MEMORY DISORDER：内存重排序作为无计时器侧信道

Sean Siddens†
华盛顿大学

Sanya Srivastava
杜克大学

Reese Levine
加州大学圣克鲁兹分校

Josiah Dykstra†
雷神*BBN*科技公司

Tyler Sorensen†
微软研究院
加州大学圣克鲁兹分校

arXiv:2601.08770v1 [cs.CR] 13 Jan 2026

摘要

为了提高效率，几乎所有并行处理单元（CPU 和 GPU）都采用宽松内存模型，在这种模型下，内存操作可能会发生重排序，即乱序执行。此前在这一领域的研究发现，当其他核心处于活动状态（例如，对内存系统施加压力）时，观察到内存重排序的频率会更高，这可能是因为触发了激进的硬件优化。

在这项工作中，我们提出 MEMORY DISORDER：一种利用内存重排序推断其他进程活动的无计时器侧信道。我们首先开展模糊测试活动，并表明许多主流处理器（X86/Arm/苹果 CPU、NVIDIA/AMD/苹果 GPU）容易受到跨进程信号的影响。随后，我们展示了如何利用这一漏洞实现经典攻击，包括隐蔽信道和应用指纹识别：前者在 Apple M3 GPU 上可实现高达 16 比特/秒的传输速率，准确率达 95%；后者则能在多款 CPU 和 Apple M3 GPU 上可靠地实现封闭世界DNN架构指纹识别。最后，我们探讨如何利用底层系统细节增加重排序，表明隐蔽信道在 X86 CPU 上有望实现接近 30K 比特/秒的传输速率。随着对该漏洞的理解不断加深，未来可能开发出更精确的攻击。

1 引言

现代共享内存并行处理器采用复杂的流水线和内存系统。这种复杂性提高了效率，但也引发了安全问题，尤其是在多处理系统中：一个进程的敏感信息可能被另一个（可能具有恶意的）进程获取。侧信道攻击尤为隐蔽，因为攻击者能够利用底层设备细节，检测看似无害的受害进程的细微特征。

侧信道有许多类型，利用许多不同的系统组件。一些侧信道需要物理访问 [49]，而其他则需要访问底层API，例如，用于监测能耗 [14]。攻击要求被称为攻击能力。一种低能力攻击的要求很少，因此可以在更多情况下部署。例如，经典的基于缓存的侧信道需要精确计时器 [26]。然而，可以通过移除计时器并改用其他（看似无害的）机制来降低攻击能力 [38]。

MEMORY DISORDER 在这项工作中，我们提出了 MEMORY DISORDER（简称 DISORDER）：一种面向并行处理器、能力要求较低的侧信道。DISORDER 利用了硬件宽松的内存一致性模型（或称内存一致性模型）所引发的内存重排序 [32]。也就是说，为了提高效率，主流消费级并行处理器允许内存操作按照其内存一致性模型的规定乱序执行。这些行为在 CPU [4, 39] 和 GPU [22]上都会出现。内存重排序可以通过小型并行程序观察到，这类程序通常称为*Litmus* 测试，通常由两个线程组成，每个线程各执行两次内存操作。检查测试执行后某些变量和内存值，即可确定是否发生了重排序。

先前研究在许多不同的处理器上运行这些 Litmus 测试，以检验它们是否符合内存一致性模型规范。在测试 GPU 时，研究发现，利用额外线程给系统施加压力，可以增加观测到的重排序数量 [3]。受这些结果启发，本文探讨将内存重排序用作侧信道的可能性：某个（可能怀有恶意的）进程反复执行 Litmus 测试，只需具备启动线程以及执行内存读写的能力。其他（受害者）进程在执行特定程序时，可能会产生可识别的系统压力模式，导致另一个进程观测到重排序。

†此项工作是在 Trail of Bits 任职期间完成的。

初始状态：*x == 0 && *y == 0

线程 0	线程 1
ⓐ 写入(x, 1)	ⓒ v0 = 读取(y)
ⓑ 写入(y, 1)	ⓓ v1 = 读取(x)

重排序检查：v0 == 1 && v1 == 0

图 1：消息传递（MP）测试。线程 0 先写入 x，再写入 y。线程 1 按相反顺序读取这些值。如果线程 1 在观察到 y 的更新值之前，还未观察到 x 的更新值，那么就发生了重排序。

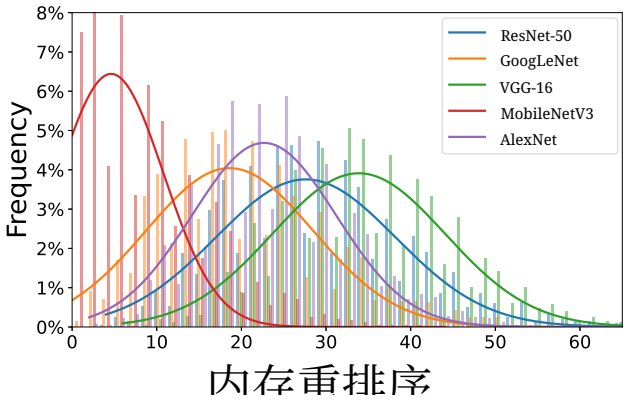

图 2：受害者在 Apple M3 GPU 上运行常见深度神经网络架构的推理时，攻击者观测到的重排序直方图。

1.1 示例：深度神经网络架构指纹识别

我们通过一个包含两个进程的示例来说明DISORDER的潜力：一个受害者和一个攻击者。受害进程反复在 Apple M3 GPU 上运行深度神经网络推理，该 GPU 已知采用较宽松的内存一致性模型 [19]。攻击者反复执行一种称为消息传递（或 MP）的经典 Litmus 测试，如图1所示。要满足该测试的重排序检查条件，线程 0 上的写入或线程 1 上的读取必须发生重排序。由于这些测试很短，攻击者可以在受害者运行深度神经网络推理期间执行许多轮测试。

图 2 展示了攻击者观察到的内存重排序频率直方图。此时，受害者正在运行 5 种不同的（常见）深度神经网络架构：resnet50、googlenet、vgg16、mobilenetv3_small（在图中标记为 mobilenetv3）和 alexnet。我们发现，每种深度神经网络都会产生独特的特征，即观察到的重排序近似呈正态分布。接着，我们将这些观测结果划分为训练集和测试集（见第 5.2 节），并表明攻击者能够自动识别受害者正在运行哪种深度神经网络架构。

1.2 MEMORY DISORDER 概述

本文详细研究了内存重排序作为侧信道的潜力。尽管存在许多能力有限的侧信道，例如参见 [51], 我们认为 MEMORY DISORDER 具有一系列独特属性：

1. 低门槛：DISORDER 只需要两个线程和共享内存；不需要计时器、性能计数器或物理访问。

2. 覆盖面广：主流处理器（CPU 和 GPU）实现了宽松的内存一致性模型，可通过常见编程语言访问，例如 C++ [11]。

3. 基于实证：许多侧信道通过细致的逆向工程针对底层硬件组件 [2, 33, 51]。DISORDER 攻击可以使用简单的实证模糊测试技术构造。

4. 潜力与缓解措施尚不明确：导致内存重排序的确切硬件组件尚无文档记载，也未得到充分理解。因此，难以制定严谨的缓解措施，而且很可能通过详细探索开发出更有针对性的攻击。

本文聚焦于可行性和广泛性DISORDER：我们表明，六种不同架构都受到影响，涵盖 CPU 和 GPU；此外，我们还表明，这种漏洞会跨越虚拟化边界（Linux 上的 KVM ）。虽然无法对所有这些设备中导致内存重排序的复杂硬件交互进行逆向工程，但我们初步探究了更有针对性的方法如何将 X86 CPU 上重排序观测值（即信号）的数量级提高数倍（第 5.3）。对于其他情况，我们在有资料可用时提出假设并提供参考文档，将更深入的探索留待未来工作。因此，DISORDER 攻击既可以（1）仅凭简单的实证数据以低精度实施，也可以（2）通过彻底的逆向工程构造，以实现更高精度。

我们只发现了一项此前提出重排序侧信道的安全研究 [12]。然而，该研究仅在 X86 CPU 上进行了小规模的简单实验，并未探讨如何在不同架构中系统地寻找信号。此外，DISORDER 攻击需要两个线程和共享内存；已有研究表明，这些条件往往足以构造高分辨率计时器（例如，参见 [38]），从而重新启用经典的基于缓存的侧信道。然而，鉴于内存重排序的确切原因仍未得到充分探索，我们认为 DISORDER 具有新的潜力，因此不应仅将其视为基于缓存的侧信道的替代方案。

针对 DISORDER 的模糊测试 我们首先设计一项模糊测试活动，以确定是否能够观测到一个信号

表1：本研究中使用的设备和配置。我们在所有设备上都观察到了 DISORDER。我们针对 Arm、X86、M1-CPU 和 M3-GPU 构造了其他攻击（以绿色突出显示）。

Type	供应商	Name	简称	核心或计算单元	配置
CPU	苹果	M1	M1-CPU	8	macOS 不同进程
CPU	Arm	A78（Nvidia Jetson Orin Nano）	Arm	6	Linux 不同进程、KVM
CPU	Intel	i7-12700K	X86	12	Linux 不同进程，KVM
GPU	苹果	M3	M3-GPU	10	macOS 不同进程
GPU	NVIDIA	GeForce RTX 4070	NVIDIA	46	Linux 不同进程
GPU	AMD	Radeon RX 7900 XT	AMD	84	Linux 不同进程

跨进程传播，并利用内存重排序（第 3 节）。该模糊测试包含两个参数化进程：压力器 和监听器。监听器运行内存重排序测试用例，而压力器施加系统压力。我们记录压力器执行和未执行时的重排序观测值数量，并检查两者是否存在差异。如果存在差异，监听器便可能确定自己是否正在与压力器并发运行。

我们研究了六种设备：三款 CPU 和三款 GPU，列于表 1。尽管信号强度各不相同，我们仍能在这些设备和配置上观测到信号。我们称能够观测到信号的设备易受 DISORDER 攻击。

DisOrder 攻击随后，我们选取部分设备（表 1中以绿色标出），并利用模糊测试活动的数据实施两种经典攻击：隐蔽信道（第 5.1 节）和应用指纹识别（第 5.2 节）。对于 Arm 和 X86 设备，我们跨越 KVM 虚拟化边界实施这些攻击。

为了实现隐蔽信道，我们找出能够产生高、低信号的测试与压力测试组合，并据此实现隐蔽信道，使其能够以最高 16 比特/秒的速率可靠通信（在 Apple M3 GPU 上）。为了实现应用指纹识别，我们在受害应用程序运行时，同时运行选定的监听器。我们将收集到的数据分为训练集和测试集，以模拟攻击。我们证明，监听器（攻击者）只需大约 5 秒，就能成功判断正在运行的是哪种候选深度神经网络架构；在 Intel X86 CPU 上，准确率最高可达 100%。我们还证明，其他应用程序活动（例如启动 Google Chrome）也会产生显著的 DISORDER 信号。

第5.3 节通过探讨如何利用底层系统细节来增加重排序，从而提高攻击精度，作为本节的结尾。我们展示了，在 X86 CPU 上访问特定的一级缓存组，可以大幅增加重排序的次数，从而有望将隐蔽信道速率提高到近 30k 比特 / 秒。这项探索可作为构建更精准的 DISORDER 攻击的基础。

贡献 总之，我们的贡献如下：

- MEMORY DISORDER：一种新颖的低能力侧信道攻击，利用内存重排序。
- 一项模糊测试活动，表明许多主流 CPU 和 GPU 容易受到 DisOrder （第 3 节）。
- 阐述如何构建 DisOrder 隐蔽信道（第5.1 节）以及应用指纹识别（第5.2 节）。
- 探讨如何利用底层系统细节，大幅增加 DisOrder (第5.3节）。

第 6 节讨论缓解措施和补救办法，负责任披露则见第9节。

2 背景

由于我们的研究对象既包括 CPU，也包括 GPU，因此我们简要介绍它们的多处理能力，并定义一些常用术语。接下来，我们概述内存一致性模型及其允许的重排序。

2.1 并行架构与多处理

几乎所有主流计算设备（笔记本电脑、手机、服务器）都包含并行处理单元（如 CPU、GPU），使硬件资源能够同时执行任务。这些处理器通常支持多处理，即来自不同进程的任务可以并发执行。进程构成了明确的安全边界——每个进程都应与其他所有进程隔离。虚拟化则提供了额外的安全边界，使恶意进程无法利用操作系统（OS）漏洞。

CPU CPU 是系统的处理中心。它们执行大部分底层系统逻辑（例如操作系统），也是许多交互式应用程序（如网页浏览器）的主要计算组件。CPU 针对低延迟进行了优化，少量复杂的核心 可优化单条指令流（通常称为线程）。不同线程通常会映射到不同核心。在某些

系统中，例如 Linux，可以通过底层 API 将软件线程固定到核心上。主流 CPU 操作系统（Linux、macOS）提供并行多处理功能，使不同进程能够在不同核心上并行执行。

GPU GPU 是高度并行的可编程加速器。其并行性采用层级结构组织，基本单元是 处理单元，它负责计算一串计算任务（同样称为线程）。处理单元被组织成 计算单元 (计算单元)[1]。与 CPU 不同，GPU 面向吞吐量，使用更简单的硬件组件并行计算多个任务。

GPU 程序称为内核；它由主机（CPU）程序启动，该程序指定线程的层级配置，即线程总数以及将线程划分为工作组的方式，并保证这些线程在同一计算单元上执行。GPU 的执行模型较为复杂，例如，有些线程会同步执行，但 DISORDER 并不需要了解这些细节。

GPU 的多处理能力和属性因系统及配置而异。传统上，GPU 不会并行执行来自不同进程的内核；相反，它们会依次执行不同的内核 [41]。然而，GPU 支持内核并行执行的情况日益普遍，即在不同计算单元上执行不同内核，例如通过 NVIDIA 的 MPS 等配置 [35] 和 MIG [34]。在第 4.2 节中，我们通过实证表明，Apple GPU 也支持内核并行执行。

CPU/GPU 共性 CPU 和 GPU 都提供通用编程接口，允许启动多个线程并在不提升权限的情况下访问内存。这些内存访问会可靠地转换为硬件内存访问，即不会经过编译器优化，并使用原子操作库。此外，CPU 和 GPU 都包含复杂的内存层次结构，其中至少有两级缓存和主存。这种共同的设计可能在一定程度上导致了这两类设备上可观察到的内存重排序。

2.2 内存一致性模型

内存一致性模型（MCM）定义了在共享内存并行程序中，加载指令允许返回哪些值。这类模型既有语言模型（例如，针对 C++ [11]），也有硬件模型（例如，针对 X86 [37] 和 Arm [13]）。这些模型可能很复杂，经过多年发展和完善。然而，本研究并不需要当今最先进的内存一致性模型所具备的全部复杂性。我们对 DISORDER 的初步探索只需要一个简单的基础来进行实证测试，具体如下文所述。

[1]在 NVIDIA 文档中称为流式多处理器 (SM)；不过，在本文中我们采用可移植性更强的 Khronos 术语

表 2：将图 1中的多处理器测试推广后，会得到另外五个经典 Litmus 测试。

测试名称	ⓐ	ⓑ	ⓒ	ⓓ	重排序检查
消息传递（MP）	*W*	*W*	*R*	*R*	v0=1& v1=0
存储缓冲（SB）	*W*	*R*	*W*	*R*	v0=0 & v1=0
加载缓冲（LB）	*R*	*W*	*R*	*W*	v0=1 & v1=1
2+2W	*W*	*W*	*W*	*W*	*x=1 & *y=2
存储（S）	*W*	*W*	*R*	*W*	*x=2 & v0=1
读取（R）	*W*	*W*	*W*	*R*	*y=2 & v0=0

指令重排序内存一致性模型本研究考察的内存一致性模型可以完全用线程局部重排序来定义，它们是 [10],中所述模型的一个子集，我们称之为 IR（指令重排序）模型。例如，某个 IR 模型可能允许 写-读 （或 *WR*）重排序。也就是说，如果程序中先有一次写入（*W*），随后有一次读取（*R*），那么二者就允许重排序。鉴于我们只考虑两种内存指令——写入（*W*）和读取（*R*）——可能发生重排序的指令对只有四种（*RR*、*RW*、*WR*和 *WW*）。枚举允许发生哪些重排序，即可定义一个 IR 模型。

IR模型通常不允许对访问同一位置的操作进行重排序，因为这可能破坏单线程顺序执行。也就是说，允许读写 重排序的IR模型，只有在这两个访问针对不同位置时，才允许重排序。在讨论每种测试处理器时（第 3 节），我们将说明哪种IR内存一致性模型最能近似描述该处理器允许的行为。

IR指令序列测试 IR模型简洁明了，因此可以通过简单的 Litmus 测试进行全面测试。我们可以推广图 1 中的多处理器测试，其中每条指令（ 、 、ⓒ、ⓓ）都可以实例化为 *R* 或 *W*。始终存在两个内存位置，x 和 y。内存操作 和ⓓ 针对 x，而内存操作 和 ⓒ 针对 y。写操作存储唯一值（从 1 开始，每次写入递增），读操作则存入唯一变量（从 v0 开始，每次加载递增）。每个测试最多可以检查两次重排序，每个线程一次。例如，图 1 中的多处理器测试同时检查 *WW* 重排序（线程 0）和*RR*重排序（线程 1）。

根据这一公式，我们可以构造 6 个 Litmus 测试，通过后置条件检查 IR 重排序。这些测试对应于内存一致性模型文献中广泛使用的经典 Litmus 测试，名称虽说颇为晦涩。我们在表 2 中介绍这些测试；在先前研究中也能更直观地看到它们，例如 [21]。每种测试所涉及的重排序并不唯一，例如，SB 和 R 都用于测试 *WR*重排序。

表 3：不同的测试框架（绿色阴影部分）和压力测试（蓝色阴影部分）、它们针对的设备（CPU 或 GPU），以及模糊测试参数的数量。

Name	设备	模糊测试参数
基本	CPU	2
Litmus7 [6]	CPU	0
永久 [30]	CPU	2
GPU 并行 [23]	GPU	4
内存 [21]	CPU、GPU	6
线程启动	CPU	2

3 DISORDER 漏洞模糊测试

现在，我们将详细介绍一种模糊测试方法，用于检查设备是否容易受到 DisOrder 的影响。该方法包含两个进程：监听器和 压力器。监听器可采用四种不同的内存排序测试技术之一来实例化，压力器则可采用两种系统压力技术之一来实例化。对于每种技术（测试和施压），我们都会概述其方法，并列出模糊测试参数，汇总见表 3。

3.1 监听器

监听进程运行 Litmus 测试多次迭代（通常超过 100K 次），并记录重排序的数量。尽管 Litmus 测试很简单，但运行这些测试可能很复杂，需要利用启发式方法来提高吞吐量并增加重排序。测试框架 接收一个 Litmus 测试和一些可调参数。我们采用了先前研究中的三个框架，并提供了一个新的框架。

基础测试框架 我们提供了 本研究中新提出的基本测试框架 。虽然它很简单，但结合适当的压力测试后，它能提供一些最可靠的信号（见第 4.1 节中的 M1-CPU 结果）。该框架使用 C++ 线程和宽松的原子内存访问来实现 Litmus 测试，从而允许所有 IR内存重排序。每次测试迭代中，线程先启动，然后汇合，最后检查重排序条件。

该框架包含两个模糊测试参数：两个内存位置的索引，即 x 和 y。对索引进行模糊测试，可以让它们位于不同的内存区域（或同一区域）中，例如不同的缓存行或内存页，从而可能促使出现不同类型的重排序。我们仅针对 CPU 系统实现了该框架，因为 GPU 并不广泛支持 C++并发构造。

Litmus7框架 Litmus7工具 [6] 在基本框架之上实现了多种启发式方法。例如：内存操作通过内联汇编实现并且线程不会在每次迭代时重新启动，而是使用线程间同步来执行重排序检查并对齐下一次迭代。此外，Litmus7还会自行进行模糊测试，随机分配内存位置，并随机排列软件线程到测试线程的映射。鉴于此，我们使用该框架的默认设置运行，不对任何自有参数进行模糊测试。由于Litmus7仅面向CPU（先前的GPU版本 [3] 并未公开），我们仅使用该框架测试CPU系统。

Perpetual框架 永久测试早期曾用于内存一致性模型测试 [48],，近期又重新受到关注 [30,42]。这种方法只需启动一次测试线程，通过消除同步来提高吞吐量。写入操作存储代数序列，而读操作则维护一份记录观测值的日志。测试运行结束后会分析日志；满足特定代数约束的轨迹表明发生了内存重排序。这种方法对 DISORDER 很有吸引力，因为它能提供细粒度的观测值，有望实现更精确的攻击。

为支持更广泛的测试，我们使用 C 重新实现了 [30] X86 专用方法++，采用原子操作和线程。此方法的模糊测试参数与基本框架相同，即两个内存索引。此外，无需进行重大修改，此方法就要求测试中包含两个读操作；因此，我们将此测试框架支持的 Litmus 测试限制为 MP、SB 和 LB。在第 4 节中，我们发现遗憾的是，该框架无法在 Arm 或 X86 CPU 上观测到任何重排序。先前研究 [30] 曾观测到 X86 重排序，因此我们的实现可能还需要更精细的调优。由于此框架最适合快速执行少量线程，因此更适用于 CPU，我们也就没有提供 GPU 实现。

GPU 并行测试框架 该框架 [23]利用 GPU 并行性实现高吞吐量测试。给定 N 个 GPU 线程后，该框架会实例化 N 个 Litmus 测试并行执行。每个线程执行测试的两个实例：首先充当线程 0，随后充当线程 1。该方法通过 WebGPU [45],实现，因此可移植到多种 GPU。该框架包含四个模糊测试参数：工作组数量、工作组大小、内存位置之间的填充，以及测试线程同步的频率。每次迭代还会将 Litmus 测试线程随机映射到 GPU 线程。

3.2 压力器

压力器执行一种压力：一种旨在对系统施加压力并增加监听器所观察到的内存重排序的程序。在本研究中，我们探讨了两种能够增加内存重排序的压力技术。其中一种技术

受到先前研究的启发，另一种则是本研究提出的新技术。这些压力进程会无限期运行，直到被终止。

内存压力 顾名思义，这种压力针对内存系统，其设计紧密遵循 GPU 内存一致性模型一致性测试领域的先前研究 [21]。这种压力会分配一个内存缓冲区，并将其划分为压力行（概念上类似于缓存行）。随后启动线程，并将它们映射到初始压力行。线程会反复按照由加载和存储组成的模式访问压力行。经过若干次迭代后，线程会转移到另一条压力行。

我们提供了 C++ 和 WebGPU 版本的内存压力实现，并首次将内存压力应用于 CPU 内存模型测试。这项技术包含 6 个模糊测试参数：压力行大小、线程在压力行内的偏移量、针对给定压力行的迭代次数、用于更新内存位置的步长、内存访问模式（由一系列加载和存储组成），以及施加压力的线程数量。

线程启动压力 这种压力是在早期实验过程中偶然发现的。我们发现，在测试运行期间打开一个新的终端，会显著增加重排序的数量。我们将这一行为提炼为一种压力方式，即反复启动并汇合线程。

我们用 C++ 实现这种压力。线程执行一个包含宽松原子内存访问的循环，以防编译器将循环优化掉。这种压力的两个模糊测试参数是线程数和线程循环迭代次数，后者控制线程重新启动的频率。由于 GPU 没有类似的线程启动机制，而是需要启动整个内核，因此我们仅为 CPU 提供这种压力。

3.3 检查 MEMORY DISORDER

要检查 DISORDER，需要将监听器与测试框架及其输入实例化，即一个 Litmus 测试和任何必需的参数。随后，它会执行指定次数的*Litmus* 测试迭代次数。这称为一次测试运行。测试运行结束后，监听器会报告观察到的重排序次数，即重排序频率。压力器通过某种压力及任何必需的参数进行实例化。如果压力器执行时，监听器观察到的重排序频率稳定地更高，我们就称压力器能够向监听器发出信号 。如果观察到信号，则说明系统容易受到 DISORDER 的影响。

为检查信号是否可靠，监听器会在隔离状态下进行 X 次测试运行，生成一组包含 X 个重排序频率样本的集合 B （基线）。随后，压力器开始运行，并持续运行直至被终止。监听器再次进行 X 次测试运行，同时与压力器运行，生成另一组样本集合 S （受压状态）。然后，我们使用统计方法（曼-惠特尼 U 检验 [29]）比较这两组集合，以确定一组中的值是否大于另一组中的值。我们还计算共同语言效应量（CLES），即从 S 中随机选取的元素大于从 B 中随机选取的元素的概率。如果 CLES 为 100%，也就是说，S 中的所有元素都大于 B 中的元素，那么我们就认为该信号是可靠的。此外，对于每项观察到信号的测试，我们都会记录观测到的内存重排序平均增幅。每项此类统计检验（即包含 X 次测试运行，分别在有压力和无压力的情况下进行）称为一次试验。

模糊测试为了针对 DisOrder进行模糊测试，可以运行一个简单脚本，遍历每种压力s、每种测试框架f（表3），以及每个适用的 Litmus 测试t （表2）。脚本使用f实例化监听器，并将t 用作输入测试。使用s实例化压力器。随后，模糊测试器会随机选择其他参数并进行一次试验。模糊测试器会执行许多信号测试，并记录测试结果。

4 模糊测试结果

现在，我们详细介绍在表1所列设备上开展模糊测试活动的结果；我们能够在所有设备上观察到具有统计显著性（且可靠）的信号。因此，DISORDER漏洞广泛存在于主流 *CPU* 和 *GPU* 中。我们选择 10 次测试运行来创建在试验中进行比较的样本集（B 和S）。我们为每台设备配置模糊测试活动，使其大约在 8 小时内（过夜）完成；这包括设置 Litmus 测试迭代次数和试验次数等参数。

我们在表中汇总了所有处理器和配置下的发现。 4 对于每种配置，我们报告出现信号的试验所占百分比，包括 任意 信号，即观测集之间存在统计差异，以及可靠 信号，即 CLES 为 100%。请注意，可靠信号的百分比会低于（或等于）任意信号的百分比。我们汇总了观测到的重排序增幅百分比，包括平均值和最大值。接着，我们展示所有信号的平均 CLES，以了解重排序增幅出现的频率。最后，我们展示进行了多少次试验。由于处理器速度和测试框架吞吐量各不相同，不同设备和配置的试验次数也有所不同。

在运行 Linux 的 Arm 和 X86 CPU 上，我们能够显式设置线程亲和性，即将程序线程映射到硬件核心。对于这些设备，我们考虑了与攻击者能力相关的两种配置：一种是可以显式映射亲和性（称为 显式绑定），另一种是由操作系统管理亲和性。显式绑定所需的能力严格高于操作系统管理，因为它要求攻击者能够将监听器和压力线程显式映射到特定的设备核心。为了找到亲和性映射，我们针对一系列映射进行了多次先导实验，并选取平均 CLES 最高的映射。

鉴于每台设备都有其自身的特性和需要考虑的因素，我们将在下文分别讨论。

4.1 CPU 结果

Arm 我们测试的 Arm 处理器有 6 个核心，运行 Ubuntu 22.04。Arm 设备采用以宽松著称的内存模型 [13],但其规范往往比实际实现更加宽松。在任何 Arm 系统上都无法观测到 LB 测试，因此我们将其从针对该处理器的模糊测试活动中排除。对于显式亲和性映射，我们将测试线程固定为仅在核心 0 和 5 上运行，同时将压力线程固定在核心 1 至 4 上。

线程启动压力结合基本测试框架，能够为该设备提供最可靠的信号。根据是否显式设置亲和性，大约在 12% 或 64% 的时间内可以获得可靠信号，这凸显了该功能所带来的额外益处。在这两种情况下，观测到的重排序最大增幅均约为 1K%，能够提供易于区分的信号。使用 litmus7 测试框架进行内存压力测试时，在不进行显式 固定的情况下，信号出现的频率较低，最多仅占测试的 25%，可靠信号最多占 7%；不过，重排序的最大增幅与线程启动压力相近。尽管持续测试可能提高精度，但我们无法使用这种方法观测到任何信号。

X86 我们的 X86 处理器运行 Ubuntu 22.04，据文档记载，它有 12 个核心，其中 8 个是支持超线程的（×2）性能核心，另外 4 个是能效核心。与 Arm 相比，X86 系统提供了更强的内存模型 [39],，只允许 *WR* 重排序。因此，在针对该设备的这次模糊测试活动中，我们使用 SB 和 R。与 Arm 类似，我们在该设备上进行了显式绑定和不进行显式绑定两种测试。为了确定合适的绑定方式，我们针对多种亲和性映射开展了先导实验，发现当两个测试线程分别绑定到性能核心 0 和 1 时，平均 CLES 最高。由于性能核心支持超线程，这两个核心（0 和 1）是运行在同一个核心上的硬件线程。

与 Arm 不同，X86 提供了关于导致内存重排序的硬件机制的文档。每个处理器都包含一个存储缓冲区，用于在写入刷新到内存子系统之前暂存这些写入。由于存储缓冲区是处理器本地的，我们最初假设它们不会受到跨进程压力测试的影响。然而，我们的结果（汇总于表4）表明事实并非如此。当核心被显式固定时，我们有时能观察到高达 100% 的信号；使用 litmus7 测试框架和线程启动压力测试时，信号的可靠性达到 71%。事实上，即使没有显式固定，这种测试与压力测试的组合仍然非常有效。不过，内存压力也能在基本框架下产生许多信号，但对显式固定更为敏感（82% 对 3%）。与 Arm 类似，永久测试框架未能为该设备提供任何信号。

虽然存储缓冲区位于处理器本地，但这些结果表明，它们容易受到跨进程内存流量的影响，而这似乎会影响刷新触发的时机，例如，为了避免内存系统过载。我们对 X86 的进一步研究（第 5.3 节）将对此进行更深入的探讨。这些结果充分展现了底层组件的复杂程度，也说明 DISORDER 攻击可能会多么微妙。

M1-CPU Apple M1-CPU 总共有 8 个核心，由 4 个性能核心和 4 个能效核心组成。尽管关于这款处理器的文档很少，但它实现了 Arm 指令集架构，因此也允许出现类似的重排序。一项初步研究表明，我们可以在该设备上观察到所有 IR 重排序，唯独无法观察到 LB（与 Arm 一样）。因此，我们也将 LB 排除在该设备的模糊测试活动之外。与 Linux 不同，macOS 不支持线程亲和性映射，因此无法运行显式绑定的实验。

回顾表 4 中该设备的结果，可以发现一些与 Arm 类似的趋势：例如，最可靠的信号来自线程启动压力，尤其是与 litmus7 结合时。不过，线程启动压力与基本测试框架结合时，百分比增幅最大，在我们的 CPU 实验中达到最高值，增幅高达 770K%。与其他 CPU 类似，内存压力也能产生较常见的信号和较大的增幅（使用 Litmus7 时比使用基本框架更明显）。与 Arm 和 X86 不同，持续测试在这款处理器上也能检测到一些信号。不过，与其他方法相比，其可靠性和百分比增幅都较低。

4.2 GPU 结果

我们测试的所有 GPU 均有文档说明其采用非常宽松的内存模型 [22]。我们在研究中的 GPU 上，针对表 *2* 中的所有 测试都观测到了内存重排序。在 GPU 上，重排序观测值的百分比增幅非常高；事实上，如果没有某种压力，通常根本观测不到内存重排序（见 [3]）。为避免基线为零时出现无穷大的百分比增幅，我们将基线计数设为 1。

AMD Radeon RX 7900 XT 我们的 AMD GPU 提供内核顺序执行模型（回顾第 2.1 节）。尽管监听器和压力器在 GPU 上并非并行执行，先前研究已表明，仍可观测到跨 GPU 内核的内存效应，例如在 Prime-and-Probe 风格攻击中 [15]。表 4 底部附近展示的结果表明，在这种执行模型下也能观测到 DISORDER。不过，该设备上的信号十分罕见，但仍存在一些可靠信号（出现概率低于 6%），最大百分比增幅为 170K。

表 4：我们在各设备上进行模糊测试活动的结果。

				信号 %		增幅 %			
设备	**固定?**	**测试 F.**	**压力**	**Any**	**可靠**	**Avg.**	**Max**	**CLES（平均值）**	**试验**
Arm	no	基本	内存	10.93	0.94	519	1703	0.86	320
			线程启动	59.06	11.87	432	2700	0.92	320
		lit mus7	内存	24.69	7.19	132	411	0.92	320
			线程启动	36.87	0.94	77	205	0.88	320
		永久	内存	0.00	0.00	0	0	0.00	256
			线程启动	0.00	0.00	0	0	0.00	256
	yes	基本	内存	17.92	4.16	424	2590	0.93	240
			线程启动	85.00	64.16	296	1097	0.97	240
		lit mus7	内存	0.83	0.41	27	38	0.90	240
			线程启动	22.92	0.42	33	106	0.86	240
		永久	内存	0.00	0.00	0	0	0.00	128
			线程启动	0.00	0.00	0	0	0.00	128
X86	no	基本	内存	3.12	0.00	109	170	0.84	128
			线程启动	0.00	0.00	0	0	0.00	128
		lit mus7	内存	10.54	1.17	89	365	0.89	256
			线程启动	99.61	52.73	72	136	0.97	256
		永久	内存	0.00	0.00	0	0	0.00	128
			线程启动	0.00	0.00	0	0	0.00	128
	yes	基本	内存	82.03	1.56	123	274	0.88	128
			线程启动	39.84	0.00	61	107	0.83	128
		lit mus7	内存	2.34	0.00	21	25	0.83	256
			线程启动	100.00	71.28	50	72	0.99	256
		永久	内存	0.00	0.00	0	0	0.00	128
			线程启动	0.00	0.00	0	0	0.00	128
M1-CPU	no	基本	内存	41.56	20.93	1910	85814	0.95	320
			线程启动	83.43	67.81	8429	770700	0.98	320
		lit mus7	内存	68.75	38.44	14725	208033	0.94	320
			线程启动	95.31	77.50	5354	120350	0.99	320
		永久	内存	5.07	1.17	101	326	0.87	256
			线程启动	19.53	4.69	40	125	0.88	256
				GPUs					
AMD	no	GPU 并行	内存	13.8	5.9	7890	170300	0.89	708
NVIDIA	no	GPU 并行	内存	5.6	2.8	27768	499600	0.88	726
M3-GPU	no	GPU 并行	内存	52.4	20.5	92470	4921300	0.92	696

NVIDIA GeForce RTX 4070 与 AMD 类似，NVIDIA GPU 默认采用顺序内核执行模型。我们的结果表明，该设备也受到 DISORDER 的影响，但信号的频率和可靠性低于 AMD（尽管重排序观测值的增幅最终高于 AMD）。

NVIDIA 还提供 多进程服务(MPS) [35],，该服务专为内核并行度不足以充分利用整个 GPU 的情况而开发。MPS 允许多个内核在 GPU 上并行执行，但文档警告称，这种模式会降低安全性。我们使用此配置运行了一些先导实验，并观察到 DISORDER 指标显著升高，这为并行内核执行环境会使 GPU 更容易受到 DISORDER 影响提供了证据。

M3-GPU 与 M1-CPU 类似，关于这款处理器的文档很少。为弄清其执行模型，我们设计了一项小型微基准测试：让两个不同的进程各自执行一个小型内核，执行时间大致相同 k。我们同时启动这两个进程，并观察到二者全部完成所需的总时间也为 k。因此，我们可以得出结论：这款 GPU 采用并行内核执行模型。表 4底部的结果表明，这种并行内核执行所产生的信号比其他 GPU 设备上的信号更频繁、更可靠且更有效（至少高出一个数量级）。因此，如果 GPU 转向更并行的执行模型，DISORDER 漏洞可能会变得更具威胁。

4.3 虚拟化边界

迄今为止，我们的结果仅在现成的消费级设备上测试了跨进程信号。然而，测试其他系统是否容易受到 DISORDER攻击尤其是多租户的关键安全系统，这一点很重要。虽然我们已表明，广泛部署于云端的 X86 系统容易受到 DISORDER攻击，但 Arm 系统允许更多重排序，因此其攻击面可能更大。我们注意到，基于 Arm 的系统在云端日益普及，例如亚马逊的 Graviton 处理器 [7]通过 Nitro 虚拟机监控程序在多租户机器上运行 [8]。同样，Google Cloud 提供基于 Arm 的 Axiom 处理器 [17] ，并可能使用 KVM 作为虚拟机监控程序 [16]。

近年来，GPU 现在已具备一定的虚拟化支持。在这些方案中，物理 GPU 硬件会被分区，并为每个虚拟 GPU 分配一块 GPU 的物理切片。NVIDIA 提供 MIG [34], , AMD 提供 SR-IOV [9]。目前我们尚未测试这些系统，原因有二：它们需要高端的新款 GPU，而我们目前无法立即获取；此外，它们还要求重写部分测试代码。

5 实现 DISORDER 攻击

利用模糊测试活动收集的数据，我们可以找出能够暴露可靠且易于识别信号的 Litmus 测试、测试框架和压力测试组合。我们展示了如何以此为基础实现两种经典攻击：隐蔽信道和应用指纹识别。鉴于这些攻击需要大量手动调优并依赖可靠信号，我们仅在部分设备上实现了它们，即所有 CPU 和 Apple GPU。

安全边界在本节中，我们会在可能的情况下提高设备上的安全边界。对于苹果 CPU 和 GPU，我们展示了 DISORDER 如何跨越进程边界。对于 Arm 和 X86，我们在 KVM 虚拟化边界上实现这些攻击：在主机操作系统上启动监听进程，在客户机上启动压力进程。与模糊测试活动类似，我们运行了一些初步实验，以确定有效的线程固定方式。在 Arm 上，KVM 实例分配了三个虚拟 CPU，分别固定到主机核心 0、2 和 4；监听进程则在主机核心 1 和 3 上运行 Litmus 测试。在 X86 上，所有实验都为 KVM 实例分配十个虚拟 CPU，并将其固定到编号为偶数的主机核心。对于隐蔽信道实验，我们将监听器固定到主机核心 1 和 3；对于深度神经网络指纹识别实验，则将其固定到主机核心 1 和 11。

我们的初步实验表明，线程启动压力跨越 KVM 边界时可靠性会降低；鉴于操作系统在线程启动中的作用，这一点并不意外。不过，内存压力仍然可靠，因此，针对 Arm 和 X86 的攻击将使用内存压力。

5.1 DISORDER 隐蔽信道

DISORDER 可通过以下方式实现隐蔽信道：接收通信的进程运行 Litmus 测试，并记录观察到的重排序次数。随后进行时间序列分析，将信号解码为高(↑)、低(↓)和空格(∅)信号。要发送数据，进程会执行不同的压力模式，以编码 ↑和 ↓信号；暂停施加压力则编码 ∅。

为简化实现，我们聚焦于一个由一个发送方进程和一个接收器进程组成的单向信道。接收器进程使用测试配置（Litmus 测试和测试框架），而发送方使用两种压力配置来编码两个信号，这些配置根据模糊测试活动的结果进行选择。我们在表5中总结了这些配置。我们看到，MP 和 R 微基准测试最为敏感；除持续测试框架外，不同的测试框架和压力技术均得到了使用。需要注意的是，表中的 X86-arch 行对应于早期利用 DISORDER 底层系统细节的实验，其更详细的描述见第5.3节。

表5：DisOrder 隐蔽信道实现。对于每种设备，我们列出所用的测试框架和 Litmus 测试。对于每种信号，我们列出压力技术、重排序的平均次数（Avg.）和标准差（Std.）。∅ 没有压力技术，因为它仅表示没有压力。X86-arch（以绿色突出显示）是针对底层 X86 系统细节定制设计的，更多内容见第5.3节。它没有 ∅ 信号

设备	框架	Test	↑压力	↑平均值	↑标准差	↓压力	↓平均值	↓标准差	∅平均值	∅标准差
Arm	litmus7	MP	mem	5859.1	1469.9	mem	3224.2	709.9	437.5	141.0
M1	基本	R	TL	137.7	20.3	TL	37.3	7.7	3.4	2.3
X86	litmus7	R	mem	803.6	46.7	mem	648.9	34.0	241.0	62.1
M3-GPU	GPU 并行	MP	mem	12.4	5.1	mem	2.4	1.9	0.0	0.0
X86-arch	arch-aware	SB	arch-aware	48.3	2.9	arch-aware	1.1	2.5	N/A	N/A

为了解码信号，接收器进程必须对其接收到的信号进行分类，并在信号状态之间转换。

信号分类 我们发现，在可靠的压力配置下，重排序的数量近似服从正态分布，因此可以计算一组样本的均值和标准差。接收方维护一个测试样本窗口，以应对弱行为的自然变异性以及发送方与接收方之间缺乏同步的问题。

发送方通过运行 ↑、↓和 ∅ 信号来传输信号。接收方使用以信号的均值和标准差为参数的 t 检验，计算窗口中的每个样本来自给定信号分布的可能性。随后，根据 t 检验结果的 p 值降序排列，将窗口中的每个样本分类为其分布与该样本最匹配的信号。接收方根据窗口中多数样本的分类结果，将整个窗口分类为相应的信号。虽然这种方法在 CPU 上效果良好，但我们发现，在 M3-GPU 上，∅信号期间的内存重排序数量通常为 0，且不服从正态分布。因此，我们在这款 GPU 上使用截断启发式方法：如果重排序数量低于截断值，就将样本分类为 ∅ 信号。

状态转换 接收方通过状态机在不同信号之间进行转换。它从< 스타일 id='3'>待机 状态开始，表示正在等待接收↑/↓信号。一旦拥有足够的样本填满一个窗口，接收方就会对该窗口进行分类，并转换到↑或↓状态。接收方会保持在该状态，直到将某个窗口分类为 ∅ 信号；此时，它记录一个比特并返回< 스타일 id='9'>待机 状态。在 M3-GPU 上，我们观察到↑信号有时会被错误分类，因此加入了↓′状态，接收方会先转换到该状态，然后再转换到↑或↓状态。

实验与结果 我们通过从发送方向接收方发送随机比特串，测试隐蔽信道的准确率和速度。信道的准确率可用 Levenshtein 距离计算，即

表6：DisOrder 隐蔽信道指标。X86-arch 系统以突出显示，因为其设计方式不同，用于展示如果能够利用底层系统细节，DisOrder 的潜力（见第5.3节）

设备	窗口大小	准确率	比特/秒
X86	5	98%	0.32
Arm	5	94%	0.36
M1	5	95%	0.66
M3-GPU	3	95%	16.05
X86-arch	5	94%	29448.90

参考字符串与接收字符串之间单比特编辑（插入、删除或交换）的数量。将距离除以参考字符串的长度，即可得出百分比。

表。 6 总结了我们的结果，展示了我们在不同设备上的隐蔽信道中能够达到的最高平均比特每秒（比特每秒），同时保持 >90% 的准确率。结果通过发送随机的 13 字符 ASCII 字符串（104 比特）10 次获得，并计算各试验中的平均速度和准确率。

在 M3-GPU 上，我们实现了 95% 的平均准确率，速度为 16 比特每秒。相反，由于操作系统上其他进程产生的系统噪声，CPU 上的比特每秒显著较低；为提高可靠性，需要进行更多测试迭代并使用更大的窗口大小。这两个因素意味着，为了达到相近的准确率，传输速率比 GPU 信道慢两个数量级。我们观察到，M1-CPU 隐蔽信道的速度大约 2× 与 Arm 和 X86 一样快；这是因为 M1-CPU 能够利用线程启动压力，回顾第 4 节可知，这能产生可靠的高信号。由于 Arm 和 X86 运行在 KVM 边界之间，线程启动压力的可靠性较低，因此它们使用内存压力。最后，X86-arch（见第 5.3 节）使用专用压力，利用底层细节，实现了高出数个数量级的比特每秒。

关于先前研究，[12]实现了一种基于 X86 重排序的隐蔽信道，传输速率约为 1 比特每秒，同时使用计时器。其他无计时器方法的传输速率可达比特每秒级别

与我们的方法类似，[24,25,47]。最后，高度专用的无计时器方法可实现每秒千比特的传输速率 [51]；我们表明，DISORDER 在 X86-arch 上有望达到这一速度，详见第 5.3节。

5.2 DISORDER 指纹识别

对于应用指纹识别，我们无法依赖模糊测试结果中经过微调的压力信号。然而，正如第1.1节所示，我们发现某些应用类别具有可区分的信号，DISORDER 能够检测到这些信号。

深度神经网络架构 不同的深度神经网络可能在架构上存在很大差异，例如层操作、深度、内存用量和计算强度各不相同。这些架构差异会体现在特定平台或加速器上的底层实现中，从而形成不同的内存访问模式。这些差异类似于我们在模糊测试活动中寻找的不同压力信号，尤其是与内存压力相关的信号。因此，DISORDER 指纹识别攻击或许能够区分运行于不同进程中的不同深度神经网络架构。

在这些实验中，我们选择了 5 种常见的深度神经网络架构，如图 3所示。为了为攻击进程选择测试项，我们再次查阅模糊测试数据，寻找对内存压力具有稳定敏感性的配置。随后，攻击进程在受害者同时运行深度神经网络推理时收集测试样本。

我们的实验细节如下：令 A 表示我们希望分类的候选模型集合，令 s 表示我们测试的系统（例如 Arm、X86、M1-CPU 或 M3-GPU），并令t 表示经判定对s上的内存压力敏感的 Litmus 测试。对于每种深度神经网络架构 $d \in A$，我们收集在s 上执行t 时的 4K 个观测值，同时在另一个进程中运行d 。我们将前 2K 个观测值指定为训练集 S_d，后 2K 个指定为测试集 T_d。对于一种深度神经网络架构 d，我们随后从 T_d 中随机抽取z 个观测值，组成样本t。接着，我们使用独立样本 t 检验，将t 与$i \in A$ 中每个模型的训练集S_i 进行比较，并将t 分类为拟合效果最佳的深度神经网络所对应的类别。

我们在两种不同的样本量下进行这些实验：小样本量（30）和大样本量（100）。图 3 展示了每种样本量下运行 1000 次试验的结果，每次试验都从随机选取的测试集 T_d中进行抽样。随着样本量增加，分类准确率有所提高；在大多数设备上，样本量为 100 时准确率超过 80%。我们调整 Litmus 测试迭代次数，使得在每台设备上获取 30 个样本所需时间少于 5 秒。

MobileNetV3 small 和 AlexNet 架构在所有设备上都能轻松分类，而 VGG-16 在除 X86 CPU 之外的每台设备上，准确率都相对较低。事实上，图中所示的 VGG-16 和 ResNet-50 的分布_ 2 在 M3-GPU 上相对接近。我们发现，M1-CPU 的准确率最低，不过，先导实验表明，增加 Litmus 测试迭代次数（而非

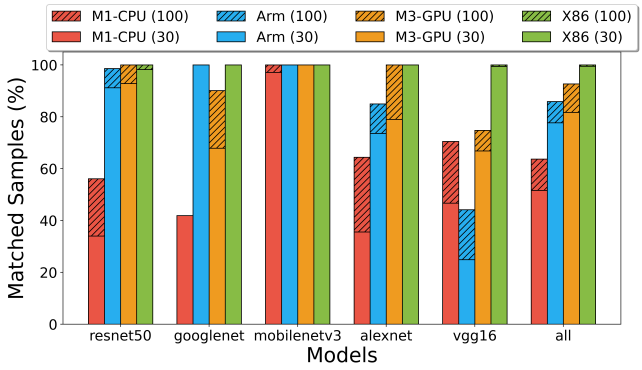

图 3：深度神经网络架构分类结果。该图展示了使用独立样本t检验，将内存重排序观测与样本量分别为30和100的训练集进行比较后，匹配样本所占的百分比。

样本数量）似乎能显著提高其准确率。因此，这种攻击需要在多个维度上进行调优，但在这些设备上都可以获得较高的准确率。此外，采用更复杂的分类技术，例如 [14]，有助于进一步提高使用 DISORDER 进行 DNN 分类的准确率。

启动应用程序 鉴于我们的模糊测试结果表明，线程启动压力会产生显著影响，我们现在探究 DISORDER 是否可用于识别其他类型的系统行为。我们选择 M1-CPU 设备，因为它对这种压力类型尤为敏感。

我们选择了一个在概念上可能与线程启动压力相似的应用程序，其中线程会反复启动并加入。为此，我们选择了 Google Chrome，因为它采用了大量多线程。我们探究 DISORDER 是否可用于识别何时打开或关闭了 Google Chrome 之类的应用程序。为此，我们运行一个脚本，反复打开和关闭 Chrome，每次启动后等待 3 秒再关闭，关闭后等待 2 秒再重新启动。

攻击者在基本测试框架中使用 Read litmus测试，因为该测试恰好最为敏感，并持续以每 1k 次迭代为间隔，对观测到的重排序数量进行采样。图 4 展示了该实验的结果，其中 x 轴表示时间戳，y 轴表示观测到的重排序数量。我们可以清楚地看到不同模式：启动 Google Chrome 时（较长且较高、以绿色标示的重排序区域）、关闭时（较短、以红色标示的重排序区域），以及系统空闲时（较低、以黑色标示的重排序）。这表明 DISORDER 有望用于识别其他应用程序行为，尤其是消费级设备上的交互式应用程序相关行为。

我们的先导实验表明，许多类型的系统行为都会增加在 M

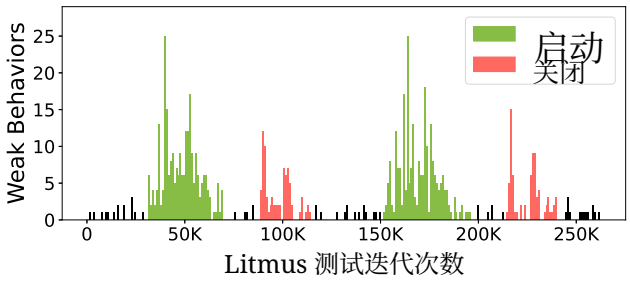

图 4：在启动和关闭 Google Chrome 期间，M1-CPU 上内存重排序的时间序列数据。

系列苹果设备上观察到的重排序数量。例如，即使是在终端中切换标签页或启动应用程序，也会显著增加观察到的重排序数量。由于这些重排序的底层成因尚未确定，因此很难充分认识它们的潜力，也难以提出严谨的缓解方法。有鉴于此，我们认为，未来的研究将能够提升 DISORDER 的指纹识别精度和影响力。

5.3 定向架构攻击

到目前为止，我们的攻击不需要任何底层架构知识，而是利用实证模糊测试结果。现在，我们展示 DISORDER 攻击利用底层硬件细节的潜力方面的初步结果。我们在 X86 处理器上展示这些结果，因为该平台最便于开展实验，而且 X86 通常有完善的文档记录。我们的实验以隐蔽信道为背景，我们设计了一个名为*arch-aware*的自定义发送方和接收方框架。我们将目标设为该设备的性能核心，每个核心都可以执行两个硬件线程（通过超线程实现），并且拥有自己的 48 KB、12 路组相联一级缓存，缓存行大小为 64 字节。

我们的假设是，X86 存储缓冲区（负责重排序）会受到 Litmus 测试中一级缓存是否包含位置 x 或 y 的影响。接收方（执行 Litmus 测试）被映射到两个不同的核心。我们发现，从一级缓存中逐出其中一个位置（例如 x），同时确保另一个位置（例如 y）仍留在一级缓存中，会产生极强且可靠的信号（即在超过 90% 的迭代中观察到了重排序）。

为在此框架中实现隐蔽信道，我们在另一个线程中模拟发送方，该线程与接收方中的线程 0 映射到同一个核心。为发送信号，发送方只需写入 12 个位置（一个缓存组的大小），这些位置映射到与内存位置 x相同的缓存组，从而将 x从线程 0 的一级缓存中逐出。表 5 显示，这些信号非常强且十分可靠。

通过扩展这一方法，我们能够监控多个缓存组：我们的 X86 具有 64 个一级缓存组，这使我们能够实例化 63 个 Litmus 测试用例。 x 位置映射到不同的缓存组，而 y 位置全部映射到第 64 个缓存组，因为该缓存组不应被驱逐。这些信号非常清晰，因此测试只需执行 15 次迭代。此外，鉴于每次会发送多个比特，因此无需使用0 信号，因为第一个比特可以充当虚拟时钟，在信号之间调制高低电平。这种高度调优的配置能够达到近 30k 比特每秒，比其他方法高出几个数量级（参见表中的 X86-arch。5 和6）。

迈向完整攻击 我们在 X86-arch 上的结果尚属初步，旨在展示利用底层系统细节时 DisOrder的潜力。我们未能将 arch-aware 完整实现为一种完整攻击，因为重排序似乎不仅受缓存组影响。我们发现，在尝试更全面地将内存划分到各个进程时，信号会以难以预测的方式衰减；反过来，在其他一些配置中也会出现强而可靠的信号，而这些配置似乎与我们所知的任何系统文档都不相符。因此，我们认为有必要开展更详细的调查；一旦对该攻击有了更深入的了解，就很可能能够完整实现高带宽隐蔽信道，以及更精细的数据提取攻击，例如密码密钥提取 [51]。

6 DISORDER 缓解措施与补救措施

与其他侧信道攻击（例如 Prime-and-Probe）类似，缓解 DISORDER 的难度很大，而且会对系统造成较大影响。此外，除非供应商发布关于内存重排序的精确说明（这不太可能），否则缓解措施只能基于推测或经验。对于现有系统，我们认为有两种缓解途径：

禁止内存重排序程序分析技术通过自动插入栅栏指令来禁止内存重排序。若要通过这种方法缓解 DISORDER，系统必须强制所有不可信程序以消除重排序的方式进行编译，并禁止使用绕过编译器分析的编程特性，例如内联汇编。

一种简单的方法是在每条内存指令后插入一个栅栏，以消除所有重排序；然而，这会禁止许多硬件优化，因此性能开销很高（如 [5,40] 所报告）。其他方法会进行更复杂的分析，以识别可能在线程间共享的内存访问，从而减少栅栏的数量。据报告，这些方法的开销要小得多，平均约为 1.5× [27]。有些语言遵循一种称为 无数据竞争程序的顺序一致性 （简称 SC-DRF）的范式 [1]。采用 SC-DRF 的语言保证：如果程序遵守某些规则（即不存在数据竞争），那么内存重排序就不会

被观察到。如果系统严格强制执行 SC-DRF 程序（例如安全 Rust），那么内存重排序（因此也包括 DISORDER）就不可能被观察到。

信号混淆一种常见的侧信道缓解措施是混淆遭到破坏的执行特征。例如，如果受害者确保内存访问与输入无关，就可以缓解基于内存的攻击 [36]。其他方法可能会以某种方式插入随机内存访问，使泄露的信息不再有用。先前研究提出了多种自动化方法来实现这一点，包括编译器技术 [44], 和透明内存管理技术 [52]。

我们展示了通过 DISORDER 泄露信息的两种特征。第一种是内存访问模式，这类问题很可能可以像其他内存侧信道一样得到缓解。第二种是线程启动行为，这是 DISORDER 中一种新颖的特征。应用程序（例如 Google Chrome）需要修改其线程管理逻辑，例如，使其与其他应用程序保持一致。鉴于大型软件的复杂性以及并行编程的复杂性，这类缓解措施将难以实施，且会带来较大侵入性。我们的初步实验还发现，DISORDER 似乎会受到各种其他系统行为的影响（例如，快速切换终端标签页）。要实现稳健的缓解措施，就需要更全面地了解这些重排序的成因。

在线检测指纹识别攻击只需两个核心来执行 Litmus 测试，并需要 <2MB 内存。后台运行 DISORDER 的开销因系统和应用而异，但我们发现，对于深度神经网络推理，DISORDER 监听器在 Apple CPU 上仅造成 25% 的性能下降。对于并行度更高的系统，这种开销会降低，例如在 M3-GPU 上，DISORDER 监听器的开销可以忽略不计。因此，资源监控程序不太可能将 DISORDER 攻击判定为异常。

7 相关工作

基于内存的侧信道近期许多研究利用各种不同的内存机制构建侧信道。例如， [20] 通过测量 Linux 系统中存储设备上的争用来实现侧信道。他们展示了如何利用这种方法构建隐蔽信道，并进行粗粒度的应用指纹识别。其他研究分析了不同 GPU 的隐式（且未公开的）内存压缩技术 [46]；压缩数据产生的内存流量可用于重建图像。最后，研究表明，X86 预取指令会泄露缓存一致性状态 [18], ，利用这些状态可以比先前研究更高效地实施瞬态执行攻击。

DISORDER 有两大显著特点：（1）DISORDER 的能力要求较低，无需计时器、硬件监控器，也无需针对低精度配置进行详细的逆向工程；（2）DISORDER 存在于多种设备上，而先前研究通常针对特定设备。

内存模型测试与建模一些先前研究探讨了重排序的成因和特征。例如， [31]通过实证测试确定 X86 存储缓冲区的大小；其他研究则展示了内存库冲突如何导致 GPU 上的重排序 [3]。这些底层细节可以大幅提高 DisOrder攻击的精度（如第 5.3节所示）。然而，它们似乎只存在于（较旧）架构的某些行为中。

其他研究对导致重排序的微体系结构特征（例如缓存协议）进行了建模 [28]；利用这些模型，他们能够验证其是否符合更高层级的规范。类似地，另一项研究使用 happens-before 关系模型来捕获多种微体系结构侧信道 [43]。尽管这一层级的建模能够深入了解内存重排序的成因，但它需要大量底层知识，其中许多属于专有信息，因此主流商用系统尚未以这种方式进行建模。

深度神经网络架构指纹识别其他研究也使用侧信道来提取或预测深度神经网络架构，这与 DISORDER 在第 5.2节中的用法类似。例如， [50] 使用 Prime+Probe 和 Flush+Reload 缓存攻击，来获知通过高性能 GEMM库计算的矩阵大小和数量。其他研究则将功耗用作侧信道，以实施模型提取攻 [49]击。 [14],展示了更复杂的、基于功耗的模型提取攻击，该攻击通过分析随时间变化的能量轨迹来恢复网络结构。这些研究高度针对特定设备，需要精确的计时器或底层功耗监控 API；而 DISORDER 所需能力较低，并且已证明适用于许多不同设备。

8 结论

我们提出了一种名为 MEMORY DISORDER 的新型无计时器侧信道攻击，它利用了内存重排序。我们展示了这种攻击会影响大多数主流处理器，包括 Arm、X86 和 Apple CPU，以及 NVIDIA、AMD 和 Apple GPU。我们展示了 DISORDER 用于经典攻击的潜力，例如隐蔽信道和应用指纹识别。如果未来的工作能够更精确地确定内存重排序的成因（可能聚焦于单一处理器），便很可能能够实现更有针对性的攻击，并为制定稳健的缓解技术提供参考。

9 负责任披露

我们向以下公司的安全团队提供了这项工作的预印本和摘要：苹果、亚马逊、AMD、Arm、谷歌、Intel、微软、NVIDIA、高通和三星。所有公司均确认了这一发现，并批准了发布时间安排。

参考文献

[1] Sarita V. Adve 和 Mark D. Hill。弱序——一种新的定义。载于《第*17*届年度国际计算机体系结构研讨会论文集》。ACM，1990年。

[2]Jaeguk Ahn、Jiho Kim、Hans Kasan、Leila Delshadtehrani、Wonjun Song、Ajay Joshi 和 John Kim。基于片上网络微体系结构的 GPU 隐蔽信道。载于 *MICRO-54*：第*54*届年度*IEEE/ACM*国际微体系结构研讨会。ACM，2021年。

[3] Jade Alglave、Mark Batty、Alastair F. Donaldson、Ganesh Gopalakrishnan、Jeroen Ketema、Daniel Poetzl、Tyler Sorensen 和 John Wickerson。GPU并发：弱行为与编程假设。载于编程语言与操作系统体系结构支持（*ASPLOS*）。ACM，2015年。

[4] Jade Alglave、Anthony Fox、Samin Ishtiaq、Mag- nus O. Myreen、Susmit Sarkar、Peter Sewell 和 Francesco Zappa Nardelli。Power 和 Arm 多处理器机器码的语义。载于 第*4*届多核编程声明式方面研讨会论文集。ACM，2009年。

[5] Jade Alglave、Daniel Kroening、Vincent Nimal 和 Daniel Poetzl。《别坐在栅栏上：一种自动插入栅栏的静态分析方法》。《*ACMTrans.Program.Lang. Syst.*》，2017年。

[6] Jade Alglave、Luc Maranget、SusmitSarkar 和 Peter Sewell。《Litmus：针对硬件运行测试》。载于系统构造与分析工具和算法。施普林格·柏林·海德堡，2011年。

[7] 亚马逊。面向独立软件供应商的 AWS Graviton2。
https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-graviton2-for-isv/aws-graviton2-for-isv.pdf，2024年。

[8] 亚马逊。AWS Nitro系统的安全设计。
https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf，2024年。

[9] AMD。一致性与安全：AMD 的 GPU 虚拟化方法。
https://www.amd.com/system/files/documents/gpu-consistency-security-whitepaper.pdf，2017年。

[10] Arvind Arvind 和 Jan-Willem Maessen。《内存模型 = 指令重排序 + 存储原子性》。载于第*33*届年度国际计算机体系结构研讨会论文集。IEEE计算机学会，2006年。

[11] Mark Batty、Scott Owens、Susmit Sarkar、Peter Sewell 和 Tjark Weber。《C 并发的数学化》。载于编程语言原理（*POPL*）。ACM，2011年。

[12] Sophia d'Antoine、Jeremy Blackthorne 和 Bülent Yener。《乱序执行作为跨虚拟机侧信道及其他应用》。载于第 *1* 届逆向与进攻导向趋势研讨会论文集。美国计算机协会，2017年。

[13] Shaked Flur、Kathryn E. Gray、Christopher Pulte、Sus- mit Sarkar、Ali Sezgin、Luc Maranget、Will Deacon 和 Peter Sewell。《ARMv8 架构的操作性建模：并发与指令集架构》。载于第 *43* 届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集。ACM，2016年。

[14] Y. Gao、H. Qiu、Z. Zhang、B. Wang、H. Ma、A. Abuadbba、M. Xue、A. Fu 和 S. Nepal。《Deeptheft：通过功耗侧信道窃取深度神经网络模型架构》。载于*2024*年 *IEEE* 安全与隐私研讨会（*SP*）。IEEE 计算机学会，2024年。

[15] Lukas Giner、Roland Czerny、Christoph Gruber、Fabian Rauscher、Andreas Kogler、Daniel De Almeida Braga 和 Daniel Gruss。《来自浏览器的通用自动化随访式 GPU 缓存攻击》。载于第 *19* 届 *ACM* 亚洲计算机与通信安全会议论文集。ACM，2024 年。

[16] 谷歌。在 Google Cloud 中强化 KVM 虚拟机监控程序安全性的 7 种方法：明文安全。
https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext，2017年。

[17] 谷歌。介绍 Google Axion 处理器：我们的新款基于 ARM 的 CPU。
https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu，2024年。

[18] Y. Guo、A. Zigerelli、Y. Zhang 和 J. Yang。对抗性预取：新型跨核心缓存侧信道攻击。载于安全与隐私（*SP*）。IEEE 计算机学会，2022年。

[19] 苹果公司 Metal 着色语言规范。 https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf。访问日期：2024年-06-05。

[20] Qisheng Jiang 和 Chundong Wang。Sync+sync：一种基于 fsync 和存储构建的隐蔽信道。发表于第*33*届 *USENIX* 安全研讨会（*USENIX Security 24*）。USENIX 协会，2024年。

[21] Jake Kirkham、Tyler Sorensen、Esin Tureci 和 Mar- garet Martonosi。经验性内存一致性测试基础。*ACM* 编程语言学报，2020年。

[22] Reese Levine、Mingun Cho、Devon McKee、Andrew Quinn 和 Tyler Sorensen。GPUHarbor：大规模测试 GPU 内存一致性（经验论文）。发表于第*32*届 *ACM SIGSOFT* 软件测试与分析国际研讨会论文集。ACM，2023年。[23] Reese Levine、Tianhao Guo、Mingun Cho、Alan Baker、Raph Levien、David Neto、Andrew Quinn 和 Tyler Sorensen。MC 变异体：评估和改进内存一致性规范测试。发表于第*28*届 *ACM* 国际编程语言与操作系统架构支持大会论文集，第*2*卷。ACM，2023年。[24] Moritz Lipp、Daniel Gruss 和 Michael Schwarz。通过功耗和时间实施 AMD 预取攻击。发表于*USENIX* 安全研讨会，2022年。[25] Moritz Lipp、Andreas Kogler、David Oswald、Michael Schwarz、Catherine Easdon、Claudio Canella 和 Daniel Gruss。PLATYPUS：基于软件的 X86 功耗侧信道攻击。发表于*2021*年 *IEEE* 安全与隐私研讨会（*SP*），2021年。[26] Fangfei Liu、Yuval Yarom、Qian Ge、Gernot Heiser 和 Ruby B. Lee。末级缓存侧信道攻击切实可行。发表于*2015*年 *IEEE* 安全与隐私研讨会。IEEE，2015年。[27] Lun Liu、Todd Millstein 和 Madanlal Musuvathi。通过推测编译加速 Java 的顺序一致性。发表于第*40*届 *ACM SIGPLAN* 编程语言设计与实现大会论文集。ACM，2019年。

[28] Yatin A. Manerkar、Daniel Lustig、Michael Pellauer 和 Margaret Martonosi。《 CCICheck：使用 UHB 图验证一致性 - 连贯性接口》。发表于 *2015*年 第*48*届 *IEEE/ACM* 国际微体系结构研讨会（*MICRO*）。ACM，2015年。

[29] Henry B Mann 和 Donald R Whitney。关于检验两个随机变量中的一个在随机意义上是否大于另一个。数学统计年刊，1947年。

[30] Themis Melissaris、Markos Markakis、Kelly Shaw 和 Margaret Martonosi。《PerpLE：提升内存一致性测试的速度和有效性》。载于*2020*年 *IEEE/ACM* 第*53*届国际微体系结构年会（*MICRO*）。IEEE，2020年。

[31] Adam Morrison 和 Yehuda Afek。《有时限的 TSO：用于无栅栏非对称同步》。载于《第二十届编程语言与操作系统体系结构支持国际会议论文集》。ACM，2015年。

[32] Vijay Nagarajan、Daniel J. Sorin、Mark D. Hill 和 David A. Wood。《内存一致性与缓存一致性导论》。施普林格国际出版公司，2020年。[33] Hoda Naghibijouybari、Khaled N. Khasawneh 和 Nael Abu-Ghazaleh。《在通用图形处理器上构建并表征隐蔽信道》。载于《IEEE/ACM 第50届国际微体系结构年会论文集》。ACM，2017年。[34] NVIDIA。《多实例GPU 用户指南（用户指南）》。https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_用户_指南.pdf，2024年3月。[35] NVIDIA。《多进程服务（r550版）》。https: // docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf，2024年6月。[36] Dag Arne Osvik、Adi Shamir 和 Eran Tromer。《缓存攻击与对策：AES 案例》。载于 David Pointcheval 编，《密码学专题——CT-RSA 2006》。施普林格·柏林·海德堡，2006年。[37] Scott Owens、Susmit Sarkar 和 Peter Sewell。《更好的 x86 内存模型：x86-TSO》。载于《第22届高阶逻辑定理证明国际会议论文集》。施普林格出版社，2009年。[38] Michael Schwarz、Clémentine Maurice、Daniel Gruss 和 Stefan Mangard。《奇妙的计时器及其所在之处：JavaScript 中的高分辨率微体系结构攻击》。载于《金融密码学与数据安全：第21届国际会议，FC 2017，斯利马，马耳他，2017年4月3日至7日，修订精选论文》。施普林格出版社，2023年。

[39]Peter Sewell、Susmit Sarkar、Scott Owens、Francesco Zappa Nardelli 和 Magnus O. Myreen。x86-TSO：适用于x86多处理器的严谨且实用的程序员模型。*Commun.ACM*，2010年。

[40] Tyler Sorensen 和 Alastair F. Donaldson。揭示 GPU 应用程序中与弱内存相关的错误。载于第 *37* 届 *ACM SIGPLAN* 编程语言设计与实现会议。ACM，2016年。

[41] Tyler Sorensen 和 Heidy Khlaaf。LeftoverLocals：通过泄露的 GPU 本地内存监听大语言模型响应，2024年。

[42] Sanya Srivastava。异构 CPU-GPU 系统内存模型测试。硕士论文，加利福尼亚大学圣克鲁兹分校，2024年。
https://www.proquest.com/dissertations-theses/testing-memory-models-heterogeneous-cpu-gpu/docview/3082136707/se-2。

[43]Caroline Trippel、Daniel Lustig 和 Margaret Martonosi。CheckMate：硬件漏洞利用与安全性测试用例的自动化合成。发表于*2018*年 第*51*届 *IEEE/ACM* 国际微体系结构研讨会（*MICRO*）。IEEE，2018年。

[44] Jeroen Van Cleemput、Bjorn De Sutter 和 Koen De Bosschere。用于缓解定时侧信道攻击的自适应编译器策略。*IEEE* 可靠与安全计算汇刊，2020年。

[45] W3C。WebGPU 规范。https://www.w3.org/TR/webgpu/，2023年。访问日期：2024年7月17日。

[46] Y. Wang、R. Paccagnella、Z. Gang、W. Vasquez、D. Kohlbrenner、H. Shacham 和 C. Fletcher。《Gpu.zip：基于硬件的图形数据压缩的侧信道影响》。载于安全与隐私（*SP*）。IEEE 计算机学会，2024年。[47] Yingchen Wang、Riccardo Paccagnella、Elizabeth Tang He、Hovav Shacham、Christopher W. Fletcher 和 David Kohlbrenner。《Hertzbleed：将功耗侧信道攻击转化为针对 x86 的远程计时攻击》。载于第*31*届 *USENIX* 安全研讨会（*USENIX Security 22*）。USENIX 协会，2022年。

[48] William W. Collier。ARCHTEST。
http://www.mpdiag.com/archtest.html，1994年1月。

[49] Yun Xiang、Zhuangzhi Chen、Zuohui Chen、Zebin Fang、Haiyang Hao、Jinyin Chen、Yi Liu、Zhefu Wu、Qi Xuan 和 Xiaoniu Yang。通过 Power 侧信道攻击打开 DNN 黑盒。*IEEE* 电路与系统汇刊 *II*：简报，2020年。[50] Mengjia Yan、Christopher W. Fletcher 和 Josep Torrellas。缓存心灵感应：利用共享资源攻击了解深度神经网络架构。载于第*29*届 *USENIX* 安全研讨会（*USENIX Security 20*）。USENIX 协会，2020年。

[51] Jiyong Yu、Aishani Dutta、Trent Jaeger、David Kohlbrenner 和 Christopher W. Fletcher。同步存储通道（S2C）：通过硬件同步指令，对 Apple M1 发起无计时器缓存侧信道攻击。载于第*32*届 *USENIX* 安全研讨会（*USENIX Security 23*）。USENIX 协会，2023年。

[52] Ziqiao Zhou、Michael K. Reiter 和 Yinqian Zhang。《一种防御末级缓存侧信道的软件方法》。载于*2016*年 *ACM SIGSAC* 计算机与通信安全会议论文集。美国计算机协会，2016年。