Artifact Evaluated NDSS Available Functional

深入探究函数内联及其对基于机器学习的二进制分析的安全影响

Omar Abusabha, Jiyong Uhm, Tamer Abuhmed, Hyungjoon Koo*成均馆大学，韩国 404970@g.skku.edu, jiyong423@g.skku.edu, tamer@skku.edu, kevin.koo@skku.edu

arXiv:2512.14045v1 [cs.CR] 16 Dec 2025

摘要——**函数内联优化是现代编译器广泛采用的一种变换，它会将调用点替换为所需的被调用者函数体。虽然这种变换能够提升性能，但也会显著影响机器指令和控制流图等静态特征，而这些特征对二进制分析至关重要。然而，尽管函数内联影响广泛，其安全影响迄今仍未得到充分研究。本文首次从基于机器学习的二进制分析视角，对函数内联进行了全面研究。为此，我们剖析了 LLVM 成本模型中的内联决策流程，并探索了能够大幅提高函数内联率、使其超出标准优化级别的编译器选项组合，我们将其称为**极端内联。**我们聚焦于五项面向安全的机器学习辅助二进制分析任务，并使用 20 个独特模型，系统评估这些模型在极端内联场景下的鲁棒性。大量实验揭示了若干重要发现：i）函数内联虽然本意是一种良性变换，却可能直接或间接影响机器学习模型的行为，并可能被利用来规避判别式或生成式机器学习模型；ii）依赖静态特征的机器学习模型可能对内联高度敏感；iii）细微的编译器设置可用于刻意构造规避型二进制变体；iv）不同应用程序和构建配置之间的内联率差异显著，这动摇了人们对机器学习模型训练与评估一致性的假设。**

I. 引言

如今，大多数应用程序都以可执行二进制文件的形式发布，其中包含低级机器指令和嵌入式数据。现代编译器通过复杂的编译流程生成这些二进制文件，该流程涉及广泛的优化。其中，函数内联是一种成熟的优化技术，它会用被调用者的函数体替换函数调用点。虽然这看起来可能只是简单的变换，但要确定最优的内联策略却颇具挑战性——其复杂度堪比 NP 完全背包问题 [94] 。

当源代码不可用时，通常会采用二进制逆向工程（即二进制逆向）来理解二进制文件的内部工作机制，揭示隐藏或未知的功能。二进制逆向可以通过静态方式 [105], [53], [2], [6] （即分析不执行二进制文件）或动态方式 [31], [90], [108]进行。由于二进制文件中的反汇编指令缺乏高层信息（例如函数名、变量名或类型），传统方法会提取各种特征，例如数值模式、控制流与数据流以及指令序列。这些特征在安全任务中发挥着关键作用，包括函数边界检测 [11], [84], 二进制代码相似性检测（BCSD） [28], [5], [85], [75], [104], [100], 漏洞检测 [38], [96],恶意软件分析 [6], 恶意软件家族分类 [60], [21], 以及崩溃根因分析 [82]。

近期的进展将机器学习（ML）（例如，深度学习）技术用于在高维特征空间中分析二进制文件。大量文献研究已证明这一新方向的有效性，能够通过推断函数名 [55], [41], [24], 变量名 [41], [15] 和类型 [41], [15], , 并恢复反编译代码 [36],[50],，为逆向工程师提供指导，此外还涵盖上述任务。

然而，函数内联作为编译器的一种常见优化，可能显著扭曲静态特征，从而影响逆向工程。内联会将被调用者的指令合并到调用者中，这可能大幅改变机器码和控制流结构。如果进行嵌套内联，或随后应用额外优化，这些影响会进一步加剧。因此，激进地（有意）应用内联时，静态特征可能不再适合作为机器学习模型决策的可靠依据。

尽管函数内联的重大影响早已广为人知，但现有文献对其对基于机器学习的模型的安全影响（例如，影响程度）仍缺乏深入研究。虽然已有一些研究考虑了函数内联 [43], [44], [14], [28], 但也有研究偶尔对其有所误述 [8], 低估了其影响 [53],[56], [64], [32],或仅以启发式方法加以处理 [14], [28], [107], [106], 或对其关注有限 [86], [32], [103], [33], [105], [46],[83], [49], [84], [11]。据我们所知，这是首项深入探讨以下方面的研究：1 在现代基于机器学习的模型背景下，揭示函数内联的安全影响；2 研究函数内联的整个决策流程，包括编译器成本模型的内部工作机制；3 探究一系列影响函数内联决策、且常被忽视的编译器选项；以及4 确定一组能够提高内联率的特定选项组合（例如，在我们的实验中，内联率最高可达 79.64%）。本研究以 LLVM 编译器工具链为基础，该工具链提供了模块化程度高的构建模块。我们利用调试信息和编译器生成的中间输出构建真实值。

* 通讯作者。

网络与分布式系统安全研讨会（NDSS） 2026年 2026年2月23日至27日，美国加利福尼亚州圣迭戈 ISBN
9
7
9-8-9919276-8-0https://dx.doi.org/10.14722/ndss.2026.241872www.ndss-symposium.org

LLVM 的内联过程本质上遵循一套结构化流程：前端（即Clang）和中端（即Opt）会对中间表示执行一系列分析和转换遍。CGSCC（调用图强连通分量）遍管理器在内部协调强连通分量（即相互可达的函数集合）中的内联及其他基于调用图的优化。作为 CGSCC 的一部分，函数内联遍使用成本模型评估每个函数候选项，将函数的内联成本与阈值进行比较。每次调用图更新后，都会迭代执行此遍。同时，许多编译器选项（表 II）也会影响这一决策。

在这项工作中，我们考察了将函数内联用作规避基于机器学习的安全模型手段的潜在滥用情形。我们确定了一种能够大幅提高内联率的特定编译器选项配置，并将这一策略称为极端内联。值得注意的是，我们展示了对手如何利用这一原本合法的编译器机制，通过极端内联进行蓄意变异，从而躲避判别式和生成式基于机器学习的安全模型。我们的威胁模型假设，标准构建流程被用于恶意目的，且编译器未被修改或混淆。此外，我们重新审视了不同的内联情景，并阐明了函数内联实践中的细微差别。

为验证我们的论点，我们从两个不同角度提出九个研究问题，并开展了大量实验。首先，我们通过训练 20 个不同的模型，探索函数内联如何影响基于机器学习的安全应用程序，涉及以下五项任务（章节VI-A）：1 二进制代码相似性检测，2 函数符号名称预测，3 恶意软件检测，4 恶意软件家族分类，以及5 漏洞检测。其次，我们从内联率及其进一步影响的角度评估函数内联优化本身，并提出以下四个问题（章节VI-B）：6不同应用程序中的函数内联率和优化级别，7 内联率与（前端）编译器选项之间的关系，8 内联率与（中端）编译器选项组合对极端内联的影响，以及9 二进制文件中的静态特征如何随（极端）内联而变化。

我们的主要发现表明：1 函数内联本身虽旨在实现良性优化，却可能被攻击者利用来规避判别式和生成式机器学习模型，尤其是那些依赖静态特征且对这类转换较为敏感的模型；2 影响决策流程的细微内联配置可以被蓄意操纵，从而轻松生成规避型二进制变体；以及3 内联率会因应用特定因素（例如，编程风格）和构建设置（例如，优化级别、编译器选项）而显著变化，为生成多样化变异提供了充足的灵活性。

以下概述了我们的贡献。

- 据我们所知，这是首个专门聚焦于函数内联的全面研究。我们深入探究 LLVM 编译器工具链中函数内联的复杂决策过程，剖析其内部启发式方法、编译器选项以及内联成本模型。

表 I：用于输出函数内联中间输出的特殊 LLVM 选项。

选项	描述
-Rpass=inline	显示内联信息（例如，成功、失败）
-Rpass-analysis=inline	显示内联分析（例如，成本、阈值）
-Rpass-missed=inline	显示未执行的内联（例如，启发式方法、成本过高）

- 我们阐明了函数内联实践中那些容易造成误解的细微之处，而这些问题此前一直被低估或忽视。
- 我们展示了通过不同的编译器选项组合实现极端内联的情况，以及它如何扭曲静态特征（例如，机器码、控制流图）。
- 我们通过研究（极端）内联如何影响机器学习辅助的二进制逆向任务，对其安全影响进行了广泛研究。

为促进未来研究，我们将数据集、分析和模型以开源形式发布 [1]。

II. 背景

LLVM 架构与中间输出。LLVM（低级虚拟机） [87] 是一组编译器工具链，提供用于前端和后端的模块化构建模块。其核心设计围绕中间表示（中间表示）展开，支持前端的多种编程语言、中端对中间表示的分析与变换，以及后端针对特定目标的代码生成。LLVM 提供了许多选项，可在变换过程中输出中间结果 [2]。在本文中，我们采用 LLVM 中的三个特殊选项（表 I）来收集有关函数内联的诊断报告。尽管输出可能并不完美 [17]，我们仍利用它们揭示成本模型如何通过复杂计算在内部判定一个函数是否可以内联（即成本、阈值）。最后，我们深入探究 LLVM 源代码，以分析其函数内联优化。

符号和链接类型。编译器维护着多种不同的链接类型，用于定义生成二进制文件时符号的可见性，并决定这些符号如何在不同模块间访问。这些类型对于处理符号可见性、代码优化和符号冲突至关重要。首先，具有外部链接的符号应在模块外可见，以便其他模块引用，包括未使用 `static` 关键字或使用 `extern` 声明的函数和变量。函数默认使用外部链接声明 [91]。其次，具有内部链接的符号仅在所属模块内可见，其他模块无法引用它（但可以避免名称冲突）。值得注意的是，静态函数一旦被内联，就会被视为未使用的全局符号（因此可以安全地移除 [23]）。第三，私有链接符号与内部链接符号类似，但有一项额外限制：即使经过链接时优化（LTO），它也无法对其他翻译单元可见。最后，弱链接允许不同模块中存在多个相同符号的定义，之后由链接器从中进行选择。

[1] https://doi.org/10.5281/zenodo.17759528 [2] GCC 提供 `-fdump-ipa-inline` 选项，用于确定单个源代码文件中的函数内联情况 [43]。

这种灵活性使得 C语言++ 中的内联函数或模板实例化需要存在多个符号。

DWARF 信息。 Debugging With Attributed Record Formats（简称 DWARF） [19] 定义了一种结构化数据格式，其中包含调试所需的重要信息，例如源代码与机器码之间的映射。调试信息条目（DIE）使人们能够创建源程序的底层表示，每个条目都包含一个标识标签和一系列属性。标签指定条目的类别，而属性则定义其具体特征。附录中的表 VIII 列出了 DWARF 中与函数内联行为相关的标签和属性。可在编译期间通过 `-g` 选项生成这些信息。由于 DWARF 提供了一种与编程语言及其底层结构正交的统一方法，因此已被广泛用于获取可执行二进制文件的真实值。同样，本研究也使用 DWARF v5 [19] 获取函数内联的真实值（章节 VII）。

函数内联的优缺点。 函数内联优化需要权衡利弊。内联的显著优势在于，通过消除额外的栈相关操作来降低函数调用的性能开销，例如，为传递参数而将值存入寄存器或从寄存器中取出，以及用于调整栈指针和基址指针的函数序言和尾声。此外，函数内联还可能提高指令缓存局部性（例如 在循环中内联函数）。除此之外，函数内联也为进一步优化提供了机会。与此同时，多次出现重复代码（即指令）不可避免地会增大可执行二进制文件的体积。另一个缺点是，由于寄存器的活跃时间更长且循环不变量增多，可能需要分配更多寄存器，从而增加寄存器分配压力。最后，函数内联过程会增加编译时间。

III. 动机与威胁模型

先前工作的函数内联。 在回顾先前工作时，我们发现，函数内联常常被错误描述 [8], 低估 [53], [56], [64], [32], 以启发式方法处理 [14], [28], [107], [106] 或未得到足够关注 [86], [32], [103], [33], [105], [46], [83], [49], [84], [11]。例如，Alves-Foss 等人 [8] 错误描述了内联的影响，声称“如果源代码中的函数被内联，它就不再是一个函数，分析工具不应声称自己在二进制文件中找到了该函数”；然而，内联函数仍可能出现在二进制文件中。Binkit [53]和 Discovre [32] 通过应用 `-fno-inline` 选项来防止内联，低估了内联的普遍性。此外，`always_inline` 指令可以覆盖这一设置。与此同时，Dispatch [56] 考察了这样一种情况：固件中的大型三角函数（例如，tan）为便于漏洞修补而避免函数内联。许多现有研究，包括关注漏洞发现、代码搜索和代码相似性检测等任务的研究 [86], [103], [33], [105], [46], [83], [49] ，都没有仔细考察函数内联的影响，因而未能考虑其对代码语义可能造成的影响。同样，XDA [84]和 BYTEWEIGHT [11]依赖函数序言和尾声，却没有深入讨论内联的影响。Dream(++) [107], [106] 通过构建签名数据库来识别（已知的）常见内联库函数，从而以启发式方法处理内联，例如 `strcpy`、`strlen`和 `strcmp`。另一方面，CI-Detector [44] 和 Jia 等人 [43] 强调了处理内联指令的重要性。Collberg 等人 [18] 将内联描述为一种有效且实用的混淆技术，因为它消除了程序中的过程抽象。与此同时，FUNCRE [4]能够在不同优化级别下以一致的方式处理内联库函数，但当多个连续的库函数被一并内联时会遇到挑战。本研究聚焦于内联的独特影响，例如它改变调用图、控制流图和函数边界的能力。

威胁模型与假设。我们的威胁模型假设，良性的标准构建流程可能被用于恶意目的。在这种情况下，攻击者无需依赖混淆来逃避检测；他们只需在不同的优化策略或构建配置下重新编译相同的代码即可。我们关注的是在不修改编译器内部逻辑的情况下，对内联进行现实的标志级控制（例如，优化级别和内联器参数）。值得注意的是，这一假设不同于传统混淆。传统混淆涉及刻意采用非标准变换，会带来不同的权衡和检测面，并可能导致性能或合规性问题。

IV. 函数内联的决策流程

图1展示了函数内联的完整决策流程，重点介绍 LLVM [87] 编译器基础设施。

A. 外部内联因素

1) 源代码中关于内联的（直接）提示：C 和 C++编程语言允许程序员使用指令或属性，直接或间接地提示函数内联。作为 C 和 C++ 标准的一部分，`inline` 说明符可在函数声明处提示内联该函数 [3]。此外，不同的编译器还提供了与内联相关的其他属性和编译指示。Clang 和 GCC 允许通过特定函数属性__属性__((always_inline))显式要求编译器内联函数，而不受优化级别影响（也就是说，`-O0` 也不例外）。反之，__属性__((noinline)) 会阻止编译器内联该函数 [4]。同样，`#pragma inline` 和`#pragma noinline` 指令也用于影响函数内联。请注意，上述编译器专用指令并非标准的一部分，具体情况可能因实现而异。与上述直接提示类似，其他属性也能间接影响函数内联。__属性__((flatten))属性允许内联函数中的所有被调用函数，就好像每个调用点都带有__属性__((always_inline))一样。同时，__属性__((naked))属性通常用于汇编函数，会间接导致编译器避免函数内联。同样，

[3] __inline说明符在 C99 标准之前也具有相同的作用；不过，它仍与现代编译器兼容。[4]同样，MicrosoftVisualC++ 编译器支持__forceinline 和 __declspec(noinline) 关键字，分别用于强制和阻止函数内联。

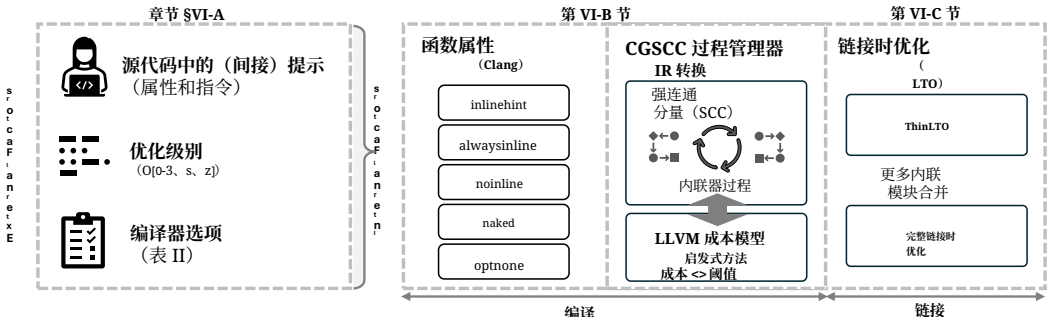


图 1：LLVM 工具链中整个函数内联流水线的概览 [87]。我们通过将影响内联的因素划分为外部因素和内部因素，简化复杂决策过程中的流水线。在源代码中使用不同的指令和属性（例如，`inline` 关键字）、优化级别和编译选项，都允许程序员影响二进制文件中的内联率（章节 IV-A）。在编译时，Clang 会根据这些因素定义函数的内部属性。Opt 会在中间表示（IR）的变换过程中，跨强连通分量（SCC）迭代执行函数内联（章节 IV-B）。请注意，该过程很大程度上依赖启发式方法和内置成本模型（例如，检查成本是否超过阈值）。如果启用了链接时优化（LTO），则可以通过合并模块执行额外的函数内联（章节 IV-C）。

表 II：影响前端和中端函数内联成本模型的全面编译器选项。

选项名称	Tool	默认值	描述
-finline-functions	Clang	已禁用	根据优化级别内联合适的函数
-finline-hint-functions	Clang	已禁用	内联显式或隐式标记为内联的函数
-fno-inline-functions	Clang	已禁用	除非函数声明为always_内联，否则禁用函数内联
-fno-inline	Clang	已禁用	禁用所有函数内联，但always_内联
-inlinedefault-threshold	Opt	225	仅为 O1 和 O2 设置初始阈值
-inline-threshold	Opt	225	为所有优化级别设置初始阈值
-inlinehint-threshold	Opt	335	设置带有`inlinehint` 属性的函数的阈值
-inline-cold-callsite-threshold	Opt	45	设置冷调用点的阈值
-inline-savings-multiplier	Opt	8	设置内联期间周期节省的乘数
-inline-size-allowance	Opt	100	设置周期节省不足时进行内联的大小限额
-inlinecold-threshold	Opt	45	设置冷调用点的阈值
-hot-callsite-threshold	Opt	3,000	设置热调用点的阈值
-locally-hot-callsite-threshold	Opt	525	设置局部作用域内热调用点的阈值
-cold-callsite-rel-freq	Opt	2	设置调用点被视为冷调用点时允许的最大基本块频率（无性能分析信息）
-hot-callsite-rel-freq	Opt	60	设置调用点被视为热调用点时要求的最低基本块频率（无性能分析信息）
-inline-call-penalty	Opt	25	为每次函数调用设置调用惩罚
-inline-enable-cost-benefit-analysis	Opt	false	启用内联器的成本效益分析
-inline-cost-full	Opt	false	启用此选项后，当成本超过阈值时，会计算完整的内联成本
-inline-caller-superset-nobuiltin	Opt	true	启用此选项后，当调用者具有被调用者的超集`nobuiltin` 属性时，允许内联
-disable-gep-const-evaluation	Opt	false	禁用对带有常量操作数的`GetElementPtr` 的评估

__属性__((optnone))属性会禁用函数的所有优化以及内联。

*2)*优化级别：编译器优化级别会显著影响中间表示和机器代码生成，需要在性能、编译时间与代码大小之间取得平衡。`-O[0-3]`通常表示优化级别（即，无、基础、中等或最高），而`-O[s,z]` 则调整代码大小（即，较小或最小）。优化策略不可避免地会影响函数内联：例如，`-O3` 会进行积极的内联以提升执行速度，`-Os`会进行适度内联，以平衡性能与代码大小，而`-Oz` 则力求将代码大小降至最低（例如，比其他级别更少地进行内联）。

3）编译器选项：LLVM [87] 提供了多种选项，用于配置所有函数内联决策的全局设置。表 II 总结了有助于精细控制内联的综合性（与样式无关）选项

LLVM 中函数内联行为的控制方式[5]。例如，`-finline-functions` 允许编译器决定是否内联某个函数。或者，`-finline-hint-functions` 指示编译器仅考虑带有特定关键字或属性的函数进行内联。相比之下，Clang 还提供选项，可使用编译器启发式方法禁用内联：`-fno-inline-functions`，或完全禁用内联，但标记为`alwaysinline` 的函数除外（章节 IV-B）。

B. 编译期间的内部内联因素

*1)*函数属性：根据提示、优化级别和编译器选项等外部因素，Clang 会在内部将函数的属性标记为以下

[5]可通过`--help-hidden`查看相关选项。适用于 LLVM 优化器（opt）的函数内联专用选项，可在 Clang 中通过`-mllvm [flagname=value]` 传入。请注意，我们排除了依赖特定风格或语言的选项（例如，GNU89、汇编语言）。

表 III：根据 LLVM 内置启发式规则，选择性调用点永不内联函数的情形。

范围	情形描述
基本块	在特定指令之外误用块地址（例如，`callbr`）
调用者	`optnone` 会禁用所有优化（包括内联）的属性 `noduplicate`用于避免出现多个实例的属性
调用者 and 被调用者	被调用者和调用者均带有冲突的属性（例如，`alwaysinline` 和 `noinline` 同时使用） 调用者与被调用者之间不兼容的空指针定义
被调用者	`noinline` 属性 可插入函数，可由另一个函数替换或覆盖，THOOK 链接时或运行时 一种无法拆分为多个部分以实现挂起的不可拆分的协程调用以及恢复
调用点	`noinline` 属性 内存地址空间不正确的按值传递 参数（即，不在该`alloca` 地址空间中） 无法确定被调用函数的间接调用， 编译时
指令	编译时未知的内存量（例如，动态分配） 具有多个返回路径的函数（返回两次） 使用可变参数初始化的函数 目标在运行时确定的间接分支 过于复杂而无法内联的内建函数（例如，icall_branch_funnel 或 `localescape`）。 可能导致无限循环或分配过多栈空间的递归调用

几种之一：`inlinehint`、`alwaysinline`、`noinline`、`naked`和`optnone`。这些属性会在 IR 生成的早期阶段（从抽象语法树生成）设置，并在后续的优化过程中遵循自定义指令。例如，尽管 `-O0` 会禁用大多数优化以提高编译速度并便于调试，但它仍会执行最少且必要的转换（例如，`alwaysinline`），以确保代码正确执行。最后需要注意的是，编译器生成的函数（例如内建函数）通常会标记为`alwaysinline`，以确保效率。

2) LLVM 内联器过程：LLVM 提供了内联器过程，它能够分析和变换中间表示，并在内部维护一个遍管理器，以特定顺序应用一系列中间表示转换遍 [69]。LLVM 的函数内联器 pass 负责决定是否进行函数内联，这与 CGSCC 过程管理器紧密耦合 [66]，该管理器作用于调用图中的一个强连通分量（SCC）。该 SCC 由调用者和被调用者组成，其中每个函数都通过一系列转换遍进行简化，例如
如控制流图简化（SimplifyCFG pass [70]）、聚合体标量替换（SROApass [71]）以及早期公共子表达式消除（EarlyCSE pass [67]）。这些预备步骤可确保在进行函数内联之前，非平凡的强连通分量得到充分优化。最后，CGSCC 根据成本模型，按照自底向上顺序，在各强连通分量中迭代调用函数内联器 pass（参见章节 IV-B3）。

3）成本模型：LLVM 中的成本模型首先会处理预先定义的不得内联情形（作为启发式方法）。表 III 列出了这些情形及其适用范围。模型确认某个情形适合函数内联后，函数内联器 pass 会评估一个调用点（即一对调用者和被调用者），并根据上述因素动态计算阈值和成本。简而言之，当成本低于阈值时，内联器 pass 会在强连通分量中机械地执行内联变换。

阈值计算。 在每个调用点，阈值都会根据优化级别进行初始化（例如，`-Oz` → 5、`-Os` → 50、`-O1` → 225、`-O2` → 225、`-O3` → 250）。此外，CGSCC 过程管理器还会利用调用点上的配置文件摘要信息（PSI；例如，函数的热点程度或冷点程度、执行时间）和基本块频率信息（BFI；例如，单个基本块的执行频率）。随后，内联器过程会根据1 调用者和/或被调用者的属性、2 代码区域的性质，以及3 复杂分支或向量化指令等其他结构（表 IV）来提高或降低阈值。例如，通过`inline`说明符指定的`inlinehint` 属性可以提高阈值。

表 IV：影响 LLVM 成本模型中函数内联成本和阈值的条件。

目标	影响	条件
Cost	↑	缺失的指令简化（例如，switch 指令、循环、加载指令） 存在内建函数 被调用者内部未优化的调用点 无法简化的复杂内存操作 冷调用约定（`coldcc`） 一般函数调用开销（调用惩罚）
	↓	静态函数的最后一次调用 带有值的调用点参数（例如按值传递） 成功的转换（例如，间接调用转为直接调用）
阈值	↑	存在`inlinehint` 属性 热点区域 基于目标特定架构的调整
	↓	存在`minsize` 或 `optsize` 属性 冷点区域 复杂分支 存在向量化指令

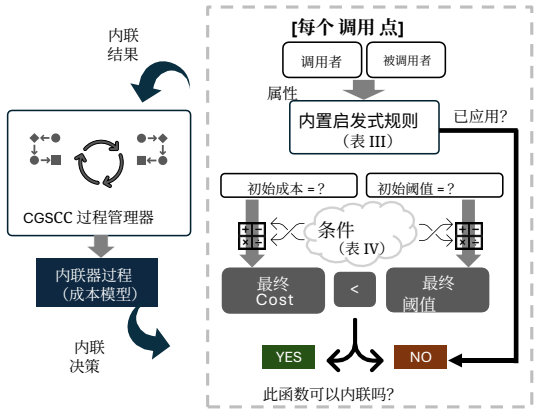


图 2：LLVM 中用于确定要内联的函数的成本模型概览。尽管 CGSCC 过程管理器在强连通分量上运行，但它会在需要时调用内联器过程。成本模型首先考虑每个调用点的启发式方法（表 III），以检查其是否永远无法内联。如果可以内联，则初始化成本和阈值，随后根据动态条件（表IV）进行更新。该过程根据成本以机械方式确定函数内联。然后，CGSCC 基于内联迭代执行进一步的优化。

成本计算。默认情况下，acost 初始化为零。特殊情况是，当被调用者是静态函数且调用点是最后一个调用执行时，初始成本会设为

一个负值 −15，000。这会大幅提高函数内联的可能性。如表 IV所总结的，各种条件都可能调整成本。本质上，内联器过程会遍历每个基本块及其中的每条指令，通过加上或减去指定值来得出最终成本。值得注意的是，可以通过编译器选项配置 acost 或阈值（表 II）。一方面，在缺失的指令简化、存在内建函数、调用点已优化、内存操作复杂或调用约定较冷的情况下，成本可能会受到惩罚（即，增加）。另一方面，当调用点参数具有值，或发生从间接调用到直接调用的变换时，成本可能会降低。LTO 会通过内部化 过程将每个符号提升为内部链接，以尝试将其消除。不过，如果 LTO 下的函数未能在所有调用点内联，它仍可能保留下来。

C. 链接时的内部内联因素

*1)*链接时优化与内联：启用 LTO [39] 实际上会在链接时从整个程序的角度为链接器提供全局优化机会，而这些机会在单个编译单元中无法实现。LLVM 中主要有两种 LTO [6]：ThinLTO [47] 和完整链接时优化 [68]，，二者对函数内联的影响各不相同。

使用完整链接时优化进行内联。完整链接时优化 [68] 通过使所有函数在全局范围内可见，实现跨整个程序的积极内联，并支持常量传播和死代码消除等进一步优化。然而，这会带来显著的资源开销（例如，内存使用量），并延长编译时间。

使用 ThinLTO 进行内联。ThinLTO [47] 旨在降低计算开销，同时兼顾完整链接时优化的优势。从本质上说，ThinLTO 会为每个模块中的函数生成摘要，从而在链接阶段结合增量跨模块信息执行额外的函数内联。

V. 函数内联实践中的微妙之处

在本章节中，我们将澄清六种关于函数内联的常见误解，这些说法在实践中都不成立。随后，我们将介绍极端内联策略。

A. 误导性案例研究

使用 `-O0` 或`-fno-inline` 选项时，函数不会被内联。有时需要生成完全不经过优化（包括内联）的二进制文件，从而保留函数及其内部结构，以便于调试（例如，追踪执行流程、定位错误 [10]）。虽然编译器会遵从请求，抑制大多数优化，但带有 `always_inline` 指令的函数仍是例外，不受给定优化级别或编译器选项的影响。简而言之，`-O0` 和 `-fno-inline` 选项都无法覆盖上述指令，但通常会阻止内联，如图 8和图 9所示。

[6]要在 LLVM 中启用链接时优化，我们使用 LLVM 链接器（即 -fuse-ld=lld）选项 -flto。

`always_inline` 指令会始终将函数内联。任何函数指令（包括 `inline` 和`always_inline`）都不一定能确保函数被内联，因为编译器会在整个决策流程中进行最终决策（图 1），并使用复杂的成本模型评估调用点（图 2）。在许多情况下，无论优化级别、指令提示或编译器选项如何，编译器都不会考虑函数内联，包括递归调用、间接调用以及带有可变参数的函数（表 III）。

没有任何与内联相关指令的函数不会被内联。在实践中，LLVM 会在编译期间将每个函数都视为内联候选对象，并综合考虑函数大小、复杂度、优化级别和编译器选项。从本质上说，函数内联是一种选择退出操作：除非函数带有明确的标记`no_inline` 属性图 7 和图 6 展示了此类示例。

内联函数会在二进制文件中消失。一种常见的误解是，内联函数已被消除（因此完全消失），因为调用点已被函数替代。答案在一定程度上是肯定的；不过，如果函数需要在全局范围内调用，那么内联函数仍可能出现在最终的二进制文件中（并带有符号）。图 7 和图 6展示了此类情况。

函数内联的冲突条件可能导致不可预测的行为。例如，可以提出这样一个假设性问题：1 如果两个函数的条件相互矛盾（例如，一个标记为`inline`，另一个标记为`noinline`），会怎样？或者 2 编译带有`inlinehint`属性的函数时，如果指定了冲突的选项（例如，`-fno-inline`），又会怎样？为解决这些问题，编译器制定了若干预定义规则。首先，属性（指令）的优先级高于编译器选项。其次，编译器按以下顺序对各指令赋予优先级：`optnone`、`noinline`、`minsize`、`optsize`、`inlinehint`、`always_inline`。例如，当函数声明为`inlinehint`或`always_inline`时，`optnone` 具有更高优先级，实际上会使其他属性失效。同样，存在`noinline`时，其优先级高于`alwaysinline`。

库中的函数不会被内联。 通常会在运行时解析库中函数的地址，并在动态加载器的协助下，通过过程链接表（PLT）和全局偏移表（GOT）在内部跟踪该函数。不过，在少数情况下，库函数也可能成为内联候选项。首先，预处理器通过处理各种指令（例如，宏定义、文件包含、条件语句）来准备源代码，这可能导致头文件中定义的函数在该头文件被多个翻译单元包含时被内联。其次，类似地，如果纯头文件库 [59], [52] 在头文件中提供了完整可见的函数定义，编译器也可能内联函数。第三，LLVM 定义了一系列作用于中间表示层的特殊内建函数 [88]（例如，`llvm.memcpy`、`llvm.memset`），它们会作为

带有 `always_inline` 属性 [7]的内联候选项。

B. 内联但仍保留的函数

具有内部链接（例如，`static`）的函数符号，在所有调用点都被内联后便会消失。然而，即使函数具有内部链接，只要有任何调用点仍需要调用该函数，该函数就可能在内联后继续保留（因为其他地方仍在使用它）。相反，具有外部链接（例如，`extern`）的函数默认情况下即使被内联后也会保留。

C. 极端内联

为区别于高优化级别带来的激进内联，我们引入了极端内联这一概念，旨在使函数内联率超越编译器的标准行为。第VI-B3 节介绍了我们如何利用决策流程来制定极端内联策略，其中包括优化级别、编译选项和链接时优化。

VI. 函数内联与安全影响

我们的实验在一台 64 位 Ubuntu 20.04 系统上进行，该系统配备 Intel(R) Xeon(R) Gold 5218R CPU 3.00GHz、512GB 内存和两块 RTX A6000 GPU。

良性软件语料库。表 V 的前半部分总结了用于评估函数内联对基于机器学习的模型影响的良性程序语料库。由于构建限制，各个软件包和应用程序的编译选项可能略有不同（例如，`-Oz`、`-O3`），内联阈值也可能有所不同（例如，2、225、200、000）。如果 LTO（无论是 Thin 还是 Full）失败，我们会禁用 LTO，但保留原有的内联阈值和优化级别。为促成极端内联行为，我们通过实证调优相关编译设置。为尽量减少评估偏差，我们按符号名称去重相同的函数，并确保测试样本不会出现在训练集中。最终的良性应用程序数据集包含 1、524 种程序变体，其中包括额外 398 个经过极端内联处理的样本。

恶意软件语料库。表 V 的后半部分总结了我们的恶意软件语料库。鉴于物联网恶意软件的普遍存在、共享代码库 [21], 以及泄露源代码的可获得性（例如，Mirai [9] 和 Gafgyt [78]），我们重点关注物联网恶意软件。为进行恶意软件检测，我们从 VirusShare 构建了一个经过筛选的数据集，其中包含761 个恶意软件样本 [77], ，包括 Mirai、Gafgyt 和 Tsunami；这些样本是从更广泛的集合中移除损坏的二进制文件和打包的可执行文件后得到的。同样地，为进行恶意软件分类，我们准备了来自 Alrawi的 12,727 个样本，涵盖 10 个家族等人 [7]: Mirai、Gafgyt、Tsunami、Lotoor、Dofloo、DDoSTF、ExploitScan、Dvmap、Gluper 和 Healerbot。为评估函数内联的影响（即，规避检测），我们对开源的 Mirai 和 Gafgyt 进行重新编译（每个100 个样本），通过应用极端内联和自定义编译器选项生成样本。最终的恶意软件数据集共包含13,688 个样本。对于恶意软件相关任务，我们移除重复样本，即那些在 62 维统计特征表示上完全相同的样本，以避免模型过拟合。最后需要说明的是，附录中的表 XI 展示了我们实验中极端内联的完整操作流程。

[7]LLVM 中的内建函数很有益处，既能简化代码并实现平台独立性，也能充分利用目标架构的特征。

表 V：用于评估函数内联对基于机器学习的模型影响的二进制语料库。我们选择了各种安全任务：（T1）二进制代码相似性检测、（T2）函数名称预测、（T3）恶意软件检测、（T4）恶意软件家族预测，以及（T5）漏洞检测。我们采用两种开源恶意软件进行人工内联。请注意，我们为 T4(*) 生成 32 位 ARM 二进制文件，而其他任务生成的是 64 位 Intel x86 二进制文件。

Type	软件包或应用程序	版本	基线 二进制文件	T1	T2	T3	T4*	T5
良性	coreutils [20]	9.3	106	742	742	624	-	-
	binutils [13]	2.40	22	118	118	78	-	-
	diffutils [26]	3.8	4	28	28	24	-	-
	查找工具集 [35]	4.9	4	28	28	24	-	-
	OpenSSL [81]	3.1.4	1	7	7	6	-	-
		1.0.2d	1	-	-	-	-	6
	lvm2 [76]	2.03.21	52	364	364	-	-	-
	GNU科学图书馆 [40]	2.7.0	2	14	14	9	-	-
	Valgrind [99]	3.21.0	4	28	28	156	-	-
	Open MPI [79]	4.1.5	7	49	49	36	-	-
	PuTTY [89]	0.79	6	42	42	36	-	-
	nginx [80]	1.21.6	1	7	7	6	-	-
	lighttpd [62]	2.0.0	2	14	14	10	-	-
	SPEC2006 [97]	-	15	-	-	83	-	-
恶意软件	Mirai [37]	-	2	-	-	261	9,215	-
	Gafgyt [27]	-	1	-	-	421	3,119	-
	Tsunami	-	-	-	-	79	228	-
	Lotoor	-	-	-	-	-	80	-
	Dofloo	-	-	-	-	-	40	-
	DDoSTF	-	-	-	-	-	15	-
	ExploitScan	-	-	-	-	-	11	-
	Dvmap	-	-	-	-	-	8	-
	Gluper	-	-	-	-	-	6	-
	Healerbot	-	-	-	-	-	5	-

A. 函数内联对机器学习模型的安全影响

研究问题。我们定义了五个研究问题，从安全角度评估函数内联对基于机器学习的模型有何影响。我们选取了五项基于机器学习的安全任务，包括二进制代码相似性检测（T1）、函数名称预测（T2）、恶意软件检测（T3）、恶意软件家族分类（T4）和漏洞检测（T5）。

- **研究问题 1**：函数内联如何影响基于机器学习的二进制代码相似性检测模型（章节 VI-A1）？
- **RQ2**：函数内联如何影响基于机器学习的函数符号名称预测模型（章节 VI-A2）？
- **RQ3**：函数内联如何影响基于机器学习的恶意软件检测模型（章节 VI-A3）？
- **RQ4**：函数内联如何影响基于机器学习的恶意软件家族分类模型（章节 VI-A4）？
- **RQ5**：函数内联如何影响基于机器学习的漏洞检测模型（章节 VI-A5）？

安全任务的机器学习模型选择。我们选择了一组涵盖判别式和生成式任务的多样化基于机器学习的模型，涉及多种架构（例如，基于嵌入的模型、图神经网络、Transformer、循环神经网络）和输入模态（例如，统计特征、动态轨迹）。请注意，对于 T1 和 T5，我们使用其原始预训练形式。同时，我们针对 T2 重新训练了 AsmDepictor 模型，使其适配基于 Clang 的语料库，因为原始模型是在 GCC 二进制文件上训练的。同样，我们也在自己的数据集上重新训练了 SymLM，使感知执行的嵌入能够反映我们的编译设置。对于 T3 和 T4，我们根据原始论文中的描述重新实现了模型架构 [2], ，并在自己的语料库上重新训练了这些模型。

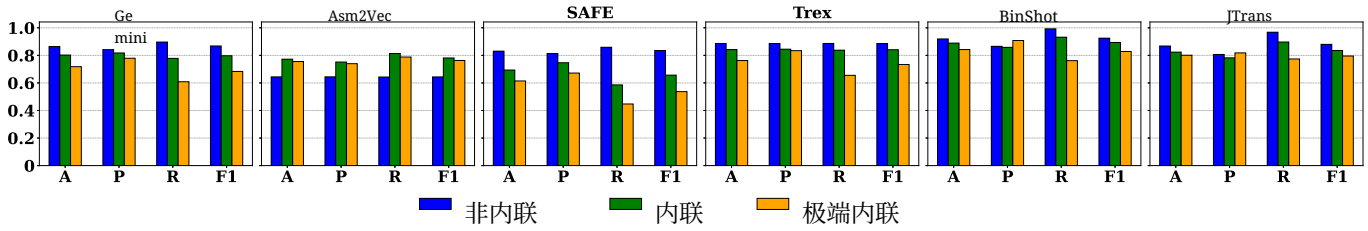


图 3：六个二进制代码相似性检测模型在三个数据集上的实验结果：非内联、内联和极端内联函数对样本。我们精心构建了一个包含非内联、内联和极端内联情况的数据集，以研究内联的有效性。总体而言，内联样本的性能逐渐下降。我们将在章节 VI-A1 中讨论 Asm2Vec 的例外情况。一般而言，与精确率相比，其他模型的召回率显著下降（即，假阴性增加）。A、P、R 和 F1 分别表示准确率、精确率、召回率和 F1 分数。

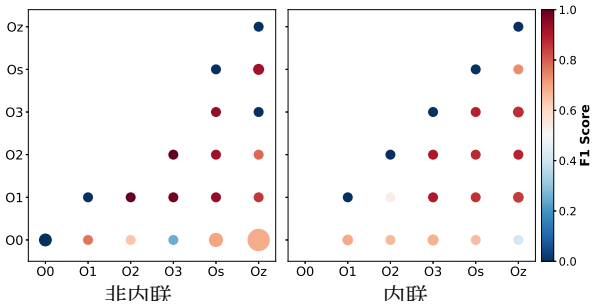


图 4：Asm2Vec 在非内联（左）和内联（右）函数对上的性能。圆的面积表示函数对的数量，颜色深浅表示其 F1 分数。分解结果显示，大多数性能下降源于涉及 `-O0` 与其他优化级别的非内联函数对，这表明 `-O0` 下的随机游走序列与更高优化级别下的序列存在显著差异。空白单元格对应于样本过少的组合。

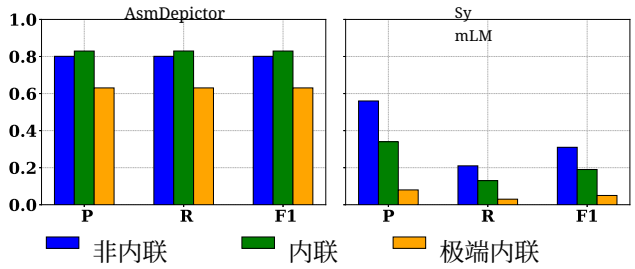


图 5：AsmDepictor [55] 和 SymLM [46] 在三个数据集上的实验结果：非内联、内联和极端内联（章节 VI-A2）。AsmDepictor 在内联情况下的性能略有提升，表明它能够利用额外的上下文信息进行推理。SymLM 在内联情况下的性能则有所下降。同时，极端内联情况中的未见模式会误导这两个模型，导致性能显著下降（例如，AsmDepictor 下降 23.99%，SymLM 下降 75.69%）。

1)（研究问题 *1*）内联对二进制代码相似性检测模型的影响：二进制代码相似性检测任务旨在确定给定一对代码片段之间的代码相似性。为考察函数内联对性能的影响，我们准备了三种不同的函数对：由两个非内联函数组成的非内联样本、由一个非内联函数和一个内联函数组成的内联样本，以及由一个非内联函数和一个采用我们极端内联策略的内联函数组成的极端内联样本。我们使用在每种优化设置下均出现的函数，以突出内联对各种二进制代码相似性检测模型的影响。我们选取六种具有代表性的二进制代码相似性检测模型进行评估：Asm2Vec [28],BinShot [5], Gemini [104], Trex [85], JTrans [100], 和 SAFE [75]。最后，我们分别从 BinShot 14,984、95,774和 23,378、JTrans [5], 和二进制代码相似性检测基准 [100], 中分别准备了函数对。 [74], 。

结果。 图 3 展示了六种二进制代码相似性检测模型在非内联、内联和极端内联两两组合下的性能比较。我们采用了二进制代码相似性检测基准 [74] 其中包括 Asm2Vec [28], Gemini [104], Trex [85], 和 SAFE [75],，用于提供同一平台中样本对之间的相似度分数（S）。我们以阈值 $T = 0.5$（例如，如果相似则为 $S > 0.5$）来衡量准确率、精确率、召回率和 F1分数，并与 BinShot [5]。总体而言，对于内联样本，观察到性能中度下降（F1分数平均下降 3.6%；召回率平均下降 6.2%），而显著下降（F1分数平均下降 12.6%；召回率平均下降 21.2%）则出现在极端内联样本中。尤其是，我们观察到召回率（假阴性）明显降低，而精确率（假阳性）并未明显降低。这表明，在评估内联函数时，相似的函数对可能显得并不相似；不过，也可以认为，额外的语义信息可能有助于模型更好地识别函数之间的相似性。然而，正如章节 IV 中所述，函数内联会带来进一步优化的机会，可能在评估过程中掩盖原本相似的函数对之间的相似性。值得注意的是，Asm2Vec 与其他模型不同：即，非内联函数对样本的性能低于内联样本。我们的调查发现，函数对之间的性能差异源自通过随机游走进行静态追踪（即潜在执行路径）以推断代码相似性的特性，如图 4 所示。我们发现，`-O0`与`-Oz` 之间的函数对几乎占总数的一半（23,378 对中有 10,614 对），导致整体性能下降。我们认为，这一现象是由于在`-O0` 下生成的随机游走序列与更高优化级别下生成的序列存在差异。

2)(RQ2) 内联对函数名称预测模型的影响：函数名称预测任务旨在根据一段汇编语言代码推断函数原始的符号名称

。与二进制代码相似性检测实验类似，我们准备了用于函数名推断的数据集（表 V）。我们选择了两个具有代表性的模型：AsmDepictor [55]（生成式）和 SymLM [46]（判别式）。我们使用精确率（P）、召回率（R）和 F1分数进行评估。

结果。图 5 展示了 AsmDe- pictor [55]和 SymLM [46]在我们的测试数据集（例如，7,863 个案例）上的实证结果。AsmDepictor 在内联后性能有所提升，并有效利用了额外的上下文信息。好奇的读者可能会将 AsmDepictor 内联性能的提升归因于训练数据分布：我们的训练数据集中有 79% 由非内联函数构成。然而，在极端内联情况下，AsmDepictor 和 SymLM 的性能都显著下降：例如，分别下降 24.0% 和 75.7%。作为生成模型，AsmDepictor 在生成函数符号名时难以应对未见过的实例，这可能是由异常的内联模式导致的。SymLM 表现不佳，可能源于其将函数名推断视为多类别多标签分类任务的做法。由于原始研究使用了四种架构，并在训练时采用了混淆技术，我们的发现表明，要达到相当的性能，需要的数据集规模远大于我们评估中所用的数据集。

3)（*RQ3*）内联对恶意软件检测模型的影响：恶意软件检测任务旨在区分恶意软件样本与良性样本。我们训练了四种传统机器学习模型：作为线性模型的逻辑回归、作为集成方法的随机森林、作为梯度提升方法的 CatBoost，以及作为基于实例的学习方法的 K近邻算法（KNN）；还训练了两种深度学习模型：Abusnaina 等人 [2]提出的深度神经网络（DNN）和卷积神经网络（CNN）。我们使用 TikNib [53]提取了 62 个统计语义特征。随后，使用蒙特卡洛交叉验证对每个模型评估 10 次。

结果。表 VI 展示了恶意软件检测的实验结果。尽管原始模型取得了较高的准确率，但在使用经过极端内联处理的内部恶意软件样本进行测试时，其性能有所下降（例如，约 20%）。这一性能下降表明，内联会显著改变代码结构，进而导致错误分类。

4)(RQ4) 内联对恶意软件家族预测模型的影响：恶意软件家族预测任务旨在预测一个未见过的恶意软件变种属于某个已知家族的可能性。与恶意软件检测类似，我们采用相同的以机器学习为中心的方法，但排除良性应用程序。需要指出的是，这些恶意软件针对的是物联网设备中普遍使用的 32 位 ARM 架构。为应对类别不平衡，我们通过过采样和欠采样，选择每个家族 500 个样本 [95]。

结果。表 VI 展示了恶意软件家族预测的实验结果（例如，在恶意软件检测实验中使用相同的语义特征时，所有机器学习模型在社区收集的物联网恶意软件上都表现不错（例如，约为 87%）。然而，它们的性能显著下降：即40% ↓ 在使用我们自有的具有极端内联的恶意软件进行测试时，大多数恶意软件家族预测模型的表现都降至这一水平。我们的研究结果表明，所有模型都容易受到我们的极端内联影响。我们推测，这种敏感性源于物联网恶意软件家族之间的代码共享 [21]，内联会模糊对细粒度分类至关重要的结构模式。相比之下，恶意软件检测中更广泛的语义差距似乎受到的影响较小。

5) (RQ5) 内联对漏洞检测模型的影响：漏洞检测任务用于识别是否存在已知漏洞（二分类）。排除基于源代码运行的漏洞检测模型 [42]，[63]，[61]，[58]，[102]，[101]，后，我们选择了两个基于二进制的检测模型 [73]，[74]，它们利用二进制代码相似性检测（BCSD）能力执行代码搜索（即探查存在漏洞的函数）。我们从两个固件映像中的 libcrypto 库里选取了 12 个存在漏洞的函数 [16] ——其中四个来自 Netgear R7000，八个来自 TP-Link Deco M4 [74]（附录中的表 IX ）。据此，我们在 OpenSSL-1.0.2d 中编译了两个包含上述漏洞的库版本 [81]：一个启用内联，另一个不启用。每个版本分别在六个优化级别（`-O[0-3,s,z]`）下编译，最终分别得到 12,958 个和 13,298 个函数。我们根据这些函数与每个已知存在漏洞函数的相似度进行排序，然后计算平均倒数排名（MRR）。我们采用 MRR@100，对相似度最高的前 100 个函数进行评估。

结果。 表 VII展示了内联前后的 MRR@100 分数，表明识别漏洞的能力受到负面影响（即，分数降低），平均下降了 69%。值得注意的是，Trex 模型在 Netgear R7000 上是个例外 [85]（$0.550 \rightarrow 0.625$）。不过，进一步分析发现，除排名最高的函数外，其他相似函数的排名均有所下降。

B. 函数内联评估

研究问题。 本章节从内联率和静态特征的角度，探讨函数内联变换本身的影响，并提出以下研究问题。

- **RQ6**：优化级别如何影响不同应用程序中的函数内联率（章节 VI-B1）？
- **RQ7**：默认编译器选项（由 Clang 提供）如何影响函数内联率（章节 VI-B2）？
- **RQ8**：在实践中，哪种编译器选项组合（由 Opt 提供）能提高函数内联率，使其趋向极端内联（章节 VI-B3）？
- **RQ9**：在多大程度上，可执行文件中的静态特征会受到函数内联的影响，例如指令、控制流图和调用图（章节 VI-B4）？

1)(RQ6) 函数内联率：图 6 展示了采用我们的极端内联策略（章节 VI-B3）编译的 coreutils 二进制文件中，内联函数的整体变化情况。从全部 2，070 个函数开始（假设在 `-O0` 下仅进行少量内联），我们观察到超过一半的函数被内联（1，183 个）。在这些内联函数中，997 个被移除，其余则保留。采用极端内联策略（比 `-O3` 更激进）后，约三分之二的函数被移除。

不同优化级别和应用中的函数内联。图 7 展示了基线数据集中八个软件包在五个优化级别下的函数内联率

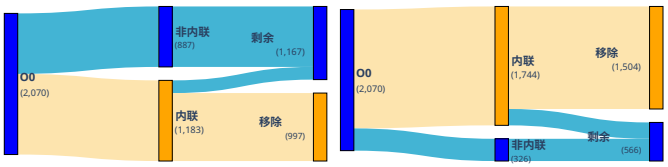
表 VI：Abusnaina 提出的四种传统机器学习模型（例如，逻辑回归、随机森林、CatBoost 和 K 近邻算法（KNN））以及两种基于深度学习的模型（例如，深度神经网络和卷积神经网络）在恶意软件检测等方面的性能比较。 [2]（章节 VI-A3）和恶意软件家族分类（章节VI-A4）任务。请注意，在源代码泄露的情况下（Mirai [37] 和 Gafgyt [27]），我们生成了采用极端内联的恶意软件变种。对于这两项任务，由于静态特征扰动（例如，调用图、控制流图），每个模型的性能都显著下降：恶意软件检测下降约 20% ↓，恶意软件家族预测下降约 40% ↓。

安全任务	机器学习模型	真实环境中的恶意软件				具有极端内联的恶意软件			
		准确率	精确率	召回率	F1	准确率	精确率	召回率	F1
恶意软件检测	逻辑回归	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.81 ± 0.03**	**0.84 ± 0.02**	0.81 ± 0.03	**0.81 ± 0.03**
	随机森林	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	0.75 ± 0.13	0.77 ± 0.13	0.75 ± 0.13	0.74 ± 0.14
	CatBoost	0.98 ± 0.02	0.98 ± 0.02	0.98 ± 0.02	0.98 ± 0.02	0.77 ± 0.01	0.84 ± 0.00	0.77 ± 0.01	0.75 ± 0.01
	KNN	0.93 ± 0.01	0.93 ± 0.01	0.93 ± 0.01	0.93 ± 0.01	0.78 ± 0.00	0.80 ± 0.01	0.78 ± 0.00	0.77 ± 0.00
	Abusnaina等人 [2](卷积神经网络)	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.01**	0.78 ± 0.01	0.75 ± 0.02	**0.83 ± 0.03**	0.79 ± 0.01
	Abusnaina等人 [2](深度神经网络)	**0.99 ± 0.01**	**0.99 ± 0.01**	0.99 ± 0.00	**0.99 ± 0.01**	0.75 ± 0.02	0.72 ± 0.01	0.82 ± 0.05	0.76 ± 0.02
	平均值	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.77 ± 0.05	0.78 ± 0.04	0.79 ± 0.01	0.78 ± 0.05
恶意软件家族预测	逻辑回归	0.88 ± 0.05	0.90 ± 0.06	0.88 ± 0.05	0.88 ± 0.06	0.48 ± 0.06	0.45 ± 0.07	0.48 ± 0.06	0.46 ± 0.05
	随机森林	**0.91 ± 0.05**	**0.92 ± 0.05**	**0.91 ± 0.05**	**0.90 ± 0.06**	0.50 ± 0.00	0.47 ± 0.06	0.50 ± 0.00	0.48 ± 0.04
	CatBoost	0.89 ± 0.03	0.90 ± 0.03	0.89 ± 0.03	0.88 ± 0.03	**0.53 ± 0.10**	**0.62 ± 0.30**	**0.53 ± 0.10**	**0.54 ± 0.17**
	KNN	0.90 ± 0.06	0.92 ± 0.06	0.90 ± 0.06	0.90 ± 0.07	0.44 ± 0.05	0.49 ± 0.03	0.44 ± 0.05	0.46 ± 0.04
	Abusnaina等人 [2](卷积神经网络)	0.82 ± 0.06	0.85 ± 0.05	0.82 ± 0.06	0.81 ± 0.06	0.23 ± 0.18	0.43 ± 0.13	0.23 ± 0.18	0.25 ± 0.14
	Abusnaina等人 [2](深度神经网络)	0.78 ± 0.07	0.80 ± 0.07	0.78 ± 0.07	0.76 ± 0.09	0.36 ± 0.15	0.29 ± 0.15	0.36 ± 0.15	0.31 ± 0.13
	平均值	0.87 ± 0.05	0.88 ± 0.05	0.87 ± 0.05	0.87 ± 0.05	0.44 ± 0.09	0.46 ± 0.09	0.43 ± 0.09	0.41 ± 0.10

表 VII：性能比较（以 MRR@100 计）

模型	Netgear R7000		TP-Link Deco-M4	
	非内联	内联	非内联	内联
Gemini [104]	0.049	0.000	0.321	0.004
Asm2Vec [28]	0.256	0.128	0.016	0.008
SAFE [75]	0.125	0.003	0.018	0.003
Trex [85]	0.550	0.625	0.286	0.048

使用函数内联的四种漏洞检测模型。我们从两个固件映像的 libcrypto 中选取 12 个易受攻击的（目标）函数 [16]（Netgear R700 和 TP-Link Deco-M4）。然后，我们在 OpenSSL-1.0.2d 中准备了一组可比较的函数 [81]，分别采用内联和非内联方式，以探测这些易受攻击的函数。应用内联后，各模型的性能大多显著下降。我们计算平均倒数排名以进行评估。（章节VI-A5）。



(a) 高度优化（-O3）　　(b) 极端内联

图 6：使用 (a) -O3 优化级别和 (b) 我们的极端内联策略进行编译时，coreutils 软件包中内联函数及其存在情况的可视化。与 -O0相比，在 -O3（章节 VI-B1）中，超过一半的函数已被内联，随后又被消除。

除 -O0 之外的优化级别。首先，在所有优化级别下，coreutils 的内联率最高，而 nginx 在大多数优化级别下排名最低。其次，在每个软件包中，-Os 和 -Oz 的内联率始终较低，因为其优化目的是减小二进制大小（即，函数内联操作会增大二进制大小）。第三，函数内联率会因应用程序而异。最后，仍有不可忽视数量的函数未被内联，这主要是因为这些函数具有外部链接，或者具有内部链接但并未在所有调用点内联。图 8 展示了基线数据集中所有二进制文件的内联率累积分布。值得注意的是，即使在 -O0 下，函数内联率也略有体现（例如，始终内联显示的最高内联率为 9.52%，平均值为 0.83%）。相比之下，-O[1-3]的内联率均显著较高，最高为 66.67%，平均值分别为 32.89%、32.77% 和 32.70%。我们观察到，-Os 和 -Oz的最高内联率也相近，分别为 65.83% 和 61.67%。然而，-Os 和 -Oz的平均内联率略有下降，分别为 27.42% 和 20.47%，表明这些优化策略侧重于减小二进制大小。

2)(RQ7)（*Clang* 提供的）编译器选项对函数内联率的影响：图 9展示了启用四个与函数内联相关的默认编译器选项（-finline-hint-functions、-fno-inline-functions、-finline-functions、-fno-inline）的影响；默认值表示未提供此类选项。使用-finline-functions 选项时的内联率与默认设置相同。同时，-finline-hint-functions的内联率相对较低。在 SPEC2006 中，-fno-inline 和-fno-inline-functions 选项均有效。值得注意的是，always_inline有助于在 coreutils 中维持少量函数内联，即使启用了抑制内联的选项也是如此。

3)（*RQ8*）探索编译器选项（由 *Opt* 提供）以实现极端内联：我们研究12 与函数内联相关的编译器选项，旨在根据 LLVM 中的成本模型，寻找一组能够实现极端内联的编译器选项组合。受 BinTuner 启发， [92]，我们首先设置一个有助于提高内联率的特定值。然而，我们发现，调整初始值可能会显著增加编译时间。为了在合理的编译时间内考察内联率，我们

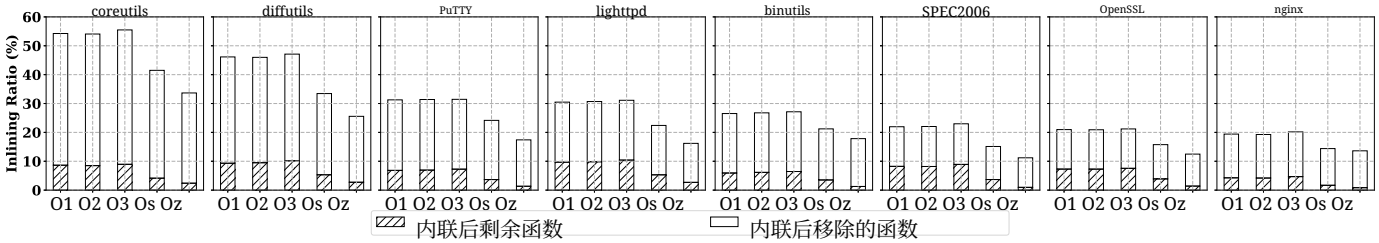


图 7：在八个应用程序中，采用不同优化级别（`-O[0-3,s,z]`）编译时函数内联比率的比较。由于单个函数的内联决策在很大程度上取决于成本模型（章节 VI-B1）中成本与阈值的动态变化，内联比率可能存在显著差异。例如，coreutils 在所有优化级别中的内联率均最高（例如，超过一半），而 nginx 的排名最低。`-Os` 和`-Oz` 中的比率相较于`-O[0-3]`更小。尽管大多数内联函数都被移除，但仍有数量不可忽略的函数（约 10%）保留下来。

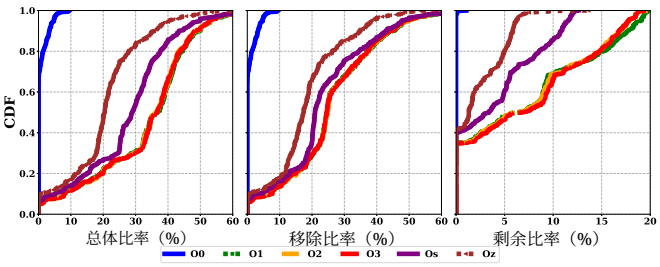


图 8：使用不同优化级别 `-O[0-3,s,z]` 编译的数据集中所有二进制文件的函数内联率累积分布函数（CDF）（章节 VI-B1）。即使在 `-O0` 下也会进行内联优化，而激进的优化（`-O3`）会促使更多函数内联。

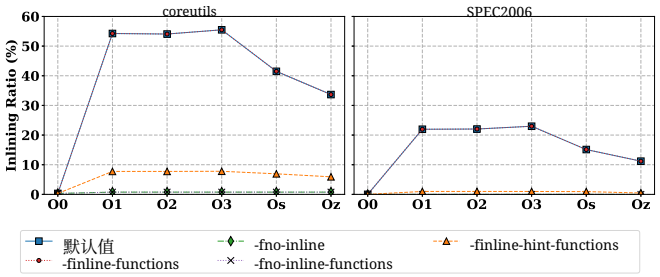


图 9：会全局影响内联行为的编译器提供的选项的有效性（章节 VI-B2）。我们测量了 coreutils 和 SPEC2006 在不同优化级别 `-O[0-3,s,z]`下的函数内联率。我们确认，`-fno-inline-functions` 和`-fno-inline`选项能够有效禁止内联行为。

以固定间隔递增某个特定值，例如，针对良性应用程序每次增加 500（例如，[0, 500, 1,000, ...]），针对恶意软件每次增加 50,000（例如，[0, 50,000、100,000，.....]）。对于定义为布尔值的选项，我们将其默认值取反（例如，$true \rightarrow false$）。

结果。 与其他对函数内联率影响甚微的选项相比，该比率通常与`-inline-threshold`成正比，与`-inline-call-penalty` 成反比（图 10）。`-inline-threshold` 选项会覆盖全局阈值，而`-inline-call-penalty` 选项则会调整调用惩罚值以计算成本。例如，当阈值从默认值提高时，内联率也会提高

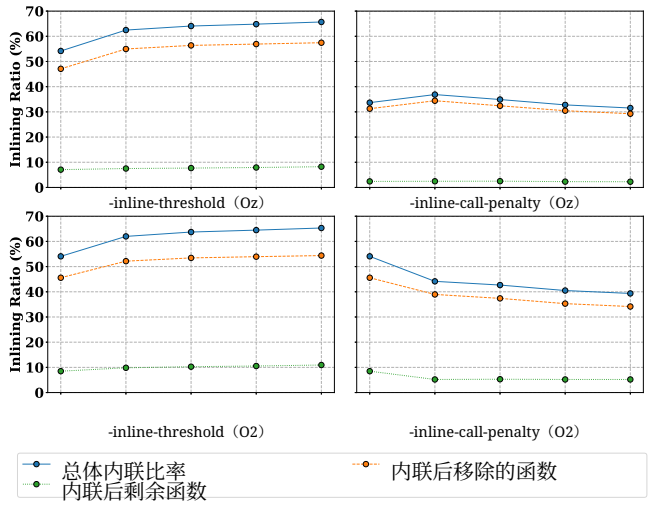


图 10：我们探索编译器选项组合时得到的部分结果。对于 coreutils，我们对编译器选项的值（即增大或减小）（表 II）进行了广泛调优，这些选项最终可能影响函数内联。我们的发现表明，启发式地增大`-inline-threshold` 并减小`-inline-call-penalty`有助于提高整体函数内联率（章节 VI-B3）。

值（例如，$225 \rightarrow 2$，225）。值得注意的是，这些编译器选项不会产生表 III中的启发式方法。

函数内联与 LTO。 接下来，我们研究 LTO 如何影响整体函数内联率。LTO 会在每个目标文件完成初始的预链接优化后，再进行一轮额外优化。我们考察了四种情形：无 LTO、完整链接时优化、ThinLTO，以及采用极端函数内联的 LTO。我们将 `-inline-threshold` 选项设为 200,000，并使用完整链接时优化成功编译了三个软件包（coreutils、diffutils 和查找工具集）。如图 11所示，完整链接时优化的内联率高于 ThinLTO，这是因为完整链接时优化在单个线程上运行，并可访问所有目标文件，而 ThinLTO 则使用多个线程和摘要信息。有趣的是，在 `-O1`下，ThinLTO 的内联率高于完整链接时优化，这可能是由于其优化更高效、更激进。LTO 还利用内部化遍，将外部链接属性转换为内部链接属性，从而提高内联率 [47]。我们的极端内联策略

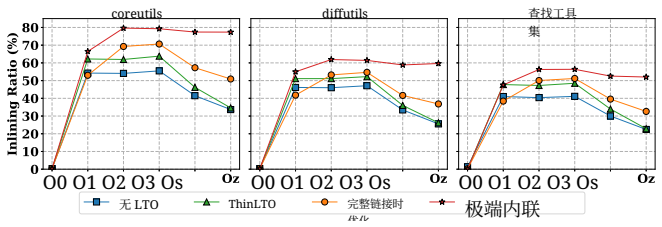


图 11：链接时优化（LTO）在不同优化级别和应用程序中的效果。启用 LTO 后，在-O1及更高优化级别下，函数内联率明显更高。值得注意的是，我们的极端函数内联策略甚至超过了完整链接时优化的内联率（例如，在 coreutils 中达到 79.64%）。

使内联率极高，在 coreutils 中达到 79.64%。

4)（*RQ9*）根据函数内联进行静态特征分析：我们使用 TikNib 分析了 62 项与指令、控制流图和调用图相关的统计特征 [53]。为简洁起见，我们选取中位数差距最大的 18 项特征（附录中的图 14）。请注意，我们在附录中的表 X列出了完整的静态特征名称。为确保公平比较，我们对数据进行归一化处理，并使用三西格玛法则剔除异常值 [51]，即排除任何超出均值三个标准差的数据点。我们比较了 coreutils 中 106 个二进制文件在-O0 和极端内联下的情况。附录中的图 14显示，极端内联会显著改变可执行二进制文件的统计特征。例如，算术指令（特征 6 和 8）显著增加，这可能是由于在多个调用点内联了包含算术运算的函数。此外，我们观察到循环数量（特征 38 和 39）减少，但循环大小（特征 46、48 和 49）增加。这表明内联会产生更少但更大的循环。极端内联可能会促成更激进的优化，例如循环展开和移除较小的循环。这些变化会显著影响统计特征，我们预计这将影响依赖此类特征的机器学习模型。

VII. 实现

可执行二进制文件的真实值提取。我们使用 -g 选项，以 ELF（可执行与可链接格式）格式编译不同的软件，该选项会生成调试信息。我们使用 Python 和 pyelftools 库 [12] 来解析 ELF，并从 DWARF 中提取DW_AT_inline 属性（即函数符号中的内联函数） [19]。我们通过每个函数的 DW_TAG_subprogram标签识别调用者列表，然后遍历每个 DWARF 条目中带有 DW_TAG_inlined_subroutine标签的子节点，该标签表明一个函数包含多个内联函数。此外，我们还利用 TikNib [54]来提取
62 种特征（附录中的表 X ）可能会影响控制流图、调用图和指令序列。

基于机器学习的模型。首先，我们利用二进制代码相似性检测基准的实现[74] 来评估二进制代码相似性检测任务，而不是重新发明轮子，其中包括 Asm2Vec [28], Gemini [104],Trex [85], 和 SAFE [75]。我们采用 BinShot 的原始实现 [5], , 该实现无法在上述基准中获取

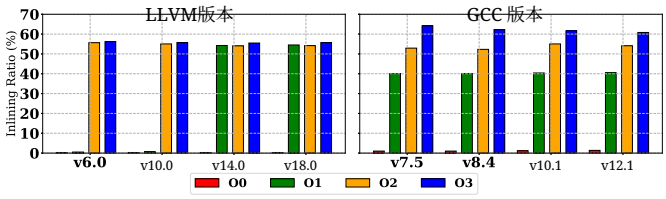


图 12：多个版本的 LLVM（左）和 GCC（右）在不同优化级别（O0–O3）下函数内联比率的对比。对于所有编译器版本，更高的优化级别始终会带来更高的内联率；与 LLVM 相比，GCC 不同版本之间的差异更大。

。其次，对于恶意软件检测和家族预测，我们采用 Lei等人 [60]的方法，使用 Scikit-learn 库生成四种传统机器学习模型（表 VI） [25]。此外，我们使用 TensorFlow 重新实现了 Abusnaina 提出的两个深度学习模型等人 [3]提出的 [1]（表 VI）。第三，我们使用通过 LLVM 编译的数据集重新训练了 AsmDepictor [55] 和 SymLM [46] ，用于函数名称预测。第四，我们通过利用二进制代码相似性检测基准的实现 [74] 和可用的易受攻击固件语料库 [16]来评估漏洞检测。

VIII. 讨论与未来工作

DWARF 中未报告的函数内联情况。尽管 DWARF 包含大量有关函数内联的信息，但有几种情况会（有意）不报告这些信息。我们发现，在变换过程中，如果整个内联函数体被移除（例如，死代码消除），就会缺少内联记录。另一种未报告的情况是，某个特定的内建函数被内联到其他函数中。这可能导致函数内联器 pass 的输出（例如，-Rpass=inline）与解析 DWARF 的结果不一致。

数据集代表性。 尽管我们为实验精心挑选了广泛多样的应用程序作为数据集，但它可能无法充分代表所有二进制文件，因此无法确保我们的研究发现能够安全地推广。我们将更多场景和边缘情况留作未来工作，并需要借助多样化的数据集进行探索，以揭示函数内联行为。

编译器之间的行为差异。 从经验上看，由于不同编译器及其不同版本在代码优化中的函数内联动态各异，结果可能并不一致。 图12 展示了 GCC 和 LLVM 编译器不同版本在各优化级别下的函数内联率。LLVM 的内联行为相对一致，而 GCC 的内联则更为激进，尤其是在较高优化级别下。尽管我们的实验聚焦于特定版本的 LLVM，但我们的观察也适用于其其他版本，可为理解现代编译器如何处理函数内联提供宝贵见解。对 GCC 内部内联决策的深入研究留待未来工作。

函数内联率评估指标。 我们采用基于（内联）函数数量的函数内联率评估指标。然而，内联决策

是在调用点粒度上做出的，同时考虑调用者和被调用者。因此，对于同一个函数被内联到多个位置，或存在一系列（嵌套）内联的情况，我们的指标可能无法体现。

潜在的缓解措施。我们提出了几种潜在的缓解措施：1 使用内联数据增强模型训练，2 采用结合编译器感知变换的对抗训练，以及 3 应用内联感知预处理（例如，Highliner [22] 采用反内联启发式方法）。我们建议感兴趣的读者参阅附录表 XII，了解使用内联函数进行增强训练的初步结果。

IX. 相关工作

二进制逆向特征工程。Genius [34]采用属性控制流图（ACFG），在基本块粒度上纳入了八种特征（统计特征和结构特征）。类似地，DL-FHMC [2] 在控制流图的基础上引入了 23 种额外特征。此外，ImOpt [45] 探索中间表示层的特征，以应对编译器优化和混淆技术。与之相反，αDiff [65] 采用深度神经网络直接提取特征。TikNib [53]利用大规模基准提取了 72 种统计特征，结果表明，简单的可解释模型使用精确特征时，其性能可与最先进的深度学习模型相媲美。请注意，我们借鉴了 62 种适用于二进制层面的特征，以研究内联的影响。

函数内联与二进制。许多先前工作从工程角度研究函数内联。Dam´asio 等人 [23]关注通过内联缩减代码体积，而 Theodoridis 等人 [98] 则探索最优内联策略。与此同时，其他研究在二进制分析背景下考察函数内联。与我们工作最为接近的一项研究是 Jia等人 [43],的工作，他们研究了函数内联对二进制相似性分析的影响。Bingo [14]提出了一种动态内联模拟策略，通过递归展开被调用函数来改进相似性检测。类似地，Asm2Vec [28] 采用选择性被调用函数展开，将 Bingo 的策略应用于静态分析。FSmell [64] 提出了一种基于 ML 的框架，通过指令拓扑图检测内联函数。与此同时，Koo 等人 [57] 和 AsmDepictor [55] 强调了深入理解函数内联的重要性。尽管如此，许多面向二进制分析的基于 ML 的研究 [86], [32], [103], [33], [105], [46], [83], [49], [84], [11] 往往在缺乏充分研究的情况下低估或忽视内联的影响。不同于以往方法，我们的工作揭示了编译器的内联决策过程，以及通过极端内联精心构造规避性二进制变异而被滥用的可能性。

机器学习辅助的静态二进制分析方法。过去十年间，机器学习辅助的安全任务在各个领域得到广泛应用，并展现出良好的效果。此类任务包括恶意软件检测、恶意软件家族分类、二进制代码相似性检测（BCSD）以及函数名称预测。在二进制代码相似性检测领域，VulSeeker [38] 采用基于孪生网络的图嵌入模型来提升相似性检测能力。InnerEye [110] 和 SAFE [75] 采用自然语言处理（NLP）方法，从汇编代码中学习语义。类似地，Asm2Vec [28]和 DeepBinDiff [30] 利用无监督学习进行训练指令的上下文信息，从而提升检测代码相似性的能力。近期，BinShot [5] 学习加权距离向量，以便更有效地区分不相似的代码；我们将其纳入评估。对于恶意软件应用，Alasmary 等人 [6] 在安卓平台上开展了全面的基于控制流图的物联网恶意软件检测任务。Cozzi等人 [21]研究恶意软件家族的演化脉络，追踪它们在不同变体之间的关系。此外，BinTuner [92] 和 CARDINAL [48]尝试使用不同的编译选项重新编译恶意软件样本，以击败依赖静态工程特征的基于特征码的方法。与此同时，近期研究致力于恢复编译过程中丢失的信息，包括推断函数名和变量名、类型，甚至反编译代码。Debin [41]专注于恢复变量名和类型，并从二进制层面的指令序列中学习其底层语义。NERO [24]基于增强控制流图训练调用点的丰富表示，以预测函数符号名。类似地，AsmDepictor [55] 将最先进的基于 Transformer 的模型用于函数名称预测任务，并将其纳入我们的评估。

编译器标志与行为研究。Dong 等 [29]和 Zhang [109] 等研究符号执行，以探索编译器标志：前者比较不同优化级别，后者则通过实证分析 GCC/Clang 优化标志。Ren 等 [93]关注用于二进制差异比对的 LLVM 窥孔优化，而 BinTuner [92] 探索大规模编译器标志空间，以实现二进制相似性。同时，我们的工作面向基于机器学习的二进制分析，并通过系统考察 LLVM 的内联代码库，进一步研究极端内联，以评估和规避机器学习模型。

X. 结论

函数内联是现代编译器广泛采用的一种优化技术。这种内联行为会显著影响二进制逆向任务中的静态特征；然而，尽管其重要性和影响显著，对此的研究仍然不够充分。在本文中，我们首先围绕函数内联开展全面研究，包括但不限于：考察其决策流程、研究直接影响内联率的编译器选项、揭示常见误解，以及评估其对面向机器学习的应用所产生的安全影响。我们的主要发现表明，函数内联可能被用于恶意目的，因此在构建基于机器学习的模型时需要对此予以关注。

致谢

感谢匿名评审提出的建设性意见。本研究部分得到了由韩国政府（MSIT；科学技术信息通信部）资助的信息通信技术规划评估院（IITP）项目支持：No. RS-2024-00337414、No. RS-2024-00437306、No. RS-2020-II201821 和 No. RS-2022-II220688。此外，本研究还得到由韩国政府教育部资助、通过韩国国家研究基金会（NRF）开展的基础科学研究项目支持：No. RS-2025-02293072。本文所表达的任何观点、研究结果、结论或建议均属于作者个人意见，不一定反映赞助方的观点。

参考文献

[1] M. Abadi、A. Agarwal、P. Barham、E. Brevdo 等，“TensorFlow：异构系统上的大规模机器学习”，2015年，https://www.tensorflow.org/。[2] A. Abusnaina、M. Abuhamad、H. Alasmary、A. Anwar、R. Jang、S. Salem、D. Nyang 和 D. Mohaisen，“DL-FHMC：一种基于深度学习的细粒度分层学习方法，用于鲁棒的恶意软件分类”，IEEE《可靠与安全计算汇刊》，2021年。[3] A. Abusnaina、A. Anwar、S. Alshamrani、A. Alabduljabbar、R. Jang、D. Nyang 和 D. Mohaisen，“系统评估基于机器学习的物联网恶意软件检测系统的鲁棒性”，载于《攻击、入侵与防御研究国际研讨会》第25届会议论文集，2022年。[4] T. Ahmed、P. Devanbu 和 A. A. Sawant，“学习在优化后的二进制文件中查找库函数的使用位置”，IEEE软件工程汇刊，2021年。[5] S. Ahn、S. Ahn、H. Koo 和 Y. Paek，“基于BERT的可迁移相似性学习实现实用的二进制代码相似性检测”，载于第38届年度计算机安全应用大会论文集，2022年。[6] H. Alasmary、A. Anwar、J. Park、J. Choi、D. Nyang 和 A. Mohaisen，“基于图的物联网恶意软件与 Android 恶意软件比较”，载于计算社会网络国际会议论文集，2018年。[7] O. Alrawi、C. Lever、K. Valakuzhy、K. Snow、F. Monrose、M. Antonakakis 等，“生命循环：物联网恶意软件生命周期的大规模研究”，载于第30届USENIX安全研讨会，2021年。[8] J. Alves-Foss 和 V. Venugopal，“二进制分析中真实值令人不便的真相”，载于二进制分析研究研讨会，2022年。[9] M. Antonakakis、T. April、M. Bailey、M. Bernhard、E. Bursztein、J. Cochran、Z. Durumeric、J. A. Halderman、L. Invernizzi、M. Kallitsis 等，“理解 Mirai 僵尸网络”，载于第26届USENIX安全研讨会，2017年。[10] ARM，“Arm 开发者”，2023年，
https://developer.arm.com/documentation/100067/0608/armclang-Command-line-Options/-fno-inline-functions。[11] T. Bao、J. Burket、M. Woo、R. Turner 和 D. Brumley，“BYTEWEIGHT：学习识别二进制代码中的函数”，载于第23届USENIX安全研讨会，2014年。[12] E. Bendersky，“pyelftools：用于解析 ELF 和 DWARF 的纯 Python 库”，2024年，https://github.com/eliben/pyelftools。[13] Binutils，“GNU Binutils 2.40”，2023年，https://ftp.gnu.org/gnu/binutils/。[14] M. Chandramohan、Y. Xue、Z. Xu、Y. Liu、C. Y. Cho 和 H. B. K. Tan，“Bingo：跨架构、跨操作系统的二进制搜索”，载于2016年第24届ACM SIGSOFT软件工程基础国际研讨会论文集，2016年。[15] Q. Chen、J. Lacomis、E. J. Schwartz、C. Le Goues、G. Neubig 和 B. Vasilescu，“利用学习得到的变量名和类型增强反编译器输出”，载于第31届USENIX安全研讨会，2022年。[16] Cisco-Talos，“二进制函数相似性”，2024年，https://github.com/Cisco-Talos/binary-function-similarity。[17] Clang，“生成优化报告的选项”，https://clang.llvm.org/docs/UsersManual.html#options-to-emit-optimization-reports，2024年。[18] C. Collberg、C. Thomborson 和 D. Low，“混淆变换分类”，1997年。[19] D. D. I. F. Committee，“Dwarf 调试信息格式第 5 版”，2017年，https://dwarfstd.org/dwarf5std.html。[20] Coreutils，“GNU Coreutils”，2023年，https://github.com/coreutils/coreutils。[21] E. Cozzi、P.-A. Vervier、M. Dell’Amico、Y. Shen、L. Bilge 和 D. Balzarotti，“物联网恶意软件的复杂谱系”，载于第36届年度计算机安全应用大会论文集，2020年。[22] L. Dall’Aglio、L. Binosi、M. Carminati、S. Zanero 和 M. Polino，“Highliner：通过基于自然语言处理的指令级 C++内联函数检测增强二进制分析”，ACM《隐私与安全汇刊》，2025。[23] T. Dam´asio、V. Pacheco、F. Goes、F. Pereira 和 R. Rocha，“通过内联缩减代码大小”，载于《第25届巴西编程语言研讨会论文集》，2021年。[24] Y. David、U. Alon 和 E. Yahav，“使用增强控制流图对剥离符号的二进制文件进行神经逆向工程”，《ACM编程语言汇刊》，2020年。[25] S.-L. Developers，“Scikit-learn：Python中的机器学习”，2023年，https://scikit-learn.org/stable/。[26] Diffutils，“GNU Diffutils 3.8”，2023年，https://ftp.gnu.org/gnu/diffutils/。[27] F. Ding，“物联网恶意软件：Lizkebab（Gafgyt）”，2023年，https://github.com/i fding/iot-malware/tree/master/lizkebab。[28] S. H. Ding、B. C. Fung 和 P. Charland，“Asm2Vec：通过代码混淆和编译器优化增强静态表示的鲁棒性，以进行二进制克隆搜索”，载于《2019年IEEE安全与隐私研讨会》，2019年。[29] S. Dong、O. Olivo、L. Zhang 和 S. Khurshid，“研究标准编译器优化对符号执行的影响”，载于《2015年IEEE第26届软件可靠性工程国际研讨会》，2015年。[30] Y. Duan、X. Li、J. Wang 和 H. Yin，“DeepBinDiff：学习程序级代码表示以进行二进制差异比对”，载于《网络与分布式系统安全研讨会论文集》，2020年。[31] M. Egele、M. Woo、P. Chapman 和 D. Brumley，“全覆盖执行：针对程序二进制文件和组件的动态相似性测试”，载于《第23届USENIX安全研讨会》，2014年。[32] S. Eschweiler、K. Yakdan、E. Gerhards-Padilla 等，“Discovre：二进制代码中漏洞的高效跨架构识别”，载于《网络与分布式系统安全研讨会论文集》，2016年。[33] Q. Feng、M. Wang、M. Zhang、R. Zhou、A. Henderson 和 H. Yin，“提取条件公式以进行跨平台漏洞搜索”，载于《2017年ACM亚洲计算机与通信安全会议论文集》，2017年。[34] Q. Feng、R. Zhou、C. Xu、Y. Cheng、B. Testa 和 H. Yin，“面向固件映像的可扩展基于图的漏洞搜索”，载于《2016年ACM SIGSAC计算机与通信安全会议论文集》，2016年。[35] Findutils，“GNU Findutils”，2023年，https://github.com/aixoss/findutils。[36] C. Fu、H. Chen、H. Liu、X. Chen、Y. Tian、F. Koushanfar 和 J. Zhao，“Coda：端到端神经程序反编译器”，《神经信息处理系统进展》，2019年。[37] J. Gamblin，“Mirai源代码”，2023年，源代码：https://github.com/jgamblin/Mirai-Source-Code。[38] J. Gao、X. Yang、Y. Fu、Y. Jiang 和 J. Sun，“VulSeeker：一种基于语义学习的跨平台二进制漏洞搜索工具”，载于《第33届ACM/IEEE自动化软件工程国际会议论文集》，2018年。[39] T. Glek 和 J. Hubicka，“使用GCC链接时优化优化实际应用程序”，arXiv预印本arXiv:1010.2196，2010年。[40] GSL，“GNU科学计算库2.7.0”，https://github.com/ampl/gsl，2023年，访问日期：2024年7月10日。[41] J. He、P. Ivanov、P. Tsankov、V. Raychev 和 M. Vechev，“Debin：预测剥离符号的二进制文件中的调试信息”，载于《2018年ACM SIGSAC计算机与通信安全大会论文集》，2018年。[42] Y. He、J. Lou、Z. Qin 和 K. Ren，“Finer：利用特征归因增强最先进的分类器，以促进安全分析”，载于《2023年ACM SIGSAC计算机与通信安全大会论文集》，2023年。[43] A. Jia、M. Fan、W. Jin、X. Xu、Z. Zhou、Q. Tang、S. Nie、S. Wu 和 T. Liu，“一对一还是一对多？探究函数内联对二进制相似性分析的影响”，《ACM软件工程与方法论汇刊》，2023年。[44] A. Jia、M. Fan、X. Xu、W. Jin、H. Wang 和 T. Liu，“跨内联二进制函数相似性检测”，载于《IEEE/ACM软件工程国际会议论文集》，2024年。[45] J. Jiang、G. Li、M. Yu、G. Li、C. Liu、Z. Lv、B. Lv 和 W. Huang，“不同优化级别和混淆下二进制文件的相似性”，载于

《欧洲计算机安全研究研讨会论文集》，2020年。[46] X. Jin、K. Pei、J. Y. Won 和 Z. Lin，“SymLM：通过上下文敏感且执行感知的代码嵌入预测剥离符号的二进制文件中的函数名”，载于《2022年ACM SIGSAC计算机与通信安全大会论文集》，2022年。[47] T. Johnson、M. Amini 和 X. D. Li，“ThinLTO：可扩展的增量式LTO”，载于《2017年IEEE/ACM国际代码生成与优化研讨会》，2017年。[48] L. Jones、A. Sellers 和 M. Carlisle，“CARDINAL：通过相似性分析对抗恶意软件编译器变体”，载于《2016年第11届恶意与非期望软件国际会议》，2016年。[49] D. S. Katz、J. Ruchti 和 E. Schulte，“使用循环神经网络进行反编译”，载于《2018年IEEE第25届软件分析、演化与再工程国际会议》，2018年。[50] O. Katz、Y. Olshaker、Y. Goldberg 和 E. Yahav，“迈向神经网络反编译”，arXiv预印本arXiv:1905.08325，2019年。[51]W. Kenton，“三西格玛限值定义”，2023年，
https://www.investopedia.com/terms/t/three-sigma-limits.asp。[52] R. Kettlewell，“C语言中的内联函数”，2024年，
https://www.greenend.org .uk/rjk/tech/inline.html。[53] D. Kim、E. Kim、S. K. Cha、S. Son 和 Y. Kim，“使用可解释特征工程重新审视二进制代码相似性分析及经验总结”，《IEEE软件工程汇刊》，2022年。[54] ——，“二进制代码相似性分析（BCSA）工具”，2023年，https://github.com/SoftSec-KAIST/TikNib/tree/master。[55] H. Kim、J. Bak、K. Cho 和 H. Koo，“基于Transformer的从汇编语言推断函数符号名称的模型，用于二进制逆向”，载于《2023年ACM亚洲计算机与通信安全会议论文集》，2023年。[56] T. Kim、A. Ding、S. Etigowni、P. Sun、J. Chen、L. Garcia、S. Zonouz、D. Xu 和 D. Tian，“使用Dispatch对机器人飞行器控制固件进行逆向工程和改造”，载于《第20届移动系统、应用与服务年度国际会议》，2022年。[57] H. Koo、S. Park 和 T. Kim，“回顾函数识别问题”，载于《第37届年度计算机安全应用大会论文集》，2021年。[58] T. Le、T. V. Nguyen、T. Le、T. Phung、P. Montague、O. De Vel 和 L. Qu，“用于二进制软件漏洞检测的最大散度序列自编码器”，载于《国际学习表征会议》，2019年。[59] M. Learn，“函数内联问题”，2021年，
https://learn.microsof
t.
c
o
m/en-us/cpp/error-messages/tool-errors/function-inlining-problems?view=msvc-170。[60] T. Lei、J. Xue、Y. Wang、T. Baker 和 Z. Niu，“物联网恶意软件分类标签来源的问题及评估实证研究”，《沙特国王大学计算机与信息科学杂志》，2024年。[61] Z. Li、D. Zou、S. Xu、X. Ou、H. Jin、S. Wang、Z. Deng 和 Y. Zhong，“VulDeePecker：用于漏洞检测的基于深度学习的系统”，《网络与分布式系统安全研讨会论文集》，2018年。[62] Lighttpd，“Lighttpd 2.0.0”，2023年，
https://download.lighttpd.net/lightt pd/snapshots-2.0.x/。[63] G. Lin、J. Zhang、W. Luo、L. Pan 和 Y. Xiang，“海报：从无标签项目中进行函数表征学习以发现漏洞”，载于《2017年ACM SIGSAC计算机与通信安全大会论文集》，2017年。[64] W. Lin、Q. Guo、J. Yin、X. Zuo、R. Wang 和 X. Gong，“FSmell：识别二进制代码中的内联函数”，载于《欧洲计算机安全研究研讨会》，2023年。[65] B. Liu、W. Huo、C. Zhang、W. Li、F. Li、A. Piao 和 W. Zou，“αdiff：使用深度神经网络进行跨版本二进制代码相似性检测”，载于《第33届ACM/IEEE自动化软件工程国际会议论文集》，2018年。[66] LLVM项目，“CGSCC Pass Manager.h”，2024年，https://llvm.org/dox ygen/CGSCCPassManager 8h.html。[67] ——，《早期公共子表达式消除遍次》，2024年，1 1
https://llvm.org/doxygen/structllvm EarlyCSEPass.html。[68] ——，《LLVM 链接时优化：设计与实现》，2024年，
https://llvm.org/docs/LinkTimeOptimization.html。[69] ——，《LLVM 遍次文档》，2024年，https://llvm.org/docs/Pa sses.html。[70] ——，《控制流图简化遍次》，2024年，https://llvm.org/doxygen/classllvm 1 1SimplifyCFGPass.html。[71] ——，《标量替换聚合遍次》，2024年，
https://llvm.org/doxygen/classllvm 1 1SR OAPass.html。[72] S. M. Lundberg 和 S.-I. Lee，《一种统一的模型预测解释方法》，《神经信息处理系统进展》，2017年。[73] Z. Luo、P. Wang、B. Wang、Y. Tang、W. Xie、X. Zhou、D. Liu 和 K. Lu，《Vulhawk：利用基于熵的二进制代码搜索进行跨架构漏洞检测》，载于《网络与分布式系统安全研讨会论文集》，2023年。[74] A. Marcelli、M. Graziano、X. Ugarte-Pedrero、Y. Fratantonio、M. Mansouri 和 D. Balzarotti，《机器学习如何解决二进制函数相似性问题》，载于第31届 USENIX 安全研讨会，2022年。[75] L. Massarelli、G. A. Di Luna、F. Petroni、R. Baldoni 和 L. Querzoni，《SAFE：用于二进制相似性的自注意力函数嵌入》，载于《入侵与恶意软件检测及漏洞评估国际会议》，2019年。[76]H. Mauelshagen，《逻辑卷管理器》，2023年，https://github.com/lvmteam/lvm2/tags。[77] Melissa，《恶意软件样本库》，2023年，https://virusshare.c om/。[78] T. Micro，《Bash 漏洞（Shellshock）利用代码现身野外，引发 BashLite 恶意软件》，2014年，201811 29100545
https://web.archive.org/web/ /
https://blog.trendmicro.com/trendlabs-security-intelligence/bash-vulnerability-shellshock-exploit-emerges-in-the-wild-leads-to-f looder/。[79] O. MPI，《Open MPI 4.1.5》，2023年，https://www.open-mpi.org/softwar e/ompi/v4.1/。[80] Nginx，《nginx》，2023年，https://github.com/nginx。[81] OpenSSL，《OpenSSL 3.1.4》，2023年，
https://github.com/openssl/openssl。[82] Y. Park、H. Lee、J. Jung、H. Koo 和 H. K. Kim，《Benzene：一种采用欠约束状态变异的实用根因分析系统》，载于2024年 IEEE安全与隐私研讨会，2024年。[83] J. Patrick-Evans、L. Cavallaro 和 J. Kinder，《剥离符号的二进制文件中函数的概率命名》，载于第36届年度计算机安全应用大会论文集，2020年。[84] K. Pei、J. Guan、D. Williams-King、J. Yang 和 S. Jana，《XDA：利用迁移学习实现准确、稳健的反汇编》，载于《网络与分布式系统安全研讨会论文集》，2021年。[85] K. Pei、Z. Xuan、J. Yang、S. Jana 和 B. Ray，《Trex：从微型跟踪中学习执行语义以实现二进制相似性》，《IEEE软件工程汇刊》，2022年。[86] J. Pewny、B. Garmany、R. Gawlik、C. Rossow 和 T. Holz，《二进制可执行文件中的跨架构漏洞搜索》，载于2015年 IEEE安全与隐私研讨会，2015年。[87] L. Project，《LLVM 14.0.0 发布版本》，2023年，https://releases.llvm.org/14.0.0/docs/ReleaseNotes.html。[88] ——，《LLVM 语言参考手册》，2024年，https://llvm.org/doc s/LangRef.html#intrinsic-functions。[89] PuTTY，《PuTTY 0.79》，2023年，
https://www.chiark.greenend.org.uk/~sgt atham/putty/latest.html。[90] D. A. Ramos 和 D. Engler，《欠约束符号执行：真实代码的正确性检查》，载于第24届 USENIX 安全研讨会，2015年。[91] M. Reddy，《C 语言的 API 设计》，2011年，elsevier，2011年。++X. Ren、M. Ho、J. Ming、Y. Lei 和 L. Li，《释放编译器优化在二进制代码差异中的隐藏潜力：一项实证研究》，载于第42届 ACM SIGPLAN 编程语言设计与实现国际会议论文集，2021年。

[93] X. Ren, M. Ren, Y. Lei, 以及 J. Ming，“重新审视优化在二进制差异比对工具中的韧性声明：来自 LLVM peephole 优化分析的见解，” *ACM* 软件工程学报，2025。

[94] R. W. Scheifler，《结构化编程语言内联替换分析》，《*ACM*通讯》，1977年。

[95] scikit-learn，“重采样”，2023年，https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html。

[96] P. Shirani, L. Collard, B. Agba, B. Lebel, M. Debbabi, L. Wang和A. Hanna，《Binarm：针对智能电子设备固件映像中漏洞的可扩展且高效的检测》，载于入侵与恶意软件检测和漏洞评估，2018年。

[97] S. P. E. C.，《SPEC CPU2006文档》，2024年，https://www.spec.org/cpu2006/docs/。

[98] T. Theodoridis, T. Grosser, and Z. Su，“理解并利用最优函数内联，”载于第 *27* 届 *ACM* 编程语言与操作系统架构支持国际会议，2022年。

[99] Valgrind，“Valgrind 3.21.0”，2023年，https://github.com/rantoniello/val grind.[100] H. Wang、W. Qu、G. Katz、W. Zhu、Z. Gao、H. Qiu、J. Zhuge 和 C. Zhang，“Jtrans：用于二进制代码相似性检测的跳转感知 Transformer”，载于第31届ACM SIGSOFT软件测试与分析国际研讨会论文集，2022年。[101] S. Wang、T. Liu 和 L. Tan，“自动学习用于缺陷预测的语义特征”，载于第38届国际软件工程会议论文集，2016年。[102] Y. Wu、D. Zou、S. Dou、W. Yang、D. Xu 和 H. Jin，“Vulcnn：一种受图像启发的可扩展漏洞检测系统”，载于第44届国际软件工程会议论文集，2022年。[103] X. Xu、Q. Zheng、Z. Yan、M. Fan、A. Jia、Z. Zhou、H. Wang 和 T. Liu，“Patchdiscovery：基于关键基本块的补丁存在性测试，用于识别二进制漏洞”，IEEE软件工程汇刊，2023年。[104] X. Xu、C. Liu、Q. Feng、H. Yin、L. Song 和 D. Song，“基于神经网络的图嵌入，用于跨平台二进制代码相似性检测”，载于2017年ACM SIGSAC计算机与通信安全大会论文集，2017年。[105] H. Xue、S. Sun、G. Venkataramani 和 T. Lan，“基于机器学习的程序二进制文件分析：一项全面研究”，IEEE Access，2019年。[106] K. Yakdan、S. Dechand、E. Gerhards-Padilla 和 M. Smith，“帮助 Johnny 分析恶意软件：一种经过易用性优化的反编译器及恶意软件分析用户研究”，载于2016年IEEE安全与隐私研讨会，2016年。[107] K. Yakdan、S. Eschweiler、E. Gerhards-Padilla 和 M. Smith，“不再使用 goto：通过与模式无关的控制流结构化和语义保持变换进行反编译”。网络与分布式系统安全研讨会论文集，2015年。[108] I. Yun、S. Lee、M. Xu、Y. Jang 和 T. Kim，“Qsym：专为混合模糊测试量身打造的实用具体符号执行引擎”，载于第27届USENIX 安全研讨会，2018年。[109] Y. Zhang、M. Sirlanci、R. Wang 和 Z. Lin，“当编译器优化遇上符号执行：一项实证研究”，载于2024年ACM SIGSAC计算机与通信安全会议论文集，2024年。[110] F. Zuo、X. Li、P. Young、L. Luo、Q. Zeng 和 Z. Zhang，“受神经机器翻译启发的函数对之外的二进制代码相似性比较”，载于网络与分布式系统安全研讨会论文集，2019年。

附录 A 补充实验与分析

A. 函数内联的 *DWARF* 信息

DWARF 中与函数内联相关的调试信息字段 [19]。

Name	Type	描述
DW_TAG_内联子例程	T	（子）例程的特定内联实例
DW_AT_抽象起源	A	指向内联子程序 DIE 的指针
DW_AT_内联	A	描述内联行为的常量
DW_AT_decl_file	A	包含源声明的文件
DW_AT_low_pc	A	机器码的低地址
DW_AT_high_pc	A	机器码的高地址
DW_AT_call_file	A	包含内联子例程调用的文件
DW_AT_call_line	A	内联子例程调用的行号
DW_AT_call_column	A	内联子例程调用的列位置

B. 漏洞检测中的代码语义

表 IX 展示了章节 VI-A5中描述的 CVE 编号及其对应的易受攻击函数。例如，使用 Trex [85], 我们发现，`BN_dec2bn`的相似度排名在内联到`BN_asc2bn`后有所提升：例如，1，044 → 14，这表明模型能够识别代码语义。因此，识别内联函数中的漏洞语义（即，包含易受攻击函数的函数）是有助于检测的良好迹象。然而，根据我们的观察，不同代码语义的混合可能会降低模型的性能，而额外优化和嵌套内联则可能造成显著影响。

表 IX：我们在章节 VI-A5 中探讨的易受攻击函数的具体名称及其 CVE 编号。

	CVE	查询函数
Netgear R7000	CVE-2016-2182	BN_bn2dec
	CVE-2019-1563	CMS_解密
	CVE-2016-6303	MDC2_更新
	CVE-2019-1563	PKCS7_数据解码
TP-Link Deco-M4	CVE-2016-2182	BN_bn2dec
	CVE-2016-0797	BN_dec2bn
	CVE-2016-0797	BN_hex2bn
	CVE-2019-1563	CMS_decrypt
	CVE-2016-2105	EVP_EncodeUpdate
	CVE-2019-1563	PKCS7_dataDecode
	CVE-2016-0798	SRP_VBASE_获取_按_用户
	CVE-2016-2176	X509_NAME_单行

*C*语言．使用内联函数进行增强训练

实验与结果。 我们针对恶意软件检测任务（T3）采用包含内联函数的增强训练，作为一种潜在的防御措施。我们生成新的极端内联变体，并将其添加到训练数据集中；该数据集包含 159 个良性样本、50 个 Mirai 样本和 25 个 Gafgyt 样本。请注意，为确保测试时遇到未见过的模式，我们采用了不同的极端内联数据增强方案（表 XI）。表 XII中的初步结果表明了两点。第一，增强训练后，CNN 和 DNN 等神经网络的模型性能（例如，F1分数）有所提升。第二，逻辑回归和随机森林等非神经网络模型并未从增强训练中受益。

表 X：Binkit 中可执行二进制文件的静态特征描述 [53]。36 个特征（1-36）描述指令级特征，20 个（37-56）和 6 个（57-62）特征分别与控制流图和调用图相关。我们分析了极端内联对每个静态特征的影响。

索引	特征描述
1	指令数量
2	指令平均数量
3	未知指令数量
4	未知指令的平均值
5	绝对算术指令数量
6	绝对算术指令的平均值
7	算术指令数量
8	算术指令的平均值
9	比较指令数量
10	比较指令的平均数量
11	绝对控制转移指令的数量
12	绝对控制转移指令的平均数量
13	条件控制转移指令的数量
14	条件控制转移指令的平均数量
15	组跳转指令的数量
16	组跳转指令的平均数量
17	绝对数据传送指令数量
18	绝对数据传送指令的平均数量
19	数据传送指令数量
20	数据传送指令的平均数量
21	控制转移指令数量
22	控制转移指令的平均数量
23	组调用指令数量
24	组调用指令的平均数量
25	组返回指令数量
26	组返回指令的平均数量
27	杂项指令数量
28	杂项指令平均数量
29	移位指令数量
30	移位指令平均数量
31	逻辑指令数量
32	逻辑指令平均数量
33	位标志指令数量
34	位标志指令平均数量
35	浮点指令数量
36	浮点指令平均数量
37	控制流图大小
38	控制流图中的循环数量
39	控制流图中的过程间循环数量
40	控制流图中的强连通分量数量
41	控制流图中的回边数量
42	控制流图中的广度优先搜索边数量
43	控制流图的最大宽度
44	控制流图的最大深度
45	控制流图中所有循环大小之和
46	控制流图中所有过程间循环大小之和
47	控制流图中所有强连通分量大小之和
48	控制流图中循环的平均值大小
49	控制流图中过程间循环的平均值大小
50	控制流图中强连通分量的平均值大小
51	控制流图中的入边数量
52	控制流图中的出边数量
53	控制流图中的边总数（入度 + 出度）
54	控制流图中入边的平均数量
55	控制流图中出边的平均数量
56	控制流图中的边数（入度 + 出度）
57	调用图中的调用者函数数量
58	调用图中的被调用函数数量
59	调用图中导入的被调用函数数量
60	调用图中的传入调用数量
61	调用图中的传出调用数量
62	调用图中的导入调用数量

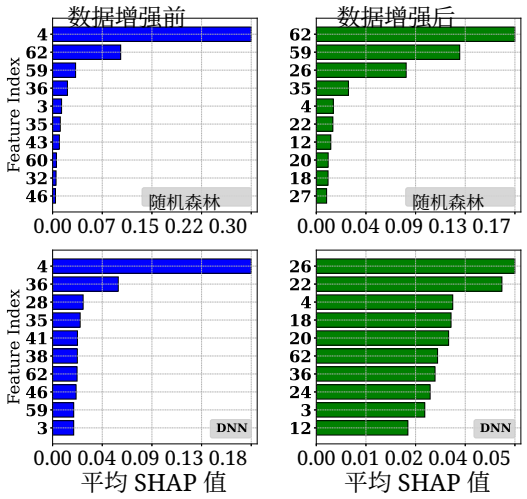


图 13：基于传统机器学习模型（表 X）和神经网络模型（例如 随机森林）及例如 深度神经网络（DNN）在极端内联数据增强前后，按 SHAP 绝对值均值排序的影响力最大的前 10 个特征。SHAP 值越高，表示该特征对模型决策的贡献越大。尽管不同模型的基线特征排名有所差异，但与随机森林相比，数据增强对 DNN 特征重要性的重塑更为显著，这表明神经架构能更好地适应极端内联。

使用 SHAP 进行深入分析。为评估数据增强如何改变模型行为，我们采用 Shapley 加性解释（SHAP） [72]，通过十次迭代衡量特征贡献，并比较数据增强前后的 SHAP 值绝对值均值。值得注意的是，我们尝试理解每个特征如何影响模型的决策，因为可解释人工智能存在保真度问题：也就是说，解释只是对决策过程的近似，并不能完美反映该过程。图 13 展示了两个代表性模型的前 10 个贡献特征：非神经网络模型中的随机森林，以及神经网络模型中的 DNN。极端内联会影响整个特征空间，不仅改变指令频率，还会改变控制流和调用图特征。我们的研究结果表明，增强训练后，与传统模型（随机森林）相比，神经网络模型（DNN）的特征重要性变化更为显著，这表明数据增强对神经架构的影响更大。有趣的是，在所有未经增强训练的模型中，决策都由特征 4（即 未知指令的数量）主导。进一步分析显示，`rep stosq, qword ptr [rdi], rax`经常出现在恶意软件样本中，这可能解释了该特征对检测的显著贡献。

附录B 工件附录

A. 摘要

函数内联是一种常见的编译器优化，它通过用函数体替换函数调用来提升性能，但也可能改变用于基于机器学习的安全分析的二进制文件的结构属性。为

表 XI：各任务（T1–T5）的极端内联方案汇总。

Type	软件包或应用程序	极端内联方案设置			
		T1–2、T5	T3	T4	T3*（数据增强变体）
良性	coreutils	-Oz -inline-threshold=2225	-O3 -flto=full；-O3 -flto=full -inline-threshold=200000	-	-O3 -inline-threshold=2225
	binutils	-O3	-	-	-
	diffutils	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	查找工具集	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	OpenSSL	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	lvm2	-O3 -flto=full	-	-	-
	gsl	-Oz -inline-threshold=2225	-O3 -flto=full	-	-O3 -inline-threshold=2225
	Valgrind	-Oz -inline-threshold=2225	-	-	-O3 -inline-threshold=2225
	Open MPI	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	PuTTY	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
	nginx	-Oz -inline-threshold=2225	-O3 -flto=full	-	-O3 -inline-threshold=2225
	lighttpd	-O3 -flto=full	-Oz -inline-threshold=2225	-	-O3 -inline-threshold=2225
恶意软件	Mirai	-	-O(1|2|3|s|z) –内联-阈值=(225|500225)		-O(1|2|3|s|z) –内联-阈值=(225|2000)
	Gafgyt	-	-O(1|2|3|s|z) –内联-阈值=(225|500225) -flto=full		-O(1|2|3|s|z) –内联-阈值=(225|2000) -flto=full

表 XII：使用新生成的极端内联变体进行恶意软件检测任务（T3）的性能比较。有趣的是，扩充训练集能够提升神经网络模型的性能（例如，卷积神经网络、深度神经网络），而非神经网络模型（例如，逻辑回归、随机森林、CatBoost）受益甚微。

训练设置	机器学习模型	真实环境中的恶意软件				具有极端内联的恶意软件			
		准确率	精确率	召回率	F1	准确率	精确率	召回率	F1
数据增强前	逻辑回归	**0.99 ± 0.00**	**0.99 ± 0.00**	0.99 ± 0.01	**0.99 ± 0.00**	0.60 ± 0.02	0.64 ± 0.03	0.47 ± 0.05	0.54 ± 0.03
	随机森林	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	0.78 ± 0.01	0.70 ± 0.02	0.95 ± 0.03	0.81 ± 0.02
	CatBoost	0.98 ± 0.00	**0.99 ± 0.00**	0.97 ± 0.01	0.98 ± 0.00	**0.78 ± 0.01**	0.70 ± 0.01	**0.97 ± 0.01**	**0.82 ± 0.01**
	KNN	**0.99 ± 0.00**	**0.99 ± 0.00**	0.99 ± 0.01	**0.99 ± 0.00**	0.68 ± 0.04	0.69 ± 0.03	0.65 ± 0.09	0.67 ± 0.06
	平均值（非神经网络）	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.99 ± 0.00	0.71 ± 0.08	0.68 ± 0.03	0.76 ± 0.20	0.71 ± 0.11
	CNN	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.75 ± 0.05	**0.76 ± 0.09**	0.77 ± 0.10	0.76 ± 0.04
	DNN	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	**0.99 ± 0.00**	0.71 ± 0.04	0.73 ± 0.03	0.67 ± 0.10	0.70 ± 0.06
	平均值（神经）	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.99 ± 0.01	0.73 ± 0.03	0.75 ± 0.02	0.72 ± 0.05	0.73 ± 0.03
数据增强后	逻辑回归	0.96 ± 0.01	**0.99 ± 0.01**	0.94 ± 0.02	0.96 ± 0.01	0.64 ± 0.07	0.98 ± 0.02	0.28 ± 0.14	0.42 ± 0.17
	随机森林	0.98 ± 0.01	**0.99 ± 0.01**	0.97 ± 0.01	0.98 ± 0.01	0.69 ± 0.02	0.90 ± 0.00	0.38 ± 0.03	0.50 ± 0.02
	CatBoost	0.98 ± 0.01	**0.99 ± 0.01**	0.96 ± 0.02	0.98 ± 0.01	**0.84 ± 0.02**	0.99 ± 0.01	**0.69 ± 0.03**	**0.81 ± 0.02**
	KNN	**0.99 ± 0.00**	**0.99 ± 0.01**	**0.99 ± 0.01**	**0.99 ± 0.00**	0.81 ± 0.01	**0.99 ± 0.00**	0.62 ± 0.03	0.77 ± 0.02
	平均值（非神经网络）	0.98 ± 0.01	0.99 ± 0.00	0.97 ± 0.02	0.98 ± 0.01	0.75 ± 0.09	0.97 ± 0.05	0.49 ± 0.18	0.63 ± 0.18
	CNN	**0.99 ± 0.00**	**0.99 ± 0.01**	0.98 ± 0.01	**0.99 ± 0.00**	0.84 ± 0.02	**0.99 ± 0.00**	0.68 ± 0.05	0.81 ± 0.03
	DNN	**0.99 ± 0.00**	**0.99 ± 0.01**	0.98 ± 0.01	**0.99 ± 0.00**	0.81 ± 0.01	**0.99 ± 0.00**	0.62 ± 0.03	0.77 ± 0.02
	平均值（神经）	0.99 ± 0.00	0.99 ± 0.01	0.98 ± 0.00	0.99 ± 0.00	0.83 ± 0.02	0.99 ± 0.00	0.65 ± 0.04	0.79 ± 0.03

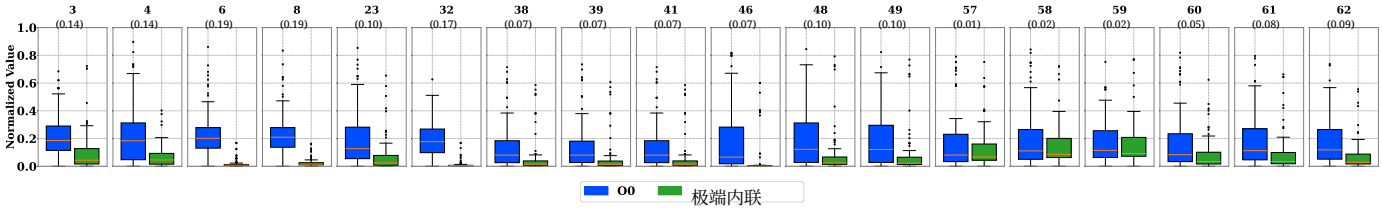


图 14：少量内联（-O0）与极端内联（例如，编译器选项：-O3 -flto=full -inline-threshold=200000）下选择性静态特征的比较。这些特征包括指令（特征 3、4、6、8、23、32）、控制流图（特征 38、39、41、46、48、49）和调用图（特征 57、58、59、60、61、62）。上方的数字表示特征及两组均值之间的差距（括号内）（附录中的表 X）。该示例展示了均值之间的显著差距，这可能会威胁基于机器学习的模型的鲁棒性。请注意，为便于简洁比较，我们对所有数值进行了归一化。详情请参见第 VI-B4 节。

了研究这一现象，我们系统地考察了内联对基于机器学习的二进制分析的安全影响。该软件工件与我们的论文配套，提供可直接使用的数据集和脚本，以验证论文中呈现的主要结果。其中包含用于探索与内联相关的编译器标志的源代码和构建脚本，旨在

极端内联、修改后的特征提取工具、可运行的机器学习模型，以及用于重新生成论文所报告主要结果的分析脚本。这些资源使审查者能够验证我们的研究发现，并进一步考察极端内联对基于机器学习的安全任务的影响，包括二进制代码相似性检测（T1）、函数名称预测

（T2）、恶意软件检测（T3）、恶意软件家族预测（T4）和漏洞检测（T5）。

B. 描述与要求

所提供的软件工件包含源代码、构建脚本、精选数据集和 Python 分析工具，用于复现论文中呈现的实验结果。该软件工件分为四个主要组成部分：

- **数据集构建**。包含用于探索不同配置下与内联相关的编译器标志的源代码和构建脚本，以实现极端内联。尽管为完整起见提供了完整的数据集构建流程，但也包含可直接使用的数据集，便于快速验证，无需重新编译。

- **特征提取**。 包含经过修改的*TikNib*版本，并配置为提取研究中任务 T3 和 T4 使用的 static 特征和语义特征。

- **机器学习模型**。 提供可运行的机器学习模型及相关数据集，用于评估五项二进制分析安全任务（T1–T5）和 18 个代表性模型在极端内联下的鲁棒性。

- **主要结果与图表**。 包含 CSV 结果文件和 Python 脚本，用于重新生成论文报告的主要分析结果。此外，还提供验证脚本，可直接使用随附的数据集复现这些结果。

范围。 此软件工件旨在验证已报告的发现，而非从头重新生成数据集。对于 T3 和 T4，由于机器学习模型和蒙特卡洛交叉验证具有随机性，结果可能与论文中报告的结果略有差异；不过，整体趋势仍然一致。

安全、隐私和伦理问题。所提供的脚本完全使用开源数据，不会执行任何破坏性或涉及隐私的操作。由于许可限制，不会共享由专有的 SPEC CPU2006 基准测试生成的二进制文件。因此，不存在安全、隐私或伦理方面的问题。

1) 获取方式： 该软件工件已在 Zenodo 上开源提供 [8]。

2) 硬件依赖项： 实验在一台工作站上进行，该工作站运行 Ubuntu 20.04（64 位），配备 Intel Xeon Gold 5218R @ 3.00 GHz CPU、512 GB RAM 和两块 NVIDIA RTX A6000 GPU。

3) 软件依赖项： 该软件工件需要以下软件环境：LLVM/Clang 14.0.0、Python 3.8 或更高版本（含 pip $\geq$ 23.0 和 Conda $\geq$ 23.7.4）、Docker 20.10.22 或更高版本，以及 IDA Pro 8.2 或 8.3。

*4)*基准测试：实验使用了多个数据集（见表 V）以及用于所评估的机器学习辅助二进制分析任务的基准实现。具体而言，对于 T1 和 T5，我们使用二进制代码相似性检测基准 [74]，其中包含 Asm2Vec、Gemini、Trex、SAFE 和 BinShot 等模型。对于 T2，我们使用自己的 LLVM 编译数据集重新训练 AsmDepictor [55]。对于与恶意软件相关的任务（T3 和 T4），我们从不同来源整理了数据集。对于 T3，我们使用了从 VirusShare 收集的样本 [77]。对于 T4，我们根据 Alrawi等人 [7], 整理了数据集，其软件工件可在 [9]公开获取。相应的模型包括使用 Scikit-learn 构建的四种传统机器学习分类器 [25]，以及根据 Abusnaina 等人重新实现的深度学习模型等人 [3]。对于 T5，我们使用二进制代码相似性检测基准以及公开可用的易受攻击固件语料库 [16]。

C. 软件工件安装与配置

1）安装： 要安装软件工件，请从提供的链接下载代码库，并进入项目根目录。所有必需的依赖项均列于第 B-B节，可使用标准包管理工具进行安装，例如 `pip` 或 `conda`。每个目录都是独立的，并包含`START_EVALUATION.md` 文件和配置脚本，用于指导安装和执行。此外，也提供了预配置的 Docker 镜像。

对于任务 T1、T2 和 T5，请创建提供的 Conda 环境：

```
cd ml-model/t1_t2_t5
conda env create -f environment.yml
conda activate inline_ae
```

对于任务 T3 和 T4，Conda 环境为可选；根目录中提供了依赖项：

```
pip install -r requirements.txt
```

2）基本测试： 要验证软件工件是否已正确安装且运行正常，请执行任意一个提供的任务脚本。例如，对于 T1：

```
cd ml-model/t1_t2_t5/T1_bcsd
bash run.sh
```

D. 实验工作流程

该软件工件由多个模块化组件组成，每个组件对应实验工作流程中的一个阶段。整体流程可概括如下：(i) 使用所提供的编译扫描脚本，在多种内联配置下构建数据集；(ii) 使用修改后的 `TikNib` 框架 [54]进行特征提取；(iii) 针对五项安全任务（T1–T5）训练并评估机器学习模型；(iv) 分析并可视化内联对二进制文件及其相关统计特征的影响，这些结果可使用所提供的分析脚本重新生成。

E. 主要主张

- (C1)：该软件工件提供了完整的源代码、构建脚本和精选数据集，用于评估多种编译器配置下函数内联的影响。这些资源有助于透明地验证数据集生成过程，并复现第 VI 章节所述的实验设置。

- (C2)：所提供的机器学习模型和特征提取工具复现了五项基于机器学习的安全任务（T1–T5）中的所有关键评估结果，支持论文关于模型鲁棒性对极端内联敏感性的

[8]https://doi.org/10.5281/zenodo.17759528

[9]https://badthings.info/#bins

研究发现，详见章节VI-A。

• (C3)：所包含的分析脚本可重新生成所有主要定量结果和图表（图 7–14），展示不同配置下的内联趋势，以及与非内联基线相比，极端内联对统计特征的影响，详见第VI-B节。

F. 评估

•(E1)：**数据集构建与编译扫描**[15 人分钟 + 数小时计算时间]运行提供的数据集构建脚本，以在多种内联配置下重现编译扫描。构建框架会系统地探索 config.yaml 中定义的优化选项和隐藏内联标志。根据所选项目和增量，每次数据集构建可能需要在多核系统上运行数小时。为进行验证，审阅者可以执行精简配置（例如，仅选择部分项目或阈值），以确认功能正常，而无需进行全面重新编译。

• (E2)：**特征提取与可视化** [10人分钟 + 适量计算时间] 运行修改后的 TikNib 流水线，提取二进制级特征并生成汇总统计或可视化结果。此步骤会生成用于训练 T3–T4 的特征表示和特征失真（图 14）。

•(E3)：**跨任务模型评估** [10 人分钟 + 计算时间（视任务而定）] 使用 ml-model/ 目录中提供的脚本，评估机器学习模型对极端内联的鲁棒性。各任务的大致运行时间如下：T1 和 T5 约 2 小时，T2 约 1 小时，T3 约 10 分钟，T4 约 20 分钟。

成功完成实验（E1）–（E3）即可复现论文的主要发现，验证所提供软件工件的可复现性和完整性。

G. 可复用性说明

所提供的软件工件旨在供未来有关极端内联、二进制分析和基于机器学习的安全研究复用和扩展。每个组件——数据集构建、特征提取、模型评估和分析——均采用模块化设计，且可独立执行。用户可通过调整配置文件（例如，config.yaml），并遵循随附 README.md 文件中的详细说明，复现实验，或将其应用于新的数据集、编译器或模型架构。为方便起见，我们还提供了预构建的 Docker 镜像，用于复现主要实验。最后，模块化设计还使研究人员能够通过我们的调优策略自动探索更极端的内联程度，为未来开展对抗性代码转换研究提供了方向。