实现虚拟化机器学习工作负载执行的可验证输出溯源

Prach Chantasantitam*、Adam Caulfield*、Sujaya Maiyya*、N. Asokan*、†*
滑铁卢大学，†瑞典皇家理工学院{*pchantas*、*acaulfield*、*smaiyya*}
@uwaterloo.ca、*asokan@acm.org*

arXiv:2611.28344v2 [cs.CR] 25 Nov 2026

摘要——**机器学习（ML）属性证明使模型提供方能够向验证者证明其模型和数据集的属性，而可信执行环境（TEE）已成为实现这一目标最实用的机制。然而，现有基于TEE的系统只能证明TEE初始配置的静态快照，使运行时行为无法审计。执行证明（PoX）架构通过要求证明者的信任根同时证明以下两点来弥合这一差距：(1) 经证明的操作确实已执行；(2) 所声明的输出仅受该操作影响。不过，现有的PoX设计面向微控制器，并依赖服务器级TEE不具备的硬件特性，因此无法直接应用于机器学习工作负载。**

我们提出了 EVOLVE，这是一个面向服务器级TEE的基于软件的PoX框架。EVOLVE 通过调控操作写入和控制流，跟踪所请求的机器学习操作是否在未发生违规的情况下执行。它将执行状态与操作输入和输出的度量值绑定，使验证者能够通过机器学习属性证明评估运行时行为。我们在 Intel TDX 上将 EVOLVE 实现为其虚拟固件中的 Rust 模块，并基于在 F* 中经过形式化验证的原语构建策略检查。评估结果表明，EVOLVE 能够在未经修改的硬件上适用于各种机器学习应用，并为其策略检查提供形式化保证。EVOLVE 产生的每请求运行时开销平均介于 1.07×（决策树）和 2.13×（多层感知机）之间。

1. 引言

机器学习（ML）属性证明 [1]–[3] 使模型提供方能够向验证者证明其机器学习资产（模型、数据集）的属性。尽管零知识证明（ZKP）等替代方案会产生巨大的开销 [4]–[9]（例如，逻辑回归训练时间的 518–4200× 倍 [4]），且学习证明方案可能被攻破 [10], [11], ，基于可信执行环境（TEE）的方法 [1], [2], [12]则能以合理的性能提供强大的安全保证。

基于可信执行环境的机器学习属性证明 [1], [2], [12], [13]面向服务器级机器，构建于英特尔软件防护扩展（SGX） [14]、Intel Trust Domain eXten- sions (TDX) [15], 或 AMD安全加密虚拟化–安全嵌套分页（SEV-SNP） [16]之上。这些技术能够防御保留更高权限并控制可信执行环境对资源的访问和执行过程的不可信主机操作系统（OS）或虚拟机管理器（VMM）。它们提供的证明支持可生成可信执行环境配置的认证证据，验证者在远程证明过程中会向基于可信执行环境的证明者请求这些证据。对于机器学习属性证明，在可信执行环境内运行的代码会对某项操作的输入和输出进行哈希运算，并将这些哈希值纳入证明证据，从而将它们与可信执行环境的初始配置绑定起来 [1], [2], [12], [13]。

由于此快照仅记录了 TEE 的初始配置和辅助数据，因此无法检测到在 TEE 内改变运行时行为的攻击。此类攻击绝非臆测：正如近期披露所示，广泛使用的机器学习库中普遍存在运行时漏洞（例如，PyTorch [17], Tensor-Flow [18], 以及基于 pickle 的加载 [19]–[21]） [22], [23]。对手可能在测量后破坏或替换模型权重 [24], [25], ，也可能操纵执行路径，使输出偏离经证明的模型所产生的结果 [26], [27]。另一个例子是破坏模型路由，该机制用于在运行时从共置模型中进行选择；近期研究表明，无须修改底层模型或代码，即可操纵路由决策 [28]–[31]。由于这些系统中的证明证据本质上是静态的，因此仅凭证明证据无法揭示 TEE 内发生的运行时偏差。

执行证明（PoX）架构 [32]–[36]将远程证明扩展到了运行时属性：证明者中的信任根还必须证明 (1) 已执行经证明的操作；(2) 所报告的输出仅受该操作最近一次执行的影响。PoX 主要在嵌入式系统中得到研究，依赖于面向微控制器的轻量级硬件扩展 [32], [33] 或微控制器专属功能 [34], [35]。

由于以下三个原因，这些设计无法直接迁移到服务器级机器上。(1) PoX 需要一种可信机制，用于检测是否有除操作本身之外的任何代码写入了该操作的输出。此前现有方法通过定制硬件或 TEE 的隔离边界来实现这一点。然而，在基于机密虚拟机（CVM）的 TEE 中，该边界包围整个虚拟机，因此要利用它隔离操作，就需要为每个操作配置一个机密虚拟机（例如， [13], [37] 中的做法）。(2) PoX 信任根必须是唯一能够更新执行证明数据的组件，这就需要硬件支持，以保护元数据免受经证明软件的篡改。服务器级TEE缺乏可信执行环境内部访问控制，而依赖客户操作系统并不安全，因为它本身可能已遭受 PoX 旨在检测的运行时攻击。(3) 近期研究提出

通过在机密虚拟机内 构建隔离区来弥补部分隔离能力的不足，但这依赖于特定平台的特征，例如 AMD SEV-SNP 中的虚拟机特权级（VMPL） [13], [38]–[42],、Intel TDX 中的管理程序保护密钥（PKS） [43],、Arm 机密计算架构（CCA）的粒度保护表 [44], , 或 TEE 外部的软件 [45], [46]。这些设计隔离的也是整个隔离区，而非单个操作，并且只对加载到各隔离区中的代码进行证明，因此无法证明某个操作确实已执行，也无法证明其输出仅反映该操作最近一次执行的结果（第 7 节将进一步讨论）。

在机密虚拟机中执行的机器学习工作负载尤其使 PoX 面临挑战，因为它们会使用共享库代码，例如训练和推理框架。因此，PoX 机制必须能够提供操作级执行证明，同时允许操作共享库。

为应对这些挑战，我们提出 EVOLVE，这是一种软件框架，旨在为服务器级TEE上的机器学习操作实现执行证明。EVOLVE 由插桩器和运行时监控器组成，通过以下方式应对上述挑战：

（1）插桩器将载荷（即机密虚拟机内的机器学习应用，包括其库代码）中的每次写入和间接调用都路由至监控器，由监控器检查调用方是否属于所请求的操作。（2）监控器运行于机密虚拟机的客户机虚拟固件中，并阻止所有载荷对执行证明数据的写入，因此只有监控器能够记录所请求的操作是否在未发生违规的情况下执行。（3）EVOLVE 仅依赖软件插桩、虚拟固件和标准远程证明，而不依赖特定于平台的特性。

作为概念验证，我们在 Intel TDX 上实现了一个原型，其中插桩器是一个 LLVM 处理遍，监控器则是 TD-Shim 中以 Rust 编写的一层 [47] ，其策略检查基于经过 F* 形式化验证的原语 [48]。总之，我们作出以下贡献：
(1) 我们指出了机密虚拟机中 PoX 面临的挑战，并提出 EVOLVE，这是首个面向机密虚拟机的基于软件的操作级执行证明框架。EVOLVE 对所有内存写入和间接调用进行中介，为远程验证者提供输出溯源证据（第 3 节）；(2) 我们将 EVOLVE 设计为两个模块：

(i) 一个 插桩模块，用于中介机器学习操作中的内存写入和目标动态定义的间接调用（第 4 节），以及

(ii) 对机密虚拟机客户机虚拟固件的一个 运行时监控器 扩展，用于维护执行证明数据，并使用经过形式化验证的原语来实施访问控制和准备 PoX 证据（第 5 节）；

（3）我们将 EVOLVE 实现为 TD-Shim 虚拟固件的 Rust+F* 扩展，并在 Intel TDX 上对其进行评估。我们展示了 EVOLVE 提供形式化保证，能够在不同 CVM 架构间灵活适用 ，并且易于集成到多个模型路由应用中。基于示例决策树（DT）和多层感知机

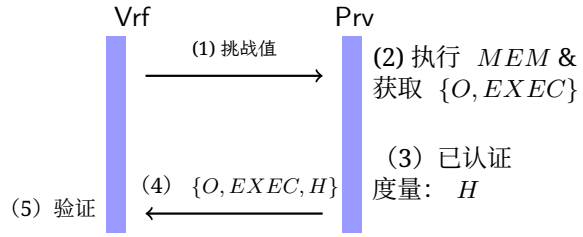

图 1：基于传统远程证明构建的执行证明（PoX）：证明者（Prv）向验证者（Vrf）证明其软件的完整性和执行属性。

（MLP）分类器，其平均每请求运行时开销分别为 DT 的 1.07×(±0.04) 和 MLP 的 2.13×(±0.81)（第 6 节）。

2. 背景

可信执行环境和证明。可信执行环境提供隔离的执行环境，防止其信任边界之外的软件直接访问受保护的数据。信任边界因架构而异：Intel SGX [14] 隔离用户空间飞地，而 Arm TrustZone [49] 隔离系统资源，包括受信任的操作系统、应用程序和外设。服务器级TEE还支持机密虚拟机，可将这种保护扩展到整个虚拟机（VM），例如 AMD SEV-SNP [16], 、Intel TDX [15],和 Arm Realms [50]。远程证明使证明者能够使用认证证据向验证者证明其身份和配置；认证证据通常是由信任根硬件生成的消息认证码（MAC）或数字签名，它将可信执行环境初始状态的度量值与运行时提供的辅助数据绑定起来。验证者检查这些证据，以确定可信执行环境是否处于预期状态。

在 Intel TDX 中，机密虚拟机被称为信任域（TD）。当不可信虚拟机监视器创建 TD 时，TDX 模块会对其组件进行度量。TD 请求 TDREPORT 以证明其状态。TDREPORT 包含一个 MAC，该 MAC 由中央处理器（CPU）针对 TD 映像、TDX 模块镜像、运行时度量寄存器（RTMR）中的运行时数据以及 REPORTDATA 计算得出。引用 enclave（QE）是一种专用的 Intel SGX enclave [51],，它验证 MAC，并使用配置认证密钥 [52]生成采用 SGX 风格的 QUOTE 签名。远程验证者检查 QUOTE，以确认预期的 TD 映像生成了其中包含的 RTMR 和 REPORTDATA。

执行证明。 现有基于 TEE 的机器学习属性证明框架以整个 TEE 为粒度进行证明（例如，SGX enclave [1] 或 TDX 机密虚拟机 [2] 的快照）。相比之下，PoX 架构 [32]–[36] 将远程证明扩展至单个操作的执行，使证明者（Prv）能够向验证者（Vrf）证明：(1) Prv 已安装指定操作；(2) Prv在请求时执行了该操作；(3) Prv 报告的任何输出都仅由该操作最近一次执行所修改。典型的 PoX 协议（图

1，改编自先前的工作 [33]）流程如下。Vrf 通过发送挑战值（*Chal*）来请求对预期应用软件（App）进行 PoX，挑战值通常是密码学随机数。随后，Prv 中的信任根（RoT）会对容纳 App 的内存区域（MEM）、App 执行的输出（O）、执行状态元数据（EXEC）以及*Chal*进行经认证的度量，并生成认证令牌（H），通常为消息认证码或数字签名。Prv 返回O、EXEC 和 H，由Vrf 检查这些内容，以判断执行和输出是否可信。

为体现执行完整性，PoX 要求根信任（RoT）独占访问 EXEC，能够监控 App 的进入与退出，并能够监控对 O 的访问；后两项决定了其如何更新 EXEC。以往工作通过可信硬件满足这些要求，例如专用硬件模块 [32]、[33] 或可信执行环境（TEE） [34], [35]；计算 H 还需要密钥，这意味着需要硬件支持的根信任。PoX 主要在嵌入式系统 [32]–[35]，中进行探索；对于高端系统，则依赖架构特定的能力（例如 AMD 中的延迟启动 64 [53]）和可信平台模块（TPM） [36]，其中也采用了类似的理念。

3. 问题陈述

3.1. 挑战

要为机器学习操作实现 PoX，需要将操作级执行证明从嵌入式系统扩展到机密虚拟机。现有的 PoX 方法和机密虚拟机隔离都无法直接适用。现有 PoX 设计能够证明单个操作的执行，但依赖机密虚拟机缺少或未统一提供的特征：用于监控 App 并维护 EXEC 的专用硬件 [32], [33]，或由可信软件在 App 运行前配置的硬件访问控制 [34], [35]。机密虚拟机隔离运行在服务器级硬件上，并将机密虚拟机划分为隔离分区，但依赖特定于某一平台的机制（SEV-SNP 上的虚拟机权限级别、Intel 上的保护密钥，或 Arm CCA 固件扩展），或者依赖可信执行环境之外的软件 [38]–[46], [54]。此外，隔离区不会记录生成给定输出的操作，因此机密虚拟机内部的代码可以在不被观察的情况下覆盖输出。

因此，只有在Vrf请求下加载、测量并启动 App 的 PoX 信任根必须运行在机密虚拟机内，并且只能依赖机密虚拟机架构所提供的机制。这正是该问题具有挑战性的原因：测量者与 App （即其所测量的软件）共享同一个机密虚拟机，而 VMM 不受信任，且没有任何硬件在二者之间强制实施访问控制。依赖某一供应商的机密虚拟机内隔离功能，只能在该供应商的平台上解决这一问题。因此，能够跨机密虚拟机平台运行的设计需要具备：(1) App 与测量者之间的机密虚拟机内隔离；(2) 能够查看 App 的进入和退出，以及对其输出的每次访问，并据此维护 EXEC；(3) 测量者对 EXEC 具有独占写入权限，并在 App 的整个运行时期间更新 EXEC；(4) 构造 PoX 证据，以向Vrf 证明 App 确实执行过，并说明其报告的输出是否可信。

3.2. 系统与对手模型

系统模型。我们的目标是具备机密虚拟机功能的服务器级处理器。证明者（Prv）是一个模型提供方，通过云服务提供商托管的机密虚拟机提供机器学习推理；验证者（Vrf）是发送推理请求并希望确认所请求模型生成了返回输出的客户端或审计者，例如确认请求没有被路由到功能较弱的较小模型。Prv 在由宿主机虚拟机监控器管理的机密虚拟机内运行机器学习负载；宿主机虚拟机监控器负责创建和配置机密虚拟机，并控制其输入/输出（I/O）访问（例如网络设备）。

CVM 使用虚拟固件，提供在 CVM 中启动客户机工作负载所需的最低限度功能。它通常支持裸机容器：直接在固件上或在简化内核之上运行的应用程序 [47], [55]。我们关注前一类中的机器学习工作负载，并将其称为载荷，同时在第 8 节讨论后一类的影响。我们假设载荷由多个操作组成，Vrf 可能会请求执行这些操作；每个操作都有自己的模型和代码，并且可能共享库代码。我们假设采用各商用 CVM 架构普遍具备的一组最低限度基线保护：数据页不可执行，代码页为只读，返回地址由影子栈保护，硬件或编译器支持提供粗粒度的前向边控制流完整性（CFI）。我们将在第 6.4 节讨论各 CVM 架构的具体特征。

对手模型。 对手 ($\mathcal{A}$dv) 会操纵载荷执行，从而使输入、操作或输出出现非预期行为，例如修改数据，或重定向控制流以覆盖另一操作的输出。 $\mathcal{A}$dv还控制宿主环境，包括 VMM 以及 CVM 之外的一切（例如其他虚拟机、存储和 I/O 外设）。我们考虑两类运行时攻击。控制流劫持 [56]–[58]会将执行重定向到正常执行时不会运行的代码（例如调用另一操作来写入输出）。直接数据操纵（DDM） [59], [60] 会破坏指针，以便在授权操作范围之外写入数据（例如覆盖输入、输出或执行状态）。

侧信道攻击 [61]–[65], 拒绝服务攻击和物理攻击 [66] 不在讨论范围之内。我们信任机密虚拟机平台的硬件和固件（例如 Intel TDX 上的 TDX 模块）能够将各机密虚拟机彼此隔离，并与宿主机隔离，同时提供硬件支持的度量和证明 [51], [67]。我们假设机密虚拟机内部没有隔离，因此一旦 $\mathcal{A}$dv 在机密虚拟机中执行代码（例如利用负载漏洞），就能操纵任何未受保护的代码或数据。由于受 $\mathcal{A}$dv控制的 VMM 会创建机密虚拟机，对机密虚拟机的信任只能来自远程证明。

3.3. EVOLVE 概述

EVOLVE 解决了前文讨论的挑战，并支持对部署在机密虚拟机中的机器学习工作负载进行 PoX。它包含两个主要模块：软件插桩器 和运行时监控器。我们使用 Intel TDX 术语来描述

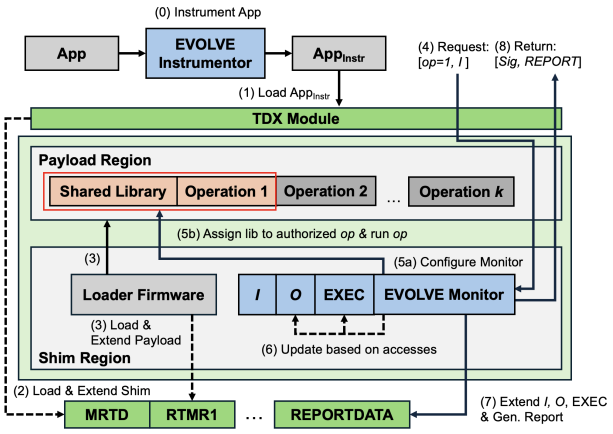

图 2：EVOLVE 概览。包含多个操作的机器学习应用经过插桩后，被加载并度量到启用 EVOLVE 的 TD 中。收到操作执行请求后，EVOLVE 监控器会将共享库分配给相应操作，并跟踪对输出的访问。

EVOLVE 的设计，因为其原型基于 Intel TDX 实现（见第 6 节）。因此，机密虚拟机即一个 TD，其虚拟固件为 TD-Shim [47]，，而 TDX 模块和引用 enclave 支持证明；第 6.4 节介绍了 AMD SEV-SNP 和 Arm CCA 上的对应组件。部署之前，EVOLVE 插桩器会处理应用源代码（步骤 0），插桩内存写入和间接调用，以生成插桩二进制文件应用$_{\text{Instr}}$，并将其加载到 TD 中（步骤 1）。

EVOLVE 将 TD 内存划分为垫片区域 和负载区域，每个区域又划分为多个具有名称的区段；这些区段的固定地址范围由链接器在构建时设定。

垫片区域包含虚拟固件（例如 TD-shim）、EVOLVE 监控器代码，以及用于输入（I）、输出（O）和 EXEC 的关键区段。经 EVOLVE 扩展的垫片区域无需信任主机，也不会增加对主机的信任；由于该区域会被度量并纳入 TD 的证明结果，因此Vrf 可见该区域，这与在 SEV-SNP 上将监控器置于特权客户机层类似 [38]。在负载区域中，每个操作都占据自己的操作区段（op）。对于机器学习，每个 op都映射到一个模型（例如，将 op_1 映射到模型$_1$）。多个操作共享的代码（例如机器学习推理库）只会放置一次，位于一个共享区段（sh）中；该区段在运行时绑定到授权操作之前不会分配给任何操作。

启动时，TDX 模块加载并测量垫片区域（步骤 2），加载器固件加载并测量插桩有效载荷（步骤 3），将载荷绑定到 TD 的初始配置。随后，用户发送请求 $[op, I]$，其中包含操作选择和输入（步骤 4）。监控器配置自身，并将输入存储在 I 中（步骤 5a），然后将共享区段绑定到所请求的 op，并对其进行调度（步骤 5b）。执行期间，每次插桩内存写入和间接调用都经过监控器，由监控器更新 O 和 EXEC（步骤 6）；来自共享区段的访问会归属于已绑定的操作。若所有访问均获授权，EXEC 将保持为 1 若发生任何违规，则将其设为 0。最后，监控器将 I、 O 和 EXEC 绑定到 TDX 模块生成的硬件证明报告（步骤 7），并将其作为证明返回给用户，以证明该操作执行期间未发生违规（步骤 8）。

EVOLVE 通过两种方式应对 PoX 在嵌入式系统之外面临的挑战。预处理和插桩通过定义垫片区域和负载区域，并确保对关键区段的独占访问，实现机密虚拟机内隔离（第 4 节将进一步讨论）。监控器检查内存写入和间接调用，并更新 EXEC，以证明所请求操作的输出溯源（第 5 节将进一步讨论）。

4. 插桩模块

本节介绍 EVOLVE 插桩器中使用的组件。我们首先讨论相关的 LLVM 特征，然后介绍插桩过程、外部函数接口（FFI）层以及链接。

4.1. LLVM 编译器

LLVM [68] 是一种编译器基础设施，它以类型化的低级中间表示（IR）为核心，并由不同源语言和目标架构共享。源程序会被转换为这种 IR；在 LLVM 后端将其降级为特定于目标的机器代码之前，可以对其进行转换或优化。由于所有前端都会生成这种 IR，且所有后端都会使用它，针对这种 IR 编写的转换无论源语言或目标架构如何，都能适用。这一特性使 IR 级插桩成为一种成熟的软件安全策略，可用于实现内存安全 [69]–[72], 和软件故障隔离（SFI） [73]–[76]。

LLVM IR 将程序表示为由基本块组成、采用静态单赋值（SSA）形式的函数 [77]。有三条指令与我们的设计相关：store 指令，它是内存写入的主要形式；alloca，它为局部变量分配栈空间；以及 call 指令，它通过静态符号直接调用函数，或通过函数指针间接调用函数。LLVM 的通道基础设施允许将自定义转换插入此流水线。通道是一种独立的单元，用于检查或改写中间表示；它通过一个带有 run 方法的类实现，并由 LLVM 的通道管理器调度 [78]。通道可以打包为插件，因此无需将其编译进 LLVM 本身。插件会在优化流水线的特定扩展点注册，这些扩展点决定了自定义通道相对于其他优化的运行时机。

4.2. 插桩过程

EVOLVE 插桩器在中间表示阶段、汇编生成之前，向 LLVM 流水线添加一个处理过程。如图 3 所示，常规流水线会将源代码编译为 LLVM 中间表示，生成汇编，并将结果链接为最终二进制文件。该处理过程在这两步之间重写载荷的内存写入和间接调用。此处理过程处理内存写入和间接调用的中间表示。

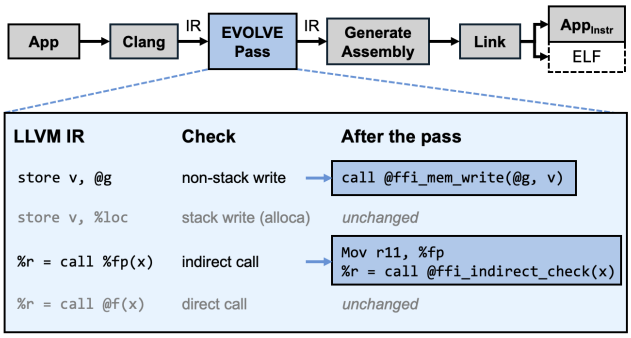

图 3：EVOLVE 插桩器。LLVM 处理遍会将每个非栈存储和间接调用重写为对监控器 FFI 层的 call 指令，然后再将代码编译为汇编。栈写入和直接调用保持不变。

内存写入。该处理过程会检查载荷中的每条 store 指令，但有两种例外情况。首先，通过精确匹配的允许列表排除监控器的 FFI 函数，因此监控器不会拦截 own 写入。其次，对栈分配变量（通过 alloca 分配）的 store 指令保持不变。栈占据的地址范围与关键区段分离，因此，源自 alloca 的指针不可能指向 I、O 或任何其他关键区段；而任何可能指向这些区段的指针都会经过插桩。对于其余每条 store 指令，处理过程都会将其替换为对 ffi mem write 的 call 指令，并传入目标_ _地址和值，以便监控器执行访问控制并更新 PoX 证据（见第 5 节）。

间接调用。该处理遍同样会检查每条 call 指令。对固定函数的直接调用保持不变。对于间接调用，其目标是编译时未知的函数指针，处理遍会将原始目标置于预留的暂存寄存器中（例如图 3 中的 r11），并将被调用函数替换为监控器 FFI 层中的 ffi_indirect_check，由后者在调用继续执行前验证目标。因此，插桩后的二进制文件中不再保留任何间接调用。修改后的中间表示随后继续进行汇编生成和链接，而载荷源代码保持不变。间接调用目标内部的写入也会像其他所有写入一样接受插桩。

FFI 层。插桩有效载荷与监控器链接，生成可执行且可链接格式（ELF）的二进制文件。为避免将载荷与监控器的内部实现耦合，载荷仅调用一组可由 C 调用的 FFI 函数（例如 ffi_mem_write、ffi_indirect_check）。每个 FFI 函数都会捕获调用者的返回地址，作为调用来源的证明，并将其传递给相应的监控器函数进行检查（第 5 节），从而使监控器的内部实现与载荷保持隔离。

自定义链接器脚本会将每个组件放入指定区段。多个操作共用的代码（例如机器学习推理库）经过插桩后只编译一次，并放入共享区段；每个操作都会调用该区段中的代码，而无需携带自己的副本。生成的 ELF 二进制文件会与加载器固件映像一起打包。

5. 监控器

本节介绍 EVOLVE 监控器，即机密虚拟机内部的 PoX 信任根。我们首先介绍它的组件，包括其维护的关键数据及其结构，然后介绍它在执行前、执行期间和执行后的行为。最后，我们讨论它所依赖的经验证的原语及其所保证的安全属性。

5.1. 组件和运行时行为

5.1.1. 组件。 图 4 所示的 EVOLVE 监控器由其维护的关键数据（I、 O 和 EXEC）、一个强制执行层以及一个经验证的原语库组成；下文将对这些内容进行介绍。载荷只能通过 FFI 层到达该监控器（回顾第 4 节）。

关键数据。EVOLVE 跟踪三个值，以记录执行了哪些内容以及执行是否正确。 I保存用户发送的输入，例如输入特征和模型选择，并由监控器在执行前写入。 O 保存操作的结果（例如模型的预测结果），且只能由请求的操作写入。EXEC 记录请求的操作是否已开始和完成，以及是否发生了策略违规：
（1）载荷写入以关键区（I、监控器或 EXEC）为目标。
（2）未授权调用方（例如未请求的操作）向 O写入。
（3）间接调用以垫片区域（I、O、监控器或 EXEC）中的地址为目标，但该地址不是允许的 FFI 入口点。
（4）间接调用以未经授权的操作为目标，或以共享区段为目标，而此时该区段尚未绑定。

EXEC 的组成EXEC。 EXEC 包含四个仅可由监控器写入的标志，每个标志都对应 EVOLVE 的 PoX 要求：

- EXEC_B (开始)：初始值为 0；监控器开始执行所请求的操作时设为 1。如果请求无效，则保持为 0，且不会执行该操作。
- EXEC_F (完成)：初始值为 0；操作返回时设为 1。
- EXEC_M (内存写入)：初始值为 1；发生此前讨论过的任何内存写入违规时设为 0。
- EXEC_I (间接调用)：初始值为 1；发生此前讨论过的任何间接调用违规时设为 0。

执行结束时， I、 O 和 EXEC 会被打包进 TDX 证明报告的 REPORTDATA 中；只有 EXEC 中的所有标志都设为 1，证据才有效。

监控器结构。 监控器由两部分组成：强制执行层 和一个经验证的原语库，这些原语是经过形式化验证的函数，用于执行监控器的安全检查。强制执行层与虚拟固件（例如 TD-Shim）集成。它提供 FFI 入口点，从二进制文件中读取区段边界，维护监控器状态（查找表、授权操作、已绑定的共享区段以及 EXEC 标志），并调用 TDX 模块生成报告。它不会判断是否允许访问，而是将相关输入（例如，

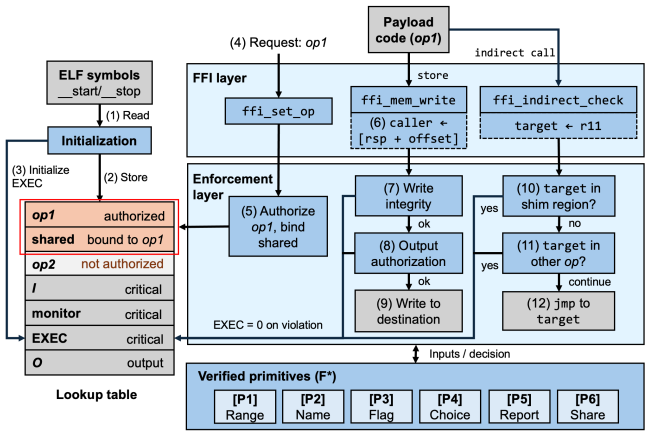

图 4：EVOLVE 监控器。启动时，监控器会将每个区段的边界注册到查找表中（1–3）。收到请求后，它会授权所选操作，并将共享区段绑定到该操作（4–5）。随后，它使用经验证的原语检查载荷中的每次内存写入（6–9）和间接调用（10–12）。共享区段初始时未绑定，并在步骤（5）中绑定到授权操作，此后便视为该操作的一部分。

目标地址、调用者地址和区段边界）传递给下文所述的原语，并根据结果采取相应操作。

这些原语分为六类，每类对应一项安全属性（见第 5.2 节）：范围检查 将地址映射到其所属区段 [P1]；有界名称操作比较并复制节名称 [P2]；标志操作 组合 EXEC [P3]；选择操作 验证所请求的操作 [P4]；报告操作将 I、O 和 EXEC 打包到 REPORTDATA 中 [P5]；以及 共享检查 将对共享区段的访问归属到绑定的操作 [P6]。

5.1.2. 运行时行为。监控器分三个阶段运行：启动时初始化、执行期间强制执行，以及执行后进行证明。图 4 展示了这些阶段，下面的步骤编号对应图中的编号。

初始化。在任何载荷代码运行之前，监控器会建立有效载荷内存映射并准备 EXEC。监控器位于垫片区域中的独立区段内，从二进制层面与载荷隔离。如图 4 所示，监控器首先从二进制文件中读取由链接器在构建时嵌入的各区段起始和结束地址（步骤 1）。随后，监控器将这些地址登记到查找表中（步骤 2），把各区段名称复制到定长字段中，并仅通过有界比较来匹配名称 [P2]。该表包含 I、监控器、EXEC、O，以及每个操作对应的一个区段（例如，模型路由场景中的 model1 和 model2）。共享区段也会被登记，初始时绑定到无操作。随后，监控器将 EXEC_M 和 EXEC_I 初始化为 1（步骤 3）。只有监控器区段中的代码才能写入 EXEC [P1]，，因此载荷无法操纵它。

接下来，监控器授权用户请求的单个操作。然后，用户发送请求（步骤 4），指定操作（例如，op_1），监控器通过 ffi_set_op 接收该请求。监控器首先验证请求的操作 [P4]；如果选择无效，则将 EXEC_B设为 0，且不运行该操作。否则，监控器将 op_1 标记为已授权，并将所有其他操作标记为未授权（步骤 5），从而只有 op_1 可以在此次执行期间写入 O。监控器还会将共享区段绑定到 op_1 [P6]，以便在此次执行的其余时间内，将共享区段代码视为 op_1 的一部分，并将 EXEC_B 设为 1。

执行期间。执行期间，监控器会根据内存映射和授权操作检查载荷的每次写入和间接调用，并将任何违规记录到 EXEC 中。载荷使用 op 运行。插桩器通过 FFI 层将每次非栈内存写入和间接调用路由至监控器，例如图 4 中由 op_1的代码执行的 store 指令和间接调用。

监控器会对每次内存写入强制实施两项属性。首先是写完整性，确保载荷绝不修改关键区（I、监控器或 EXEC）：当写入通过 ffi_mem_write 到达时，FFI 层会从栈中读取该调用的返回地址（步骤 6）（在 x86 上为 [rsp + offset]），以确定调用 ffi_mem_write 的载荷代码位置。FFI 层将此调用方地址连同目标地址一起传递给监控器。随后，监控器使用查找表 [P1]（步骤 7）检查目标是否位于关键区（I、监控器或 EXEC）内。如果是，则阻止该写入，并将 EXEC_M 设为 0。

其次，输出授权会跟踪是否只有请求的操作写入 O。如果目标是 O，监控器会检查（步骤 8）调用者是否位于步骤（5）[P1]中设置的授权操作区段集合内。位于共享区段中的调用者同样获得授权，因为该区段绑定到 op_1 [P6]。如果调用者位于其他任何位置（例如op_2），则写入会被阻止，并将 EXEC_M 设置为 0。通过这两项检查的写入会继续执行到目标地址（步骤 9）。任何违规都会记录在 EXEC 中，并在证明时对验证者可见。

对于间接调用，监控器会实施粗粒度控制流完整性。ffi_间接_check 将调用目标传递给监控器（步骤 10），监控器会检查该目标是否位于 shimregion [P1] 内。除非目标是允许的 FFI 入口点，否则此类调用会被阻止，并将 EXEC_I 设置为 0，从而防止访问监控器的内部函数（例如设置 EXEC 标志的函数）[1]。随后，监控器会检查目标是否位于未授权操作中，或位于共享区段中但该区段未绑定到授权操作 [P6]（步骤 11）。如果是，则将 EXEC_I 设置为 0。步骤（10）中未被阻止的调用随后会继续执行到其目标（步骤 12）。

执行后。当操作返回时，监控器将 EXEC_F 设为 1，并解除共享区段 [P6],的绑定，使其不再被视为任何操作的一部分。随后，监控器检查四个标志是否全都恰为 1 [P3],，以确认未检测到任何违规行为。它将 I、O 和 EXEC 打包到 REPORTDATA [P5]中，并请求一份硬件证明报告

1. 只需检查入口点，因为假定 Intel 控制流强制技术（CET）的间接分支跟踪（IBT）已启用（回顾第 3.2 节），且它会阻止间接调用和跳转到函数内的任意偏移位置 [79]。

由 TDX 模块提供，Vrf 可进行远程验证（一种基于EVOLVE的PoX协议见附录 A）。

5.2. 安全属性

每一类经验证的原语都维护了 EVOLVE 监控器的以下一项安全属性；第 5.3 节给出它们的契约，第 6.3 节说明这些契约如何确立各项属性。

[P1] 健全且完备的包含性：区段成员资格通过对地址 a 与内存范围 $(start, end)$之间的$start \leq a < end$执行精确的区间检查来判定，并且每个报告的匹配/不匹配结果都真实可靠。

[P2] 名称完整性：名称以有界 C 字符串的形式进行比较，读取时绝不会越过任一字符串的边界。存储的名称会截断至 31 字节，并在 32 字节字段中以 NUL 填充，不留任何过期字节。查找返回第一个匹配项，因此攻击者随后提供的重复项无法覆盖较早的名称。**[P3] 失败即关闭的 EXEC 标志**：仅当四个 EXEC 标志都恰好等于 1 时，执行才会成功。任何其他值均视为未设置，且只要有一个标志未设置，PoX 结果就会失败。**[P4] 操作选择授权**：单个例程将选择整数转换为操作索引，且仅接受合法范围 $1 \ldots$ 最大选择值内的整数；上限最大选择值 在编译时确定。无论注册表如何，超出范围的选择都会被拒绝；范围内的选择则仅在其操作区段已注册时才会被接受。**[P5] 忠实的报告打包**： I、 O以及四个 EXEC 标志会写入 REPORTDATA 中固定且互不重叠的字段，不会修改其他任何字节；失败的标志绝不会显示为通过。

[P6] 共享区段隔离：仅当共享区段在执行期间绑定到授权操作时，才允许从共享区段进行访问；若其未绑定、绑定到其他操作，或在执行之外使用，则绝不允许访问。

5.3. 形式化验证的原语

本节将每个监控器原语描述为一个契约，其中包含调用时必须满足的前置条件，以及该原语保证在返回时成立的后置条件。对于每个满足前置条件的输入，每个原语都满足其后置条件。因此，监控器的正确性取决于这些规约的正确性。第 6.3 节介绍我们如何使用 F* 验证这些契约 [48], 、将其提取为 C 代码，以及它们如何确立属性 P1–P6。在本节中，有效(b) 表示缓冲区 b有效且可访问。条件 $h_0 = h_1$ 表示调用不会改变堆，其中 h_0 和 h_1 分别表示调用前后的堆。

5.3.1. 成员关系与查找检查。 图 5 给出了范围检查成员关系原语函数的契约 1 和契约 2。这两个函数都接受以下输入：

契约 1：$r \leftarrow$索引_属于_范围$(starts, ends, n, a)$:: 要求： $live(starts) \wedge live(ends) \wedge (|starts| = |ends| = n)$ 确保：$(h_0 = h_1) \wedge [(r < n) \wedge (a \geq starts[r] \wedge a < ends[r]) \wedge (\forall k.\ 0 \leq k < r \Rightarrow \neg($ $a \geq starts[k] \wedge a < ends[k])] \vee [(r \geq n) \wedge \forall k.\ 0 \leq k < n \Rightarrow \neg($ $a \geq starts[k] \wedge a < ends[k])]$契约 2： $res \leftarrow$属于_任意_范围$(starts,$ $ends, n, a)$:: 要求： $live(starts) \wedge live(ends) \wedge (|starts| = |ends| = n)$ 确保：$(h_0 = h_1) \wedge (res \Rightarrow (\exists j < n.\ (a \geq starts[j] \wedge a < ends[j])))$

图 5：范围成员查找契约（[P1]）。

- 起始地址：一个包含内存范围起始地址列表的缓冲区；
- 结束地址：一个包含内存范围结束地址列表的缓冲区；
- n：*starts*和*ends*中的地址数量；
- a：正在进行检查的地址。两个契约都要求输入缓冲区的大小为n。

契约 1 规定了函数`index_of_range`，该函数返回 r ，作为包含 a 的第一个范围的索引；若没有区域包含它，则返回值 $r \geq n$ 。后置条件保证堆保持不变（$h_0 = h_1$）。如果 $r < n$，则 a 属于第 r 个范围（即 $starts[r] \leq a < ends[r]$） ，且之前没有任何区域包含它。否则，没有任何区域包含 a。

契约 2 指定了`in_any_range`，它返回一个布尔值*res* ，用于指示 a 是否属于n中的任一范围。其后置条件保证堆保持不变（$h_0 = h_1$） ，并且*res*当且仅当某个区域包含 a 时为真（即，存在 $j < n$ 使得$starts[j] \leq a < ends[j]$）。

5.3.2. 名称比较与注册。 图 6 展示了有界 C 字符串和节名称查找原语函数的契约 3–6。

契约 3 指定了`cstr_len`，它接收一个缓冲区 b 和一个上限 n，然后返回字符串长度 r。前置条件要求 b 有效，且 n 不超过 b 的大小。后置条件保证堆保持不变 $(h_0 = h_1)$，并且 $r \leq n$，且之前的每个字节均非零（即，对于所有 $i < r$，$b[i] \neq 0$）。此外，如果 $r < n$，那么 r 就是长度，因此索引 r 处的字节必须为零（即空字符终止符）。

契约 4 指定了 `str_eq`，它会返回一个布尔值*res* ，用于表示两个输入字符串 (a, b) 在界限(n_a, n_b) 内是否相等。前置条件要求两个(a, b) 均有效，且各自的长度不超过指定界限。后置条件保证堆保持不变（$h_0 = h_1$））。它还保证 *res* 为真，当且仅当（a, b）包含相同的有界 C 字符串。具体而言，必须存在某个索引 $k < min(n_a, n_b)$ ，使得该索引之前的所有字节都相等且非零。如果 k 落在任一界限内，则索引 k 处的字节也必须为零。

契约 5 指定了 名称_零_尾部，它接受一个目标缓冲区 *dst* 和一个索引 *k* ，并返回更新后的缓冲区 *dst'*。前置条件要求 *dst* 有效，大小为

契约 3： $r \leftarrow$**cstr_len**(b, n)：
要求： $live(b) \land n \le |b|$
确保： $h_0 = h_1 \land r \le n$
$\land\ \forall i < r.\ b[i] \neq 0 \land (r < n \Rightarrow b[r] = 0)$

契约 4： $res \leftarrow$**cstr_eq**(a, n_a, b, n_b)：
要求： $live(a) \land live(b) \land n_a \le |a| \land n_b \le |b|$
确保： $h_0 = h_1 \land [res \Leftrightarrow \exists k \le \min(n_a, n_b).$
$(\forall i < k.\ a[i] = b[i] \neq 0)$
$\land (k < n_a \Rightarrow a[k] = 0) \land (k < n_b \Rightarrow b[k] = 0)]$

契约 5： $dst' \leftarrow$**name_zero_tail**(dst, k)：
要求： $live(dst) \land |dst| = 32 \land k \le 32$
确保： $\forall i < 32.\ (i < k \Rightarrow dst'[i] = dst[i])$
$\land (i \ge k \Rightarrow dst'[i] = 0)$

定义 1：
$entry_matches(tbl,\ i,\ q,\ q_{len}) \triangleq \exists k \le \min(32,\ q_{len}).$
$(\forall j < k.\ tbl[32i + j] = q[j] \neq 0)$
$\land (k < 32 \Rightarrow tbl[32i + k] = 0)$
$\land (k < q_{len} \Rightarrow q[k] = 0)$

契约 6： $r \leftarrow$**find_name**$(tbl,\ cnt,\ q,\ q_{len})$：
要求： $live(tbl) \land live(q) \land 32 \cdot cnt \le |tbl| \land q_{len} \le |q|$
确保： $h_0 = h_1 \land r \le cnt$
$\land (r < cnt \Rightarrow entry_matches(tbl,\ r,\ q,\ q_{len}))$
$\land \forall i < r.\ \neg entry_matches(tbl, i, q, q_{len})$

图 6：有界 C 字符串比较和节名称查找的契约（[P2]）。

32，并满足 $k < 32$。后置条件保证 *dst'* 保留索引 k 之前的每个字节，并将从*k* 开始的每个字节都置为零。

契约 6 的后置条件通过在定义 1 中定义谓词 表项匹配 而得到简化。该谓词作用于一个表*tbl*、一个索引*i*、一个查询*q*以及查询长度q_{len}。它规定，当表项*i* 和*q* 在某个索引*k*之前包含相同的非零字节，且索引*k*不超过 32 或 q_{len}时，表项*i* 匹配*q* 。如果 $k < 32$，则表项必须在索引*k*处包含一个零字节。如果 $k < q_{len}$，查询也必须在该索引处包含一个零字节。

契约 6 规定了 查找_名称，接收相同的输入，并返回查询名称*q* 存在处的索引*r* 。它要求*tbl* 和*q* 处于活动状态，要求*tbl* 包含*cnt* 个每个 32 字节的表项，并要求*q* 不超过其边界 q_{len}。它保证堆保持不变（$h_0 = h_1$），并且 $r \le cnt$。如果 $r < cnt$，则它确保表项 r 根据定义 1 匹配 q。它还保证*r* 之前没有任何表项匹配*q*。因此，当没有表项匹配时，*r* 标识第一个匹配的表项，或等于*cnt* 。

5.3.3.EXEC 标志判定。 图 7 展示了用于管理 EXEC 标志的已验证原语函数的契约 7 和契约 8。契约 7 展示了函数 `all_set4`，该函数返回一个布尔值*res* ，用于确定所有 EXEC 标志是否均已设置。它接收一个保存这些标志的缓冲区*x* 作为输入。其前置条件要求 x 至少保存 7 个元素，这些元素对应于 EVOLVE 扩展到报告中的数据（详见第 5.3.5 节）。作为后置条件，它确保堆状态不会发生变化（$h_0 = h_1$）），并且当所有

契约 7： $res \leftarrow$**all_set4**(x)：要求： $live(x)$
$\land 7 \le |x|$确保： $h_0 = h_1 \land (res \iff \forall i \in [3, 7)$
$. x[i] = 1)$**契约 8：** $res \leftarrow$**flag_set_at**(x, off)：要
求： $live(x) \land off < |x|$确保：
$h_0 = h_1 \land (res \iff x[off] = 1)$

图 7：设置 EXEC 标志的契约（[P3]）。

契约 9： $res \leftarrow$**选择_已授权**(reg, c)：
前置条件： $live(reg) \land |reg| = max_choice$
后置条件： $h_0 = h_1 \land [res \Leftrightarrow (1 \le c \le max_choice) \land reg[c - 1]]$

Figu图 8：模型选择授权的契约 （[P4]）。

x 中预期用于存放 EXEC 标志（3-6）的元素被设为 1。契约 8 展示了函数 `flag_set_at`，该函数返回布尔值 *res*，用于指示 *x* 中偏移量 *off* 处的 EXEC 标志是否已设置。其前提是 *x* 处于有效状态，且 *off* $< x$。其后置条件是堆状态不会发生变化（$h_0 = h_1$），并且仅当 $x[off] = 1$ 时，*res* 才为 true。

5.3.4. 模型操作选择授权。 监控器根据输入整数确定有效操作；如果节表中没有已注册的区段，则拒绝运行。图 8 展示了原语函数 `choiceauthorized`的契约 10。它返回布尔值 *res*_，用于指示输入选择 c 是否根据缓冲区 *reg*中各寄存器区段的状态获得授权。其前提是 *reg* 是一个有效缓冲区，且其大小等于最大选择数（$max\ choice$）。其后置条件是堆状态不会发生变化（$h_0 = h_1$），并且仅当 c 是有效选择（即 $1 \le c \le max\ choice$），且区段 c 确已注册（$reg[c - 1]$ 为 true）时，*res* 才为 true。_

5.3.5. 打包报告数据。 图 9 展示了用于构造 PoX 证据中所使用报告数据的原语函数的契约 11-12。契约 11 展示了 打包_报告，即用于更新打包报告数据的函数。它更新缓冲区 w，该缓冲区将在下一个状态中变为 w'。它的输入包括输入区域（ir_0、ir_1）、输出区域 or以及第 5.1 节中的四个 EXEC 标志（b、f、m、i）。它要求 w处于活动状态且至少有七个槽位。它确保 w' 在前七个缓冲区中按顺序包含所有输入。如果 w 大于七个槽位，则 w' 的上部槽位等价于 w。

契约 12 展示了 解包_标志，该函数将所有 EXEC 标志解包为一个返回的单一布尔结果 *res* 。它将报告数据缓冲区 *w* 作为输入。它要求与契约 11 相同的前置条件。它确保堆状态不发生变化，并返回 *res* ；仅当所有标志均已设置（位于索引 3 至 6）时，该值才为 true。

5.3.6. 共享代码授权。图 10 展示了共享区段授权原语函数的合约 12–14。合约 12 展示了 `in_bound_share`，它接受以下输入：(1) in_{ex}：一个布尔值，用于说明当前是否正在执行；(2) *own*：当前的操作

契约 10: w' ←pack_report($w, ir_0, ir_1, or, b, f, m, i$): 要求:
$live(w) \wedge 7 \leq |w|$确保: $w' = \langle ir_0, ir_1, or, b, f, m,$
$i\rangle \parallel w[7..|w|]$契约 11: res ←unpack_flags(w): 要求:
$live(w) \wedge 7 \leq |w|$确保: $h_0 = h_1 \wedge [res \Leftrightarrow ($
$w[3] = 1 \wedge w[4] = 1 \wedge w[5] = 1 \wedge w[6] = 1)]$

图 9：打包报告数据的契约（[P5]）。

契约 12: res ←**in_bound_share**(in_{ex}, own, c, a, ss, se):
$res \iff in_{ex} \wedge own \neq 0 \wedge own = c \wedge a \geq ss \wedge a < se$
契约 13: res ←**write_auth**(in_{ex}, own, c, a, st, en, n, ss, se):
前置条件: $live(st) \wedge live(en) \wedge |st| = |en| = n$
后置条件: $h_0 = h_1 \wedge [res \Leftrightarrow in_{ex} \wedge ($
$\exists j < n.\ c = j + 1 \wedge a \geq st[j] \wedge a < en[j]$
$\vee\ (own \neq 0 \wedge own = c \wedge a \geq ss \wedge a < se))]$
契约 14: res ←**call_unauth_at**(in_{ex}, own, c, tgt, n, cs, ss, se):
$res \iff (cs < n \Rightarrow\ (\neg in_{ex} \vee c \neq cs + 1)\ )$
$\wedge\ (cs \geq n \Rightarrow\ (tgt \geq ss \wedge tgt < se$
$\wedge \neg\ (in_{ex} \wedge own \neq 0 \wedge own = c)\ )\ )$

图 10：共享区段授权的契约（[P6]）。

拥有共享区段；(3)c：模型操作选择；(4) a：关注地址；(5) *(ss,se)*：共享区域的起始/结束地址。这是一个纯谓词，当 in_{ex} 为真、*own* 等于 c 且非零，并且关注地址位于共享区域内（即 $ss \leq a < se$）时，返回布尔结果*res* 为 true。

契约 13 规定`write_auth`，除相同的先前输入外，还接收包含操作区域起始/结束地址的缓冲区（*st,en*）。它要求*st* 和 *en* 是大小为*n*的有效缓冲区。它确保堆状态不会发生变化（$h_0 = h_1$）），并且仅当 in_{ex} 为真（满足以下任一条件）时，*res* 才返回 true：
(1) 存在一个与选择 c 对应的已注册区域（即存在 $j < n$，使得 $c = j+1$ 且 $st[j] \leq a < en[j]$）；或者 (2) *own* 等于 c 且不为零，并且 a 位于共享区域中（即 $ss \leq a < se$）。

最终的契约是`call_unauth_at` 契约 14，它返回一个布尔结果*res*，说明对*tgt* 的间接调用是否经过授权。它是一个纯谓词：当调用点关联的操作（cs）不是当前正在执行的所选操作，或者调用目标（tgt）位于共享区域中且不存在与所选操作匹配的所有权绑定时，返回*res* 为 true。

具体而言，如果 $cs < n$，当 call 指令不在所选操作范围内（即，in_{ex} 为 false），或调用点索引与该操作不对应（即，$c \neq cs + 1$）时，就会标记该 call 指令。当 $cs \geq n$ 时，如果 tgt 位于共享区域之外（即，$ss \leq tgt < se$），且共享区域未绑定到当前正在执行的所选操作 c（即，in_{ex} 为 true 且 own等于非零 c这一条件不成立），就会标记该 call 指令。

6. 实现与评估

6.1. 原型配置

设置。 EVOLVE 原型运行于一台搭载 Intel TDX 的主机上，该主机配备 Intel Xeon Silver 4514Y 处理器和 512 GB 随机存取存储器（RAM）。信任域配有 4 个虚拟 CPU 和 2 GB RAM，QEMU [80] 会启动固件映像。运行信任域的主机 CPU 固定在 2.0 GHz 基础频率。我们使用 CPU 的时间戳计数器测量信任域内的负载加载、模型执行和引文生成时间，并在主机上测量构建和启动时间。用户输入通过虚拟串行端口发送到信任域；该端口由 QEMU 通过传输控制协议（TCP）套接字提供。EVOLVE 加载载荷并配置环境后，会等待用户请求。用户通过另一个串行控制台请求模型进行推理，随后 EVOLVE 执行模型并生成 PoX 证据（详见第 3.3 节）。

工作负载。 表 1 总结了这些模型。我们在来自加州大学欧文分校机器学习资料库 [81] 的五个数据集上，为每种模型类型训练小型（S）和大型（L）配置：威斯康星乳腺癌诊断数据集（BCWD） [82], ISOLET [83],MuskV2 [84], 图像分割（图像） [85], 和 Iris [86]。我们将每个模型称为“数据集_模型_大小”（例如，BCWD_MLP_S）。对于决策树，配置指训练后的结构（树的数量、节点数、叶节点数和深度）；对于多层感知机，则指网络架构（层大小和深度）。对于决策树，S 配置将特征数的平方根作为每次分裂的候选特征，使用浅层树（深度受较小常数或数据集大小限制），并包含 10 棵树。

L 配置在每次分裂中使用更多候选特征，树的深度随样本量增加，并且树的数量与输入维度成正比。例如，ISOLET_DT_S 有 10 棵树、582 个节点、296 个叶节点，深度为 5；而 ISOLET_DT_L 有 246 棵树、231K 个节点、116K 个叶节点，平均深度为 20.7。对于多层感知机，S 配置有一个隐藏层，其神经元数量比输入特征多一个；而 L 配置有四个隐藏层，各层宽度从输入维度到一或两个神经元线性递减。例如，BCWD_MLP_S（30 个特征）有一个包含 31 个神经元的隐藏层，而 BCWD_MLP_L 的各层大小为[31, 21, 11, 1]。编译后的模型内存占用最高可达 29.17 MB（ISOLET_DT_L）。

负载布局。 我们使用 emlearn [87] 将模型编译为 C 头文件，用于 TD-Shim 裸机工作负载。每种类型的 S 和 L 模型分别链接到一个载荷中，每个模型位于其自身的操作区段内，请求则选择要运行的模型。对于 MLP，共享推理库（层前向传播、激活函数和最大值索引）只需编译一次，并置于共享区段中。由于该库具有通用性，并在运行时接收层大小、权重和偏置等模型数据，因此每个 MLP 的共享区段都相同，而每个操作区段仅包含模型的入口点和库调用。对于 Iris_MLP_S 和 Iris_MLP_L，每个操作区段

表 1：评估集包含五个数据集上决策树和多层感知机分类器的小型（S）和大型（L）配置。对于决策树，配置指定模型结构：树的数量、节点数、叶节点数和平均深度。对于多层感知机，配置指定隐藏层大小。最后一列报告每个模型的内存占用。

数据集	类型	配置（S，L）	内存大小（S，L）
BCWD [82]	DT	（10，318，164，5），（12，356，184，6.6）	24.53 KB，28.23 KB
	多层感知机	[31]，[31，21，11，1]	12.68 KB，23.64 KB
ISOLET [83]	DT	(10, 582,296,5), (246,231K, 116K, 20.7)	45.22 KB,29.17MB
	多层感知机	[618]，[618，413，207，2]	4.58 MB，8.35 MB
MuskV2 [84]	DT	(10, 512, 261, 5), (66, 16.5K, 8.3K, 19.8)	39.49 KB，1.86 MB
	MLP	[167]，[167，112，57，1]	323.42 KB，612.73 KB
图像分割 [85]	DT	(10, 196, 103, 5), (7, 217, 112, 9)	16.39 KB，20.17 KB
	多层感知机	[20]，[20，14，8，2]	7.56 KB，11.70 KB
Iris [86]	DT	(10, 168, 89, 4.7), (2, 26, 14, 5)	13.73 KB, 3.01 KB
	多层感知机	[5]，[5，4，3，2]	1.73 KB, 2.79 KB

的大小为 0.4 KB，共享区段中包含一个 3.6 KB 的库。DT 模型包含自包含的树和投票逻辑，不使用共享区段。

对于每个模型，I 和 O 均初始化为 0。在运行时，I 根据用户请求更新，而 O 则根据操作的输出更新。所有四个 EXEC 标志初始值均为 0，并由 EVOLVE 将其设为 1（如第 5.1 节所述）。

6.2. 成本分析

为了评估 EVOLVE 在机器学习工作负载中的表现，我们以模型路由 [88], [89]作为代表性场景。路由器会在复杂度不同的随机森林决策树和多层感知机分类器之间进行选择。我们从两类成本分析 EVOLVE：固定安装成本和运行时性能成本。

6.2.1. 固定成本。 EVOLVE 的 LLVM 处理遍使构建系统增加了 245 KB，但不会影响已部署固件的大小或 TD 内存占用。EVOLVE 监控器使 TD-Shim 二进制文件从 221.5 KB 增至 232.4 KB，增加了 10.9 KB（4.92%）。这项成本反映了分配给监控器的内存；在载荷的整个运行时期间，监控器都会保持活动状态。相比之下，标准 TD-Shim 只会启动载荷。

6.2.2. 性能成本。 性能成本体现了 EVOLVE 对载荷构建和运行的影响。我们测量以下指标：

- **基线：** 不使用 EVOLVE 时载荷的构建和执行时间。基线仍会生成 TD 初始状态的 TDX 报价，但不会提供有关运行了哪项操作或哪个操作写入了其输出的证据。有关非 TDX 基线下的 TDX 开销，请参阅 [90]。
- **构建时开销：** 由于纳入 EVOLVE 的模块而增加的编译和链接时间。
- **运行时开销：** 启动 TD 并执行载荷所需的额外时间，反映了通过监控器路由内存写入和间接 call 指令的成本。

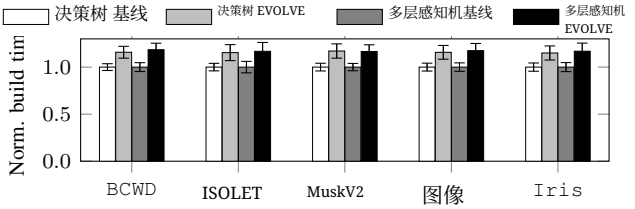

图 11：DT 和多层感知机工作负载使用 EVOLVE 的构建时间；每种工作负载均包含 S 和 L 模型，并相对于基线进行归一化（60 次构建的均值 ± 标准差）。

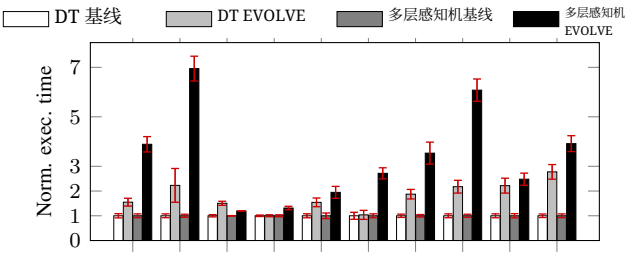

图 12：在五个数据集上，尺寸为 S 和 L 的决策树和多层感知机模型采用 EVOLVE 时的模型执行时间，相对于基线（基线 = 1）归一化（均值 ± 标准差，$n = 30$；决策树基线在 Image-S 上的一次异常运行已排除）。

- **监控器调用时间：** 随着已注册操作区段总数增加，监控器执行检查所花费的时间。

构建时开销。 图 11 展示了各模型类型和数据集的模型路由载荷（包含 S 和 L）构建时长相对于基线的归一化结果。对于每种模型类型，五个数据集的构建时开销均保持一致。对于决策树模型，EVOLVE 将构建时长增加至基线的 1.15–1.17× 倍（从 25.7–29.2 秒增加至 29.7–33.7 秒）。对于多层感知机模型，由于其规模更大，增幅略高，为基线的 1.17–1.18× 倍（从 25.6–35.3 秒增加至 30.0–41.2 秒）。这项开销完全来自 EVOLVE 对载荷执行的插桩过程，以及包含 EVOLVE 监视器的虚拟固件的编译。构建时开销是每次部署的一次性成本，因为固件映像只需在启动前构建一次，之后即可用于处理多个请求。

运行时开销。 所有配置的运行时开销都可以忽略不计，均在基线（平均 3.04 s）的 ±0.4% 以内；但 Iris_MLP_S 除外，它因两次运行较慢而产生了 0.89% 的开销。我们将运行时分为三个阶段：(1) 初始化并加载载荷，(2) 配置监控器并执行操作，以及 (3) 引文生成。图中未展示第一阶段，因为 TDX 初始化占据了该阶段的大部分时间：所有配置均耗时 2.46 s，且在基线的 ±0.4% 以内（Iris_MLP_S 为 1.01%）。

模型执行开销（如图 12 所示）取决于模型类型、大小和数据集。对于决策树，EVOLVE 会使执行时间增至基线的 1.50–2.78× 倍；除最大的两棵树外（每次推理仅需 2–4 μ 秒），这是因为树

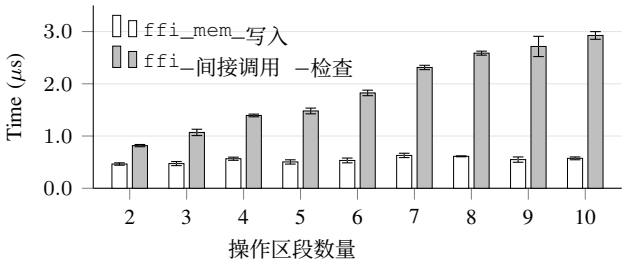

图13：随着操作区段数量增加，EVOLVE 监控器对内存写入（ffi_mem_write）和间接调用检查（ffi_indirect_check）的调用时间（30 次运行的均值 ± 标准差，第 1 次运行作为预热不计入）。每次运行平均调用 100 次。

推理所需的内存写入较少。对于最大的两棵树（ISOLET_决策树_L 和 MuskV2_决策树_L），差异仍在一个标准差以内。对于多层感知机，EVOLVE 会使执行时间增至基线的 1.19–6.95× 倍。ISOLET 上的增幅最低（1.19–1.31× 倍），而 BCWD 和图像的 L 变体增幅最高（6.08–6.95× 倍）。这是计算与受监控写入之间的平衡所致。每次受监控写入都会增加约 0.5 μ 秒的固定开销（图13），与模型无关。BCWD 和图像多层感知机在受监控写入之间执行的计算很少，因此花在监控器上的时间超过了计算时间，而它们的 L 变体会产生更多写入。相比之下，ISOLET 在写入之间执行的计算多得多，因此相同的监控开销只占运行时的一小部分。图中未列出引文生成，因为它在所有配置下都相同：无论是否使用 EVOLVE，耗时均为 0.020–0.022 毫秒。这一固定开销来自执行后的一次 TDCALL，与模型大小或数据集无关。

总体而言，与基线相比，EVOLVE 使决策树的每请求运行时间（即操作执行和引文生成）平均增加 1.07× ±0.04，使多层感知机平均增加 2.13× ±0.81；增幅范围从 ISOLET_决策树_L 中无法测出的增加，到 BCWD_多层感知机_L 中的 3.73×；绝对增幅最大的是 ISOLET_多层感知机_L，为 1.864 毫秒（1.31×）。

监控器调用时间。图13进一步探究了 EVOLVE 监控器运行时开销的增长情况：通过测量操作区段数量从 2 个 N 增加到 10 个时的监控器调用时间。对于内存写入，每次调用耗时为 0.46–0.63 μs，在 N中没有呈现一致趋势。这种变化在多次重复运行中均可复现（误差在 1% 以内），说明其原因可能是每次运行的内存布局不同导致的 CPU缓存行为，而非操作区段数量。每次调用的耗时差异最多为 0.17 μs。间接调用的耗时则呈线性增长，从 N=2时的 0.82 μs 增至 N=10时的 2.93 μs，即每增加一个区段耗时增加 0.28 μs。

摘要。 EVOLVE 带来两项主要开销。首先，构建载荷所需时间增加 15–18%，但这是每次部署只需支付一次的开销。其次，EVOLVE 使每请求运行时间平均增加 1.07× ±0.04（DT）和 2.13× ±0.81（MLP）。这一增长源于每个监控器每次写入增加 0.46–0.64 μ 秒，每次间接调用增加 0.82–2.93 μ 秒。作为这些开销的交换，EVOLVE 提供形式化安全保证（第 6.3 节），并能适用于不同的 CVM 架构（第 6.4 节）。

6.3. 安全性分析

EVOLVE 在 TD-Shim [47] 虚拟固件中加入了经 F* 验证的策略检查。每个原语都有契约，具体说明见第 5.3 节，这些契约在形式上保证第 5.2 节中的安全属性 P1–P6。我们使用 F* 实现这些原语，并通过 KaRaMeL [91]将其提取为 C 代码。我们通过 FFI 将 C 代码链接到 Rust 固件中。我们的保证依赖于正确的 F* 工具链、C 语言提取和 C 编译器。

[**属性1]：健全且完备的包含性**。此属性要求进行精确的区间测试：起点 $\leq a <$ 终点。每个报告的匹配项都必须有效（可靠性），且每个有效匹配项都必须被报告（完备性）。检查还必须使用正确的表和严格上界。契约 1（索引_属于_范围）和契约 2（在_任意_范围中）确立了这些条件。契约 1 仅当范围 r 包含 a 且不存在更早的范围时，才返回 $r < n$ 。仅当没有范围包含 a时，它才返回 $r \geq n$。当且仅当某个 $j < n$ 满足相同的区间测试时，契约 2 才返回 true。这种等价关系确保了可靠性和完备性。严格上界会排除位于终点处的地址。

[**属性2]：名称完整性**。此属性要求进行有界 C 字符串比较、最多存储 31 个名称字节、在 32 字节字段的剩余部分进行 NUL 填充，并查找第一个匹配项。契约 3（C 字符串_长度）确保长度 r 不超过 n，且例程不会读取超出该界限的字节。其前置条件要求 n 适合放入缓冲区。契约 4（字符串_相等）当且仅当两个输入包含相同的有界 C 字符串时返回 true。它只在索引 $k <$ 最小值(n_a, n_b)处比较字节，因此不会读取超出任一缓冲区的数据。契约 5（名称_清零_尾部）保留 32 字节字段的前 k 个字节，并将从 k 起的所有字节清零。由于 $k \leq 31$，不会残留过期字节。契约 6（查找_名称）返回索引 r ，其对应的条目与查询匹配，且此前没有条目匹配。因此，后出现的重复项不会遮蔽较早的名称。

[**P3]：失败关闭的EXEC 标志**。此属性要求仅当四个标志都恰好等于1时才判定成功。任何其他值都必须视为未设置，且只要有一个标志未设置，就必须使 PoX 结果无效。契约 7（全部_设置4）和契约 8（标志_设置_于）规定了这些条件。契约 7 检查恰好四个标志槽位（索引 3 至 6），并且仅当每个槽位都等于1时才报告成功。契约 8 对一个标志应用相同规则。只有当$x[off] = 1$时，它才返回 true。两个例程都不会将任何非零值视为 true。由于结果合并了全部四个标志，只要有一个标志未设置，结果就为 false。前置条件要求缓冲区至少包含七个元素，因此所有读取操作都不会越界。

[**P4]：操作选择授权**。此属性要求例程仅在 $1 \leq c \leq$最大选择值 以及相应的操作区段已注册时，才对选择进行授权。无论注册表中是否存在，任何超出范围的选择都必须失败

。契约 9（选择_已授权）规定了这些条件。监控器没有其他途径作出授权决策。其后置条件要求真结果必须满足两个条件： c位于 $1...$最大选择值范围内，并且$reg[c-1]$ 为真。因此，超出范围的 c 始终失败。前置条件要求reg 具有最大选择值 个条目，因此每次有效查找都保持在边界内。对于范围内的选择，授权取决于其操作区段的注册状态。

[**P5]：忠实的报告打包**。 此属性要求将 I、 O 和四个 EXEC 标志放置在 REPORTDATA 中固定且互不重叠的字段内。它还要求打包操作保持所有其他字节不变，并保留失败标志。契约 10（`pack_report`）和契约 11（`unpack_flags`）确立了这些条件。契约 10 按固定顺序将 ir_0、 ir_1、 or 和标志 (b、f、 m、 i）写入 w 的前七个槽位。每个值都有自己的槽位，第七个槽位之后的所有槽位均保持不变。该例程按所提供的值存储每个标志，因此不会将失败标志转换为通过值。契约 11 读取相同的标志索引（3 至 6），并且仅当四个标志全部等于 1 时返回 true。因此，打包和解包使用相同的布局，解包不会将失败标志视为通过。

[**P6]：共享区段隔离**。此属性要求仅在执行期间，且仅当共享区段属于授权操作时，才允许访问共享区段。共享区段未绑定、属于其他操作，或在执行之外被访问时，必须拒绝访问。契约 12（`in_bound_share`）、契约 13（`write_auth`）和契约 14（`call_unauth_at`）确立了这些条件。契约 12 仅在 in_{ex} 成立、$own = c$、own 非零且 $ss \leq a < se$ 时授予访问权限。因此，它会拒绝访问未绑定的区段（$own = 0$）、绑定到其他操作的区段（$own \neq c$），或执行之外的访问（$\neg in_{ex}$）。契约 13 对共享区段写入使用相同的绑定条件。仅当in_{ex} 成立，并且 a 位于操作 c 的已注册区域中，或共享区段属于 c 时，它才返回 true。契约 14 对间接调用应用相同的隔离机制。当调用点属于另一个操作时，它会标记来自这些调用点的调用；当调用进入共享区域且该区域不属于当前正在执行的操作时，它也会标记该调用。

6.4 多功能性

EVOLVE 具有多功能性，因为其模块（一个插桩遍和面向客户机虚拟固件的监控器扩展）仅需要所有主要商业机密虚拟机平台都提供的特征。表 2 显示，Intel TDX、AMD SEV-SNP 和 Arm CCA 提供四项必需特征：数据执行保护（DEP）、控制流完整性、远程证明和客户机虚拟固件。

这三个平台都通过 CPU 的内存管理单元（MMU）实施 DEP，内存管理单元会使用客户机页表中的执行权限位。Intel TDX 通过英特尔控制流强制技术（Intel CET）支持控制流完整性（CFI） [79],，该技术为返回地址提供影子栈，并为间接调用和跳转提供 ENDBR64 落地点。Arm 提供指针认证码（PAC）来保护

表 2：EVOLVE 能够跨商业机密虚拟机实现多功能性，因为它们都提供四项基线功能：DEP、控制流完整性、远程证明和客户机虚拟固件（GVF）。

CVM	DEP	CFI	远程证明	GVF
Intel TDX	MMU	CET（影子栈 +IBT）	TDX 模块 +引用 enclave	TD-Shim
AMD SEV-SNP	内存管理单元	CET影子栈 +编译器	SP	OVMF/Oak stage0
Arm CCA	MMU	指针认证码 +分支目标识别	RMM	EDK2/直接启动

返回地址，并通过分支目标识别（BTI）为间接调用和跳转强制使用着陆垫 [92]。AMD SEV-SNP 使用 CET 影子栈 [93] 以及编译器对基于类型的前向边控制流完整性的支持 [94]。

每个平台都使用不同的信任根进行远程证明。Intel TDX 使用 TDX 模块和引用 enclave（详见第 2 节）。AMD 使用其安全处理器（SP） [95],，而 Arm CCA 使用 Realm 管理监控器（RMM） [96]。这三个平台也都提供虚拟固件，用于配置并启动来宾负载。Intel TDX 使用 TD-Shim [47],，EVOLVE 在此基础上进行了扩展。AMD SEV-SNP 使用开放虚拟机固件（OVMF） [97] 或 Oak stage 等精简固件0 [98]。AMD-SP 会将其中任一固件映像纳入启动摘要。Arm CCA 使用适用于 Arm虚拟机的 EFI开发工具包II（EDK2）固件（ArmVirtQemu） [99]。该固件会参与领域初始度量（RIM），并在领域中加载负载 [96]。

7. 相关工作

PoX 架构。 Flicker [36] 是一种早期的类 PoX 机制。它利用延迟启动和可信平台模块，将安全敏感代码与系统其他部分隔离开来。近期研究将 PoX 扩展到了低端微控制器 [32]–[35]。APEX [33] 是一种经过形式化验证的架构，可在软件完全攻陷的情况下为微控制器提供 PoX。其硬件设计会在可执行文件、输出或监控元数据遭到破坏或篡改时，使 PoX 证据失效。EVOLVE 基于 APEX 的 PoX 定义，在执行期间控制 EXEC 值。ASAP [32]将 APEX 扩展到异步软件，并支持使用中断的程序实现 PoX。PEARTS [34] 提出实时执行证明，并在实时调度下保持执行完整性。它还将 APEX 的概念引入使用 FreeRTOS 的设备。SLAPP [35]将 PoX 扩展到有状态执行，并防止嵌入式设备上的联邦学习和差分隐私遭到投毒。与这些设计不同，EVOLVE 通过软件在服务器级机密虚拟机上提供 PoX。EVOLVE 的编译器插桩和客户机虚拟固件中的监控器可保护 EXEC 并观测输出。

机器学习证明。 密码学方法无需依赖可信硬件扩展即可验证机器学习属性。安全多方计算（SMPC）可验证数据集属性，但每次证明都需要验证者参与 [3]。Arc 将安全多方计算与公开承诺相结合，以关联训练与推理 [100]。零知识证明可验证训练 [4], [5], [7],推理 [6], [8], [9],差分隐私 [101], [102],

和公平性 [103], [104]。然而，高昂的成本限制了它们的部署。有些方法验证单次训练迭代就需要数分钟 [5]。相比之下，Sentry 会在数据集和模型加载到图形处理器（GPU）内存时，根据签名摘要对其进行验证 [105], ，但不会证明它们在运行时的使用方式。基于TEE的方法提供了一种实用的替代方案 [1], [2], [12], [13], [35], [37]。Laminator 使用英特尔 SGX 证明分类器属性 [1]。Schnabl 等人证明代码、模型和推理输出 [37]。PAL*M 将属性证明扩展到支持可信执行环境的 GPU 上的大型生成模型 [2]。ExclaveFL 在联邦学习中证明训练输入和输出 [13],，而 Atlas 则证明机器学习生命周期中的工件转换 [12]。这些系统能够证明机器学习操作，但侧重于静态测量或特定环境。没有一种系统能为机密虚拟机中的机器学习工作负载提供操作级 PoX。

机密虚拟机内隔离。许多工作在机密虚拟机内构建隔离区。Veil [38], 、NestedSGX [39],、Cabin [40], 和 EXIA [42] 利用 AMD SEV-SNP 的虚拟机权限级别来承载飞地、限制不可信程序，或证明外部输入。Wallet [41] 使用它们在一个机密虚拟机内隔离无服务器函数，并证明每次调用的输入和输出。SHELTER [44] 利用 Arm CCA 的粒度保护表来隔离应用程序，而 CAEC [54] 扩展 CCA 的运行时管理器，以在领域之间共享内存。在没有虚拟机权限级别的 TDX 上，TETD [45] 信任宿主机虚拟机监控器来向 TD 内核隐藏一个代理。Erebor [43] 使用保护密钥和 CET，通过内核内监控器限制沙箱。COFUNC [46] 使用微内核承载隔离的无服务器函数。在机密虚拟机之外，BlackBox [106] 使用虚拟机监控器级监控器保护容器免受不可信宿主机的影响。这些系统依赖于特定平台，或依赖于不属于我们信任模型的软件。TETD 信任宿主机虚拟机监控器，而证明者可能控制该监控器；Cabin 则信任客户操作系统。TD 分区是 TDX 中最接近虚拟机权限级别的功能，但它要求在 TD 内运行嵌套虚拟机监控器，从而使每次 TD 退出的开销增加一倍 [107]。这些系统还隔离每个隔离区，而非单独的操作。因此，操作级隔离需要为每个操作设置一个隔离区。它们证明已加载代码，最多再证明其输入和输出，却无法审计究竟是哪段代码生成了某个输出。ExclaveFL [13] 将这一方法应用于机器学习属性证明，在单独的机密虚拟机中运行每个联邦学习任务，但也存在这些限制。相比之下，EVOLVE 在 TD 的客户机固件中运行，无需修改平台。它仲裁载荷的内存写入和间接调用，并将 EXEC 绑定到 I 和 O，以实现逐操作绑定。其策略检查使用形式化验证的原语。这一方法遵循了对可信监控器进行形式化验证的相关工作，例如 CCA 的运行时管理器 [108]。但是，EVOLVE 只验证这些原语，而不是整个监控器。

8. 讨论与未来工作

结合 EVOLVE与机器学习证明。EVOLVE与现有的机器学习工件证明或认证系统互为补充 [1], [2], [12], [37], [105]。这些工作证明机器学习模型或数据集是真实的，或满足特定属性，但假设执行环境能够正确执行使用它的计算。相比之下，EVOLVE能够证明特定操作确实执行过，并且所报告的输出仅由该操作生成，未遭到未经授权的修改。因此，EVOLVE可作为这些系统底层的执行证明层，并成为其证据的一部分。例如，Sentry [105] 可以在模型加载前对其进行认证，而后，EVOLVE证据可用于确立使用该模型的推理或训练操作的完整性。Laminator [1] 或 PAL*M [2]可以证明生成工件的更高层属性，而 Atlas [12] 可以在其生命周期溯源中记录这些证据。

适配内核载荷。EVOLVE 的当前设计假设使用一种直接运行在虚拟固件之上的裸机容器载荷。虚拟固件（例如 TD-Shim）也支持基于内核的部署，但要将 EVOLVE 扩展到此类部署，需要更复杂的设计。对应用程序和内核的 store 指令及间接调用进行插桩会带来很高的开销；此外，由于内核管理应用程序内存和页表，仅凭固定的区段边界已无法区分各项操作。一种可能性是借鉴飞地模型（例如英特尔 SGX [14]），在保持机密虚拟机内隔离的同时，于运行时将应用程序与内核隔离开来。要确保此类设计维持相同的安全保证，需要进行仔细验证，这将留待未来工作完成。

大型生成模型。EVOLVE 的当前版本兼容完全在机密虚拟机内执行的基于 CPU 的推理工作负载。未来的工作可以将 EVOLVE 扩展到 llama.cpp [109], 等大型语言模型推理框架，从而为提示处理和生成启用 PoX。然而，将 EVOLVE 扩展到现代生成式人工智能工作负载，需要解决类似于 PAL*M [2] 中的 GPU 管理挑战。尽管可以像 PAL*M 中那样将 GPU 直接分配给机密虚拟机，但负责保护 GPU 与载荷之间通信的 NVIDIA 驱动程序需要内核，这使得该方向与内核载荷紧密相关。此外，GPU 执行的内存写入对 EVOLVE 的插桩不可见，因此，要证明 GPU 侧执行的真实性，需要将监控器策略扩展到 GPU。

9. 总结

这项工作借助 EVOLVE，将执行证明扩展到嵌入式系统之外。EVOLVE 是一种面向机密虚拟机中机器学习工作负载的操作级执行证明系统。LLVM 插桩器会将载荷的写入操作和间接调用路由至信任域虚拟固件中的运行时监控器，而监控器会将其 EXEC 标志与输入和输出绑定，并纳入硬件证明报告。其策略检查依赖于经过 F* 形式化验证的原语。因此，验证者可以确认加载的是哪个载荷，以及请求的操作确实生成了相应输出，同时端到端开销微乎其微。

致谢

本研究部分由瓦伦堡访问教授计划、加拿大自然科学与工程研究委员会（资助编号 RGPIN-2026-04826），以及国家网络安全联盟和加拿大政府（CSIN）（项目编号 2024-1345）资助。

参考文献

[1] V. Duddu 等，“Laminator：利用硬件辅助证明实现可验证机器学习属性卡”，载于《第十五届 ACM数据与应用安全和隐私会议论文集》，2024年，第317–328页。[2] P. Chantasantitam 等，“PAL*M：大型生成模型的属性证明”，arXiv 预印本 arXiv:2601.16199，2026年。[3] V. Duddu 等，“证明机器学习训练数据的分布属性”，载于《欧洲计算机安全研究研讨会论文集》。Springer，2024年，第3–23页。[4] S. Garg 等，“训练零知识证明的实验研究”，载于《2023年 ACM SIGSAC 计算机与通信安全大会论文集》，2023年，第1880–1894页。[5] K. Abbaszadeh 等，“深度神经网络训练的零知识证明”，载于《2024年 ACM SIGSAC 计算机与通信安全大会论文集》，2024年，第4316–4330页。[6] T. Liu 等，“zkCNN：卷积神经网络预测与准确率的零知识证明”，载于《2021年 ACM SIGSAC 计算机与通信安全大会论文集》，2021年，第2968–2985页。[7] B. Roy 等，“zkLoRA：高效的 LoRA验证零知识证明”，arXiv 预印本 arXiv:2501.13965，2025年。[8] H. Sun 等，“zkLLM：大语言模型的零知识证明”，载于《2024年 ACM SIGSAC 计算机与通信安全大会论文集》，2024年，第4405–4419页。[9] C. Huang 等，“zkMLaaS：一种机器学习即服务的可验证方案”，载于《GLOBECOM 2022——2022年 IEEE 全球通信会议论文集》。IEEE，2022年，第5475–5480页。[10] R. Zhang 等，“学习证明的‘对抗样本’”，载于《2022年 IEEE 安全与隐私研讨会（SP）》。IEEE，2022年，第1408–1422页。[11] C. Fang 等，“学习证明目前的漏洞比你想象的更多”，载于《2023年 IEEE 第八届欧洲安全与隐私研讨会（EuroS&P）》。IEEE，2023年，第797–816页。[12] M. Spoczynski 等，“Atlas：机器学习生命周期来源追踪与透明性的框架”，载于《2025年 IEEE 欧洲安全与隐私研讨会研讨会（EuroS&PW）》。IEEE，2025年，第448–461页。[13] J. Guo 等，“ExclaveFL：利用隔离域为联邦学习提供透明性”，arXiv 预印本 arXiv:2412.10537，2024年。[14] Intel，“Intel软件防护扩展（英特尔 SGX）”，https://www.
i
nt
el.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html，2015年。[15] ——，“Intel信任域扩展（Intel TDX）”，https://www.in
t
el.c
om/content/www/us/en/developer/tools/trust-domain-extensions/overview.html，2023年。[16] AMD，“AMD安全加密虚拟化（SEV）”，https://ww w.amd.com/en/developer/sev.html，2020年。[17] PyTorch，“PyTorch代码库”，https://github.com/pytorch/pytorch，[访问于] 2025。[18] TensorFlow，“TensorFlow 代码仓库”，https://github.com/tensorflow/ tensorflow，[访问日期] 2025。[19] D. Cohen，“数据科学家成为带有静默后门的恶意 Hugging Face 机器学习模型的攻击目标”，https://jfrog.com/blog/data-scientist
s-targeted-by-
malicious-hugging-face-ml-models-with-silent-bac kdoor/，2024年。[20] B. Milanov，“利用 pickle 文件攻击机器学习模型：第 1 部分”，https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-1/，2024 年。[21] J. Zhao 等人，“模型即代码：旨在衡量预训练模型中心中的恶意代码投毒攻击”，载于《第 39 届 IEEE/ACM 自动化软件工程国际会议论文集》，2024 年，第 2087–2098 页。[22] NIST 国家漏洞数据库（NVD），“CVE-2025-32434 详情”，https://nvd.nist.gov/vuln/detail/CVE-2025-32434，2025 年。[23] ——，“CVE-2025-32434 详情”，https://nvd.nist.gov/vuln/detail/CVE-2026-24747，2026 年。[24] ——，“CVE-2025-53630 详情”，https://nvd.nist.gov/vuln/detail/C VE-2025-53630，2025 年。[25] ——，“CVE-2025-8747 详情”，https://nvd.nist.gov/vuln/detail/CVE-2025-8747，2025 年。[26] ——，“CVE-2022-459076 详情”，https://nvd.nist.gov/vuln/detail/C VE-2022-45907，2022 年。[27] ——，“CVE-2025-32444 详情”，https://nvd.nist.gov/vuln/detail/CVE-2025-32444，2025 年。[28] A. Shafran 等人，“重新路由大语言模型路由器”，arXiv 预印本 arXiv:2501.01818，2025 年。[29] Q. Lin 等人，“大语言模型路由器的生命周期路由漏洞”，arXiv 预印本 arXiv:2503.08704，2025 年。[30] H. Tang 等人，“通往罗马攻击：通过对抗性后缀优化将大语言模型路由器引向高成本模型”，载于《第 64 届计算语言学协会年会论文集》（第 1 卷：长篇论文），2026 年，第 44 333–44 347 页。[31] Y. Jiang 等人，“稀疏模型，稀疏安全：混合专家大语言模型中的不安全路由”，arXiv 预印本 arXiv:2602.08621，2026 年。[32] A. Caulfield 等人，“ASAP：协调简单嵌入式系统中的异步实时操作与执行证明”，载于《第 59 届 ACM/IEEE 设计自动化大会论文集》，2022 年，第 721–726 页。[33] I. D. O. Nunes 等人，“APEX：一种在软件完全攻陷情况下针对远程设备执行证明的经验证架构”，载于《第 29 届 USENIX 安全研讨会（USENIX Security 20）》，2020 年，第 771–788 页。[34] A. J. Neto 等人，“PEARTS：实时嵌入式系统中的可证明执行”，载于《2025 年 IEEE 安全与隐私研讨会（SP）》。IEEE，2025 年，第 3765–3782 页。[35] N. Rattanavipanon 和 I. D. O. N. Nunes，“SLAPP：通过有状态执行证明防止联邦学习和差分隐私中的投毒”，《IEEE 信息取证与安全汇刊（TIFS）》，2025 年。[36] J. M. McCune 等人，“Flicker：用于可信计算基最小化的执行基础设施”，载于《第 3 届 ACM SIGOPS/EuroSys 2008 年欧洲计算机系统会议论文集》，2008 年，第 315–328 页。[37] C. Schnabl 等人，“ 可证明审计：利用可信执行环境验证 AI 安全基准”，arXiv 预印本 arXiv:2506.23706，2025 年。[38] A. Ahmad 等人，“Veil：面向机密虚拟机的受保护服务框架”，载于《第 28 届 ACM 国际编程语言与操作系统体系结构支持会议论文集》第 4 卷，2023 年，第 378–393 页。

[39] W. Wang 等人，“信任之路：在机密虚拟机中构建飞地”，载于 *NDSS*，2025 年。[40] B. Mei等人，“Cabin：在机密虚拟机中隔离不可信程序”，载于国际信息与通信安全会议。Springer，2024 年，第 165–184 页。[41] P. Sabanic 等人，“Wallet：机密无服务器计算”，载于第 *23* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 26*），2026 年，第 1275–1302 页。[42] Z. Huang 等人，“EXIA：通过外部输入证明实现飞地的可信转换”，载于*NDSS*，2026 年。[43] C. Zhang 等人，“Erebor：用于在不可信机密虚拟机中进行私有数据处理的即插即用沙箱解决方案”，载于第二十届欧洲计算机系统会议论文集，2025 年，第 1210–1228 页。

[44] Y. Zhang 等人，“SHELTER：通过用户空间隔离扩展 Arm CCA，”载于第 *32* 届 *USENIX* 安全研讨会（*USENIX Security 23*），2023 年，第 6257–6274 页。

[45] Z. Wang 等，“TETD：信任域中的可信执行，”载于第 *34* 届 *USENIX* 安全研讨会（*USENIX Security 25*），2025 年，第 1187–1206 页。

[46] J. Shi 等人，《通过分裂容器实现机密且高效的无服务器函数》-发表于第*34*届 *USENIX* 安全研讨会（*USENIX Security 25*），2025年，第1091–1110页。

[47] 机密容器，《TD Shim规范（版本0.7）》，https://github.com/confidential-containers/td-shim/blob/main/doc/tdsh im spec.md, _2023年。

[48] N. Swamy 等，《F中的依赖类型和多单子效应》，载于第*43*届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集，R. Bod´ık 和 R. Majumdar 编，2016年，第256–270页。

[49] Arm，《面向Cortex-A的Arm TrustZone》，https://www.arm.com/techno logies/trustzone-for-cortex-a，2004年。

[50] A. Limited，《概览：Realms》，https://learn.arm.com/learning-p aths/servers-and-cloud-computing/cca-container/overview/。

[51] Intel，《Intel信任域扩展（Intel TDX）模块基础架构规范》，https://cdrdv2 8532 86 348549006 2025 -public.intel.com/ / intel-tdx-module-base-spec- .pdf，4月。

[52] ——，“Intel ECDSA 证明服务”，https://api.portal.trustedservices.intel.com/provisioning-certi fication。[53] AMD，《安全虚拟机架构参考手册》，https://www.0x04.net/doc/amd/33047.pdf，2005，使用 SKINIT 安全启动，第 3.1 章。[54] S. Abdollahi 等，“CAEC：使用 Arm CCA 实现机密、可证明且高效的跨机密虚拟机通信”，载于 *2026 IEEE* 第 *11* 届欧洲安全与隐私研讨会（*EuroS&P*）。IEEE，2026 年，第 789–803 页。[55] D. Kuvaiskii 等，“Gramine-TDX：面向机密虚拟机的轻量级操作系统内核”，载于 *2024* 年 *ACM SIGSAC* 计算机与通信安全大会论文集，2024 年，第 4598–4612 页。[56] H. Shacham，“无辜骨肉的几何：无需函数调用的返回至 libc（在 x86 上）”，载于 第 *14* 届 *ACM* 计算机与通信安全大会论文集，2007 年，第 552–561 页。[57] R. Roemer等，“面向返回的编程：系统、语言与应用”，*ACM* 信息与系统安全汇刊（*TISSEC*），第 15 卷，第 1 期，第 1–34 页，2012 年。[58] T. Bletsch 等，“面向跳转的编程：一种新型代码复用攻击”，载于 第 *6* 届 *ACM* 信息、计算与通信安全研讨会论文集，2011 年，第 30–40 页。

[59] S. Chen 等，“非控制数据攻击是现实威胁”，载于 *USENIX* 安全研讨会，第 5 卷，2005 年，第 146 页。[60] L. Cheng 等，“利用现有及潜在防御方法实施面向数据攻击的技术”，*ACM* 隐私与安全汇刊（*TOPS*），第 24 卷，第 4 期，第 1–36 页，2021 年。[61] L. Wilke 等，“TDXdown：针对 Intel TDX 的单步执行与指令计数攻击”，载于*2024* 年 *ACM SIGSAC* 计算机与通信安全大会论文集，2024 年，第 79–93 页。[62] F. Rauscher 等，“TDXploit：针对 Intel TDX 的单步执行与缓存攻击新技术”，载于 第 *34* 届 *USENIX* 安全研讨会（*USENIX Security 25*），2025 年，第 1207–1222 页。[63] W. Lee等，“T-Time：针对 Intel TDX 的细粒度计时型受控通道攻击”，载于 欧洲计算机安全研究研讨会。Springer，2025 年，第 323–341 页。[64] U. Mandal 等，“探索 Intel 信任域扩展中的侧信道”，密码学电子印本档案，2025 年。[65] N. Mishra 等，“‘OOPS!’：针对 Intel SGX 和 TDX 的带外远程功耗侧信道攻击”，载于 *2025* 年第 *62* 届 *ACM/IEEE* 设计自动化大会（*DAC*）。IEEE，2025 年，第 1–7 页。[66] J. Chuang等，“TEE.fail：通过 DDR5 内存总线插桩攻破可信执行环境”，载于 *2026 IEEE* 安全与隐私研讨会（*SP*）。IEEE，2026 年，第 4408–4426 页。[67] E. Aktas 等，“英特尔信任域扩展（TDX）安全评审”，*Google* 安全评审，2023 年。[68] C. Lattner 和 V. Adve，“LLVM：用于持续程序分析与转换的编译框架”，载于*2004* 年国际代码生成与优化研讨会（*CGO 2004*）。IEEE，2004 年，第 75–86 页。[69] A. J. Mashtizadeh等，“CCFI：以密码学方式强制实施控制流完整性”，载于 第 *22* 届 *ACM SIGSAC* 计算机与通信安全大会论文集，2015 年，第 941–951 页。[70] M. Orenbach 等，“CoSMIX：用于飞地安全内存插桩与执行的基于编译器的系统”，载于 *2019* 年 *USENIX* 年度技术会议（*USENIX ATC 19*），2019 年，第 555–570 页。[71] C. Tice等，“在 GCC 和 LLVM 中强制实施前向边控制流完整性”，载于 第 *23* 届 *USENIX* 安全研讨会（*USENIX Security 14*），2014 年，第 941–955 页。

[72] Y. Kim 等人，“*值得*在安全*堂*，载于 *2020* 年第*53*届*IEEE/ACM*国际微体系结构研讨会（*MICRO*）。IEEE，2020年，第1153–1166页。

[73] S. Nagarakatte 等，“SoftBound：适用于 C 语言的高度兼容且完整的空间内存安全”，载于 第 *30* 届 *ACM SIGPLAN* 编程语言设计与实现大会论文集，2009 年，第 245–258 页。[74] T. Palit 等，“通过保护内存驻留的敏感数据来缓解数据泄露”，载于 第 *35* 届年度计算机安全应用大会论文集，2019 年，第 598–611 页。[75] D. Sehr 等，“使软件故障隔离适应当代 CPU架构”，载于第 *19* 届 *USENIX* 安全研讨会（*USENIX Security 10*），2010 年。[76] K. Braden 等，“面向移动设备的抗泄露布局随机化”，载于*NDSS*，第 16 卷，2016 年，第 21–24 页。[77] L. C. Infrastructure，“LLVM语言参考手册”，https://llvm.org/docs/LangRef.html，[访问日期] 2026。[78] ——，“LLVM用户指南：使用新通道管理器”，https: // llvm.org/docs/NewPassManager.html，[访问日期] 2026。[79] Intel Corporation，*Intel 64* 和 *IA-32* 架构软件开发人员手册，第 *1* 卷：基本架构，Intel Corporation，2026 年，第 18 章：控制流强制技术（CET）。

[80] F. Bellard 等，“QEMU：快速且可移植的动态翻译器”，载于 Usenix ATC，Freenix 专题，2005 年，第 41–46 页。[81] M. Kelly 等，“UCI机器学习库”，https: //archive.ics.uci.edu。[82]W. Wolberg 等，“威斯康星乳腺癌（诊断）”，UCI机器学习库，1993 年，DOI：https://doi.org/10.24432/C5DW2B。[83] R. Cole 和 M. Fanty，“ISOLET”，UCI机器学习库，1991 年，DOI：https://doi.org/10.24432/C51G69。[84] D. Chapman 等，“Musk（版本2）”，UCI机器学习库，1994 年，DOI：https://doi.org/10.24432/C51608。[85] “图像分割”，UCI机器学习库，1990 年，DOI：https://doi.org/10.24432/C5GP4N。[86] R. A. Fisher，“Iris”，UCI机器学习库，1936 年，DOI：https://doi.org/10.24432/C56C76。[87] J. Nordby 等，“emlearn：面向微控制器和嵌入式设备的机器学习推理引擎”，https://github.com/emlea rn/emlearn，[访问日期] 2026。[88] T. Shnitzer 等，“使用基准数据集进行大语言模型路由”，arXiv 预印本 arXiv:2309.15789，2023 年。[89] N. Guha 等，“Smoothie：无标签语言模型路由”，《神经信息处理系统进展》，第 37 卷，第 127 645–127 672 页，2024 年。[90] M. Chrapek 等，“机密大语言模型推理：CPU 和 GPU 可信执行环境中的性能与成本”，载于 2025 IEEE 工作负载特征国际研讨会（IISWC）。IEEE，2025 年，第 84–98 页。[91] FStarLang，“Karamel github 仓库”，https://github.com/FStarLang/karamel，[访问日期] 2026。[92] Arm，“了解架构：为复杂软件提供保护”，https://support.arm.com/documentation/102433 0200, / 访问[ ] 2026日期。[93] J. Allen，“为 SVM 启用影子栈虚拟化”，https://lwn.net/Articles/1032872/，2025 年。[94] LLVM项目，“Clang 控制流完整性”，https://clang.llvm.o rg/docs/ControlFlowIntegrity.html，[访问日期] 2026。[95] AMD，“SEV 安全嵌套分页固件ABI规范”，https://www.amd.com/content/dam/amd/en/documents/developer/60 [ ] 2026 568.pdf，访问日期。[96] Arm，“Realm管理监控器规范（DEN0137）”，https: //developer.arm.com/documentation/den0137 [ ] /latest，2026访问日期。[97] TianoCore，“OVMF：开放虚拟机固件”，https://github .com/tianocore/tianocore.github.io/wiki/OVMF，[访问日期] 2026。[98] Project Oak，“Oak stage0固件”，https://github.com/project-oak /oak/tree/main/stage0, [访问日期] 2026。[99] S. Mujawar，“面向 Arm CCA 的EDKII客户机固件”，https://support.arm.com/documentation/den0127/300/Arm-CCA-reference-software-stack/EDKII-guest-firmware-for-Arm-CCA，[访问日期] 2026。[100] H. Lycklama 等，“让秘密承担责任：隐私保护机器学习审计”，载于 USENIX 安全研讨会，2024 年。[101] A. S. Shamsabadi 等，“Confidential-DPproof：差分隐私训练的机密证明”，载于国际学习表征大会，第 2024 卷，2024 年，第 4029–4044 页。[102] O. Franzese 等，“差分隐私协作学习的安全噪声采样”，载于 2025 ACM SIGSAC 计算机与通信安全大会论文集，2025 年，第 4649–4663 页。[103]——，“在线公平性的安全机密证书”，《神经信息处理系统进展》，第 38 卷，第 40 077–40 107 页，2026 年。[104] A. S. Shamsabadi 等，《Confidential-PROFITT：树的公平训练机密证明》，发表于第十一届国际学习表征会议，2022年。[105] A. Gan 和 Z. Ghodsi，《Sentry：实时验证机器学习制品》，发表于 *2025*年 *ACM SIGSAC* 计算机与通信安全大会论文集，2025年，第3550–3563页。[106] A. Van’t Hof 和 J. Nieh，《BlackBox：用于保护不可信操作系统上容器的容器安全监视器》，发表于第*16*届 *USENIX*操作系统设计与实现研讨会（*OSDI 22*），2022年，第683–700页。[107] J. T. Lim 和 J. Nieh，《利用直接虚拟硬件优化嵌套虚拟化性能》，发表于第二十五届国际编程语言与操作系统体系结构支持会议论文集，2020年，第557–574页。[108] X. Li 等，《Arm机密计算架构的设计与验证》，发表于第*16*届 *USENIX*操作系统设计与实现研讨会（*OSDI 22*），2022年，第465–484页。[109] ggml-org，《llama.cpp代码仓库》，https://github.com/ggml-org/lla ma.cpp，[访问于] 2026。

附录 A. 基于EVOLVE的PoX协议

图 14 展示了Vrf 与Prv之间基于 EVOLVE 的 PoX 协议。Prv 由主机上的不可信 VMM 和 QE、可信 TDX 模块，以及 EVOLVETD 组成；EVOLVETD 分裂为载荷区域和容纳 EVOLVE 监视器的垫片区域。

离线阶段。 在发出任何请求之前，先使用 EVOLVE 的 LLVM 处理遍对应用程序进行插桩（回顾第 4 节），生成$\texttt{App}_{\texttt{Instr}}$，其中每次非栈内存写入和间接调用都会经由监控器执行（步骤 1）。此步骤在每次部署时执行一次。

加载时。VMM 指定并创建 TD（步骤 2）。TDX 模块对 TD 固件和 EVOLVE 监控器进行度量。随后，固件对$\texttt{App}_{\texttt{Instr}}$进行度量，这些共同构成 h_{TD}（步骤 3）。启动时，监控器记录所有区段的地址边界（I、O、监控器、EXEC、每个操作对应的一个区段，以及最初未绑定的共享区段），并将 EXEC_M 和 EXEC_I 设为 1（步骤 4）。

正在接收请求。Vrf 通过向Prv发送操作 op、输入 I 和证明质询*Chal* 来启动协议。VMM 将请求转发至信任域（步骤 5）。EVOLVE监视器存储 I 并验证 op；无效的 op会将 EXEC_B 设置为 0，且不会运行任何内容（步骤 6）。否则，监控器会授权 op 区段（因此在此次执行期间只有该区段可以写入 O），将共享区段绑定到该区段，将 EXEC_B 设置为 1，并调用 op（步骤 7）。

执行。 有效载荷运行于 op，所有插桩内存写入和间接调用都会被路由至监控器（步骤 8）。监控器根据策略检查每个事件（步骤 9）。对于关键数据的写入，EVOLVE 会阻止这些写入并清除 EXEC_M。EVOLVE 监控器还会在从 op 或其共享区段外部写入 O 时清除 EXEC_M。除允许的 FFI 入口点之外，进入垫片区域的间接调用会被阻止并清除 EXEC_I；而进入另一个操作或未绑定共享区段的间接调用会继续执行，但会清除 EXEC_I。所有其他事件均继续执行。

协议 1： 启用 EVOLVE 的 PoX

记号
– I、 O：有效载荷操作的输入和输出。– op、挑战值：由 Vrf选择的操作选择和挑战随机数。– h_{TD}：TD 固件（包括监控器）和插桩有效载荷的度量值。– EXEC：四个执行标志（EXEC_B、EXEC_F、EXEC_M、EXEC_I）；– TDREPORT / QUOTE：由 TDX 模块生成的经 MAC 认证的报告 / 由 QE 签名的引用。

离线。
1 App 由 LLVM 处理遍插桩，将每个非影子栈 store 指令替换为 ffi_mem_write，并将每个间接调用替换为 ffi_indirect_check，从而生成 $\text{App}_{\text{Instro}}$

加载时。
2 VMM 指定 TD 映像并调用 TD 创建。3. TDX 模块测量 TD 固件（包括监控器）；随后，固件测量 $\text{App}_{\text{Instr}}$，共同构成 h_{TD}，并启动它。4. 监控器注册各区段（I、 O、监控器、EXEC、每个操作对应一个区段，以及未绑定共享区段），并设置 $\text{EXEC}_M = \text{EXEC}_I = 1$。

请求时间。
5 Vrf 将 $\{op, I, Chal\}$ 发送给 VMM，VMM 再将其转发给 TD。6. 监控器存储 I 并验证 op；无效的 op 会设置 $\text{EXEC}_B = 0$，不会执行任何内容。7. 监控器授权 op 的区段，将共享区段绑定到该区段，设置 $\text{EXEC}_B = 1$，并调用 op。

执行。
8 载荷执行请求的操作；每次插桩内存写入和间接调用都会路由至监控器。9. 监控器根据策略检查每个事件：– 对 I、监控器或 EXEC 的写入会被阻止，并清除 EXEC_M；– 从 op 或其已绑定的共享区段外部对 O 的写入会继续执行，但会清除 EXEC_M；– 除允许的 FFI 入口点之外，进入垫片区域的间接调用会被阻止，并清除 EXEC_I；– 进入其他操作或未绑定共享区段的间接调用会继续执行，但会清除 EXEC_I；– 所有其他事件继续执行。

执行后。
10 当 op 返回时，监控器设置 $\text{EXEC}_F = 1$，解除共享区段的绑定，并请求带有 REPORTDATA：

$= \{I,\ O,\quad ,\quad \}$ EXEC$Chal$的

11 TDREPORT。TDX 模块创建带 MAC 的 TDREPORT。12. 引用 enclave 验证 MAC 并签署 QUOTE，VMM 将其发送给Vrf。13. Vrf 验证 QUOTE 签名和 h_{TD}，检查 $Chal$，解包 EXEC，并验证 I 和 O。

图 14：启用 EVOLVE 的 PoX 协议。

生成 QUOTE。 当 op返回时，监控器将 EXEC_E 设置为 1，解除共享区段的绑定，并向 TDX 模块请求 TDREPORT，同时将 REPORTDATA 设置为 $=\{I, O,$EXEC、$Chal\}$（步骤 10）。TDX 模块生成带消息认证码的 TDREPORT（步骤 11）。引用 enclave 验证消息认证码并签署 QUOTE，然后由 VMM 将其发送至Vrf （步骤 12）。

验证。Vrf 验证 QUOTE 签名，以确认其源自预期的 TD；Vrf 还会检查$Chal$ ，以确认其时效性（步骤 13）。随后，它会将 h_{TD}与已知参考值进行比较，以确认所用固件和插桩二进制文件符合预期。最终检查是 EXEC。Vrf 解包 EXEC 以获取四个标志。如果它们全都等于 1，Vrf 即可确信：请求的操作已启动并完成， O 仅由授权操作写入，并且执行期间未发生策略违规。如果其中任何一个等于 0，则无论 O的值如何，结果都不可信。