_init 标签被引入[36], 用于标记那些在初始化阶段被读写而在初始化后阶段只会被读的全局数据。通过给 Linux kernel 的全局数据打上标签, 可以将对数据的访问权限限制在最低限度, 做到更细粒度的权限管控。但 Linux kernel 开发者仅对少量数据打上了这些标签。如果能找到更多可以打上标签, 但实际未加标签的全局数据, 就能让 Linux kernel 变得更加安全。

DataProt 基于 SVF 的指针分析结果和程序赋值图(program assignment graph, PAG), 获取代码对全局数据的读写情况, 再根据读写操作发生的阶段对数据进行分类。数据分类必须是保守的, 因为错误的分类会导致 Linux kernel 在运行时崩溃。比如, Linux kernel 实际上在初始化阶段和非初始化阶段都对一个数据进行了读写操作, 但静态分析将其分类为 RoAfterInit 类(即初始化后只读), 那么当初始化后阶段发生写操作时, 内核就会因为向只读内存区域写入数据而发生崩溃。因此, 降低 DataProt 的误报及提升分析精度就变得尤为关键。导致 DataProt 出现误报的关键原因之一是内核中存在大量内嵌汇编代码, 比如原子写操作的实现是通过内嵌汇编实现的。

在实验中我们使用 defconfig 配置将 Linux kernel 5.10.112 编译成待分析的 LLVM IR。然后分别使用 DataProt 分析原版的 Linux kernel 和经过 LASM 提升处理后的 Linux kernel, 并对分析结果进行人工校验, 以评估 LASM 对 DataProt 的支持情况。如果经过 LASM 处理后, DataProt 的误报减少了, 则证明 LASM 能支持 DataProt 对 Linux kernel 的分析。

6.2 实验结果与分析

6.2.1 正确性测试结果分析

以 LLVM Test Suite 为测试集的提升后再编译执行实验结果如表 4 所示。所有测试用例的汇编代码均能被 LASM 提升为 LLVM IR，且都能再被编译成可执行文件，经提升再编译得到的 119 个可执行文件中，110 个可以正确执行。即 LASM 正确提升了 92.44% 的测试用例。其余 9 个测试用例的输出结果与基准输出不一致。我们对这 9 个测试用例进行了逐一分析，出错的原因是测试用例中包含了我们暂未支持或未完全支持的语法现象和程序行为，比如多线程程序等，详细分析见表 5。实验结果表明，那些 LASM 已经完全支持的指令和语言特性都被正确处理了。

Table 4 Experimental Results of Lifting and Recompiling LLVM Test Suite

表 4 提升再编译 LLVM Test Suite 实验结果

测试用例总数	提升的数量	再编译数量	正确执行数量	正确提升的比例/%
119	119	119	110	92.44

以 SPEC CPU 2006 作为测试集，测试 LASM 语义融合正确性的实验中，我们选择性地将每个测试用例的部分.C 文件编译成汇编代码，再对其进行提升处理。被选中的.C 文件被称为编译—提升测试文件。

我们遵循以下原则来选择编译—提升测试文件。首先，该文件中必须包含热点函数的实现。所谓的热点函数就是被其他.C 文件中的代码调用超过 5 次。满足这个条件意味着 Lifted-IR 和 High-IR 之间有较多跨边界的调用，这种调用需要通过胶水函数。其次，编译—提升测试文件中不能包含浮点指令和向量指令，或者指令操作数包含有 XMM 等向量寄存器。LASM 目前尚未支持这些指令。被选为编译—提升测试文件的.C 文件如表 6 所示。经提升—再编译后，所有 SPEC CPU 2006 中的测试用例均可正确执行，这表明 LASM 的语义融合是正确的。同时，所有编译—提升测试文件中的 C 代码行数总量达到了 18 518 行，也进一步验证了 LASM 语义提升的正确性。

Table 5 Reasons of Execution Failure After Lifting and Recompiling LLVM Test Suite

表 5 提升再编译 LLVM 测试套件后执行错误的原因

序号	测试用例名称	执行错误原因
1	2006-01-29-SimpleIndirectCall.c	使用了变参函数
2	2008-07-13-InlineSetjmp.c	使用了 *setjump*
3	tls.c	多线程程序
4	ConstructorDestructorAttributes.c	使用了__attribute__((constructor)) 属性
5	2003-05-14-AtExit.c	使用了 *atexit* 注册函数
6	2006-01-23-UnionInit.c	使用了#pragma pack() 语法
7	ccc.c	使用了变参函数
8	sign.c	使用了__attribute__((bitwidth(24))) 属性
9	2004-08-12-InlinerAndAllocas.c	使用了变参函数

LASM 的设计目标是提升内嵌汇编代码以辅助静态分析。表 5 所列的 LASM 不支持或部分支持的语言特性，对提升内嵌汇编几乎不构成影响。使用内嵌汇编实现复杂的功能逻辑非常容易出错，会增大软件的开发、调试和维护难度，推高研发成本。哪怕在内嵌汇编中只调用了库函数 *printf*，也容易出现微妙且不易排查的错误[37]。因此，开发人员会尽力避免在内嵌汇编中实现复杂的功能逻辑和使用复杂的语言特性[26]。在本文的实验中，被提升的内嵌汇

Table 6 Details of Compile-Lift Test Files in SPEC CPU 2006 Test Set

表 6 SPEC CPU 2006 测试集中编译—提升测试文件详情

测试用例编号	测试用例名称	测试用例代码量	编译-提升测试文件的名称	编译-提升文件代码量	热点函数数量
400	perlbench	15.72 万行	doop.c	1 431 行	15
401	bzip2	0.83 万行	huffman.c	246 行	5
403	gcc	52.10 万行	cfgbuild.c	883 行	13
429	mcf	0.27 万行	implicit.c，treeup.c，psimplex.c	761 行	7
445	gobmk	19.72 万行	readconnect.c	1 669 行	6
456	hmmer	3.60 万行	hsregex.c，sqio.c	3 290 行	48
458	sjeng	1.39 万行	attacks.c，moves.c，neval.c	3 731 行	26
462	libquantum	0.44 万行	omuln.c，oaddn.c，qft.c	459 行	16
464	h264ref	5.16 万行	mb_access.c，sei.c	2 338 行	58
482	sphinx3	2.51 万行	dict.c，hmm.c，lextree.c，mdef.c，vithist.c	3 710 行	39

编代码均未使用__attribute__和#pragma 等特性，也未调用 *atexit*, *setjump* 和变参函数。LASM 暂不支持或不完全支持的特性几乎不会出现在内嵌汇编代码中。因此，LASM 是否正确提升了单条指令，以及是否正确生成了的胶水函数，才是实现其设计目标的关键。

6.2.2　支持静态分析工具的测试结果分析

在 SVF Test-Suite 的内存泄漏和重复释放 2 组测试用例中插入影响数据流的内嵌汇编后，Saber 对每个测试用例的分析表现都变差了。插入内嵌汇编前，Saber 能检测到的内存泄漏和重复释放缺陷，在插入能影响该缺陷数据流的内嵌汇编后就无法检测到该缺陷的存在。在经过 LASM 的代码提升处理后，Saber 对所有测试用例的检测结果和插入内嵌汇编前是一致的。实验结果表明，LASM 能消除测试用例中的内嵌汇编对 Saber 的副作用，让它能跟踪程序中完整的数据流。

下面以测试用例 df0.c 为例，分析 LASM 如何帮助 Saber 分析内嵌汇编中的数据流。测试用例 df0.c 的代码图 14(a)所示，代码存在重复释放缺陷。程序的第 2 行代码让指针 *i* 指向了动态开辟的内存，随后在第 4 行通过内嵌汇编将指针 *i* 赋值给指针 *ptr*，最后在第 5 行和第 6 行分别释放了指针 *i* 和 *ptr* 指向的内存。显然，指针 *ptr* 和 *i* 指向的是同一块内存，第 6 行代码是对该内存的重复释放。

图 14(b)是编译图 14(a)代码得到的 LLVM IR，其中的第 2~6 行 IR 将指向动态内存的指针存到了变量 %i 中。第 8 行是 1 条 call asm 指令，对应图 14(a)中第 4 行的内嵌汇编。这条 call 指令的参数是虚拟寄存器 %1，它是指向动态内存的指针，返回值存到了虚拟寄存器 %2 中。因为 Saber 无法分析第 8 行代码，所以它无法在虚拟寄存器 %1 和虚拟寄存器 %2 之间建立起正确的数据流关系，进而无法分析出在代码执行完第 9 行后，变量 %i 和 %ptr 存储的是指向同一块动态内存的指针。所以 Saber 无法分析出第 13 行代码是对该内存的重复释放。显然，Saber 不能检测到重复释放程序缺陷，是因为在它无法完整跟踪 %i → %1 → %2 → %ptr → %4 的数据流传播链。第 8 行代码中的内嵌汇编让 Saber 无法建立 %1 和 %2 之间数据流关系。

使用 LASM 对图 14(b)所示代码进行提升后，得到了图 14(c)所示的 Mixed-IR。可以看到，IR 中已经没有了内嵌汇编，Saber 能分析其中的每一条指令的语义。图 14(c)的第 3 行完成了动态内存申请，并将内存首地址存入虚拟寄存器 %call 中，第 4 行将 %call 的值赋值给虚拟寄存器 %1，并在第 32 行通过 %1 完成对内存的第 1 次释放操作。第 2 行代码创建了一个局部 Env 结构用于模拟硬件环境。第 5~30 行代码是 LASM 提升内嵌汇编得到的 IR 指令，已经被内联到了函数@main 中。其中，第 17 行模拟的是将函数第 1 个参数传递给 Rdi 寄存器。第 3, 18, 30, 31, 33 行完成重复释放操作的数据流传播，即 %call → %RAX.i → %15 → %16。图 14(c)的代码是经过编译优化得到的。图 14(b)第 8 行代码调用内嵌汇编的语义是将第 1 个参数的值作为返回值。被优化前的 Lifted-IR 数据流应该是 %call → %Env.RDI → %Env.Rax，被优化成了 %call → %Env.Rax。经过优化之后，数据流关系变得更加清晰了。Saber 可以检测跟踪上述数据流，并分析出 %1 和 %16 是指针别名，所以能检测出第 33 行的重复释放缺陷。

通过上述分析可以看出，程序中的内嵌汇编被 LASM 提升为 LLVM IR 后，原本在 IR 层断裂的数据流被衔接上了。实验结果表明 LASM 能支持 Saber 检测和跟踪内嵌汇编代码中的数据流，帮助它获得程序完整的数据流信息，从而发现程序中的安全缺陷。

作为对比，我们先将插入了内嵌汇编的内存泄漏和重复释放测试用例编译成可执行文件，然后分别使用二进制代码提升工具 RetDec 和 mctoll 将其提升为 LLVM IR，再使用 Saber 对其进行检测。实验结果显示，在 2 个测试集上，一部分测试用例经 RetDec 提升后，其中的安全缺陷能被 Saber 检测到。但仅有个别测试用例经 mctoll 提升后，其中的安全缺陷能被 Saber 检测到。这一结果和文献 [34] 的实验结果相互印证。

使用-O0 编译优化选项将源码文件 df0.c(如图 14(a)所示)编译成二进制文件并进行反汇编，其中函数 main 的汇编代码如图 15(a)所示。RetDec 提升 df0.c 含调试信息的二进制文件，得到如图 15(b)所示的 LLVM IR，Saber 能检测到其中存在重复释放安全缺陷。用它提升不含调试信息的二进制文件，则得到如图 15(c)所示的 IR，它将函数@DOUBLEFREEM-ALLOC, @DOUBLEFREE, @SAFEFREE 的原型推断错了，进而导致了 Saber 的检测失败。使用 mctoll 进行提升，得到如图 15(d)所示的 IR，Saber 无法检测到其中的安全缺陷，主要原因是存在指针和整型之间的强制类型转换，以及@DOUBLEFREE 和@SAFEFREE 的函数原型推测错误。mctoll 的提升结果不受调试信息影响。

在深入分析所有测试用例后发现，二进制代码

```
1  int main(){
2        int *i=DOUBLEFREEMALLOC(sizeof(int));
3        int *ptr;
4        __asm__volatile(" mov %1, %0\n\t" :"=r"(ptr) :"r"(i));
5        SAFEFREE(i);
6        DOUBLEFREE(ptr);}
```

(a) 测试用例df0.c

```
1  define dso_local i32 @main(){
2     %i=alloca i32*, align 8
3     %ptr=alloca i32*, align 8
4     %call=call i8* @DOUBLEFREEMALLOC(i32 4)
5     %0=bitcast i8* %call to i32*
6     store i32* %0, i32** %i, align 8
7     %1=load i32*, i32** %i, align 8
8     %2=call i32* asm sideeffect " mov $1, $0\0A\09 ","=r, r, ~{dirflag}, ~{fpsr}, ~{flags}"(i32* %1)
9     store i32* %2, i32** %ptr, align 8
10    %3=load i32*, i32** %i, align 8
11    %call1=call i8* @SAFEFREE(i32* %3)
12    %4=load i32*, i32** %ptr, align 8
13    %call2=call i8* @DOUBLEFREE(i32* %4)
14    ret i32 0}
```

(b) 编译df0.c得到的df0.bc

```
1  define dso_local i32 @main(){
2     %0=alloca %struct.X86ENV, align 8
3     %call=tail call i8* @DOUBLEFREEMALLOC(i32 4)
4     %1=bitcast i8* %call to i32*
5     %2=bitcast %struct.X86ENV* %0 to i8*
6     %RAX.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 1, i32 1, i32 0, i32 0
7     %RAX.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 11, i32 1, i32 0, i32 0
8     %RAX.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 13, i32 1, i32 0, i32 0
9     %RAX.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 33, i32 1, i32 0, i32 0
10    store i8* null, i8** %RIP.i, align 8
11    %3=load i64, i64* bitcast (i8** @__fake_sp to i64*), align 8
12    %4=add i64 %3, -8
13    %5=inttoptr i64 %4 to i8*
14    %6=inttoptr i64 %4 to void (%struct.X86ENV*)**
15    store void (%struct.X86ENV*)* @_Z21__c2aglue_retP6X86ENV, void (%struct.X86ENV*)** %6, align 8
16    store i8* %5, i8** %RSP.i, align 8
17    store i8* %call, i8** %RDI.i, align 8
18    store i8* %call, i8** %RAX.i, align 8
19    %7=load i8*, i8** %RSP.i, align 8
20    %8=bitcast i8* %7 to void (%struct.X86ENV*)**
21    %9=load void (%struct.X86ENV*)*, void (%struct.X86ENV*)** %8, align 8
22    %10=ptrtoint i8* %7 to i64
23    %11=add i64%10, 8
24    %12=inttoptr i64 %11 to i8*
25    store i8* %12, i8** %RSP.i, align 8
26    call void %9 (%struct.X86ENV* nonnull %0)
27    %13=bitcast i8** %RSP.i to i64*
28    %14=load i64, i64* %13, align 8
29    store i64%14, i64* bitcast (i8** @__fake_sp to i64*), align 8
30    %15=bitcast i8** %RAX.i to i32**
31    %16=load i32*, i32** %15, align 8
32    %call1=call i8* @SAFEFREE(i32* %1)
33    %call2=call i8* @DOUBLEFREE(i32* %16)
34    ret i32 0}
```

(c) 提升df0.bc的内嵌汇编后得到的Mixed-IR

Fig. 14 LASM supports Saber to analyze the data flow of inline assembly

图 14 LASM 支持 Saber 分析内嵌汇编的数据流

被 RetDec 和 mctoll 提升后，程序的缺陷能否被 Saber 检测到的最关键因素是这 2 个工具本身。所以我们将原始的仅包含 C 语言代码的 SVF Test-Suite 编译成二进制文件进行实验，排除插入的内嵌汇编的干扰。

大量测试用例经 RetDec 提升后，其包含的安全缺陷无法被 Saber 检测到。导致 Saber 检测失败的原因包括错误的代码提升、指针和整型之间的强制类型转换、过于复杂的控制流、冗余的双层条件判断等，如表 7 所示。RetDec 使用连续的 2 条 LLVM IR 指令 icmp 来表达汇编上 cmp+jne 的条件分支语义，本文称之为冗余的双层条件判断。冗余的 icmp 指令可被 LLVM 的编译优化消除。Saber 在优化后的 IR

```
1  <main>:
2     push    %rbp
3     mov     %rsp, %rbp
4     sub     $0x20, %rsp
5     mov     $0x4, %edi
6     call    401160 <DOUBLEFREEMALLOC>
7     mov     %rax, -0x8(%rbp)
8     mov     -0x8(%rbp), %rax
9     mov     %rax, %rax
10    mov     %rax, -0x10(%rbp)
11    mov     -0x8(%rbp), %rdi
12    call    4011c0 <SAFEFREE>
13    mov     -0x10(%rbp), %rdi
14    mov     %rax, -0x18(%rbp)
15    call    4011f0 <DOUBLEFREE>
16    xor     %ecx, %ecx
17    mov     %rax, -0x20(%rbp)
18    mov     %ecx, %eax
19    add     $0x20, %rsp
20    pop     %rbp
21    ret
```

(a) df0.c中main函数的汇编指令

```
1  define i32 @main(){
2  dec_label_pc_401220:
3     %0=call i8* @DOUBLEFREEMALLOC(i32 4)
4     %1=bitcast i8* %0 to i32*
5     %2=call i8* @SAFEFREE(i32* %1)
6     %3=call i8* @DOUBLEFREE(i32* %1)
7     ret i32 0
8  }
```

(b) 有调试信息下RetDec提升main函数汇编代码的结果

```
1  define i64 @main(i64 %argc, i8** %argv){
2  dec_label_pc_401220:
3     %0=call i64 @DOUBLEFREEMALLOC(i64 4)
4     %1=call i64 @SAFEFREE(i64 %0)
5     %2=call i64 @DOUBLEFREE(i64 %0)
6     ret i64 0
7  }
```

(c) 无调试信息下RetDec提升main函数汇编代码的结果

```
1  define dso_local i32 @main() {
2  entry:
3     %stktop_8=alloca i8, i32 40, align 1
4     %tos=ptrtoint ptr %stktop_8 to i64
5     %0=add i64 %tos, 16
6     %RBP_N.24=inttoptr i64 %0 to ptr
7     %1=add i64 %tos, 24
8     %RBP_N.16=inttoptr i64 %1 to ptr
9     %2=add i64 %tos, 32
10    %RBP_N.8=inttoptr i64 %2 to ptr
11    %3=add i64 %tos, 0
12    %RSP_P.0=inttoptr i64 %3 to ptr
13    store i64 3735928559, ptr %RSP_P.0, align 8
14    %RBP=ptrtoint ptr %RSP_P.0 to i64
15    %4=call ptr @DOUBLEFREEMALLOC(i32 4)
16    %RAX=ptrtoint ptr %4 to i64
17    store i64 %RAX, ptr %RBP_N.8, align 1
18    %memload=load i64, ptr %RBP_N.8, align 1
19    store i64 %memload, ptr %RBP_N.16, align 1
20    %memload1=load i64, ptr %RBP_N.8, align 1
21    %RAX2=call i64 @SAFEFREE(i64 %memload1)
22    %memload3=load i64, ptr %RBP_N.16, align 1
23    store i64 %RAX2, ptr %RBP_N.24, align 1
24    %RAX4=call i64 @DOUBLEFREE(i64 %memload3)
25    store i64 %RAX4, ptr %stktop_8, align 1
26    ret i32 0
27 }
```

(d) mctoll提升main函数汇编代码的结果

Fig. 15 Lifting results of the df0.c binary via RetDec and mctoll

图 15 RetDec 和 mctoll 提升 df0.c 的二进制文件的结果

Table 7 Reasons for Saber Detection Failure After RetDec Lifting

表 7 经 RetDec 提升后 Saber 检测失败的原因

缺陷类别	Saber 检测失败的测试用例数量	导致检测失败的原因	合计	失败率/%
内存泄漏	25	指针与整型的强制类型转换	44	47.83
	11	冗余的双层条件判断		
	6	错误的代码提升		
	2	复杂的控制流		
重复释放	3	错误的代码提升	3	6.38

上能得到正确的检测结果。在将测试集编译成二进制代码时，如果不使用-g 选项生成调试信息，经 RetDec 提升后，Saber 在 2 个测试集上均无法得到正确的检测结果。

使用 mctoll 提升测试集的二进制代码后，Saber 在重复释放测试集中没有正确的检测结果；在内存泄漏测试集上，Saber 虽然正确检测了 12 个测试用例，但可以认为这些都是无效的检测。因为这些测试用例都仅包含从未释放缺陷，而 Saber 对所有测试用例都只报告该缺陷。在 LASM 的实验中，我们在仅含有从未释放缺陷的测试用例中使用内嵌汇编来调用动态申请内存的函数。因此，经 LASM 提升后，Saber 能正确检测到缺陷就表明内嵌汇编被正确提升了。

为了测试 LASM 对 CRIX 的支持情况，我们先用 CRIX 分析 Linux kernel 5.10.112，在其分析报告中随机选择了 10 个安全检查缺失的关键数据，并插入内嵌汇编影响与之相关的数据流。插入内嵌汇编的.C 文件和函数以及 CRIX 的检测结果如表 8 所示。CRIX 直接分析插入了内嵌汇编的 Linux kernel 无法找到这 10 处程序缺陷，而使用 LASM 将这些插入的内嵌汇编代码提升为 LLVM IR 之后，CRIX 能摆脱内嵌汇编的干扰，成功将这 10 处程序缺陷检查出来。

我们以 drm/drm_memory.c 文件中插入的内嵌汇编代码为例，分析 LASM 如何支持 CRIX 检测内嵌汇编中的数据流。如图 16 所示，插入的内嵌汇编代码位于图 16(a)的第 6~10 行，用于调用函数 *ioremap_wc*，其功能等价于被注释的第 5 行代码。如果使用高级语言调用该函数，CRIX 能检测到其返回值需要进行安全检查。内嵌汇编代码未被 LASM 提升的 LLVM IR 如图 16(b)所示，CRIX 通过分析其中的第 9 和 10 这 2 行代码得到的信息是：虚拟寄存器 %7 是 call asm 指令的返回值，随后它被赋值给了虚拟寄存器

Table 8 Detecting Results of CRIX with and without the Support of LASM

表 8 有无 LASM 支持下 CRIX 的检测结果

插入汇编的文件	插入汇编的函数	影响的关键数据	CRIX	LASM 联合 CRIX
drm/drm_memory.c	*drm_legacy_ioremap_wc*	*map->handle*	失败	成功
mm/page_alloc.c	*alloc_pages_exact*	*addr*	失败	成功
mm/swapfile.c	*map_swap_entry*	*sis*	失败	成功
kernel/hpet.c	*hpet_set_mapping*	*hpet_virt_address*	失败	成功
init/initramfs.c	*dir_add*	*de->name*	失败	成功
mac80211/mlme.c	*ieee80211_send_assoc*	*ifmgd->assoc_req_ies*	失败	成功
pci/irq.c	*pci_request_irq*	*devname*	失败	成功
acpi/boot.c	*mp_config_acpi_gsi*	*ioapic*	失败	成功
mm/vmstat.c	*init_mm_internals*	*mm_percpu_wq*	失败	成功
ipv4/devinet.c	*__inet_insert_ifa*	*ret*	失败	成功

%handle4。但 CRIX 无法分析出保存在 %7 中的值是通过调用函数 *ioremap_wc* 产生的。因此，它无法做出是否应该对 %handle4 进行检查的判断。

用 LASM 提升内嵌汇编代码后得到如图 16(c)所示的代码，其中的第 4~29 行代码模拟内嵌汇编调用函数 *ioremap_wc* 的语义，并将返回值存入虚拟寄存器 %handle4 中，该虚拟寄存器中存放的是指向变量 *map->handle* 的指针。经过代码提升后，CRIX 就能检测到代码中的完整数据流了。实验结果表明，在 LASM 的支持下，CRIX 跨越了内嵌汇编带来的数据流跟踪障碍。

作为对比，我们分别使用 RetDec 和 mctoll 联合 CRIX 对内核进行检测。但这 2 个工具均无法提升内核的二进制文件。二者均在运行过程中发生崩溃，未能生成对应的 LLVM IR。

DataProt 通过分析 Linux kernel 对全局数据的读写，根据代码在 kernel 的初始化阶段和初始化后阶段对数据的读写情况，将数据分成 5 类，类别如表 9 所示。不同类别对数据的访问限制的程度是不一样的。比如属于 Data 类的数据可以在 Linux kernel 的初始化阶段和初始化后阶段被读和写，相当于没有限制。InitConst 类别则限制数据只能在初始化阶段被读，对数据的访问限制是最强的。如果 DataProt 通过静态分析判断一个数据可以被划分到比它原来所属类别具有更强读写限制的类别中，就意味着 DataProt 找到了一个可以被保护的数据。实验中使用 Clang 10.0.1 将 defconfig 配置的 Linux kernel 5.10.112 编译成 LLVM IR。在使用 LASM 提升内嵌汇编前后，我们 2 次使用 DataProt 对内核进行分析，得到 2 份分类结果，并人工逐一确认了每个数据是否被正确分类。

2 次分类的情况对比如图 17 所示，上方的实线段表示在 LASM 的支持下 DataProt 对数据的分类结果，下方的虚线段表示仅使用 DataProt 进行分类的结果，2 条线段的重叠部分表示在 2 次分类中都被标记的数据。实验表明，没有 LASM 的支持下，DataProt 找到 1 886 个可保护数据，其中被错误分类的数据有 237 个，即精度为 87.43%(1 649/1 886)。在 LASM 的支持下，可以找到 1 994 个可保护数据，被错误分类的数据有 90 个，即精度为 95.49%(1 904/1 994)。需要注意的是，这 90 个数据在有和没有 LASM 支持的情况下，DataProt 的分析结果是一样的。有了 LASM 的支持后 DataProt 的分析精度提高了 8.06 个百分点。对比是否有 LASM 支持的 2 份分类结果可以看到，在 LASM 的支持下，有 147 个数据的分类错误被纠正了，占其错误分类总数量的 62.03%(147/237)。同时它还能帮助 DataProt 多找到了 244 个可保护数据，经人工分析确认，分类结果都是正确的。即在 LASM 的帮助下，DataProt 多找到了 14.80% 的可保护数据，且没有引入错误的分类结果。

我们以 Linux kernel 中的数据 atomic_t *dm_global_event_nr* 为例，分析 LASM 对 DataProt 的支持。没有 LASM 支持时，DataProt 将 *dm_global_event_nr* 分类为 Const 类，即没有检测到任何对该数据的写操作。但实际上函数 void *dm_issue_global_event*(void) 对它有写操作。如图 18(a)的 C 代码所示，代码以 *dm_global_event_nr* 的地址为实参调用了函数 *atomic_inc*。函数 *atomic_inc* 则通过内嵌汇编实现了原子写操作，如图 18(b)所示。函数 *dm_issue_global_event* 不带 __init 标签，意味着它在初始化后阶段被调用。因此，它对 *dm_global_event_nr* 的写操作发生在初始化后阶段，应该将 *dm_global_event_nr* 划分到 Data 类。

函数 *dm_issue_global_event* 的 LLVM IR 代码如

```
1  void drm_legacy_ioremap_wc(struct drm_local_map *map, struct drm_device *dev){
2        if(dev->agp && dev->agp->cant_use_aperture && map->type==_DRM_AGP)
3              map->handle=agp_remap(map->offset, map->size, dev);
4        else{
5              // map->handle=ioremap_wc(map->offset, map->size);
6              __asm__("movq %1, %%rdi      \n\t"
7                      "movq%2, %%rsi       \n\t"
8                      "callq ioremap_wc    \n\t"
9                      "movq %%rax, %0      \n\t"
10             :"=r"(map->handle) :"r"(map->offset), "r"(map->size));}
11 }
```

(a) 插入了内嵌汇编的C代码

```
1  define dso_local void @drm_legacy_ioremap_wc(%struct.drm_local_map* nocapture %map, %struct.drm_device* nocapture readonly %dev){
2  ⋯
3  if.else:                                          ; preds=%land.lhs.true, %entry, %land.lhs.true3
4    %handle4=getelementptr inbounds %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 4
5    %offset5=getelementptr inbounds %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 0
6    %5=load i64, i64* %offset5, align 8
7    %size6=getelementptr inbounds %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 1
8    %6=load i64, i64* %size6, align 8
9    %7=tail call i8* asm "movq $1, %rdi\0A\09movq $2, %rsi\0A\09callq ioremap_wc\0A\09movq %rax, $0\0A\09", "=r, r, r, ~{dirflag}, ~{fpsr}, ~{flags}"(i64 %5, i64 %6)
10   store i8* %7, i8** %handle4, align 8
11   br label %if.end
12 ⋯}
```

(b) 编译C代码得到的IR

```
1  define dso_local void @drm_legacy_ioremap_wc(%struct.drm_local_map* nocapture %map, %struct.drm_device* nocaptu rereadonly %dev){
2  ⋯
3  if.else:                                          ; preds=%land.lhs.true3, %land.lhs.true, %entry
4    %handle4=getelementptr inbounds %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 4
5    %6=getelementptr %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 0
6    %7=load i64, i64* %6, align 8
7    %size6=getelementptr inbounds %struct.drm_local_map, %struct.drm_local_map* %map, i64 0, i32 1
8    %8=load i64, i64* %size6, align 8
9    ⋯
10   %RAX.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 1, i32 1, i32 0, i32 0
11   %RSI.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 1, i32 9, i32 0, i32 0
12   %RDI.i=getelementptr inbounds %struct.X86ENV, %struct.X86ENV* %0, i64 0, i32 1, i321 1, i32 0, i32 0
13   ⋯
14   %19=bitcast i8** %RDI.i to i64*
15   store i64%7, i64* %19, align 8
16   %20=bitcast i8** %RSI.i to i64*
17   store i64 %8, i64* %20, align 8
18   ⋯
19   %24=load i64, i64* %19, align 8
20   %25=load i64, i64* %20, align 8
21   ⋯
22   %26=tail call i8* @ioremap_wc(i64 %24, i64 %25)
23   store i8* %26, i8** %RAX.i, align 8
24   ⋯
25   %29=bitcast i8** %RAX.i to i64*
26   %30=load i64, i64* %29, align 8
27   ⋯
28   %31=bitcast i8** %handle4 to i64*
29   store i64 %30, i64* %31, align 8
30   br label %if.end
31 ⋯}
```

(c) 提升内嵌汇编后得到的Mixed-IR

Fig. 16 LASM supports CRIX to analyze the data flow of inline assembly

图 16 LASM 支持 CRIX 分析内嵌汇编的数据流

Table 9 Data Categories in Linux Kernel

表 9 Linux 内核中的数据类别

数据类别	标签/关键字	读写操作发生的阶段
Data		初始化阶段和初始化后阶段具有写操作
Const	const 关键字	初始化阶段和初始化后阶段均只有读操作
RoAfterInit	__ro_after_init	初始化阶段有读写操作， 初始化后阶段仅有读操作
Init	__initdata	仅初始化阶段有读写操作
InitConst	__initconst	仅初始化阶段有读操作

图 18(c)所示。因为 DataProt 无法分析其中内嵌的汇编代码，所以检测不到该函数对数据 *dm_global_event_nr* 的写操作。使用 LASM 提升内嵌汇编代码后得到的 IR 代码如图 18(d)所示。LASM 对提升内嵌汇编得到的 IR 做了面向静态分析的优化，简化了原子操作的语义，让数据流更精简。同时，编译优化还将模拟 incl 指令更新 eflags 寄存器的指令消除了。图 18(d)中第 2~4 行代码完成了对 *dm_global_event_nr* 的写操作，这 3 行 IR 的语义很容易被 DataProt 分

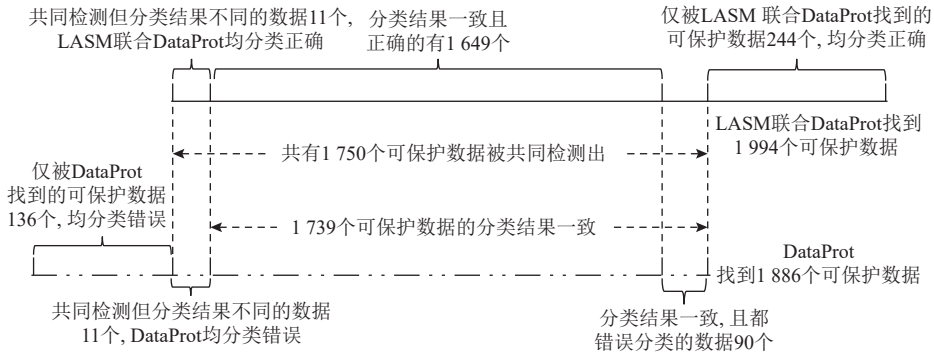


Fig. 17 Comparison of data classification results by DataProt with and without the support of LASM

图 17 有无 LASM 支持下 DataProt 数据分类结果对比

```
1 void dm_issue_global_event(void){
2       atomic_inc(&dm_global_event_nr);
3       wake_up(&dm_global_eventq);
4 }
```

(a) 函数*dm_issue_global_event*的C代码

```
1 static inline void atomic_inc(atomic_t *v){
2       asm volatile(LOCK_PREFIX "incl %0"
3                    : "+m" (v->counter));
4 }
```

(b) 函数*atomic_inc*的C代码

```
1 define dso_local void @dm_issue_global_event(){
2   tail call void asm sideeffect ".pushsection .smp_locks, \22a\22\0A.balign 4\0A.long 671f - .\0A.popsection \0A671:\0A\09lock; incl $0","=*m, *m, ~{memory},
        ~{dirflag}, ~{fpsr}, ~{flags}" (i32* getelementptr inbounds (%struct.kuid_t, %struct.kuid_t* @dm_global_event_nr, i64 0, i32 0),
        i32* getelementptr inbounds (%struct.kuid_t, %struct.kuid_t* @dm_global_event_nr, i64 0, i32 0))
3   tail call void @__wake_up (%struct.wait_queue_head* nonnull @dm_global_eventq, i32 3, i32 1, i8* null)
4   ret void
5 }
```

(c) 函数@dm_issue_global_event包含内嵌汇编的IR

```
1 define dso_local void @dm_issue_global_event(){
2   %0=load i32, i32* getelementptr inbounds (%struct.atomic_t, %struct.atomic_t* @dm_global_event_nr, i64 0, i32 0)
3   %add.i.i.i.i=add i32 %0, 1
4   store i32 %add.i.i.i.i, i32* getelementptr inbounds (%struct.atomic_t, %struct.atomic_t* @dm_global_event_nr, i64 0, i32 0)
5   tail call void @__wake_up (%struct.wait_queue_head* nonnull @dm_global_eventq, i32 3, i32 1, i8* null)
6   ret void
7 }
```

(d) 提升内嵌汇编后函数@dm_issue_global_event的Mixed-IR

Fig. 18 LASM supports DataProt to analyze the data flow of inline assembly

图 18 LASM 支持 DataProt 分析内嵌汇编的数据流

析，数据流非常清晰，所以它能把数据 *dm_global_event_nr* 正确分类为 Data 类。

在通过静态程序分析对 Linux kernel 的全局数据进行分类的实验中，LASM 帮助分析工具分析和跟踪内嵌汇编中的数据流，减少了误报。在所有的静态程序分析中，跟踪被分析程序的数据流都是一项基础性任务。LASM 对 DataProt 的有效支持很大程度上反映了 LASM 对静态程序分析的支持。同时，鉴于 Linux kernel 的复杂性，实验在一定程度上说明了 LASM 具备支持静态程序分析工具分析大型复杂程序的能力。

7 讨 论

LASM 对程序的改变体现在 2 个方面。一是在程序中引入了新的数据类型和全局变量。被提升程序和 LASM 各自定义的全局变量只要不重名，便不会对静态分析造成影响。LASM 不修改程序中已有的数据类型定义，仅增加了少量数据类型定义，不影响静态分析工具对程序的分析。二是把内嵌汇编替换成了与其语义等价的 Lifted-IR。首先，LASM 能正确提升它已经支持的指令，也能正确生成胶水函数。那些 LASM 暂不支持的语言特性一般不会出现在内嵌汇编中。这意味着 Lifted-IR 中的数据流和控制流关系是正确的。实验表明，在经过 LLVM 的编译优化和 LASM 面向静态分析的优化后，静态分析工具能跟踪 Lifted-IR 中的数据流。相当于消除了内嵌汇编带来的分析盲区，这对静态分析具有正向的促进作用。

正确的提升与可支持静态分析的提升并不是等价的。假设在某些情况下静态分析工具无法跟踪 Lifted-IR 中的数据流，那么这部分 Lifted-IR 和它对应

的内嵌汇编一样，成为了分析盲区。对基于 LLVM IR 的分析工具而言，它此时面临的情况和分析提升前的代码是一样的。这种情况并不会让分析结果比提升内嵌汇编前更差。

8 相关工作

8.1 全程序分析

全程序分析寻求在静态程序分析中覆盖更多的代码，从而检测和跟踪更完整的数据流和控制流，发现更多潜在的漏洞或安全缺陷。现有的研究可以分为 2 类：一类是对程序中更多的信息进行建模，为静态分析器提供更多可被分析的语义信息。另一类是改进静态分析器，提高其分析效率，扩大分析规模，或增加对某类复杂语言特性的支持。前者聚焦在程序信息的表示上，后者力求提高静态分析器的处理能力。

WLLVM[38] 是一个将程序不同编译模块的 IR 合并成一个模块的工具，在分析 Linux kernel 时常常用到。CodeSonar 将程序包含的文件编译成抽象表示，然后将各个编译模块合成为全程序模型，并在此基础上进行全程序分析[39]。Quinlan 等人[40] 提出了一种紧凑精确的全程序 AST 表示，以完整表示不同编译模块的代码信息，让分析器能检测 C++程序中的 ODR（one-definition rule）安全缺陷。实现世界的复杂软件往往有多个组成部分，除了实现程序功能的代码，还包括外部库、开发框架的代码以及配置文件等。将软件的不同组成部分纳入分析中能获得更高的分析精度。例如，在分析基于 Spring 和安卓等框架开发的程序时，将程序中的 XML 等配置文件纳入分析，或者将框架代码和应用程序的代码结合起来分析，能提升分析精度[41-43]。增加对语言的动态属性的支持，可以帮助静态分析器发现更多函数调用，建立更完整的函数调用图，显著提升分析结果[44-45]。

很多研究致力于改进静态分析器的算法或数据结构，实现更加高效的分析，让分析器能分析更大规模的代码[46-49]，实现对整个程序的静态分析。静态分析需要消耗庞大的计算资源，Garbervetsky 等人[50] 实现了分布式的程序分析框架，通过增加计算资源来扩大分析器的输入规模。Schubert 等人[25] 则通过将分析模块化和组合化来加速分析过程。K-Miner 以系统调用接口为切入点，对内核进行划分来减少单次分析规模，并复用中间分析结果来完成对整个 Linux kernel 的分析[51]。

LASM 与上述工作是正交的，与下游静态分析工具之间是解耦的，通过在静态分析器的输入中融入内嵌汇编代码的语义信息，让它能捕捉到程序中更多的控制流和数据流信息，实现对 C/C++和汇编混合编程的软件进行全程序分析。

内嵌汇编是全程序分析的一大阻碍。目前缺乏通用且高效的面向全程序分析的内嵌汇编代码处理方案。TINA[3] 可将内嵌汇编代码转换成 C 语言代码，让面向 C 代码的形式化验证工具能处理内嵌汇编。Inception[52] 则将嵌入式程序中的 ARM 平台内嵌汇编代码转成 LLVM IR，让符号执行引擎 KLEE 能处理整个程序。这 2 个工具的设计思想和 LASM 相似，都是将内嵌汇编代码转换成更高级的代码表示，并且与程序中原始的高级代码相融合。但面向的应用场景和目标不一样，实现方法也有明显区别。kCPA[53] 在分析 Linux kernel 时将内嵌汇编转换成更低级的二进制代码，以使用现成的二进制代码分析工具对其进行分析，然后合并二进制代码与高级语言代码的分析结果。这种方法的缺点在于无法在同一种代码表示上完成全程序分析。另一种处理方法是人工对内嵌汇编的语义进行建模，CRIX[14] 和 LRSan[15] 都采用了该方法，缺点是耗时长且容易出错。

8.2 反编译

反编译是编译的逆过程，它将二进制代码转成高级语言[54-56]。目前最成功的反编译器是 Hex-Rays[57]，2007 年由 Ilfak Guilfanov 发布，它作为商业软件 IDA Pro (interactive disassembler pro)[58] 的内置反编译器，已经成为事实上的行业标准[59]。Hex-Rays 能够通过语义分析、程序控制流分析、函数边界分析等手段，将汇编代码反编译生成 C 代码。当前最优秀的开源反编译器是 Ghidra[60]，由 NSA 组织开发和维护，于 2019 年向公众开源。Ghidra 的反编译过程包含一系列的代码变换过程，类似于 LLVM，它具有不同的前端和后端来处理不同类型的代码[61]。DREAM[59,62]，Phoenix[63] 则是学术界 state-of-the-art 的 2 个反编译器。Phoenix 强调反编译的正确性，但引入了很多 goto 语句，DREAM 则强调减少 goto 语句。反编译技术难以恢复高级语义信息[64]，且只能将整个程序都编译成二进制代码后整体反编译，无法有效利用 High-IR 中的高级语义信息。

LASM 的设计目标是让基于 LLVM IR 的静态程序分析工具能分析程序中的汇编代码，这点与反编译器不同。反编译器以二进制作为输入，从反汇编开始到转换成高级语言，多个环节都存在语义推测，

会引入错误。而 LASM 以汇编代码为输入,通过模拟汇编指令语义的方式进行代码提升,能保证正确性。同时,维持 High-IR 不变,能更好地支持静态程序分析。

8.3 二进制代码提升

二进制代码提升是将二进制代码转化为某种 IR,有多个工具具备此功能[32-33,52,65-74],其中 angr[74] 将二进制代码提升为 VEX IR, McSema[66] 等工具将二进制代码提升为 LLVM IR。BinRec[68] 依赖动态反汇编,目的是维持正确性,缺点在于难以覆盖所有代码。其他几个工具则先通过静态反汇编将二进制代码转成汇编代码,然后进行代码提升。有 2 种方法将汇编代码提升为 LLVM IR。第 1 种是通过在 IR 上模拟机器状态的方式将汇编代码提升为 LLVM IR,该方法能保证正确性,McSema 和 BinRec 为代表的工具采用了这种方法。RetDec[32] 和 mctoll[33] 则是通过模式匹配等方法推断出高级语义,在提升过程中重建局部变量和恢复函数原型,好处是能生成更简洁的 IR,但不能保证正确性。

这些工具都有各自的设计目标和用途。BinRec 用于二进制程序加固和二进制重写,关注正确性。Instrew[70] 用于动态二进制重写和插桩,执行性能是其最关心的。McSema, mctoll, Rev.ng[72] 用于二进制分析。RetDec 主要用于逆向工程,提供了反编译工具 llvmir2hll。这些工具都没有把支持指针分析等静态程序分析手段作为其首要设计目标,产生的 IR 并不适合用于静态分析,比如 Instrew 输出的 IR 不连续,BinRec 将所有代码提升到一个 LLVM IR 函数中,彻底丧失了原始程序的调用图信息,McSema 提升的 IR 会阻碍 SVF 的指针分析。RetDec 在支持指针分析上略胜于 mctoll 和 McSema,但在 SVF 测试集上也只能得到数个正确的结果[34]。上述工具除 Inception[52] 都是将整个二进制文件提升为 LLVM IR, C/C++中的高级语义难以被恢复。反汇编、函数原型推测、局部变量重建等环节上的任何错误都会影响到静态分析的精度。如果为了让程序中的汇编代码能被静态分析器分析到,就彻底丢弃编译器生成的 High-IR,转而使用通过提升整个二进制文件得到的 IR,分析的精度很可能不升反降。Inception 为了让 KLEE 对嵌入式系统固件的符号执行能覆盖到内嵌汇编,做了面向符号执行的代码提升,针对 KLEE 进行了定制化设计。它以二进制可执行文件作为输入,在 IR 上将可执行文件的整个数据段表示为一个数组,不利于细粒度的数据流分析。

LASM 以支持静态分析为设计目标,在实现上与上述工具的不同之处在于,保留了编译器生成的 High-IR,仅仅提升程序中的汇编代码,保证汇编代码提升的正确性,并将 2 种 IR 融合在一起。这种技术路线和支持静态程序分析的目标更加契合。

9 结 论

高级语言和汇编语言混合编程被广泛使用,但使用汇编语言编程非常具有挑战性,容易出错,且错误难以被发现和定位。近年来涌现出了大量基于 LLVM IR 的静态程序分析工具,但这些工具都无法分析程序中的内嵌汇编代码。本文提出和实现了 LASM 系统,该系统能将软件中的内嵌汇编提升为 LLVM IR,并与编译高级语言得到的 IR 相融合,让基于 LLVM IR 的静态程序分析工具能对软件进行全程序分析,检测到涉及内嵌汇编的数据流,提升分析的精度,降低误报率。

作者贡献声明:赖远明是论文工作的主要完成人,负责设计与实现 LASM 系统,设计并实施了实验,分析实验结果,撰写和修改论文;王喆对系统设计提出了建议并修改论文;武成岗为论文撰写提供了建议和指导;于丁参与了部分汇编指令语义模拟函数的实现;侯承轩参与了面向静态分析优化的实现。

参 考 文 献

[1] Anguiano R. Linux kernel mainline 4.9 sloccount. txt [EB/OL]. (2016-12-12)[2024-10-16]. https://gist.github.com/ricardoanguiano/18125-b7eb3f26cf83724fb60662bdd2c

[2] OpenMathLib.OpenBLAS [EB/OL]. [2024-10-16]. https://github.com/OpenMathLib/OpenBLAS

[3] Recoules F, Bardin S, Bonichon R, et al. Get rid of inline assembly through verification-oriented lifting[C]//Proc of the 34th IEEE/ACM Int Conf on Automated Software Engineering (ASE'19). Piscataway, NJ: IEEE, 2019: 577−589

[4] Rigger M, Marr S, Kell S, et al. An analysis of x86−64 inline assembly in C programs[C]//Proc of the 14th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2018: 84−99

[5] Larabel M. Linus Torvalds takes on a performance patch: "I relax by playing with inline assembly" [EB/OL]. (2023-06-27)[2024-10-16]. https://www.phoronix.com/news/Linus-Torvalds-Relax-Inline-ASM

[6] Petkov B. [Git pull] x86/misc for 6. 5 [EB/OL]. (2023-06-27)[2024-10-16]. https://lore.kernel.org/lkml/fe807f09-9e4a-52d6-c057-b159-bee55510@linux.intel.com/T/

[7] Biesheuvel A. [Patch v2] ARM: Avoid literal references in inline assembly [EB/OL]. (2021-12-22)[2024-10-16]. https://lore.kernel.org/linux-arm-kernel/YcN4xJfH5NaPSibU@archlinux-ax161/T/

[8] Thomson P. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity[J]. Queue, 2021, 19(4): 1−13

[9] National Aeronautics and Space Administration (NASA). Software engineering and assurance handbook[EB/OL]. [2024-10-16]. https://swehb.nasa.gov/display/SWEHBVC

[10] Nichols Jr W R. The cost and benefits of static analysis during development[J]. arXiv preprint, arXiv: 2003.03001, 2020

[11] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proc of the 2nd Int Symp on Code Generation and Optimization(CGO 2004). Piscataway, NJ: IEEE, 2004: 75−86

[12] Sui Yulei, Xue Jingling. SVF: Interprocedural static value-flow analysis in LLVM[C]//Proc of the 25th Int Conf on Compiler Construction. New York: ACM, 2016: 265−266

[13] Li Guangwei, Yuan Ting, Li Lian. Study of state-of-the-art open-source C/C++ static analysis tools[J]. Journal of Software, 2022, 33(6): 2061−2081 (in Chinese)
(李广威, 袁挺, 李炼. 开源 CC++ 静态软件缺陷检测工具实证研究[J]. 软件学报, 2022, 33(6): 2061−2081)

[14] Lu Kangjie, Pakki A, Wu Qiushi. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences [C]//Proc of the 28th USENIX Security Symp (USENIX Security 19). Berkeley, CA : USENIX Association, 2019: 1769−1786

[15] Wang Wenwen, Lu Kangjie, Yew Pen-Chung. Check it again: Detecting lacking-recheck bugs in OS kernels[C]//Proc of the 25th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 1899−1913

[16] Liu Dinghao, Wu Qiushi, Ji Shouling, et al. Detecting missed security operations through differential checking of object-based similar paths[C]//Proc of the 28th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2021: 1627−1644

[17] Wu Qiushi, Pakki A, Emamdoost N, et al. Understanding and detecting disordered error handling with precise function pairing [C]//Proc of the 30th USENIX Security Symp (USENIX Security 21). Berkeley, CA: USENIX Association, 2021: 2041−2058

[18] Cifuentes C, Gauthier F, Hassanshahi B, et al. The role of program analysis in security vulnerability detection: Then and now[J]. Computers & Security, 2023, 135: 1−14

[19] Bessey A, Block K, Chelf B, et al. A few billion lines of code later: Using static analysis to find bugs in the real world[J]. Communications of the ACM, 2010, 53(2): 66−75

[20] Machiry A, Spensky C, Corina J, et al. Dr. Checker: A soundy analysis for Linux kernel drivers[C]// Proc of the 26th USENIX Security Symp (USENIX Security 17). Berkeley, CA : USENIX Association, 2017: 1007−1024

[21] SVF-tools.Test-Suite[EB/OL].[2024-10-16].https://github.com/SVF-tools/Test-Suite

[22] Sui Yulei, Ye Ding, Xue Jingling. Static memory leak detection using full-sparse value-flow analysis[C]//Proc of the 21st Int Symp on Software Testing and Analysis (ISSTA 2012). New York: ACM, 2012: 254−264

[23] LLVM-admin Team. The LLVM compiler infrastructure [EB/OL]. [2024-10-16]. https://llvm.org/

[24] Sattler F, Böhm S, Schubert P D, et al. SEAL: Integrating program analysis and repository mining[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(5): 121: 1−121: 34

[25] Schubert P D, Hermann B, Bodden E. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis [C]//Proc of the 35th European Conf on Object-Oriented Programming (ECOOP 2021). Dagstuhl, Germany: Schloss Dagstuhl -- Leibniz Zentrum für Informatik, 2021: 2: 1−2: 31

[26] Wohlferd D. Reasons you should not use inline asm [EB/OL]. (2016-04-26)[2024-10-16]. https://gcc.gnu.org/wiki/DontUseInlineAsm

[27] CVEdetails. com. Vulnerability details: CVE−2024−26706 [EB/OL]. [2024-10-16]. https://www.cvedetails.com/cve/CVE-2024-26706/

[28] CVEdetails. com. Vulnerability details: CVE−2023−52874 [EB/OL]. [2024-10-16]. https://www.cvedetails.com/cve/CVE-2023-52874/

[29] Recoules F, Bardin S, Bonichon R, et al. Interface compliance of inline assembly: Automatically check, patch and refine[C]//Proc of the 43rd IEEE/ACM Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2021: 1236−1247

[30] The LLVM Organization. LLVM-test-suite [EB/OL]. [2024-10-16]. https://github.com/llvm/llvm-test-suite

[31] The Standard Performance Evaluation Corporation (SPEC). SPEC CPU 2006 [EB/OL]. [2024-10-16]. https://www.spec.org/cpu2006/

[32] Avast. RetDec [EB/OL]. [2024-10-16]. https://github.com/avast/RetDec

[33] Yadavalli S B, Smith A. Raising binaries to LLVM IR with MCTOLL (WIP paper)[C]//Proc of the 20th ACM SIGPLAN/SIGBED Int Conf on Languages, Compilers, and Tools for Embedded Systems. New York: ACM, 2019: 213−218

[34] Liu Zhibo, Yuan Yuanyuan, Wang Shuai, et al. Sok: Demystifying binary lifters through the lens of downstream applications[C]//Proc of the 43rd IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2022: 1100−1119

[35] Li Guoren, Zhang Hang, Zhou Jinmeng, et al. A hybrid alias analysis and its application to global variable protection in the Linux kernel[C]//Proc of the 32nd USENIX Security Symp (USENIX Security 23). Berkeley, CA: USENIX Association, 2023: 4211−4228

[36] Cook K. Introduce post-init read-only memory [EB/OL]. [2024-10-16]. https://lwn.net/Articles/676145/

[37] Petch M. Calling printf in extended inline ASM [EB/OL]. [2025-02-25]. https://stackoverflow.com/questions/37502841/calling-printf-in-extended-inline-asm/37503773#37503773

[38] Travitch.whole-program-LLVM[EB/OL].[2024-10-16].https://github.com/travitch/whole-program-llvm

[39] Verifysoft Technology. CodeSonar—Static code analysis [EB/OL]. [2025-02-25]. https://www.verifysoft.com/en_codesonar.html

[40] Quinlan D, Vuduc R, Panas T, et al. Support for whole-program analysis and the verification of the one-definition rule in C++[R]. Livermore, CA: Lawrence Livermore National Lab, 2006

[41] Dietrich J, Gauthier F, Krishnan P. Driver generation for Java EE web applications[C]//Proc of the 25th Australasian Software Engineering Conf (ASWEC). Piscataway, NJ: IEEE, 2018: 121−125

[42] Arzt S, Rasthofer S, Fritz C, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps[J]. ACM SIGPLAN Notices, 2014, 49(6): 259−269

[43] Blackshear S, Gendreau A, Chang B Y E. Droidel: A general approach to Android framework modeling[C]//Proc of the 4th ACM SIGPLAN Int Workshop on State of the Art in Program Analysis. New York: ACM, 2015: 19−25

[44] Chen Miao, Tu Tengfei, Zhang Hua, et al. Jasmine: A static analysis framework for spring core technologies[C]//Proc of the 37th IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE, 2022: 1−13

[45] Laursen M R, Xu Wenyuan, Møller A. Reducing static analysis unsoundness with approximate interpretation[C]//Proc of the 45th ACM on Programming Languages(PLDI). New York: ACM, 2024: 1165−1188

[46] Hardekopf B, Lin C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code[C]//Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 290−299

[47] Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code[C]//Proc of the 9th Int Symp on Code Generation and Optimization (CGO 2011). Piscataway, NJ: IEEE, 2011: 289−298

[48] Yu Hongtao, Xue Jingling, Huo Wei, et al. Level by level: Making flow-and context-sensitive pointer analysis scalable for millions of lines of code[C]//Proc of the 8th annual IEEE/ACM Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2010: 218−229

[49] McPeak S, Gros C H, Ramanathan M K. Scalable and incremental software bug detection[C]//Proc of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). New York: ACM, 2013: 554−564

[50] Garbervetsky D, Zoppi E, Livshits B. Toward full elasticity in distributed static analysis: The case of callgraph analysis[C]//Proc of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). New York: ACM, 2017: 442−453

[51] Gens D, Schmitt S, Davi L, et al. K-Miner: Uncovering memory corruption in Linux[C]//Proc of the 25th Annual Network and Distributed System Security Symp (NDSS). Reston, VA: The Internet Society, 2018: 1−15

[52] Corteggiani N, Camurati G, Francillon A. Inception: System-wide security testing of real-world embedded systems software[C]//Proc of the 27th USENIX Security Symp (USENIX Security 18). Berkeley, CA: USENIX Association, 2018: 309−326

[53] Yang Yuting, Tu Jinjing, Shen Wenbo, et al. kCPA: Towards sensitive pointer full life cycle authentication for OS kernels[J]. IEEE Transactions on Dependable and Secure Computing, 2023, 21(4): 3768−3784

[54] Cifuentes C. Reverse Compilation Techniques[M]. Brisbane: Queensland University of Technology, 1994

[55] Cifuentes C, Gough K J. Decompilation of binary programs[J]. Software: Practice and Experience, 1995, 25(7): 811−829

[56] Cifuentes C. Structuring decompiled graphs[C]//Proc of the 6th Int Conf on Compiler Construction. Berlin: Springer, 1996: 91−105

[57] Hex-rays. Hex-rays decompiler[EB/OL]. [2024-10-16]. https://www.hex-rays.com/products/decompiler/

[58] Hex-rays. Ida pro [EB/OL]. [2024-10-16]. https://www.hex-rays.com/products/ida/

[59] Yakdan K, Eschweiler S, Gerhards-Padilla E, et al. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations[C]//Proc of the 22nd Annual Network and Distributed System Security Symp (NDSS). Reston, VA: The Internet Society, 2015: 1−15

[60] NationalSecurityAgency.Ghidra[EB/OL].[2024-10-16].https://github.com/NationalSecurityAgency/ghidra

[61] Toor T S. Decompilation of binaries into LLVM IR for automated analysis[D]. Waterloo: University of Waterloo, 2022

[62] Yakdan K, Dechand S, Gerhards-Padilla E, et al. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study[C]//Proc of the 37th IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2016: 158−177

[63] Schwartz E J, Lee J H, Woo M et al. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring[C]//Proc of the 22nd USENIX Security Symp (USENIX Security 13). Berkeley, CA: USENIX Association, 2013: 353−368

[64] Liu Zhibo, Wang Shuai. How far we have come: Testing decompilation correctness of C decompilers[C]//Proc of the 29th ACM SIGSOFT Int Symp on Software Testing and Analysis (ISSTA 2020). New York: ACM, 2020: 475−487

[65] lifting-bits. Remill [EB/OL]. [2024-10-16]. https://github.com/lifting-bits/remill

[66] lifting-bits.McSema[EB/OL].[2024-10-16].https://github.com/lifting-bits/mcsema

[67] cojocar. Bin2llvm [EB/OL]. [2024-10-16]. https://github.com/cojocar/bin2llvm

[68] Altinay A, Nash J, Kroes T, et al. BinRec: Dynamic binary lifting and recompilation[C]//Proc of the 5th European Conf on Computer Systems. New York: ACM, 2020: 1−16

[69] GaloisInc.Reopt[EB/OL].[2024-10-16].https://github.com/GaloisInc/reopt

[70] Engelke A, Schulz M. Instrew: Leveraging LLVM for high performance dynamic binary instrumentation[C]//Proc of the 16th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments. New York: ACM, 2020: 172−184

[71] Repzret. Dagger [EB/OL]. [2024-10-16]. https://github.com/repzret/dagger

[72] rev.ng Labs. Revng [EB/OL]. [2024-10-16]. https://github.com/revng/revng

[73] fay59. Fcd [EB/OL]. [2024-10-16]. https://github.com/fay59/fcd

[74] angr. angr [EB/OL]. [2024-10-16]. https://github.com/angr/angr



Lai Yuanming, born in 1991. PhD. His main research interests include binary translation, computer systems security, and code obfuscation.

赖远明, 1991 年生。博士。主要研究方向为二进制翻译、计算机系统安全、代码混淆。



Wang Zhe, born in 1990. PhD, associate professor, master supervisor. Member of CCF. His main research interests include computer systems security, operating systems, and computer architecture.

王　喆, 1990 年生。博士，副研究员，硕士生导师。CCF 会员。主要研究方向为计算机系统安全、操作系统、体系结构。



Wu Chenggang, born in 1969. PhD, professor, PhD supervisor. Distinguished member of CCF. His main research interests include binary translation, compiler optimization, and computer systems security. (wucg@ict. ac. cn)

武成岗, 1969 年生。博士，研究员，博士生导师。CCF 杰出会员。主要研究领域为二进制翻译、编译优化、计算机系统安全。



Yu Ding, born in 1996. Master. His main research interests include software security and compiler optimization.

于　丁, 1996 年生。硕士。主要研究方向为软件安全、编译优化。



Hou Chengxuan, born in 1998. PhD candidate. His main research interests include operating system security and vitrualization technology. (houchengxuan16@mails.ucas.ac.cn)

侯承轩, 1998 年生。博士研究生。主要研究方向为操作系统安全、虚拟化技术。