可重复的容器

奥马尔·S·纳瓦罗·莱亚 [omarsa@seas.upenn.edu](mailto:omarsa@seas.upenn.edu)

宾夕法尼亚大学 凯利·希普托斯基 [kship@seas.upenn.edu](mailto:kship@seas.upenn.edu)

宾夕法尼亚大学 瑞恩·G·斯科特 [rgscott@indiana.edu](mailto:rgscott@indiana.edu)

印第安纳大学 王宝军 [wangbj@gmail.com](mailto:wangbj@gmail.com)

印第安纳大学 尼古拉斯·雷纳

宾夕法尼亚大学 [nrenner@seas.upenn.edu](mailto:nrenner@seas.upenn.edu)

瑞恩·R·牛顿

印第安纳大学 [rrnewton@indiana.edu](mailto:rrnewton@indiana.edu)

约瑟夫·德维埃蒂 宾夕法尼亚大学 [devietti@cis.upenn.edu](mailto:devietti@cis.upenn.edu)

# 摘要

我们描述了 DetTrace 的设计和实现，这是一 种在用户空间中实现的 Linux 的可重现容器 抽象。在 DetTrace 容器内发生的所有计算都 是容器初始文件系统状态的纯函数。可重现容器 可用于多种用途，包括用于容错的复制、可重现 的软件构建和可重现的数据分析。我们使用 D etTrace 以自动方式实现 12,130 个 De b ian 包构建的可重复性，其中包含超过 8 亿行 代码，以及生物信息学和机器学习工作流。我们 表明，虽然这些领域中的软件最初是不可重现的 ，但 DetTrace 带来了可重现性，而无需任何硬 件、操作系统或应用程序更改。DetTrace 的性 能取决于系统调用的频率：IO 密集型软件构建 的平均开销为 3.49 × ，而受计算约束的生物 信息学工作流程低于 2%。

# 简介

在数据处理上下文中，将每个输入重复映射到唯一 的确定性输出通常很重要。确定性在软件构建 [1 、2]、可重现数据分析 [3、4] 和容错分布式系统 [5-8] 中很有用。然而，尽管之前在确定性语言 [9 – 11] 和操作系统 [12 – 14 ]，在 实践。因此，我们寻求一种实用的容器抽象来隔 离正在运行的软件并针对明确分隔的输入数据 执行它，从而实现端到端的可重现数据处理。对 于可部署性，还必须为商品硬件和软件提供这种 保证。 先前关于确定性操作系统的工作既不需要也不 足以满足我们对可重复数据处理的定义，因为需 要额外的封装来确保程序以相同的状态启动， 而不会出现系统时间或标识符（如 pid）的差异 。例如，Determinator [12] 没有提供可重复 的确定性时间概念。dOS [14] 提供了确定性的 进程组抽象，并且可以记录和重放与时间相关的 系统调用。但是 dOS 还使用记录和重放进行文 件系统交互，将文件系统置于“确定性框”之外 。因此 dOS 无法确定必须包含文件 I/O 的数 据处理作业。 最终，确定性也是一个比我们想要的更弱的属 性——确定性保证了在给定机器上重复运行的相 同结果，但在这项工作中，我们寻求跨机器的相 同结果（第 7.3 节），我们称之为可重复性的 属性。在本文中，我们描述了 DetTrace 系统 ，它朝着x86-64 Linux 程序的可重现容器抽象。容器内 运行的所有代码都被强制以可重现的方式运行， 而无需更改任何源代码。DetTrace 封装了 Li nux 进程树及其执行的 IO，并在商品硬件和库 存 Linux 发行版上运行。DetTrace 运行时使用 Linux 命名空间、绑定挂载和 ptrace 工具的 组合来拦截具有不可重现语义的系统调用和 x 86 指令。虽然 DetTrace 支持进程级并行性， 但进程中的线程当前是序列化的。虽然许多先前 的确定性执行系统支持线程级并行性，但我们专 注于为复杂的多进程工作负载提供强大的容器实 现。 DetTrace 导出容器内的进程树原本会看到的相 同 POSIX API — 在每种情况下，我们只需从众 多行为中选择一个有效行为以确保可重复性。 例如，DetTrace 提供了一个可重现的时间概念 ，因此由 stock tar 实用程序添加到存档的时 间戳（源于类似时间的系统调用）因此是可重现 的。通过在系统调用和 ISA 级别强制执行再现 性，我们可以透明地将再现性导出到所有更高级 别，包括任意语言 VM。 本文做出以下贡献： ∙ We present the design of DetTrace, the first repro- ducible 容器抽象，在用户空间运行并 支持未修改的程序。 ∙ We give the first taxonomy of the sources of irre- Linux 系统调用和 x86-64 指令中的可 生产性。对于我们不处理的来源，我们描 述了这样做所涉及的挑战。 ∙ We use DetTrace to run bioinformatics workflows, 训练 TensorFlow 模型，并可重复地构建 12,130 个 Debian 软件包，包括大型软 件包，如 llvm、clang 和 blender。默认 情况下，该软件的大部分运行不可重现， 但 DetTrace 能够使其可重现。 ∙ We show that DetTrace’s performance overhead 与给定工作负载中的系统调用频率相关： 例如，计算密集型进程并行生物信息学工作 流的开销低于 2%，而系统调用密集型软件 构建的开销平均为 3.49 倍。

# 2 为什么重现性很重要？

可重复性为软件开发带来了许多优势。可重复 性在调试过程中至关重要；无法重现的错误更 难修复。在分布式系统中，可再现性确保所有副 本的行为方式相同，从而加速共识 [ 15 ] 并启用透明故障恢复 [ 6 ]。可重复性 在一系列软件领域中也有更具体的好处，我们将 在接下来进行探讨。

## 可重现的构建。

按位可重现的构建具有许多优势 。由于构建工件缓存中的更多命中，构建可以更 快地运行，并且构建可以自信地分布，因为知道 相同的工件将在集群的任何节点上生成。可重现 的构建还提高了软件完整性，增强了给定二进制 文件源自特定源代码版本的信心。由于这些原因 ，许多 Linux 发行版在 Debian Reproduci ble Builds (DRB) [ 2 ] 努力的推动下，都 以所有包的按位可再现性为目标。Microsoft 正在寻求可复制的软件构建 [16]，并在其 C# 和 VB 编译器 [17] 中提供支持。Googl e 的 Blaze/Bazel 构建系统 [1] 鼓励可重 现的构建生态系统，以防止由于不可重现性而导 致的虚假更改在 Google 的统一内部软件存储 库中导致大量额外的下游重建。

为了实现可重现性，软件构建工具链的每一部 分都需要可重现：预处理器、编译器、构建过程 中使用的脚本等等。例如，为了处理 tar 为 tarball 中的每个文件记录的时间戳，tar 被扩展为 --clamp-mtime 标志 [18] 强制这些时间戳 为固定值。然后需要打包和分发修改后的 t ar 程序，并更新构建脚本以使用新标志，然后 才能实现可再现性。 一次敲打一个不可复制的痣是可以预见的费力 的。在第一年之后，DRB 在 5,151 个受支持的 包中的 3,193 个可重现地构建 [19]，两年后 有 18,800 个可重现的包（共 21,782 个）。 经过两年的开发，我们已经实现了 12,130 个受支持包年龄的 100% 的可重复性。1 目前 ，经过数十位贡献者 5 年多的努力，目前 De bian 软件包的 5.2% （总共 1,289 个）仍然无法重现。虽然存在用于识 别不可再现性来源的工具 [20]，但修复构建仍然 是一个手动过程。即使 DRB 达到 100% 的可重现 性，也需要保持警惕，以确保错误的代码更改不会重 新引入不可重现性。

## 计算科学。

具有讽刺意味的是，虽然重现结果是科 学方法的基石，但许多计算科学工具却不可重现 。虽然化学反应和生物体本质上是可变的，但没有 充分的理由让计算表现出类似的行为。计算科学 的可重复性将加速科学进步，因为 科学家们可以更轻松地分享、复制和建立彼此的工 作。提高科学结果的可重复性是资助机构的一个重 点[21]，并且可以在我们社区中日益增长的人工制 品评估运动中看到。我们发现一个常见的生物信息 学工具是不可重现的（第 6.1 节）。

## 机器学习。

人们对可重现机器学习 (ML) [2 2] 的兴趣日益浓厚。可再现性使审计模型能 够了解他们做出某些决定的原因。它还可以更 容易地查看性能变化是否归因于，例如，有意 识的设计变化或偶然的随机性，如训练集的抽 样。我们将 DetTrace 应用于流行的 Tens orFlow 框架，该框架众所周知是不可重现的 [23, 24]

# 3 可重现的容器

在这项工作中，我们旨在提供可重现的容器抽象 。容器本身被指定为初始文件系统状态和要运行 的程序（来自文件系统）。该程序可以反过来启动 其他程序，例如，如果它是一个外壳程序。容器中 运行的程序可能会尝试执行任意 x86 指令和 Linux 系统调用，但我们不保证所有此类尝试都 会成功。在我们最初的原型中，容器化代码只能与 其文件系统和容器中同时运行的其他程序交互。 然而，在未来，我们设想允许有限形式的外部交互 ，如果它们保持可重复性，例如，下载具有已知校 验和的文件。

我们的可再现性目标可以分解为两个子属性：确 定性和可移植性。对我们来说，确定性是数据流确 定性 [25]，这意味着，在给定的机器上，每次读 取都会在每次运行时返回相同的值。这隐藏了诸 如时间和显式随机性等不可再现性的来源。确定 性意味着许多有用的属性：所有进程完成后的文 件系统状态将是相同的，打印到标准输出和标准 错误的消息也是如此。严格来说，由于无法确定的 外部错误的可能性，例如磁盘空间不足，我们的保 证是准确定性之一 [26]：任何两次运行要么是 数据流确定性，要么至少有一次运行崩溃到外部 故障。 可移植性意味着数据流确定性也可以跨机器 扩展，具有不同的微架构或操作系统版本。我 们的容器总是通过报告一个简单的 x86-64 单处理器和 Linux 4.0 内核来隐藏这些细 节。实际上，我们的容器只能抽象出有限数量 的硬件或操作系统细节：在 ARM 微控制器上 运行时，我们不会模拟 x86-64 芯片。DetT race 还需要某些硬件和操作系统支持提供这种抽象，尤其是 至少一个 Intel Ivy Bridge 处理器和 Li nux 4.12。DetTrace 可以在较旧的处理器和 Linux 版本上运行，但可移植性保证较少（第 5.8 节）或性能较低（第 5.11 节）。DetTra ce 还提供了前向兼容性的衡量标准。 虽然未来的 Linux 版本可能会引入新的不可 生产的 API，DetTrace 将支持这些 API，但 今天使用现有 Linux API 的软件无法访问这 些，因此如果软件今天与 DetTrace 一起使用 ，它将在未来保持可重复性。 最终，DetTrace 容器作为容器配置和初始文件 系统状态的纯函数运行。文件内容影响计算，但 文件元数据仅部分可见。仅文件的 mtime 变化 的两次运行将产生相同的输出，但权限更改会影 响输出。图 1 说明了什么构成输入，即在 Det Trace 计算中可以引起输出变化的因素。

现有的容器技术（如 Docker）不提供可重复性 ：它们既不是确定性的也不是可移植的，因为主 机操作系统和处理器微架构的许多细节在容器 内是直接可见的。虚拟机提供了更强的硬件抽 象，但缺乏确定性，而且重量也很大。我们相信 ，DetTrace 可重现容器抽象为构建和测试可 重现性至关重要的软件等领域的现有方法提供 了显着优势。

4 再现性要求 Linux 和 x86-64 在我们的用户空间可重现容器中运行的代码可 以访问两个主要接口：x86-64 指令集和 Linu x 系统调用 API。因为我们对容器中的代码没有 任何限制，它可以包含任意指令并尝试任意系统 调用。受 Popek 和 Goldberg 虚拟化要求 [ 27] 的启发，这些要求定义了提供虚拟机抽象的 要求，我们定义了再现性要求集。我们分析每个 记录在案的 x86-64 ISA 指令 2 和系统调用 ，以查看它是否可能是不可再现性的来源，如果 是，在什么条件下。特别重要的是识别接口的关 键成员——那些允许不可重现但在执行过程中不 能可靠地检测到的成员。任何关键指令或系统调 用都可能默默地引入不可再现性。 我们使用 ptrace 意味着我们可以看到从容 器中进行的所有系统调用，因此没有潜在的关 键系统调用（我们还处理 vDSO 调用，参见第 5.3 节）。如果给定的系统调用是 ir 可重复 性的来源，则有许多潜在的缓解措施：包装系 统调用或完全用确定性对应物（如时间调用） 替换它，将其转换为 nop（如睡眠调用），或不 支持它并引发（可重现的）容器级错误。 在最新的 x86-64 指令集 [29] 中有许多不可再 现性的来源。特权指令通常是不可重现的，但会在 我们的用户级容器中引发异常。一些不可重现的 用户级 x86-64 指令虽然可能，但很难捕获。rd rand 和 rdseed 从硬件熵源返回随机位，并且可 以通过 VT-x 扩展在管理程序级别捕获，但不能 从环 0 捕获。有时可以从用户空间访问像 rdp mc（从性能计数器读取）这样的指令但可以通过适 当的内核设置配置为导致陷阱。 一些浮点指令，如 cvtsd2si（将双精度数转换 为整数）被记录为具有某些指令编码的“跨不同 处理器代的不可预测的行为”。我们还没有调查 这种行为的程度，但是，通过承诺可移植性，它 是不可再现性的潜在关键来源。 TSX 不可重复性。最终，我们只发现了一组绝对 关键的指令：用于事务内存和锁省略的 TSX 指令。一个事务可以由于多种原因中止，其中一 些——比如时间中断的到来——是 2 有一些未记录的 x86-64 指令 [28]。处理 这些将是未来工作的有趣途径。 高度不可复制。程序可以通过使用 xbegin 指令 注册的中止处理程序监视自己的中止，并因此执 行不可重现的计算。虽然可以通过制作 cpuid 的返回值来隐藏 TSX 的存在，但无效或对抗性 程序可以忽略 cpuid 并运行这些指令。尽管英 特尔的微码更新禁用了先前有缺陷的 TSX 版本 [30]，但我们不知道有任何捕获 TSX 指令执行 的能力，这表明软件可配置性在某种程度上确实 存在。捕获关键指令的硬件支持对于有效和完整 的检测是必要的，因为硬件确切地知道程序正在 执行什么指令。在对抗性程序中检测 xbegin 的存在是不切实际的：该程序可能会跳转到其他 有效指令的中间，或者使用自修改代码来混淆其 行为，超出静态二进制分析的范围。动态分析或 仿真原则上可以捕获此类行为，但只能以高昂的 运行时成本进行。 因为当前的硬件不允许我们的 DetTrace 原型 捕获所有不可重现的指令，我们依赖于表现良好 的程序不会执行非法或丢失的指令（即尊重 cp uid 的输出）。尽管如此，我们对 Linux 系统调 用和 x86-64 指令的描述是实现 100% 可重现 容器的有用标准，这些容器甚至可以抵抗对抗性 程序。 5 DetTrace 设计 DetTrace 将轻量级沙盒容器与系统调用拦截 相结合，以实现任意 Linux 程序的可重复性 强制执行。DetTrace 实现了这一功能，同时满 足了我们的设计目标：纯软件用户空间解决方 案，支持未修改的二进制文件，不需要特权（r oot）访问，并且不需要记录和重放。DetTrac e 使用标准的 Linux 容器功能：用户、PID 和挂载名称、绑定挂载和 chroot。这些机制有 助于将容器中的程序与其外部的程序和文件隔 离开来。 DetTrace 使用 ptrace 拦截容器中运行的代 码进行的所有系统调用。Linux ptrace 机制允 许一个进程（跟踪器）监视另一个进程（跟踪器 ）的执行。跟踪器可以拦截被跟踪者的系统调用 （在它们到达内核之前和返回到被跟踪者之前） 、信号等等。跟踪器还可以读取和写入跟踪内存 和寄存器。由于跟踪器是它自己的进程，因此它 与跟踪器故障隔离良好（反之亦然）。但是，每次 拦截的事件都需要额外的上下文切换才能跳转 到跟踪器。在 DetTrace 中，具有可重现语义的 系统调用

被允许通过，而具有不可重现效果的那些要么 被可重现地包装，要么被标识为不受支持，从而 触发运行时错误。 接下来我们详细介绍不可再现性的来源，并描 述 DetTrace 如何使每一个都具有可再现性 。为简单起见，我们使用术语“用户进程”来指 代在 DetTrace 容器内运行的进程。 5.1 进程、用户和组 ID 由于我们的进程命名 空间，我们容器内的进程接收到独立于容器外 部世界的唯一 PID。用户进程不能命名容器外 的任何进程。由于用户进程的创建和终止是确 定性的，并且 Linux 在每个命名空间中顺序 分配 PID，容器内的 PID 自然是确定性的。 我们同样利用 uid 和 gid 命名空间来达到 类似的目的。第一个用户进程以 root 权限开 始，并且可以通过 setuid 更改身份。 5.2 操作系统生成的随机性 Linux 用户进程可 以通过 getrandom 系统调用或从特殊的 /de v/random 或 /dev/urandom 文件中读取来向 操作系统请求随机性。DetTrace 拦截 getran dom 系统调用，并用简单的 LFSR 伪随机数生成 器生成的值填充指定的用户缓冲区。同样，/dev /random 和 /dev/urandom 是命名管道，DetTrace 从我 们的 PRNG 中写入值。PRNG 种子可以在调用 DetTrace 时指定，以受控方式引入“真正的 随机性”。用户进程还可以通过 x86-64 指令 rdrand 和 rdseed 获得随机性，稍后将在 5.8 节中讨论。 出于安全原因，某些应用程序需要真正的随 机性。DetTrace 可以提供这样的应用程序 可以直接访问例如真实的 /dev/urandom 并可 选择记录读取的值以保持可重复性。 5.3 时间和时钟 各种系统调用返回某种形式 的时间信息。对于直接报告挂钟时间的系统调用 （如 gettimeofday），DetTrace 报告而不是可 重现的逻辑时间值。对于逻辑时间，DetTrace 使用用户进程执行的时间调用次数的计数。这 确保了时间在调用之间单调前进，这对于一些检 查计时行为的用户程序很重要。 为了启用高分辨率计时，Linux 使用虚拟动态共 享对象 (vDSO) 机制来实现计时系统调用，例如 gettimeofday。出于性能原因，这些系统调用被 实现为库调用，因此不会被 ptrace 拦截。虽然 Linux 的 LD PRELOAD 机制是拦截库调用的自 然选择，但它在小而重要的方面是不完整的。首先 ，它不支持静态链接的二进制文件。其次，进程可 以在其地址空间内找到vDSO库（通过getauxval ），并直接调用vDSO函数；事实上，libc 在它的 mk stemp 函数中就是这样做的。为了确保对 v DSO 调用的密封拦截，DetTrace 在每次 exec ve 系统调用之后，将 vDSO 库代码替换为我们的 实现，其中每个 vDSO 函数都进行直接系统调用 ——通过 ptrace 适当地拦截。此外，我们使 vva r 页面不可读，以禁止对 vDSO 计时调用使用的 原始非确定性数据进行任何访问。虽然用普通系 统调用替换 vDSO 调用会导致性能损失，但我们 计划扩展我们的 vDSO 库以在 DetTrace 的未 来版本中直接处理计时调用。

x86 rdtsc 指令以当前循环计数的形式返回时 序信息。幸运的是，rdtsc 可以被捕获和重复模 拟，参见第 5.8 节。文件系统时间戳是我们在 5.5 节中讨论的时间信息的最终来源。使用非确 定性并行性，竞速线程可以重新创建高分辨率时 钟，但我们的确定性调度使这种情况变得毫无意 义[31]。

5.4 信号和定时器 信号是不可再现性的主要 来源，因为它们的到达通常是异步的。原则上， 信号生成和传递可以通过可重现的逻辑时钟完 全重现，就像确定性共享内存同步 [32]。但是 ，我们还没有发现这对于我们当前的工作负载 是必要的。相反，DetTrace 为 Linux 信号的 子集提供了可重复性。首先，DetTrace 不支持 在用户进程之间发送信号。然而，重要的是用户 进程可以向自己发送信号。一些这样的

信号自然是可重现的：SIGSEGV、SIGILL 和 SIGABRT 就像“精确异常”一样，在定义明确的 可重现状态下停止程序执行。通过系统调用（ 如警报）请求的定时器是另一种常见的自信号 源。为了使计时器到期可重现，DetTrace 中的 计时器“即时”到期，并在适当时调用信号处理 程序。 我们将生成信号的计时器调用（如警报）转换为 阻止用户进程的暂停系统调用。然后，跟踪器向 用户进程发送必要的信号，并在适当的情况下调 用已注册的信号处理程序。这会导致暂停调用返 回，并且用户进程恢复执行。计时器调用永远不 会到达操作系统，而是由跟踪器模拟。

5.5 文件和目录 由于复杂的 API 和广泛的元数 据，文件和目录是不可再现性的丰富来源。我们为 文件和目录提供可重现抽象的第一步是隔离用户 进程拥有的主机文件系统的视图，通过 chroot 系统调用完成。DetTrace 也可以嵌套在 Dock er 等标准容器中，以提供与主机更强的文件系统 隔离。文件和目录的所有权和权限是 DetTrace 计算的输入（图 1）。Linux命名空间控制着命名 空间内部的uid/gid到宿主机上的uid/gid的映射 ；此映射也是 DetTrace 输入的一部分。默认情况 下，我们将当前用户帐户映射到容器内的 root ，并将所有其他用户帐户映射到nobody/nogrou p。返回目录条目的顺序由文件系统实现控制。为 了使 getdents 系统调用可重现，DetTrace 在 将目录条目返回给用户进程之前按名称对其进行 排序。read 和 write 系统调用具有不可重现的 语义，因为它们可能读取/写入的字节数可能比请 求的字节数少。虽然在实践中我们从未见过对常 规文件进行这样的“部分”操作，但它们确实在访 问管道时经常出现。为了使这些系统调用在所有 情况下都可重现，DetTrace 会自动重试部分读取 和写入，直到它们处理请求的字节数，或者读取返 回 EOF。这是通过递减用户进程程序计数器以重 新运行系统调用指令并调整参数来实现的，例如 ，告诉当前读取在前一次读取结束的地方继续。 索引节点是文件系统挂载中文件或目录的唯一标 识符。stat 系列系统调用向用户进程报告 ino de，并且仅仅报告一个固定值是不够的，因为许多 用户进程比较 inode 值以快速识别相同的文件 。相反，DetTrace 维护一个来自真实（不可重现） 的映射

inode 到可重现的虚拟 inode。需要特别注 意确定何时创建新文件 𝑓，因为操作系统可能 会为 𝑓 回收真正的 inode，但 DetTrace 必须分配新的虚拟 inode 以保持可重复性 （参见文件时间戳讨论，下一个）。 文件时间戳为用户进程提供了一个时间概念， 未经过滤，可用于重建不可重现的时钟。因此， DetTrace 虚拟化了文件时间戳。在 Linux 上 ，每个文件或目录都有三个关联时间：最后一次 内容修改时间 (mtime)、最后一次访问时间 (atime) 和最后一次内容或元数据修改时间 (ctime)。在 DetTrace 中，我们总是将 ati me 和 ctime 报告为 0。但是，我们发现在许多 程序中总是返回一个固定的 mtime 值会违反 健全性检查。例如，从 GNU Autotools 配置通 过创建一个新文件来检查时钟偏差，然后将其 mtime 与现有文件的 mtime 进行比较，如果 mtime 没有意义，则会引发错误。 DetTrace 实现了真实 inode 和虚拟 mtime 之间的映射，允许来自报告 mtime 的系统调用 （如 stat）的可重现但合理的响应。每当用户进 程打开文件时，在 open 调用到达内核之前，我 们会检查指定路径中是​​否存在文件。在 ope n 调用返回容器中的进程之前，我们通过检查 /proc 文件系统来识别底层的真实 inode，以 获取新创建的文件描述符的路径和真实 inod e。通过在打开调用到达操作系统之前和之后检 查路径，我们可以可靠地识别何时创建新文件。 如果文件是新创建的，我们将其 mtime 分配为 当前虚拟 mtime，并递增当前虚拟 mtime。否则 ，该文件存在于初始容器映像中，我们为其分配 虚拟 mtime 0。写入文件当前不会更新其虚拟 mtime，因为我们在工作负载中没有发现这有必 要，但是可以很容易地添加它以提供更逼真的虚 拟时代。

对于 stat 调用，我们参考我们的真实 ino de → virtual mtime map 以适当地报告 mtime。表中没有条目的任何 inode 的虚拟 mtime 都为 0，因为它必须作为初始容器映像 的一部分存在。我们懒惰的 inode 映射群为 容器中的每个文件分配了可重现的虚拟 ino de 和 mtime，同时避免了在启动时索引整个 容器镜像的需要。

5.6 Reproducible Scheduler DetTrace

通过对系统调用执行进行排序来支持多个并发 进程，并允许进程并行运行以进行其他操作。我 们的跟踪器在系统调用、进程生成和进程退出时 做出调度决策。 DetTrace 实现了一个可重现的调度器，它由 三个队列组成。并行队列包含

当前并行运行的进程，其他队列包含当前需要调 度以进行顺序系统调用执行的进程。如图 3 所 示，进程在 并行队列。当并行队列前面的进程需要进行系统 调用时，它会移动到 Runnable 队列的后面。R unnable 队列头部的进程被允许执行下一个系 统调用，如果这个系统调用没有阻塞则进程返回 到 Parallel，如果阻塞则进程移动到阻塞队列 的末尾并重新访问之后。然后查询阻塞队列前面 的进程以查看其系统调用是否仍将阻塞，并相应 地移动到并行或返回阻塞。 5.6.1 阻止系统调用。可能阻塞的系统调用在 DetTrace 的顺序系统调用执行中表现出潜在的 死锁。DetTrace 通过提前（当然，可重复地）识 别系统调用是否可能阻塞来避免死锁。在来自进 程 𝑝 的任何给定的潜在阻塞系统调用 𝑠 上， 𝑠 可以立即成功，或者 𝑝 必须等到另一个进程 中的某个事件使 𝑠 完成。如果是前者，我们执行 𝑠，将 𝑝 移动到 Parallel 集合，将其从它所 在的队列中移除，然后并行恢复 𝑝。如果是后者 ，我们通过将𝑝移动到阻塞队列来抢占它。 为了检测系统调用是否会阻塞，我们将阻塞调 用转换为非阻塞调用，例如，将 wait4 调用修 改为使用 WNOHANG 标志。当非阻塞系统调用 返回并指示资源不可用时，我们抢占进程并将 其移动到阻塞队列的末尾。我们重置进程状态 以在将来重试系统调用。 一些系统调用，例如对管道的写入，可能会解 除对一个或多个其他进程的阻塞。我们不跟踪 流程之间的这种依赖关系；当进程 𝑝 写入管 道时，我们并不确切知道哪些阻塞进程 （如果有的话）这将取消阻止。但是，因为调度程序 在 Runnable、Blocked 和 Parallel 进程上进 行了相当大的迭代，所以任何未阻塞的进程最终 5.7 线程 线程和进程的 ptrace API 是相 同的，允许 DetTrace 支持对调度程序的扩 展很少的线程。进程内的线程被顺序化以呈现 可重现的共享内存交互。 futex 系统调用是 Linux 对快速用户空间锁 的实现。我们将 futex 等待调用视为任何其 他阻塞系统调用（第 5.6.1 节）。如果线程忙 于等待而不是阻塞，我们的顺序调度程序无法 取得进展，这是程序可能与 DetTrace 不兼容 的原因之一（第 5.9 节）。 5.8 CPU 指令 虽然无法通过 ptrace 拦截不可再现的 CPU 指令，但最近的 x86 硬件提供了拦截许多不可 再现指令的机制（第 4 节）。我们当前的 DetT race 实现拦截 rdtsc 和 rdtscp 指令，它们 通过 prctl 系统调用返回当前周期的计数。对 于 rdtsc[p]，我们用迄今为止执行的 rdtsc [p] 指令的线性函数覆盖它们的不确定结果。 其他不可重现的指令包括 TSX 指令、rdran d、rdseed 和 cpuid。意外的是，后者为前者提 供了一个解决方案：我们使用 cpuid 拦截来 报告 TSX 和硬件随机性支持的缺失，如第 4 节所述 （虽然对抗性程序可以尝试以任何方式运行它 们，但支持此类程序不是我们的目标）。虽然 管理程序早就能够拦截 cpuid，但在 tel 的 Ivy Bridge 微架构中引入了一种 rin g 0 机制，Linux 内核（从 4.12 开始）将其 导出到用户空间。 使用 Ivy Bridge 或更新的机器，我们可以在 重新运行作业时实现前向可移植性：固定报告的 系统信息，同时支持后续处理器。我们还简化了 呈现给用户进程的硬件细节，例如列出单个内核 和规范缓存大小。这进一步增加了机器的等价类 ，它们必须为一项工作观察相同的答案。 较旧的 Intel 架构，例如 Sandy Bridge， 缺少用户空间 cpuid 拦截，但它们也缺少 rdrand 和 TSX。因此，DetTrace 仍然可以在 这些旧机器上重复运行，但可移植性保证的范 围要小得多，因为我们无法隐藏 cpuid 信息 。

图 4. 为了使读取系统调用可重现，DetTr ace 重试不返回请求的字节数的读取操作。 实线箭头表示用户进程认为已经发生了什么 。虚线箭头表示 DetTrace 为提供再现性假 象而进行的额外操作。 5.9 不支持的操作这里我们描述了当前Det Trace原型的一些限制。如果用户进程尝试使用 这些功能之一，DetTrace 会引发错误。第 7. 1.1 节更详细地评估了由于这些原因而无法构 建的 Debian 软件包的数量。 DetTrace 不支持忙等待线程，因为我们的调度 程序仅在线程创建/退出和系统调用时为线程执 行上下文切换。也不支持套接字，因为任意套接 字用于网络通信是一个重大的可重复性挑战。 我们计划研究有限形式的套接字通信，例如，作 为我们容器内的进程间通信，可以重现。 5.10 系统调用修改 DetTrace 使用 ptrac e 拦截但随后跳过某些系统调用，例如 DetT race 内部模拟的计时器调用（第 5.4 节）。 虽然不能使用 ptrace 直接跳过系统调用， 但可以通过在内核检查之前替换系统调用号来 间接跳过它。我们将时间用作方便的“NOP”系统 调用，它不带参数并且总是成功。 我们可以利用系统调用拦截来任意修改、重放或 注入新的系统调用。作为一个更复杂的示例，图 4 说明了我们执行的系统调用注入，当用户进 程执行读取系统调用时请求 8 个字节，但内核 最初只返回 7 个字节。DetTrace 调整读取参 数以填充用户缓冲区剩余字节并重置 PC 以执 行另一次读取。一旦用户缓冲区已满（或者我们 到达 EOF），用户进程就可以继续通过 read 调用，第一次尝试时缓冲区似乎已填满。 有时系统调用要求我们在跟踪地址空间中 分配内存。例如， utime 系统调用为给定路径的文件设置 ati me 和 mtime。如果时间被指定为空，那么内核 将 atime/mtime 设置为当前时间。为了避免内 核设置不可重现的时间戳，DetTrace 需要在被 跟踪地址空间中分配一个时间戳结构，使用可重 现的时间戳进行初始化，并以该结构作为参数调 用 utime。为此，DetTrace 在每次 execve 系统调用之后在每个被跟踪对象的地址空间中 分配一页内存。我们的自定义时间戳结构是从这 个页面分配的，以避免干扰被跟踪者的堆或堆栈 。 5.11 使用 seccomp-bpf 提高性能 默认情况 下，ptrace 会为每个系统调用两次停止跟踪， 但 Linux 的 seccomp-bpf 机制允许选择性系 统调用拦截，避免在我们的环境中自然可重现的 系统调用的开销（如 getcwd）。seccomp 还允许 在系统调用到达内核之前运行的拦截代码在系 统调用完成后动态决定是否进行拦截，从而进一 步减少开销。Linux 内核版本 > = 4 。8 通过 提供单个事件而不是单独的预系统调用和 se ccomp 事件来额外优化上下文切换。我们支持内 核版本 < 4 。8 通过回退到较慢的实施。 6 实验方法 我们使用 Debian 7 运行我们的包构建评估 (Wheezy) 包，2013 年 5 月首次发布的稳定 版本，总共包含 17,145 个包。我们选择这个 版本的 Debian 是为了避免 Debian Repro ducible Builds 项目的努力造成的混淆影响 ，该项目始于 2013 年底。我们想要捕捉 De bian 软件包生态系统的 DRB 前的准确图景 我们在 CloudLab c220g5 节点上构建我们 的包，其中每个节点都有两个 Intel Xeon Silver 4114 Skylake 处理器，每个处理 器有 10 个内核（20 个线程），运行频率为 2.2GHz，内存为 192GB。这些处理器支持拦截 cpuid 指令（第 5.8 节）。我们使用完整的 seccomp-bpf 优化。每个节点都运行带有 对于我们的生物信息学工作流程，我们使用支持 AVX 的 RAxML 8.2.10 [33]、-ALIGN 模式下 的 Clustal 2.1 [34] 和 HMMER 3.1b2 [35 ]。我们在 ML 实验中使用了 TensorFlow v1. 14，使用 alexnet 和 cifar10 教程 [36] 来 执行模型创建、训练和推理。生物信息学和 ML 工作负载在具有两个 Intel Xeon E5-2618Lv 3 (Haswell) 处理器的机器上运行，每个处理器 具有 8 个内核（16 个线程），运行频率为 2.3 18.10 与 Linux 4.18。

6.1 验证重现性 包构建。我们在新的 Docker 实例中构建带有 和不带有 DetTrace 的包，以轻松控制文件系 统状态。3 在容器内部，我们使用 DRB 项目中 的 reprotest 实用程序版本 0.7.8 [ 37 ] 的略微修改版本。reprotest 构建每个包两 次，改变每个构建的条件以加剧不可重复性。我 们配置 reprotest 以改变环境变量、构建路径 、ASLR、CPU 数量、时间、用户组、主目录、语言环 境、执行路径和时区。我们关闭了域主机、内核 和文件排序，因为我们运行构建的旧版本 De bian 不支持它们。同样，umask 变体将随机化 文件权限，DetTrace 不会对用户进程隐藏这些 权限。 默认情况下，reprotest 随机选择变体；我们对 其进行了修改，为所有包的第一次构建使用一致 的配置，为所有第二次构建使用不同的一致配置 ，以便向 DetTrace 提供与基线完全相同的环境 。 我们创建一个最小 Wheezy 安装的 contro l-chroot，通过 apt-get source 下载源代 码，然后通过 apt-get build-dep 安装包的 （引用磁盘镜像以避免网络请求并确保跨构建 的一致性）。最后我们复制 control-chroot 来创建一个 Experiment-chroot，从而保证 两个构建的起始镜像相同。reprotest 使用这 些启动 chroot 来运行带或不带 DetTrace 的 dpkg buildpackage。使用 DetTrace 时，dpkg-buildpackage 所做的一切都在 D etTrace 下运行，包括编译、运行测试（如果包 被配置为这样做）和创建最终的 .deb 包。两 个构建完成后，重新抗议通过两个 .deb 包的 按位比较来验证再现性。reprotest 调用另一 个 DRB 工具 diffoscope，它比较两个目录， 检查按位相同的内容。如果 diffoscope 报告 没有差异，则认为该包装是可再现的，否则认为 该包装是不可再现的。 在这个 Debian/reprotest 配置下，有 15,7 61 个或 91.9% 的可用软件包完全构建，而 3 0 分钟后有 40 个超时和 1,344 个构建失败 。对于下一节的评估，我们将重点放在基线中构 建的 15,761 个包的集合上，无论是可重现的还 是不可重现的。事实上，在股票 Wheezy 系统，由于 tar 嵌入了时间戳，可重复构 建零包。所以我们调整我们的驱动程序脚本，使用 dpkg-deb 解压 deb 包，然后在单个文件上运 行 strip-nondeterminism [38]， 3 DetTrace 还可以提供隔离的文件系统环境， 但是 Docker 在我们的集群中提供了简单的图像 分发。DetTrace 嵌套在 Docker 中没有问题。 剥离时间戳。最后，diffoscope 可以进行有 意义的逐位比较。DetTrace 构建不需要这种 解决方法，因为它们对时间戳自然是健壮的。 使用 tar-timestamp 解决方法，3,803 (24.1%) 包装可在库存 Wheezy 系统中重 现。其他 11,958 个包需要额外的人工干 预才能实现重现性。 生物信息学。虽然我们没有利用生物信息学工具 的重新抗议等不利的不可重现的环境，但对 H MMER 和 RAxML 的输出使用 hashdeep 揭示 了在单台机器上连续运行的不可重现性。我们确 认（使用 hashdeep）在 DetTrace 下运行时消 除了不可再现性。clustal 工作流程在本地和 使用 DetTrace 时似乎都是可重现的。 机器学习。为了检查我们的 TensorFlow 工作 负载的可重复性，我们记录了训练过程中每一步 的损失函数的值。不出所料，这些值在本机运行 时是不可重现的，即使使用序列化的 TensorF low（参见第 7.6 节），例如由于训练集的随机化 。DetTrace 使这些工作负载无需任何代码更改 即可重现。 7 评价 在本节中，我们使用 DetTrace 系统和软件构建 、生物信息学和机器学习应用程序来描述我们的 结果。 7.1 包构建再现性包构建在 DetTrace 下构 建时可以属于四个类别之一。如第 6.1 节所述 ，某些包构建是可重现或不可重现的。 超时包不会在 2 小时内完成构建。 我们为 DetTrace 分配了较高的超时时间 ，以考虑其性能开销，并避免在我们的性能评 估中忽略严重的减速。最后，由于我们在第 7.1.1 节中讨论的各种原因，可能不支持一 个包。 在 DetTrace 支持的 12,130 个包中（即，使 用 DetTrace 的构建既不受支持也不会超时） ，DetTrace 能够呈现每个包的可重现性。这代 表了来自 DetTrace 下构建的超过 330 万个 源文件的超过 8 亿行（非注释/非空白）代码。 表 1 显示了从基线移动到 DetTrace 时包状 态如何变化，反之亦然，仅关注那些在基线中构 建（可重现或不可重现）的包。上表显示了使用 DetTrace 运行时基线包会发生什么。例如，第一 行显示在基线中不可重现的 11,958 个包龄中 ，其中 8,688 个被 DetTrace 自动呈现为可重 现的。

令人欣慰的是，在基线中可重现的包在 DetT race 下永远不会变得不可重现。 表 1 中的底部表格显示，对于具有给定 D etTrace 状态的包，基线中发生的情况。超 时或 DetTrace 不支持的包在基线中通常 是不可重现的，这表明这些是更复杂的构建 。 7.1.1 不受支持的软件包。由于已知的 Det Trace 限制，总共有 1,912 个包龄未能构建 。最常遇到的问题是忙于等待，因为 876 个 Java 包（45.8% 的失败）未能构建。下一个最 常见的原因是套接字操作（302 个包，15.8% ）和发送进程内信号（79 个包，4%） 其余形成 了 DetTrace 尚不支持的各种系统调用的长 尾。其他忙等待的情况会导致超时。 7.1.2 与 DRB 的比较。在 DetTrace 下可重现 的包中有 407 个在 DRB [39] 的当前拉伸版本 中被确定为不可重现。虽然这些软件包比我们使 用的 Wheezy 版本更新，但 DRB 的努力也对这 些软件包不可复制的原因进行了分类。常见原因 包括构建工件中捕获的构建路径、文件中嵌入的 时间戳以及影响构建工件的随机性。尽管这些问 题已在数百个其他软件包中得到解决，但每个软 件包都需要分析不可重现的原因并让维护人员接 受补丁。相反，DetTrace 会自动使构建不受此类 变化的影响。 7.1.3 与 Mozilla rr 的比较。Record-and -replay (RnR) 系统与 DetTrace 类似，需要 拦截非确定性的来源。然而，记录和回放系统并 不能直接促进可重现的构建，因为不透明的记录 文件无法检查包的源代码。录音也需要存储，通 常比纯源代码要多得多。我们使用 rr 工具的 最新版本 (5.2.0) 进行了一个小型实验，因为 它是我们所知道的最强大的 RnR 系统。我们选 择了 81 个包裹 在 Ubuntu 18.04 中从源代码构建（为 rr 提供比 Debian Wheezy 更现代的构建环境） ，并尝试使用 rr 构建它们。不幸的是，由于不 支持 ioctl 调用的已知错误，rr 在其中的 46 个上崩溃了。在使用 rr 构建的 35 个包 中，平均运行时开销为 5.8 ×（范围为 3.3- 22.7 ×），与 DetTrace 相当。与 RnR 不同， DetTrace 避免了不透明的记录，并提供了从 输入到输出的人类可读的审计跟踪。 7.2 包构建正确性 为了验证 DetTrace 的 功能正确性，我们使用了几个使用我们的系统 构建的包来确保它们正常工作。例如，我们使 用 DetTrace 构建了流行的 3D 图形包 b lender，将生成的 .deb 安装在 Debian wheezy 虚拟机上，并使用 UI 渲染示例项目 。 我们使用 DetTrace 构建了核心 TeX/ LaTeX 包，并使用它们来构建您正在阅读 为了在复杂的软件系统上验证 DetTrace 的正确性，我们首先从源代码构建 LLVM 3 .0 编译器，而不使用 DetTrace。我们通过 make 检查运行 LLVM 的测试套件，发现 5 ,594 个测试通过，48 个预期失败，15 个在 此基线配置中不受支持。然后，我们在 Det Trace 下运行 LLVM 构建（也使用使用 De tTrace 构建的 clang 版本）并收到相同的 测试结果。鉴于 LLVM 源代码的复杂性，我们 发现“自托管”LLVM 的这些结果令人鼓舞， 证明使用 DetTrace 构建的软件可以正确运 7.3 包构建可移植性 为了评估 DetTrace 的可移植性，我们在具有不同微架构和操作 系统版本的两台不同机器上执行包构建。一台 机器是我们的标准 CloudLab 节点（在第 6 节中描述），另一台具有运行 Ubuntu 18. 10（而不是 18.04，Linux 版本分别为 4.18 和 4.15）。 我们使用相同的基于 reprotest 的构建方法 来扰乱环境，并确保每个构建都基于每个machine 产生一个按位相同的包。由于时间 限制，我们随机选择了 1,000 个可通过 D etTrace 重现的包。每一个都在两个系统中 构建相同。 实现这些包的可移植性需要对 DetTrace 进 行一个扩展。我们发现目录的大小（由 stat 返回）因机器而异，尽管目录内容相同，是通过 从同一个 tarball 中提取创建的，并且文件 系统类型和块大小相同。这种行为在我们之前 使用单一机器类型的任何实验中都没有出现， 从经验上说明了可移植性和确定性之间的区别 。DetTrace 通过将大小报告为目录条目数量 的确定性函数来实现可重现的目录大小。

7.4 包构建性能 DetTrace 旨在实现可重复性 ，但仅针对性能开销进行了适度优化。考虑到总 体构建，DetTrace 总共会导致挂钟时间减慢 3.49 倍。图 5 显示了 860 个随机选择的 D etTrace 支持包的散点图，显示了 DetTrace 在基线上的减速 （对数刻度）相对于构建的每秒系统调用率（ 由 DetTrace 测量）。我们排除了在基线中运 行时间少于 5 秒的构建，并且我们仅在每台 机器上运行一个包构建以避免性能干扰。我们 从图中裁剪了一些异常值以使其更易于阅读 ：4 个包执行超过 25,000 次系统调用/秒（ 最大值为 82,533 个）并表现出 3.97-30. 11 × 的减速，以及 3 个运行大约两倍的包使 用 DetTrace 比在基线中快——尽管它们似乎 构建正确，例如，它们的内部测试在构建结束 时全部通过。 图 5 中的浅橙色点显示不使用线程的包，而 深蓝色点显示线程包。总体而言，DetTrace 开销与系统调用率之间存在正相关关系。尽管 此示例中只有 76 个线程包，但由于常见的 futex 操作从阻塞转换为非阻塞，它们表现出 一些最慢的速度。 我们发现系统调用在包构建中很频繁，平均构 建中超过 800,000 次（表 2）。 我们还在所有包装中发现了许多潜在的不可重 复性来源。加载器 ld 使用 rdtsc 指令进行 内部分析，并使用 libc 为 gcc 生成临时文 件名。gcc 也从 /dev/urandom 生成唯一的符号名称。 7.5 生物信息学工作流程 我们的三个生物信 息学工作流程使用流程级并行机制来提高性能 ，并表现出一系列开销 0K 5K 10K 15K 20K 25K system calls per second 1 2 5 10 20 50 DetTrace slowdown (x) 图 5. DetTrace 开销（y 轴，对数刻度）很大 程度上取决于执行系统调用的速率 （x 轴）。使用线程的包（深蓝色点）通常比不使 用线程的包（浅橙色点）慢。 clustal native DT hmmer native DT raxml native DT 1 4 16 1 4 16 1 4 16 1 4 16 1 4 16 1 4 16 1 2 3 4 5 6 7 8 speedup over sequential native 1.00 1.98 4.24 0.85 2.01 4.17 1.00 2.96 7.46 0.66 2.24 4.78 1.00 2.76 6.88 0.29 0.86 1.11 图 6. 加速生物信息学工作流程 1、4 和 16 个并行进程，标准化为顺序本机 执行（越高越好）。深蓝色条是本机执行，浅橙 色条是 DetTrace。 使用 DetTrace，由它们受计算约束的程度决 定。图 6 显示了每个工作负载通过更多并行进 程观察到的加速，标准化为顺序本机执行。高度 计算绑定的集群表现最好，可以很好地扩展额 外的进程，并且在 16 个进程中表现出低于 2% 的开销。相比之下，hmmer 和 raxml 以高 出 19 倍的速度执行系统调用（平均超过 55 ,000/秒），从而导致更多的序列化。raxml 尤 其频繁地写入标准输出，这可能是阻塞操作， 对于 DetTrace 来说成本更高，导致 16 个进 程的 6.2 × 开销。hmmer 有更多的非阻塞系 统调用，可以实现更好的扩展，16 个进程的开 销仅为 1.56 倍。

7.6 TensorFlow 我们以三种配置运行 alexnet 和 cifar10 程序，每一种都只在 CPU 上运行：1）本机并行 ，2）本机但 TensorFlow 配置为使用单线程， 3）使用 DetTrace。由于 TensorFlow 通过 OpenMP 使用线程级并行，DetTrace 的序列化 线程在 16 核上的本机并行执行导致大幅减速 ：在 alexnet 上慢 17.49 倍，在 cifar10 上慢 11.94 倍。然而，与序列化的本机执行相 比，DetTrace 的表现要好得多，速度分别降低 了 1.51 × 和 1.08 × ，这表明 DetTrace 对非线程计算绑定工作负载的性能代价很小。 8 相关工作DetTrace 独特的可重现容器抽象从许多以前 的系统中汲取灵感。我们将之前的工作分为记 录和回放系统和确定性执行系统。 学术界 [40 – 46] 和工业界 [47 – 49] 都提 出了许多记录和回放 (RnR) 系统。这些系统记 录一个非确定性执行的跟踪，以启用该执行的后 续重放，通常用于调试目的。这些系统具有与 DetTrace 大致相似的拦截要求，因为系统调用 是必须记录在跟踪中的不可再现性的主要来源 。DetTrace 借鉴了 Mozilla 的 rr [47] 的 一些实现技术，因为它还依赖于 ptrace（与 r r 的定量比较见第 7.1.3 节）。许多 RnR 系统 以多线程工作负载为目标，因为这些工作负载在 没有 RnR 支持的情况下非常难以调试，并提供 高性能的并行记录和回放。 确定性执行方案在程序执行期间强制执行确定 性。确定性操作系统解决了我们在本文中描述的 几个系统问题，提供了确定性版本的操作系统抽 象，如进程和线程。而 Determinator [12] 为 确定性提供了新的操作系统抽象 fork-join 并行性，DDOS [13] 专注于本地 网络交互，dOS [14] 在提供确定性进程组抽 象方面更接近我们的工作。dOS 中的 shim 抽象与 Linux 的 ptrace API 相似。与 D etTrace 不同，dOS 支持线程和进程的并行 执行。然而，dOS 使用 RnR 进行文件系统交互 ，将其确定性抽象的边界定义得太窄，对于与 文件系统进行广泛交互的软件构建来说是有 用的。更一般地说，自定义操作系统是执行确 定性计算的重量级先决条件，而现有的确定性 操作系统尚未评估跨不同微架构的可移植性 。 其他确定性执行方案侧重于单个多线程进程，通 过共享内存确定交互。一些方案针对任意二进制 程序 [50 – 58]，以适度的性能开销提供通用性 。其他方案利用语言支持为 Haskell [10, 26 , 59 – 62] 或 Java [9, 63] 程序提供确定性 。无论是语言无关的还是特定的，这些方法都消除 了线程调度的影响，但不能确定与底层操作系统 和文件系统的 IO 交互。因此，它们的保证范围太 小，无法用于可重现的构建。一个例外是 DetFl ow [11]，它为批处理作业提供确定性并行执行， 尽管它缺乏强大的系统调用拦截并且需要用 H askell 编写的协调器层。 9 结论我们已经描述了 Det Trace 的设计和实现， 它提供了一种新的可重现容器抽象。DetTra ce 自动为软件构建、生物信息学处理和 ML 工作流程提供可重复性，而无需对硬件、操作 系统或应用程序代码进行任何更改。为了促进 对 DetTrace 的进一步试验，我们计划在发布 时开源其代码。 致谢这项工作得到了美国国家科学基金会的资助 ，编号为 #1703541。本材料中的观点或发现是 作者的观点，不一定反映 NSF 的观点。