arXiv:2601.12270v1 [cs.CR] 18 Jan 2026

SplittingSecrets：一种用于防止数据内存依赖预取器侧信道的基于编译器的防御

Reshabh K Sharma DanGrossman David Kohlbrenner华盛顿大学 华盛顿大学 华盛顿大学

摘要

传统侧信道会利用秘密数据作为不安全指令的输入、用于内存访问，或用于控制流决策。常数时间编程通过限制此类代码模式，已被广泛采用，以防御这些漏洞。然而，苹果公司、英特尔和 ARM CPU 中采用数据内存依赖预取器（DMP）这一形式的新型硬件优化表明，这些防御措施并不足够。与传统预取器不同，这些预取器会利用内存内容以及先前访问轨迹来确定预取目标。已有研究表明，攻击者可以滥用此类预取器发动攻击，泄露静态数据；这些数据即使在推测执行中也从未被程序以不安全的方式使用。

为此，本文介绍了*SplittingSecrets*，这是一种基于编译器的工具，可加固软件库，防范由数据内存依赖预取器引发的侧信道。*SplittingSecrets*的方法无需推理不同数据内存依赖预取器复杂的内部机制，而是依赖所有此类预取器的一项关键特性：要发生泄漏，数据必须类似于地址。为防止秘密数据泄漏，*SplittingSecrets* 会转换内存操作，确保秘密数据不会以类似地址的形式存储在内存中，从而避免这些秘密数据触发 DMP 激活。*SplittingSe-crets* 并非完全禁用 DMP，而是可以完全通过软件，仅针对特定秘密数据提供定向加固。

我们使用 LLVM 实现了*SplittingSecrets* ，支持 AArch64 架构的源代码级内存操作以及由编译器后端生成的内存操作。我们分析了在 Apple M 系列 CPU 上构建时，使用流行密码学库 libsodium 中的常用原语来保护秘密数据免受 DMP 诱发攻击所产生的性能开销。

1 引言

通用 CPU 对性能的追求催生了巧妙而令人惊讶的优化。其中许多优化都伴随着显著的权衡，尤其是可能通过微架构侧信道导致意外的信息泄漏 [44, 53]，涉及缓存 [43, 58]、TLB [24,25], 、分支预测器 [4,20],、片上互连、内存管理单元、推测执行 [22, 39], 等。

大多数微架构侧信道会泄露使用中的数据 [51] ——这些数据要么被推测性地 [39, 42] 传递给不安全指令，要么被非推测性地 [43, 58] 传递给不安全指令。当秘密数据影响缓存中包含哪些地址 [29, 30, 46]，和/或被用作操作数会导致执行时随操作数变化的指令的输入时，这类漏洞就会泄露秘密数据 [5, 27]。常数时间（CT）编程试图通过确保秘密数据从不用于已知会泄露其操作数某个函数的指令，来防止此类泄露。

不同于使用中数据侧信道，数据静态存储侧信道发生在程序序列泄露了未被计算的数据时，即使这些数据未被推测性地计算也是如此。这方面最显著的例子是近期利用数据内存依赖型预取器（数据内存依赖预取器） [11, 54, 55] 的研究。这类预取器不仅依赖程序之前执行的内存访问，还依赖内存内容 [6,50,59] 来获取额外数据。已知 Apple A 系列和 M 系列处理器以及英特尔第 13+ 代 CPU 部署了此类预取器。我们将在第2节讨论数据内存依赖型预取器的设计、本文所考察的具体系列，以及已有研究如何证明它们能够泄露秘密数据。

我们提出的防御利用了任何 DMP 侧信道的一项关键要求：预取必须导致页表遍历至少部分成功 [55]。如果内存中的所有秘密数据都预先填入某个值，确保页表遍历无法成功（甚至确保预取不会启动），就不会有秘密数据泄露。这可能是因为预取器会通过启发式检查判断某个地址是否可能是有效地址，也可能是因为该前缀属于未映射（或无法映射）的内存区域。

为实现这一目标，我们重新设计秘密数据在内存中的存储方式。每个秘密数据被拆分为 32 位数据块，并将每个数据块放置在 64 位字的低半部分，而高半部分则使用一个选定的前缀，该前缀不满足所考虑 DMP 的要求。因此，每个对齐的值

只要在内存中包含秘密数据，就会无法完成地址转换，甚至根本不会尝试进行预取。这也要求重新设计加载操作，因为它们需要重构拆分后的值。我们的方法无需对数据内存依赖预取器的复杂细节进行建模，只需更改所选前缀，便可直接适配所有已知的、已部署的数据内存依赖预取器。我们认为，未来的数据内存依赖预取器实现也将采用类似结构，因此我们的防御同样适用于它们。

虽然厂商引入了一些硬件配置机制（ARM 数据独立时序（DIT）和英特尔数据操作数独立时序（DOIT）），可在部分处理器上禁用 DMP 行为，但这些机制并非始终能在用户空间中使用，也无法在所有处理器上始终禁用 DMP 激活。与以往工作一致， [21] 我们为敏感软件提供了一种纯软件防御方案，可适用于所有已知支持 DMP 的 CPU，且无需任何特权。此外，鉴于数据内存依赖预取器（DMP）的设计目的是提升性能，理想的方案应具有针对性，仅保护密码学秘密，而不必在整个 CPU 上彻底禁用该优化，也不依赖厂商对威胁严重性的判断。

SplittingSecrets 是我们的基于编译器的工具，可自动执行这种内存变换方法，保护秘密数据免受由数据内存依赖预取器（DMP）引发的侧信道攻击。我们的方法仅作用于内存操作级别，无需更改其他操作。

我们的主要贡献如下：

- 首个基于编译器（且支持多平台）的缓解策略的提出与设计，用于防御基于地址的DMP 静态数据侧信道。
- 针对 ARM64，在 LLVM [40] 中实现我们的方法，面向密码学库。
- 使用 libsodium [19]密码学库，分析我们的内存表示转换所引入的性能和内存开销。

我们的实现以及分析中使用的测试和微基准测试可在以下地址获取：https://github.com/orgs/dmp-mitigation。

2 背景

2.1 传统预取器

传统预取器 [35] 旨在通过预测未来的数据需求，并在 CPU 直接请求数据之前将其取入缓存，从而提高 CPU 内存访问的效率 [56]。

大量已有研究表明，传统预取器同样可能暴露漏洞。通常，这些攻击会利用由预取器引发的未来缓存访问的时序差异，其中预取器的行为由依赖秘密的行为所导致。例如，研究表明，步长预取器可用于放大同一核心上的缓存攻击 [9,52]，或不太常见地，用于放大跨核心缓存攻击 [13]。传统预取器还可以充当隐蔽信道机制 [14,18]。

几乎所有 传统预取器的一个关键特性是，它们只根据程序访问的地址轨迹运行。因此，传统预取器攻击只会针对违反现有常数时间编程规则的代码模式奏效。

2.2 数据内存依赖预取器

数据内存依赖预取器（DMP）会结合先前的内存访问模式以及这些访问返回的数据，从而更准确地推测未来的内存请求 [6,8,10,15,34,38,50,59]。

这类预取器可处理两种数据：地址和索引。基于索引的预取器假设内存中包含数组索引或偏移量，并且预取器需要推断基址，同时计算最终可能的目标地址。基于地址的预取器则更简单，它假设内存中包含虚拟地址，可直接进行预取，无需额外计算。

CPU 中所有已知的已部署 DMP 示例都是基于地址的预取器；除非另有说明，我们用“DMP”指代基于地址的 DMP。

2.2.1 DMP 的安全影响

与经典预取器不同，DMP 会使用内存中的值来决定预取什么，因此能够对遵循当前常数时间原则编写的软件发起新的攻击。

当 DMP 选择预取由秘密（例如密码学密钥值）导出的目标地址时，攻击者便可利用它。预取完成后，攻击者可以使用标准的缓存侧信道技术，例如 PRIME+PROBE [46]，来确定预取了哪些缓存行，并（部分）获知用于生成预取地址的值。

这与经典预取器的行为不同：经典预取器不会基于秘密作出预取决策，除非程序流程已经导致根据秘密进行内存访问。

近期研究 [11,54,55] 表明，这种行为能够从 RSA、DH 密钥交换和后量子 ML-KEM 的实现中恢复密钥，还能攻击地址空间布局随机化（ASLR）并恢复内核内存。这些密码学攻击假设本地攻击者能够通过标准侧信道测量缓存状态，并能够向目标程序发送选择密文。选择密文经过精心构造，使受害程序执行的某个内存操作根据密钥的特定位存储值X或Y。正确构造时，X会通过 DMP 有效性检查，并触发对虚拟地址X的预取，而Y则无法通过此类检查。关键在于，这个中间状态在语义上并非指针，不用于推导加载的内存地址，并且

若没有 DMP，也不会导致信息泄漏。本研究针对这些攻击。

2.2.2 防御数据内存依赖预取器

厂商一直试图通过提供配置位来提升对微架构侧信道攻击的整体防护能力，这些配置位可保证指令时序（或其子集）不依赖于操作数。

自 ARMv8a 起，ARM 引入了数据无关时序（DIT）[1] 标准，任何异常级别都可以在运行时设置 PSTATE.DIT 配置位来启用 DIT。DIT 并未明确说明是否涵盖 DMP 等行为。也可以通过未公开的配置位禁用 Apple 的 DMP [12, 33], , Asahi Linux 支持通过此方式全局禁用 DMP。

英特尔的数据操作数无关时序（DOIT）标准 [37]（2022年）与 ARM 的 DIT 类似。与 DIT 不同，DOIT 只能由内核启用，但会明确影响数据内存依赖预取器（DMP）（英特尔称之为数据相关预取器 [36]）。

这些机制并不一致且含糊不清，对于几乎无法控制其执行环境的软件而言，并非有效的缓解方案。ARM 的 DIT 适用于 Apple M3，但不适用于 M1/M2 [11, 54]。据我们所知，没有操作系统支持用户空间请求启用 DOIT。

此前有两项研究勾勒出了与我们系统基于相同直觉的防御方案，但并未对其进行开发或评估。Augury [54] 指出，只要将秘密数据存储在每个 64 位对齐区域的低位中，就应该能够防御 Apple DMP。类似地，ZebraFix [47] 提出修改其用于缓解 DMP 侧信道的防御方案，采用数据交错。

2.2.3 现实环境中的数据内存依赖预取器

所有已知的已部署数据内存依赖预取器都以地址而非索引为操作对象，但在是否跟踪并响应先前的预取准确率、维护何种历史记录以及采用何种启发式方法等方面各不相同。

我们的缓解策略有意限定在稳定且跨代际的特征上，具体而言，就是检查对齐的 64 位字以及是否需要部分页表遍历。如果未来的设计接受更多高位模式，我们可以轮换前缀，或缩小秘密片段的大小（例如，缩减为低 16 位片段），使重构出的字仍不像合理的地址。

3 威胁模型

我们的工作旨在加固这样一些软件库：它们在没有 DMP 的系统上运行是安全的，但在配备 DMP 的系统上运行则不安全。我们特别关注密码学软件：这类软件被认为采用了常数时间编程技术并实现了适当的推测攻击防御，但并未意识到 DMP 带来的威胁。

我们假设目标程序运行在配备基于地址的 DMP 的 64 位系统上，类似于 Apple M 系列 DMP 或英特尔第 13 代 DDP。这两种数据内存依赖预取器都会检查内存以识别可能的指针，并对其解引用，但它们在何时以及如何进行分析方面采用了不同的方法。为了涵盖所有类型的基于地址的预取器，我们考虑一种激进的 DMP 实现：DMP 可以随时检查任意一段内存，而且被检查的目标值不必是有效的虚拟地址。我们假设，作为一种过度近似，只有当所考虑的值导致页表遍历部分成功时，DMP 才会泄漏信息。

我们的对手既可以利用经典缓存侧信道（无论是否使用共享内存） [30, 46], , 也可以利用页表遍历侧信道 [26, 55]。这意味着，在我们的威胁模型中，DMP 尝试预取的任何数据都会泄露。假定对手能够在任意时间触发这一 DMP 过程，并处理任意内存位置。

这一模型与苹果公司 M 系列 DMP 的行为非常接近：该 DMP 会对缓存行中通过启发式检查的任何值进行解引用。利用该 DMP 构建攻击的先前工作 [11,54] 也表明，类似的威胁模型是恰当的。

4 设计

所有启用 DMP 的侧信道，其核心都是允许秘密数据位流入虚拟地址转换，并（可选地）在尝试预取时更新缓存状态。我们的防御会重写内存存储操作和加载，使秘密数据仅存在于 64 位对齐内存区域的低半部分；由于采用了所选的高半部分前缀，这些数据绝不会用于页表遍历。

具体的前缀可以是虚拟内存中的未映射区域，也可以是无效的地址前缀，不会触发预取。例如，在 ARM64 上采用标准的 48 位虚拟寻址模式时，使用 0x00FF0000 作为前缀将不会尝试进行预取。

SplittingSecrets 是一种基于编译器的变换防御，可防范数据内存依赖预取器引入的侧信道，确保秘密数据不会进入页表遍历。例如，请参见图 3 中的代码，该代码展示了一个恒定时间程序，为防范 DMP 攻击，需要对其进行变换。

为实现这一点，我们将秘密数据拆分并填充为 32 位片段，每个片段单独存储，并附带一个 32 位固定前缀，以防其看起来像有效地址。对于 64 位数据，此方法会将其拆分为高位和低位两个 32 位值（v_u，v_l），然后将它们分别存储在两个 64 位块中，并附加固定前缀 p，即$p||v_u$ 和$p||v_l$。我们采用这种方法，是因为地址长度为 64 位，而 32 位固定前缀足以使页表遍历失败，甚至阻止预取启动。我们在图2中展示了存储指令的变换。根据体系结构，我们的技术可以通过确定

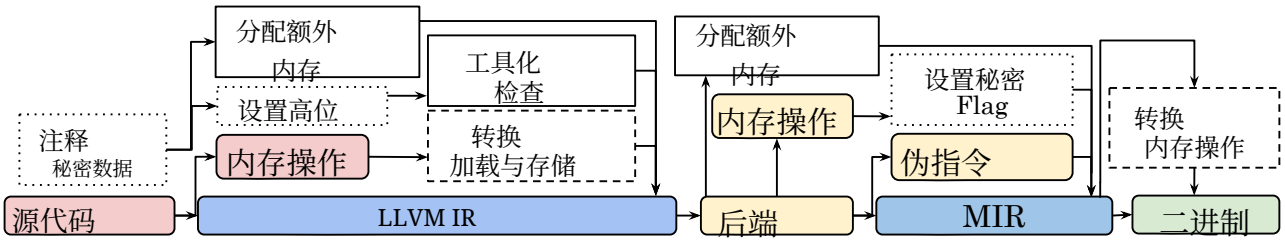

图 1：*SplittingSecrets*的端到端流水线。为标注的秘密数据分配额外内存，并设置地址的高位以进行动态跟踪。在 LLVM IR 层，对内存操作插入检查，并将加载和存储操作转换为用于防御的形式。后端生成的内存操作会被标记，或降级为伪指令，随后由 MIRpass 处理。

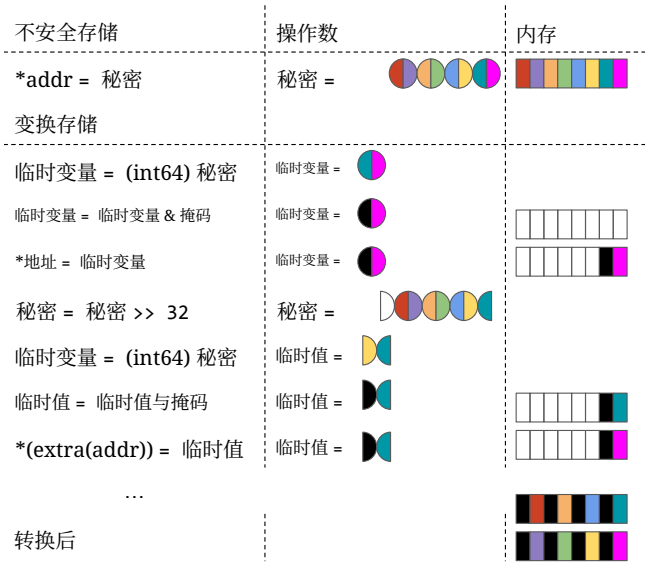

图 2：256 位不安全存储的转换示例。每个圆形代表 64 位。输入的 64 位数据块（例如洋红色和蓝色的半圆）会被拆分，并用 32 位固定常量值填充（黑色半圆）。带填充的下半部分存储在原始内存位置，带填充的上半部分则存储在为秘密数据分配的额外内存中。

前缀的大小进行调整，使其能够导致有效性检查失败，并在页表遍历失败时避免激活 DMP 或泄露数据。

我们会转换所有包含秘密数据的存储操作。由于这些转换在功能上并不等价，因此也必须转换加载操作，才能正确取回已存储的数据。只有内存中的数据需要防范 DMP 引发的侧信道，因此寄存器和 CPU 可以保留未修改形式的秘密数据，不必担心其通过静态数据攻击泄露。因此，我们无需转换任何使用这些数据的非内存操作。

在执行时，我们的防御中有些组件需要支持检查某个特定存储是否为秘密数据、为秘密数据分配额外内存，以及保持一个

```
void ctswap(uint64_tcondition, uint64_
t*a, uint64_t *b): uint64_t val_a =load(a)
uint64_t val_b =load(b) uint64 t xor =
val a ^ val b_ _ _uint
64_t mask = ~(condition-1) xor = 异或 &
mask val a = val a ^异或_ _
store(a, val_a) val b = val b ^异或
_ _store(b, val_b)
```

图 3：需要进行变换的 64 位恒定时间条件交换示例。

跟踪该内存。我们将此额外功能链接到加固二进制文件中，并称之为运行时。

4.1 加载与存储

对于 ≤ 32 位的值，我们的操作很简单：分配 64 位空间，用前缀填充上半部分，并将存储的值放入下 32 位。随后对该地址进行加载时，只需加载下 32 位即可恢复真实值。

处理 64 位存储和更大的分配并非易事，因为这需要将存储的值拆分为多个独立的、带 64 位前缀的组成部分。根据秘密标签，现有的加载和存储操作都需要拆分为两个操作。

对于 64 位存储，我们将存储的值拆分为两个 32 位数据块，如图 4所示。要在加载时恢复该值，我们必须从原始分配和额外分配中加载数据，然后移除前缀并重新组合两半，如图4所示。

对于像数组这样可以逐个元素访问的分配，情况会更加复杂。在这种情况下，我们需要分配额外内存；如果这块区域与原始分配连续，就将值溢出到这块区域中，或者

存储(秘密_地址)：
如果 (isSecret(secret_addr))： addr = unsetBit(secret_addr) **对**每个 4-字节 块，位于 [addr, addr + size)： chunk_addr = isShadow(addr)? heapAddr(addr) : addr*chunk_addr = merge(extractBits(value, chunk),Prefix)

加载(秘密_地址)：

如果 (isSecret(秘密_地址)): addr = unsetBit(秘密_地址) value = 0遍历 [addr, addr + size) 中的每个 4 字节块：chunk_addr = isShadow(addr) ? heapAddr(addr) : addr chunk = extractBits(*chunk_addr) value = insertBits(value, chunk) 返回 value

图 4：秘密数据 ≥ 4 字节的转换后加载和存储操作的伪代码。这些变换假定已根据需要为秘密数据分配了额外内存。对于这两种转换，isSecret() 会检查地址是否设置了高位以将其标记为秘密，unsetBit() 会清除秘密跟踪位，isShadow() 会检查地址是否指向额外分配的内存，heapAddr() 会将常规分配的地址映射到额外内存中的对应地址。对于存储操作，extractBits() 会提取 32 位块，这些块将通过 merge() 与特殊前缀合并。对于加载操作，extractBits() 会提取先前与前缀一起存储的 32 位块，而 insertBits() 会将提取出的块组合成最终值。

跟踪非连续区域的地址。随后处理每个 4 字节块，将其与前缀交替存储到原始分配和额外分配中。

直接采用这种做法，无论额外分配位于何处，都会导致需要对数组的新偏移量进行复杂的额外计算。为避免这种复杂性，我们按数据元素拆分：前 64 位将分散存储在原始分配和额外分配各自的前 64 位中，从而简化偏移量计算。这种方法还需要跟踪额外分配的长度，或其地址和长度。只有在分配大小固定且较小（例如寄存器）时，才可以简单地让数据溢出到相邻的已分配内存区域。

5 实现

在图1中，我们展示了编译流水线：它接收标注了秘密数据的源程序，并生成一个二进制文件，确保秘密数据不会通过 DMP 侧信道泄漏。我们在 LLVM 中将其实现为一个转换遍，该转换遍运行于 LLVM IR，并处理由源程序生成的内存操作。我们还针对 AArch64 后端实现了一个 MIR pass，用于处理编译器为函数调用、通过栈传递参数以及寄存器溢出和寄存器填充而隐式生成的内存操作。我们还在编译器的多个位置进行了修改，以更新调用约定和溢出大小，并支持更新后的溢出大小、保留寄存器和变更后的调用约定。我们的 LLVM 转换遍共使用 1850 行 C 代码编写++ ，运行时则使用 652 行 C 代码编写++。

5.1 源代码级内存存储处理

由源代码直接生成的内存操作会在 LLVM IR 中由 *SplittingSecrets* 进行转换。

5.1.1 标注和跟踪秘密数据

假设所有内容都是秘密数据并据此转换所有内存操作并不可取，因为这些变换会带来开销。因此，为了能够有选择地应用这些变换，*SplittingSecrets* 支持将内存分配标注为秘密数据。

注解机制。开发者通过三种机制在源代码中标注秘密数据：(1) 对于堆上分配，程序员将标准的`malloc/free` 调用替换为*SplittingSecrets*的自定义分配器和释放器包装器（例如，`secret_malloc` 和`secret_free`）。(2) 对于栈上分配，开发者在变量声明中添加属性（例如，`__attribute__((secret))`），将栈变量标记为秘密数据。(3) 对于全局变量，*SplittingSecrets* 提供一个编译器标志，将所有全局变量视为秘密数据；不过，由于 Mach-O 格式的限制，AArch64 不支持对单个全局变量进行注解（见第8节）。我们会转换源程序中访问分配时已标注为秘密数据的位置的内存操作。

由于这些变换会改变操作语义，因此修改存储操作时，也必须处理所有访问同一存储地址的相应加载操作。仅依靠静态分析无法达到这种精确度，因此我们使用指针的高位作为标签，进行动态机密跟踪。对于 AArch64 架构，寻址仅使用低 48 位 [1]，而最高字节会被忽略，因此我们可以用剩余的 8 位为秘密数据添加标签。理论上，这种方法支持在同一二进制程序中使用 256 种替代内存格式

[1]我们不支持不常见的 52/56 位寻址模式 es.

存储数据。在我们的实现中，我们采用一种保守内存格式：将秘密数据划分为 32 位片段，并以 32 位固定常量分隔；排列方式确保每个 64 位块都以该常量开头。这样可确保内存中任何包含秘密数据的 64 位对齐位置都不会呈现为有效地址。我们的方法允许多种格式共存，并通过地址高位中的不同位掩码来表示，但目前仅使用一位来表示秘密属性。

指针认证（PAC）[2] 使用相同的高位地址位来签名指针，因为地址转换会忽略高位，且没有其他可用的非地址位。在启用 PAC 的系统中，我们可以通过使用特定的 PAC 密钥对所有与秘密数据相关的地址进行签名，将我们的方法与其集成。这确保了 PAC 与我们的防御机制之间的兼容性。然而，我们特意选择不依赖 PAC，以避免将我们的防御限制在支持或启用了 PAC 的系统上。

5.1.2 为秘密数据分配内存

SplittingSecrets 转换需要额外的内存来存储数据及其常量，从而使秘密数据所需的内存几乎翻倍，但数据小于 32 位的情况除外。对于堆分配，这些额外内存由自定义分配器提供。对于栈分配，额外内存由运行时函数在堆上分配。对于全局变量，我们创建一个类似构造函数的函数，在初始化期间调用该函数，在堆中分配额外内存，并通过将其清零来完成初始化，以模拟全局存储变量的行为。

5.1.3 为秘密数据设置位

要在堆上分配秘密数据，源程序必须使用 *SplittingSecrets* 提供的内存分配器和内存释放器封装函数。这些封装函数会根据秘密数据的大小，在堆上分配额外内存，用于存储秘密数据及其常量前缀。额外内存的分配位置不必紧邻原始分配。这些内存封装函数属于我们的运行时，负责跟踪额外分配的地址，并将其与原始分配的地址关联起来。程序执行期间，运行时会根据数据的地址查询相应额外分配的地址。分配器封装函数还会调用运行时，在地址中设置相应的位，以标记这些地址所指向的数据为秘密数据。如果未经变换的存储或加载操作尝试访问这些分配器返回的地址，由于地址的高位已置位，程序将会崩溃，从而安全地处理变换失败或与意外的第三方代码交互的情况。

在 LLVM 中，栈分配通过 alloca 指令执行。为管理这一过程，我们对所有函数进行插桩以跟踪每个栈帧，模拟函数调用中典型的压入和弹出操作。对于每个函数，我们转换所有 alloca 操作，并添加对运行时的调用，由运行时获取栈上已分配的地址。随后，运行时会在堆上分配存储秘密数据所需的额外内存，并跟踪秘密数据的各个片段及其内存位置。它通过设置地址的高位将该分配标记为秘密数据，并用修改后的地址替换 alloca 返回的地址的所有使用处。

由于 `alloca` 和 `malloc` 都会返回指针，因此可以轻松设置高位，以便在整个程序中进行跟踪。然而，尽管全局变量也是指针，它们并不会由 LLVM IR 中的任何函数返回，而是直接存在。为了设置它们的高位，我们扩展了符号重定位逻辑；该逻辑通常只支持较小偏移量的递增和递减。使用带有足够大偏移量的加法等操作，可以通过适当的偏移量模拟设置高位。不幸的是，由于 Mach-O 文件格式带来的限制，这种方法不适用于 AArch64；更多细节请参见局限性一节，第 8 节。尽管我们支持将秘密数据作为全局变量，但在 AArch64 上不允许将单个全局变量标注为秘密数据。相反，我们提供了一个选项，可将所有全局变量指定为秘密数据，或一个也不指定。

5.1.4 转换秘密

要转换秘密数据，我们首先为每个内存操作的地址添加运行时检查，通过检查地址高位中的位模式，判断该地址是否被标记为秘密。由于此验证发生在运行时，我们必须加入代码来处理两种情况：数据是秘密数据，以及数据不是秘密数据。如果内存操作使用了秘密数据，我们就在相应的代码路径中对其进行转换。

数据被划分为 32 位片段，每个交替的 32 位片段都存储在为该秘密数据分配的额外内存中。这种组织方式简化了对额外内存中数据的跟踪，且开销极小，因为我们只需知道额外内存中与秘密数据分配相对应的分配起始地址。我们的内存拆分方法支持所有数据类型，包括向量、数组、字符和整数，但不包括浮点数和双精度浮点数。排除浮点数和双精度浮点数，是因为 LLVM IR 不支持对浮点参数执行位移等操作，而这些操作是转换所必需的。幸运的是，浮点数和双精度浮点数通常不会用于存储秘密数据。如果内存操作使用了浮点数，我们会发出编译时警告。我们仍会先将其类型转换为整数，再进行转换。如果秘密数据不是浮点数，那么程序在执行时绝不会走这条代码路径。

处理常见大小（如 8、16、32、64、128 和 512 位）的内存操作由运行时直接支持。在变换过程中，会添加对相应大小的运行时函数的调用。我们还处理未对齐的

访问和非标准大小，方法是直接转换 LLVM IR。

为确保与秘密 alloca 分配相关的所有内存操作都经过转换以保证正确性，我们将此转换遍作为 LLVM IR 遍次流水线的最后一步执行。这样可以涵盖优化期间可能生成的任何存储操作或加载操作。此外，我们会禁用某些遍次，例如循环习语遍次，因为它们可能会使我们的转换无法识别内存操作，或在编译流水线的其他部分处理这些操作。

5.1.5 拦截 libc

我们的转换会改变数据在内存中的存储和访问方式，因此第三方代码或库必须使用此转换重新编译，才能正确处理数据。其中一个库是 libc，必须对其进行转换，以确保与我们的系统兼容。Libc 包含两类函数：一类可由客户端实现（例如 memcpy），另一类则提供无法以可移植 C 代码实现的功能，例如 fopen 等系统调用包装函数。由于我们不转换内核，系统调用无法通过指向已修改数据的指针正常工作。因此，我们会在系统调用执行前将数据转换回原始格式，以确保正确性并尽量减少数据暴露。libc 中也大量使用内联汇编来提升性能优化，例如 memcpy 函数中的内联汇编。LLVM 无法转换内联汇编，因此，为确保兼容性，我们会尽可能重写不含内联汇编的 libc 函数。这是常用 libc 函数的参考实现，可使用我们的转换遍进行编译。对 libc 函数的调用会被拦截，并重定向到此转换后的参考实现。

任何不含内联汇编且可使用 Clang 编译的 libc 实现，都可以替代我们的参考实现，并由我们的工具进行转换。

5.2 编译器生成的内存操作

源程序会执行内存分配和内存操作，但编译后的二进制文件可能发生的内存交互并不止这些。编译器也会执行内存操作，尤其是在栈上，从而增加额外的内存事务。目前我们尚未发现有基于 DMP 的攻击利用编译器添加的操作泄露秘密数据，但没有理由认为这类攻击无法实现。因此，*Split-tingSecrets* 支持这样的场景：秘密数据可能通过编译器间接引入的内存操作存储。这些操作包括对被调用者保存寄存器的加载与存储、寄存器溢出和填充，以及通过栈传递参数。

5.2.1 标注和跟踪秘密数据

通过动态机密跟踪，我们可以在程序执行期间准确判断某项内存操作是否需要转换。由于无法预先完全知晓经编译器转换的操作，而且这些操作与源代码生成的操作无法区分，因此必须在编译器内部进行标注，同时采取保守策略。我们会转换所有编译器生成的内存操作，但与栈保护值相关的操作除外。我们可以改为修改栈保护值，选择一个无法通过 DMP 地址检查的金丝雀值，这与我们选择的前缀类似。

编译器生成的内存操作会在特定于体系结构的代码生成过程中，于流水线的不同阶段添加。LLVM 的代码生成从有向无环图开始，其中可能包含秘密数据的内存操作节点会通过添加新的元数据进行标记。这些节点会转换为机器中间表示（MIR），并在降级过程中丢失元数据。由于存在多个变换点，直接转换每个操作会很复杂且容易出错。因此，我们将带有秘密数据元数据的节点降级为用于秘密数据的自定义伪指令。直接生成为 MIR 指令的 MIR 存储操作和加载操作会通过元数据标志进行标记。我们会在流水线末尾、生成汇编之前，将所有被识别为伪指令或带有秘密标志的内存操作进行转换。

5.2.2 为秘密分配内存

我们系统地扩大所需管理的每类内存操作的分配大小。对于寄存器溢出与填充，我们调整所有 AArch64 寄存器的溢出大小。由于溢出大小变得大于寄存器大小，因此需要更新 LLVM 内部对寄存器大小和溢出大小的使用。对于栈传递参数，我们修改调用约定以使用适当的寄存器大小。相应地，我们确保调整分配大小，并让所有依赖的编译器组件知悉这一调整。

这些分配和变换发生在代码生成流水线中，通常位于寄存器分配之后，从而使这些变换变得复杂。我们通过将额外分配的内存放置在现有内存分配的相邻位置来解决这一问题：可以调整负责原始分配的参数，也可以手动更新分配参数。这种方法在保持与编译器操作对齐的同时，无缝整合了扩展后的内存需求。

5.2.3 转换秘密

虽然我们专门处理由源代码直接生成的内存操作，但要对编译器在代码生成过程中生成的内存操作应用类似的变换，则更加复杂。在代码生成流水线的后期转换这些内存操作会带来诸多限制，例如难以添加运行时调用，以及必须

在变换过程中使用保留寄存器作为临时存储。考虑到这些限制以及潜在变换点数量众多，我们调整了方案，使其轻量且简单。

为了简化实现，我们直接存储相邻的内存段，而不涉及运行时。我们在后端修改所有可能存储秘密数据的内存分配，并转换每种加载和存储指令，同时假设额外内存是连续的。这意味着偏移量计算是在转换后的代码中进行的，而不是像原始分配和新增分配那样维持相同的偏移量。之所以可以采用这种简化，是因为寄存器大小以及溢出、保存被调用者保存寄存器和通过栈传递参数等任务所需的内存都是明确且有限的。

这些后端变换必须在所有其他可能引入额外内存操作的后端优化遍之后执行。这些变换需要临时寄存器，因此我们在编译器中预留两个 64 位寄存器。

5.3 正确性、安全性与健全性

以下属性共同确保秘密数据免受启用 DMP 的攻击，同时不影响程序的原有行为。

5.3.1 正确性

正确性意味着变换后程序的外部可见行为与原始程序一致。对于 *Split-tingSecrets*，这要求每次变换后的确定性加载都产生与未变换程序相同的值。

为了验证这一点，我们依靠对真实应用程序和自有测试套件进行的大量测试。我们的定制测试套件包含针对实现中特定功能的专项测试，总计超过一百个单元测试。此外，我们还在密码学库 libsodium 上测试变换。 [19]我们将所有内存分配——堆、栈和全局内存——都视为秘密数据，以便在寄存器溢出、被调用者保存寄存器和栈传递参数等多种情形下，尽可能增加变换后的存储操作和加载。然后，我们使用 libsodium 自带的测试套件对变换进行压力测试。

压力测试无法保证正确性，但能通过暴露潜在问题来增强信心，例如未变换的指令或不匹配的内存分配。举例来说，遗漏与秘密数据相关的内存操作变换，很可能导致内存访问错误或程序故障。我们的实现在整个测试套件、libsodium 的全部测试套件，以及使用 libsodium 进行评估的微基准测试中的正确性测试中均全部通过。

5.3.2 安全性

安全性意味着，任何被修改的存储操作所存储的值，都不会导致预取器成功进行页表遍历。我们的变换在设计上就是安全的，确保任何变换后的存储操作都会使内存包含一个或多个 64 位对齐的数据块，且每个数据块都有一个固定的 32 位前缀。

我们专门考察运行基于 MacOS 的系统的 Apple M 系列 DMP。GoFetch [11]（第 4 节）对 Apple M 系列 DMP 进行了全面的逆向工程，并识别出它用于确定预取候选地址的若干启发式方法。这种实现利用了 DMP 的一项要求：值的存储位置必须与将该值解释为指针所得的地址位于同一个 4 GB 对齐区域。因此，只要使用一个能确保生成的“地址”距离任何用户空间分配至少 4 GB 的前缀即可，因为 DMP 只会解引用距离存储位置 4 GB 以内的指针 [11]。我们使用前缀 0*xdeadceef*，它满足这一要求，因为任何带有该前缀的“地址”都高于 XNU 中允许的最高虚拟内存地址 4 GB 以上 [2]。类似的选择方法也适用于 Intel DMP，因为它相对于指针的存储位置，同样具有范围有限的可预取区域。在我们的实现中，更改所用的具体前缀十分简单，甚至可以在执行时选择。攻击者可能会试图安排分配，使带前缀的 64 位字看起来落在 DMP 有效的局部性窗口内，或者部分覆盖转换后数据块的高位，以构造出看似合理的指针。我们选择的前缀会使合成地址落在 DMP 可解引用的所有用户可映射区域之外（例如，超出 4 GB 局部性范围），因此，在不违反虚拟内存约束的情况下，无论如何调整堆或栈布局，都无法使其进入有效范围。要伪造可用地址，就必须破坏多个拆分数据块中恒定的高 32 位；这实际上构成针对应用程序的内存安全攻击，而不是预取侧信道。如果未来的启发式方法扩大了有效范围，则可以轮换前缀或进一步缩减可用的秘密数据片段，从而维持这样一项不变条件：转换后的字始终不是有效的预取候选对象。

5.3.3 健全性

健全性保证所有与秘密或秘密派生的内存位置交互的操作都会得到一致的转换。对于*SplittingSecrets*，我们允许手动标注秘密数据；如果无法进行标注，则对所有存储的值进行完整转换。为验证*SplittingSecrets*会转换所有标记为秘密的内存区域，我们在 macOS/AArch64 上使用 QBDI 进行了一项动态插桩实验， [45] 重点测试一种经过强化的常数时间 64 位条件交换。我们通过预加载插桩每一次函数内存写入，将 8 字节存储操作归类到 16 字节簇中，并检查偏移量 0 和 8 处的常量前缀值（两者均为`0xdeadceef`），以及偏移量 4 和 12 处依赖于数据的 LOW/HIGH 分量。经过多次运行，我们观察到

[2] https://github.com/apple-oss-distributions/xnu/blob/8d741a5de7ff4191bf97d57b9f54c2f6d4a15585/osfmk/mach/arm/vm_param.h#L137

完全覆盖，未发现任何原始 64 位、字节大小或其他未插桩的存储操作，这为标记为秘密的内存操作均经过一致转换提供了证据。

6 评估

转换后的内存操作（位于 libsodium 中）	libc used	编译时间（秒）	二进制大小（KB）
None	glibc	18.54	397.63
None	libc⋆	18.64	397.69
源代码	libc	18.94	720.72
源代码•	libc	19.27	749.30
编译器与源代码	libc	18.98	1121.24
编译器与源代码•	libc	19.40	1193.85

表 1：使用*SplittingSecrets*对 libsodium 库进行的编译时间和二进制文件大小分析。Source 和 Compiler 指内存操作的来源，•表示所有源代码分配均为秘密数据，libc 是我们采用的 glibc 参考实现，而 ⋆ 表示未经转换的 libc。

SplittingSecrets 会影响目标程序和编译器的多个方面。本节将从四个维度评估我们的防御所引入的开销：

- 编译时间（第6.1节）
- 二进制文件大小（第6.2节）
- 运行时性能影响（第6.3节）
- 运行时内存消耗（第 6.4节）

我们的评估使用 libsodium 库进行。该库是一种功能全面的密码学库，提供了丰富的密码学原语和应用程序接口。使用 LLVM 实现的一个局限是，我们无法修改内联汇编。因此，我们确保启用 `-disable-asm`（WebAssembly 目标使用此选项），以强制大多数操作使用参考实现。我们还禁用变长数组（见局限性第 8.5节）。除此之外，我们使用常规编译标志构建 libsodium。我们确保每次构建的 libsodium 测试套件均通过所有测试。

我们的防御要求重新编译程序所依赖的库，并集成我们的变换，以确保库代码能够妥善管理已注释的秘密数据，并正确处理与这些秘密数据相关的内存操作。鉴于常见的 libc 实现通常包含用于优化的内联汇编，且可能没有禁用这些汇编的选项（如 libsodium 中的情况），我们开发了一个不含内联汇编的 libc 参考实现。随后，我们使用我们的转换遍编译此实现，以确保数据能够正确传递给系统调用。

由于大多数应用程序执行过程中并不会将大部分时间用于密码学操作，我们使用微基准测试分析 libsodium 的开销。所有评估（编译和执行）均在一台未改装的 Apple Mac Mini 上进行，该设备搭载 Apple M1 芯片（8 核 CPU），配备 8 GB 统一内存，并使用固态硬盘存储。

6.1 编译时间开销

SplittingSecrets 包含用于在 LLVM IR 和 MIR 两个层级处理内存操作的转换遍，这可能会增加编译时间。表 1 展示了使用单个作业编译 libsodium 的实际耗时，作业数通过 `-j 1`指定。我们的基准对比对象是未经修改的 LLVM 编译器，例如未使用保留寄存器、未禁用 *SplittingSecrets* 的转换遍，且未拦截 libc 函数。我们对 libsodium 进行了 10 次干净构建，并报告平均编译时间。

为了解我们的 libc 参考实现对基准的影响，我们使用一种用于拦截 libc 函数的变换来编译 libsodium。无论 libc 是否使用我们的变换进行构建，结果都没有区别，因为编译器只需添加对它的调用。我们发现，拦截 glibc 函数会使性能开销增加 0.53%。如第 5.1.5 节所述，经过变换的 libsodium 无法与未经变换的 libc 一起工作。转换所有源代码生成的秘密数据后，编译时间增加了 2.15%；同时处理编译器生成的内存操作后，编译时间增加了 2.37%。这部分增加归因于变换内存操作，以及添加用于跟踪秘密数据及其额外分配的运行时调用所耗费的时间。接下来，我们假设 libsodium 内部的所有分配都是秘密数据，并据此进行编译。当仅修改源代码生成的操作时，开销增加了 3.93%；同时纳入源代码和编译器生成的操作时，开销增加了 4.63%。这是因为跟踪秘密数据及其分配所需的运行时调用数量增加了。

6.2 二进制文件大小影响

SplittingSecrets 中的 LLVM IR 转换遍会引入运行时检查，以确定内存操作是否正在处理敏感数据。如果是，则会建立一条控制流路径，调用运行时函数或执行一系列指令。此外，MIR pass 会修改带标志的内存操作，在存储时拆分秘密数据，并在加载时重新组合，从而增大二进制文件大小。

表 1 展示了与第 6.1 节所述、用于了解编译时间开销的相同基准 LLVM 版本的二进制文件大小。为了解我们的 libc 参考实现对基准的影响，我们使用一种用于拦截 libc 函数的变换编译了 libsodium。我们发现二进制文件大小增加了 0.01%，因为编译器会将某些 glibc 调用直接替换为目标特定指令。启用用于从源代码修改秘密数据的变换后，我们观察到二进制文件大小增加了 81.25%，

libsodium	执行时								内存使用量							
函数	B2	B3	S○	S◐	S●	SC○	SC◐	SC●	B2	B3	S○	S◐	S●	SC○	SC◐	SC●
xchacha20-*e*	1	1.03	1.38	1.48	12.38	1.44	1.48	12.29	1.16	1.16	1.2	1.2	3.62	1.22	1.22	3.66
xchacha20-*d*	1	1	1.42	1.48	13.83	1.48	1.51	13.62	1.16	1.16	1.2	1.2	3.62	1.22	1.22	3.66
xsalsa20-*m*	1	1.03	3.54	3.53	41.64	3.59	3.59	41.02	1.16	1.16	1.19	1.19	6.2	1.21	1.21	6.25
xsalsa20-*v*	1	1.03	3.54	3.6	43.36	3.63	3.64	42.5	1.16	1.16	1.19	1.19	6.2	1.21	1.21	6.25
salsa20-*e*	1	1	1.64	1.64	18.64	1.64	1.64	18.61	1.16	1.16	1.19	1.19	1.72	1.21	1.21	1.77
salsa20-*d*	1	1	1.64	1.64	18.82	1.64	1.64	18.77	1.16	1.16	1.19	1.19	1.72	1.21	1.21	1.77
blake2b-*h*	0.98	1.24	1.24	1.24	1.24	1.32	1.32	1.32	1	1	1	1	1	0.98	0.98	0.98
h-sha256-*m*	0.99	1.02	3.50	3.49	41.21	3.55	3.55	40.66	1.16	1.16	1.19	1.19	6.2	1.21	1.21	6.25
h-sha256-*v*	0.99	1.02	3.57	3.56	42.82	3.60	3.60	42.09	1.16	1.16	1.19	1.19	6.2	1.21	1.21	6.25
h-sha512-*m*	1	1.10	3.82	3.94	69.71	3.44	3.53	69.15	1.16	1.16	1.19	1.19	10.55	1.2	1.2	10.6
h-sha512-*v*	1	1.08	3.88	3.99	70.39	3.46	3.54	69.54	1.16	1.16	1.19	1.19	10.55	1.2	1.2	10.6
ed25519-*s*	1	1.02	3.67	4.30	98.22	5.85	5.85	90.63	1.17	1.17	1.25	1.25	15.71	1.37	1.37	27.1
ed25519-*v*	1.01	1.01	2.43	2.44	67.83	3.36	3.37	68.46	1.17	1.17	1.25	1.25	15.71	1.37	1.37	27.1
带关联数据的认证加密-*e*	1.02	1.04	1.48	1.48	13.10	1.52	1.54	13.02	1.17	1.17	1.2	1.2	3.37	1.23	1.23	3.41
带关联数据的认证加密-*d*	1	1.02	1.51	1.51	14.25	1.55	1.55	14.15	1.17	1.17	1.2	1.2	3.37	1.23	1.23	3.41

表 2：我们测量了变换所导致的运行时开销。结果表示相对于未应用任何传递或更改的基准 LLVM 构建版本 libsodium 的倍数开销。*B2* 和*B3* 分别表示使用 libc 参考实现的基准版本，以及使用经过我们的变换处理后的 libc 参考实现的版本。*S* 表示仅处理源代码中的秘密数据时的开销；SC 表示同时处理编译器生成的操作时的开销。我们用 ○、◐ 和 ● 分别表示源代码未标注秘密数据、选择性标注秘密数据，以及假设所有内容都是秘密数据。前缀*h-* 表示 HMAC，后缀*-e*、*-d*、*-m*、*-v* 和*-s* 分别表示加密、解密、消息认证码、验证和签名 。另请注意，实验中使用的 AEAD 是 chacha20 poly1305。

因为这会为每个内存操作添加检查和运行时函数调用。支持编译器生成的内存操作也会使二进制文件大小增加 181.98%，因为编译器生成的大多数操作都会被转换为一系列未经进一步优化的指令。此外，当将 libsodium 中的所有分配都视为秘密数据时，由于跟踪秘密数据及其分配需要更多运行时调用，我们分别观察到二进制文件大小增加了 88.44% 和 200.24%。

6.3 运行时性能开销

为了评估 *SplittingSecrets* 的运行时性能开销，我们使用 libsodium 库开发了八个微基准测试。这些基准测试涵盖常见的密码学用例，包括流密码和分组密码、哈希、密钥生成及加密算法，并在源代码中添加注释以标明秘密数据。我们比较了不同场景下每种密码学原语的性能：使用基线编译器，分别搭配标准 libc 和参考 libc 实现运行（详见第 6.3 节）；配置编译器以对直接由源代码生成的内存操作应用变换；以及对源代码和编译器生成的内存操作都应用变换。我们还比较了带注释和不带注释的源代码（假定不存在秘密数据），以及假定所有内容都是秘密数据时的情况，以便更好地了解开销。

对于每种密码学原语，我们都进行了 1,000 次迭代的评估循环，并在此前进行了 25 次预热迭代。运行时间通过 AArch64 汇编指令 `msr` 访问虚拟计数寄存器进行测量。报告的开销以使用基准 LLVM 编译的 libsodium 的平均运行时间为基准进行了归一化，结果列于表 2。

研究结果表明，处理直接由源代码生成的内存操作时，平均会减慢 262%；处理由编译器生成的内存操作时，平均会减慢 274%。

6.4 内存消耗开销

我们使用了此前用于评估运行时开销的相同微基准测试，详情见第 6.3 节。我们的评估重点比较经过 100 次预热运行和 1000 次评估运行后观察到的最大常驻集大小，它反映了

进程在执行期间占用的峰值物理内存。我们首先考察使用参考 libc 实现（无论是否进行变换）相较于未拦截的 libc 搭配基线编译器的影响，如第 6.1 节所述。我们的研究结果表明，参考 libc 导致的内存消耗增幅可以忽略不计。这一增幅源于各个函数实现中使用的简单算法；对参考 libc 应用变换后，这些算法的规模显著扩大。内存消耗的变化如表 2 所示。

此外，与采用参考 libc 实现的基线编译器相比，在启用用于处理直接由源代码生成的内存操作的变换后，我们观察到内存消耗增加了 118%。值得注意的是，当我们假设程序不包含秘密数据时，内存消耗的变化可以忽略不计。这一观察结果表明，鉴于秘密数据数量稀少，用于存储秘密数据的替代 32 位片段及对其进行跟踪所需的内存，并不会显著影响总体内存消耗。

此外，我们的分析表明，内存消耗增加的主要原因是运行时调用次数上升，从而导致更多的寄存器溢出。同样有所贡献的是：在将每个数据项传递给系统调用之前，必须先将其转换回原始形式。这种转换需要为原始格式的数据分配内存，并使用插桩 memcpy 将数据从其变换后的形式复制过来。此外，启用用于处理编译器生成的内存操作的后端变换，会进一步使内存消耗增加 121%。这是因为对于每个发生寄存器溢出、由被调用者保存或作为参数传递的寄存器，都需要分配两倍的内存。

7 讨论

SplittingSecrets 会给其保护的程序带来明显的编译时间、二进制文件大小、执行时长和内存消耗开销。这些结果在预料之中，因为我们会将每个内存操作转换为一系列指令或运行时调用。即使所操作的数据并非秘密数据，除栈保护值生成等显而易见的情况外，所有编译器生成的内存操作都会被转换；而对于所有直接由源代码生成的内存操作，都会插桩检查其是否涉及秘密数据。

7.1 数据分割大小

基于体系结构和微体系结构的分割策略越专门，产生的开销就越小。在 *SplittingSecrets* 中，我们将所有数据拆分为交替排列的 32 位片段，并用固定常数值屏蔽高 32 位，确保数据绝不会呈现为地址。对于 128 位存储，我们现在使用 256 位内存，以及四条 64 位指令，从原始值中提取 32 位片段，并用固定常量屏蔽高半部分。如果我们屏蔽的是高 16 位，而非高 32 位，就可以将其缩减为 192 位内存和三次 64 位存储操作，从而降低运行时和内存消耗开销。

7.2 针对秘密数据的选择性变换

我们的目标是有选择地保护秘密数据，防止其通过数据相关预取器诱导的侧信道泄露。为此，我们会对所有处理秘密数据的函数进行插桩，包括常见 libc 函数。不过，这并非总是必要的。例如，如果某个函数只通过寄存器接收参数，且能够证明这些参数绝不可能是秘密数据，那么该函数就无需进行变换。此外，如果某个函数接收一个指针作为参数，且能够证明该指针不指向承载秘密数据的内存，那么该函数也无需插桩。这些改进留待未来工作。

7.2.1 转换编译器生成的内存操作

如果能够确保编译器绝不会将秘密数据存储在栈中，那么对于编译器为处理寄存器溢出、被调用者保存寄存器以及通过栈传递的参数而生成的内存操作，就无需进行处理；这与处理为栈保护值生成的内存操作类似。可以采用安全寄存器分配和内联等技术，避免转换编译器生成的内存操作。

在寄存器分配之后，分析内存操作以检查它们是否作用于秘密数据，并没有太大用处；除非已知某个特定的内存地址范围不会与秘密数据发生交互。在这种情况下，我们无需为这些数据分配额外内存。

7.3 去分类边界

我们的防御机制会改变数据在内存中的存储方式，使数据相关预取器无法将其解读为地址。数据存储在内存中时，会被转换为另一种形式。不过，当数据从内存加载到寄存器时，最终必须将其转换回原来的形式，也就是进行去分类。数据进入寄存器后，使用这些寄存器的代码无需关心数据最初在内存中以何种格式存储。这一概念同样适用于库函数；这些函数需要了解如何处理处于转换状态或已分类状态的数据。因此，必须对库重新编译并插桩，以便正确处理内存中已分类的数据。

此过程会创建一个去分类边界，数据在此处恢复为原始形式。某些库和应用二进制接口无法进行转换以适配这种插桩，

因此无法管理机密数据。我们将寄存器定义为缓解策略中的自然去分类边界，因为预取器根据存储在内存中的数据运行，而非寄存器中的数据；寄存器中的数据无需被分类。同样，我们也将系统调用视为去分类边界，因为无法通过插桩修改其实现来管理机密数据。

8 局限性

8.1 内联汇编支持

我们的方法存在一项重要局限：缺乏对内联汇编的编译器支持。LLVM 将内联汇编和汇编文件视为带有定义和使用关系的字符串，但不支持对其进行变换。这使我们只能保护 libsodium 中不使用内联汇编的参考实现。通常，可以将编译器变换移植到二进制重写器中，以解决此类局限。然而，这种方法仍有不足，因为必须调整为秘密数据分配的内存，以适应溢出大小的变化，以及其他编译器内部参数（例如被调用者保存寄存器的大小）的变化。二进制重写器和分析器无法满足这些要求。未来的工作可以扩展 *SplittingSecrets* ，使其能够提供建议的汇编输出，供经验丰富的开发者进一步调整，以实现性能优化。

8.2 架构无关的实现

我们的插桩遍直接处理由源代码生成的内存操作，并运行于 LLVM IR，因此与体系结构无关。然而，为了处理编译器添加的内存操作，我们需要一个针对特定体系结构的遍，该遍作用于 MIR，具体来说是 AArch64 后端。若要支持其他后端，就需要调整此实现。虽然可能需要进行类似的修改，但这些修改不会完全相同。例如，在 AArch64 中，Mach-O 紧凑展开格式要求将被调用者保存寄存器存储在相邻的寄存器对中。

8.3 秘密数据的精确跟踪

为了在整个程序中准确追踪秘密数据，必须只对与秘密数据交互的内存操作进行插桩。然而，由于静态分析的局限性，无法实现完全精确的别名分析。任何错误分类的加载都会破坏程序行为，因此我们选择按照第 5.1.1 节所述，在分配时动态标记地址。

8.3.1 秘密派生值的跟踪：

目前，秘密派生值必须像密码学秘密一样处理和标注，否则就必须将所有值都视为秘密。这意味着，对于 memcpy 这样的函数，如果要保护秘密数据，就必须将源和目标都标注为秘密。更好的折中方案是对已标注的秘密数据进行污点分析，以识别派生值并将其也标记为秘密。此外，还需要在密码学操作完成后添加显式的反机密化操作。我们将这些改进留待未来工作，并指出，“将所有值视为秘密”所产生的开销是我们这些变换对性能影响的上限。

8.4 全局秘密的注释支持

SplittingSecrets 的某些功能依赖于体系结构。在我们针对 AArch64 的原型实现中，目前尚不支持将全局变量注释为秘密数据。不过，这一限制并非根本性的，可以通过修改重定位代码加以解决。例如，ARM 的指针认证（PAC）已经使用了类似机制来设置更高位的地址位。由于秘密数据很少存储在全局变量中，我们将这项支持留待未来工作。

对于使用 ELF 格式的 x86 架构，支持对全局变量进行注释。符号重定位逻辑可以处理 64 位偏移量，从而允许我们设置全局地址的高位，并将其标记为秘密。在采用 Mach-O 格式的 AArch64 上，符号重定位不允许使用可用于设置高位的偏移量。因此，AArch64 上的全局变量只能整体被视为秘密，或者完全不被视为秘密。如果被标记为秘密，它们将由我们的系统进行全面加固。虽然这一限制专门针对在 AArch64 上将单个全局变量标记为秘密，但不会影响 *SplittingSecrets* 的整体功能。未来可以探索扩展 AArch64 的重定位支持（如果确有需要），但鉴于将秘密数据存储为全局变量的情况很少，我们认为这属于低优先级增强。

8.5 动态栈分配

我们的原型实现不支持对变长栈数组或可变参数函数的参数进行加固。这些功能需要在编译器中单独分配栈空间，因此必须提供额外支持并仔细记录，以管理栈分配。分配的内存也必须打上标签，以便追踪秘密数据。幸运的是，libsodium 提供了禁用变长数组的选项，我们已在构建时启用该选项。至于可变参数函数，它们只出现在 libsodium 的正确性测试中；我们在测试时将 printf 调用替换为 puts，从而避免使用可变参数。我们支持 alloca，需要时可用它在栈上动态分配内存。

9 相关工作

本节概述了与本文相关的研究，这些研究大致可分为三个主要方向：利用编译器防御侧信道攻击、修改数据表示的现有技术，以及软件预取器攻击。

9.1 编译器中的侧信道缓解措施

基于编译器的方法是防御侧信道攻击的常见手段。这些技术无需修改硬件即可提供保护，从而增强了在现有系统中的可部署性。

同期工作，*Zebrafix* [47] 提出了一种基于编译器的方法，通过将秘密内存与非秘密值交错排列来缓解静态数据攻击。在其方案中，他们使用递增计数器——而不是静态前缀——来缓解静默存储和密文侧信道。作者认为，通过减小块大小，该方法可以扩展到涵盖数据内存依赖预取器，但其实现和评估仅涉及非 DMP 攻击。

一种常见策略是强制执行恒定时间（CT）执行，以消除时序侧信道。这包括使用分析来查找程序点，并将其转换为与秘密数据无关的形式 [16, 57]。然而，CT 并不总能充分防御侧信道攻击；有时泄露噪声或错误数据比防御攻击更容易。推测性加载加固（SLH）是 LLVM 中实现的 Spectre v1 防御机制，会在推测执行期间污染相关值 [3]。Rane 等人 [48] 开发了 Raccoon，这是一种基于编译器的系统，可转换程序以混淆内存访问模式，使其能够抵抗缓存时序攻击。有时，攻击依赖于某种硬件特性，例如特定的算术运算优化，而该特性可能引入侧信道。对此类攻击，可以通过避免生成能够触发攻击的模式，或将触发模式转换为安全模式来加以阻止 [21]。

我们的编译器方法通过避免触发 DMP 来进行防御，而不是依靠代码模式，而是修改内存中的数据表示。

9.2 与存储数据攻击的相似之处

尽管它们采用截然不同的技术来恢复数据，但与通过外部测量来恢复存储在磁盘或随机存取存储器中的数据的攻击有一些值得关注的相似之处。一个经典例子是冷启动攻击 [32]，它同样能够恢复那些在未被主动处理时被认为安全的数据。这表明，为保护存储在磁盘或随机存取存储器中的数据而设计的策略，或许能为防御 DMP 启用型攻击提供思路。这两种情境都需要保护静态数据免遭绕过传统访问控制的间接推断攻击 [30, 53]。

9.2.1 数据掩码

数据掩码是指将数据表示为多个元素的组合，使这些元素无法与原始数据建立关联 [23]。当数据被划分为d 个份额时，这称为d 阶掩码，其中数据由d 个份额表示 [49]。可以使用二进制表示技术将这些掩码数据重新组合为原始形式，这通常称为布尔掩码 [17]。例如，将数据与常量进行异或是一种常见的布尔掩码技术 [49]。此外，这些掩码技术也可以作用于十进制表示，例如 Shamir 秘密共享 [23]。

现有的掩码技术无法用于防御由 DMP 引起的侧信道，因为它们不会专门修改数据以使地址转换失败，因此仍会泄露部分信息。与掩码技术不同，我们的防御方案将数据修改限制在内存中，而不是寄存器中，从而能够在编译器中高效实现。这也是基于编译器的高阶数据掩码实现并不常见的原因之一。数据掩码技术在硬件中则要常见得多，通常用于防御功耗分析侧信道。

9.2.2 数据空间随机化

数据空间随机化（DSR） [7] 会对内存中的数据值进行随机化，以防止它们变得可预测或呈现特定模式。虽然 DSR 的概念目标同样是防止数据呈现为有效地址，但它在防御 DMP 方面存在根本性局限。DSR 会应用概率变换（通常是与随机掩码进行异或），因此随机化后的数据仍有非零概率呈现为有效地址并触发 DMP。相比之下，*SplittingSecrets* 通过使用架构感知前缀提供确定性保证，确保任何 64 位对齐值都无法通过地址转换。此外，DSR 可能会因需要为每项数据生成和管理随机掩码而产生更高开销，而我们采用固定前缀的方法只需使用编译时常量进行简单的位操作。

9.3 软件预取攻击

在另一条并行研究路线中，多个项目也发现，攻击者可以滥用软件预取指令。例如，PREFETCHW 可用于构建高容量隐蔽信道，并通过时序分析增强侧信道攻击 [28, 31]。此外，预取器可以操纵分支预测器等辅助组件，从而实现定向缓存驱逐 [60]。最后，不同架构也存在漏洞；例如，Lipp 等人 [41]

讨论了影响内核隔离的 AMD CPU 时序泄漏，凸显了保护所有形式的预取机制这一更广泛的挑战。

10 结论

本文介绍了*SplittingSecrets*，这是一个基于编译器的框架，旨在应对新兴的数据内存依赖预取器（DMP）攻击威胁。鉴于恒定时间编程难以应对这类复杂漏洞，*SplittingSecrets* 提出了一种新颖的方法，通过转换内存操作，防止秘密数据以内存地址的形式暴露，从而有效防止预取器对其进行潜在利用。

借鉴以往应对时序攻击和 Spectre 攻击等微架构威胁的经验，我们的研究凸显了主动采用编译器和软件防御的重要性，以跟上不断演进的漏洞威胁。我们的解决方案基于 LLVM 实现，证明了其兼容性，既支持源代码层面的操作，也支持编译器生成的操作，并适用于 Apple M1 的 AArch64 架构。尽管我们的方法会带来性能权衡，但分析结果证实，实现可靠防护是可行的。

SplittingSecrets 为防御未来的微架构攻击奠定了基础，增强了现有系统的安全性。随着微架构复杂性不断增加，我们的框架提供了至关重要的防御层；我们也鼓励进一步开发变换策略，以缓解潜在的侧信道威胁。

参考文献

[1] Armv8 架构寄存器，适用于 Armv8-A 体系结构配置文件。https://developer。arm。com/docs/ddi0595/latest/aarch32-system-registers/cpsr。

[2]Armv8.1-M 指针认证与分支目标识别扩展。https://community。arm。com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension。

[3]推测加载加固；LLVM。https://llvm。org/docs/SpeculativeLoadHardening。html。

[4] Onur Acıiçmez、Çetin Kaya Koç 和 Jean-Pierre Seifert。通过分支预测推测秘密密钥。载于密码学主题——*RSA-CT 2007*：*2007* 年 *RSA* 会议密码学家专题，美国加利福尼亚州旧金山，*2007* 年 *2* 月 *5* 日至 *9* 日。论文集，第 225–242 页。施普林格，2006年。

[5] Marc Andrysco、David Kohlbrenner、Keaton Mowery、Ranjit Jhala、Sorin Lerner 和 Hovav Shacham。关于次正规浮点数和异常时序。载于 *S&P'15*，2015年。

[6] Murali Annavaram、Jignesh M Patel 和 Edward S Davidson。基于依赖图预计算的数据预取。*ACM SIGARCH* 计算机体系结构新闻，29(2)：52–61，2001 年。

[7] Sandeep Bhatkar 和 R Sekar。数据空间随机化。载于 入侵检测与恶意软件及漏洞评估国际会议，第 1–22 页。施普林格，2008 年。

[8] Abhishek Bhattacharjee。通过加速内存重放打破地址转换壁垒。*IEEEMicro*，2018年。

[9] Sarani Bhattacharya、Chester Rebeiro 和 Debdeep Mukhopadhyay。硬件预取器泄漏：重新审视用于缓存时序攻击的 SVF。载于*MICROW*，2012年。

[10] Mustafa Cavus、Resit Sendag 和 Joshua J. Yi。面向间接内存访问的感知型预取。*ACM Trans. Archit.Code Optim.*，2020年。

[11]Boru Chen、Yingchen Wang、Pradyumna Shome、Christopher W Fletcher、David Kohlbrenner、Riccardo Paccagnella 和 Daniel Genkin。GoFetch：利用数据内存依赖型预取器攻破常数时间加密实现。载于*Proc. USENIX Secur.Symp*，第 1–21 页，2024年。

[12]Boru Chen、Yingchen Wang、Pradyumna Shome、Christopher W. Fletcher、David Kohlbrenner、Riccardo Paccagnella 和 Daniel Genkin。GoFetch：利用数据内存依赖型预取器攻破常数时间密码实现。https://gofetch。fail，2024年。

[13] Yun Chen、Ali Hajiabadi、Lingfeng Pei 和 Trevor E. Carlson。新型跨核心、缓存无关的基于预取器的侧信道与隐蔽信道。发表于 *arXiv*，2023年。

[14] Yun Chen、Lingfeng Pei 和 Trevor E. Carlson。After-Image：通过硬件预取器泄露控制流数据并追踪加载操作。发表于 *ASPLOS*，2023年。

[15] Robert Cooksey、Stephan Jourdan 和 Dirk Grunwald。无状态、内容导向的数据预取机制。*ACM SIGPLANNotices*，2002 年。

[16] Bart Coppens、Ingrid Verbauwhede、Koen De Bosschere 和 Bjorn De Sutter。针对现代 x86 处理器上基于时序的侧信道攻击的实用缓解措施。载于*2009* 年第 *30* 届 *IEEE* 安全与隐私研讨会，第 45–60 页。IEEE，2009 年。

[17] Jean-Sébastien Coron、Emmanuel Prouff、Matthieu Ri- vain 和 Thomas Roche。高阶侧信道安全与掩码刷新。载于快速软件加密国际研讨会，第410–424页。施普林格，2013年。

[18] Patrick Cronin 和 Chengmo Yang。预取的故事：利用硬件预取器进行隐蔽通信。载于*IEEEHOST*，2019年。

[19] Frank Denis. libsodium。https://github.com/jedisct1/libsodium。[20] Dmitry Evtyushkin、Ryan Riley、Nael CSE Abu-Ghazaleh、ECE 和 Dmitry Ponomarev。《Branchscope：一种针对方向分支预测器的新型侧信道攻击》。*ACM SIGPLANNotices*，53(2):693–707，2018年。[21] Michael Flanders、Reshabh K Sharma、Alexandra E Michael、Dan Grossman 和 David Kohlbrenner。《通过二进制代码变换避免以指令为中心的微架构时序通道》。载于第*29*届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第*2*卷，第120–136页，2024年。[22] Freddy Gabbay 和 Avi Mendelson。基于值预测的推测执行。Citeseer，1996年。[23] Louis Goubin 和 Ange Martinelli。使用 Shamir 秘密共享方案保护高级加密标准。载于密码硬件与嵌入式系统——*CHES 2011*：第*13*届国际研讨会，奈良，日本，*2011*年*9*月*28*日至*10*月*1*日。论文集第*13*卷，第79–94页。施普林格，2011年。[24] Ben Gras、Kaveh Razavi、Herbert Bos 和 Cristiano Giuffrida。转换旁路缓冲区泄漏：利用 {TLB} 攻击击败缓存侧信道防护。载于第*27*届 *USENIX* 安全研讨会（*USENIX Security 18*），第955–972页，2018年。[25] Ben Gras、Kaveh Razavi、Herbert Bos 和 Cristiano Giuffrida。转换旁路缓冲区泄漏：利用 TLB 攻击击败缓存侧信道防护。载于*USENIX* 安全大会，2018年。[26] Ben Gras、Kaveh Razavi、Erik Bosman、Herbert Bos 和 Cristiano Giuffrida。ASLR 风险重重：针对内存管理单元的实用缓存攻击。载于*NDSS*，2017年。

[27] Johann Großschädl、Elisabeth Oswald、Dan Page 和 Michael Tunstall。通过提前终止乘法对密码软件进行侧信道分析。载于 信息、安全与密码学——*I CISC 2009*：第*12*届国际会议，首尔，韩国，*2009*年*12*月*2*日至*4*日，修订精选论文第*12*卷，第176–192页。施普林格，2010年。

[28] Daniel Gruss、Clémentine Maurice、Anders Fogh、Moritz Lipp 和 Stefan Mangard。预取器侧信道攻击：绕过 SMAP 和内核地址空间布局随机化。载于*2016*年*ACM SIGSAC*计算机与通信安全会议论文集，CCS '16，第368–379页，美国纽约州纽约市，2016年。计算机协会。

[29] Daniel Gruss、Clémentine Maurice、Klaus Wagner 和 Stefan Mangard。Flush+flush：一种快速而隐蔽的缓存攻击。载于《第*13*届入侵与恶意软件检测及漏洞评估国际会议论文集》第*9721*卷，DIMVA 2016，第279–299页，柏林、海德堡，2016年。施普林格出版社。[30] David Gullasch、Endre Bangerter 和 Stephan Krenn。缓存游戏——将基于访问的缓存攻击应用于 AES 实践。载于《*2011*年*IEEE*安全与隐私研讨会》，第490–505页，2011年。[31] Yanan Guo、Andrew Zigerelli、Youtao Zhang 和 Jun Yang。对抗性预取：新型跨核心缓存侧信道攻击。载于《*2022*年*IEEE*安全与隐私研讨会（*SP*）》，第1458–1473页，2022年。[32] J. Alex Halderman、Seth D. Schoen、Nadia Heninger、William Clarkson、William Paul、Joseph A. Calandrino、Ariel J. Feldman、Jacob Appelbaum 和 Edward W. Fel- ten。以免我们遗忘：针对加密密钥的冷启动攻击。《*ACM* 通讯》，52(5)：91–98，2009年5月。[33] Hector Martin。Mastodon 帖文：marcan@treehouse.systems。https://social.treehouse.systems/@marcan/112136566078412530。[访问日期：04-11-2024],已不再提供。[34] Tyler J Huberty、Stephan G Meier 和 Mridul Agarwal。内容定向预取电路及质量过滤，2018年2月6日。美国专利9,886,385。[35] Sorin Iacobovici、Lawrence Spracklen、Sudarshan Kadambi、Yuan Chou 和 Santosh G Abraham。有效的基于流和基于执行的数据预取。载于《第*18*届年度国际超级计算会议论文集》，第1–11页，2004年。[36] 英特尔公司。数据依赖预取器，2023年。访问日期：2025年4月12日。

[37] 英特尔公司。数据操作数无关时序指令集架构（ISA）指南，2023年。访问日期：2025年4月12日。

[38] Anirudh Mohan Kaushik、Gennady Pekhimenko 和 Hiren Patel。Gretch：一种用于图分析的硬件预取器。*ACM*体系结构与代码优化汇刊，2021年。

[39] Paul Kocher、Daniel Genkin、Daniel Gruss、Werner Haas、Mike Hamburg、Moritz Lipp、Stefan Mangard、Thomas Prescher、Michael Schwarz 和 Yuval Yarom。Spectre 攻击：利用推测执行。载于 *S&P'19*，2019年。

[40] Chris Lattner 和 Vikram Adve。LLVM：用于程序全生命周期分析与转换的编译框架。载于 *CGO'04*。

[41] Moritz Lipp、Daniel Gruss 和 Michael Schwarz。通过功耗和时间进行 AMD预取攻击。在 第*31*届*USENIX*安全研讨会（*USENIX Security 22*），第643–660页，2022年。[42] Moritz Lipp、Michael Schwarz、Daniel Gruss、Thomas Prescher、Werner Haas、Stefan Mangard、Paul Kocher、Daniel Genkin、Yuval Yarom 和 Mike Hamburg。熔断：从用户空间读取内核内存。在*USENIXSecurity'18*，2018年。[43] Fangfei Liu、Yuval Yarom、Qian Ge、Gernot Heiser 和 Ruby B Lee。末级缓存侧信道攻击具有实用性。在 *2015*年*IEEE*安全与隐私研讨会，第605–622页。IEEE，2015年。[44] Xiaoxuan Lou、Tianwei Zhang、Jun Jiang 和 Yinqian Zhang。密码学中的微架构侧信道漏洞、攻击与防御综述。*ACM*计算调查（*CSUR*），54(6)：1–37，2021年。[45] nsurbay、Cédric Tessier、Charles Hubain、toffan、pbrunet、Jonathan Salwan、Romain Thomas、bla5r、Christian Heit-man、badguy123、hellodword、red thing、J. Ryan Stin-nett、Sal、pistach3 和 Robert Yates。Qbdi/QBDI。https://github.com/QBDI/QBDI，2025年10月15日。[46] Dag Arne Osvik、Adi Shamir 和 Eran Tromer。缓存攻击与对策：AES案例。在 *CT-RSA'06*，2006年。[47] Anna Pätschke、Jan Wichelmann 和 Thomas Eisen-barth。Zebrafix：通过交错缓解以内存为中心的侧信道泄漏。*arXiv*预印本 *arXiv:2502.09139*，2025年。[48] Ashay Rane、Calvin Lin 和 Mohit Tiwari。Raccoon：通过混淆执行关闭数字 {侧信道} 。在 第*24*届*USENIX*安全研讨会（*USENIX Security 15*），第431–446页，2015年。

[49] Matthieu Rivain 和 Emmanuel Prouff。AES的可证明安全高阶掩蔽。在国际密码硬件与嵌入式系统研讨会，第413–427页。施普林格，2010年。

[50] Amir Roth、Andreas Moshovos 和 Gurindar S Sohi。用于链式数据结构的基于依赖的预取。载于 第八届国际编程语言与操作系统体系结构支持会议论文集，第 115–126 页，1998年。

[51] Jose Rodrigo Sanchez Vicarte、Pradyumna Shome、Nandeeka Nayak、Caroline Trippel、Adam Morrison、David Kohlbrenner 和 Christopher W. Fletcher。《打开潘多拉魔盒：系统研究微体系结构泄露私有数据的新方式》。发表于国际计算机体系结构研讨会，2021年。

[52] Youngjoo Shin、Hyung Chan Kim、Dokeun Kwon、Ji Hoon Jeong 和 Junbeom Hur。揭示基于硬件的数据预取器：信息泄漏的隐藏源。在计算机与通信安全会议，2018年。[53] Jakub Szefer。微体系结构侧信道与隐蔽信道、攻击和防御综述。硬件与系统安全期刊，3(3)：219–234，2019年。[54] Jose Rodrigo Sanchez Vicarte、Michael Flanders、Ric- cardo Paccagnella、Grant Garrett-Grossman、Adam Mor- rison、Christopher W Fletcher 和 David Kohlbrenner。Augury：利用数据内存依赖型预取器泄漏静态数据。在*2022*年*IEEE*安全与隐私研讨会（*SP*），第1491–1505页。IEEE，2022年。[55] Alan Wang、Boru Chen、Yingchen Wang、Christopher W. Fletcher、Daniel Genkin、David Kohlbrenner 和 Ric- cardo Paccagnella。Peek-a-Walk：通过页表遍历侧信道泄漏秘密数据。在*2025*年*IEEE*安全与隐私研讨会（*SP*），2025年。[56] Michael E Wolf 和 Monica S Lam。一种优化数据局部性的算法。在*1991*年*ACM SIGPLAN*编程语言设计与实现会议论文集，第30–44页，1991年。[57] Meng Wu、Shengjian Guo、Patrick Schaumont 和 Chao Wang。使用程序修复消除时序侧信道泄露。在第*27*届*ACM SIGSOFT*国际软件测试与分析研讨会论文集，第15–26页，2018年。[58]Yuval Yarom 和 Katrina Falkner。{FLUSH+RELOAD}：一种高分辨率、低噪声的三级缓存 {侧信道} 攻击。在第*23*届*USENIX*安全研讨会（*USENIX*安全大会*14*），第719–732页，2014年。[59] Xiangyao Yu、Christopher J Hughes、Nadathur Satish 和 Srinivas Devadas。Imp：间接内存预取器。在第*48*届微架构国际研讨会，第178–190页，2015年。

[60]Zhiyuan Zhang、Mingtian Tao、Sioli O'Connell、Chitchanok Chuengsatiansup、Daniel Genkin 和 Yu- val Yarom。BunnyHop：利用指令预取器。在*USENIX* 安全大会，2023年。