# eAudit：快速、可扩展且可部署的审计数据收集系统

## 文章信息

| | |
|---|---|
| **Title** | eAudit: A Fast, Scalable and Deployable Audit Data Collection System |
| **Journal** | () |
| **Authors** | Sekar R,Kimm Hanke,Aich Rohit |
| **Pub.date** | |

## 摘要

当今的高级网络攻击活动通常可以绕过所有现有的保护措施。针对它们的主要防御措施是事后检测，然后进行取证分析以了解其影响。此类分析需要忠实捕获每个主机上的所有活动和数据流的审计日志（也称为溯源日志）。虽然 Linux 审计守护进程 (auditd) 和 sysdig 是最流行的审计数据收集工具，但由研究人员和从业人员编写的许多其他系统也可用。通过一项激励性的实验研究，我们表明这些系统会产生很高的开销，使工作负载减慢 2 倍到 8 倍；在持续的工作负载下丢失大部分事件；并且容易受到日志篡改的影响，日志篡改会在日志条目提交到持久存储之前将其删除。我们提出了一种克服这些挑战的新方法。通过依赖最新 Linux 版本中内置的扩展伯克利数据包过滤器 (eBPF) 框架，我们避免了对内核代码的更改，因此我们的数据收集器可以在大多数 Linux 发行版上开箱即用。我们提出了新的设计、调整和优化技术，使我们的系统能够承受比现有系统造成重大数据丢失的工作负载高一个数量级的工作负载。此外，我们的系统只产生以前系统的一小部分开销，同时大大减少了数据量，并将日志篡改窗口缩小约 100 倍。

## 研究背景 & 研究目的

# 介绍

我们正处于一个涉及“高级和持续威胁”（APT）的长期网络攻击活动的时代[65]，[9]。这些活动由老练的参与者实施，他们优先考虑秘密行动而不是其他目标，这些活动通常在数月内未被发现[75]，[22]，[76]，[7]，[71]。在此期间，攻击者仍然隐藏在受害者的网络中，同时跨主机移动、安装恶意软件并收集数据。

由于 APT 的隐蔽性，针对它们的主要手段是事后检测，然后进行取证分析以了解其全部影响。此类分析需要日志忠实地捕获企业中主机上的所有重要系统活动。虽然应用程序日志（例如 Web 服务器日志）很有用，但它们并不完整，因为它们没有涵盖系统上所有应用程序的活动。例如，没有涵盖恶意软件活动或攻击者在远程登录会话中执行的活动的应用程序日志。因此，有必要收集涵盖所有应用程序的系统范围日志。关于 APT 调查的最新研究工作 [40]、[38]、[31]、[70]、[41]、[91]、[100]、[98]、[23]、[101]、[36] 、[8]、[60]、[13]、[79] 依赖于大致在系统调用级别运行的系统审计日志。这些日志启用的粗粒度溯源跟踪可确保取证分析的完整性，即分析不会错过任何攻击的影响。

对于倾向于基于Linux的服务器系统，Linux审计守护进程auditd是审计数据收集的明显选择[32]、[55]、[44]。不幸的是，auditd 在系统调用粒度数据收集方面会产生非常高的开销，从而使程序速度降低 5 倍或更多。这促进了对更高性能审计数据收集系统的研究[80]、[64]、[78]、[62]、[59]。然而，这些系统需要修改操作系统内核，这从部署角度来看是一个挑战。生产系统的操作员不愿意对内核代码进行更改，因为这会影响系统稳定性和/或引入兼容性问题。此外，这些日志收集系统的实现通常与特定的内核版本相关，增加了对更新和升级选项的担忧。

除了这些可移植性和可部署性问题之外，我们还发现现有的审计数据收集系统还存在严重的数据丢失和性能问题：

- 数据丢失：在中度到重度负载下，大多数现有系统都会丢失很大一部分事件。即使在单核工作负载上也会发生这种情况。数据丢失随着多核负载成比例增加，导致大部分数据丢失的情况。这违背了审计数据收集的主要目的，即确保捕获攻击活动和后续影响之间的每个链接。
- 高开销：即使在可以处理的负载下运行，当今的日志收集系统也会产生高开销，使工作负载减慢 2 倍到 8 倍。
- 日志数据篡改漏洞[77]：在APT攻击中，攻击者可以获得足够的权限来篡改审计系统。典型的防御是将数据记录到攻击者无法到达的远程主机上。如果数据立即发送到远程主机，即使攻击者能够控制攻击完成后记录的所有记录，攻击的证据也会保留在日志中。不幸的是，我们表明现有系统可以在数据输出之前在内存中缓冲数十万条日志记录。成功的攻击可以擦除这些缓冲区，从而从日志中删除所有攻击证据。
- 数据量大：现有系统每台主机每天产生的日志量从几GB到数百GB不等。存储如此多数

据的成本可能令人望而却步，因为许多 APT 活动是在几个月后才被发现的 [75]、[22]、[76]。

在本文中，我们提出了一种新的审计日志收集方法来克服这些挑战。它已被实施到一个轻量级且易于部署的系统中，称为 eAudit。下面，我们总结了我们的方法和贡献。

eAudit 基于最新 Linux 版本中内置的扩展伯克利数据包过滤器 (eBPF) 框架。该框架支持在 Linux 内核中定义的各种挂钩上部署安全探针，例如 Linux 跟踪点和 LSM（Linux 安全模块）挂钩。探针使用受限接口，并针对终止和内存安全等安全属性进行静态验证。因此，它们不会导致操作系统崩溃，也不会使用过多的 CPU 或其他资源。 eBPF 探针可以动态加载到现有的 Linux 内核中，例如大多数 Linux 发行版中打包的内核。因此，eAudit 可以轻松部署在这些内核上，而无需加载内核模块或更改内核代码。因此，eAudit 在大多数最新的 Linux 发行版上“按原样”运行。我们的主要贡献是：

- 现有工具的性能研究：我们在第二部分的激励性研究中包括两个广泛使用的软件工具 auditd [35] 和sysdig [50]；两个已有工作软件的研究系统，即 CamFlow [78] 和 PROVBPF [59]；以及第4节中研究的另外两个基于 eBPF 的工具 Tracee [83]、[84] 和 Tetragon [18]。我们发现所有这些系统存在以下问题：
  - 产生较高的性能开销，使系统速度减慢 2 倍到 8 倍，
  - 会丢弃大部分事件，并且
  - 将事件长时间存储在内存中，使攻击者更容易在永久记录可疑活动之前将其清除。
- 设计高效且有弹性的审计收集系统：在第3节中，我们描述了 eAudit 设计，特别关注避免数据丢失、减少运行时开销以及最大限度地减少日志篡改机会的功能。具体来说，我们提出：
  - 紧凑的数据编码方案，导致日志文件比其他系统小 10 倍；
  - 两级缓冲区设计可减少争用并避免数据丢失；
  - 一个简单的分析模型，支持系统吞吐量和延迟的最佳平衡；
  - 精细且可调的事件优先级方案，可进一步减少日志篡改机会。

  我们的技术也可以应用于其他基于 eBPF 的系统，并提高它们的性能。
- 实验评估：该评估确定了我们设计的几个主要优点：
  - 即使在峰值负载下，eAudit 也能避免数据丢失，而峰值负载会导致之前最好的系统丢失超过 90% 的数据。
  - 我们的两级缓冲区设计和参数调整优化非常有效，在我们的基准测试中平均减少了 18.4 倍的开销（图 17）。

- 根据我们的动机研究中使用的基准和指标，eAudit 的开销仅为 4.5%（图 20）。
- 我们的设计和优化将日志篡改窗口比以前的系统减少了约 100 倍。对于最重要的系统调用，我们的事件优先级方案将这个窗口再缩小 100 倍。

eAudit 的源代码可以在 https://eprov.org/ 找到。

## 激励性实验研究

我们通过实验研究来激发我们的研究，该研究显示了现有系统的缺点：丢失事件（第 2.1 节）、高开销（第 2.2 节）、大篡改窗口（第 2.3 节）和高数据量（第 2.4 节）。

我们的研究包括（a）广泛使用的两个主要软件系统，即Linux审计守护程序（auditd）和 sysdig[50]； (b) 我们可以获得工作系统的两个研究原型，即 PROVBPF 和 CamFlow。我们省略了 PASS [72]、HiFi [80] 和 LPM [11] 等较旧的研究系统，因为它们基于 10 多年前的 Linux 内核，因此让它们与当今的 Linux 发行版一起工作，或者绘制与当今 Linux 内核上运行的其他系统进行有意义的性能比较。PROTRACER [64] 和 KCAL [62] 也被省略，因为它们的代码不可用。最后，我们没有考虑细粒度的溯源收集系统，例如 RAIN [45]，因为它们优先考虑精度，即使它会降低性能。然而，请注意，许多粗粒度和细粒度的溯源收集系统，包括 TRACE [44]、Spade [32]、Winnower [91]、MCI [54]、MPI [63]、BEEP [55] 和 ALchemist [99]，依赖于在auditd 上，因此继承了auditd 的所有性能挑战。

在可行的范围内，我们配置了所有四个系统来收集大致相同的溯源信息。我们遵循各个系统附带的文档，并遵循在线资源中建议的最佳实践。每个可用的与性能相关的配置参数都经过了尝试，我们使用了产生最佳性能的设置。我们唯一坚持的是所有流程的完整溯源收集。

PROVBPF 和 CamFlow 都旨在捕获整个系统的溯源，因此除了打开它们之外，它们不需要进一步的配置。相反，auditd 和 sysdig 都需要配置为记录我们感兴趣的特定系统调用。我们将它们配置为记录与粗粒度溯源相关的所有系统调用，包括所有读/写（或发送/接收）数据的操作。由于这些系统调用使用文件描述符，因此还需要记录创建或修改文件描述符的操作。还记录了创建、修改或更改进程权限的操作，以及修改文件名和权限的操作。记录的调用的完整列表如表 1 所示。

| |
|---|
| accept accept4 bind chdir chmod clone clone3 close connect creat dup dup2 dup3 execve execveat exit exit_group fchdir fchmod fchmodat finit_module fork ftruncate getpeername init_module kill link linkat mkdir mkdirat mknod mknodat mmap mprotect open openat pipe pipe2 pread pread64 preadv ptrace pwrite pwrite64 pwritev read readv recvfrom recvmmsg recvmsg rename renameat renameat2 rmdir sendmmsg sendmsg sendto setfsgid setfsuid setgid setregid setresgid setresuid setreuid setuid socket socketpair splice symlink symlinkat tee tgkill tkill truncate unlink unlinkat vfork vmsplice write writev |

**Table 1:** List of system calls logged. This list of 80 calls is a superset of that of TRACE [44] (66 calls), the most mature and comprehensive provenance-collection system for Linux. All syscall arguments are logged, except for the data buffer argument to `read`, `write`, etc.

(Sekar 等, p. 3) 表1 记录的系统调用列表。这个 80 个调用的列表是 TRACE [44]（66 个调用）的超集，TRACE 是 Linux 最成熟、最全面的溯源收集系统。除了读取、写入等的数据缓冲区参数之外，所有系统调用参数都会被记录。

其中一些系统（例如 sysdig）可以生成二进制格式的输出，而其他系统仅支持可打印格式。为了保持一致性，我们将它们全部配置为使用可打印的输出格式，记录到 /var/log 中的普通文件。实验平台尽可能匹配。

我们的实验平台是Ubuntu 22.04系统，配备i7-6500U处理器（2核/4线程）、8GB内存和1TB SSD。 Auditd 和 sysdig 在此硬件上本地运行，以避免虚拟化开销并最大限度地减少导致时间测量变化的因素。然而，PROVBPF 和 CamFlow 需要 Fedora，并且与特定的内核版本相关；因此我们在同一硬件上运行的 VirtualBox VM 中运行它们。

我们使用了两个基准：postmark [47]，一种广泛使用的文件系统基准，模拟邮件服务器的行为； shbm，一个重复派生另一个二进制文件 (/bin/echo) 的脚本。后者旨在对 shell 脚本最常见的行为进行建模并执行常见的与进程相关的系统调用。我们选择这些相对简单的基准测试是因为我们需要提前知道它们进行的系统调用的数量和类型。这对于确定溯源系统捕获的调用比例是必要的。所有基准测试都是单线程的，以避免这些日志系统过载。出于同样的原因，我们使用2核（4线程）平台进行本次评估。

### 事件丢失

**CamFlow 和 PROVBPF。**这些系统根据 LSM 事件捕获溯源，其级别比系统调用要低一些。因此，并不是所有的系统调用都可以直接与其日志记录匹配。因此，我们重点关注这些基准测试中最常用的两类系统调用，它们在 CamFlow 和 PROVBPF 生成的日志中具有可识别的记录。具体来说，我们计算了为 shbm 基准捕获的 execve 系统调用的比例，以及针对邮戳的文件创建/删除调用的比例。

图 2 中最左边的点显示了这些系统在未修改邮戳时丢失的数据比例。此时 CamFlow 和 PROVBPF 都会丢弃大多数事件（分别约为 66% 和 98%）。为了确定丢失是否是由于这些系

统无法跟上基准而发生的，我们在邮戳的主循环中插入了延迟（nanosleep）。 X 轴显示由于这些延迟而导致基准测试速度减慢的因素。减速因子是减速基准的运行时间（挂钟时间）与未修改基准的运行时间之比。当基准放慢时，数据丢失逐渐减少，这证实了主要因素是系统调用率。

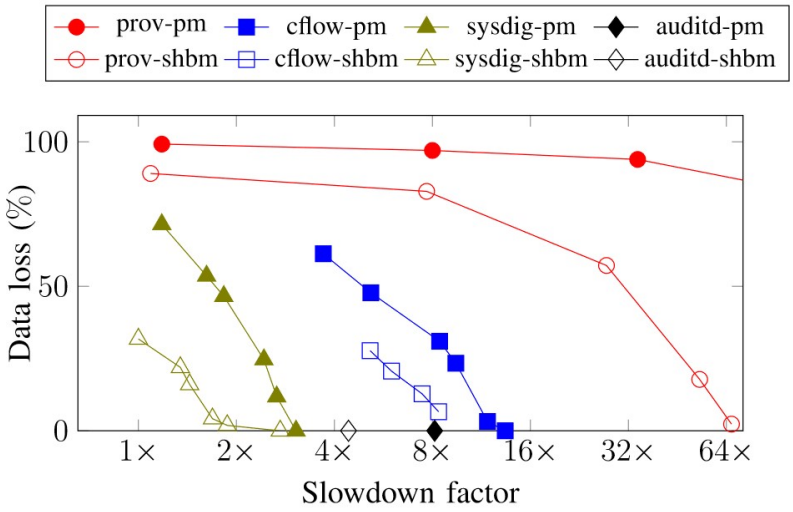


**Fig. 2:** Data Loss: Fraction of events dropped by existing provenance collection systems. The base for calculating the slowdown factor is the benchmark runtime in the absence of provenance collection. The left-most point on each line shows the data loss on unmodified benchmarks. In the legend, "pm" stands for postmark, "prov" for PROVBPF, and "cflow" for CamFlow.

(Sekar 等, p. 3)

CamFlow 在捕获所有事件之前需要减速一个数量级。在这种情况下，一个能说明问题的指标是 camflowd 的 CPU 使用率，camflowd 是用于 CamFlow 中数据记录的用户级进程。当达到 100% 时，记录开始被丢弃。直到我们将基准测试速度降低 8 倍（shbm）到 16 倍（邮戳）时，才达到这个目标。

PROVBPF 需要更大的减速（大约两个数量级）才能捕获大部分数据。此时，很难将基准测试产生的开销与后台系统活动的开销分开。因此，我们在本节的其余实验中省略了 PROVBPF。

Sysdig 和auditd。这些系统直接跟踪系统调用，因此可以轻松根据预期数量检查事件数量。对于 sysdig，我们如前所述在基准测试中插入了延迟，并在图 2 中绘制了数据丢失与减速的关系。它需要减速约 3 倍才能捕获所有数据。 Auditd 不需要延迟，因为它似乎能够放慢基准测试，直到它能够跟上基准。因此，其性能由 X 轴上的单个点表示，该点指示零数据丢失，shbm 减速约 5 倍，邮戳减速约 8 倍。

### CPU利用率和开销

现有的溯源收集技术还受到高开销和 CPU 使用率的影响。请注意，当所研究的系统运行不

正常时，即很大一部分事件被丢弃时，性能测量就没有意义。因此，我们再次像以前一样通过插入延迟来减慢基准测试。这些延迟会不断增加，直到数据丢失率降至 10% 以下。我们还确保这些事件中至少 95% 是由基准引起的，后台活动占比低于 5%。

我们将开销衡量为溯源收集系统使用的 CPU 时间与基准测试的 CPU 时间之比。我们使用 CPU 时间（用户加上操作系统报告的系统时间）而不是挂钟时间，因为它不受基准测试中引入的延迟或进程空闲时间的影响。

我们的结果如图 3 所示。请注意，开销很高，即使在本节中研究的单线程基准测试中，程序也会减慢 4 倍到 16 倍。

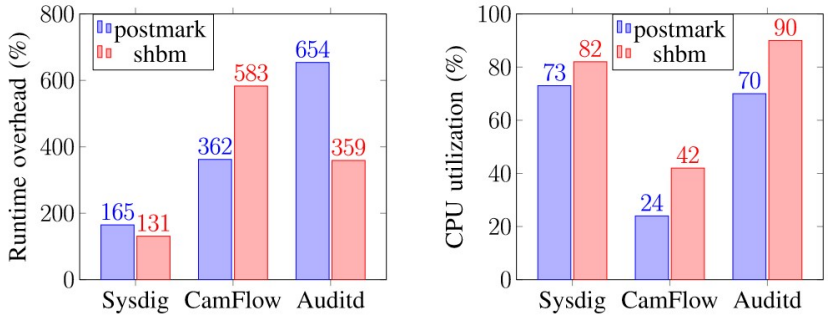


**Fig. 3:** Runtime overhead and CPU utilization: Benchmarks were first slowed down by inserting sleeps to ensure a data loss rate below 10%. Overhead is defined as $t_a/t_b$, where $t_a$ is CPU time of the provenance collection system ("agent time") — specifically, the CPU time of `camflowd` in the case of CamFlow, `sysdig` in the case of sysdig, and `auditd` and `kauditd` for Linux auditd; and $t_b$ is the CPU time of the benchmarks (with embedded sleep's) in the absence of provenance collection.

(Sekar 等, p. 4) 图3 运行时开销和 CPU 利用率：首先通过插入睡眠来降低基准测试速度，以确保数据丢失率低于 10%。开销定义为 ta/tb，其中 ta 是溯源收集系统的 CPU 时间（"代理时间"）——具体来说，对于 CamFlow，是 camflowd 的 CPU 时间；对于 sysdig，是 sysdig；对于 sysdig，是auditd 和 kauditd。 Linux 审计； tb 是在没有溯源收集的情况下基准测试（具有嵌入式睡眠）的 CPU 时间。

## 日志篡改窗口

通过将数据记录到（受保护的）远程服务器，可以保护持久存储中的日志内容免受攻击者的攻击。然而，成功利用漏洞时仍在内存缓冲区中的日志记录可能会被攻击者篡改。在最简单的情况下，攻击者可以立即杀死记录器，导致所有这些条目丢失。获得篡改所需特权的漏洞一般来说可能只占少数，但它们在 APT 活动中很常见。此外，攻击后检测正是针对此类对手——那些强大到足以突破所有部署防御的对手，因此只能在事后检测到。

为了测量日志篡改窗口，基准测试在随机选择的时间被终止。日志的长度立即被记录下来。这些系统继续将记录写入日志文件 —— 这些必须是内存中缓冲的记录，因为基准测试已经终止。该阶段的结束是由数据收集系统的CPU使用率急剧下降决定的，日志增长的突然下降

进一步证实了这一点。再次记录日志文件的长度。我们计算两次长度测量之间的记录数量，并在 10 次重复中平均该数量。

如图4所示，这些系统的日志篡改窗口数量级为几十到几十万条记录。就时间而言，这接近一秒左右的范围，使得自动利用能够消除围绕入侵的关键活动。

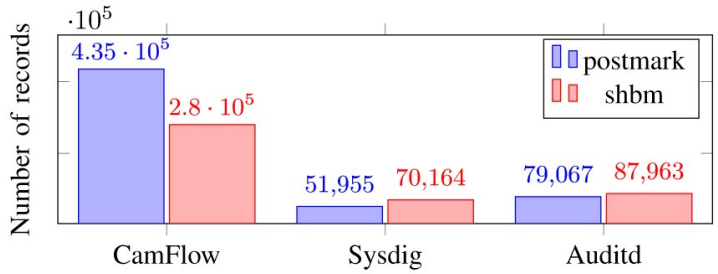


**Fig. 4:** Log tampering windows of CamFlow, sysdig and auditd.

(Sekar 等, p. 4)

**数据量**

现有溯源收集框架的另一个主要缺点是数据量。运行 3 分钟的邮戳基准测试，sysdig 生成 9GB，Linux auditd 生成 26GB，CamFlow 生成 41GB。如果我们取这些数字中最小的一个，并进一步假设平均事件发生率仅为该比率的十分之一，则该数字加起来仍超过每台主机每天 400GB。

## 相关工作

过去 25 年多来，系统调用数据支撑了大多数关于入侵检测、预防和恢复的研究 [28]、[93]、[86]、[25]、[30]、[51]、[33]、[ 88]、[89]、[52]、[57]。部分受此推动，粗粒度溯源收集方法通常基于审计日志，例如 Spade [32] 和 Trace [44]。其他依赖于直接检测操作系统内核，例如 LPM [11]、HiFi [80]、CamFlow [78] 和 KCAL [62]。操作系统检测的一个优点是覆盖范围扩大到包括一些与系统调用无关的内核事件，但这些事件对于 APT 分析的实际意义尚不清楚。不利的一面是，涉及内核更改的方法很难维护：只有其中一个系统 (CamFlow) 可用于过去 10 年发布的操作系统内核 [94]。PROVBPF [59] 是使用 Linux eBPF 框架对容器环境进行 CamFlow 风格溯源捕获的实现。然而，他们的方法基于称为 SABPF 的 eBPF 扩展。因此，它们仍然需要内核修改，从而导致相同的维护/可部署性问题。

**溯源收集。**W3C [92] 将溯源定义为“有关参与生成数据或事物的实体、活动和人员的信息，可用于对其质量、可靠性或可信度进行评估。”早期的溯源研究是出于对科学再现性的考虑，这需要记录操作系统上每个进程执行的完整细节，包括所使用的所有代码和数据文件的内容。溯源感知存储[72]被提议作为记录此类信息的自然场所。再现性本质上需要保留历史文件内容，从而导致高昂的存储成本。

攻击检测和取证等安全应用程序不需要完全可重复性，但它们受益于记录更广泛的安全相关事件，这是导致最近许多溯源工作基于系统调用的因素，例如 Spade [32 ]、Trace [44] 和 CamFlow[78]。

**操作系统审计和日志记录工具。** Linux审计系统auditd最初是由Red Hat为了Common Criteria [27]认证而开发的。它支持一系列安全事件，包括系统调用。这一功能使auditd成为众多溯源收集系统的首选[32]、[44]、[91]、[54]、[63]、[55]、[99]以及APT分析[40]，[38]、[31]、[70]、[41]、[91]、[100]、[98]、[23]、[101]、[79]。然而，auditd 的作者可能并不打算让他们的系统进行连续的系统调用日志记录。因此，它在用于粗粒度溯源收集时面临性能挑战。 go-audit 项目 [87] 的目标是比auditd 更快的用户级处理，但由于它继续依赖 kauditd 和 netlink 套接字，而这些套接字占 Linux 审计系统开销的 50%（[62] 中的图 5），它无法实现我们的性能目标。

Tracee [83]、[84] 是一个“基于 eBPF 的威胁检测引擎”，专门研究检测可疑和/或规避行为的规则。还有一个用于事件收集的独立工具 (Tracee-eBPF)，我们的评估中已经讨论过该工具。 Sysdig 是一个“深度系统可见性的简单工具”，可以捕获系统调用和其他操作系统事件，并提供“strace + tcpdump + htop + iftop + lsof ...”的功能。对于系统调用监控，Sysdig 显著提高了 auditd 的性能，促使最近的研究工作[98]，[23]转向Sysdig。 Sysdig 的默认发行版使用内核模块，但也可以使用 eBPF 版本 [14]。正如 eBPF 版本 [14] 的作者所承认的，我们发现它的性能比 sysdig 的本机版本差，因此我们的实验使用（性能更高）本机版本。

Sysmon for Linux [26] 是 Windows Sysmon [66] 主要功能的基于 eBPF 的实现。它专注于系统调用的一小部分，包括进程创建/终止和网络连接。跟踪的事件不足以跟踪数据来源，例如确定文件对系统的依赖性。此外，之前的工作表明，与数据来源相关的调用（例如读、写、打开和关闭）占系统调用的绝大多数[40]、[42]，这意味着数据来源跟踪本质上要求更高。因此，将 Sysmon 的性能与本文讨论的其他工具进行比较并没有多大意义。

Datadog [5] 使用 eBPF 提供日志管理框架和基于规则的威胁检测，但不具备系统调用日志记录的通用功能。在早期的工作 [1] 中，我们研究了构建基于 eBPF 的审计集合的可行性，但并未对本文核心的性能问题进行检验。

**细粒度的溯源。**粗粒度的溯源可能会导致依赖性爆炸，从而降低攻击取证能力。最近的一些工作开发了一些技术来减轻分析时的依赖性爆炸[70]、[41]、[37]。更常见的替代方案是依靠细粒度信息流（又名污点）跟踪[74]、[96]、[10]、[49]、[53]、[45]、[46]，但不幸的是，它使系统速度减慢 2 倍到 10 倍，同时大大增加日志大小。为了解决性能挑战，BEEP [55]、PROTRACER [64] 和 MPI [63] 开发了一种新的细粒度跟踪技术，称为执行分区。 MCI [54] 和 PROPATROL [69] 使用基于模型的推理执行细粒度跟踪。 ALchemist [99] 将应用程序日志与系统审计日志相结合，以获得更细粒度的来源。请注意，所有这些技术仍然需要系统调用审计数据，因此它们可以直接受益于我们方法的性能和可扩展性收益。

**日志篡改。** Paccagnella 等人[77]表明，auditd 很容易受到内存记录的篡改攻击。我们表明其他数据收集系统也容易受到攻击，而且它们的窗口可能非常大。 Hoang 等人 [39] 提出了一种算法，该算法在性能和安全性方面比 Paccagnella 等人有所提高。这些工作都不能防止审计记录的泄露，但可以确保篡改行为被检测到。相比之下，我们的工作旨在最小化篡改窗口，从而最大化保留的攻击证据。在这方面，HARDLOG [6]的目标与我们的目标相似。它们确保关键系统调用（例如 execve）同步记录，而其他系统调用则以有限的延迟记录。然而，为了实现这一点，他们需要专门的硬件，而且需要对内核进行重大更改，以避免将日志数据发送到用户级别。相比之下，我们的工作优先考虑当今硬件上的可部署性以及与现有 Linux 发行版的开箱即用兼容性。尽管有这些限制，我们表明我们可以实现非常小的日志篡改窗口，特别是对于关键和重要的系统调用。

**数据缩减技术。**许多研究人员都致力于减少大量的审计日志。其中一项研究是（无损）压缩[16]、[95]、[17]、[24]、[20]。由于压缩数据不适合通用搜索或分析算法，因此这些技术主要用于降低存储成本而不是分析成本。相比之下，删除“不重要”事件的（有损）数据缩减也可以降低分析成本。

一类数据缩减技术认为良性事件不重要，并且仅存储可能属于攻击一部分的事件。 Winnower [91] 使用 DFA 学习技术来消除良性事件。他们报告说，如果在许多主机上复制相同的应用程序，则会显著减少。然而，当应用于单个进程时，该技术似乎不是很有效[43]。 Rapsheet [37] 建议剔除良性事件，除非它们与可疑事件有因果关系。

这些方法的一个主要缺点是恶意事件可能被错误地分类为良性事件，导致在取证分析过程中遗漏攻击步骤或其影响。 LogApprox [68] 避免了此类遗漏，但允许可能引入虚假依赖关系的泛化。相比之下，许多研究工作都避免了依赖关系中的误报和漏报。 LogGC [56]、[61] 是一种受垃圾收集启发的方法，用于丢弃对单个进程专用的临时文件的操作。显然，删除这些操作并不能切断任何因果链。虽然这种技术在与细粒度单元仪器一起使用时是有效的 [55]，但它在粗粒度设置中仅具有适度的效果[42]、[43]。NodeMerge [90] 识别重复活动（例如库加载）的模板，并将其替换为单个节点。在某些工作负载上，它可以显著减少，但在其他工作负载上，其效果可能不大[43]。

虽然 LogGC 和 NodeMerge 确定了事件冗余的两个重要实例，但 Xu 等人 [97] 开发了一种更通用的特征，称为完全可跟踪性等效性，可实现数据大小约 2 倍的减少 [42]、[43]。通过可证明地始终保留节点之间的可达性关系，他们的技术确保了取证追溯或前向结果的准确性。在后续工作 [42] 中，我们表明，为了获得忠实的取证分析结果，当节点的状态可能发生变化时，保持节点的前向可达性就足够了。这种放松，与我们对问题的版本化图表述所实现的全局优化相结合，使我们的完全依赖性保留（FD）技术能够比 Xu 等人进一步实现约 4× 数据缩减 [42]、[43]。 Zhu等人[102]探索依赖关系保留以及更积极的良性事件修剪技术，但他们的主要关注点是产生更快运行时间的更简单的算法。

上述工作将数据缩减视为对本文研究的工具生成的初始数据进行操作的后处理阶段。相比之

下，eAudit 的目标是减少初始数据的大小。这里的减少有可能转化为数据减少算法的输出大小的相应减少。我们已经在数据压缩技术中展示了这一点（表 24）。此外，这些数据大小的减少可以导致整个数据减少管道的运行时性能的相应提高。

## 研究内容

---

### eAudit系统设计

我们设计的主要目标是：

- 减少数据量，以加快数据管道中涉及的每个组件的速度；即使在系统负载高峰时也能消除数据丢失；
- 减少开销，以最大程度地减少可持续的峰值工作负载的恶化；
- 减少数据捕获的延迟，以便将记录快速记录到安全存储中，从而最大限度地减少对手篡改这些记录的机会。

#### 目标、威胁模型和基准选择

本文的主要目标是开发有效的溯源收集技术，这种技术不会降低目标系统上可维持的工作负载。正如我们在上一节中所示，现有方法将溯源收集的 CPU 工作负载增加了 100% 以上。如果日志记录带来了巨大的开销，操作员可以通过在持续或高峰系统活动期间将其关闭来做出响应。反过来，攻击者可能会利用这一点来隐藏他们的活动。首先，攻击者可能会在受害者系统的高负载时期进行攻击。其次，攻击者本身可能会生成类似于良性活动的大量工作负载。通过消除性能开销这一重要问题，我们可以消除这种攻击途径。

第二个目标是阻止利用现有溯源收集系统在高负载期间丢失记录的倾向的攻击。因此，即使溯源收集永久开启，攻击者仍然可以在峰值负载期间隐藏其活动：由于很大一部分记录被删除，因此攻击者的大部分活动可能不会出现在日志中。这为依赖高负载的规避攻击提供了第二条途径。

为了解决这些逃避途径，我们的目标是构建一个溯源收集系统，即使在底层操作系统和硬件可以承受的峰值系统负载下，该系统也可以在不丢失记录或显著性能开销的情况下运行。我们选择的基准（图 5）反映了这一目标。例如，shbm 的多个实例可以轻松地最大化操作系统执行程序的速率。同样，rdwr 可以实现读写的峰值速率，而 find、tar 和 postmark 可以强调操作系统的文件系统吞吐量。最后，我们添加了 httperf 和 kernel，这是两个众所周知的网络和 CPU 相关负载基准测试。

我们的最后一个目标是降低攻击者通过日志篡改隐藏其活动的能力。攻击者可能会尝试通过

在保存尚未写入（安全）存储的溯源记录的队列中主动创建大量积压来最大程度地增加机会。或者，他们可能会在预计积压量很大的时期进行攻击。大量的积压使攻击者有足够的时间侵入受害者，提升其权限，然后在写出与攻击相关的活动之前删除队列中的记录。我们的目标是将积压的日志减少到成功篡改日志变得极其困难的程度。

除了已经可以使用所有可用核心的内核基准测试之外，其余基准测试都是通过添加顶级循环来并行化的，该循环创建指定数量的进程，每个进程都运行基准测试的副本。这使我们能够创建可配置的多核负载。我们在第二节中用于实验的硬件平台上运行了这些基准测试：i7-6500U 处理器，2 核（4 线程）、8GB 内存和 1TB SSD。我们还添加了具有更新硬件的第二个平台：12 核 i7-12700 处理器（禁用超线程）、16GB 内存和 512GB SSD。今后，我们将使用这两个平台来验证我们的结果是否具有普遍性。

**eBPF概述**

本节提供了 eBPF 框架的简短概述，重点关注我们系统设计的核心功能。 eBPF 框架允许将称为 ebpf 探针的小代码片段安全地部署在 Linux 内核中定义良好的挂钩上。目前支持的钩子包括Linux跟踪点、kprobes、LSM钩子等。当内核控制流到达钩子时，注册在该钩子上的探针回调函数将被调用。虽然其中一些挂钩点（例如跟踪点）在大多数 Linux 发行版上都是开箱即用的，但其他挂钩点（例如 Ubuntu 上的 LSM 挂钩）则需要使用非默认选项重建内核。在操作设置中，此类非默认内核可能不是一个选项，因此 eAudit 仅使用跟踪点挂钩进行系统调用进入和退出。

eBPF 运行时定义了一个虚拟机，支持用于编写探针的虚拟指令集。程序员通常在 C 子集中编写探针代码，然后将其编译为 eBPF 指令。在加载探针之前，Linux 内核会验证多个属性，例如内存安全性和不存在循环。这些检查可确保探针不会使内核崩溃或过度减慢内核速度。最后，虚拟指令被 JIT 编译为本机代码并加载到内核中。

探针可以使用一小组（非常）小的辅助函数，这些函数经过精心设计，可以最大限度地降低风险。一组辅助函数用于安全读取内存（用户或内核）。还提供了写入内存的功能，但通过在每次使用时在控制台上打印警告消息来阻止其使用。因此，大多数探针（包括所有跟踪点探针）都具有“只读”行为。

第二组辅助函数提供了称为 eBPF 映射的键值存储。由于验证者不允许静态变量，因此 eBPF 映射成为了在探针回调中维持任何状态的唯一机制。映射可以是每个 CPU私有的，也可以在所有 CPU 之间共享。每个核心上的每个 CPU 映射都有一个实例，这意味着同一探针会在不同核心上看到这些映射的不同实例。共享映射上的操作包含并发控制，因此它们不如可以安全访问而无需此类控制的每个CPU映射高效。

第三组辅助函数支持与用户级应用程序的通信。事实上，eBPF 探针的目的是拦截选定的操作并将观察到的数据发送给用户级“消费者”。 Perf 缓冲区是与用户级通信的较旧机制，而建议使用最近引入的环形缓冲区以获得更好的性能[73]。

虽然Linux内核实现了eBPF的所有核心功能，但它提供的接口相当低级，需要应用程序使用eBPF字节代码。为了简化开发，基于 LLVM 的编译器可用于将以高级语言（C 的受限子集）编写的探针转换为 eBPF 字节码。还有许多工具包和库，例如 bcc [12]、bpftrace [2] 和 libbpf [58]，它们简化了其他低级方面，例如解析 ELF 二进制文件、调用 eBPF 加载器、设置和访问 eBPF 映射用户级别等。 BCC [12]（可以说）是其中最成熟的，因此 eAudit 实现当前使用此工具包。有关 eBPF 的更多详细信息，请参阅[21]、[81]、[34]、[15]、[29]。

**eAudit架构**

图 6 描述了 eAudit 的架构。它由一个在内核中运行的组件 eCap 和两个用户级组件 eLog 和 eParse 组成，分别用于记录和解析/打印。下面更详细地描述每个组件。

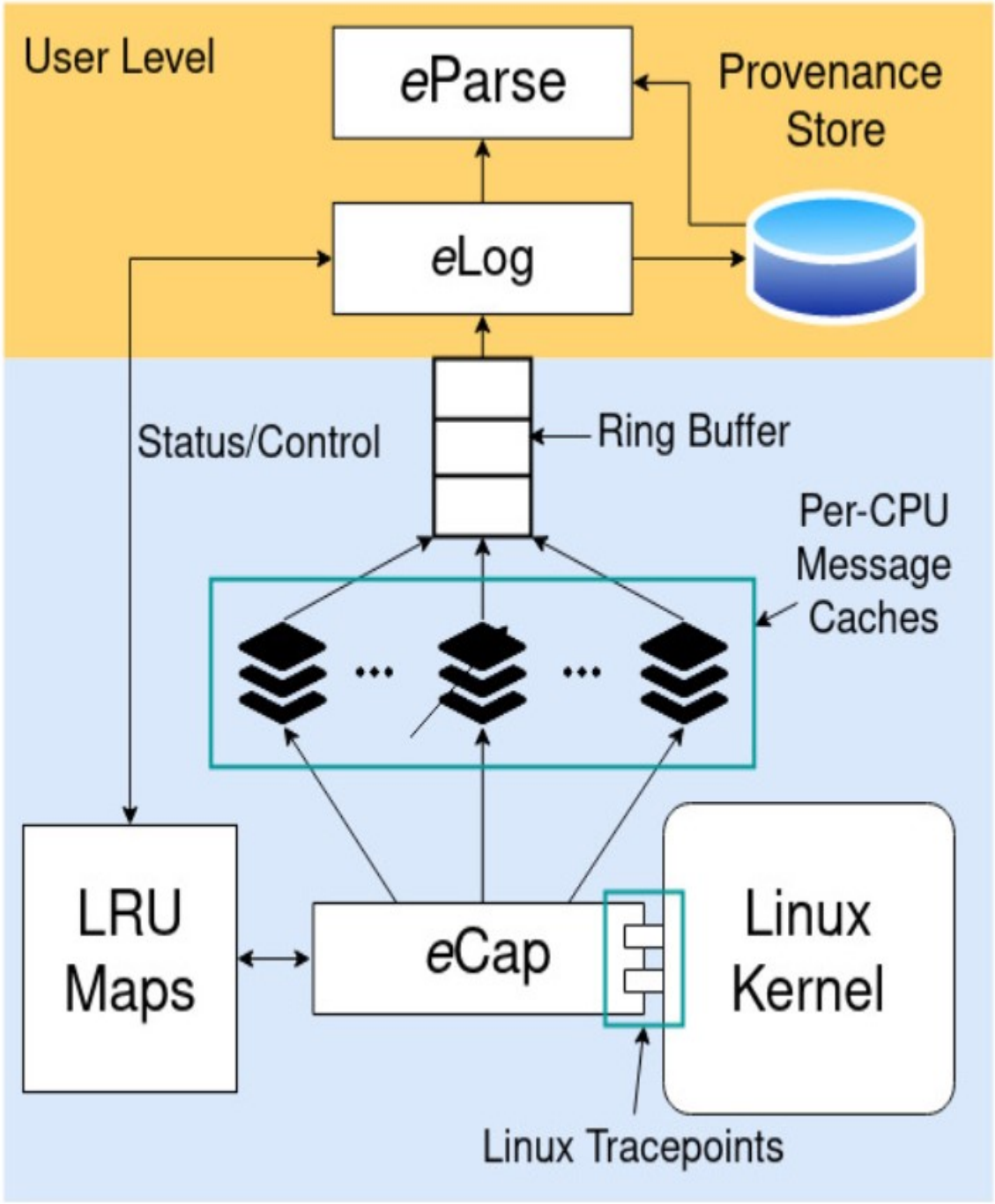


**Fig. 6:** *e*Audit Architecture.

eCap 关注内核中的数据捕获。它在 Linux 内核跟踪点接口 [48] 上附加探测器，特别是表 1 中列出的那些与溯源相关的系统调用相关联的探针。每个探针都是一个具有在sysfs伪文件系统中的/sys/kernel/debug/tracing/events/syscalls/⟨scevent⟩ /format指定的签名的函数，其中⟨scevent⟩代表特定系统调用的进入或退出，例如sys enter execve。系统调用参数可以在进入事件中访问，而返回值可以在退出事件中访问。系统调用信息，包括参数和返回值，首先

被序列化并存储在我们称为消息缓存的每个CPU缓冲区中。当这些缓存之一被填满时，它将被写入（共享）环形缓冲区。

eCap 使用 eBPF 映射将状态维护为一组键值对。资源泄漏（以不再使用的分配状态的形式出现）给 eAudit 等长时间运行的软件带来了严重的问题。不幸的是，泄漏检测算法通常需要循环，而这在 eBPF 中是不允许的。最近最少使用 (LRU) 映射在这种情况下提供了一种优雅的机制 —— 当新项目需要空间时，最旧的条目将被自动清除。

eLog 是用户级组件，它读取 eCap 发送的数据并立即将其记录到溯源存储中。溯源存储可以位于锁定的远程计算机上，但为了简单性和与我们比较的其他工具的一致性，我们当前的实现使用本地文件。因为我们的实现依赖于 BCC [12]，所以 eLog 使用 Python 程序将 eBPF 探针加载到内核中，并访问用于查询探针状态或配置探针的映射子集。然而，为了最大限度地提高性能，环形缓冲区处理程序和 eLog 的其余部分都是用 C 编写的。这意味着用户级代码中的关键路径避免了 Python 代码中可能出现的性能瓶颈。

eParse 是一个用户级组件，用于以可读格式或适用于入侵检测和取证分析的体系结构中立格式解析和打印来自 eCap 的（二进制）数据。 eParse 可以链接在 eLog 之上以实时解析数据，或使用溯源存储中的数据进行离线操作。

> ## 1. eAudit的总体架构
>
> eAudit由三个主要组件组成，分别位于**内核层**和**用户层**，通过高效的数据传输和处理机制确保系统的安全性和性能。这三个组件分别是：
>
> - **eCap**：内核中的数据捕获模块。
> - **eLog**：用户层的日志记录模块。
> - **eParse**：用户层的日志解析和展示模块。
>
> ### a. eCap（内核中的数据捕获模块）
>
> **eCap**是eAudit框架的核心组件，负责在**Linux内核层**捕获系统调用和相关事件。它使用了eBPF框架中的**tracepoints**（跟踪点）来监控系统调用的入口和出口。具体而言，eCap通过内核钩子捕获与安全相关的系统调用，并将这些信息**序列化**后存储在每个CPU的缓存中。这个缓存叫做"**消息缓存**"，它是一个**双层缓冲设计**的一部分。
>
> eCap的职责包括：
>
> - 在关键的系统调用点挂载eBPF探针（如`sys_enter_execve`，`sys_exit_execve`）。
> - 序列化捕获的系统调用及其参数，并暂时存储在每个CPU的消息缓存中。
> - 当缓存达到预定容量或满足优先级要求时，将这些日志条目推送到环形缓冲区供

用户层处理。

**b. eLog（用户层的日志记录模块）**

**eLog**是位于用户层的模块，负责从**环形缓冲区**读取内核中捕获的数据并将其安全存储到持久化的日志文件中。eLog具备高效的处理能力，能够实时从内核中读取大量的系统调用数据，而不会造成数据积压。

eLog的工作流程包括：

- 监听内核的环形缓冲区，获取缓存中的系统调用记录。
- 每当缓冲区中的数据达到一定量时，eLog会被触发，并将这些数据写入本地或远程的**日志存储**。
- eLog采用了一个优化的唤醒机制，减少不必要的系统唤醒，降低了CPU开销。

**c. eParse（用户层的日志解析模块）**

**eParse**是用于解析日志数据的模块，它可以将二进制的日志数据转换为可读的文本格式，供分析工具或安全分析人员使用。它的功能包括：

- 实时解析日志数据，帮助安全团队追踪事件的发生过程。
- 支持将日志转换为跨平台通用的格式，用于后续的**安全取证**或**攻击调查**。

eParse既可以实时运行，也可以在离线模式下处理已经记录的日志。

---

## 2. 双层缓冲设计

**双层缓冲设计**是eAudit体系中重要的一部分，它的目的是减少内核和用户空间之间的数据传输开销，同时避免数据丢失。

1. **消息缓存**：每个CPU核心都有自己的消息缓存。当捕获到系统调用时，数据首先会被写入该缓存。缓存存满后，数据才会被传输到下一层的环形缓冲区。
2. **环形缓冲区**：这是一个位于内核空间和用户空间之间的共享缓冲区。当CPU的消息缓存满时，数据会被批量传输到环形缓冲区。这样设计的好处是减少了频繁的上下文切换和数据拷贝操作，提高了系统的整体性能。

双层缓冲设计通过减少直接将每次系统调用日志写入用户空间的频率，**降低了CPU开销**，并**防止了数据丢失**。

---

> ### 3. 系统调用优先级机制
>
> 为了进一步优化日志处理流程，eAudit引入了**系统调用优先级机制**。不同的系统调用被赋予了不同的优先级，这意味着对安全影响较大的调用（如`execve`、`setuid`）将会被优先处理和记录。
>
> **工作流程**：
>
> 1. 当捕获到关键系统调用时（如权限提升或进程创建），这些调用会立即被推送到环形缓冲区，并触发用户空间的eLog进行日志写入。这种设计极大地减少了**关键事件**的日志篡改窗口。
> 2. 对于较低优先级的系统调用，eAudit会根据设定的缓存大小或时间阈值来批量推送，减少频繁的上下文切换，从而优化性能。

**紧凑数据编码**

减少数据量的一个明显策略是开发紧凑的数据表示。先前的研究倾向于淡化此步骤的重要性，强调可以在初始数据收集之后稍后执行压缩。然而，正如我们的结果所示，大量的溯源数据使系统不堪重负，导致数据在到达后处理之前就丢失了。我们对最新 Fedora 发行版的性能研究充分说明了这一点：由于这些发行版默认打开文件压缩，文件系统性能会降低，这意味着现有溯源收集系统的数据丢失率要高得多。因此，eCap 从紧凑编码开始。

我们的紧凑编码方案的一些基本元素是：(a) 使用单个字节对每个事件进行编码，(b) 抑制单线程进程的线程 ID 信息，以及 (c) 每个事件仅包含时间戳的最低有效 3 个字节（以纳秒为单位）。当时间戳的前 5 个字节发生变化（即每 16 毫秒）时，会发出单独的时间戳记录。参数编号和类型信息不包含在事件记录中，因为它可以从事件名称推断出来。

许多整数参数具有较小的值，例如文件描述符和系统调用返回值（通常为零）。我们对整数参数使用可变长度编码，使用 1、2、4 或 8 个字节，具体取决于参数的实际值。该长度信息被编码为单个字节，足以表示最多 3 个参数加上一个返回值。具有更多参数的系统调用使用 8 个字节来存储其余参数。

文件名和套接字地址使用可变大小的表示形式，并使用前缀字节对其长度进行编码。对于 execve，它在采用可变数量的参数方面很不寻常，参数号被编码在记录中。目前，我们的实现将参数和环境变量的数量限制为最多 32 次，尝试记录更多参数会导致 eBPF 验证程序拒绝权限。我们可以使用 eBPF 尾部调用来克服这个限制，但这留待将来的工作。 （请注意，在我们的基准测试中从未达到此限制，因此我们的性能数据和比较不受影响。）

紧凑编码的一个附带好处是我们可以单独记录进入和退出事件，而不必过度关心日志大小。生成单个组合记录似乎更有效，但通过单独记录它们，我们可以减少延迟，即安全存储信息之前的时间。这对于具有重大安全隐患的系统调用（例如 setuid、kill 和 execve）尤其重

要。对于频繁发生和/或安全隐患较小的系统调用，我们在系统调用退出时捕获组合记录。

由于这种编码，我们的系统调用记录平均为 17 字节（邮戳基准），而 sysdig 为 175 字节，auditd 为 850 字节。请注意，eAudit 以及 sysdig 和auditd 提供有关系统调用的相同信息 —— 所有参数值均可用，但读取或写入数据的系统调用的数据缓冲区参数除外。

为了评估我们的紧凑编码的影响，我们测量了eAudit的性能开销，作为系统调用记录大小的函数。我们使用了postmark基准测试。记录大小通过添加指定数量的填充字节来增加。填充字节数量从0变化到160。这使得总记录大小从eAudit产生的平均大小变化到sysdig产生的平均大小。请注意，开销还受到下一节讨论的参数p和w的影响。我们将它们设置为第4节中使用的相同（最优）值。

图 7 显示，挂钟开销（定义为基准测试完成时间的增加百分比）与系统调用记录大小大致呈线性增长。对于 1 核和 2 核负载，这一增幅小于 50%，但对于 8 核和 12 核负载，这一增幅迅速增加至 4 倍和 11 倍。当填充大小为 128 和 160 字节时，12 核负载使系统不堪重负，导致开销突然增加，并导致一些数据丢失。

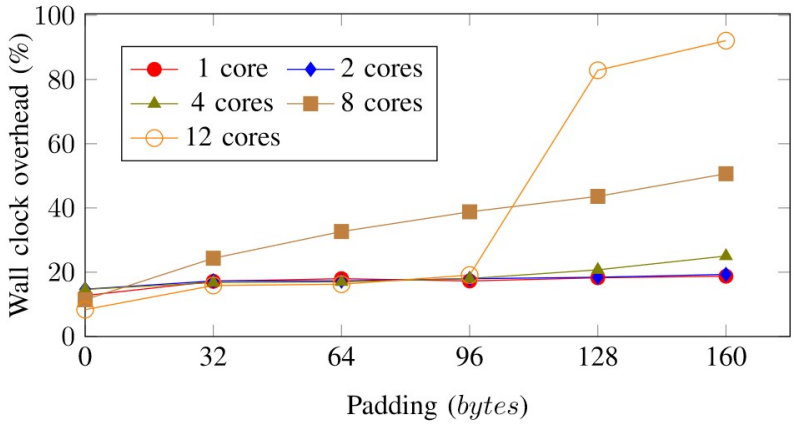


**Fig. 7:** Impact of compact encoding on *e*Audit overhead.

(Sekar 等, p. 7) 图7 紧凑编码对 eAudit 开销的影响。

### 两级缓冲设计避免数据丢失

如前所述，eBPF 提供了两种内核数据探针和用户级消费者进程之间的通信机制：perf 和环形缓冲区。环形缓冲区是较新的机制，旨在解决性能缓冲区的一些缺点[73]。特别是，环形缓冲区在所有 CPU 核心上使用单个大缓冲区，而 perf 对 N 个核心使用 N 个缓冲区。单个大缓冲区比 N 个缓冲区（每个缓冲区大小均为此大小的 1/N）更好地容忍数据量峰值。此外，由于其对性能的关注，环形缓冲区 API 包括减少不必要的数据复制的功能，以及对向用户级消费者发送有关数据可用性的信号（使用 poll/epoll 机制）的频率的显式控制。

尽管设计注重性能，但我们发现直接使用环形缓冲区 API 不足以实现无损溯源收集。因此，我们开发了一种两级缓冲方案来减少对环形缓冲区的访问次数。在此方案中，系统调用记录

首先组装在每个 CPU 的"消息缓存"中。当缓存变满时，其内容（"消息"）将在环形缓冲区中排队。尽管这种方法引入了额外的数据复制（从消息缓存到环形缓冲区），但这是不可避免的：保留/提交 API，用于就地构造环形缓冲区消息，要求消息大小是编译时常量[73]。产生重要数据的事件都与文件名或其他可变大小的数据相关联。因此，我们必须依赖ringbuf输出，它需要将数据组装在临时缓冲区中，然后复制到环形缓冲区中。

下表概述了影响该两级缓冲器设计性能的关键参数：

| | |
|---|---|
| P | **每个 CPU 消息缓存的尺寸，以系统调用记录的数量来衡量。** |
| R | **环形缓冲区的大小。** |
| W | **唤醒间隔—对环形缓冲区的每w次操作之一将向用户级指示数据可用性。** |
| N | **CPU 核的数量。** |

在这四个参数中，我们发现p对数据丢失的影响比w大得多。增加 r 可以减少短期运行基准的数据丢失，但不会减少长期运行基准的数据丢失。特别是，当系统无法按照事件产生的速率处理事件时，环形缓冲区中的条目就会开始激增。这种不平衡甚至会导致大型环形缓冲区最终被填满，此时数据将被丢弃。

这给我们留下了两个影响数据丢失的主要参数，即p和N（基准测试使用的核心数量）。图 8 重点关注单个基准测试结果，并显示 p 和 N 对数据丢失的影响。当工作负载使用一或两个CPU核心时，所有p值都可以避免数据丢失。因此，1 核和 2 核工作负载的数据点沿 X 轴排列。但随着工作负载使用的核心数量增加，需要更大的消息缓存大小。对于所研究的最大配置（i7-12700 处理器上有 12 个本机内核），需要 p = 10 才能消除数据丢失。

图 9 从不同的角度检查了数据丢失：基准有所不同，同时将 N 固定为 12。请注意，p ≥ 18 的值在所有情况下都可以避免数据丢失。 eAudit 使用默认值 p = 100，这为该最小值提供了一个舒适的余量。 （对于这两个图表，我们使用 w = 8 和 r = 16MB，这些值足够大，进一步增加不会影响数据丢失。）

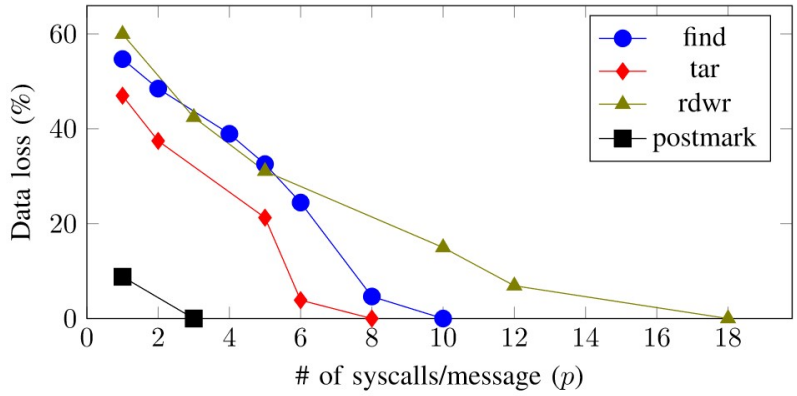


**Fig. 9:** Data loss Vs Message cache size ($p$). Curves correspond to different benchmarks, each parallelized to use all cores on a 12-core system.

(Sekar 等, p. 8) 图 9：数据丢失与消息缓存大小 (p)。曲线对应于不同的基准，每个基准都并行化以使用 12 核系统上的所有核心。

在我们的实现中，设置 p = 1 会绕过每个 CPU 缓存。图 8 和 9 显示，在没有 CPU 消息缓存的情况下，eAudit 会遭受大量数据丢失（高达 60%）。随着消息缓存大小的增加，数据丢失逐渐减少，对于所有基准测试和我们实验中使用的所有 N 值，数据丢失在 p = 18 时降至 0%。尽管 p = 18 时可以避免数据丢失，但较大的 p 值会在性能方面提供额外的好处。

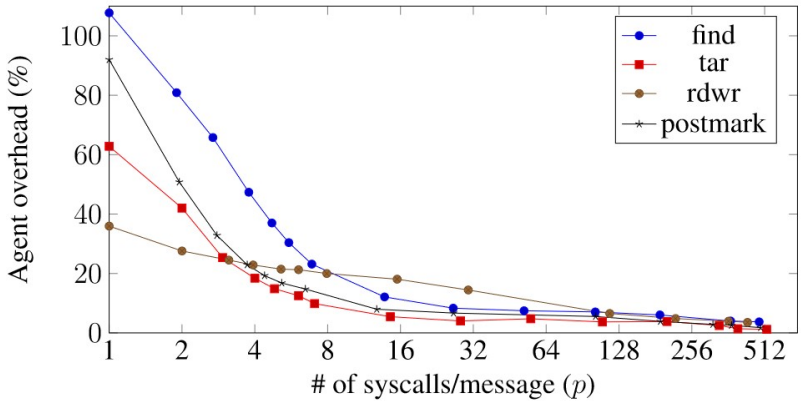


**Fig. 10:** *e*Audit's Agent Overhead Vs Message Cache Size ($p$). For $p < 18$, overhead numbers are an underestimate in cases where there is data loss.

(Sekar 等, p. 8)

图 10 显示，随着 p 从 16 到 100，代理开销持续下降。平均而言，在图 10 中的基准测试中观察到开销减少了 2 倍。总而言之，这些结果表明每个 CPU 缓存非常低。有效提高溯源收集的性能。因此，eAudit 能够避免基准测试中的数据丢失，这些基准测试的强度（就系统调用率而言）是现有溯源收集系统所维持的基准的 10 倍以上。

### 优化开销与延迟的权衡

在上一节中，我们表明可以通过增加 p 来避免数据丢失，但较大的 p 值也会产生负面影

响：它们会增加延迟，即事件在内存中缓冲的时间段，因此容易受到篡改（第 2.3 节）。类似的评论也适用于 w 参数：较大的 w 值会减少开销，但代价是增加延迟。然而，这两个参数以不同的方式影响延迟和开销，使我们能够制定一个有趣的优化问题来决定正确的权衡。表 11 总结了我们的公式和由此得出的分析解。我们在下面描述这个公式并进行实验验证。

在每个事件的基础上，代理的执行时间来自： (a) 在每个 CPU 消息缓存中收集和存储事件信息并复制它直到数据最终记录到溯源存储中的时间 tb ； (b) 在环形缓冲区上将消息入队/出队的时间 tq，每 p 个事件发生一次； (c) 唤醒用户级消费者的时间ts，每w条消息，即每个wp事件发生一次。其中，(a) 不受缓冲方案的影响，并且代表事件记录的"基本"成本。因此，我们在表 11 中关注缓冲开销 O，其中包括 (b) 和 (c)。

| | | |
|---|---|---|
| **Parameters** | $N$ | number of CPUs |
| | $p$ | message cache size (number of events) |
| | $w$ | wake-up interval |
| | $t_q$ | time to queue messages on the ring buffer |
| | $t_s$ | time to context-switch (to wake up user-level agent) |
| **Metrics** | $T$ | Agent CPU time per event: $t_b + t_q/p + t_s/w{\cdot}p$ |
| | $O$ | Buffering overhead per event: $t_q/p + t_s/w{\cdot}p = (t_q + t_s/w)/p$ |
| | $L$ | Maximum length of buffered data: $N{\cdot}p + p{\cdot}w = (N + w){\cdot}p$ |
| **Goal** | *Minimize* $O{\cdot}L = (t_q + t_s/w)(N + w)$. *Solution:* $w = \sqrt{N{\cdot}t_s/t_q}$ $\left(\text{by setting } \frac{d(O{\cdot}L)}{dw} = 0\right)$. *Note:* Objective emphasizes $O$ and $L$ equally. At the optimal $w$ value, $d(O{\cdot}L)/dw = 0$, which is equivalent to: $$\frac{1}{L}\frac{dL}{dw} = -\frac{1}{O}\frac{dO}{dw}$$ In other words, the relative change in $L$ with $w$ will be equal and opposite of the corresponding change in $O$. At all other $w$-values, it can be shown that any change to $w$ will improve one metric more than it degrades the other. | |

**Table 11:** A simple performance model for tuning $w$ to find an optimal trade-off between overhead and latency.

(Sekar 等, p. 8) 表 11：一个简单的性能模型，用于调整 w 以找到开销和延迟之间的最佳权衡。

w 的最佳值取决于 N 以及 ts 和 tq 的比率。我们发现具有较多核心数 (N ) 的处理器往往具有较大的 tq （排队开销），但 ts 似乎并没有受到 N 的太大影响。结果，N 和 tq 的增加往往会相互抵消。这个因素与最优 w 公式中的平方根运算符相结合，使得 w 的最优值在跨平台和基准测试中相当稳定。具体来说，我们计算出的三个实验平台的最佳值在 5 和 8 之间。更重要的是，在所有这些平台上，O·L 曲线在 w = 5 和 w = 10 之间变得非常平坦。因此，选择 5 到 10 之间的任何 w 值都会产生高度相似的性能和延迟结果。这个因素在图 14

中很突出，其中 w = 7 和 w = 9 的线彼此非常接近。因此，我们选择 w = 8 作为 eAudit 中的默认值。

解释 2 级缓冲区设计的性能。根据我们的性能模型（表 11）以及图 13 和图 12 中所示的测量，开销有两个主要溯源，即用于在环形缓冲区上对消息进行排队的 tq 和用于向用户层发送信号/唤醒的 ts代理等待数据。重点关注用于我们大部分实验评估的 12 核平台，tq ≈ 0.8μs 和 ts ≈ 2.9μs（图 12 中的蓝色圆形标记）。如果没有本节和前面几节中描述的优化，每个系统调用都会产生 tq 的开销，用于在环形缓冲区上对系统调用记录进行排队，以及用于唤醒用户级代理的 ts 开销，每个系统调用的总开销为 3.7μs。这将每秒可以处理的最大系统调用数量限制为数十万。然而，多核负载（例如我们的基准测试中呈现的负载）每秒可以发出数百万个系统调用，结果是其中大多数都会被丢弃。

我们优化的两级缓冲区设计每个 p 系统调用都会产生 tq 一次，而每个 p·w 系统调用都会产生 ts 一次。通过如上所述调整参数，我们预计每个系统调用的开销将降至 0.8/100 + 2.9/800 ≈ 0.01μs，这意味着 eAudit 可以跟上每秒数千万次的系统调用率。相比之下，不采用此类优化的系统预计会在中等到高强度负载下面临数据丢失，这一预期已通过我们对现有溯源收集系统的实验评估得到证实。

**系统调用优先级**

前两节中描述的技术有助于将日志篡改窗口从以前系统的数万条记录（图 4）显着减少到数百条。之前的工作[77]已经表明，数百个系统调用可能足以执行日志篡改攻击，因此我们在本节中介绍进一步减少此窗口的技术。

我们的方法基于这样的观察：并非所有系统调用都会导致权限升级和/或日志篡改。事实上，并非所有系统调用对于攻击调查都同样重要，例如，execve 的重要性远高于 read 的重要性。因此，我们开发了一种事件优先级方案，进一步减少日志篡改。我们对关键和非关键系统调用的分类与 HARDLOG [6] 类似，并且是由对现实世界漏洞利用 [67]、[85]、[82]、APT 攻击中使用的关键系统调用的相同观察驱动的[19]，以及进行日志篡改攻击所需的权限分析。我们的优先级还基于我们从系统调用级审计数据检测 APT 活动的研究经验 [40]、[70]、[41]。同时，值得注意的是，我们提供的不是单一的预定义优先级方案，而是一个框架，使我们系统的用户能够以适合他们需求的方式定义这些优先级。

尽管我们的优先级是由与 HARDLOG 相同的推理驱动的，但我们的设计在两个重要方面比他们的设计有所改进。首先，eAudit 支持用户可配置的粒度优先级方案，用户可以根据自己的具体需要和要求为每个系统调用分配 256 种可能的权重之一。相比之下，HARDLOG 只有两个预定义的优先级。这意味着无法将具有中等重要性的系统调用（例如用于删除文件的系统调用）优先于不太重要的系统调用（例如读取文件）。

其次，HARDLOG需要专用的硬件设备，而且对内核代码进行了重大更改，以避免将事件发送到用户层。相比之下，eAudit 是纯软件解决方案，不需要更改内核。虽然它没有在关键

系统调用上实现 HARDLOG 的零篡改窗口，但我们的评估表明它已经接近了，即使在非常密集的工作负载上也能实现只有几个系统调用的窗口。我们的设计具体如下。

我们将系统调用分为几个类别，并为每个类别关联一个权重。当单CPU消息缓存中的事件累加权重达到指定阈值Wth时，立即将缓存写入环形缓存，并立即唤醒消费者eLog。换句话说，当满足权重标准时，我们忽略 p 和 w。我们的系统调用分类如下，按权重递减顺序列出：

- 权限提升和篡改：该组由通常涉及权限提升、篡改系统中其他进程、启动恶意软件执行等的系统调用组成。此类别中的关键示例包括 execve、setuid、kill、ptrace 和变体。
- 进程溯源：该组由少量影响进程溯源和代码加载的系统调用组成，包括fork、clone、exit 和mmap。
- 文件名和属性更改：该组包括更改文件持久属性（例如文件名、权限等）的系统调用。它包括重命名、链接、取消链接和 chmod 等系统调用。
- 数据端点创建：下一个重要级别是系统调用，例如open、connect 和accept，它们创建用于读取或写入数据的新端点。
- 数据报网络操作：该组包括系统调用，例如sendto 和recvfrom。
- 文件描述符操作：该组中的系统调用包括dup、dup2、pipe、sockpair、fcntl 等。
- 读取和写入：该组包括read、write、send、readv、pwrite 和recvmsg 等系统调用。
- 其他：这包括不太重要的调用，例如关闭。

每个类别的权重分配以及全局权重阈值 Wth 都是可配置的。在默认设置中，Wth 等于分配给权限提升和篡改类别的权重。这意味着每个关键事件都将立即传播到用户级别并记录，从而最大限度地减少篡改窗口。接下来两个类别的权重设置为 Wth 的八分之一，这意味着在记录之前最多可以在每个 CPU 缓冲区中累积 8 个类别。对于每个连续类别，权重减半。这实际上意味着前面讨论的参数 p 和 w 将主要控制涉及最低类别事件（例如读取和写入）的延迟。

最后，我们设置了每个事件可以在每个 CPU 缓存中缓冲的最大时间。超过此时间限制后，即使缓存仅包含单个事件，缓存内容也会被推送到环形缓冲区。默认情况下，此时间限制设置为 224 纳秒 ≈ 16 毫秒。

## 📌 研究结论

---

## 实验评估

eAudit 与 64 位 x86 处理器上的最新版本的 Linux 兼容。它已经在多个最新版本的

Ubuntu（20.04、21.04 和 22.04）和 Fedora 上进行了测试。它使用 bcc 工具链 [12]。我们发现从源代码 [3] 安装 bcc 更可取，因为它可以跨不同的 Linux 发行版和内核版本工作。一旦以这种方式安装了 bcc，eAudit 的所有依赖项都会得到满足。我们实验的目标是评估 eAudit 的以下方面：

- 在不丢失事件的情况下捕获数据的能力，
- 我们的设计实现的性能改进，
- 运行时开销，
- 日志篡改窗口，以及
- 数据量。

根据其中许多标准，我们将我们的结果与 sysdig 的结果进行比较，sysdig 是现有系统中性能最高的完整溯源收集系统。在第二节中提到，sysdig 配置为使用可打印日志文件格式，以与其他日志记录工具保持一致。然而，对于本节中的比较结果，我们将其配置为使用其二进制（“捕获文件”）格式，因为它更快、更紧凑。

我们使用第 3.1 节中描述的基准。除非另有说明，本节中的所有结果均在具有 16GB 内存和 500GB 固态硬盘、Ubuntu 22.04 和 Linux 内核 5.19 或 6.2 的 i7-12700 系统上获得。该处理器有 8 个性能核心和 4 个效率核心，总共 12 个核心，并配置为禁用超线程。所有基准测试都使用可配置数量的 N 个核心，其中 N 的范围在 1 到 12 之间。对于单线程基准测试（例如 postmark），这意味着运行基准测试的 N 个副本，每个副本都有自己的数据文件和工作目录副本。我们使用前面提到的默认值 p = 100 和 w = 8。环形缓冲区设置为 16 MB，并记录表 1 中的所有系统调用。

**数据丢失比较**

我们运行了所有基准测试，同时将基准测试使用的核心数量从 1 更改为 12。在 eAudit 的情况下，所有基准测试都没有导致任何数据丢失。即使 rdwr（旨在最大限度地提高与溯源相关的系统调用率的基准）也不会导致数据丢失。 （它在 12 个核心上每秒生成 ~ 17M 系统调用。）

Sysdig 数据丢失。如图 15 所示，sysdig 能够在很大程度上跟上内核基准测试，因为该基准测试是 CPU 密集型的，并且进行的系统调用相对较少。其余的基准测试 I/O 密集程度较高，sysdig 无法应对它们。特别是在多核工作负载上，它最终会丢弃大多数事件，最坏的情况大概有超过 95% 的数据丢失。

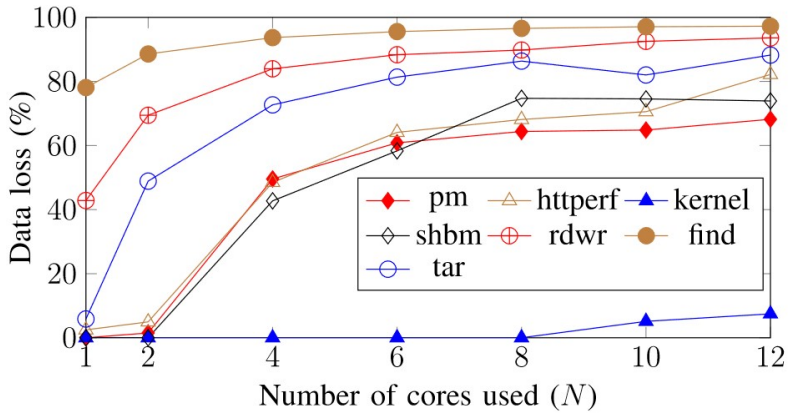


**Fig. 15:** Data loss experienced by Sysdig. *e*Audit does *not* lose any data on any of these benchmarks.

(Sekar 等, p. 11)

Tetragon [18] 和 Tracee [83]、[84] 是两个基于 eBPF 的工具，可以记录系统调用。目前尚不清楚这些工具的设计者是否设想了涉及整个系统溯源记录的用例。尽管如此，为了完整性，我们还是将它们包含在这里。

Tetragon 为用户提供了一种语言来指定要记录的系统调用。该规范的级别有些低，需要识别特定的 Linux 跟踪点和感兴趣的参数的低级别类型信息。支持记录最常见的参数类型，例如整数和文件名，但我们没有找到一种方法来记录更复杂的参数，例如传递到 execve 的字符串数组。我们使用这种规范语言为我们的基准测试中使用的大多数系统调用开发记录器。此外，Tetragon 还附带了一个示例规范，用于记录原始系统调用，记录出现在处理器寄存器中的系统调用号和参数。

Tracee 提供了一个功能齐全的系统调用记录器，使用起来更简单，因为它只需要需要记录的系统调用的规范。与 sysdig 和迄今为止考虑的其他工具类似，Tracee 不需要系统调用参数规范。

如图 16 所示，Tetragon 和 Tracee 都比 sysdig 经历了更多的数据丢失。因此，我们评估了 sysdig 上使用的四个较低强度的基准。虽然 sysdig 在内核基准测试中几乎没有损失，但这两个系统却遭受了重大损失。在邮戳后，这些系统丢弃了大约 90% 的记录，而 sysdig 的记录为 60%。

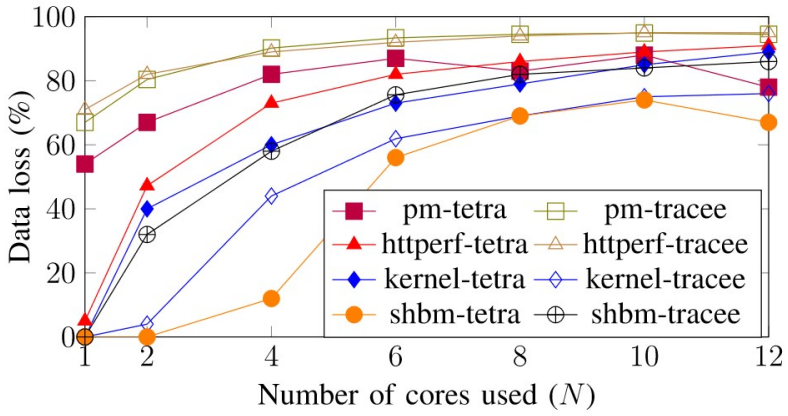


**Fig. 16:** Data loss experienced by Tracee and Tetragon.

(Sekar 等, p. 12)

## 2级缓冲的有效性和参数调整

我们的优化设计大大减少了数据丢失和运行时开销。图 17 中的数据丢失图表显示，未经优化的 eAudit 在许多多核工作负载上将出现超过 50% 的数据丢失。在我们的优化设计中完全消除了这种数据丢失。图 17 的下半部分评估了我们的优化设计所实现的开销减少。它绘制了未优化（p = 1，w = 1）和优化（p = 100，w = 8，优先缓冲）设计之间的代理 CPU 时间比率。当数据丢失时，代理 CPU 时间会低估开销，因此此图表仅考虑没有数据丢失的数据点。在这些基准测试中，我们的优化设计将代理开销降低了 18.4 倍。

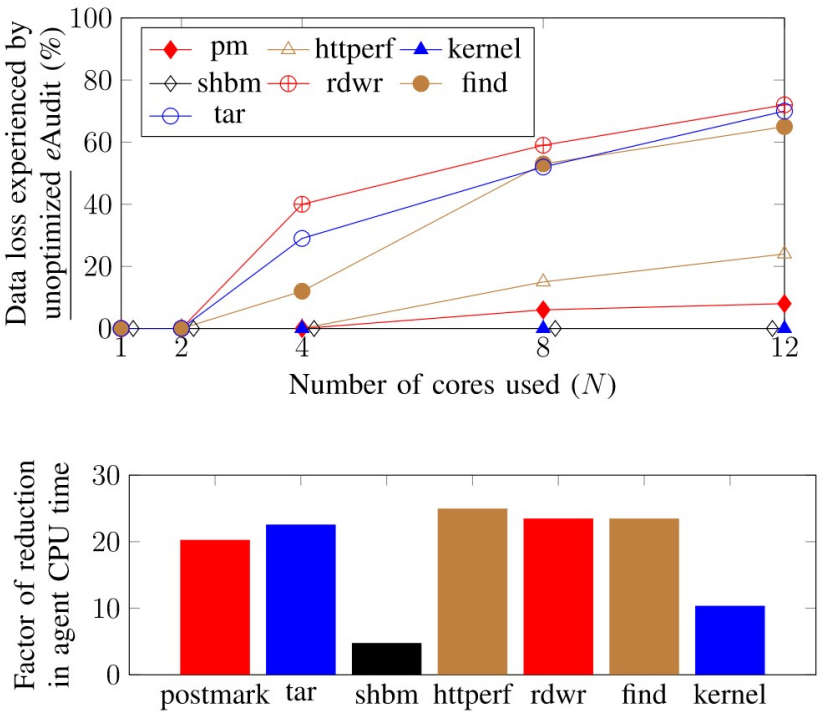


**Fig. 17:** Performance gains of 2-level buffer design and parameter tuning.

(Sekar 等, p. 12)

## 代理开销

代理开销定义为代理与基准 CPU 时间的比率，是我们迄今为止使用的主要性能衡量标准。图 18 将不同基准的开销绘制为所用核心数量的函数。所有这些数据点的平均开销为 3.1%。

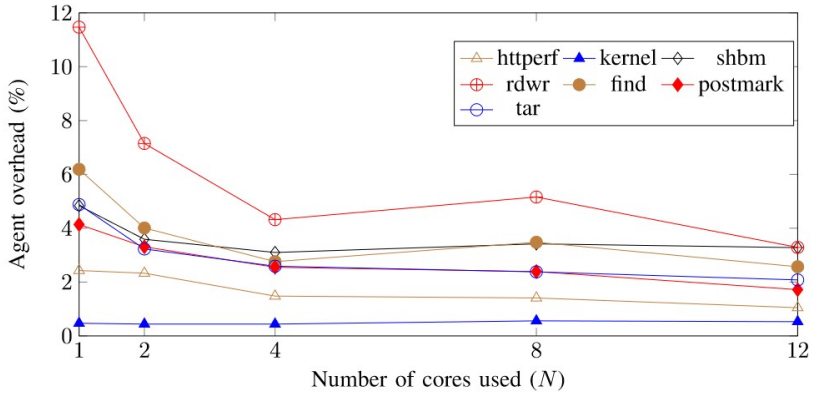


**Fig. 18:** *e*Audit Agent Overhead, defined as the ratio of the CPU time of the agent to the base CPU time of the benchmark.

(Sekar 等, p. 12)

图 18 显示，随着核心数量的增加，代理开销略有下降。这并不意味着与单核相比，代理处理多核工作负载所需的时间更少。相反，这种减少的发生是因为工作负载的可扩展性通常比代理的可扩展性差。为了显示这种效果，我们在图 19 中绘制了每个系统调用的平均开销。这些平均值的中位数是 48 纳秒。

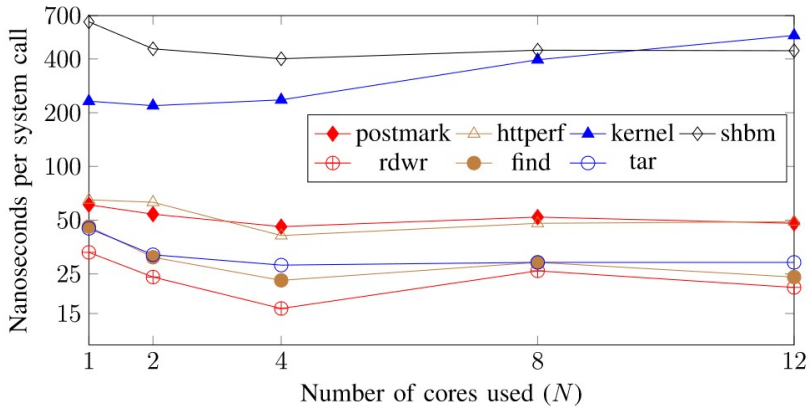


**Fig. 19:** *e*Audit's per-system-call Agent Overhead.

(Sekar 等, p. 13)

这些曲线的平坦度表明代理不会面临太多争用，并且可以很好地适应多核负载。由于 shbm 和内核使用许多具有较大参数值的系统调用，因此它们的每系统调用开销较高。此外，CPU 密集型内核构建的扩展性非常好，超过了 eAudit 的可扩展性。这导致内核基准测试开销呈上升趋势。 （由于 eAudit 在 CPU 密集型负载上的开销非常小，因此这种增加不必担心。）

### 与现有系统比较

由于现有系统在大多数基准测试中都会丢失数据，因此我们限制与第 2 节中使用的相同两个基准测试 postmark 和 shbm 的比较。为了简化我们的测量并更容易重现我们的结果，我们没有像第 2.2 节那样放慢基准测试速度来避免数据丢失。因此，图 20 低估了 CamFlow 等遭受严重数据丢失的系统的开销。

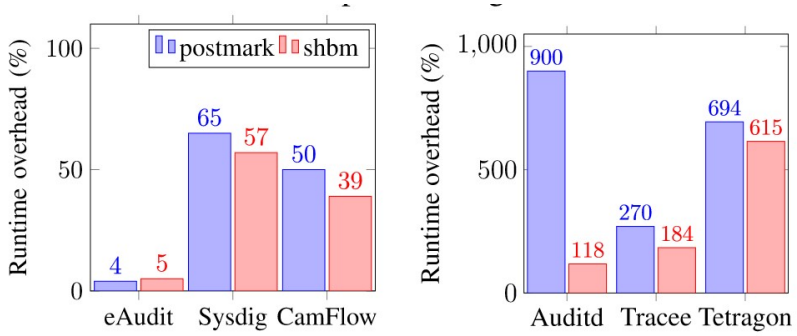


**Fig. 20:** Runtime Overhead of *e*Audit Vs Existing Systems.

(Sekar 等, p. 13)

请注意，eAudit 在这些基准测试中产生 4% 到 5% 的开销，而所有其他系统的开销要高得多。 Tetragon 特别高的开销表明其作者可能并没有打算将其用于连续的系统调用日志记录。也许正因为如此，Tetragon 倾向于生成大量的系统调用记录，其中包含大量重复信息，例如父进程信息、命令行和参数、所有用户 ID 等。

### 日志篡改窗口

在本节中，我们分析 eAudit 的篡改窗口，并将其与 sysdig 的篡改窗口进行比较。对于 sysdig，我们重用了第 2.3 节中的测量方法。在该方法中，基准测试终止和日志文件长度测量之间有一段时间。在此期间写入日志文件的记录不包含在篡改窗口中，因此导致近似值不足。当窗口很大时，这种延迟引入的相对误差是微不足道的，就像第 2 节中讨论的工具的情况一样。但我们发现它明显低估了 eAudit 的日志篡改窗口。因此，我们开发了第二种基于直接检测 eAudit 的方法。

日志篡改攻击成功的影响。虽然篡改窗口大小的小幅减小可能不会产生足够的影响，但 eAudit 实现了 100 倍的减小，这显着提高了日志篡改攻击的门槛。此外，由于我们的系统调用优先级，关键和重要事件的窗口是个位数。我们对 DARPA TC 数据 [19] 中的权限升级攻击的研究表明，重要和关键系统调用的总数（表 22 中的 6-9 个）不足以执行这些攻击。这并不是说这种攻击不可能，但确实很困难。

### 数据量

由于整个系统的溯源数据往往非常庞大，因此最大限度地减少存储需求非常重要。在这方

面，我们将 eAudit 与 sysdig 进行比较。我们在比较中省略了auditd 和CamFlow——如第 2.4 节所述。他们的数据量远远大于sysdig。

eAudit 和 sysdig 都支持比出处数据的打印版本更紧凑的二进制格式，因此我们使用这种格式来测量未压缩的数据大小。我们将比较范围限制为 sysdig 未遇到数据丢失的基准子集。结果如表 23 所示。sysdig 生成的数据量平均比 eAudit 多约 11 倍。表 24 显示使用 gzip 压缩后的日志大小。 Sysdig 日志的可压缩性更高，因此日志大小的差异缩小了。但仍然存在巨大差距，eAudit 日志的大小约为 sysdig 日志的六分之一。

| Benchmark | Syscalls | *e*Audit | Sysdig | Ratio |
|---|---|---|---|---|
| postmark | 32M | 430MB | 5.72GB | *13.3* |
| httperf | 0.8M | 25.3MB | 290MB | *11.5* |
| kernel | 27M | 605MB | 5.03GB | *8.3* |
| Geo. mean | **9M** | **187MB** | **2.03GB** | ***10.8*** |

**Table 23:** Uncompressed Log Size Comparison.

| Benchmark | *e*Audit | Sysdig | Ratio |
|---|---|---|---|
| postmark | 141MB | 653MB | *4.6* |
| httperf | 9.6MB | 116MB | *12* |
| kernel | 178MB | 619MB | *3.5* |
| Geo. mean | **62.3MB** | **361MB** | ***5.78*** |

**Table 24:** Log sizes after gzip compression.

## 讨论

**APT 检测的适用性。**我们使用 eAudit 数据来构建溯源图。我们采用了简单的场景，例如在运行基准测试期间观察到的场景，构建了相应的图表，对其运行查询并进行导航。尽管我们还没有对其检测攻击的有效性进行实验评估，但这是因为已经有大量的研究表明系统调用数据和参数足以检测这些攻击[40]，[38] 、[31]、[70]、[41]、[91]、[100]、[98]、[23]、[101]、[4]、[36]、[8]、[60]、 [13]，[79]。正如表 1 中所讨论的，我们收集的系统调用和参数列表是 TRACE [44] 收集的系统调用和参数列表的超集，研究人员已广泛使用 TRACE 来收集和分析 APT 数据。 eAudit 和 TRACE 都提供所有系统调用参数（除了用于读取、写入等的数据缓冲区参数之外）。还包括其他重要信息，例如时间戳、进程和线程标识符以及序列号。

**局限性和未来的工作。**我们已经证明，eAudit 可以扩展到具有十几个核心的处理器，并且可以维持此类机器上的峰值工作负载。扩展到具有更多内核的处理器可能需要额外的设计措施，例如并行化用户级日志记录过程、使用多个环形缓冲区和/或增加消息缓存的大小。以前的一些溯源收集系统（例如，LPM [11]、HiFi [80]、CamFlow [78]）超越了系统调用接口，可以收集一些与系统调用无关的内核事件。这种能力对于 APT 检测的实际意义尚未确定。就其有用而言，我们注意到 eAudit 在这个方向上还有扩展的余地，因为 eBPF 可以挂

接到 Linux 内核中的许多接口。

## 总结

本文提出的研究确定并分析了现有审计收集系统中的关键瓶颈，包括高性能开销、在持续工作负载下丢弃大部分事件以及日志篡改的大窗口。我们提出了几种新技术来克服这些挑战，包括可以显著减少数据量的紧凑数据编码技术；两级缓冲方案，即使在密集的多核工作负载下也能最大限度地减少争用并避免数据丢失；用于优化调整延迟和吞吐量的分析模型；以及减少日志篡改机会的事件优先级方案。通过有针对性的实验，我们表明我们的技术实现了目标。本文开发的技术可以直接应用于提高其他收集大量数据的基于 eBPF 的系统的性能。它们还适用于涉及收集大量数据并将其发送到用户级别的内核扩展。