RADAR：揭示未记录日志的 NoSQL 操作

Mahfuzul I. Nissan、James Wagner

美国新奥尔良大学，新奥尔良

arXiv:2602.12600v1 [cs.CR] 13 Feb 2026

摘要

随着 NoSQL 数据库的广泛采用，数字取证日益困难，因为存储格式多种多样且通常不透明；当特权内部人员（例如开发运维人员或管理员）能够禁用、抑制或篡改日志记录以掩盖活动时，审计日志也不能被假定为可信。我们提出了 RADAR（记录与工件检测、对齐和报告），这是一种日志对抗感知框架，通过将底层存储工件与高层应用程序日志进行交叉比对，得出取证基准事实。 RADAR分析由自动化 NoSQL 雕刻器（ANOC）重建的工件；该雕刻器推断存储布局并直接从原始磁盘字节中雕刻提取记录，完全绕过数据库 API 和管理系统，从而将物理存储视为独立的证据来源。 RADAR 随后将雕刻提取的工件与审计日志进行核对，以识别差异制品，例如磁盘上存在但逻辑历史中缺失的未记录日志的插入、静默删除和字段级更新。我们在十种 NoSQL 引擎上评估了 RADAR，包括 BerkeleyDB、LMDB、MDBX、etcd、ZODB、Durus、LiteDB、Realm、RavenDB 和 NitriteDB，涵盖键值存储和文档存储，以及多种存储设计，例如写时复制 / 多版本并发控制、B/B+-树和仅追加存储。在日志规避场景下，例如日志抑制和维护后攻击（包括历史字节被清除的情况），RADAR始终能够揭示未归属操作，同时保持 31.7–397 MB/分钟的处理吞吐量，证明了独立于日志且可信的 NoSQL 取证的可行性。

关键词：内部人员威胁、NoSQL 安全、数据库取证、NoSQL 数据库、篡改检测、审计日志验证、存储雕刻

1. 引言

数据库管理系统 (DBMS) 是存储和处理敏感数据的基础。因此，人们投入了大量精力，通过访问控制和审计日志记录来保护这些系统。然而，一旦用户获得提升的权限，无论是合法获得，还是利用漏洞取得，都可能绕过这些安全措施。攻击者可以篡改数据，随后修改或禁用日志记录，以掩盖其活动。

*通讯作者电子邮件地址： minissan@uno.edu(Mahfuzul I. Nissan), jwagner4@uno.edu(James Wagner)

因此，当日志无法完全信任时，取证分析需要可靠的方法来检测违规行为，并收集有关未授权活动的证据。

更可信的证据来源存在于底层存储本身。任何操作，无论是插入、更新还是删除，都会在物理存储层留下痕迹。攻击者虽可修改日志文件，但若要篡改磁盘或 RAM 上的字节级数据库结构而不造成不一致或损坏，难度极大。因此，直接存储分析是提供独立的数据库活动存储层证据的有力方法。

尽管以往研究已探讨关系型数据库的存储分析，但这一挑战在 NoSQL 系统中尤为突出。NoSQL 生态系统涵盖了种类繁多的架构，每种架构都采用专有、未标准化且往往未公开说明的存储格式。拥有足够权限的恶意行为者可以利用这种复杂性。通过修改启动脚本或部署配置，此类行为者可以暂停日志记录、执行未记录操作，然后恢复日志记录配置，从而在审计日志中留下难以通过传统监控发现的空档。

为应对这些挑战，我们提出 RADAR，这是一个用于 NoSQL 数据库取证数据雕刻和日志验证的统一框架。该框架系统地分析底层存储工件，并将其与应用程序审计日志交叉验证。 RADAR 利用我们的自动化 NoSQL 雕刻器 [1] (ANOC)，自动推断存储布局，并从磁盘和内存中雕刻提取记录。通过将这些雕刻提取的痕迹与审计日志进行比较， RADAR能够识别不一致之处，例如未记录日志的修改、静默删除或字段级篡改。本研究的主要贡献如下：

- 我们提出了一种日志对手感知框架（RADAR），将存储雕刻与审计日志验证相结合，以检测 NoSQL 数据库中无法归因的操作（第 4 节）。

- 我们针对不同的存储架构提出了两种检测方法：面向版本化系统的单快照分析，以及面向原位修改系统的比较分析（第 5 节）。

- 我们对相关算法进行了形式化描述，这些算法通过协调雕刻提取的痕迹与审计日志，识别无法归因的插入、删除和更新操作（第 5 节）。

- 我们在十种 NoSQL 数据库（包括键值存储和文档存储）上，在多种篡改场景下进行评估，其中包括维护后攻击和日志抑制（第 6 节）。

2. 相关工作

数据库系统的取证分析涵盖两条长期并行的研究脉络：恢复存储中物理存在的内容，以及审计其据称如何被写入其中。我们回顾关系型数据库和 NoSQL 引擎的存储层重建工作，随后调查日志完整性和数据溯源方面的研究，这些研究促使人们将日志与独立的真实基准进行交叉核验。

2.1. 数据库存储的取证恢复

关系型数据库管理系统提供了完善的页布局文档和集中式目录，例如 MySQL 的 INFORMATION_SCHEMA 或 PostgreSQL 的 pg 目录，调查人员可以利用它们解析页面和行目录 [2, 3]。早期取证工作着重利用这种规律性，从 SQL Server 或 InnoDB 等独立数据库引擎中重建受损或已删除的记录 [4, 5]。在这些特定于数据库的方法基础上，Wagner 等人提出了内部结构雕刻这一更系统化的方法，之后又将其扩展为 DBCarver 等通用工具 [6, 7, 8, 9]。

除页面雕刻外，其他研究也扩展了取证工具集。Frühwirt 等人通过分析 InnoDB 重做日志重建数据操作查询，展示了预写日志制品在关系型数据库调查中的价值 [5]。文件雕刻研究也对该领域产生了影响：Garfinkel 将雕刻对象的快速验证方法系统化，Poisel 和 Tjoa 则对雕刻技术进行了全面综述 [10, 11]。近期，Lenard 等人提出了 Sysgen，这是一种用于取证方法合成测试的系统状态语料生成器，有助于为评估建立受控基线 [12]。

NoSQL 引擎打破了这些假设。存储设计差异很大：Berkeley DB、LMDB 和 WiredTiger 依赖基于页的 B/B$^+$-树结构，而 ZODB 使用序列化对象流 [13, 14, 15]。缺乏统一目录，使得重建比在关系系统中更加困难。因此，先前的工作主要聚焦于单个数据库，例如针对 MongoDB 或 Redis 的案例研究，而每种数据库都需要专门的工具 [16, 17, 18, 19]。为应对这种异质性，我们此前的工作提出了 ANOC，它可以动态推断结构参数，并直接从原始字节重建记录 [1]。虽然 ANOC 确认了数据在物理上的存在，但它无法根据声明的操作历史判断这些数据是否合法。

2.2. 日志分析、完整性与篡改检测

审计日志仍然是重建用户操作时间线最常见的来源。然而，在对抗环境中，其可靠性受到限制，因为特权攻击者可以抑制或选择性地编辑条目 [20, 21]。这一挑战在 NoSQL 生态系统中尤为突出。不同于许多提供强大内置审计框架的关系型数据库，NoSQL

系统通常缺乏原生日志记录能力，或将其限制为仅限企业版功能。这迫使应用程序开发者负责生成可验证的审计轨迹，从而产生异构的应用级日志，更容易被绕过和篡改。

研究人员早已认识到这一风险，并提出了多种保障日志完整性的解决方案。Schneier 和 Kelsey 开创性地提出了安全的前向安全审计日志，以抵御追溯性修改 [22]。随后，Crosby 和 Wallach 利用 Merkle树链式结构开发了高效的防篡改数据结构 [23]。与此同时，标准制定机构发布了相关指南，其中 NIST 的 SP 800-92 强调了集中式日志管理和完整性验证 [24]。硬件辅助设计则提供了另一条途径，通过使用可信执行环境或专用硬件来创建防篡改审计追踪 [25, 26], ，不过这类方案依赖专门的部署。

另一条互补的研究路线聚焦于数据溯源。基于数据溯源的入侵检测系统会捕获系统级事件之间的因果关系。CamFlow 展示了实用的全系统溯源捕获能力，而 HOLMES 则利用图关联检测高级持续性威胁 [27, 28]。此类系统能够揭示操作系统层面的不一致，但尚未验证记录级数据库内容。这一差距促使研究者探索将审计日志与独立物理证据进行比对的方法。

2.3. 统一取证方法

尽管传统上人们一直将存储伪影与审计日志分开研究，但越来越多的人认识到，将二者结合能够加强篡改检测。Wagner 等人证明，页面级提取能够揭示关系系统中的文件篡改[29]。近期的数据库内存取证研究则通过重建查询语义或缓存访问模式，提供并行的验证流 [30, 31, 32]。基于这些见解，我们的 RADAR 框架首次系统地将通用的、经数据雕刻提取的 NoSQL工件与规范化应用审计日志进行关联，以识别未归属操作。通过协调两条独立的证据流， RADAR 能够检测未记录的删除、静默更新或恶意插入等异常；若仅采用存储雕刻或日志审计，这些异常将无法被发现。

3. 威胁模型

在典型的 NoSQL部署中，多个运维角色会在不同的权限层级与系统交互，如图 1 所示。我们根据这些角色的操作范围和权限边界，将其区分为三类：应用级数据库用户（ALDU）、DevOps 工程师（部署操作员，DO）和系统管理员（SA）。ALDU 是

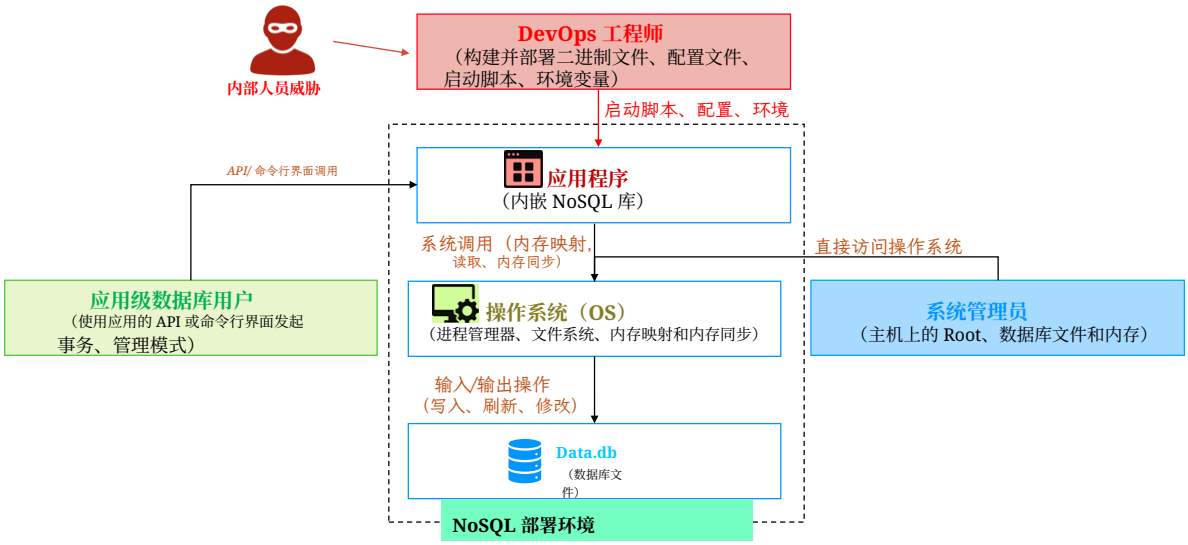


图 1：威胁模型

一名合法操作员，只通过应用程序提供的接口（例如 API 或命令行客户端）与数据库交互。该角色可以根据应用程序逻辑的许可执行查询和事务，所有操作都受应用程序强制实施的访问控制和审计机制约束。在本文中，ALDU 无法修改部署制品、直接访问存储文件或更改操作系统配置，并被视为非恶意行为者。相比之下，DO 负责构建和部署 NoSQL 应用程序，包括其二进制文件、配置文件和启动脚本。通过控制运行时实例化和运行参数，恶意 DO 可以实施绕过应用层审计的部署层篡改。然而，DO 不具备 root 权限或内核级权限，这些权限由 SA 掌控。SA 以完整的 root 权限管理主机操作系统，负责进程控制、文件系统权限和存储设备，并负责安装操作系统级完整性监控和快照机制。虽然 SA 可以直接访问数据库存储文件，但不控制构建或部署流程。在我们的模型中，SA 是可信角色，负责独立于应用层维护主机级监控。

鉴于这些权限边界，本模型中的对手是部署操作员。他们对部署流水线的访问可促成三种主要滥用形式：

1. **配置滥用：** 操纵配置以禁用或降低审计日志记录效果，执行未经授权的操作，然后恢复原始设置。

2. **植入后门的部署：** 部署植入后门的应用程序二进制文件，在执行隐蔽写入的同时抑制或伪造审计事件。

3. **带外修改：** 执行带外脚本或工具，直接修改数据库文件，绕过官方应用程序进程。

每种途径都会规避应用层控制，但不可避免地会在操作系统边界留下可观察到的副作用。例如，意外进程执行文件写入、二进制文件完整性不匹配，以及入站请求与写入字节之间出现异常偏差。关键在于，由于部署操作员不具备系统管理员级权限，因此无法在不超出其指定角色权限的情况下压制或删除这些操作系统级痕迹。

在这种对手模型下，应用程序和 NoSQL 层审计日志并非可靠的信息来源。由部署操作员控制的部署时配置可以暂停、重定向或降低日志记录采样率（例如，更改容器日志驱动程序或服务单元），从而产生看似常规但覆盖不全的部署。许多引擎会将明文或 JSON 日志写入普通文件或外部日志接收端，却未将日志与正在执行的二进制文件或可信时间进行加密绑定，因此有权限的部署人员可以截断日志或选择性地修改日志。此外，环境和加载器操纵（例如，使用预加载库或包装启动器）可以在不修改应用程序本身的情况下介入日志记录路径。因此，日志可能缺失、遭到选择性压制或被重写。在我们的模型中，日志仅作为佐证信号，而证据的完整性则由通过ANOC恢复的存储层工件来确定。

为应对这些威胁，我们假设 SA 能够借助主机控制的遥测和变更控制策略，在应用程序之外触发取证快照采集。在实践中，SA 可以依靠操作系统级审计或来源追踪信号（例如，Linux `auditd`/`fanotify`/ eBPF；Windows SACL / ETW / Sysmon），将数据库文件修改归因于执行写入的进程（例如，PID、UID / SID、可执行文件路径以及文件路径或索引节点）。如果条件允许，可使用 Linux IMA 或 Windows 代码完整性控制（例如 WDAC）等主机机制验证可执行文件完整性，但 `RADAR` 不要求提供逐次写入的哈希保证。操作系统还提供原生快照机制（例如 Linux 上的 LVM / ZFS / Btrfs，或 Windows 上的卷影复制）。SA 会对变更控制规则进行编码，并在策略表明取证风险升高时触发快照。

在此模型下，来自已批准数据库服务且处于预期工作负载条件下的写入被视为正常操作，可以记录日志而无需进行快照采集。与已批准变更请求相对应、并附带已声明迁移标识符的已授权批量变更写入被视为维护操作，并可触发信息性快照，以供文档记录和事后审计。任何写入

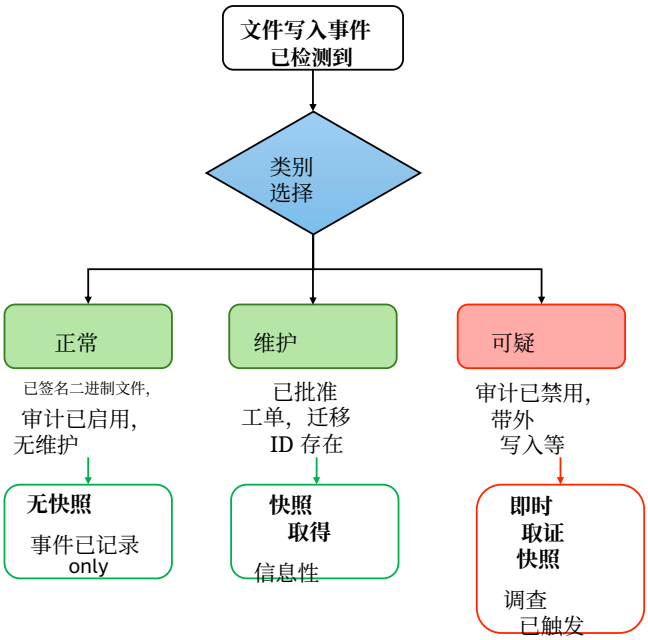


图2：文件写入事件分类及相应的快照决策。

若违反数据溯源或变更控制要求（例如，非服务写入者、审计抑制/切换、完整性不匹配、维护窗口之外的意外服务重启，或无法解释的漂移异常），则被视为可疑，并立即触发快照采集。重要的是，可疑性由数据溯源或控制平面策略方面的违规来界定，而非仅凭原始写入量，从而能够以更高保真度触发快照，同时避免良性活动使系统不堪重负。

图2和表 1总结了这种分类及快照触发流程。当 SA 收到表明有内容写入数据库文件的操作系统审计事件时（例如来自 `auditd` 或 Sysmon），会根据数据溯源和策略检查将该事件归入三类之一（正常、维护或可疑）。此分类决定该事件是仅记录日志、生成信息性快照，还是触发即时取证快照采集并随后进行调查。

例如，表 2 展示了适用于 Linux 和 Windows 的 SA自有配置示例，可检测后门应用程序写入和带外修改。在 Linux 上，不可变审计规则会监控数据库文件中的任何写入或属性变更事件，并将其标记以供监控。其他规则会标记来自用户标识符与授权服务账户的用户标识符不同的进程的写入尝试，从而确保只有批准的二进制文件才能修改数据库。随后，审计配置会被锁定，直到重新启动，以防止篡改。在 Windows 上，成功的写入操作会启用文件系统审计，并生成安全事件4663条目。如果写入者的安全标识符与授权

类别	条件	快照 操作	用途
正常	已批准、已签名二进制文件；审计已启用；工作负载处于预期范围内；没有维护窗口	仅记录日志	尽量减少假阳性 来自常规操作
维护	已批准的变更单；已签名二进制文件；审计已启用；已声明迁移标识符	快照（信息性）	文档已授权变更以确保取证完整性
可疑	以下任一情况：审计已禁用/已切换；启动时二进制文件不匹配；非服务写入者（带外）；意外窗口外的停止/启动或配置变更窗口；未关联工单的漂移异常；写入受保护的维护窗口外的命名空间	立即取证快照	捕获潜在的内部人员篡改以供调查

表 1：快照触发策略

平台	命令	触发条件	用途
Linux	`auditctl -w /var/lib/db/data.db -p wa -k db_watch`	写入或属性更改数据库文件	将事件标记为 `db_watch` 以便监控
Linux	`auditctl -a always,exit -F arch=b64 -S openat,openat2 -F dir=/var/lib/db -F perm=w -F uid!=1001 -k oob_db_write`	由以下对象尝试写入未经授权的用户标识符	检测带外写入未经授权的进程
Linux	`auditctl -e 2`	审计配置更改	锁定审计规则，直至重新启动，以防止篡改
Windows	`AuditPol /set /subcategory:"File System" /success:enable`	文件写入成功操作	启用审计以监控写入；触发卷影复制以监控未经授权的写入者

表 2：Linux 和 Windows 的快照触发配置示例

服务账户不匹配，则可将这些事件与卷影复制触发器关联，确保可疑写入会自动生成取证快照。

这些配置在 SA 控制下应用于主机层，因此能够抵御没有提升权限的恶意 DevOps 工程师的绕过尝试。Root 或内核级攻击（例如伪造 inode 更改时间、卸载内核钩子或直接写入原始存储设备）不在本文的讨论范围内；此类攻击通过安全启动、内核锁定以及管理访问的双人控制来缓解。

4. RADAR 框架与数据源

RADAR框架提供了一套端到端流程，通过关联底层存储工件与高级审计日志，检测未归属（未记录）的操作。其主要功能是识别从存储中提取的物理证据与审计日志中记录的逻辑历史之间的差异；这些差异可能是威胁模型中所述对手掩盖未授权活动的有力迹象。

4.1. 总体架构

RADAR 工作流如图3所示，遵循一个结构化的多阶段流程。流程首先获取数据库的取证快照，其中包含根据第 3 节定义的主机级策略自动捕获的相关数据库文件或文件集。随后，框架的雕刻引擎会处理这些快照，以提取原始物理记录。这个关键步骤由我们的自动化 NoSQL 雕刻器（ANOC）执行。它是一种通用工具，旨在直接从原始磁盘或内存映像中重建数据库内容，而无需调用数据库的 API。如第 4.3 节进一步详述的那样，ANOC 由 INFERNOS 算法引导，可根据原始字节自动推断存储布局。此过程会生成一组结构化的雕刻提取的痕迹，其中包含对账阶段所需的元数据。对于 Berkeley DB 这类基于页面的数据库，其中包括用于比较分析的页级哈希值；对于 ZODB 这类日志结构型数据库，其中则包括每条记录的历史版本。

与此同时，系统会收集应用层审计日志，将其作为已声明操作的正式记录。最后，RADAR 分析引擎会导入来自 ANOC 的雕刻提取的痕迹和审计日志。作为框架的核心，RADAR 会尝试将每个物理制品与日志中的相应逻辑操作进行匹配，以此进行交叉验证。任何无法由日志解释的雕刻记录都会被标记为未归因操作，或称“差异制品”。具体数据源详见第 4.2 节，协调逻辑见第 5 节。

4.2. 取证数据源

RADAR 的威胁检测方法会协调两条独立的证据流：从取证快照中雕刻提取的存储层工件，代表**物理真实情况**；以及应用层审计日志，记录**预期操作的声明历史记录**。本节将定义我们接受的日志流，以及如何对其进行规范化以供分析；存储流和数据雕刻过程将在第 4.3 节介绍。

具体而言，第4.2.1节阐述了采用应用程序生成的 JSON Lines（JSONL）审计日志的理由，并详细说明了使



图3：RADAR 的架构

每条日志条目都能与雕刻记录进行比较所需的模式和规范化规则。最后，第4.2.2节说明了有意排除特定于引擎的预写式日志和日志的理由，因为它们性质各异、可有可无且容易被抑制，因此不适合作为主要证据来源。

这种方法使框架既能保持通用性，又具备韧性。提取存储提供独立于日志的真实基准，而标准化的应用程序日志则提供声明的历史记录。RADAR 的分析引擎会交叉核对两者，并将任何缺少相应日志说明的雕刻记录报告为未归因操作。

4.2.1. 应用程序审计日志（*JSONL*）

关系数据库通常具备强大的内置审计功能，而 NoSQL 领域在安全准备方面却存在明显缺口。许多 NoSQL 数据库完全不具备原生审计日志功能 [33, 34]。在其他一些重要系统中，这项关键安全功能仅作为高级企业级附加组件提供，免费版或社区版用户因此无法使用内置解决方案。因此，由于缺少易于获取的标准化审计机制，生成安全且可验证日志的责任便落回到应用层。

为填补这一空白，业界广泛采用 JSON Lines（JSONL）作为事实上的标准，以创建可移植且透明的审计追踪 [35]。因此，我们的工作采用 JSONL 格式，使我们的方法与安全监控领域的既定实践保持一致。在我们的威胁模型中，此应用程序审计被视为预期操作的权威记录，而 WAL 或日志等引擎管理的日志则不在考虑范围内。JSONL 非常适合此用途。其面向流、仅追加的特性非常适合持续数据流。就数据完整性而言，其逐行结构（每行都是一个独立对象）使日志文件比单体式 JSON 文件更能抵御损坏 [35, 36]。就合规性而言，这种格式非常适合生成萨班斯—奥克斯利法案和 HIPAA 等法规要求的、包含丰富上下文信息的事件记录 [37, 24]。

模式。 每一行日志 e 都是一个包含六个字段的 JSON 对象：

$$e = \{\texttt{ts}, \texttt{op}, \texttt{key}, \texttt{user}, \texttt{old_value}, \texttt{new_value}\},$$

其中 `op` 指定**操作** 类型，取值为{`insert`, `delete`, `update`,`update_fields`, `delete_fields`}之一。对于每个写入者，**时间戳** `ts` 单调递增：每个新条目的时间必须等于或晚于前一条目，当多个操作发生在同一秒内时允许时间相同。例如，同一批次中提交的两次插入可能具有相同的时间戳，或具有连续的秒级时间戳（例如 12:00:01、12:00:02）。时间戳仅用作排序辅助，因为协调依赖于键值一致，而非仅依赖时间。我们同时支持 UNIX纪元（自 1970 年起的整数秒）和 RFC3339（类ISO字符串）编码。 `user` 字段包含执行该操作的用户标识符，并支持归因，但它并非授权所必需。 `key` 字段保存记录的唯一标识符。

在键值存储中， `old_value` 和 `new_value`字段以标量字符串形式包含操作前后的记录状态（例如 `"CANADA_00001"`）。在文档存储中，这些值可以显示为原生JSON对象，或以带转义引号的序列化JSON字符串形式出现（例如 `{"_id":"part#000002000",...}`）；两种编码均受支持。

规范化。 值通过确定性的规范化函数 canon(·) 进行比较，以避免因 JSON键顺序、空白字符或存储元数据导致的无关差异。当值以 JSON 字符串形式到达时，规范化步骤会对其进行解析，并将其转换为键已排序且空白字符最少的规范JSON。当值以 JSON 对象形式到达时，同样的步骤会直接将其转换为规范JSON。标量字符串按字节逐一进行相等性比较。匹配前可以进行键规范化（例如大小写折叠）。 `status`、 `page_id`、 `page_MD5`和 `page_offset` 等引擎元数据字段会从相等性测试中排除，以防止因布局或页哈希变化而产生假阳性。[1] 对于文档存储，比较使用完整的规范文档。除非审计本身是字段级的，否则不会使用字段级差异（见第5.4节）。在这种情况下，结果中的匹配项会标记为“已授权（字段级）”。

4.2.2. 排除的引擎专用日志（预写日志/日志）

尽管预写日志是具有取证价值的工件，但出于若干原因， RADAR框架有意不将其作为主要证据来源。首先，预写日志格式

[1]这与取证实践一致：该实践不会将纯粹的物理重排列为逻辑编辑。

在各 NoSQL 系统中往往是专有的、二进制的且缺乏文档，需要专门的、特定于数据库的工具才能可靠解析。为通用框架开发并维护此类解析器超出了本研究的范围。其次，许多 NoSQL 数据库（如 LMDB 和 MDBX ）根本不实现预写日志，而其他数据库（包括 ZODB 和 Berkeley DB）则允许在某些配置下禁用或绕过日志记录或事务日志记录，例如出于性能优化或批量数据操作的需要。即便在为实现持久性而实施事务日志记录的引擎中，例如带有预写日志的 RavenDB，或使用 Raft 日志在集群节点之间复制并持久化操作的 etcd，特权管理员仍可强制进行日志轮转并删除较旧的日志文件，从而抹去未授权操作的证据。此外，LiteDB 和 RealmDB 不提供传统的持久化预写日志，而 NitriteDB 虽然支持一定的日志记录，却依赖非持久且缺乏防篡改能力的应用级日志。鉴于这些复杂性，以及预写日志制品很容易缺失、被更改或被抑制， RADAR 框架转而强调对持久化存储工件的分析，后者为恶意活动检测提供了更普遍适用且更具韧性的基础。

4.3. 采集与雕刻 *(ANOC)*

RADAR 的分析始于由 Automated NoSQL Carver(ANOC) 经数据雕刻生成的数据。正如我们先前的工作 [1], 所详述，ANOC 是一种新颖的工具，旨在不依赖数据库 API 或日志，从原始字节重建数据库内容。它分两个阶段运行。第一阶段是一次性参数收集器，它利用 INFERNOS （NoSQL 存储推断）算法。INFERNOS 通过对原始二进制字节应用一系列统计分析和启发式方法，自动完成数据库存储架构的逆向工程。它无需事先了解具体数据库类型，即可推断页大小、记录目录和记录编码格式等关键参数。第二阶段是只读 雕刻器 ，它应用 INFERNOS 生成的结构配置文件，从目标映像中提取记录。

RADAR 利用这些雕刻结果，为数据库状态创建统一的逻辑视图，以便与审计日志进行协调。对于每个雕刻项，该视图都提供三个必要组成部分：一个与审计日志中 key 字段兼容的 correlation key、一个用于准确比较内容的 canonicalized value ，以及一个 status indicator（例如“活动”或“已删除”），用于区分当前记录与历史残留。RADAR 的归因逻辑依赖这三个组成部分，同时也会利用其他雕刻元数据，例如原位修改数据库的页级哈希值，以进行初步筛选并高效识别已修改页面。

ANOC 依赖学习得到的配置文件，而非硬编码布局，因此具有很强的可扩展性，并已成功应用于十种具有代表性的 NoSQL 引擎，包括键值存储（Berkeley DB、LMDB、MDBX、ZODB、Durus、etcd）和文档存储（RavenDB、LiteDB、Realm、NitriteDB）。雕刻器通过标准化接口输出这些数据，这一设计使 RADAR 与引擎内部实现保持解耦。这些雕刻结果是 RADAR 从存储中获取数据所需的唯一输入。对于保留历史版本的引擎，RADAR 会进行单快照分析（第5.2节）；对于执行原位更新的引擎，则会对*之前* 和*之后* 的快照对进行比较分析（第5.3节）。在所有情况下，最终都会依据第5.4节详述的判定规则，与应用程序审计日志进行协调。

5. 威胁检测方法

5.1. NoSQL 存储中的取证痕迹

对 NoSQL 系统进行取证分析，基于一个基本原则：每项逻辑数据库操作都会在持久化存储层留下相应的物理痕迹。由于这些底层改动无法轻易绕过，因此可将其作为可靠的真实基准进行验证。*存储残留* 原理确保残留制品会一直存在，直到被物理覆盖，而这些残留数据可通过数据雕刻恢复。不过，这些制品的持久性和可恢复性在很大程度上取决于引擎底层的存储架构。这些架构分为两类：采用仅追加和写时复制（CoW）的系统，以及采用原位修改的系统。

仅追加与写时复制（*CoW*）系统。 这些引擎通过写入记录或数据块的新版本来处理修改，同时让原始数据在物理上保持完整。这种方法会在数据库文件本身中保留完整的历史状态记录。已删除记录不会被擦除，而只是被更新的条目取代，因此可直接从单个取证快照中恢复。

原位修改系统。相比之下，这些引擎通过覆盖现有记录槽或页中的数据来执行更新，因此先前的值不会作为可寻址制品持久保留在数据库文件中。删除操作通常会使记录槽失效，或将空间返还给分配器（标记为空闲）。虽然旧字节可能会暂时留存在未分配空间中，但它们已无引用，随时可能被覆盖。因此， RADAR 不会依赖这些不确定的制品。

因此，为了应对痕迹保留方面的这一差异， RADAR采用了两种不同的方法，分别针对底层

存储架构的特点专门设计：对于保留历史记录的系统，采用单快照分析（Method1）；对于不保留历史记录的系统，则采用比较分析（Method 2）。第 5.2 节和第 5.3 节详细介绍了这些方法，以及通过交叉比对雕刻提取的痕迹与审计日志来检测未归因操作所使用的核对规则

5.2. 方法*1*：仅追加和 *CoW* 数据库的单快照分析

对于保留历史版本的存储架构，例如仅追加或写时复制（CoW）系统， RADAR 可以通过分析单个取证快照来检测未归属操作。该快照包含活动记录和历史记录，因此可以全面重建数据库的状态转换。此方法旨在验证每次状态转换都有应用程序审计日志中的条目作为依据。分析分为三个阶段，分别由算法1、2和3形式化描述。

首先，框架会识别所有状态为“已删除”的雕刻记录，并将其中无法与日志中相应的“删除”操作或“更新”操作的“旧值”匹配的记录标记为 Unattributed Delete。其次，框架会检查所有“活动”记录，并将其中无法对应日志中“插入”操作或“更新”操作的“新值”的记录标记为 Unattributed Insert。最后，由于一次未记录的更新会针对同一个键同时产生一条未归属的已删除记录（旧值）和一条未归属的活动记录（新值），第三个算法会关联这些发现，并将这类记录对合并为单个 Unattributed Update。

5.2.1. 检测无归属删除

分析的第一阶段会找出数据库中标记为已删除、但审计日志中没有记录其删除操作的记录。检测过程会将这些雕刻残留与日志进行交叉比对，以找出差异，具体如算法 1 所示。对于键值存储，值通常是标量；对于文档存储，值则是整个文档对象。由于这些系统中的更新操作实际上会删除文档的旧版本，因此该算法会将删除时的数据完整状态与日志中的删除和更新操作进行比对验证。

算法 1： 检测未归因删除

输入： C：雕刻记录集
输入： L：审计日志
输出： Δ_{del}：未归因删除

1 $\Delta_{\text{del}} \leftarrow \emptyset$;
2 $S \leftarrow \emptyset$;
3 **遍历** 记录 $r \in C$ **部署操作员**
4 **如果** $r.status = Delete \wedge (r.key,\ r.value) \notin S \wedge \neg\exists l \in L : ((l.op = delete \wedge l.key = r.key \wedge l.value = r.value) \vee (l.op = update \wedge l.key = r.key \wedge l.old_value = r.value))$ **则**
5 $\Delta_{\text{del}} \leftarrow \Delta_{\text{del}} \cup \{r\}$;
6 $S \leftarrow S \cup \{(r.\texttt{key}, r.\texttt{value})\}$;
7 **end**
8 **结束**
9 **返回** Δ_{del};

算法以雕刻记录集 C 和审计日志 L为输入，生成集合 Δ_{del}，其中包含所有在日志中缺少归因的已删除记录。集合 Δ_{del} 和S 初始化为空（第 1–2 行），其中 S 用于跟踪已处理的键值对并防止重复报告。主循环会逐一遍历 C中的每条雕刻记录 r （第 3 行）。对于每条记录，第 4 行的条件判断会验证该记录的状态是否为 Delete，其（key、 value）键值对是否尚未处理（即不在 S中），以及 L中是否不存在任何能够说明该删除操作的日志条目 l 。具体而言，只有当日志中存在一个键和值均相同的显式 delete 操作，或者存在一个键相同且 old_value匹配的 update 操作时，才能确立归因。如果这两个条件都不成立，则将记录 r 添加到 Δ_{del} （第 5 行），并将其（key、 value）键值对添加到S（第 6 行），以确保后续迭代中的唯一性。评估完 C 中的所有记录后，算法返回集合 Δ_{del} （第 9 行），其中包含在审计日志中没有对应操作的唯一已删除记录。

对于键值存储， value 字段通常保存标量。对于文档存储，值是完整的文档（例如，JSON 对象），因此算法会确保将已删除状态的全部内容进行比较，以进行归因。此方法通过确保仅将无法解释且非冗余的已删除记录报告为未归因记录，支持精确的取证重建。

示例（键值存储）。如图4所示，通过交叉比对审计日志和 ANOC 输出，识别无归属删除。通过匹配日志条目，可以解释（K001, Austin）和（K002, Houston）这两条已删除记录。然而，（K004, Plano）和（K006, Memphis）这两条已删除记录没有任何日志条目作为依据，因为没有

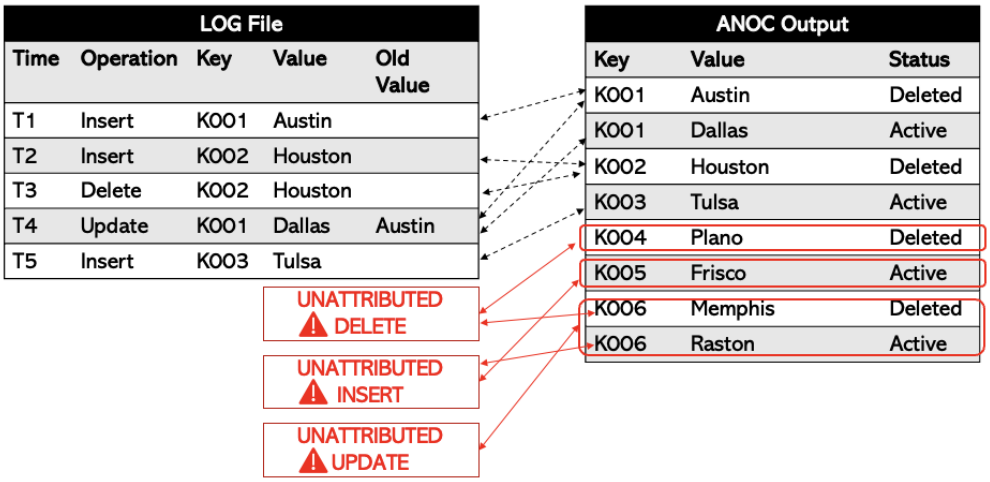


图4：未归属操作（键值存储）

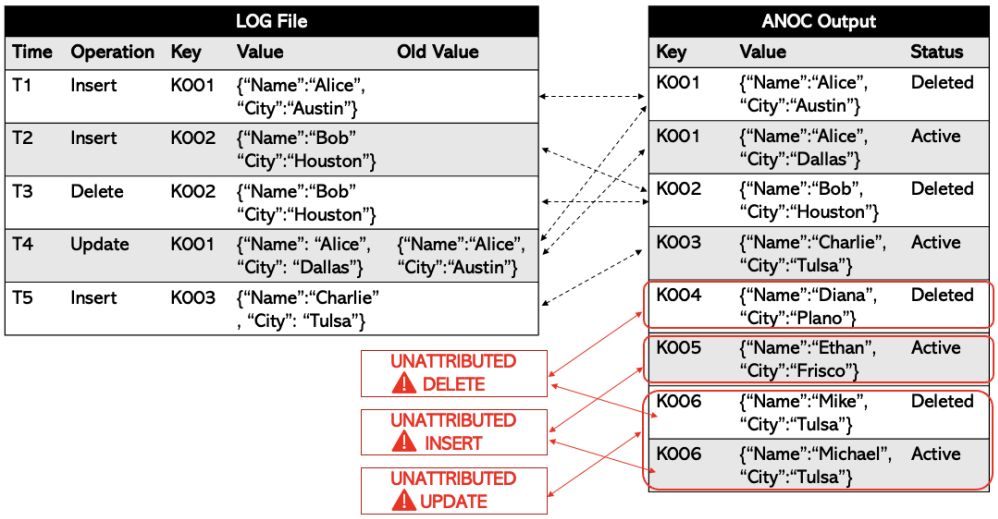


图5：未归属操作（文档存储）

与之对应的删除或更新（`old_value`）操作。根据算法 1，因此将它们报告为无归属删除。

示例（文档存储）。在文档存储场景中（见图5），每个值都是一份完整的文档。日志中记录了 `K001` 和 `K002` 对应的已删除记录。然而，已删除的文档（`K004`, `{"Name":"Diana", "City":"Plano"}`）和（`K006`,`{"Name":"Mike", "City":"Tulsa"}`）无法由任何日志条目解释。因此，算法将这些文档标记为未归因删除。

5.2.2. 检测未归因插入

分析的第二阶段聚焦于由 ANOC 提取的“活动”记录。这些记录可能代表合法插入、更新后的状态，或由

未记录日志的恶意进程创建的记录。为区分这些可能性，必须将每条提取出的活动记录与审计日志中的“插入”和“更新”操作进行交叉比对。只有当某条活动记录无法由这两类已记录的操作中的任何一类解释时，才会将其标记为无归属插入，具体定义见算法 2。

算法 2： 检测未归因插入

输入： C：雕刻记录集
输入： L：审计日志
输出： Δ_{ins}：未归因插入

1 $\Delta_{\text{ins}} \leftarrow \emptyset$;
2 $S \leftarrow \emptyset$;
3 **对每条**记录 $r \in C$ **执行**
4 **如果** $r.\mathit{status} = \mathit{Active} \wedge (r.\mathit{key},\ r.\mathit{value}) \notin S \wedge \neg\exists\, l \in L : ((l.\mathit{op} = \mathit{insert} \wedge l.\mathit{key} = r.\mathit{key} \wedge l.\mathit{value} = r.\mathit{value}) \vee (l.\mathit{op} = \mathit{update} \wedge l.\mathit{key} = r.\mathit{key} \wedge l.\mathit{new_value} = r.\mathit{value}))$ **则**
5 $\Delta_{\text{ins}} \leftarrow \Delta_{\text{ins}} \cup \{r\}$;
6 $S \leftarrow S \cup \{(r.\texttt{key}, r.\texttt{value})\}$;
7 **end**
8 **结束**
9 **返回** Δ_{ins};

该算法接收雕刻记录集 C 和审计日志 L作为输入，并生成集合 Δ_{ins}，其中包含所有在日志中缺少归因的活动记录。集合 Δ_{ins} 和S 初始化为空（第 1–2 行），其中 S 用于跟踪已处理的键值对并防止重复报告。主循环遍历 C中的每条雕刻记录 r （第 3 行）。对于每条记录，第 4 行的条件语句会验证该记录的状态是否为 `Active`、其 (`key`, `value`) 对是否尚未处理（即不在 S中），并且不存在任何可说明该插入的日志条目 l （属于 L ）。具体而言，只有当日志中存在键和值均相同的显式 `insert` 操作，或者存在键相同且 `new_value`匹配的 `update` 操作时，才能确立归因。如果这两个条件均不成立，则将记录 r 添加到 Δ_{ins} （第 5 行），并将其 (`key`, `value`) 对添加到S （第 6 行），以确保后续迭代中的唯一性。评估完 C 中的所有记录后，算法返回集合 Δ_{ins} （第 9 行），该集合表示审计日志中不存在相应操作的唯一活动记录。

对于键值存储， `value` 字段通常存放标量。对于文档存储，该值是完整的文档（例如，JSON 对象），因此算法会确保将整个活动状态纳入比较以进行归因。通过确保仅将无法解释且不冗余的活动记录报告为未归因插入，这种方法支持精确的取证重建。

示例（键值存储）。如图4所示，ANOC 的输出会与审计日志交叉比对，以识别无法解释的活动记录。对于每个处于 Active状态的（key，值）对，都会检查日志中是否存在匹配的 insert 或 update（new_value）操作。诸如（K001， Dallas）和（K003， Tulsa）之类的记录可由审计日志解释，而活动记录（K005， Frisco）和（K006， Raston）则没有对应的日志条目。因此，按照算法 2 的规定，这些记录会被标记为未归因插入。

示例（文档存储）。如图5所示，ANOC 的输出从存储中重建出一组活动文档。系统将每个活动的（key，文档）对与审计日志进行比较，以确定其存在是否有相应的 insert 或 update（new_value）操作作为依据。大多数活动文档都能由匹配的日志条目解释，但记录（K005，{"Name":"Ethan", "City":"Frisco"}）和（K006，{"Name":"Michael", "City":"Tulsa"}）虽作为活动记录出现在存储中，却没有对应的日志操作。根据算法 2，这些记录会被报告为未归因插入。

5.2.3. 检测未归因更新

在现代 NoSQL 系统中，尤其是采用仅追加或写时复制架构的系统，一次 UPDATE 操作通常会为同一键留下两处取证痕迹：一个已删除的“之前”版本（旧值）和一个活动的“之后”版本（新值）。在初始分析识别出所有未归因删除和插入之后，该框架的最后一步是将这些发现关联起来。因此，未归因更新被定义为一对记录，由来自 Δ_{del} 的一条记录和来自 Δ_{ins}的一条记录组成，它们共享同一键但具有不同的值。该方法通过关联初始结果集来分离更新对，而无需重新扫描大得多的雕刻集 C。算法 3 概述了匹配这些更新对的过程。

算法 3： 检测未归因更新

输入 ： Δ_{del}：未归因删除
输入 ： Δ_{ins}：未归因插入
输出： R_{del}， R_{ins}， R_{upd}：最终互不相交集合

```
1  R_del ← Δ_del;
2  R_ins ← Δ_ins;
3  R_upd ← ∅;
   // Build a map of active records for fast lookup
4  插入_映射 ← new Map<Key, List<Record»;
5  遍历 记录 r ∈ R_ins 部署操作员
6  |  将 r 添加到 ins_map[r.key];
7  结束
   // Find and consolidate update pairs
8  对每条记录 r_del 于 Δ_del 部署操作员
9  |  如果 ins_map 包含 r_del.key则
10 |  |  对每条记录 r_act 于 ins_map[r_del.key]部署操作员
11 |  |  |  如果 r_del.value ≠ r_act.value 则
12 |  |  |  |  R_upd ← R_upd ∪ {(r_del, r_act)};
13 |  |  |  |  R_del ← R_del \ {r_del};
14 |  |  |  |  R_ins ← R_ins \ {r_act};
15 |  |  |  |  break;
16 |  |  |  end
17 |  |  end
18 |  end
19 end
20 return R_del, R_ins, R_upd;
```

它处理未归因删除和插入的集合，以生成三个互斥集合： R_{del} （最终未解释删除）、 R_{ins} （最终未解释插入）和 R_{upd} （更新对）。集合 R_{del} 和 R_{ins} 分别被初始化为 Δ_{del} 和 Δ_{ins}的副本，而R_{upd} 被初始化为空（第 1–3 行）。为了高效查找，会创建并填充一个辅助映射 ins_map ，以便按键对活动插入记录进行分组。这使得能够针对每条已删除记录快速查找潜在的更新候选项。

主循环遍历 Δ_{del} 中的每条记录 r_{del} （第 8 行）。对于每个候选删除，算法会检查 ins_map 是否包含键相同的任何活动记录（第 9 行）。如果有，则检查每条对应的活动记录 r_{act} （第 10 行），并比较它们的值（第 11 行）。如果值不同，这两条记录就构成一个候选更新对：该更新对（r_{del}， r_{act}）会被添加到 R_{upd} （第 12 行），同时分别从 R_{del} 和 R_{ins} 中移除 r_{del} 和 r_{act} （第 13–14 行），以维持互斥性质。内层循环会立即终止（第 15 行），从而确保每个键只对应一对记录。

识别出所有可能的更新对后，算法会返回经过细化且彼此互斥的未解释删除、插入和更新集合（第 20 行）。此方法确保每个未解释的更新都恰好归因于一条已删除记录和一条键相同但值不同的活动记录，从而能够清晰地对未解释的修改进行取证分离。

示例（键值存储）。同一图示说明了识别未归因更新的过程。在图4中，键值对（K006, Memphis）[已删除]和（K006, Raston）[活动] 具有相同的键，但值不同。初始检测将二者分别标记为未归因发现项。如算法 3 所述，框架随后将这两项发现合并为一个未归因更新对，揭示了审计日志无法解释的状态转换。

示例（文档存储）。同一图示展示了合并未归因更新的过程。在图5中，键值对（K006, {"Name":"Mike", "City":"Tulsa"}）[已删除] 和（K006, {"Name":"Michael", "City":"Tulsa"}）[活动]具有相同的键，但值不同。如算法 3 所述，这种组合会被标记为未归因更新，反映了一种未记录或未经授权的更改，只有通过取证式雕刻才能检测到。

5.3. 方法*2*：原位修改数据库的比较分析

对于会执行原位修改的存储引擎，记录的历史版本会被覆盖并销毁，因此方法1的单快照分析无法奏效。为检测此类情况下的未归属操作， RADAR采用 comparative analysis 方法。该方法需要由 ANOC 提取两个取证快照：可信的基线（“之前”）状态和可疑的（“之后”）状态。通过在页面级比较这些快照， RADAR可以识别物理变化，然后将其与审计日志关联，以找出差异。比较分析过程在算法4中进行了正式定义，首先比较两个快照中每个页面的 MD5 哈希。哈希不匹配时，会对该页面进行详细的记录级分析，以确定检测到的变化（例如更新、插入或删除）是否有对应的合法日志条目。如果没有此类条目，该操作就会被标记为未归属操作。即使数据库会销毁历史证据，此方法仍能让 RADAR 检测未记录活动。

该算法的方法详述如下。首先，将整个审计日志解析为键索引映射 L_{map}，以便高效查找（第 1 行）。此外，还会初始化空集合，用于存储发现的任何未归因更新、插入操作或删除操作。

分析的核心是遍历基线快照 S_{before} （第 3 行）中的每一页 p_{before}。对于每一页，都会从

算法4： 就地修改分析

输入 ： S_{before}：“之前”状态的雕刻快照
输入 ： S_{after}：“之后”状态的雕刻快照
输入： L：审计日志
输出： U_{updates}，U_{inserts}，U_{deletes}：未归属操作的集合

1 $L_{\text{map}} \leftarrow$ `ParseAuditLog(`L`)`;
2 $U_{\text{updates}}, U_{\text{inserts}}, U_{\text{deletes}} \leftarrow \emptyset, \emptyset, \emptyset$;
3 **遍历**页 p_{before} 中的 `GetPages(`S_{before}`)`**执行**
4 $p_{\text{after}} \leftarrow$ `GetPageByIndex(`S_{after}，p_{before}`.index)`;
`// Analyze page only if its content has changed`
5 **如果** `GetPageHash(`p_{before}`)` $\neq$ `GetPageHash(`p_{after}`)` **则**
6 $R_{\text{before}} \leftarrow$ `GetRecordsOnPage(`p_{before}`)`;
7 $R_{\text{after}} \leftarrow$ `GetRecordsOnPage(`p_{after}`)`;
8 $K \leftarrow R_{\text{before}}$`.keys` $\cup\, R_{\text{after}}$`.keys`;
9 **对每个** 键 $k \in K$ **执行**
10 $r_{\text{b}}, r_{\text{a}} \leftarrow R_{\text{before}}[k], R_{\text{after}}[k]$;
11 **如果** r_{b} 存在且 r_{a} 存在且 $r_{\text{b}}.\textit{value} \neq r_{\text{a}}.\textit{value}$ **则**
12 **如果** `IsOperationLogged(`L_{map}, k,不存在 ‘更新’) **则**
13 $U_{\text{updates}} \leftarrow U_{\text{updates}} \cup \{(k, r_{\text{b}}, r_{\text{a}})\}$;
14 **end**
15 **否则如果** r_{b} 存在且 r_{a} 不存在 **则**
16 **如果**不是 `IsOperationLogged(`L_{map}, k, ‘删除’) **则**
17 $U_{\text{deletes}} \leftarrow U_{\text{deletes}} \cup \{(k, r_{\text{b}})\}$;
18 **end**
19 **否则，如果** r_{a} 存在且 r_{b} 不存在 **则**
20 **如果**不是 `IsOperationLogged(`L_{map}, k, ‘插入’) **则**
21 $U_{\text{inserts}} \leftarrow U_{\text{inserts}} \cup \{(k, r_{\text{a}})\}$;
22 **end**
23 **end**
24 **end**
25 **结束**
26 **返回** $U_{\text{updates}}, U_{\text{inserts}}, U_{\text{deletes}}$;

“之后”快照 S_{after} 中按索引获取对应的页面。第 5 行执行了一项关键优化：算法会将 p_{before} 的 MD5 哈希与 p_{after}的 MD5 哈希进行比较。只有在哈希不同时，才会启动细粒度记录级分析，这表明该页面的内容已被修改。页面级哈希大幅缩小了调查范围，仅需检查存在明显更改的页面。

检测到哈希不匹配后，算法会进行详细的取证检查。它从页面的“之前”和“之后”版本中提取所有记录，并将其存储在集合 R_{before} 和 R_{after} 中。为确保不遗漏任何操作，它会创建所有

出现在任一集合中的记录键的并集（第 8 行）。随后，它遍历每个唯一键 k （第 9 行），以评估其状态转换。对于任何给定的键，可能存在三种变化：

- **更新：** 如果键为 k 的记录同时存在于两个快照中，但其值发生了变化（第 11 行），算法会查询日志映射，以确认是否记录了合法的“更新”操作。如果没有，则将此变化标记为未归因更新，并记录其键以及更新前后的值。

- **删除：** 如果键为 k 的记录存在于基线快照中，但不存在于“之后”快照中（第 15 行），则表示发生了删除。算法会检查日志中是否有对应的“删除”条目。如果没有，则将该操作添加到未归因删除集合中。

- **插入：** 如果键为 k 的记录存在于“之后”快照中，但不存在于基线快照中（第 19 行），则将其识别为插入。算法会检查日志中是否记录了针对该键的“插入”操作。如果没有相应的日志条目，则将该操作归类为无归属插入。

最后，在分析完所有已修改页面后，算法会返回包含所有已识别未归因更新、无归属插入和无归属删除的三个集合，从而全面记录未记录活动。

5.4. 审计—取证协调与归因（判定规则）

在最后阶段，即协调阶段， RADAR 会系统地将 ANOC 提取的底层证据与高层应用程序审计日志进行交叉比对。此过程应用一组正式的判定规则，将每项物理状态变化归因于一项有日志记录的操作，并将任何无法协调的制品标记为未归因操作，以供取证审查。两项关键原则构成了 RADAR 匹配逻辑的基础。首先，为避免结构化数据（如 JSON）中空白字符或键排序的细微差异造成假阳性，框架会在比较前将所有值规范化为规范字符串格式。其次， RADAR 的分析逻辑会为每条记录生成仅基于内容的“指纹”，并明确排除页面标识符或磁盘偏移量等存储层元数据字段。这种设计可防止框架将压实等常规数据库操作误判为逻辑数据修改。

采用这些匹配语义后，框架会应用其归因逻辑。如果某条记录在后续状态中缺失，但其规范值与已记录的“删除”操作或相应“更新”操作的“旧_值”不匹配，框架就会将其标记为Unattributed Delete 。反之，如果出现一条新记录，但其值不

对应已记录的“插入”操作或“更新”操作的“新_值”，框架就会将其标记为 `Unattributed Update` 。框架通过以下方式识别`Unattributed Update`：在单个快照分析中，将针对同一键的删除和插入操作关联起来；或者检测两个快照之间某条记录的内容指纹发生了变化，且没有记录“更新”操作。框架依赖使用键和规范值而非时间戳的基于状态的协调，因此能够天然抵御时钟偏差和批量操作等操作差异。这一设计对于框架在日志不完整或被抑制这一主要威胁模型下运行至关重要，因为它将提取存储视为真实基准的最终来源。

6. 实验

我们的实验在键值型和面向文档的 NoSQL 数据库上评估该框架，以捕获广泛的底层存储架构。所选引擎共同体现了截然不同的设计范式，包括原位修改、仅追加日志记录和写时复制版本控制。这种多样性确保该框架的恢复与关联能力能够在多种操作模型和数据布局下得到验证。我们在 Windows 与 Linux 系统上开展了全部实验，以验证可移植性。本节所报告性能指标的实验环境为一台运行 Ubuntu 22.04 LTS 的工作站，配备 AMD Ryzen 9 7950X3D 处理器（16 核、32 线程 @ 4.2 GHz）以及 64 GB RAM。

数据集。 我们使用星型模式基准测试（SSBM）[38]生成实验数据集。SSBM 数据的关系结构经过调整，以适配实验中使用的两种 NoSQL 模型：文档型和键值型。对于文档数据库，我们将 SSBM 表中的每一行转换为单个 JSON 文档，原始列名成为该文档的字段。对于键值数据库，我们通过选取两个具有代表性的列分别作为键和值，将每一行转换为单个键值对，详见第 3 表。

SSBM 表	**键列**	**值列**
Customer	name	国家
供应商	name	city
Part	部件键	name
Date	日期键	date
Lineorder	订单键	运输方式

表 3：SSBM 在 LMDB 和 ZODB 中的表映射。

为了在所有模型中为每条记录创建唯一且可识别的键，我们标准化了键生成过程。对于具有整数主键的表，例如 `Part`，

DB	快照 (MB)	ANOC (m)	制品 (MB)	日志文件 (MB)	RADAR (m)	总计 (m)
LMDB	1 个文件（100）	0.11	212	187	2.25	2.36
ZODB	1 个文件（121）	0.24	235	190	2.42	2.66
BerkeleyDB	2 个文件（62、64）	0.12 + 0.14	105 + 108	184	0.71	1
MDBX	1 个文件（70）	0.15	126	180	1.29	1.44
etcd	1 个文件（135）	0.41	239	186	2.48	2.89
Durus	1 个文件（120）	0.21	231	189	2.36	2.57
LiteDB	2 个文件（603，607）	30.71 + 31.33	1751 + 1752	712	2.21	64.25
RavenDB	1 个文件（512）	32.16	1755	711	41.33	73.49
Realm	1 个文件 (159)	33.41	1757	713	42.17	75.58
NitriteDB	1 个文件 (724)	34.93	1758	712	42.89	77.82

表 4：实验1摘要（大小单位为 MB，时间单位为分钟）

Date和 Lineorder，我们将表名作为前缀加到零填充整数键之前（例如，"Part#00000001"）。对于 Customer和 Supplier这类表，现有的名称字段即作为键。这种一致的键值方案使得无论底层数据库模型如何，都能统一引用记录。

采集。本节中的所有快照均由 SA 根据第 3 节中的主机级策略（图2、表 1）捕获。被分类为正常的事件用作基线，仅记录日志；维护操作会生成信息性快照；而可疑事件（例如非服务写入者、审计抑制、二进制完整性不匹配或漂移异常）会触发即时取证快照。因此，默认情况下，我们分析在首次可疑事件发生时自动捕获的单个之后快照。仅在需要进行比较分析的实验中纳入基线快照（在经批准的维护时段内作为信息性快照捕获），具体包括原位修改和删除情形。

6.1. 实验*1*：执行时间测量

本实验旨在测量 RADAR 框架在十 种采用不同存储架构（例如写时复制、仅追加和原位修改）的 NoSQL 数据库上的总执行时间。我们首先在每个数据库中填充一百万条记录，然后执行包含数千次已记录和未记录的插入、更新及删除操作的混合工作负载，以生成残留制品和相应的应用程序审计日志。总处理时间按与 RADAR 工作流程相对应的两个阶段（图3）之和计算：（i）对原始数据库快照进行 ANOC 数据雕刻；（ii） RADAR 对雕刻提取的痕迹和审计日志进行关联，以识别未归属操作。

表 4 汇总了快照大小（MB）、ANOC 数据雕刻时间（分钟）、雕刻提取的痕迹大小（MB）、日志大小（MB）、 RADAR 关联时间（分钟）以及总计时间（分钟）。对于原地写入引擎（LiteDB 和 BerkeleyDB），我们采集了“之前”和“之后”快照。当快照由多个文件组成时，ANOC 会逐个处理文件，并将数据雕刻时间和痕迹大小相加，作为总计。快照较小的大多数引擎（LMDB、ZODB、BerkeleyDB、MDBX、etcd 和 Durus）均可在 3 分钟内完成。相比之下，最大的快照导致运行时间最长。LiteDB 的总时间主要耗在两个快照的数据雕刻上（总计 64.25 分钟）；而 RavenDB、Realm 和 NitriteDB 的总时间主要耗在对 $\sim$1750 MB 的雕刻提取痕迹和 $\sim$711 MB 的日志进行关联上（总计 73.49–77.82 分钟）。

尽管需要两个快照，LiteDB 和 BerkeleyDB 的 RADAR 关联时间仍然更短，因为 RADAR首先执行快速的 MD5 页面哈希比较，仅对哈希不匹配的页面执行记录级分析。对于使用单个快照进行分析的引擎（例如写时复制或仅追加存储）， RADAR 必须将提取出的（通常是历史的）全部制品与审计日志进行协调，这一过程本质上成本更高。

6.2. 实验 *2*：检测仅追加架构中的未归属操作

本实验旨在评估 RADAR在两种不同存储场景下检测仅追加架构（*ZODB* 和 *NitriteDB*）中未记录的删除和修改的有效性。在**场景 A**中，数据库正常运行，将新的记录修订版本追加到文件末尾（不进行打包或压实），因此先前的记录状态会作为残留制品保留在较早的文件偏移量处。在**场景 B**中，对手试图通过调用维护操作（ZODB `pack`或 NitriteDB MVStore 压缩）来掩盖篡改；该操作会重写数据库并清除这些历史字节。表 5 概述了这些场景及其产生的检测结果。如表所示， RADAR 会调整检测逻辑：在历史记录得以保留时（场景 A），通过提取已删除制品进行检测；在历史记录被清除时（场景 B），则转而将无法解释的活动记录识别为未归因插入。

场景	操作	日志记录	ANOC 结果	RADAR 结果
A. 删除 （历史记录保留）	删除 1（合法）	已启用	活动 + 已删除（历史）	匹配日志
	删除 2（恶意）	已禁用	活动 + 已删除（历史）	10 次未归因删除
B. 维护 （历史记录已擦除）	更新（恶意）	已禁用	仅活动 （历史记录已清除）	无归属插入 （检测为插入的更新）
	插入（恶意）	已禁用	仅活动 （历史记录已清除）	无归属插入

表 5：在 ZODB 和 NitriteDB 中检测未归属操作的结果摘要。

过程。 我们使用 SSBM Customer 表，并在每个数据库中填充了 30,000 条初始记录。在 ZODB 中，每个条目以 Customer 名称作为键，以国家作为值。在 NitriteDB 中，每条 Customer 记录都存储为一个文档。我们将两个场景作为独立运行执行，每次运行均从全新的基线数据库状态开始，并在运行结束时捕获单个取证快照。对于每个快照，我们将数据库文件提供给 ANOC ，以便直接从存储中重建逻辑记录。随后， RADAR 将雕刻提取的痕迹与相应的应用程序审计日志进行核对。

- **场景 A（历史记录保留：未记录的删除）**。

  – **删除 1（合法）：** 已删除键 Customer#000000100–Customer#000000120（21 条记录），且已启用日志记录。– **删除 2（恶意）：** 已删除键 Customer#000000121–Customer#000000130（10 条记录），且已禁用日志记录。– **快照：** 在恶意删除后立即捕获快照。未执行打包或压实。

- **场景 B（历史记录已擦除：维护后的篡改）**。

  – **更新（恶意）：** 在已禁用日志记录的情况下，将 Customer#000000200 从 NATION="GERMANY" 修改为 NATION="MALICIOUS_UPDATE"。– **插入（恶意）：** 在已禁用日志记录的情况下，插入 Customer#999999 和 NATION="ROGUE_AGENT"。– **维护：** 在 NitriteDB 中执行 ZODB pack 和 MVStore 压缩，以重写活动数据并清理历史数据块。– **快照：** 在维护完成后立即捕获快照。

结果：场景 *A*（基于历史记录的检测）。在没有维护操作的情况下，两个数据库都保留了历史字节，为删除操作提供了物理证据。对于 ZODB，每笔事务都会按顺序追加新的对象修订版本，并将先前的修订版本完整保留在较低的文件偏移量处。对于 NitriteDB，MVStore 会保留较早的版本和数据块。因此， ANOC恢复了目标键对应的活动记录和已删除状态残留。当记录在存储中仍以先前修订版本的形式物理存在，但未出现在最新的已提交数据库状态中时，它会将记录状态标记为 已删除 。RADAR 使用算法 1 将这些从磁盘雕刻恢复的已删除 制品与审计日志进行比对。它将全部 21 次合法删除（删除 1）与日志中记录的 delete(key,value)事件相匹配，并将 10 次恶意删除（删除 2）标记为**无归属删除** ，因为没有

这些已删除键值对对应的日志条目。这些结果证实，当历史字节仍然存在时， RADAR能够恢复准确的已删除值，并通过审计日志偏差识别未经授权的移除。

结果：场景 *B*（历史记录擦除后的检测）。ZODB pack 和 MVStore 压缩的执行移除了物理历史记录。因此， ANOC仅恢复了与最终状态对应的活动记录，没有留下任何已删除状态制品来构成更新对。尽管历史记录丢失， RADAR 仍通过比较提取出的活动状态与审计日志，检测到了未记录的更改。使用算法 2， RADAR将Customer#000000200和 Customer#999999都标记为**无归属插入**，因为日志中既没有对应的 update(new_value)，也没有 insert说明。由于打包/压实消除了先前版本，算法 3 无法为 Customer#000000200 构成已删除–活动记录对。

6.3. 实验 *3*：检测写时复制分页架构中的未归属操作

本实验旨在评估 RADAR在两种由写时复制（CoW）机制决定的不同存储场景下，检测 B$^+$-tree 架构（ $LMDB$ 和 $MDBX$ ）中未记录删除和修改的能力。在**场景 A** 中，数据库通过将受影响的叶子页重写到新位置来处理更新和删除，同时将旧版本原封不动地保留在物理存储中，作为“陈旧”页面，从而能够直接恢复已删除制品。在**场景 B**中，我们模拟对手试图通过产生大量写入活动来隐藏篡改，这会导致页面复用，使陈旧页被回收并覆盖，从而移除更新前的残留内容。表 6 总结了 RADAR如何在这些瞬态存储状态下检测篡改。如表所示，当旧版本仍然存在时（场景 A 和场景 B 中的快照 T_1 ）， RADAR 会通过恢复陈旧页中的制品来标记未记录的删除和更新；而当页面复用覆盖这些残留内容后（快照T_2），由于无法再配对更新，系统会改为将相同的篡改报告为**无归属插入** 。

场景	操作	日志记录	ANOC 结果	RADAR 结果
A. 删除 （陈旧页完好）	删除 1（合法）	已启用	活动 + 已删除 (陈旧页)	已授权 (匹配日志)
	删除 2（恶意）	已禁用	活动 + 已删除 （陈旧页）	10 次未归因删除
B. 更新与复用 （复用前/后）	更新 1（合法）	已启用	活动 + 已删除 （陈旧页）	已授权 （匹配日志）
	更新 2（恶意）（T_1）	已禁用	活动 + 已删除 （陈旧页）	10 个未归属更新
	插入（恶意）（T_2）	已禁用	仅活动 (过时且已覆盖)	无归属插入 (被报告为插入的更新)

表 6：在 LMDB 和 MDBX 中检测未归属操作的结果摘要。

流程。 我们使用 SSBM**供应商** 表，在两个数据库中各填充了 20,000 条初始记录。随后，我们执行了两个场景：

- **场景 A（陈旧页完好：未记录删除）**。

  – 删除 1（合法）： 已删除键 Supplier#000000010至 Supplier#000000030，且启用了日志记录。**– 删除 2：** 已删除键 Supplier#000000110 至 Supplier#000000119，且禁用了日志记录。**– 快照：** 在删除 2 后立即捕获。未施加重写入负载，从而确保陈旧页未被回收。

- **场景 B（复用前/后：未记录操作）**。

  – 更新 1（合法）： 更新键 Supplier#000000200 至 Supplier#000000220，且启用了日志记录。**– 更新 2（恶意）：** 更新键 Supplier#000000221 至 Supplier #000000230，且禁用了日志记录。**– 快照 T_1 （复用前）：** 在执行更新 1 和更新 2之前开启一个长期读事务，并保持事务开启以固定更新前页面。这样可以防止这些页面被回收并复用，确保在 T_1仍能观察到两个版本。两个更新提交后捕获快照。**– 插入（强制复用）：** 关闭读事务，并插入 5,000 条新记录（ Supplier#000020001 至 Supplier#000025000），以触发空闲列表复用并覆盖陈旧页。**– 快照 T_2 （复用后）：** 在这批插入完成后捕获快照。

结果：场景 *A*（通过陈旧页进行检测）。 在没有大量后续写入的情况下，LMDB 和 MDBX 均保留了包含已删除记录的叶子页。尽管这些页面已无法从活动 B^+ 树的 Root 到达， ANOC 仍识别出活动集合中未被遍历的已格式化叶子页，并将其记录标记为已删除。 RADAR 利用算法 1 将这些制品与审计日志进行比对。它将合法删除（删除 1）与已记录事件相匹配，并将恶意删除（删除 2）标记为**未归因删除**。这证实了写时复制架构会在陈旧页中保留删除证据，直至存储压力迫使进行回收。

结果：场景 *B*（对页面复用的韧性）。 在快照 T_1 处，数据库呈现出双状态足迹：活动页面存储了新值，而陈旧页保留了更新前的值。 RADAR使用算法 3 将这些版本配对，并将未记录日志的修改标记为 **未归因更新**。在快照 T_2 处，插入工作负载引发了空闲列表复用，并覆盖了保存 `Supplier#000000221` 到`Supplier#000000230`更新前残留的陈旧页。 ANOC 仅恢复了活动状态，而 RADAR 通过算法 2 将同样的十个键标记为 **未归因插入** ，因为日志中不存在相应的 `insert` 或`update(new_value)` 解释。在T_2处没有已删除侧残留时，更新配对不再可能，因此同样的未记录日志的修改会通过剩余的活动记录呈现出来。

6.4. 实验 *4*：原位修改检测

本实验旨在评估 RADAR检测原位存储引擎（$LiteDB$ 和$Berkeley\ DB$）中细微的未记录日志修改的能力。与仅追加或写时复制设计不同，原地写入引擎会覆盖现有页中的字节，因此修改前的内容不会保留在单个快照中。因此， RADAR 会比较修改前快照（T_1）和修改后快照（T_2），利用页面级哈希定位已更改的页，并进行记录级差异比较，以协调正常的页面活动与审计日志。表 7 汇总了实验结果。如表所示， RADAR 能够定位已修改的页、协调已记录的删除操作，并将未记录日志的原位编辑标记为**未归因更新**。

场景	**操作**	**日志记录**	**ANOC 结果**	**RADAR 结果**
原位修改	快照（基线）（T_1）	已启用	–	–
	更新（恶意）：Part#000000500	已禁用	–	–
	删除（合法）：Part#000000501–502（同一页面）	已启用	–	–
	比较快照：T_1 与 T_2	–	所有活动记录来自 T_1 & T_2	1 条未归因更新

表 7： 检测 LiteDB 和 Berkeley DB 中未归因操作的结果摘要。

过程。 我们使用了包含 20,000 条记录的 **Parts** 数据集。在 LiteDB 中，每条记录都存储为一个文档；在 Berkeley DB 中，每个条目都使用 `partkey` 作为键，使用 `p_name` 作为值。执行了以下操作序列：

- **快照 T_1 （基线）：** 在执行任何操作之前，已捕获基线快照（T_1）。

- **更新（恶意）：** 就地编辑了 part#000000500，将 P_Name从 "azurespring"更改为 "SM PKG"，但没有相应的审计日志。
- **删除（合法）：** 在同一数据页上执行了两次常规且已记录的删除操作（part#000000501和 part#000000502），以创建无害的页面活动。
- **快照 T_2：** 捕获了一个 after 快照， T_2 并运行了 ANOC，以计算每页哈希并重建两个快照中的记录。

ANOC 为每个快照中的每个数据页计算了 MD5 哈希。 RADAR 随后将 before 和 after 快照都作为输入，比较各自的页面哈希映射，以识别内容发生变化的页面，并枚举这些变更页面上的记录。根据这些差异， RADAR 推导出记录级差异，分别表示仅存在于基线快照（Δ^-）中的内容，以及仅出现在修改后的快照（Δ^+）中的内容，随后将其与审计日志进行核对。

$$\Delta^- = \bigcup_{p \in \text{ChangedPages}} \{(k,\ v_{\text{before}}) \mid \text{位于页 } p \text{ 上的记录仅存在于 before}\},$$
$$\Delta^+ = \bigcup_{p \in \text{ChangedPages}} \{(k,\ v_{\text{after}}) \mid \text{位于页 } p \text{ 上的记录仅存在于 after}\}。$$

RADAR 将这些差异与审计日志进行核对：首先应用算法 2，找出未归因插入（Δ^+ 没有匹配的日志证据）；然后应用算法 3，对 Δ^- 和 Δ^+ 中的键进行配对，并根据值的差异识别出未归因更新。同一页面上的合法删除和插入均与相应日志条目匹配，因此予以排除。

LiteDB 结果.. 页面级哈希将搜索范围缩小到单个已修改的数据页（例如，**第 25 页**）。在该页中， RADAR 将 part#000000501 和part#000000502 的已记录删除与审计日志进行匹配，并将它们排除在考察范围之外。随后，它将 (part#000000500， "azure spring") $\in \Delta^-$ 与 (part#000000500，
"SM PKG") $\in \Delta^+$ ，且未找到对应的 update(new_value)事件。因此，算法 3 将 part#000000500 归类为**无归因更新**，揭示了恶意的原位编辑。快照之间的索引维护页也发生了变化，但未产生记录级差异，因此被忽略。

Berkeley DB 结果.. 相同的流程隔离出一个经过修改的数据页，其中包含 part #000000500。 RADAR 协调了该页上两项已记录删除，然后比较了 part#000000500的 before/after 值。与 LiteDB 一样，它观察到转变"azure spring" $\rightarrow$ "SM PKG" ，但没有找到匹配的审计更新，因此将该键标记为

未归因更新（通过算法 3）。由于 Berkeley DB 会执行原地覆盖，单个快照中没有可用的历史字节；检测依赖于两个快照之间的差异以及页面范围限定。

这些结果表明，无论是面向文档的引擎（LiteDB）还是键值引擎（Berkeley DB），原地写入引擎都不会在单个快照中保留更改前的字节。通过对之前/之后 的快照中的页面进行哈希并比较记录，ANOC 和 RADAR能够精确定位发生变化的键和字段，并且只将合法的页面活动归入审计日志。其余缺少对应日志事件的逐键值转换，则会被标记为未归因更新。

7. 未来工作

虽然 RADAR展现了检测未记录操作的强大能力，但未来仍有多个值得研究和开发的方向。

实时检测。 目前， RADAR 依赖在可疑事件发生后获取的取证快照。未来可以探索持续监控方法，在数据库文件发生变化时进行增量分析，从而实现实时威胁检测并缩短脆弱窗口。

加密数据库支持。 许多生产环境中的数据库都采用静态数据加密，这给直接进行存储雕刻带来了挑战。研究如何将 RADAR 扩展至加密环境（例如与密钥管理系统集成，或分析加密页面结构），将有助于扩大其适用范围。

分布式与集群系统。 现代 NoSQL 部署通常跨越多个节点，并采用复杂的复制和分片策略。将 RADAR 扩展至关联分布式系统中的制品，并检测集群成员之间的协同篡改尝试，是一个重要的研究方向。

机器学习增强。 当前基于规则的协调机制可以结合机器学习模型加以增强，这些模型经过训练，能够识别雕刻提取的痕迹中的异常模式，而这些模式可能表明存在复杂的篡改尝试。这有助于检测可规避传统基于特征的检测的新型攻击模式。

性能优化。 尽管当前实现已展现出实用的性能，但仍可通过并行处理、增量雕取以及缓存已分析的页面来进一步优化。对于超大型数据库而言，这些改进尤为重要。

与 *SIEM* 平台集成。 开发标准化接口，以便将 RADAR与现有的安全信息与事件管理 (SIEM) 系统集成，从而促进其在生产环境中的部署，并实现与其他安全信号的关联。

云原生数据库扩展。随着组织日益采用云原生 NoSQL 服务，调整 RADAR 以适配云存储抽象和无服务器数据库架构，既带来了技术挑战，也提供了机遇。

未来在这些领域的工作将扩展 RADAR的能力，使其不再局限于事件后分析，而是迈向持续的防篡改安全监控。通过将底层存储观测与更高层次的异常检测和系统集成相结合，RADAR 能够发展成为一个稳健的平台，在本地部署和云环境中可靠地检测未经授权的数据篡改。现代数据库基础设施日益复杂，这凸显了加强完整性验证和运营韧性的必要性，以应对内部人员和系统级威胁。

8. 结论

在这项工作中，我们介绍了 RADAR，这是一个独立于日志的取证框架，旨在应对检测 NoSQL 数据库中未归属操作这一关键挑战。通过将底层存储雕刻与高层审计日志验证相结合， RADAR 提供了一种识别日志规避活动的方法，即使特权攻击者操纵或抑制日志记录机制，也能进行检测。

我们的框架为数据库取证领域作出了多项重要贡献。首先，我们形式化定义了恶意 DevOps 工程师的威胁模型：此类工程师可以篡改部署配置和审计日志，但无法抹除操作系统级痕迹。其次，我们针对不同的存储架构开发了两种互补的检测方法：针对仅追加和写时复制系统（这类系统会保留历史痕迹）的单快照分析，以及针对会覆盖先前状态的原位修改系统的比较分析。第三，我们提出了形式化算法，通过系统地协调雕刻提取的痕迹与审计日志，识别无法归因的插入、删除和更新操作。

通过在多种代表性 NoSQL 数据库（包括键值存储和文档存储）上进行广泛实验，我们证明了 RADAR在检测各种篡改场景方面的有效性。结果表明，即使在打包或压实等会从物理层面移除历史字节的维护活动之后，该框架仍能成功识别未记录操作。根据数据库架构的不同，该框架的处理速率为 31.7 至 397 MB/min，展现出适用于取证调查的实际性能。

重要的是， RADAR将提取存储视为真实基准，而非仅依赖可能已遭篡改的日志，这一设计理念代表了数据库安全监控范式的转变。该框架立足于应用层语义与存储层工件的交汇处，提供了纵深防御，能够抵御传统基于审计的方法无法检测到的内部人员威胁。

致谢

本研究部分由路易斯安那州董事会资助，资助项目编号为 LEQSF(2022-25)-RD-A-30。

参考文献

[1] M. I. Nissan、J. Wagner、A. Rasin，《ANOC：自动化 NoSQL 数据库雕刻工具》，《国际法证科学：数字调查》53（2025年）301929。[2] 甲骨文公司，《MySQL 8.4 参考手册：信息_模式表》，访问日期：2025-01-29（2025年）。网址 https://dev.mysql.com/doc/refman/8.4/en/information-schema.html[3] J. M. Hellerstein、M. Stonebraker、J. Hamilton 等，《数据库系统架构》，《数据库基础与趋势》1（2）（2007年）141–259。[4] H. Choi、S. Lee、D. Jeong，《SQL Server 数据库取证恢复：实用方法》，《IEEE Access》9（2021年）14564–14575。[5] P. Frühwirt、P. Kieseberg、S. Schrittwieser、M. Huber、E. Weippl，《InnoDB 数据库取证：从重做日志重建数据操作查询》，载于：2012年第七届可用性、可靠性与安全国际会议（ARES），IEEE，2012年，第625–633页。[6] J. Wagner、A. Rasin、J. Grier，《通过内部结构雕刻进行数据库取证分析》，《数字取证》14（2015年）S106–S115。[7] J. Wagner 等，《使用 DBCarver 进行数据库取证分析》，载于：创新数据系统研究会议，2017年。[8] J. Wagner、A. Rasin、K. Heart、R. Jacob、J. Grier，《Db3f 与 df-toolkit：数据库取证文件格式与数据库取证工具包》，《数字取证》29（2019年）S42–S50。

[9] J. Wagner、A. Rasin、J. Grier，《数据库镜像内容探索器：雕刻官方记录中不存在的数据》，《数字取证》18（2016年）S97–S107。[10] S. Garfinkel，《使用快速对象验证雕刻连续与碎片化文件》，《数字取证》4（2007年）2–12。[11] R. Poisel、S. Tjoa，《文件雕刻综合文献综述》，载于：2013年可用性、可靠性与安全国际会议（ARES），IEEE，2013年，第475–484页。[12] B. Lenard、J. Wagner、A. Rasin、J. Grier，《Sysgen：系统状态语料生成器》，载于：第15届可用性、可靠性与安全国际会议（ARES）论文集，ACM，2020年，第1–6页。[13] I. MongoDB，《WiredTiger B树架构》，访问日期：2025-01-29（2023年）。网址 https://source.wiredtiger.com/11.0.0/arch-btree.html[14] Z. Foundation，《ZODB》，访问日期：2025-01-29（2024年）。网址 https://zodb.org/en/latest[15] C. J. F. Candel、D. S. Ruiz、J. J. García-Molina，《NoSQL 与关系数据库的统一元模型》，《信息系统》104（2022年）101898。[16] J. Yoon、D. Jeong、C.-h. Kang、S. Lee，《面向文档存储 NoSQL 数据库管理系统的取证调查框架：以 MongoDB 为案例研究》，《数字取证》17（2016年）53–65。[17] J. Yoon、S. Lee，《恢复 MongoDB 中已删除数据的方法与工具》，《数字取证》24（2018年）106–120。[18] H. Sung、M. Jin、M. Shin、H. Roh、W. Choi、S. Park，《LESS：利用快照进行日志记录》，载于：2019年IEEE大数据与智能计算国际会议（BigComp），IEEE，2019年，第1–4页。[19] M. Xu、X. Xu、J. Xu、Y. Ren、H. Zhang、N. Zheng，《基于 RDB 和 AOF 文件的 Redis 数据库取证分析方法》，《计算机杂志》9（11）（2014年）2538–2544。[20] K. E. Pavlou、R. T. Snodgrass，《数据库篡改取证分析》，《ACM 数据库系统汇刊》（TODS）33（4）（2008年）30。

[21] R. T. Snodgrass, S. S. Yao, C. Collberg, 《审计日志篡改检测》，载于：《第三十届超大型数据库国际会议论文集》第 30 卷，VLDB 基金会，2004 年，第 504–515 页。[22] B. Schneier, J. Kelsey, 《支持计算机取证的安全审计日志》，《ACM 信息与系统安全汇刊》2 (2) (1999) 159–176。[23] S. A. Crosby, D. S. Wallach, 《用于防篡改日志记录的高效数据结构》，载于：USENIX 安全研讨会，2009 年，第 317–334 页。[24] K. Kent, M. Souppaya, 《计算机安全日志管理指南》，技术报告 NIST SP 800-92，美国国家标准与技术研究院，马里兰州盖瑟斯堡（2007 年）。[25] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, D. Tian, 《Custos：使用可信执行对操作系统进行实用的防篡改审计》，载于：网络与分布式系统安全研讨会，2020 年。[26] A. Ahmad, S. Lee, M. Peinado, 《Hardlog：使用新型审计设备进行实用的防篡改系统审计》，载于：2022 年 IEEE 安全与隐私研讨会（SP），IEEE 计算机学会，2022 年。[27] T. F. J.-M. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, J. Bacon, 《实用的全系统数据溯源捕获》，载于：ACM 云计算研讨会（SoCC），2017 年。[28] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, V. N. Venkatakrishnan, 《HOLMES：通过关联可疑信息流进行实时高级持续性威胁检测》，载于：IEEE 安全与隐私研讨会（S&P），2019 年。[29] J. Wagner, A. Rasin, K. Heart, T. Malik, J. Furst, J. Grier, 《通过页面雕刻检测数据库文件篡改》，载于：第 21 届数据库技术拓展国际会议，2018 年。[30] M. I. Nissan, J. Wagner, S. Aktar, 《数据库内存取证：一种逆向工程查询活动的机器学习方法》，《国际法证科学：数字调查》44 (2023) 301503。[31] M. I. Nissan, 《使用支持向量机分析数据库内存中的取证痕迹》，硕士论文，新奥尔良大学，论文编号 3036（2022 年）。URL https://scholarworks.uno.edu/td/3036

[32] J. Wagner, M. I. Nissan, A. Rasin, 《数据库内存取证：识别用于日志验证的缓存模式》, 《国际法证科学：数字调查》45 (2023) 301567。[33] R. M. A. P. Rajatheva, S. M. A. H. Senavirathna, L. A. N. Madushani, P. M. T. P. San-darenu, 《大数据环境下 NoSQL 系统漏洞比较研究》, 《国际信息技术管理杂志》（IJMIT）11 (4) (2019)。[34] L. A. Tawalbeh, M. Tawalbeh, R. G. Al-Tawil, 《NoSQL 数据库中的安全问题》, 载于：2013 年第 8 届互联网技术与安全交易国际会议（ICITSS），IEEE，2013 年，第 443–447 页。[35] M. Kleppmann, 《设计数据密集型应用：可靠、可扩展和可维护系统背后的核心理念》，O'Reilly Media，2017 年。[36] jsonlines.org, 《JSON Lines》, https://jsonlines.org，访问日期：2025 年 6 月 27 日（无日期）。[37]美国卫生与公众服务部, 《联邦法规汇编》第 164.312 节：技术保障措施，检索自 https://www.hhs.gov/hipaa/for-professionals/security/laws-regulations/index.html （2013 年）。[38] P. O.Neil 等, 《星型模式基准测试与增强事实表索引》, 载于：《性能评估与基准测试》, 2009 年。