# 支持微架构防御的安全集成

Kartik Ramkrishnan、Stephen McCamant、Antonia Zhai、
Pen-Chung Yew 计算机科学与工程系 明尼苏达大学，美国明尼苏达州明尼阿波利斯 {ramkr004, mccamant, zhai, yew}@umn.edu

arXiv:2601.05057v1 [cs.CR] 8 Jan 2026

摘要——**微架构层面的攻击层出不穷，因此提出了许多对策。这带来了一个意料之外且尚未解决的安全问题：对这些防御措施进行简单集成，可能会导致安全漏洞。当一种防御改变了对另一种防御的安全性至关重要的微架构特性时，就会出现这种情况。我们将此问题称为**微架构防御假设违背**（MDAV）。**

**我们提出了一种两步方法，用于在集成早期筛查潜在的MDAV。第一步是设计并集成一个组合模型，并以安全属性的有界模型检测为指导。第二步是在模拟器上具体实现该模型，并通过模拟攻击进行评估。**

**作为支持第一步的一项贡献，我们提出了一种基于事件的建模框架，称为*Maestro*，用于测试和评估集成了防御措施的微架构模型。在评估中，Maestro 揭示了 MDAV（8），支持紧凑表达（≈ 15x Alloy 代码行数比），实现语义可组合性，并消除了性能下降（>100x）。**

**作为支持第二步的一项贡献，我们使用基于事件的模拟器（GEM5）来研究集成的微架构防御措施。我们表明，对某些先进防御机制进行简单集成后，其实现可能遭受隐蔽信道攻击；而采用我们的集成方法修复后的实现则能够抵御该攻击。**

## I. 引言

微架构攻击的种类繁多且不断增加，尤其是时序攻击 [33], [34], [49], [54], [62]–[77], [100], [104], [106], [109]（见 §IX）。在这类攻击中，攻击者可以通过观察受害者程序在处理器核心状态 [21]–[43], [48], [94], [95] 和/或缓存状态 [33], [52], [58], [74]–[76]上产生的微架构副作用，窥探其运行情况。攻击者还可以利用这些副作用建立隐蔽信道，并绕过针对这类非法通信的体系结构保护，传输秘密信息。为应对这些微架构攻击，人们已提出大量防御措施 [9], [29], [40], [45], [78], [97], [101] （见 §IX）。

防御措施通常旨在应对特定攻击类别，例如推测执行攻击 [43], 基于缓存命中的攻击 [100] 或基于缓存未命中的攻击 [34]。这些防御措施会在微架构层面修改处理器核心和/或缓存设计，以缓解隐蔽信道和侧信道。

在设计初期，检查安全属性的两种重要方法是模型检测 [1], [6]和 攻击模拟 [8], [9], [51], [97]。模型检测能够对抽象设计进行穷尽式检查，并在几秒或几分钟内发现安全漏洞 [37], [99]。它是发现有限规模（例如数千位）内属性问题的实用工具，通常适用于以抽象方式表达、仅包含最重要特性的设计 [99]。微架构仿真能够让我们在几分钟内对更贴近实际的设计进行特定攻击测试，但这会退化为非穷尽式测试 [9], [97]。由于这些技术的优势各不重叠，因此适合结合使用，以筛除不安全的设计，从而节省后期检查成本 [86]。

然而，尽管早期安全设计取得了这些进展，当前实践仍存在一个重要缺陷：各项防御措施都是彼此隔离地设计的，没有考虑它们对针对其他攻击类别的防御措施所产生的影响。

实际上，攻击者可以选择实施哪种攻击以及何时发起攻击。因此，我们需要将所有这些防御措施整合到微架构中。如果设计单项防御措施时只考虑其针对的特定攻击类别，那么设计中所采用的假设和规范很可能彼此冲突。可以预见，这可能会导致我们所称的“微架构防御假设违背(MDAV)”并产生可被攻击者利用的新型意外漏洞。因此，本文的核心研究问题如下。

针对不同攻击类别的单项防御措施集成到微架构中时，会不会引发意想不到的安全问题？我们又该如何检查和识别这种可能性？

我们提出了一种直观的两步方法来回答上述研究问题。

第一步是创建一个多重防御模型，使设计者能够检查针对 MDAV 的集成防御。如果模型存在 MDAV 问题，设计者可以提出模型修复方案。随后，可以重新评估修复后的模型。模型检查器可以穷举检查系统的一小部分，但模型检查器无法评估该部分与系统其余部分的配合情况。因此，在第二步中，设计者在微架构模拟器上实现设计，评估一种或多种针对 MDAV 的攻击，并检查修复后模型的实现是否能够抵御这些攻击。

在第一步中，设计者构建代表各项独立防御措施的模型，然后将这些防御措施组合起来，构建多重防御模型。第一个基础设施挑战是，使用简单补丁集成这些模型（以支持多重防御属性）可能导致合并错误和构建错误。通过语义组合消除集成工作流程中的这些错误，有助于将注意力集中在底层 MDAV 上，尤其是在设计者希望评估多种可能组合时。

另一项关键的基础设施挑战是，事件之间的多周期延迟会导致模型检测的成本激增。这是因为当前的模型检验框架 [99] 不支持步骤之间具有多周期延迟的事件触发。相反，它们通过逐周期状态变化 [37], [99]来表达时序。因此，即使只有少数事件，表达大量周期也会使运行时间激增。这使得表示和组合主要防御措施变得不切实际（见 §VIII ）。

为应对这些挑战，我们提出一个涵盖安全性的术语：***Maestro***。它指一种建模框架，原生支持步骤之间的**多周期延迟**，从而消除成本激增。它也指一种可组合变换策略，用于支持**语义组合**，避免组合过程中出现合并错误和构建错误。

我们通过设计用于模型规范（见 §III-B）和语义组合（见 §IV-C）的领域特定语言（DSL），实现了 Maestro。我们使用 Alloy [91], ——一种基于关系逻辑和线性时序逻辑的标准建模工具——作为后端模型检验器。Alloy 本身使用 SAT 求解器或 SMT 求解器后端，在模型空间中搜索断言违例（在我们的语境中，断言是安全属性）。借助 Maestro 的语义组合，该工具揭示了八项重要的 MDAV（见 §VIII-A）；我们还展示了在某些情况下，正确集成可以避免 MDAV。

尽管 Maestro 无法在模型检测步骤中对所有特性进行建模（例如，系统软件对硬件进行的、以安全性为导向的更新），但 Maestro 的结果仍可为更具体的评估提供指导。作为第二步，我们的方法会在模拟器 GEM5 上模拟更具体的防御版本。模型检查器的结果有助于第二步中的实现：既可以增强对安全设计的信心，也可以提供反例，作为实现攻击的依据。让这两个步骤协同工作，使相对简单、抽象的模型也能发挥作用。

在某个实现中，设计者面临的一项关键挑战，是构造最坏情况攻击场景，以考验已实现的防御措施。我们的方法使设计者能够从第一步得到的反例中获得启发。他们可以在特定实现中放大反例所体现的依赖秘密的时序差异。例如，如果反例表明时间差取决于某个特定推测窗口的持续时间，那么挑战就在于通过一串真实指令（例如 `x86_64`）将其放大。在 §VI 中，我们展示了这样一个场景：防御实现能够抵御经放大的时序——攻击。在 §IX 中，我们指出了 MDAV 重新引入的两类重要侧信道。我们还举例说明，考虑如何在模拟器上构建防御措施，可以揭示该防御措施不切实际的原因。

总体而言，我们的工作有三项主要贡献。

- 我们指出，微架构防御的假设违反（MDAV）这一概念是从事安全微架构研究的系统设计人员需要关注的重要问题。我们提出了一种两步方法，用于筛查 MDAV。
- 对于方法的第一步，我们提出了一个系统化建模框架 *Maestro*，用于研究 MDAV。Maestro 支持基于周期和基于事件的建模，兼顾细节与抽象。在我们的评估中，Maestro 可无缝实现语义组合，实现 $\approx$ 15 倍的 Alloy 代码行数（LoC）比，发现八个 MDAV，并消除 100 倍的性能下降。
- 作为我们方法的第二步，我们在GEM5模拟器上对集成防御开展案例研究，以测试一种已解析其MDAV的安全微架构能否抵御原始攻击，以及由时序放大指令序列促成的新型攻击。

据我们所知，本研究是对多种微架构防御集成开展的首项系统性研究，从而使我们能够识别MDAV。本文其余部分安排如下。

§II介绍与我们的集成研究相关的各类侧信道攻击和隐蔽信道攻击背景。作为两步筛查方法的第一步，§III介绍通用建模框架Maestro，用于评估集成防御的早期事件驱动硬件模型。§IV讨论如何利用Maestro实现防御集成的工作流程。§V和§VIII通过实例讨论如何运用该工作流程进行集成。

§VI介绍以*S*开头的*MESI* (SS-MESI)和远端未命中时延迟推测性更改 (DSRM)防御在GEM5中的实现；这些防御基于集成工作流程中的固定模型。§VI和§VII在这些防御上实现了一种新提出的隐蔽信道攻击，并展示了它们的抵御能力。§VIII评估Maestro。§IX讨论相关工作及其他问题。§X总结全文。

## II. 背景

本节将介绍与我们所研究的防御集成问题相关的主要攻击和防御措施的背景信息。Spectre 隐蔽信道利用缓存 [43], [70] 及微架构 [43], [44] 的其他部分中的推测执行副作用 [43], [44] 。主要的侧信道攻击包括基于命中的缓存侧信道 [77], [100] 和基于驱逐的缓存侧信道 [34], [40]。

**TORC 防御。** 一类重要的防御策略是 TORC（远程缓存行时序混淆）策略 [63], [71], [98],，它会混淆缓存命中时间

使其看起来与缓存未命中时间相同。这可以消除基于缓存命中的攻击。

**DSRC 防御**。另一类重要的防御措施是推测延迟防御 [9], [97]。这类防御措施会消除微体系结构状态（包括缓存状态）中的推测性状态变化。一种基于延迟的重要防御策略可保护一致性状态免遭推测性泄漏，这就是 DSRC（延迟远程缓存行的推测性更改）策略 [9]。在 DSRC 方法中，任何会在缓存命中时更改易受攻击的一致性状态的加载（例如，将 E/M 状态更改为 S 状态）都不允许执行此更改。相反，系统会向处理器核心发送拒绝信号。处理器核心会确认该加载指令是否位于正确的执行路径上，然后再将其重新发送至缓存。随后发生缓存命中，加载的其余部分则以与基线不安全处理器相似的方式执行。

**多重防御的集成**。我们可能希望同时启用多种防御措施，例如带去分类的核心内推测延迟（SIDD） [8], [19]、未命中时延迟 （DoM） [9]、[80]、隔离 [72], [98]、带数据无关性的核心内推测延迟（SIDO） [102] 以及安全 *DRAM* 刷新 （SDR） [14]。SIDD 会根据非推测性访问是否已泄漏推测性访问的数据来对这些数据进行去分类。SIDO 允许推测执行，同时确保操作不受推测性访问的数据影响。隔离 将缓存划分为彼此独立运行的分区 [42]。一致性隔离 （CI）维护各分区之间的数据一致性 [72]。SDR 可缓解 Rowhammer [61] 攻击，这类攻击可能导致 DRAM 位翻转，从而造成未经授权的数据更改。

### III. 建模框架与方法

我们两步方法的第一部分，主要目标是构建一个严谨的建模框架。本节首先在 §III-A 中介绍用于研究防御集成的高层框架，然后在 §III-B 中提出一种工作流实现方案，即使用 Alloy Analyzer [91]（一种关系与时序建模工具）进行基于事件的建模。

### *A. Maestro* 建模框架

Maestro 是一个用于微体系结构级安全领域的抽象事件驱动建模框架。它将事件视为逐步执行的一部分，这些事件能够触发状态变化和更新。

Maestro 便于在早期研究防御集成，而无需完整描述控制/数据通路和控制/数据逻辑。这通常正是防御设计的详细程度 [8], [9], [78]。Maestro 通过允许防御模型向基线模型添加事件或修改其中的事件（见 §IV-A1），实现防御措施的集成。

**机器状态**。 机器状态是对机器中存储位的高层抽象。它抽象地表示处理器的寄存器、内存、缓冲区或其他存储组件中存储的数据。我们将包含机器状态的硬件表示为 $\mathcal{H}$。在 $\mathcal{H}$ 中，有 M 个位$[B_1, B_2 ... B_M]$。每个 $B_1 ..... B_M$的值为 0 或 1。

**逐步执行与时间**。 一次执行包含多个步骤，每个步骤中都可能发生状态转换。这代表了处理器执行的多种情况。我们可以将任意步骤中的机器状态表示为 $V^x$ ，其中 $x \in [0, N-1]$，且 $N \in \mathbb{N}$ 是步骤总数。在每个执行步 $V^x$中，机器状态的内容都可能与前一步骤不同。因此，在转换序列 $V^0 \rightarrow V^1 \rightarrow ...V^x ... \rightarrow V^{N-1}$中，任意步骤 $x$ 对应的机器状态为 $[B_1^x, B_2^x ... B_M^x]$。系统的初始状态称为 $V_0$ ，步骤 $x$ 的状态称为 $V_x$。

压缩序列。 在压缩序列中，每个步骤都关联一个时间。时间是一个非负整数。时间的一个具体例子是同步电路中的时钟。每个步骤中，时间可以增加一个或多个周期，并且始终至少增加一个周期。

使用这一扩展，我们现在可以将状态变化的时间序列表示为 $V_{t_0}^0 \rightarrow V_{t_1}^1 \rightarrow ...V_{t_x}^x ... \rightarrow V_{t_{N-1}}^{N-1}$。步骤计数始终每次增加一，但时间可以增加多个周期。

**事件**。 Maestro 模型包含一个事件规范列表。列表中的每个事件规范 $[E(\alpha,\ \beta,\ \gamma,\ \delta,\ \epsilon)]$ (即， $\mathcal{E}$简称为)，由事件规范名称（$E$）、携带数据 （$\alpha$）、条件事件触发条件序列（$\beta$）、条件状态转换 序列（$\gamma$）、时间延迟 （$\delta$）以及初始时存在 标志（$\epsilon$）组成。

事件规范的任何事件实例 $e(\alpha、\ \beta、\ \gamma、\ \delta、\ \epsilon)$ 都包含以下数据： $\alpha$ 包含字段 $[d_1, d_2 ... d_r]$，每个字段的位值均为 0 或 1。$\beta$ 包含触发事件 $[c_1$ 的条件： $E_1$、 $c_2$： $E_2$ ... $c_p$： $E_p]$ ，其中 $c_i$是基于当前机器状态以及 $\alpha$ 的布尔函数，而 $E_i$ 是列表中的一个事件规范。

$\gamma$ 包含状态转换及触发这些转换的条件。它表示为一个列表 $[cc_1$： $st_1 \leftarrow NV_1$、 $cc_2$： $st_2 \leftarrow NV_2$，...， $cc_q$： $st_q \leftarrow NV_q]$。 $\delta$ 字段表示在准备好触发其子事件和状态转换之前应经过多长时间。这意味着，如果某个事件在特定步骤 $x$ 的时间 $t_x$出现，那么其子事件只有在步骤 $y > x$ 的时间 $t_y \geq t_x + d$才会变为“活动”状态，即子事件可以在之后的步骤中触发。PresentAtStart 起始时存在标志$\epsilon$ 表示该事件实例在步骤 0 时是否存在。其取值可以是 0 或 1。

**事件序列**。一次执行是一个事件序列$(V_{t_0}^0,\ [e]_{t_0}^0) \rightarrow (V_{t_1}^1,\ [e]_{t_1}^1) \rightarrow ...V_{t_x}^x,\ [e]_{t_x}^x ... \rightarrow V_{t_{N-1}}^{N-1},\ [e]_{t_{N-1}}^{N-1}$。
$[e]_{t_x}^x$ 表示特定步骤 x 期间的所有事件实例。

在第一步（$x = 0$）中，会将特定的状态值和事件实例指定为初始条件。在任意步骤 $x >= 0$中，我们都会检查所有触发条件，即其 $\beta$中的布尔函数 $c_i$ 。如果在步骤（$x$）中满足任一条件 $c_i$ ，则相应事件$E_i$ 的一个实例会在下一步（$x+1$）中触发。在这种情况下，下一步与当前步骤之间的时间差，即（$t_{x+1} - t_x$），设为 1。如果我们检查前一步中的所有事件，并发现有些事件尚未达到其延迟阈值，这些事件将继续

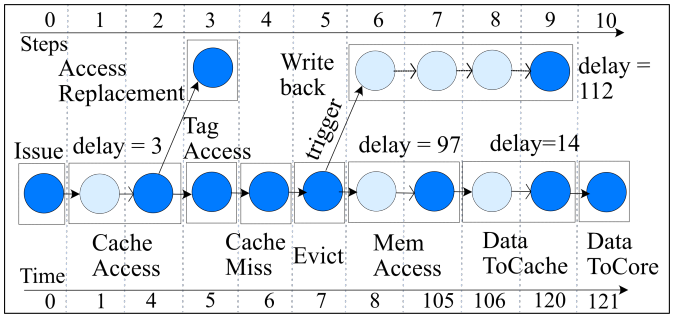


图 1：展示事件树构建中关键概念的示例。方框表示一个事件实例。在给定步骤（列）中，事件处于激活状态（深蓝色圆圈）或待处理状态（浅蓝色圆圈）。事件最终总是处于激活状态。该事件树表示缓存未命中的事件序列，具体解释见第 III-A 节。

```
#Maestro DSL 5—位计数器示例
———
机器状态:
  — 类型规范:
      —计数器: {entry: BV[5]}
  — 实例规范:
      —ctr1: 计数器
事件:
  —名称: “ClockEdgeEvent”
      携带数据: “无”
      触发事件: “触发时钟边沿事件{NONE}”
      状态变化: “SC ctr1.entry <— ctr1.entry+1”
      时序延迟: “0”
      初始时存在: "Yes"
断言:
  —名称: "递增_计数器"
      断言: "ALWAYS ctr1.entry'=ctr1.entry+1"
初始状态:
  —  约束1: "ctr1.entry = 0"
最大步数: 33
整数位宽: 7
```

清单 1：一个使用 Maestro DSL 的示例，表示一个每个步骤都会更新 5 位计数器的简单电路§III-B。

延续到这一步。最后，如果前一步中有某个事件已达到其触发阈值，但它的 $c_i$条件均未激活，那么该事件也会原样出现在下一步中。在这种情况下，时序将调整为大 1，因为我们不知道它最终会在何时触发。

**事件树**。根据上述约束规范，事件序列实际上会根据事件的触发条件和时序形成一棵事件树。输入配置为初始机器状态和初始事件的状态赋值。不同的输入配置（即不同的初始条件）可以形成不同的事件树。下面的示例用于阐释事件树的概念。

在图 1 中，我们从初始步骤 $s_0^0$ 中的发起事件开始。上标包含步骤编号，下标包含时间值。发起事件在 $s_1^1$ 触发缓存访问事件。假设指定的延迟$\delta$ 为 3，缓存访问事件会一直处于待处理状态（如浅蓝色节点所示），直到步骤 $s_4^2$。随后，缓存访问事件在 $s_5^3$ 同时触发访问替换事件和标签访问事件。标签访问事件之后，由于其中存在标签不匹配，在 $s_6^4$ 出现缓存未命中事件；否则将出现缓存命中事件，从而形成不同的事件树。缓存未命中事件在 $s_7^5$ 触发驱逐事件。由于被驱逐的缓存行是脏的，此次驱逐会触发回写事件以及针对未命中缓存行的内存访问事件。在 $s_8^6$ 中，这两个事件出现。内存访问事件会持续到下一步骤 $s_{105}^7$，以满足其内存访问延迟为 97 个周期的延迟要求。随后，它触发数据写入缓存事件。该事件在下一步骤 $s_{106}^8$ 被触发，并继续处于待处理状态 14 个时间单位，以便在下一步骤$s_{120}^9$将数据发送至缓存。同时，在 $s_{120}^9$，回写事件已满足其 112 的延迟要求。在最后一步 $s_{121}^{10}$，数据写入缓存事件触发数据传输至核心。

**安全属性**。 在执行的每个步骤中，我们可以强制实施安全属性 $\mathcal{S}(Sequence)$，该属性是当前事件序列的真值函数。要满足该安全属性，真条件应始终为真。如果在任何步骤中为假，则意味着该属性已被违反。

一项关键的安全属性是非干扰 属性。这意味着攻击者无法观察到依赖机密的状态所产生的影响。非干扰通过考虑系统的两次不同执行来形式化，这两次执行的依赖机密状态不同，并要求攻击者在两次执行中观察到的内容相同。在 Maestro 模型中，可以将其表示为两棵事件树（代表两台相同的机器），每棵树都访问一份输入配置的副本；两份配置除初始依赖机密值不同外完全相同。

### *B.* 用于实现 *Maestro* 的领域特定语言

Maestro 是一种领域特定语言（DSL），它基于 YAML 格式 [7] 构建，用于为 Maestro框架创建微体系结构规范。其规范包含几个部分：机器状态、事件、断言、初始状态、最大步数、和整数位宽。下面我们以 5 位计数器为例介绍各个部分（见清单 1）。

**机器状态（$\mathcal{H}$）**。规范的第一部分是对机器状态的描述。

这一部分包含两项内容。第一项是类型规范，其中列出了系统中各种硬件组件的类型。第二项是实例规范，其中列出并命名了系统中每个组件的实例。例如，我们可以有一个数据缓存和一个指令缓存，它们都是同一种缓存类型的实例。实例规范和类型规范共同构成机器状态。

在清单 1 中，有一个类型为计数器的系统组件。它有一个名为 ctr1 的实例。

其机器状态的大小由入口字段指定，即 ctr1.entry；在此示例中为 BV[5]（一个 5 位位向量）。

**事件**（$\mathcal{E}$）。本节介绍系统的不同事件规范。每个事件都有一个名称字段，用于为事件规范提供名称。携带数据、触发事件、状态变化、时序延迟和初始时存在分别对应于 $\alpha$、$\beta$、$\gamma$、$\delta$ 和 $\epsilon$。

在清单 1 中，计数器事件的名称是 ClockEdgeEvent。携带数据包含事件中的数据字段列表。

由于 ClockEdgeEvent 不携带任何数据，因此将其标记为 None。触发事件规范允许事件有条件或无条件地触发另一个事件，并根据需要为该事件中的数据字段赋值。对于计数器，该字段为 Trigger ClockEdgeEvent {NONE}，这意味着 ClockEdgeEvent 会在下一步骤中无条件触发，且不会为该 ClockEdgeEvent 赋值。

状态变化规范表示一系列有条件的状态更新。在清单 1 中，状态变化规范为 SC ctr1.entry <- ctr1.entry+1。这意味着，触发 ClockEdgeEvent 时，计数器值会递增 1。时序延迟字段是一个非负整数，表示触发的事件完成前还需经过的时间单位数。初始时存在字段是一个值为“Yes”或“No”的字符串，用于表示事件规范的实例在步骤零时是否存在。用于编写 YAML 规范并将其编译为 Alloy 的 $\alpha$、$\beta$ 和 $\gamma$ 的完整语法见在线补充材料 [5]。

**断言**（$\mathcal{S}$）。断言用于指定安全属性，例如非干扰和常数时间要求。断言用于在执行的每个步骤（ALWAYS）或执行结束时（FINALLY）检查安全属性。在清单 1 中，断言 ALWAYS ctr1.entry'=ctr1.entry+1 表示计数器 entry 在下一步骤中的值始终比当前步骤中的计数器 entry 大 1。

**初始状态**$[B_1^0, B_2^0 ... B_M^0]$。默认情况下，初始状态没有约束。如果初始状态存在任何约束，则会在 InitialState 字段中指定。在清单 1 中，“ctr1.entry = 0”表示计数器的初始状态为零。

**最大步数（N）**。YAML 规范的下一个字段是最大步数。此字段限定了模型的搜索空间。其值应足够大，以涵盖我们要检查的完整功能。具体值可根据上下文设定。在清单 1 中，该值为 33，涵盖了 5 位计数器回绕的情况。

**Int 位宽**。 此字段决定最终生成的 Alloy 代码中 Int 字段的位宽。Int 字段位宽越高，我们就能测试模型的时序和步骤范围越大，但运行时间也会增加。在清单 1 中，该值至少需要为 7，才能纳入 33 个时序步骤。

```
-- 模块和签名定义 1 open bitvector as bv 2 sig ClockEdgeEvent { 3 var 状态:
```

清单 2：展示签名和约束初始化的 Alloy 代码片段，由 Maestro 清单 1 生成。

*C.* 将 *Maestro* 的领域特定语言转换为 *Alloy*

下面介绍从 Maestro 转换为 Alloy 的实现方法。随后，我们将在 §III-C.1 中讨论关键对象和关系。在 §III-C.2 中讨论事件序列和断言。清单 2 和 3 是根据 Maestro 规范生成的 Alloy 代码的删节版本。注释以“-”或“///////”开头。注释末尾的“///////”表示由于篇幅限制，省略了该部分的代码。

**Alloy 中的关系和对象**。

在此语境下，Alloy 的关键特性是定义对象（即原子）、对象之间的关系，以及这些关系随不同步骤发生的变化。对象是 Alloy 中最基本的单元。Alloy 定义两个（或多个）对象集合之间的关系。有关对象的所有信息都通过关系表示。例如，计数器的值是计数器对象与整数对象（如“5”）之间的关系。每个对象都属于一个签名，类似于 Java/C++ 中的类。例如，语法 (sig counter {val: Int}) 定义了一个对象类，其中计数器类的每个对象都有一个指向整数对象的 val 关系。

*C.1)* 初始化与定义

**步骤与时间**。 Alloy 有“步骤”的概念，但没有“时间”的概念。此外，它本身不会维护步骤计数。为了表示这些概念，我们定义了一个步骤记录 和一个计时记录。我们约束每个类都恰有一个对象，并在第一步将步骤记录和计时记录都初始化为零。步骤记录被约束为每一步递增一。计时记录被约束为每一步递增，但递增量不受约束。相应的约束

Alloy 声明（清单 2 的第 10 行和第 11 行）为 StepRecord.step'=add [StepRecord.step,1] 和 TimingRecord.time'>TimingRecord.time。左侧的 ' 符号（step' 而非 step）表示某个关系将在下一步骤发生变化，这在 Alloy 中也称为变异。

**机器状态**。 我们使用 sig 关键字将机器状态定义为 Alloy 对象。Maestro 后端会将 Maestromachine 状态中的每个条目（来自实例规范和类型规范）转换为 Alloy签名。清单 2 第 4 行展示了 Maestro 中机器状态的一种表示。在本例中，为 Maestro 机器状态 ctr1.entry 生成的对象签名是 one sig ctr1_entry {var val: 5 位向量}。这表示存在一个名为 ctr1_entry 的对象，并且它与一个名为 val 的 5 位向量对象相关联。sig 关键字前的 one 表示该签名恰好对应一个对象。若省略 one 关键字，则对象数量可以是任意的。

**事件**。 事件规范由签名定义。事件实例由该签名的对象表示。清单 2 第 2–3 行展示了 ClockEdgeEvent 的事件签名。状态关系指定对象在其生命周期中的位置。它可以是未部署(0)、已部署但待处理(1) 或 已部署且处于活动状态(2)。出现时间关系指定对象实例被部署的时间。时序延迟表示触发的事件可以退出前的延迟。事件 ID 是分配给事件对象的唯一整数。原因是导致事件被触发的条件的索引。父 ID 是触发事件的事件 ID。

初始约束。 使用 fact 指定的关系初始约束（在第一步）描述了系统的初始状态。清单 2 第 13 行约束 ClockEdgeEvent 对象的状态 至少为 1。同时，它还约束计数器对象的值为零（使用 bitVecFromBits5 谓词），并将步骤值和时间值设为零。第 14 行约束所有已部署的 ClockEdgeEvent 的出现时间和延迟均为零。

范围与唯一性约束。 清单 2 第 19 至 24 行展示了范围与唯一性框架。唯一事件 *ID* 约束规定，每个事件对象都有不同的整数事件_ID。事件 *ID* 范围约束确保具有相同签名但属于不同对象的事件具有不同的事件 ID。状态范围约束要求状态字段的值为 0、1 或 2。事件实例计数约束规定每种事件签名对应的对象数量上限。绑定位向量代码约束每个机器状态都引用一个唯一的位向量对象。

*C.2*）事件序列、状态更新与断言

事件序列构造包含四个部分，即部署、维护、状态更新和完成（清单3第4行至第8行）。这些不同部分各自用于强制更新关系，这些关系共同实现 Maestro 规范。部署部分强制触发事件，即确保存在一个状态关系从 0 更新为 1，同时还强制维护对象的数据关系。维护部分规定了约束条件，这些条件必须成立，才能使对象的状态关系在下一步为 1（已部署但待处理）或 2（已部署且处于活动状态）。如果延迟在下一步尚未结束，状态为 1；否则为 2。状态更新部分中的约束规定，机器状态对象的关系会随着事件完成而更新。完成部分约束事件在延迟结束后退出，即状态关系从 2 更新为 0。

**断言**。 Maestro 中的 ALWAYS 断言在 Alloy 中表示为 assert 语句内的 always 块。清单3第 10–17 行展示了使用此模式的示例。第一个断言（第 10–12 行，alwaysOneClock）检查 status 字段值至少为 1 的时钟边沿事件对象的数量（#）是否为 1。第二个断言（第 13–15 行）检查位向量 ctr1_entry.val 是否在每个步骤都递增 1（addBitsToVec5 谓词）。第三个断言（第 16–17 行）检查下一步骤中的整数 time 值（TimingRecord.time'）是否始终比当前步骤中的 time 值大 1。add 函数执行加 1 操作，而 = 运算符用于比较整数 time 值。

check 语句用于评估断言。对于每条 check 语句，如果断言失败，Alloy 都会生成一个反例。

Alloy 的 check 功能可以识别 Maestro 模型中的任何 MDAV。辅助脚本会将反例中的关系变异序列转换回 Maestro 事件树。

**最大步数和 Int 位宽**。 可以使用 step 关键字在 Alloy 中设置最大步数（本例中为 33）。Int 对象的位宽 7 被设置为 7 Int（见清单3第 18 行）。

## IV. 使用语义组合的两级模型集成工作流

为了将阶乘级复杂度约束在实际可行的工作流内，一种直观的策略是将集成过程拆分为两个 层级。在第一级中，我们有需要集成的不同模型。在第二级中，我们有一个累积模型，以及一个作为当前集成重点的选定模型。

基于上述两级策略，我们提出了一种模型集成工作流。在此工作流中，设计者首先选择希望依次集成的一组防御措施。为了促进这一集成过程，我们提出了一种可组合防御模型定义策略，将基线 Maestro 模型（不包含防御措施）以及希望集成的防御措施（例如 Spectre 防护）表示为变换函数。简而言之，我们称其为变换（见 §IV-A）。

设计者迭代地将所选的新 Maestro 防御模型与先前集成的防御措施相结合，并检查是否存在 MDAV（表示为通过 Alloy 检查获得的 Maestro 事件序列

```
... 初始化 ...
1-- 生命周期约束
2 事实{始终{
3 StepRecord.步骤 < 32 ⟹{
4 ////// 部署、维护、状态更新 ,
    ↪ 完成 //////
8 }}}
9 -- 要检查的断言
10断言 alwaysOneClock {
11     always {StepRecord.step < 32 ⟹ (#{e:时钟边沿事件
    ↪ | e.状态 > 1} = 1)}
12 }
13断言 alwaysIncrementCounter {
14     始终 { StepRecord.步骤 < 32 ⟹addBitsToVec5[One,
    ↪  零, 零, 零, 零, ctr1_entry.val] }
15 }
16断言 alwaysIncrementTime {
     始终 { StepRecord.步骤 < 32 ⟹(TimingRecord.时间'
    ↪ = 添加[TimingRecord.time, 1])}
17 }
18 运行 {} 持续 33 个步骤, 7 整数
19 检查 alwaysOneClock 持续 33 个步骤, 7 整数
20 检查 alwaysIncrementCounter持续 33 个步骤, 7 整数
21检查 alwaysIncrementTime 持续 33 个步骤, 7 整数
```

清单3：事件生命周期约束和断言，由清单 1 中的 Maestro 规范生成。

语句的反例）。如果存在 MDAV，设计者就会修改防御措施并再次进行集成。迭代会持续进行，直到不存在 MDAV，或没有明显的解决方案为止（参见 §IV-D）。

### *A.* 支持可组合模型的变换

*1*）事件变换：事件变换可以向事件规范列表中添加事件，也可以以可组合的方式修改现有事件规范。例如，可以向事件添加数据字段，但不能删除数据字段。另一个例子是，Maestro 允许设计者对事件的布尔条件使用逻辑与或逻辑或变换，但不能同时使用两者。

*2*）机器状态、断言和初始状态：机器状态变换会向 Maestro 模型添加机器状态。

断言和初始状态变换会将断言和初始状态约束添加到断言和初始状态列表中。有关可组合事件变换的完整详情，请参阅在线补充材料 [5]。

*3*）模型可组合性：组合防御措施的顺序不会影响组合模型。例如，如果我们将防御措施 A 和 B 添加到基线模型中，那么按基线 + A +B 的顺序组合，与按基线 + B +A 的顺序组合，得到的组合模型相同。

### *B.* 非干扰变换

A 非干扰属性意味着攻击者无法观测到依赖密钥的状态所产生的影响。非干扰通过以下方式形式化：考虑系统的两个不同执行过程，这两个执行过程在依赖密钥的状态上存在差异，并要求攻击者对这两个执行过程的观测结果相同。

为了在 Maestro 中对非干扰属性建模，我们复制给定模型的机器状态和事件规范，从而实际得到两台机器。在初始步骤中，依赖机密的状态不受约束，因此两台机器中的该状态可以不同，而其余状态则被约束为在两台机器中相同。为了检查非干扰属性，ALWAYS 断言会检查两台机器中的可观测机器状态是否始终相同。示例见 §V-A。此变换并非用于添加防御措施，而是用于检查 MDAV。

### *C. Integra* 变换

为了实现具有所需组合属性的变换，我们在Maestro中实现了另一种领域特定语言，称为*Integra*。Integra中有15种可组合的变换 [5]。两个或更多Integra程序通过将变换汇集到一个程序中进行组合。非干扰检查使用四种关键变换。

前两个变换实际上创建了一个建模系统的副本，该副本具有相似的状态，但状态值可能不同。第三个变换允许某个状态的初始值与另一台机器中的对应状态之间存在依赖密钥的差异。第四个变换检查某个状态及其在另一台机器中的对应状态是否始终具有相同的值（非干扰）。

### *D.* 使用 *Maestro* 的模型集成工作流

设计者从两个或更多 Maestro 模型开始，每个模型代表针对同一基线模型的一种不同防御。

图2以流程图的形式展示了防御集成工作流程。在步骤 (1) 中，工作流程从一个基线模型开始。在步骤 (2) 中，设计者选择一个尚未集成的防御模型。随后，该模型与已集成的模型组合。在步骤 (3) 中，检查生成的 Maestro 规范是否满足安全属性（包括使用非干扰变换的属性）。如果不存在 MDAV，工作流程便转到步骤 (4)。在步骤 (4) 中，工作流程要么结束，要么转到步骤 (2)。如果没有更多防御模型需要添加，工作流程就此结束。否则，在步骤 (2) 中选择下一个防御模型。另一方面，如果组合模型不满足所需的安全属性，

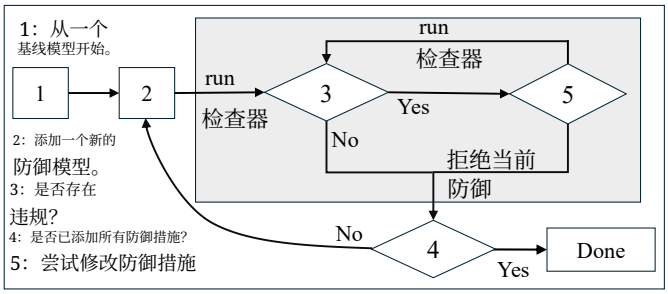


图 2：我们用于研究多重防御之间交互作用的模型集成工作流。我们一次添加一个防御措施，以逐步构建更大的防御体系，同时保留拒绝导致违规且无法修改的防御措施的可能性（见第 III-A 节）。

设计者会在步骤 (5) 中利用提供的反例修改防御措施，然后返回步骤 (2)。

在步骤（5）中，如果设计者想不出修改方案，则可以选择返回步骤（4）。

## V. 示例：Maestro 集成

在本节中，我们使用 TORC 和 DSRC 防御措施来说明 Maestro框架的集成工作流程。这是我们的两步 MDAV筛查方法的第一步示例。我们选择 TORC 和 DSRC 进行研究，是因为它们针对不同但重要的攻击，即利用缓存一致性状态和基于缓存刷新的侧信道的 Spectre攻击。与 DSRC 不同，TORC 防御措施可以消除基于缓存命中的攻击，而此类攻击不一定使用推测执行。在 Maestro 中，我们将一个具有两个核心的简单处理器建模为基线模型，该处理器执行一条可能为推测执行、也可能不是推测执行的加载指令。

基线模型和防御变换将在 §V-A 中讨论。我们将在 §V-B 中使用我们的工作流集成这两个模型。

### *A.* 基线模型和变换

**基线模型**。 在加载指令的事件序列中，我们首先从一个发起事件开始，然后是缓存命中事件或缓存未命中事件。如果是缓存命中事件，则接下来会发生完成事件。否则，将先发生内存访问事件，随后是缓存更新事件，最后是完成事件。

用于表示缓存行的相关机器状态包括缓存行地址、存在/缺失位、共享者位向量，以及表示独占/共享（或 E/S）的一致性状态位。E 状态用位值 0 表示，S 状态用位值 1 表示。还有一个完成位，在处理器核心中的加载完成时置位。在初始状态下，完成位置为 0。如果存在位为 0，则该缓存行中的其他机器状态位均为 0。此外，如果共享者位向量中有多个位被置位，则一致性位为 S。

基线模型没有需要通过断言来强制执行的安全属性。不过，加载指令应能正确运行。具体来说，模型要求在执行结束时完成位置位。这通过 FINALLY 断言来表达。

**远程缓存行时序混淆（TORC）变换**。对于 TORC 防御，延迟变换会增加时序延迟

CacheHitEvent 规范中的字段，使缓存命中与未命中的可观察时间相同。

为检查安全性断言（即非干扰），我们执行两种变换（见 §IV-B）。第一种变换复制机器状态规范和事件规范，但初始内容可以不同。第二种变换关联两台机器的初始状态。初始时，两台机器中的存在位或缺失位可以不同，但两台机器的其余机器状态相同。

断言是，两台机器上的完成位会在同一时间设置。因此，无论缓存行是否

存在（可能是密钥）/不存在，都无法通过缓存命中的时序观察到。

**延迟远程缓存行的推测性更改（DSRC）变换**。对于 DSRC 防御（见 §II），事件变换会修改 CacheHitEvent，使其根据推测位的值采取不同的行为。

如果初始时未设置推测位，则执行序列与基线相同。如果初始时设置了推测位，则 CacheHitEvent 会被拒绝，随后依次发生 ReturnToCoreEvent、ReIssueEvent，然后再次发生 CacheHitEvent（重新启动）。引入的 ReturnToCoreEvent 向处理器核心表明，由于一致性状态可能发生变化，缓存命中不能以推测方式发生。引入的 ReIssueEvent 会在处理器核心确认加载位于正确路径后发生。ReturnToCoreEvent 和 ReIssueEvent 会阻止 CacheHitEvent 在处理器核心确认加载位于正确路径之前修改一致性状态。重新启动的 CacheHitEvent 之后的其余执行过程与基线相同。

该安全断言仍然是一种非干扰形式。第一个变换会复制机器状态和事件规范。第二个变换按如下方式关联两台机器的初始状态：加载操作的地址位可以不同，但机器状态的其余部分相同。推测位设为 1（对两台机器均如此）。断言要求两台机器的缓存状态和完成位始终相同。因此，推测访问的地址（可能是密钥）所产生的影响无法被观察到。

### *B.* 组合变换并检查安全性

我们从基线模型开始工作流的步骤 (1)。我们从一组选定的防御措施中选择一项（此处只有两项防御措施：TORC 和 DSRC）。本例中，我们先选择 TORC，并将其变换应用于基线模型。Maestro 会检查是否满足非干扰属性。本例中，它发现该属性得以满足，并在步骤 (3) 报告未发现反例。在步骤 (4)，我们检查是否已整合所选的所有防御措施，然后转到步骤 (2)。本例中，我们选择 DSRC，并将其与基线 +TORC 模型组合。

在步骤 (3) 中，Maestro 会检查组合模型，并报告一个反例。该反例表明，TORC 的非干扰属性遭到违反，因为组合后的事件规范在 CacheHitEvent 路径上存在双缓存未命中延迟。当两台机器上发出加载请求的核心的共享者位均为零时，在存在位已设置的机器上会观察到较长的延迟。

长延迟的来源是 TORC 与 DSRC 之间的 MDAV。DSRC 会导致缓存命中事件发生两次：一次是在被拒绝时，另一次是在重新启动时。TORC 在这两种情况下都会施加未命中惩罚延迟。这导致总延迟加倍，从而重新引入时序通道。

**修订防御措施**。这个反例表明，DSRC 命中路径的拒绝/重启行为会导致 TORC 延迟的应用不均匀。

解决时间差的两部分改动是：减慢未命中，并将命中加速到相同的时序。首先，我们修改 DSRC 的未命中情况，使得在推测缓存未命中路径上，缓存未命中事件会生成返回核心事件和重新发出事件。其次，我们修改 DSRC 的命中情况，将推测路径上的缓存命中事件替换为新引入的推测缓存命中事件，使其不受 TORC 变换的影响。因此，推测路径无论命中还是未命中都不会产生 TORC 延迟。非推测路径在缓存命中时始终会产生 TORC 延迟。由此满足非干扰属性。

我们还考虑了另一种修订方案，将命中和未命中路径都减慢到相同的时序。除了速度更慢之外，这种方案还需要同时修改 TORC 和 DSRC 的变换。我们已在在线补充材料 [5] 中介绍了不安全的组合以及各种修订后的防御措施。

### C. 使用另一种基线模型

另一种可能性是使用不同的基线模型，其中一致性位始终为 1（共享），这可通过在基线模型中添加额外的初始状态约束来实现。这相当于将一致性协议从 MESI 更改为 MSI 的效果。在这种情况下，TORC 和 DSRC 的非干扰属性均未被违反。由于推测执行不会触发拒绝/重启行为，因此不存在 MDAV。我们在在线补充材料 [5] 中提供了这种替代方案的完整细节。**工作流优化**。

我们可以并行应用这两种集成工作流程，从而无需任何人工干预即可满足所需的安全属性。还可以扩展这一策略，同时运行多个并行工作流，以更快地覆盖各项安全要求。

## VI. 示例：扩展模型

在本节及下一节中，我们将在模拟器上研究 TORC 和 DSRC 防御。这是两步 MDAV 筛选方法中第二步的一个示例。在本节中，我们将研究一种基于 §V 中反例的攻击实现，该实现针对 TORC 和 DSRC 组合不当及组合恰当的变体。在 §VII 中，我们将在 GEM5模拟器中模拟该攻击。

### A. 实现攻击环境

我们基于采用 MESI 协议的包含式三级缓存中的缓存一致性协议构建攻击环境，该协议类似于 GEM5 的包含式三级缓存协议 [3]。我们在类似 GEM5 O3 [51], 的双核乱序处理器上集成了 TORC 和 DSRC 支持，并支持推测执行（例如分支推测和加载重排序推测）。

*1) TORC*：假设一个处理器核心（发送方）访问某个缓存行地址，并将该缓存行加载到末级缓存中。随后，另一个处理器核心（接收端）访问末级缓存中的这条远程缓存行。末级缓存中的远程缓存行命中需要延迟，因此会发起一次主存访问来制造延迟。我们会将缓存命中的响应缓存在接收核心附近的私有缓冲区中，直到延迟访问返回后，再将其释放给该处理器核心 [71]。

*2) DSRC*： DSRC 及类似防御措施需要硬件支持，以识别可能受到推测性攻击影响的指令。

常用的推测保护决策模型（SPDM）有两种: BranchShadow [8], [97], [103]和 ROB-Head [9], [10], [97]。在*BranchShadow*中，当加载位于分支影区内时，即当 ROB 中存在一个尚未解析的更早分支时，该加载会被推测性地发出。在*ROB-Head* 模型中，所有加载都会被推测性地发出，除非该加载在发出时位于 ROB 的队首。在将加载请求发送到缓存之前，通过检查 ROB，会将一个表示 SPDM 结果的推测保护标志添加到加载请求中。该标志值为 0 表示该请求不是受保护的加载，因此无需触发 DSRC 反馈。值为 1 表示该请求可能是不安全的推测性加载，需要受到保护，并且确实需要触发 DSRC 反馈。从 L1 到 L2，以及从 L2 到 L3（即末级缓存）发送的 GETS[1] 请求中也包含此标志。

*DSRC*缓存反馈。 当缓存行发生 L3缓存命中时，系统会检查其是否处于远端E/M一致性状态（即接收核心的共享者位未置位，且一致性状态为 E 或 M），此时最终会用到该标志。如果该标志已置位，则会执行额外检查。如果发现远端E/M一致性状态，系统便会向接收核心发送 REMOTE-EM响应消息。如果该标志未置位，则应用常规 MESI协议。如果向接收核心返回了 REMOTE-EM，则当推测保护决策模型判定加载安全时（如果判定安全），接收核心会重新发出该加载。

*3)* 缓解措施： 我们实现了两个经过修复的模型的更具体版本，并使用 Maestro 对其进行了实现和评估（参见 §V-B、§V-C）。首先，远端未命中时延迟推测性更改 （**DSRM**）通过在远程缓存行发生缓存命中和缓存未命中时，向私有缓存发送 REMOTE-EM响应消息来实现时序均衡。只有当 GETS 的推测保护标志置为 1 时，才会进行时序均衡。MESI协议的其他方面（例如一致性状态）保持不变。其次，在以 *S* 状态启动的 *MESI* （**SS-MESI**）中，我们向末级缓存添加了一个控制标志，使其将所有加载未命中都以 S 状态而非 E 状态启动。

### B. 实现攻击代码

我们提出了一种基于 Maestro 计数器示例的攻击。该攻击的核心部分是一个加载右路径分支影子 (LRBS) 探测，它是一段由

[1]MESI协议中的 GETS请求是指缓存控制器在发生读未命中时向下一级缓存级别发出的读取请求。其目的是获取数据并更新一致性状态。

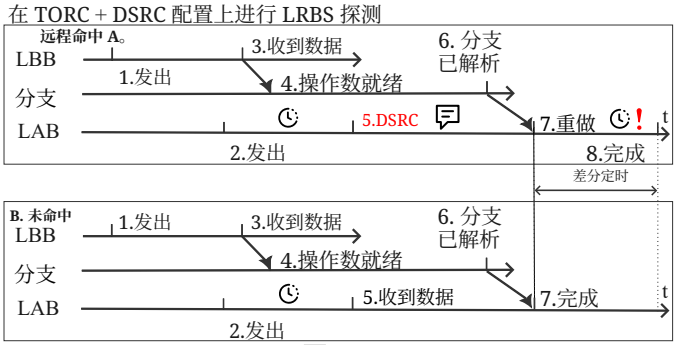


图 3：时间线展示了在 TORC + DSRC 配置中，如何利用 LRBS 探测来利用因向处理器核心发送一致性信息而产生的 MDAV。情况 A 展示了探测的时间线，此时末级缓存（LLC）中存在处于 E 状态的远程缓存行（因重做而较慢）。情况 B 展示了发生缓存未命中时的时间线（较快）。

接收端执行的代码序列。首先，我们讨论 LRBS 探测中的三条关键指令（§VI-B1）。其次，我们用时序图解释 LRBS 探测的工作原理（§VI-B2）。第三，我们描述该探测的一种 x86_64实现（§VI-B3）。

*1)* 攻击代码中的关键指令：首先是加载-分支前（LBB）指令，它在执行期间会导致缓存未命中。这会放大推测效应（即产生一个较大的推测窗口）。LBB 访问的缓存行地址不会被发送端核心触及，且仅由探测控制。其次是分支指令，它依赖于 LBB 指令的结果，之后解析为不跳转。该分支会在探测中创建一个推测窗口。第三是加载-分支后（LAB）指令，它位于不跳转路径（即右侧或正确路径）上。该 LAB 指令利用推测来触发重做操作。LAB 加载的缓存行地址对应于发送端核心所触及的缓存行地址。

**训练探测程序。** 我们首先多次运行探测程序来训练分支预测器。分支预测器被训练为预测“未跳转”，因此在分支解析之前，处理器核心会推测性地执行 LAB 指令。训练完成后，探测程序便准备好检测发送方留下的副作用。

*2)* 在 *TORC* 中制造时序差异 + *DSRC*： 探测程序的目标是根据是否存在远程缓存行来有条件地触发重做。图 3 展示了一个触发重做的时序图示例。这里有两种情况：一种是远程缓存行命中（情况 A），另一种是缓存未命中（情况 B）。

**远程缓存行命中。**

我们首先讨论远程缓存行命中场景的事件时间线。

LBB 和 LAB 指令向缓存发出加载访问。LBB 首先被调度到执行阶段（时间点 *1*）。由于发出得更早，LBB 加载首先从缓存中获取数据（时间点 *3*）。分支指令的解析条件依赖于推测性加载访问（由从时间点 *3* 到时间点 *4*的箭头表示）。因此，LBB 加载未命中对于将分支解析延迟到更晚的时间点至关重要；如果 LBB 缓存命中，分支解析则会早得多。

与此同时，出于性能考虑，LAB 加载会在分支影区内进行推测性发出（时间点 *2*）。不久之后，分支指令便可获得分支条件（时间点 *4*）。再过一段时间，分支得到解析（时间点 *6*）。缓存对远程缓存行命中施加 TORC 延迟后，LAB 加载便会收到响应，向处理器核心表明远端 E 状态（时间点 *5*）。然而，LAB 加载尚不能提交，因为它是推测性地发往缓存的，因此在远程缓存行命中时并未收到任何数据。因此，LAB 指令会在流水线中等待，直到分支得到解析（时间点 *6*）。随后，处理器核心会将 LAB 加载重新发往缓存（时间点 *7*）；加载在此处经历 TORC 延迟，最终将数据返回给处理器核心（时间点 *8*）。

时间点 5 和 7 是利用该漏洞的关键部分。在时间点 5，DSRC 反馈发生；在时间点 7，不安全重做操作（以红色感叹号表示）发生。攻击者滥用 DSRC 反馈来触发重做，最终导致依赖秘密的时序差异。

**缓存未命中。** 在缓存未命中期间（图 3-B），会发生类似的事件序列，但 LAB 的时间线存在两个关键差异。第一个差异是，在时间点 5，数据会返回处理器核心，但不带有一致性状态。第二个差异是，LAB 没有重做步骤，因此会更早完成。缓存未命中与远程缓存命中之间的时序测量差异以虚线表示。

*3)* 在 *x* 上实现探测*86_64*： 清单 4 展示了 LRBS 探测的实现。在该探测中，LBB 位于第 8 行，LAB 位于第 11 行。LBB 将数据加载到 %r12d 寄存器中。分支指令位于第 10 行，使用依赖于 %r12d 的 jne。时序测量从第 5 行开始，即 LBB 之前；在分支完成跳转至第 12 行后，于第 14 行结束。lfence 指令会将计时器开始事件之前（第 4 行）和之后（第 6 行）的加载串行化。计时器结束事件同样应用了 lfence。最后，第 15 行计算时间差，第 16 行和第 17 行清除与 LAB 和 LBB 相关的缓存行。第 18 行表示时序结果记录在 %eax 寄存器中。第 19 行表示包含 LBB 和 LAB 加载地址的两个寄存器。

## VII. 攻击模拟

我们采用一种常见的GEM5模拟器配置，其中包括一个乱序执行核心、三级缓存、标准互连和DRAM。详细配置见 [5]。

我们将讨论攻击仿真配置和攻击仿真结果（见§VII-B）。

```
//LRBS 探测
1 asm__volatile__ ( 2 " xorq %%r12, %%r12\n" //将 %r12 初始
```

清单 4：加载-右路径-分支-影子（LRBS）探测的实现。代码的功能是触发依赖密钥的重做操作，相关说明见第 VI-B3 节。

### A. 攻击仿真配置

我们针对每次攻击模拟，在 GEM5 中实现了五种配置：$C_1$、$C_2$、$C_3$、$C_4$ 和 $C_5$。$C_1$ 对应不安全的缓存配置。$C_2$ 对应 TORC 实现。$C_3$ 对应 TORC + DSRC 配置。$C_4$ 对应 TORC +DSRM 配置，$C_5$ 对应 TORC + DSRC + SS-MESI 配置。当要传输的秘密值为零时，发送方不会产生缓存副作用。否则（秘密值为一），它会将一个远程缓存行置入末级缓存。我们针对每次攻击模拟进行了两次测量：一次是发送方传输秘密值零，另一次是发送方传输秘密值一（共 10 次实验）。每次实验运行 100 次，并记录中位数计时结果。

### B. 攻击结果摘要

**关于新型攻击。** 表 I（攻击结果）针对配置 $C_1$、$C_2$、$C_3$、$C_4$ 和 $C_5$ 各设一行。每列中的不同条目表示针对每次攻击进行的时序测量。时序列之间的差异表明攻击成功。DSRM（$C_4$）和 SS-MESI（$C_5$）防御措施的时序相等，表明攻击已得到缓解。该攻击的时序结果不能代表应用程序性能（参见 [5]）。

**隐蔽信道比特率估计。** 我们在一台真实机器（Xeon E5-2699 v3）上，使用基于 FLUSH + RELOAD 的策略，模拟基于 LRBS 的隐蔽信道攻击。纪元大小为 100 万个周期。为进行纠错，我们采取保守做法，将每个位传输 16 次，并在接收端进行多数表决。对 16 ×12288 = 196608 次单比特传输进行测量后，错误率为 0.3%。有效传输速率为 6KB/s。在此实验中，通过访问或不访问某个特定缓存来传输 1/0

| 使用 LRBS 的攻击模拟 | | |
|---|---|---|
| **防御配置** | **密钥=0** | **密钥=1** |
| 1. 不安全（$C_1$） | 205 | 199 |
| 2. TORC（$C_2$） | 205 | 205 |
| 3. TORC + DSRC（$C_3$） | 205 | 364 |
| 4. TORC + DSRM（$C_4$） | 364 | 364 |
| 5. TORC+DSRC+SS-MESI（$C_5$） | 205 | 205 |

表 I：$C_1$、$C_2$、$C_3$、$C_4$ 和 $C_5$ 分别对应不安全、TORC、TORC + DSRC、TORC + DSRM 和 TORC + DSRC + SS-MESI。$C_3$ 并不具备我们预期的、由 TORC 和 DSRC 防御措施组合所带来的攻击韧性。DSRM 或 SS-MESI 可恢复对 LRBS 探测的韧性。

行。我们通过调用等效的延迟循环（参见 [5]），模拟远程缓存行命中（DSRM）的较长缓存命中时间。

**关于原始攻击。** 我们还模拟了原始缓存命中攻击以及一种针对一致性状态的 Spectre 攻击，TORC 和 DSRC 分别对此进行防御（见 §II）。正如预期，$C_4$ 和 $C_5$ 也能缓解这些攻击。

## VIII. 通用性与可扩展性评估

我们讨论与通用性和可扩展性相关的若干问题。§VIII-A 介绍了 8 种 MDAV 和两种已解析的集成。§VIII-B 给出了两项压力测试的运行时间结果。Maestro 框架的实现工作量为 ≈9K 行 Python 代码。它使用 ≈ 450 行 Maestro DSL 代码生成了超过 11K 行 Alloy 防御模型，以及少于 200 行 Integra领域专用语言代码。表 II 展示了 MDAV 研究的关键结果。

### A. 展示 *MDAV* 的通用性。

重点关注的 MDAV 包括 TORC [63],[73], [98],DSRC [9], SIDD [8], SIDO [102], DoM [80],隔离 [42],CI [72] 和 SDR [14], , 如 §II 所述。

**八种 MDAV。** 这些 MDAV 有五种成因。1）由于另一种防御而导致推测窗口变长或变短（TORC+DSRC、TORC+SIDO）。2）防御措施错误地使用去分类位（TORC+SIDD）、（SIDD+CI）和（SIDD+DoM+CI）。3）干扰数据无关操作，使其依赖于密钥（TORC+SIDO）。4）防御措施意外触发不安全的一致性事务（CI+隔离）。5）防御措施未考虑到的 DRAM 刷新延迟（SDR+隔离）。

**两种已解析的 MDAV。** TORC+DSRM 和 TORC+SS-MESI 通过增加延迟，或移除导致延迟的密钥状态（见 §V），来避免 MDAV。CI*+ 隔离在允许单向信息流的场景中，使用单向一致性操作来避免 MDAV。

### B. *Maestro* 的可扩展性

可扩展性的一个重要维度是增加位数，并观察运行时间如何响应。我们使用 Maestro 表示了 2500 位的 FIFO 缓冲区（每个为 25 位）。

| Integra 中选定的防御模型变换与组合 | | | | | | | |
|---|---|---|---|---|---|---|---|
| **防御配置** | **受保护组件** | **相关攻击类别** | **Integra LoC** | **Alloy LoC** | **集成** | **Alloy LoC** | **Num MDAV** |
| 1. TORC [63], [71] | 缓存 | 缓存命中 [30] | 3 | 1022 | TORC+DSRC | 1480 | 1 |
| 2. DSRC [9] | 缓存 | Spectre [43] | 29 | 1480 | TORC+DSRM | 1661 | 0 |
| 3. SIDO [102] | 缓存、处理器核心 | Spectre [43] | 29 | 1445 | TORC+SIDD | 1370 | 1 |
| 4. SIDD [8] | 缓存，处理器核心 | Spectre [43] | 27 | 1376 | TORC+SIDO | 1445 | 2 |
| 5. DoM [80] | 缓存 | Spectre [43] | 43 | 1055 | DoM+SIDD+CI | 1330 | 2 |
| 6. 一致性隔离 [31], [72] | 缓存、处理器核心 | P+P [34] | 15 | 1560 | CI+隔离 | 1560 | 1 |
| 7. 隔离 [42] | 缓存，处理器核心 | Spectre [43] | 12 | 1480 | CI*+隔离 | 1950 | 0 |
| 8. 安全 DRAM 刷新 [9] | DRAM | RowHammer [61] | 23 | 1773 | SDR+隔离 | 1885 | 1 |

表 II：由 Maestro 建模并检查的部分防御措施、集成和 MDAV。每项实验均在 16G MacBook Air M4 上于 2 分钟内完成。Maestro 找到 8 个 MDAV，并实现 ≈ 15x 的 LoC Alloy 比率。所使用的三个 Maestro DSL 基线模型各包含约 150 LoC （见 §VIII-A ）。

使用 SAT 求解器 plingeling.parallel 求解该问题（共 10 个步骤）的总时间不到 2 分钟。使用 Alloy 的默认设置，CNF 生成步骤（Java）耗时不到 30 分钟。在另一项实验中，我们在一个类似双核乱序处理器的系统上设置了周期精确压力测试（包括重排序缓冲区、保留站、发射队列以及加载-存储队列）。该测试在检查 1550 位状态的同时，在不到 2 分钟内完成了 20 个步骤。

**高周期数**。为检验 Maestro 的翻译方法，我们使用两个相隔 62 个周期延迟的事件。Maestro 的检查在 2 秒内完成，而采用朴素的逐周期方法则需要 200 多秒。时间差会随着延迟增大而增加。

**多重防御**。为检验 Maestro，我们将 4 种防御措施组合起来，即 TORC、DSRC、SIDD 和 SIDO；它们在语义上与同一个基线模型组合。组合模型能在 1 分钟内检测到一个 MDAV。

## IX. 讨论与相关工作

**MDAV 会导致侧信道吗？** Maestro 检查非干扰属性；若这些属性成立，则表明不存在侧信道和隐蔽信道。侧信道可能更危险，因为它们涉及不知情的参与者。新的隐蔽信道（见表 I）可被重新用于攻击易受侧信道影响的软件（例如 mbedTLS [4]），原因是其存在依赖密钥的内存访问模式（例如 T表）。类似地，TORC 和 SIDO 的不当集成可能导致推测泄漏，因为 TORC 会影响错误路径上的数据无关操作（见 §VIII）。

**检测第二步中的实现问题**。考虑这样一种模型：所有易受攻击的共享数据，其推测访问都会根据页表中的共享者位延迟执行。然而，为了准确更新共享者位，需要对系统软件（例如 Linux内核）进行许多复杂的修改，这源于页面管理的复杂性（对比 [2]，后者需要新增指令）。在构建仿真时，这种难以承受的复杂性便显现出来，但在第一步中可以省略。

**早期阶段建模**。 Pensieve [99] 提出了一种建模规范，其中每个防御的实现都是一个模块。然而，由于采用类 RTL建模，引入新模块需要修改与该模块通信的其他模块，例如线连接。与 Maestro 相比，Pensieve 不支持基于事件的建模，也不支持将变更限制在单个模块内的组合方式。

另一种与 Maestro 抽象层次类似的方法，是让设计者指定一个事件图，用于处理内存一致性 [36], [53], [55]–[57], [89], [93], [105] 以及推测执行或其他泄漏问题 [59], [67], [88]。一个重要区别是，这些系统通常没有明确的时间概念。 [37] 中的工作支持与 Maestro 类似的周期精确事件图，但缺少用于组合的变换，也不支持父事件与子事件之间的多周期延迟。

**验证与模糊测试**。
Maestro 是一种早期阶段的集成补充，可用于更细致层级的分析 [46], [75], [81], ，包括 RTL级 [16],[17], [25]–[27], [37], [86], 单元级 [84]和门级 [107]。另一项相关研究是硬件模糊测试，其策略是使用 RTL 输入 [13], [22], [83], [96] ，或在微架构模拟器上运行防御措施实现 [28]。

**其他防御措施**。 除了迄今讨论的微架构攻击之外，还有许多针对其他微架构攻击的防御措施 [9], [10], [12], [23], [24], [31], [42], [47], [65], [69], [78], [79], [90], [97] [15], [18], [32], [41], [60], [60], [61], [66], [68], [82], [85], [92]。通过以防御方式修改缓存 [24], [31], [39], [64], [72],[78], [87], [108] 和非缓存 [11], [20], [35], [38], [41],[50]组件来缓解攻击。MDAV 留待未来研究。

## X. 结论

随着微架构攻击不断涌现，微架构安全日益受到关注。然而，在当前文献中，多重防御很少被放在一起研究。因此，缺乏经过验证、能够确保不存在微架构防御假设违规（MDAV）的集成设计。为此，我们提出了一种研究防御组合的两步方法。第一步，我们提出并实现了一个建模框架*Maestro*，以及一个用于迭代语义组合的工作流程。Maestro 揭示了先进防御机制之间的八项 MDAV，实现了紧凑表达（Alloy 代码行数比为 15 倍），支持无缝语义组合，并消除了 100 倍的性能下降。

第二步，我们在微架构模拟器上实现了一种集成的先进防御机制。在评估中，我们提出了一种针对 MDAV 的新型隐蔽信道攻击，并证明该防御机制既能阻止以往已知的攻击，也能阻止我们提出的新型攻击。

总的来说，这项工作表明，在将多重防御集成到微架构中时，务必考虑 MDAV。我们提出的方法能够在此类集成的早期阶段检测并评估潜在的 MDAV。

## 参考文献


[1] “Alloy 文档”，%https://alloy.readthedocs.io/en/latest/，访问日期：2025年- 07-01。[2] “AMD SEV”，https://docs.enclaive.cloud/confidential-cloud/technology-in-depth/amd-sev/technology/fundamentals/features/reverse-map- table，访问日期：2025-07-01。[3] “gem5 Ruby”，https://www.gem5.org/documentation/general_docs/ ruby/，访问日期：2024-11-08。[4] “Mbed TLS”，https://github.com/Mbed-TLS/mbedtls，访问日期：2025年- 07- 01。[5] “在线补充材料”，https://anonymous.4open.science/r/ nvogng60g7f2xaqxcx00，访问日期：2025-07-01。[6] “Rosette”，%https://emina.github.io/rosette/，访问日期：2025-07-01。[7] “YAML 格式”，https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started，访问日期：2025-07-01。[8] P. Aimoniotis、A. B. Kvalsvik、X. Chen、M. Själander 和 S. Kaxiras，“Recon：高效检测、管理和利用非推测性信息泄漏”，载于《第56届IEEE/ACM微体系结构年度国际研讨会论文集》，2023年，第828–842页。[9] S. Ainsworth，“Ghostminion：用于 Spectre 缓解的严格性排序缓存系统”，载于《MICRO-54：第54届IEEE/ACM微体系结构年度国际研讨会》，2021年，第592–606页。[10] S. Ainsworth 和 T. M. Jones，“Muontrap：通过捕获推测状态防止跨域 Spectre 类攻击”，载于《2020 ACM/IEEE 第47届国际计算机体系结构研讨会（ISCA）》。IEEE，2020年，第132–144页。[11] N. A. Anagnostopoulos、S. Katzenbeisser、J. Chandy 和 F. Tehra- nipoor，“基于 DRAM 的安全原语概述”，《密码学》，第2卷，第2期，第7页，2018年。[12] R. Bahmani、F. Brasser、G. Dessouky、P. Jauernig、M. Klimmek、A.- R. Sadeghi 和 E. Stapf，“{CURE}：具有{可定制} 弹性隔离区的安全架构”，载于《第30届USENIX安全研讨会（USENIX Security 21）》，2021年，第1073–1090页。[13] P. Borkar、C. Chen、M. Rostami、N. Singh、R. Kande、A.-R. Sadeghi、C. Rebeiro 和 J. Rajendran，“{WhisperFuzz}：用于检测和定位处理器时序漏洞的{白盒} 模糊测试”，载于《第33届USENIX安全研讨会（USENIX Security 24）》，2024年，第5377–5394页。[14] F. Bostancı、O. Canpolat、A. Olgun、 ̇I. E. Yüksel、K. Kanellopou- los、M. Sadrosadati、A. G. Ya ̆glıkçı 和 O. Mutlu，“理解和缓解 Rowhammer 防御机制引入的侧信道与隐蔽信道漏洞”，arXiv 预印本 arXiv:2503.17891，2025年。[15] S. Briongos、P. Malagón、J. M. Moya 和 T. Eisenbarth，“Reload+refresh：滥用缓存替换策略实施隐蔽缓存攻击”，载于《第29届 {USENIX} 安全研讨会（{USENIX} Security 20）》，2020年，第1967–1984页。[16] K. Ceesay-Seitz、F. Solt 和 K. Razavi，“$\mu$cfi：微架构控制流完整性的形式化验证”，载于《2024年ACM SIGSAC计算机与通信安全会议论文集》，2024年，第213–227页。[17] C. Chen、R. Kande、N. Nguyen、F. Andersen、A. Tyagi、A.-R. Sadeghi 和 J. Rajendran，“{HyPFuzz}：{形式化辅助的} 处理器模糊测试”，载于《第32届USENIX安全研讨会（USENIX Security 23）》，2023年，第1361–1378页。[18] Y. Chen、L. Pei 和 T. E. Carlson，“通过硬件预取器泄漏控制流信息”，arXiv 预印本 arXiv:2109.00474，2021年。[19] R. Choudhary、J. Yu、C. Fletcher 和 A. Morrison，“推测隐私跟踪（SPT）：在不损害隐私的情况下从推测执行中泄漏信息”，载于《MICRO-54：第54届IEEE/ACM微体系结构年度国际研讨会》，2021年，第607–622页。

[20] M. H. I. Chowdhuryy 和 F. Yao，《Ivleague：利用动态完整性树的隔离域实现抗侧信道攻击的安全架构》，载于2024年第57届IEEE/ACM国际微体系结构研讨会（MICRO）。IEEE，2024年，第1153–1168页。[21] M. H. I. Chowdhuryy、Z. Zhang 和 F. Yao，《Powspectre：利用基于TSX的重放强化推测执行攻击》，载于第19届ACM亚洲计算机与通信安全会议论文集，2024年，第498–511页。[22] A. de Faveri Tron、R. Isemann、H. Ragab、C. Giuffrida、K. von Gleis-senthall 和 H. Bos，《Phantom trails：瞬态数据泄漏的实用硅前发现》，载于USENIX安全会议，2025年。[23] G. Dessouky、T. Frassetto 和 A.-R. Sadeghi，《Hybcache：面向可信执行环境的混合型抗侧信道缓存》，载于第29届{USENIX} 安全研讨会（{USENIX} 安全会议20），2020年，第451–468页。[24] G. Dessouky、A. Gruler、P. Mahmoody、A.-R. Sadeghi 和 E. Stapf，《Chunked-cache：面向安全架构的按需、可扩展缓存隔离》，arXiv预印本 arXiv:2110.08139，2021年。[25] L. Deutschmann、A. Meza、D. Stoffel、W. Kunz 和 R. Kastner，《Fast-path：一种高效硬件安全验证的混合方法》。[26] L. Deutschmann、J. Müller、M. R. Fadiheh、D. Stoffel 和 W. Kunz，《面向数据无关硬件的可扩展形式化验证方法》，IEEE 集成电路与系统计算机辅助设计汇刊，第43卷，第9期，第2551–2564页，2024年。[27] S. Dinesh，《可扩展验证及其在硬件安全中的应用》，2025年。[28] B. Fu、L. Tenenbaum、D. Adler、A. Klein、A. Gogia、A. R. Alameldeen、M. Guarnieri、M. Silberstein、O. Oleksenko 和 G. Saileshwar，《Amulet：安全推测执行对策的自动化设计时测试》，载于第30届ACM编程语言与操作系统体系结构支持国际会议论文集，第2卷，2025年，第32–47页。[29] N. Gaudin、P. Cotret、G. Gogniat 和 V. Lapotre，《通过缓存锁定防御基于缓存时序攻击的细粒度动态分区》，载于2024年IEEE计算机学会超大规模集成电路年度研讨会（ISVLSI）。IEEE，2024年，第173–179页。[30] L. Giner、S. R. Neela 和 D. Gruss，《Cohere+ reload：重新启用针对AMD SEV-SNP的高分辨率缓存攻击》，载于入侵与恶意软件检测及漏洞评估国际会议。Springer，2025年，第191–212页。[31] L. Giner、S. Steinegger、A. Purnal、M. Eichlseder、T. Unterluggauer、S. Mangard 和 D. Gruss，《Scatter and split securely：击败缓存争用与占用攻击》，载于2023年IEEE安全与隐私研讨会（SP）。IEEE计算机学会，2022年，第1101–1115页。[32] D. Gruss、C. Maurice 和 S. Mangard，《Rowhammer.js：JavaScript中的远程软件诱发故障攻击》，载于入侵与恶意软件检测及漏洞评估国际会议。Springer，2016年，第300–321页。[33] D. Gruss、C. Maurice、K. Wagner 和 S. Mangard，《Flush+ flush：一种快速而隐蔽的缓存攻击》，载于入侵与恶意软件检测及漏洞评估国际会议。Springer，2016年，第279–299页。[34] D. Gruss、R. Spreitzer 和 S. Mangard，《缓存模板攻击：对包含式末级缓存的自动化攻击》，载于第24届 {USENIX} 安全研讨会（{USENIX} 安全会议15），2015年，第897–912页。[35] A. Harris、S. Wei、P. Sahu、P. Kumar、T. Austin 和 M. Tiwari，《Cyclone：通过循环干扰检测基于争用的缓存信息泄漏》，载于第52届IEEE/ACM微体系结构年度国际研讨会论文集，2019年，第57–72页。[36] Y. Hsiao、D. P. Mulligan、N. Nikoleris、G. Petri 和 C. Trippel，《从RTL综合硬件形式化模型，以高效验证内存模型实现》，载于MICRO-54：第54届IEEE/ACM微体系结构年度国际研讨会，2021年，第679–694页。[37] Y. Hsiao、N. Nikoleris、A. Khyzha、D. P. Mulligan、G. Petri、C. W. Fletcher 和 C. Trippel，《Rtl2m$\mu$path：多$\mu$路径综合及其在硬件安全验证中的应用》，载于2024年第57届IEEE/ACM国际微体系结构研讨会（MICRO）。IEEE，2024年，第507–524页。[38] J. Juffinger、F. Rauscher、G. La Manna 和 D. Gruss，《Secret spilling drive：通过SSD争用泄漏用户行为》，载于2024年网络与分布式系统安全研讨会，2025年。

[39] J. Kaur 和 S. Das，《Rspp：用于缓解跨核隐蔽通道攻击的受限静态伪分区》，ACM电子系统设计自动化汇刊，第29卷，22，2024 2，1第 – 页。[40] T. Kessous 和 N. Gilboa，《Prune+ plumtree：大规模查找驱逐集》，载于2024年IEEE安全与隐私研讨会（SP）。IEEE，2024年，第3754–3772页。[41] S. A. Khaliq、U. Ali 和 O. Khan，《针对PCIe连接的分布式嵌入式系统的基于时序的侧信道攻击及缓解方法》，载于2021年IEEE高性能极限计算会议（HPEC）。IEEE，2021年，第1–7页。[42] V. Kiriansky、I. Lebedev、S. Amarasinghe、S. Devadas 和 J. Emer，《Dawg：推测执行处理器中针对缓存时序攻击的防御机制》，载于2018年第51届IEEE/ACM微体系结构年度国际研讨会（MICRO）。IEEE，2018年，第974–987页。[43] P. Kocher、J. Horn、A. Fogh、D. Genkin、D. Gruss、W. Haas、M. Hamburg、M. Lipp、S. Mangard、T. Prescher 等，《Spectre攻击：利用推测执行》，载于2019年IEEE安全与隐私研讨会（SP）。IEEE，2019年，第1–19页。[44] E. M. Koruyeh、K. N. Khasawneh、C. Song 和 N. Abu-Ghazaleh，《Spectre再现！利用返回栈缓冲区的推测执行攻击》，载于第12届 {USENIX} 攻击性技术研讨会（{WOOT} *18*），2018年。[45] A. B. Kvalsvik、P. Aimoniotis、S. Kaxiras 和 M. Själander，《Doppelganger加载：一种安全且复杂度效益高的安全推测方案优化》，载于第50届国际计算机体系结构研讨会论文集，2023年，第1–13页。[46] S. Lau、T. Bourgeat、C. Pit-Claudel 和 A. Chlipala，《硬件飞地强时序隔离的规范与验证》，载于2024年ACM SIGSAC计算机与通信安全会议论文集，2024年，第1121–1135页。[47] D. Lee、D. Kohlbrenner、S. Shinde、D. Song 和 K. Asanovi´c，《Keystone：用于构建可信执行环境架构的开放框架》，arXiv预印本 arXiv:1907.10119，2019年。[48] L. Li、H. Yavarzadeh 和 D. Tullsen，《Indirector：{高精度}分支目标注入攻击，利用间接分支预测器》，载于第33届USENIX安全会议（USENIX安全会议24），2024年，第2137–2154页。[49] F. Liu、Y. Yarom、Q. Ge、G. Heiser 和 R. B. Lee，《末级缓存侧信道攻击具有实用性》，载于2015年IEEE安全与隐私研讨会。IEEE，2015年，第605–622页。[50] K. Loughlin、I. Neal、J. Ma、E. Tsai、O. Weisse、S. Narayanasamy 和 B. Kasikci，《Dolma：以瞬态不可观测性原则保障推测执行安全》，载于第30届USENIX安全研讨会（USENIX安全会议21），2021年，第1397–1414页。[51] J. Lowe-Power、A. M. Ahmad、A. Akram、M. Alian、R. Amslinger、M. Andreozzi、A. Armejach、N. Asmussen、B. Beckmann、S. Bharadwaj 等，《gem5模拟器：版本20.0+》，arXiv预印本 arXiv:2007.03152，2020年。[52] M. Luo、W. Xiong、G. Lee、Y. Li、X. Yang、A. Zhang、Y. Tian、H.-H. S. Lee 和 G. E. Suh，《Autocat：利用强化学习自动探索缓存时序攻击》，载于2023年IEEE国际高性能计算机体系结构研讨会（HPCA）。IEEE，2023年，第317–332页。[53] D. Lustig、G. Sethi、M. Martonosi 和 A. Bhattacharjee，《Coatcheck：验证硬件-操作系统接口处的内存排序》，ACM SIGPLAN Notices，第51卷，第4期，第233–247页，2016年。[54] L. Maar、J. Juffinger、T. Steinbauer、D. Gruss 和 S. Mangard，《Kernelsnitch：针对内核数据结构的侧信道攻击》，载于2025年网络与分布式系统安全研讨会：NDSS 2025，2025年。[55] Y. A. Manerkar、D. Lustig、M. Martonosi 和 A. Gupta，《Pipeproof：针对微架构规范的自动化内存一致性证明》，载于2018年第51届IEEE/ACM微体系结构年度国际研讨会（MICRO）。IEEE，2018年，第788–801页。[56] Y. A. Manerkar、D. Lustig、M. Martonosi 和 M. Pellauer，《Rtlcheck：验证RTL设计的内存一致性》，载于第50届IEEE/ACM国际微体系结构研讨会论文集，2017年，第463–476页。[57] Y. A. Manerkar、D. Lustig、M. Pellauer 和 M. Martonosi，《Ccicheck：利用 $\mu$hb图验证一致性-连贯性接口》，载于第48届国际微体系结构研讨会论文集，2015年，第26–37页。[58] B. Morgan、G. Horowitz、S. O'Connell、S. van Schaik、C. Chuengsatiansup、D. Genkin、O. Maennel、P. Montague、E. Ronen 和 Y. Yarom，"Slice+ slice 宝贝：眨眼之间生成末级缓存驱逐集合"，载于 2025 年 IEEE 安全与隐私研讨会（SP）。IEEE，2025年，第 3479–3496 页。[59] N. Mosier、H. Lachnitt、H. Nemati 和 C. Trippel，"面向安全性的公理化硬件—软件契约"，载于第49届年度国际计算机体系结构研讨会论文集，2022年，第 72–86 页。[60] K. Murdock、D. Oswald、F. D. Garcia、J. Van Bulck、D. Gruss 和 F. Piessens，"Plundervolt：针对 Intel SGX 的基于软件的故障注入攻击"，载于 2020 年 IEEE 安全与隐私研讨会（SP）。IEEE，2020年，第 1466–1482 页。[61] O. Mutlu 和 J. S. Kim，"Rowhammer：回顾"，《IEEE 集成电路与系统计算机辅助设计汇刊》，第39卷，第8期，第 1555–1571 页，2019年。[62] S. O'Connell、L. A. Sour、R. Magen、D. Genkin、Y. Oren、H. Shacham 和 Y. Yarom，"Pixel thief：利用 Firefox 和 Chrome 中的 {SVG} 滤镜信息泄漏"，载于第33届 USENIX 安全研讨会（USENIX Security 24），2024年，第 3331–3348 页。[63] D. Ojha 和 S. Dwarkadas，"Timecache：利用时间消除软件共享时的缓存侧信道"，载于 2021 年 ACM/IEEE 第48届年度国际计算机体系结构研讨会（ISCA）。IEEE，2021年，第 375–387 页。[64] H. Omar 和 O. Khan，"Ironhide：一种能够高效缓解交互式应用中微体系结构状态攻击的安全多核处理器"，载于 2020 年 IEEE 国际高性能计算机体系结构研讨会（HPCA）。IEEE，2020年，第 111–122 页。[65] A. Pashrashid、A. Hajiabadi 和 T. E. Carlson，"Hidfix：通过隐藏回滚高效缓解基于缓存的 Spectre 攻击"，载于 2023 年 IEEE/ACM 国际计算机辅助设计会议（ICCAD）。IEEE，2023年，第 1–9 页。[66] P. Pessl、D. Gruss、C. Maurice、M. Schwarz 和 S. Mangard，"{DRAMA}：利用 {DRAM} 寻址发起跨 CPU 攻击 "，载于第 25 届 {USENIX} 安全研讨会（{USENIX}安全 16），2016年，第 565–581 页。[67] H. Ponce-de León 和 J. Kinder，"猫与 Spectre：一种对推测执行攻击建模的公理化方法"，载于 2022 年 IEEE 安全与隐私研讨会（SP）。IEEE，2022年，第 235–248 页。[68] A. Purnal、F. Turan 和 I. Verbauwhede，"Prime+ scope：克服高精度缓存争用攻击中的观测者效应。"[69] M. K. Qureshi，"Ceaser：通过加密地址和重映射缓解基于冲突的缓存攻击"，载于 2018 年第51届 IEEE/ACM 国际微体系结构研讨会（MICRO）。IEEE，2018年，第 775–787 页。[70] H. Ragab、A. Mambretti、A. Kurmus 和 C. Giuffrida，"Ghostrace：利用并缓解推测性竞态条件"，载于 USENIX 安全会议，2024年。[71] K. Ramkrishnan、S. McCamant、P. C. Yew 和 A. Zhai，"首次未命中：针对共享内存缓存侧信道的低开销缓解方法"，载于第49届国际并行处理会议（ICPP），2020年，第 1–11 页。[72] K. Ramkrishnan、S. McCamant、A. Zhai 和 P.-C. Yew，"基于非融合的相干缓存随机化：利用跨域访问"，载于第19届 ACM 亚洲计算机与通信安全会议论文集，2024年，第 186–202 页。[73] K. Ramkrishnan、A. Zhai、S. McCamant 和 P. C. Yew，"随机化缓存的新型攻击与防御"，arXiv 预印本 arXiv:1909 12302，2019年。。[74] F. Rauscher、C. Fiedler、A. Kogler 和 D. Gruss，"对新型及现有缓存侧信道的系统评估"，载于网络与分布式系统安全研讨会（NDSS）2025，2025年。[75] C. Rojas、H. Morales 和 E. Roa，"针对基于 RISC-V 的处理器的低成本漏洞搜寻验证方法"，载于 2021 年 IEEE 国际电路与系统研讨会（ISCAS）。IEEE，2021年，第 1–5 页。[76] S. Rüegge、J. Wikner 和 K. Razavi，"分支特权注入：利用分支预测器竞态条件破坏 Spectre v2 硬件缓解措施。"[77] G. Saileshwar、C. W. Fletcher 和 M. Qureshi，"Streamline：通过实现异步串通发起快速、无刷新缓存隐蔽信道攻击"，载于第26届 ACM 编程语言与操作系统体系结构支持国际会议论文集，2021年，第 1077–1090 页。[78] G. Saileshwar、S. Kariyappa 和 M. Qureshi，"定制缓存飞地：通过灵活的集合分区实现精细且可扩展的缓存侧信道隔离。"

[79] G. Saileshwar 和 M. Qureshi，“{MIRAGE}：采用实用的全相联设计缓解基于冲突的缓存攻击”，载于第30届 {USENIX} 安全研讨会（{USENIX} 安全 21），2021年。[80] C. Sakalis、S. Kaxiras、A. Ros、A. Jimborean 和 M. Själander，“通过选择性延迟和值预测实现高效的不可见推测执行”，载于 2019 年 ACM/IEEE 第46届年度国际计算机体系结构研讨会（ISCA）。IEEE，2019年，第 723–735 页。[81] P. Saravanan、R. Parthasarathy、B. Sona 和 A. Nihar Ahmed Fathima，“使用 JasperGold 对 Ascon 密码进行形式化验证”，载于 2024 年智能电子与通信系统国际会议（ISENSE）。IEEE，2024年，第 1–6 页。[82] D. Skarlatos、Z. N. Zhao、R. Paccagnella、C. W. Fletcher 和 J. Torrellas，“Jamais vu：挫败微体系结构重放攻击”，载于第26届 ACM 编程语言与操作系统体系结构支持国际会议论文集，2021年，第 1061–1076 页。[83] F. Solt、K. Ceesay-Seitz 和 K. Razavi，“Cascade：通过复杂的程序生成进行 {CPU} 模糊测试”，载于第33届 USENIX 安全研讨会（USENIX Security 24），2024年，第 5341–5358 页。[84] F. Solt、B. Gras 和 K. Razavi，“{CellIFT}：利用单元实现 {RTL} 中可扩展且精确的动态信息流跟踪”，载于第31届 USENIX 安全研讨会（USENIX Security 22），2022年，第 2549–2566 页。[85] M. Tan、J. Wan、Z. Zhou 和 Z. Li，“隐形探针：利用 PCIe 拥塞侧信道发起时序攻击”，载于 2021 年 IEEE 安全与隐私研讨会（SP）。IEEE，2021年，第 322–338 页。[86] Q. Tan、Y. Yang、T. Bourgeat、S. Malik 和 M. Yan，“利用契约影子逻辑验证安全推测执行的 RTL”，载于第30届 ACM 编程语言与操作系统体系结构支持国际会议论文集，第1卷，2025年，第 970–986 页。[87] D. Townley、K. Arıkan、Y. D. Liu、D. Ponomarev 和 O. Ergin，“可组合缓存小块：保护飞地免受缓存 {侧信道}攻击”，载于第31届 USENIX 安全研讨会（USENIX Security 22），2022年，第 2839–2856 页。[88] C. Trippel、D. Lustig 和 M. Martonosi，“Checkmate：硬件漏洞利用和安全性测试用例的自动化综合”，载于 2018 年第51届 IEEE/ACM 国际微体系结构研讨会（MICRO）。IEEE，2018年，第 947–960 页。[89] C. Trippel、Y. A. Manerkar、D. Lustig、M. Pellauer 和 M. Martonosi，“Tricheck：在软件、硬件和指令集体系结构的交汇处验证内存模型”，《ACM SIGPLAN 通讯》，第52卷，第4期，第 119–133 页，2017年。[90] T. Unterluggauer、A. Harris、S. Constable、F. Liu 和 C. Rozas，“变色龙缓存：通过随机替换近似全相联缓存，以防止基于争用的缓存攻击”，载于 2022 年 IEEE 安全与隐私执行环境设计国际研讨会（SEED）。IEEE，2022年，第 13–24 页。[91] A. Vakili 和 N. A. Day，“在 Alloy 中进行时序逻辑模型检测”，载于抽象状态机、Alloy、B、VDM 与 Z 国际会议。Springer，2012年，第 150–163 页。[92] J. Wan、Y. Bi、Z. Zhou 和 Z. Li，“Volcano：利用网格互连发起无状态缓存侧信道攻击”，arXiv 预印本 arXiv:2103.04533，2021年。[93] J. Wickerson、M. Batty、T. Sorensen 和 G. A. Constantinides，“自动比较内存一致性模型”，载于第44届 ACM SIGPLAN 编程语言原理研讨会论文集，2017年，第 190–204 页。[94] S. Wiebing、A. de Faveri Tron、H. Bos 和 C. Giuffrida，“{InSpectre} gadget：检查跨特权级 Spectre v2 的残余攻击面”，载于第33届 USENIX 安全研讨会（USENIX Security 24），2024年，第 577–594 页。[95] J. Wikner 和 K. Razavi，“突破屏障：屏障后 Spectre 攻击”，载于 2025 年 IEEE 安全与隐私研讨会（SP）。IEEE，2025年，第 3516–3533 页。[96] J. Xu、Y. Zhou、X. Zhang、Y. Li、Q. Tan、Y. Zhang、Y. Zhou、R. Chang 和 W. Shen，“Dejavuzz：结合动态可交换内存、差分信息流跟踪辅助处理器模糊测试，揭示瞬态执行漏洞”，arXiv 预印本 arXiv:2504.20934，2025年。[97] M. Yan、J. Choi、D. Skarlatos、A. Morrison、C. Fletcher 和 J. Torrellas，“Invisispec：让推测执行在缓存层次结构中不可见”，载于 2018 年第51届 IEEE/ACM 国际微体系结构研讨会（MICRO）。IEEE，2018年，第 428–441 页。

[98] M. Yan、J.-Y. Wen、C. W. Fletcher 和 J. Torrellas，“Secdir：抵御目录侧信道攻击的安全目录”，载于第*46*届国际计算机体系结构研讨会论文集，2019年，第332–345页。[99] Y. Yang、T. Bourgeat、S. Lau 和 M. Yan，“Pensieve：用于安全评估的微架构建模”，载于第*50*届国际计算机体系结构年度研讨会论文集，2023年，第1–15页。[100] Y. Yarom 和 K. Falkner，“Flush+ reload：一种高分辨率、低噪声的 L3 缓存侧信道攻击”，载于第*23*届 {*USENIX*} 安全研讨会（{*USENIX*} 安全性*14*），2014年，第719–732页。[101] L. Yin、H. Wang、Y. Lyu、C. Hu 和 D. Wang，“Vericache：针对侧信道安全飞地的形式化验证细粒度分区缓存”，*IEEE*可信与安全计算汇刊，2025年。[102] J. Yu、N. Mantri、J. Torrellas、A. Morrison 和 C. W. Fletcher，“推测性数据无关执行：利用安全预测实现安全且高效的推测执行”，载于*2020*年*ACM/IEEE*第*47*届计算机体系结构国际研讨会（*ISCA*）。IEEE，2020年，第707–720页。[103] J. Yu、M. Yan、A. Khyzha、A. Morrison、J. Torrellas 和 C. W. Fletcher，“推测性污点跟踪（stt）：针对推测访问数据的全面保护”，载于第*52*届*IEEE/ACM*微体系结构年度国际研讨会论文集，2019年，第954–968页。[104] M. Zaheri、Y. Oren 和 R. Curtmola，“基于争用的侧信道支持更快速、更隐蔽的浏览历史嗅探”，*IEEE*可信与安全计算汇刊，2025年。[105] A. Q. Zhang、A. Goens、N. Oswald、T. Grosser、D. Sorin 和 V. Nagarajan，“Pipegen：针对给定内存一致性模型，将单核流水线自动转换为多核流水线”，载于*2024*年国际并行体系结构与编译技术会议论文集，2024年，第1–13页。[106] J. Zhang、C. Chen、J. Cui 和 K. Li，“CPU微体系结构中的时序侧信道攻击与对策”，*ACM*计算调查，第56卷，第7期，第1–40页，2024年。[107] Y. Zhao 、 G. Qu 、 Q. Zhang 、 Y. Li 、 Z. Li 和 J. He，“ 利用有界模型检测进行硬件信息安全的静态门级信息流分析”，载于*2024*年*IEEE*第*42*届*VLSI*测试研讨会（*VTS*）。IEEE，2024年，第1–7页。[108] Z. N. Zhao、A. Morrison、C. W. Fletcher 和 J. Torrellas，“ Untangle：设计低泄漏、高性能动态分区方案的原则性框架”，载于第*28*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第*3*卷，2023年，第771–788页。[109] Z. N. Zhao、A. Morrison、C. W. Fletcher 和 J. Torrellas，“现代公有云中最后一级缓存侧信道攻击是可行的”，载于第*29*届*ACM*编程语言与操作系统体系结构支持国际会议论文集，第*2*卷，2024年，第582–600页。