Rethinking Tamper-Evident Logging: A High-Performance, Co-Designed Auditing System

Rui Zhao
University of Virginia
Charlottesville, USA
dkw7xn@virginia.edu

Muhammad Shoaib
University of Virginia
Charlottesville, USA
ewe4gy@virginia.edu

Viet Tung Hoang
Florida State University
Tallahassee, USA
tvhoang@cs.fsu.edu

Wajih Ul Hassan
University of Virginia
Charlottesville, USA
hassan@virginia.edu

ABSTRACT

Existing tamper-evident logging systems suffer from high overhead and severe data loss in high-load settings, yet only provide coarse-grained tamper detection. Moreover, installing such systems requires recompiling kernel code. To address these challenges, we present Nitro, a high-performance, tamper-evident audit logging system that supports fine-grained detection of log tampering. Even better, our system avoids kernel recompilation by using the eBPF technology. To formally justify the security of Nitro, we provide a new definitional framework for logging systems, and give a practical cryptographic construction meeting this new goal. Unlike prior work that focus only on the cryptographic processing, we co-design the cryptographic part with the pre- and post-processing of the logs to exploit all system-level optimizations. Our evaluations demonstrate Nitro's superior performance, achieving 10×-25× improvements in high-stress conditions and 2×-10× in real-world scenarios while maintaining near-zero data loss. We also provide an advanced variant, Nitro-R that introduces in-kernel log reduction techniques to reduce runtime overhead even further.

CCS CONCEPTS

• **Security and privacy** → **Systems security**.

KEYWORDS

Logging; System Auditing; Message Authentication Codes;

1 INTRODUCTION

Audit logs are essential for tracking system activity and supporting security operations [52, 55, 62]. They capture detailed event sequences that help reconstruct attacks, identify stealthy behavior, and support root cause analysis, compliance auditing, and incident response [36, 42, 49, 55, 57]. Their reliability makes them vital for both real-time detection and forensic investigation.

Insecure Logging Systems

Current audit logging systems [3, 4, 10, 36, 48, 65, 67] predominantly trace and intercept system calls to capture log data. For example, Linux Auditd [10], the native Linux auditing tool, monitors system calls to record security-relevant events, but suffers from performance degradation under high workloads. To address these issues, Sekar *et al.* [67] introduced eAudit, which uses eBPF technology [24] within the Linux kernel to significantly improve audit logging performance. However, eAudit [67] and similar systems remain vulnerable to *race condition attacks* as demonstrated by Paccagnella *et al.* [63]. These systems operate asynchronously, queuing system calls for later disk logging using a FIFO mechanism. This creates a tamper window, defined as the delay between when a system call occurs and when it is logged, which attackers can exploit to manipulate the log queue. Although eAudit reduces this window, it does not *completely* eliminate it.

Tamper-Proof Logging Systems

To eliminate the tamper window, tamper-proof logging systems [33, 41, 51] employ specialized hardware to make log modification infeasible. NoDrop [51] uses Intel Memory Protection Keys (MPK) [27] to isolate memory regions containing logs, while HardLog and OmniLog rely on external hardware devices and synchronous logging that pushes events to non-rewritable storage as they occur. These methods ensure strong integrity guarantees by preventing any post-generation log manipulation. However, this comes at a steep cost. Hardware-based isolation introduces substantial runtime overhead, especially under high throughput, as each log must be synchronously flushed before the system can proceed. Moreover, these systems require specialized hardware, making them impractical to deploy in general-purpose or cloud environments.

Tamper-Evident Logging Systems

As an alternative to costly hardware-based approaches, tamper-evident logging systems [45, 63] offer software-based integrity

guarantees and tackle race conditions attacks. These systems generate cryptographic tags for log entries, allowing auditors to detect tampering after the fact. For example, KennyLoggings [63] generates an integrity tag per log using forward-secure MACs, while QuickLog2 [45] improves verification efficiency by aggregating all per-log tags into a single value.

Challenges in Tamper-Evident Logging Systems

Despite their advancements, tamper-evident logging systems face several challenges, which are described below.

C1: Coarse-Grained Tamper Detection. Systems like QuickLog2 [45] generate a single aggregate tag for an entire batch of logs, meaning that any tampering triggers a global verification failure without revealing which log was altered. While KennyLoggings [63] improves detection granularity by tagging each log individually, it increases the storage overhead by 5% and the running time by 40%. Moreover, its security definition doesn't provide any theoretical guarantees for the extraction of unmodified logs.

C2: Insecure Tag Storage. Prior systems fail to ensure secure tag storage. In KennyLoggings [63], tags are stored in plaintext alongside logs, enabling truncation attacks where an adversary can delete recent logs and restore a previously stored tag to hide tampering. While QuickLog2 [45] avoids storing tags altogether, this reduces auditability.

C3: Large Secret. The cryptographic methods employed by tamper-evident systems [45, 63] rely on secret keys and state information to ensure the integrity of the logs. However, keeping the size of those secrets small is crucial, as a larger secret state presents a greater risk of side-channel attacks. For instance, KennyLoggings [63] precomputes about 3.2MB of secret keys, storing them in kernel space before system startup. While this approach accelerates the signing process, it is more vulnerable to side-channel attacks [45].

C4: Difficult Maintenance. Both KennyLoggings [63] and Quick-Log2 [45] extend Auditd [10] and require modifying and recompiling kernel modules to insert cryptographic hooks. These modifications introduce fragility into the kernel audit pipeline, risking crashes and making the systems error-prone to maintain across kernel updates.

C5: Excessive Overhead. Existing loggers often introduce significant overhead, which can become intolerable under high system loads. In our experiments, we measured runtime overhead as the ratio of CPU time used by the logging systems to that of the benchmark itself. QuickLog2 [45] incurred a runtime overhead of 15.9%–895.1%. QuickLog2 outperforms KennyLoggings in their experiments, suggesting that KennyLoggings may impose even greater overhead.

C6: Severe Data Loss. Under high load, tamper-evident logging systems often drop large volumes of logs, compromising forensic analysis. This differs from race condition attacks, as it stems from overload rather than tampering. While acknowledged in non-secure loggers like eAudit [67], this issue is largely ignored in tamper-evident systems. Our experiments show that existing systems such as KennyLoggings [45, 63] experience data loss rates of up to 98%.

Our Approach: Nitro

To address the above-mentioned challenges, we present Nitro, a high-performance, tamper-evident audit logging system. To the best of our knowledge, Nitro is the first tamper-evident logging system to operate fully in eBPF. Figure 1 illustrates the overall architecture, and the main contributions are summarized below.

1. Provable security. To address C1 and C2, we extend the definitional frameworks in prior work [45, 63] to allow tags to be encrypted before being sent to storage. Our notion provides a unified syntax for KennyLoggings [63] (where all tags are stored but not encrypted), QuickLog2 [45] (where no tag is stored), and Nitro (where tags are *occasionally* encrypted and stored). We formally show that our notion ensures that the extracted logs are unmodified.

Instead of building a logging system directly from a MAC like-QuickLog2 [45], we define a new primitive that we call a *MAC combiner*. This provides an abstraction on the tag aggregation process, and also a useful tool that may find applications beyond the context of logging systems. The idea of MAC combiner is inspired from the notion of aggregate MAC [54], but the latter unfortunately does not work for the setting of logging systems.

2. eBPF-Compliant Implementation. To address C4, Nitro implements cryptographic logging entirely within the constraints of the eBPF runtime. These constraints include bounded stack sizes, static control flow, the absence of AES-NI [23] support, and the prohibition of dynamic loops. To navigate these limitations, we use Chaskey, an ISO-standard lightweight MAC, as our underlying MAC, selected for its compatibility with eBPF's verifier. Our contribution is not the use of Chaskey itself, but the careful pre-processing of the logs before running Chaskey on them. The rationale for this design choice is further discussed in Section 4.

3. Parallel MAC Signing. A major cause of C5 is the bottleneck of centralized cryptographic state. To tackle this issue, Nitro assigns a separate signing context (tag, key, state) to each logical CPU core using eBPF's Per-CPU Arrays. Each core signs its own logs independently, enabling decentralized MAC computation without synchronization overhead. While the parallel signing approach brings a big improvement on speed, it worsens C3, because the state size is amplified by a factor of N, where N is the number of logical cores. In our machine, $N = 36$, but we still manage to keep the state size at 1.2KB, which is much smaller than KennyLoggings [63].

4. Two-level Cache with Time-Aware Flow Control. To overcome C6, Nitro introduces a novel caching strategy that reduces log loss during high load. It uses a two-level architecture with Per-CPU Arrays and ring buffer, managed by time-aware controllers that schedules log transfers based on system parameters such as core count, buffer saturation, and event rate. This adaptive mechanism minimizes I/O pressure and maintains reliable logging.[1]

5. Co-Designed Logging and Cryptographic Stack. To resolve C5 and C6, Nitro avoids retrofitting cryptographic layers onto existing loggers like Auditd [10] and instead builds a unified logging and MAC framework from scratch. This integration enables optimizations that reduce both computational and memory overhead. For example, Nitro introduces eBPF-aware padding that avoids runtime padding of all log messages by statically defining

---

[1]eAudit [67] uses a two-level caching mechanism to reduce data loss, but its simple weight-based scheduling strategy leads to frequent data copying, resulting in higher I/O overhead as shown in our experiments.
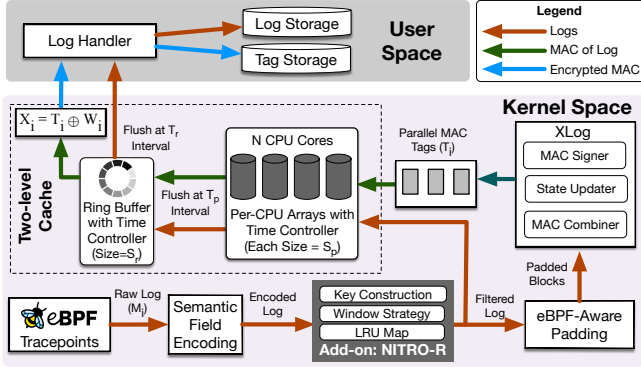
Figure 1: High-level workflow of NITRO.

log structures. It also applies semantic field encoding, such as mapping syscall names to 32-bit integers, to shrink MAC input size. These design choices lower CPU cycles and simplify deployment by removing the need for kernel patches or hardware dependencies.

Even Better: NITRO-R

Building on NITRO, NITRO-R is the first in-kernel, eBPF-based implementation of an audit log reduction technique inspired by Auditrim [68], significantly lowering I/O overhead by filtering redundant logs before they reach user space. Unlike prior user space approaches [47, 49, 56, 68, 69], NITRO-R operates entirely within the eBPF environment, avoiding added latency and complexity. Porting Auditrim to eBPF introduces key challenges: adapting time windows to system load, operating within eBPF's restrictive execution model, and efficiently managing in-kernel data structures for log reduction. NITRO-R addresses these with an eBPF-based LRU hash map and verifier-safe logic for real-time, low-overhead log reduction. Applied before NITRO's XLog, NITRO-R preserves log integrity while improving throughput and storage efficiency. Our goal is not to propose a new log reduction method but to demonstrate the feasibility of efficiently implementing existing techniques within the kernel using eBPF for improved performance.

Evaluation Results

We evaluated NITRO against state-of-the-art logging systems using stress-test benchmarks (from eAudit [67]) and real-world benchmarks (aligned with HardLog [33]). NITRO demonstrated a substantial performance advantage, with improvements of 10×–25× on stress-test benchmarks and 2×–10× on real-world benchmarks, while preserving tamper-evident properties. In terms of data loss, NITRO achieved near-zero loss, compared to other logging systems, which experienced 31%–98% data loss under identical conditions. Additionally, NITRO-R reduced runtime overhead by an average of 34% beyond NITRO's performance. To understand how each optimization in NITRO contributes to its performance, we also conduct a factorial design analysis. To assess integration feasibility, we built eAudit-SEC by embedding XLog into eAudit. eAudit-SEC suffered from poor performance and significant data loss, confirming the need for architectural rethinking.

Availability Our source code and benchmarks are available at https://github.com/DART-Laboratory/Nitro.
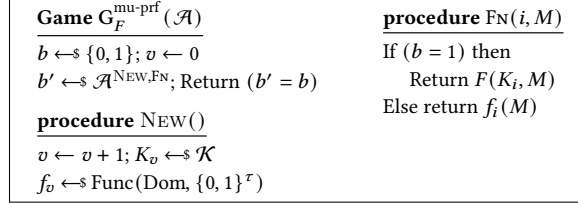
Figure 2: Game defining the (multi-user) PRF security of a function $F : \mathcal{K} \times \text{Dom} \rightarrow \{0,1\}^\tau$.

2 PRELIMINARIES

NOTATIONS. By $\text{Func}(\text{Dom}, \text{Rng})$ we denote the set of all functions $f : \text{Dom} \rightarrow \text{Rng}$. We use $\perp$ as a special symbol to denote rejection, and it is assumed to be outside $\{0,1\}^*$. We write $X\|Y$ to denote the concatenation of two bit strings X and Y.

If $\mathcal{X}$ is a finite set, we let $x \leftarrow_\$ \mathcal{X}$ denote picking an element of $\mathcal{X}$ uniformly at random and assigning it to x. If A is an algorithm, we let $y \leftarrow A(x_1, \ldots; r)$ denote running A on inputs $x_1, \ldots$ and coins r, and assigning the output to y. By $y \leftarrow_\$ A(x_1, \ldots)$ we denote picking r at random and letting $y \leftarrow A(x_1, \ldots; r)$. We write $\mathcal{A}^f$ to indicate that adversary $\mathcal{A}$ has oracle access to a function f.

GAMES. We use the game-playing framework of Bellare and Rogaway [37]. (See Figure 2 for an example.) We write $\mathbf{G}(\mathcal{A}) \Rightarrow b$ to denote the event of running game $\mathbf{G}$ with an adversary $\mathcal{A}$ that results in b. We also write $\mathbf{G}(\mathcal{A})$ to abbreviate $\mathbf{G}(\mathcal{A}) \Rightarrow true$.

MAC. A Message Authentication Code (MAC) scheme is a function $F : \mathcal{K} \times \text{Dom} \rightarrow \{0,1\}^\tau$. It takes as input a key $K \in \mathcal{K}$ and a message $M \in \text{Dom}$, and then deterministically produces a tag $T \leftarrow F(K, M)$.

A standard security goal for a MAC scheme is to be a pseudorandom function (PRF). Informally, this means that an efficient adversary can't tell the tags of its chosen messages from truly random strings. Traditionally, PRF security is defined for the single-user setting, but in our application, it is more convenient to use the multi-security PRF notion. Specifically, we define the advantage of an adversary $\mathcal{A}$ in breaking the (multi-user) PRF security of F as

$$\mathbf{Adv}_F^{\text{mu-prf}}(\mathcal{A}) = 2 \cdot \Pr[\mathbf{G}_F^{\text{mu-prf}}(\mathcal{A})] - 1 \ ,$$

where game $\mathbf{G}_F^{\text{mu-prf}}(\mathcal{A})$ is defined in Figure 2. If the adversary only attacks user 1 then the game is in the single-user setting. It is well-known that one can generically reduce the multi-user PRF security to the single-user one via a hybrid argument.

3 TAMPER-EVIDENT LOGGING SYSTEMS

In this section, we provide a foundational treatment for tamper-evident logging systems. In particular, we give a more general syntax and a stronger security definition (that are needed by our system) than prior work. Moreover, instead of building a logging system by aggregating tags from a MAC like [45], we abstract this via a new tool that we call *MAC combiner*, making the construction conceptually simpler. Finally, we show how to build a practical logging system XLog meeting our notion.

3.1 Defining Security For Logging Systems

THREAT MODEL AND ASSUMPTIONS. Our threat model is similar to that of prior work [45, 63]. In particular, we consider an adversary

that mounts an attack to escalate privilege; the system calls of this attack are recorded in the logs. Once the adversary gains root access, it can modify the logs in both memory and storage to hide the traces of the attack. Between the time that the logs of the attack are generated and the moment that the adversary has root privilege, there is a short window. This time gap is too short for the logs to be sent *asynchronously* to the user space for cryptographic processing, as demonstrated empirically via the race attack in [63]. Still, we assume that it is long enough for a quick, *synchronous* cryptographic processing.

We assume that logs are periodically sent to a different, trusted machine for forensic analysis. For the auditor to detect the tampering of the logs, the logging system will include some short tags as a proof of integrity. KennyLoggings [63], for example, uses a tag for every single log. QuickLog2 [45] instead uses a single *aggregate* tag for the entire collection of logs to save storage cost and reduce running time. Instead of going to either extreme, our system Nitro still maintains an aggregate tag in the memory, but occasionally saves the current tag to the log storage. By doing so, we provide finer-grained information about the tampering to the auditor than QuickLog2, with only a small increase of storage cost and running time. Still, if we store the tags in the clear like KennyLoggings then this is vulnerable to a *truncation attack*: an adversary can replace the current tag by an old tag from the storage and delete recent logs. As a result, the tags in the storage needs to be encrypted, and thus the syntax of our system deviates from prior work.

For the auditor to verify the integrity tags, we assume that initially the auditor and the logging system share a short secret state; the logging system will then update the state immediately each time it signs a log message. We assume that if a variable is overwritten, one cannot recover the old value. We also assume that before gaining root access, the adversary can't retrieve information of secrets in kernel memory.

We say that the logs are *unaltered* if the attacker only modifies the logs created *after* it gains root access, meaning the logs containing the information of the attack still remains intact. Our goal is to ensure that if logs are altered then the auditor should be able to detect it.

<u>Syntax.</u> We now give a general syntax that allows a logging system to store encrypted tags. A *logging protocol* Π consists of a pair of deterministic algorithms (Update, Sign) and is associated with a state space $\mathcal{S}$, key space $\mathcal{K}$, and a tag length τ. Algorithm Update takes as input a state S^* and a boolean b. If $b = false$ then it outputs an updated state S and a key K. If $b = true$, in addition to (K, S), it outputs another string $X \in \{0, 1\}^\tau$. We require that the output (K, S) of Update is independent of the boolean b. Algorithm Sign takes as input a key K, a message M, a prior tag $T \in \{0, 1\}^\tau$, and produces an updated tag $T^* \in \{0, 1\}^\tau$.

- Initially, the auditor samples a root state $S \leftarrow\$ \mathcal{S}$ and a key $K \leftarrow\$ \mathcal{K}$. He then stores (K, S, T) in the server's memory, with $T \leftarrow 0^\tau$.
- Once the i-th log message M_i is available, the logging system first updates the tag $T \leftarrow \text{Sign}(K, M_i, T)$. If it doesn't want to store an encryption of the tag, it will update the key and state via $(K, S) \leftarrow \text{Update}(S, false)$. Otherwise, it will run $(K, S, X) \leftarrow \text{Update}(S, true)$ and encrypts the tag via $X \leftarrow T \oplus X$. The choice of encrypting the tag or not does not affect the output (K, S)

$\boxed{\begin{array}{l} \textbf{Game } \mathbf{G}^{\text{fa}}_{\Pi}(\mathcal{A}) \\ \hline (M_1, \ldots, M_q, \sigma) \leftarrow\$ \mathcal{A}; S_0 \leftarrow\$ \mathcal{S}; K_0 \leftarrow\$ \mathcal{K} \\ (X_1, \ldots, X_q, T) \leftarrow \text{Tag}(M_1, \ldots, M_q) \\ (M'_1, \ldots, M'_r, T') \leftarrow\$ \mathcal{A}(S, K, X_1, \ldots, X_q, T, \sigma) \\ \text{//Attacker must alter messages to win} \\ \text{If } (r \geq q) \wedge ((M_1, \ldots, M_q) = (M'_1, \ldots, M'_q)) \text{ then return } \textit{false} \\ (X^*_1, \ldots, X^*_r, T^*) \leftarrow \text{Tag}(M'_1, \ldots, M'_r) \\ \text{Return } (T^* = T') \\ \\ \text{//Generate final aggregate tag and encrypt intermediate tags} \\ \textbf{procedure } \text{Tag}(M_1, \ldots, M_v) \\ \hline T_0 \leftarrow 0^\tau; S \leftarrow S_0; K \leftarrow K_0 \\ \text{For } i \leftarrow 1 \text{ to } v \text{ do} \\ \quad T \leftarrow \text{Sign}(K, M_i, T) \\ \quad (K, S, X_i) \leftarrow \text{Update}(S, true); X_i \leftarrow T \oplus X_i \\ \text{Return } (X_1, \ldots, X_v, T) \end{array}}$

Figure 3: Game defining the FA **security of a logging protocol** Π.

and the subsequent tags. Note that the key and state are updated immediately after signing a new message. This ensures that in a race attack [63], by the time the adversary gains root access, the keys signing the logs of its attack are already overwritten with subsequent keys.

The auditor is later given messages $(M'_1, \ldots, M'_r)$ and an aggregate tag T'. Using procedure Tag in Figure 3, the auditor can derive the final tag T^* for $(M'_1, \ldots, M'_r)$.[2] If the logs are altered then $T^* \neq T'$.

The syntax above captures all existing logging systems. For example, in KennyLoggings [63], the string X that Sign generates is simply 0^τ, meaning that tags are stored in the clear instead of being encrypted. Moreover, KennyLoggings always calls $\text{Sign}(\cdot, true)$, meaning that all tags are stored. In QuickLog2 [45], one always calls $\text{Sign}(\cdot, false)$, meaning that there is no tag storage.

<u>Defining security.</u> For an adversary $\mathcal{A}$ attacking a logging protocol Π, we define its advantage in breaking the *forward authenticity* (FA) of Π as

$$\mathbf{Adv}^{\text{fa}}_{\Pi}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{fa}}_{\Pi}(\mathcal{A})] \ ,$$

where game $\mathbf{G}^{\text{fa}}_{\Pi}(\mathcal{A})$ is defined in Figure 3. Initially the adversary generates log messages $(M_1, \ldots, M_q)$ and an internal state σ.[3] The game then samples a state $S_0 \leftarrow\$ \mathcal{S}$ and a key $K_0 \leftarrow\$ \mathcal{K}$, and derive the tag of $(M_1, \ldots, M_i)$ for ever $i \leq q$, and encrypts it. The adversary is then given back its internal state σ, the final tag T, the encrypted tag X_i of $(M_1, \ldots, M_i)$, and the current key and state. Its job is to produce a forgery $(M'_1, \ldots, M'_r, T')$.

Recall that the auditor will later derive the aggregate tag T^* of messages $(M'_1, \ldots, M'_r)$. The adversary wins the game if (1) $r < q$ or $(M'_1, \ldots, M'_q) \neq (M_1, \ldots, M_q)$, meaning the logs are altered, and (2) $T' = T^*$, meaning that the auditor fails to detect that the logs are altered.

In the definition above, the adversary is given all encrypted tags $(X_1, \ldots, X_q)$. In our system, the adversary only has access to some of those, because we only encrypt and store tags occasionally. In

---

[2]The Tag procedure always calls $\text{Sign}(\cdot, true)$, whereas the logging system may call $\text{Sign}(\cdot, b)$ for any $b \in \{true, false\}$. Still, since Sign and Update are deterministic, and since the key and state that Sign outputs are independent of b, if the logs are not modified then the output of Tag is consistent with what the logging system produces.
[3]In practice, those log messages are honestly generated by the logging system to record the syscalls. It can only help the adversary by letting it choose those log messages.

QuickLog2, the adversary is given no encrypted tag at all. Giving this additional information can only help the adversary.

Discussion. It is instructive to see why KennyLoggings [63] fails the definition above. Recall that KennyLoggings, all tags are stored in the clear, meaning that X_i is the aggregate tag of $(M_1, \ldots, M_i)$. Given $(S, K, X_1, \ldots, X_q, T)$, the adversary can win the game by picking an arbitrary number $r < q$, and then outputting $(M'_1, \ldots, M'_r) = (M_1, \ldots, M_r)$ and $T' = X_r$. In this case the messages are altered, and yet T' will match the tag T^* derived by running procedure Tag in Figure 3 on $(M'_1, \ldots, M'_r)$. As a result, the adversary wins with advantage 1.

Extracting unmodified logs. All prior work stops when the auditor realizes that the logs have been altered. But in practice, one wants to extract unmodified logs for forensic analysis. We now show how to do this.

Suppose the auditor is given a list $L = \{(X'_j, j)\}$ of index j and encrypted tag X'_j for messages $(M'_1, \ldots, M'_j)$. Recall that the auditor can compute a list of $(X^*_1, \ldots, X^*_r)$ of encrypted tags by running Tag on $(M'_1, \ldots, M'_r)$. Let s be the largest index that $X'_s = X^*_s$; if there is no such index then we let $s = 0$. Suppose that the original logs are $(M_1, \ldots, M_q)$. If the logs have been altered then we claim that the forward authenticity notion ensures that $M'_i = M_i$, for every $i \leq \min\{s, q\}$. To see why, fix an index d.

- First consider the case $d \leq q$. Recall that the adversary $\mathcal{A}$ is given the state S_q, the key K_q, the tag T_q, and the encrypted tags $(X_1, \ldots, X_q)$. If we instead give $(S_d, K_d, T_d, X_1, \ldots, X_d)$ to $\mathcal{A}$, this can only help because $\mathcal{A}$ can derive $(K_q, S_q, T_q, X_{d+1}, \ldots, X_q)$. Forward authenticity then ensures that the adversary cannot produce the correct aggregate tag T' of another $(M'_1, \ldots, M'_d) \neq (M_1, \ldots, M_d)$. As a result, it also cannot produce the encrypted tag $X' = T' \oplus (T_d \oplus X_d)$ of T'.
- Suppose that $d > q$. Since the logs are altered, we must have $(M'_1, \ldots, M'_q) \neq (M_1, \ldots, M_q)$. Since the adversary knows (S_q, K_q), it can compute the mask W for the d-th tag. Forward authenticity ensures that given the aggregate tag T of $(M_1, \ldots, M_q)$ and the list of encrypted tags $(X_1, \ldots, X_q)$, the adversary cannot produce $(M'_1, \ldots, M'_d)$ with a correct tag T'. As a result, it also cannot produce the encrypted tag $X' = T' \oplus W$ of T'.

Still, aiming for log extraction would amplify the advantage by a factor q.

Discussion. Given that the adversary can modify all logs in the storage that haven't been sent to the auditor, one may question the significance of log extraction. We argue that log extraction is meaningful if one couples a tamper-evident logging system with some non-rewritable hardware like HardLog [33]. That is, each log will be immediately signed but there will be some bounded delay before it is written to the non-rewritable storage. In that case, most logs created before the adversary gets root privilege would remain intact, and thus it is critical to extract them.

3.2 MAC Combiner

In this section, we formalize a new primitive that we call *MAC combiner*. It allows one to sign q messages (each with a fresh key) and then combine the q tags into a single, short tag. This tool gives an abstraction of the XOR construction that QuickLog2 [45] uses.

$$
\begin{array}{l}
\hline
\textbf{Game } G^{\text{forge}}_{\text{MC}}(\mathcal{A}) \\
\hline
(M_1, \ldots, M_q, \sigma) \leftarrow\!\!\$ \; \mathcal{A}; \; T_0 \leftarrow 0^\tau \\
\text{For } i \leftarrow 1 \text{ to } q \text{ do } K_i \leftarrow\!\!\$ \; \mathcal{K}; \; T_i \leftarrow T_{i-1} \boxplus G(K_i, M_i) \\
(M'_1, \ldots, M'_r, T) \leftarrow \mathcal{A}(\sigma, T_q); \; T'_0 \leftarrow 0^\tau \\
\text{For } i = 1 \text{ to } r \text{ do } T'_i \leftarrow T'_{i-1} \boxplus G(K_i, M'_i) \\
\text{Return } ((M'_1, \ldots, M'_r) \neq (M_1, \ldots, M_q)) \land (T'_r = T) \\
\hline
\end{array}
$$

Figure 4: Game defining unforgeability security of a MAC combiner $\text{MC} = (G, \boxplus)$.

Definition. An MAC combiner MC is a pair $(G, \boxplus)$, where $G : \mathcal{K} \times \{0, 1\}^* \rightarrow \{0, 1\}^\tau$ is a MAC, and $\boxplus : \{0, 1\}^\tau \times \{0, 1\}^\tau \rightarrow \{0, 1\}^\tau$ is an operator. To compute the aggregate tag T_q of messages $(M_1, \ldots, M_q)$ for keys $(K_1, \ldots, K_q)$, one would compute $T_i \leftarrow T_{i-1} \boxplus G(K_i, M_i)$ for every $i \leq q$, where $T_0 = 0^\tau$.

We define the advantage of an adversary $\mathcal{A}$ breaking the unforgeability security of MC as

$$
\mathbf{Adv}^{\text{forge}}_{\text{MC}}(\mathcal{A}) = \Pr[G^{\text{forge}}_{\text{MC}}(\mathcal{A})] \; ,
$$

where game $G^{\text{forge}}_{\text{MC}}(\mathcal{A})$ is defined in Figure 4. Here the (stateful) adversary first requests to sees the aggregate tag T_q of messages $(M_1, \ldots, M_q)$. The parameter q is called the *message count* of $\mathcal{A}$. The adversary then produces forgery messages $(M'_1, \ldots, M'_r)$ and a forgery tag T, with $r \leq q$. It wins if the aggregate tag T'_r of $(M'_1, \ldots, M'_r)$ under keys $(K_1, \ldots, K_r)$ is also T.

Discussion. Our notion is inspired by the definition of aggregate MAC of Katz and Lindell [54] but the latter doesn't work in the setting of logging systems. Under the notion of aggregate MAC, the adversary $\mathcal{B}$ is given *individual* tags of some messages M_i and has to predict the aggregate tag of $(M'_1, \ldots, M'_r)$. In the forward security for logging systems, the attacker $\mathcal{A}$ is instead given the aggregate tag of $(M_1, \ldots, M_q)$ and has to find the aggregate tag of $(M'_1, \ldots, M'_r)$. The reduction from forward security to aggregate MAC simply fails. Specifically, given $\mathcal{A}$ that picks $r < q$ and $M'_i = M_i$ for every $i \leq r$, the only plausible way $\mathcal{B}$ can provide $\mathcal{A}$ with the aggregate tag of $(M_1, \ldots, M_q)$ is to obtain the individual tags $T_1, \ldots, T_q$ of all $M_1, \ldots, M_q$. But then what $\mathcal{B}$ later obtains from $\mathcal{A}$ is the aggregate tag of $(M_1, \ldots, M_r)$ that is useless because $\mathcal{B}$ can generate this via $T_1 \boxplus \cdots \boxplus T_r$.

Security analysis. We say that $\boxplus$ is *commutative* if $X \boxplus Y = Y \boxplus X$. Moreover, $\boxplus$ is *invertible* if there is an associated inverse operator $\boxminus : \{0, 1\}^\tau \times \{0, 1\}^\tau \rightarrow \{0, 1\}^\tau$ such that if $Z = X \boxplus Y$ then $Z \boxminus Y = X$. Note that if $\boxplus$ is both commutative and invertible then for any fixed $X \in \{0, 1\}^\tau$, if we pick $Y \leftarrow\!\!\$ \; \{0, 1\}^\tau$ then $X \boxplus Y$ is uniformly distributed over $\{0, 1\}^\tau$.

The following result shows that if G is a good PRF and $\boxplus$ is commutative and invertible then $\text{MC} = (G, \boxplus)$ is a good MAC combiner. To have an intuition of why this is true, let's pretend that G is a truly random function (instead of merely a PRF). Then given the aggregate tag T of $(M_1, \ldots, M_q)$, the aggregate tag T' of $(M'_1, \ldots, M'_r)$ is still uniformly distributed over $\{0, 1\}^\tau$, thanks to the commutativity and invertibility of $\boxplus$. Hence the chance that the adversary can guess T' is about $2^{-\tau}$. The proof is in Appendix A.

---

procedure $\text{XLog}[\text{MC}, F].\text{Update}(S, b)$

$S' \leftarrow F_S(0); K' \leftarrow F_S(1); X \leftarrow F_S(2)$
If b then return (K', S') else return (K', S', X)

procedure $\text{XLog}[\text{MC}, F].\text{Sign}(K, M, T)$

$T' \leftarrow T \boxplus G_K(M);$ Return T'

Figure 5: The $\text{XLog}[\text{MC}, F]$ **logging system, with** $\text{MC} = (G, \boxplus)$.

PROPOSITION 1. *Let* $\text{MC} = (G, \boxplus)$ *be a MAC combiner of tag length* τ, *where* $\boxplus$ *is commutative and invertible. Then for any adversary* $\mathcal{A}$ *whose message count is at most* q, *we can construct an adversary* $\mathcal{B}$ *of about the same running time such that*

$$\mathbf{Adv}_{\text{MC}}^{\text{forge}}(\mathcal{A}) \leq \mathbf{Adv}_{G}^{\text{mu-prf}}(\mathcal{B}) + \frac{1}{2^\tau} \ .$$

Adversary $\mathcal{B}$ *makes at most* $2q$ *queries, with two queries per user.*

INSTANTIATION. A simple choice for $\boxplus$ is the xor operator, which is used in the aggregate MAC construction of Katz and Lindell [54]. This is the choice in the QuickLog2 logging system [45], and also our system. But this is not the only possible choice. One can use, for example, modular addition, but this is less efficient.

3.3 The XLog Construction

THE XLOG CONSTRUCTION. We now show how to build a logging system $\text{XLog}[\text{MC}, F]$ from a MAC combiner $\text{MC} = (G, \boxplus)$ where the key length and tag length are both n bits, and a PRF $F : \{0,1\}^n \times \{0,1,2\} \rightarrow \{0,1\}^n$. In practice n should be 128. The specification of XLog is given in Figure 5.

SECURITY ANALYSIS. The following result shows that if $\boxplus$ is invertible then XLog provides forward authenticity; the proof is in Appendix B.

THEOREM 2. *Let* $F : \{0,1\}^n \times \{0,1\}^* \rightarrow \{0,1\}^n$ *be a PRF, and let* $\text{MC} = (G, \boxplus)$ *be a MAC combiner of* n-*bit key and* n-*bit tag where* $\boxplus$ *is invertible. For an adversary* $\mathcal{A}$ *making* q *signing messages, we can construct adversaries* $\mathcal{B}$ *and* $\mathcal{D}$ *of about the same running time such that*

$$\mathbf{Adv}_{\text{XLog}[\text{MC},F]}^{\text{fa}}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{\text{mu-prf}}(\mathcal{B}) + \mathbf{Adv}_{\text{MC}}^{\text{forge}}(\mathcal{D}) \ .$$

Adversary $\mathcal{B}$ *makes* $3q$ *oracle queries, with exactly 3 queries per user. The message count of* $\mathcal{D}$ *is also* q.

PROOF IDEAS. Let r be the number of forgery messages that $\mathcal{A}$ outputs, and q be the number of signing messages. To reduce the forward authenticity of XLog to the unforgeability of MC, we need to model the keys as independent, uniformly random strings. Unfortunately, if $r > q$ then one can't treat the keys $K_{q+1}, K_{q+2}, \dots$ as independent, because they can be derived from the state S_q, and this state is given to the adversary. Our proof gets around this issue by exploiting the fact that $\boxplus$ is invertible, which allows us to reduce the case $r > q$ to the case $r = q$.

Assume that $r \leq q$. Since we derive the keys, states, and task masks via a PRF, one can view those as independent, uniformly random strings. As a result, the key, state, and encrypted tags that the adversary $\mathcal{A}$ receives are independent of the prior keys, states, and tags, and thus can be ignored. In this simplified view, the forward

authenticity game becomes the unforgeability game, meaning that the security of MC implies the security of XLog.

4 NITRO

Building on the theoretical foundation of XLog and the MAC combiner, we now present NITRO, a practical logging system designed for real-world deployment without kernel modification. In this section, we discuss how the cryptographic guarantees of XLog are integrated into the constrained execution environment of eBPF, addressing various performance bottlenecks through strategic architectural optimizations.

4.1 Overview of eBPF & Its Limitations

The Extended Berkeley Packet Filter (eBPF) [8, 18, 19] is a revolutionary technology that enables user-defined programs to be safely executed within the operating system kernel. Developed from the Berkeley Packet Filter (BPF) [32], eBPF significantly broadens the scope of BPF beyond networking. It now supports general-purpose system tracing, security monitoring, and performance analysis by allowing programs to attach to various kernel hooks such as tracepoints, kprobes, and network events without requiring kernel modifications or additional modules [66].

To ensure safety and stability, eBPF introduces a static verification engine that analyzes each program before loading, disallowing unsafe operations such as unbounded loops, invalid memory access, or kernel crashes. Once verified, eBPF programs are compiled using a Just-In-Time (JIT) compiler to run with near-native efficiency, while maintaining strict isolation from core kernel components. Figure 6 shows how an eBPF program is prepared and executed. It is first compiled in user space into bytecode, then passed to the kernel where it goes through a verifier for safety checks. If accepted, the bytecode is JIT-compiled into native machine code and attached to a system call. The hook is triggered when the corresponding system call is invoked. It can inspect both the call and its return value, and even block the system call if necessary. For efficient communication between kernel and user space, eBPF provides high-performance data transfer mechanisms including perf buffers and ring buffer. Ring buffer is now widely recommended due to its lower latency and better multi-core scalability [15]. Consequently, NITRO adopts ring buffer as the foundation of its logging pipeline.

Integrating existing logging systems, such as QuickLog2 [45], into eBPF initially appears straightforward, but practical obstacles render this approach infeasible. eBPF imposes several stringent limitations: it prohibits dynamic loops (L1), severely restricts memory usage (L2), rejects programs with high computational complexity (L3), and lacks support for hardware acceleration such as AES-NI [23] (L4). These constraints render standard cryptographic constructions impractical. For example, implementing AES without AES-NI leads to severe performance penalties, and memory limits prevent the inclusion of essential components like permutation tables. Exceeding these storage constraints results in memory allocation failures, making direct integration impractical.

4.2 Selecting a Suitable PRF for eBPF

Given eBPF's strict limitations, selecting an appropriate pseudorandom function (PRF) was essential. We required a PRF lightweight
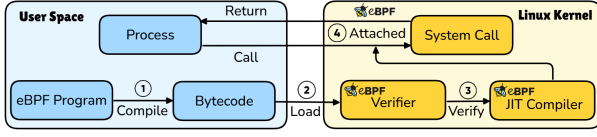
Figure 6: Workflow of eBPF program compilation, verification, and attachment to system call hooks in the Linux kernel.

enough to fit eBPF's constraints, avoiding AES-based constructions and adhering to verifier limitations. The chosen PRF should have 128-bit output because XLog needs to use it to derive 128-bit keys. These constraints ruled out candidates like SipHash [35] due to their 64-bit output limitation. After evaluating alternatives, we selected Chaskey [61], an ISO-standard PRF that provides provable security, significant performance advantages (approximately 8 faster than AES without AES-NI), and a compact, modifiable codebase suitable for broad industrial and governmental adoption. While Chaskey operates sequentially, which might suggest potential advantages in parallel designs, our compact encoding drastically reduces log message sizes (e.g., down to 24 bytes on average), diminishing any parallelization benefits in this context and emphasizing simplicity and efficiency.

4.3 Integrating XLog with eBPF: Challenges & Optimizations

We start with a naive implementation of the blueprint above, by adding XLog on top of eAudit [67], with Chaskey as the underlying PRF. We keep the architecture of eAudit, but remove its weighting mechanism. This mechanism aims to classify logs based on their priority, so that eAudit can send critical logs to the storage (that is *assumed* to be non-rewritable in [67]) as quickly as possible. In our case, the security model is different; the adversary may be able to write to the log storage after it gets root privilege. Hence the weighting mechanism is useless in our threat model and can be dropped. We refer to this extended version of eAudit as eAudit-SEC.

However, evaluation (detailed further in Section 6) revealed significant performance drawbacks with this naive approach. Specifically, integrating XLog increased runtime overhead by approximately 5.8× and data loss by about 35×. These outcomes underscored that eAudit was fundamentally unsuitable for pairing with tamper-evident logging systems.

To address these challenges, we systematically optimized the architecture by exploiting XLog 's design properties and adapting key aspects of the logging pipeline. We identified two core inefficiencies in eAudit-SEC and redesigned the system accordingly:

- *Streamlined MAC Computation with Parallel Signing:* In the naive approach, logs are retrieved entirely from memory before being processed by the MAC, introducing unnecessary latency. To mitigate this, we directly integrate eBPF calls into Chaskey's processing pipeline, building log message blocks incrementally and immediately performing MAC operations on each block. Additionally, the original approach utilizes a centralized signing key updated sequentially after each signing operation, severely limiting parallelism during system peaks. To alleviate this bottleneck, we implement a distributed signing mechanism by maintaining

individual signing keys per CPU logical core using Per-CPU Array structures, dramatically reducing contention and increasing throughput.
- *Fewer I/O requests:* In eAudit [67], the storage is assumed to be non-rewritable, so one needs to push log messages to the user space frequently to minimize the tampering window. In contrast, in our setting, the adversary can write to the storage after it gets root privilege, and thus having the logs in the storage or in memory make no difference in terms of security. Hence in our case, once a log is signed and the aggregate tag is updated, the log can continue to reside in the memory as long as space permits. We therefore can push the logs to the user space with much less frequency.

These optimizations must work within the limitations imposed by eBPF. Specifically, we adopt a set of verifier-compatible implementation techniques to handle its limitations. For L1, we combine a fixed-loop structure with a padding strategy (see Section 4.4) to safely process variable-length inputs. For L2, we minimize memory usage through compact log structures and avoid large buffers or dynamic allocation. For L3, we restructure the MAC computation using pointer-based staged processing to stay within the verifier's instruction limits. Lastly, for L4, our lightweight PRF Chaskey well-suited for software-only execution. These design and implementation choices together enable efficient and secure integration of XLog into the constrained eBPF environment.

4.4 Parallel MAC Signing

How to implement signing. Under Chaskey, a message needs to be parsed into a sequence $M_1 \cdots M_m$, where each $|M_i| = 128$. (If the message is fragmentary, padding is needed for the last block.) Initially, the tag X is initialized to 0^{128}. One then iteratively goes over each message block M_i and updates X. Instead of running Chaskey on top of eAudit, we directly make eBPF calls to derive a message and run Chaskey processing on each 128-bit block of the message on the fly. Compared to the naive approach in Section 4.3, our approach hides the memory cost of retrieving the log message from memory, and also saves some cost due to type casting. Still, implementing this in the eBPF framework is tricky since only static loops are allowed (L1).

To get around the limitation above, we view a message as a pair (P_1, P_2), where the first part P_1 consists of fixed-length arguments, such as the syscall name or PID, and the second part P_2 consists of dynamic-length arguments. Let $\text{pad}(P_1)$ be the string obtained by appending 0's to P_1 until the next multiple of 128 bits. We run a static loop to build the message blocks of $\text{pad}(P_1)$ and implement Chaskey's processing to derive an intermediate tag X. We then run another static loop; the number of iteration is a constant big enough to exceed the number of blocks in P_2. In this loop, we resume Chaskey's processing with the tag X above. This novel strategy, which we refer to as *eBPF-aware padding*, enables MAC signing in eBPF by carefully structuring the message layout to comply with verifier constraints while minimizing runtime costs. Conceptually, the process above is to run Chaskey on the string $\text{pad}(P_1)\|P_2$. Since the length of P_1 is uniquely determined by the syscall name, the string $\text{pad}(P_1)\|P_2$ is a unique encoding of (P_1, P_2).

At the first glance, one may wonder why we sign a longer message pad$(P_1)\|P_2$ instead of $P_1\|P_2$. While the latter option may save some cryptographic cost, it also adds overheads because (1) P_1 and P_2 are built from different data types, and some type casting is needed, and (2) this may cause memory misalignment if the length of P_1 is not a multiple of 4 bytes.

SEMANTIC FIELD ENCODING. We designed a compact encoding of the logs that significantly reduces the log size similar to [46, 67]. For example, in the Postmark benchmark, the average size of logs generated by the Linux daemon auditd is 850B, whereas ours is only 24B. The benefit of this log compression is even more important to our setting than that of eAudit [67], because smaller logs means less cryptographic cost.

MULTIPLE SIGNING LINES. If we have a single aggregate tag for all the logs, this creates a bottleneck during peak time because we can only use a single CPU core to sign the logs one by one. A natural solution is to maintain multiple tags to enable parallel processing. This approach however has never been considered in the past, because it is unclear how to use multiple cores to sign logs concurrently without resorting to expensive multi-threading mechanisms. To deal with this issue, we leverage the Per-CPU Array data structure to have one tuple of (tag, key, state) per CPU logical core. Each CPU core only signs logs generated by that core and updates its own tag.

Our approach parallelizes the cryptographic cost for both the server and the auditor. This performance gain however comes with a cost for security. First, the security definition in Figure 3 only considers a single user, but here we have N users, where N is the number of logical CPU cores. Single-user security still implies multi-user security via a standard hybrid argument, but concretely, the advantage is amplified by a factor of N. Moreover, instead of keeping just 32-byte secret in the memory (16B for the key and 16B for the state), we now have to keep $32N$ bytes of secret, which increases the risk of a side-channel attack. Still, in practice N is small. For example, in our machine, $N = 36$, and thus we keep about 1.2KB secret in memory. In contrast, KennyLoggings [63] has 3.2MB secret.

4.5 Two-level Cache Design

In eAudit [67], the Per-CPU Array pushes data to the ring buffer at a maximum interval of 2^{24} ns, and the ring buffer moves data to the user space after every 8 operations. That is, there are at least 8 transfers from the ring buffer to the user space per second. This frequency is unnecessary in our setting, because once a log is signed and the key is updated, the log can safely reside in the memory as long as space permits. We instead use a two-level time controller to reduce the I/O cost.

- *Per-CPU Array → ring buffer:* The Per-CPU Array would push data to the ring buffer periodically, for every T_p seconds.
- *Ring buffer → user space:* The ring buffer moves data to the user space after every T_r seconds.

HOW TO CHOOSE OPTIMAL PARAMETERS We provide a flexible parameter setting mechanism in NITRO. Here, we discuss how to choose the optimal parameters as a reference. The runtime performance and reliability of our system are mainly controlled by
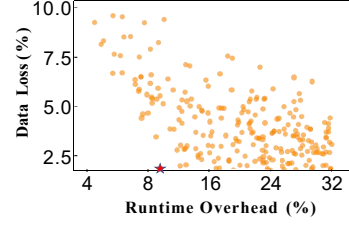
Figure 7: Trade-off between data loss and overhead under different parameter settings on seven stress-test benchmarks. The red pentagram marks the selected optimal configuration.

four key parameters: the Per-CPU Array size S_p, the ring buffer size S_r, the first-level flush interval T_p, and the second-level flush interval T_r. To understand the trade-off between runtime overhead and data loss, we simulate all parameter combinations across a large design space. The evaluation is conducted on all seven stress-test benchmarks, each running with 16 threads, and we report the average results. Specifically, we sweep S_p from 0.5KB to 32KB, S_r from 1MB to 64MB in exponential steps, T_p from 5ms to 300ms in linear steps of 50ms, and T_r from 500ms to 3000ms in steps of 500ms. This results in a total of 1,764 configurations, each representing a candidate deployment strategy.

Our results, presented in Figure 7, include only the configurations that keep data loss below 10% and overhead below 32% to avoid visual clutter and highlight practical deployment choices. We observe that overly aggressive flushing (i.e., small T_p and T_r) can significantly reduce data loss but at the cost of high I/O overhead. Conversely, extremely lazy flushing schedules (e.g., large T_r or small S_r) may keep overhead low but risk frequent buffer overflows. In particular, we identify that settings with $S_p \geq$ 8KB, $S_r \geq$ 16MB, $T_p \in$ [55ms, 255ms], and $T_r \in$ [1000ms, 2000ms] offer a favorable balance between runtime overhead and data loss.

We pick the Per-CPU Array size S_p to be 32KB, which is empirically enough to avoid queue overflow in the Per-CPU Array. To avoid queue overflow in the ring buffer, if we have N CPU logical cores and the ring buffer size is S_r, we need $S_r \geq T_r \cdot (N \cdot S_p)/T_p$. In our system, $N = 36$, and we pick $T_p = 200$ ms, $T_r = 1$ s, and $S_r = 64$ MB, which also fall within the optimal range identified in our experiments. Compared to eAudit-SEC, our system reduces the number of transfers from the ring buffer to user space by at least 8×, and reduces the frequency of transfers from the Per-CPU Array to the ring buffer by approximately 12×.

4.6 Log Integrity Checking

Once the log entries $(M'_1, \ldots, M'_r)$ and tags are transferred to user space, they are stored in their respective storages, as illustrated in Figure 1. Later, during audit, the auditor can recompute a final tag T^* over $(M'_1, \ldots, M'_r)$ using the procedure Tag shown in Figure 3. If the logs have been tampered with, the recomputed tag will not match the original. Unlike prior work [63], we consider a stronger adversary who may arbitrarily modify the logs stored on a compromised machine. This setting reflects more realistic threat scenarios but also introduces new challenges for verifying log integrity. In particular, the attacker may attempt to truncate logs

or insert forged entries without detection. To address this, XLog provides formal guarantees that such tampering will be detected during audit. Moreover, the auditor can still identify all unaltered logs before the point of modification. These properties are formally proven in Section 3 under the security model of XLog.

4.7 Putting It All Together

Creating and managing logs consists of three steps: (i) pre-processing (e.g., monitoring events to create logs and encoding them), (ii) cryptographic processing (e.g., signing logs with a MAC), and (iii) post-processing (e.g., caching the logs). Prior works either study step (ii) and use Auditd for the other steps or focus on steps (i) and (iii) without incorporating (ii) into their design. This results in a disconnection: one cannot combine existing state-of-the-art solutions such as QuickLog2 and eAudit. Our work adopts a holistic yet modular approach, addressing all three components. We first establish the theory for step (ii) in Section 3, independent of steps (i) and (iii). Then, in this section (Section 4), we co-design steps (i) and (iii) and select an instantiation for step (ii).

4.8 Security Analysis

Section 3 provides the formal proof. Here, we follow the analysis style of prior work [63]. We examine key attack scenarios and explain how Nitro offers full or partial protection.

- **A1: Log Truncation Attack.** An adversary attempts to remove or truncate part of the system logs to hide malicious activities or erase forensic evidence.
- **A2: Side-Channel Attack.** An adversary performs side-channel attacks targeting kernel memory to extract the tag generation keys, which could enable the forgery of logs and subsequent tags.
- **A3: Replay Attack.** An adversary reuses previously captured valid log messages to deceive the system or forge historical events, potentially leading to incorrect analysis or audit results.

A1 poses a threat to all tamper-evident systems that do not encrypt tags, in cases where the system outputs tags together with plaintext logs. In such systems, the adversary can truncate the log by removing the later part that contains malicious activities. The remaining earlier logs and their corresponding tags remain consistent, allowing the adversary to bypass detection. Nitro addresses this threat by introducing tag encryption. In Nitro, the output tag is encrypted. Without the correct round key and state, the adversary cannot recover the original tag value from the encrypted tag. Additionally, Nitro's key update and tag aggregation mechanisms ensure each tag reflects the latest log content, preventing mismatches. As defined in Section 3, Nitro fully prevents log truncation attacks.

A2 is partially mitigated in Nitro through key rotation and per-core logging, which significantly reduce side-channel attack surfaces. Unlike prior work [63] that stores megabytes of precomputed secrets in memory, Nitro requires only 48 bytes of secret material per core, including the key, state, and tag. This design supports parallel signing and removes synchronization overhead. The use of per-core registers in modern CPUs and frequent access to secrets during MAC computation lower the risk of cache-based leakage. Per-core isolation also ensures that compromising one core does not affect the security of others.

A3 is fully prevented in Nitro through its forward-secure design. Even if an adversary gains access to the log storage device along with plaintext logs and encrypted tags, tampering with existing logs or forging new entries is infeasible. This is because each log and its tag are derived from prior logs, per-round secret keys, and an evolving state, rather than generated independently. Without access to the initial state and keys, an adversary cannot perform a successful replay attack, even with full knowledge of the cryptographic algorithms.

5 NITRO-R

Prior work [38, 43, 47, 49, 56, 57, 68] shows that system logs often contain redundant entries that inflate storage and processing overhead without improving detection or forensic value. Auditrim [68] demonstrated that suppressing such duplicates is both feasible and effective. Inspired by these findings, we present Nitro-R, an in-kernel mechanism that eliminates duplicate log entries in real time *before* any cryptographic processing.

Nitro-R is the first log reduction system implemented entirely within the kernel using eBPF. Unlike previous systems [47, 49, 56, 59, 69] that require kernel modifications or rely on post-processing in user space, Nitro-R performs safe, real-time log reduction directly inside the kernel within the strict constraints of the eBPF verifier. This design required addressing several non-trivial technical challenges:

- First, eBPF imposes strict restrictions such as prohibiting unbounded loops, dynamic memory allocation, and complex control flow. To comply with these limitations, we designed a verifier-safe log reduction logic using only static memory access patterns, bounded loops, and minimal branching, ensuring real-time performance without verifier rejections.
- Duplicate detection requires efficient identification of repeated log entries based on their semantic content. Nitro-R addresses this by constructing a compact, structured key composed of the syscall name, process ID, and arguments, and mapping it to timestamps using an in-kernel LRU hash table. This enables precise and efficient duplicate detection under tight memory and instruction constraints.
- The effectiveness of log reduction depends on an appropriate time window for determining whether two events are duplicates. Nitro-R supports dynamic adjustment of this time window, computed as $\min(1, 2 \times (t_1 - t_0) + 0.001)$. Here, t_0 and t_1 represent the previous and current timestamps of the same event. While our evaluation uses a fixed value of $T_W = 1$ second as suggested by Auditrim [68], the system is capable of adapting to changing system conditions.
- Continuous log generation poses risks of unbounded memory growth. Nitro-R mitigates this by leveraging the built-in eviction capabilities of eBPF's LRU hash map, ensuring bounded memory usage and maintaining long-term scalability without requiring kernel modification.

Crucially, Nitro-R performs log reduction before any cryptographic signing by XLog, ensuring that only unique log entries are authenticated. This ordering guarantees the integrity and authenticity of final logs while significantly reducing both MAC computation and I/O overhead. Moreover, Nitro-R is designed to be modular.

Although we implement a time-window-based log reduction inspired by Auditrim, the system can easily be extended to support alternative log reduction techniques such as semantic filtering, priority-aware sampling, or event coalescing [49, 56, 69], as long as they can be implemented under eBPF's programming model. The detailed algorithm is provided in Algorithm 1 (Appendix), which processes each log entry in real time, identifies near-duplicates using structured keys and timestamps, and emits only unique entries to the reduced log buffer.

6 EVALUATION

In this section, we evaluate the performance and reliability of NITRO. In particular, we investigate the following research questions (RQs):

RQ1 What is the runtime overhead of NITRO compared to state-of-the-art logging systems? (Sections 6.1)

RQ2 How much does NITRO reduce data loss during high system loads compared to state-of-the-art systems? (Section 6.2)

RQ3 How much parallel signing and two-level cache improve NITRO 's performance and reduce data loss? (Section 6.3)

RQ4 How does NITRO-R perform in terms of runtime overhead and data loss? (Section 6.4)

RQ5 What is the runtime memory usage of NITRO? (Appendix F)

RQ6 How do NITRO and NoDrop compare in terms of runtime overhead?[4] (Appendix G)

RQ7 How well does NITRO detect log tampering and how quickly does it respond under real-world attacks? (Appendix H)

RQ8 Do the logs generated by NITRO and NITRO-R maintain correctness under causality analysis principles? (Appendix I)

IMPLEMENTATION We implement NITRO via the BPF Compiler Collection (BCC) version 0.29.1 [24]. We select 68 critical syscalls as our monitoring targets; see Appendix D for the list of those. To monitor system calls, we use APIs from BCC like TRACEPOINT_PROBE to listen for syscall invocations, and use args to get parameters of each syscall [25]. We implement the XLog construction in C. The signing of a message involves two static for-loops; they are manually unrolled for performance improvement. Informed by Theorem 2, we pick the tag length to be 64 bits, and represent each tag as an unsigned 64-bit integer. Each log would contain an additional 32-bit identifier for the CPU core. This identifier is just a bookkeeping and is *not* processed by the MAC. The log handler is written in Python 3. It receives data from kernel mode through bpf.ring_buffer_poll [25], then parses and stores them on disk.

EVALUATION SETUP Our experiments used VMs on an AMD EPYC 9654 server. Each VM was configured with 300 GB RAM, 200 GB SSD, and 36 logical CPU cores, running Ubuntu 22.04 (kernel 6.5.0). Only essential software was installed to minimize interference, and each experiment was repeated 30 times. In all experiments, the relative standard deviation ranged from 3.91% to 14.05%, ensuring result reliability.

BENCHMARKS SELECTION CRITERIA. To comprehensively evaluate NITRO 's runtime overhead, we used a diverse set of benchmarks to compare it against state-of-the-art logging systems [45, 51, 67]. For high-load conditions, we adopted all seven benchmarks from eAudit [67], referred to as *Stress-test Benchmarks*, including Postmark (mail server simulation) and Linux kernel compilation to represent resource-intensive workloads. To evaluate performance in practical settings, we incorporated benchmarks from HardLog [33], labeled as *Real-world Benchmarks*, which include Firefox (web browsing) tested with Speedometer [20], OpenSSL for cryptographic operations [12], and lmbench [60]. Further details on both sets of benchmarks are provided in Tables 4 and 5 in Appendix C. Although QuickLog2 [45] shares similar evaluation goals with eAudit [67], we excluded its benchmarks for two reasons: (1) its syscall-level tests (e.g., open, write) generate significantly lower workloads than our stress-test benchmarks, and (2) its application-level benchmarks overlap with those already included in our real-world evaluation. Consistent with prior work in secure audit logging [33, 41, 45, 63], we did not include DARPA and ATLAS datasets [6, 7, 34], as they are not designed to assess the performance of audit logging systems.

BASELINE SELECTION. We selected state-of-the-art systems for the evaluation of NITRO. QuickLog2, the best-performing tamper-evident system, was chosen over QuickLog and KennyLoggings due to its superior performance. For tamper-proof systems, we selected NoDrop [51], which offers strong performance and accessible hardware requirements, excluding OmniLog [41] and HardLog [33] due to their stricter hardware demands. Prior evaluations confirm that NoDrop outperforms both alternatives [51], making it an optimal baseline. For non-secure systems, we included eAudit, which provides the best balance of runtime overhead and data loss, excluding other non-secure tools [3, 4, 10, 16, 64] as eAudit already outperforms them [67]. To assess the feasibility of integrating cryptographic protections into eAudit, we implemented eAudit-SEC, which incorporates NITRO 's cryptographic method (XLog) within eAudit 's architecture.[5] We also excluded existing log reduction systems [38, 39, 47, 49, 69, 70], as our goal is not to propose a new log reduction scheme but to demonstrate that existing techniques can be adapted for in-kernel processing via eBPF.

EVALUATION METRICS. We define runtime overhead $O_{\text{runtime}} = (T_{\text{total}} - T_{\text{benchmark}})/T_{\text{benchmark}}$, where $T_{\text{benchmark}}$ is the CPU time used by the benchmark without logging and T_{total} is the time with logging enabled. This metric follows the methodology in eAudit and quantifies the additional overhead introduced by the logging system. We define data loss as $P_{\text{loss}} = D_{\text{discarded}}/D_{\text{total}}$, where $D_{\text{discarded}}$ is the amount of log data dropped during operation and D_{total} is the total generated log data. This measures the system's ability to capture logs reliably.

6.1 RQ1: Runtime Overhead

STRESS-TEST BENCHMARKS. Figure 8 shows that NITRO consistently outperforms QuickLog2 across all seven stress-test benchmarks. During repeated high-load tests, particularly with PostMark [53], systems running QuickLog2 frequently crashed. This instability

---

[4]Since NoDrop [51] requires Intel-specific hardware, we conducted its experiments separately on a machine equipped with the necessary Intel chips. To avoid skewing the comparisons in RQ1 and RQ2, we isolate this evaluation in RQ6, where we directly compare NITRO and NoDrop on various benchmarks.

[5]We did not attempt to combine eAudit with QuickLog2, as QuickLog2 depends on AES-NI hardware acceleration, which is incompatible with eBPF due to its restrictive verifier and lack of hardware instruction support.
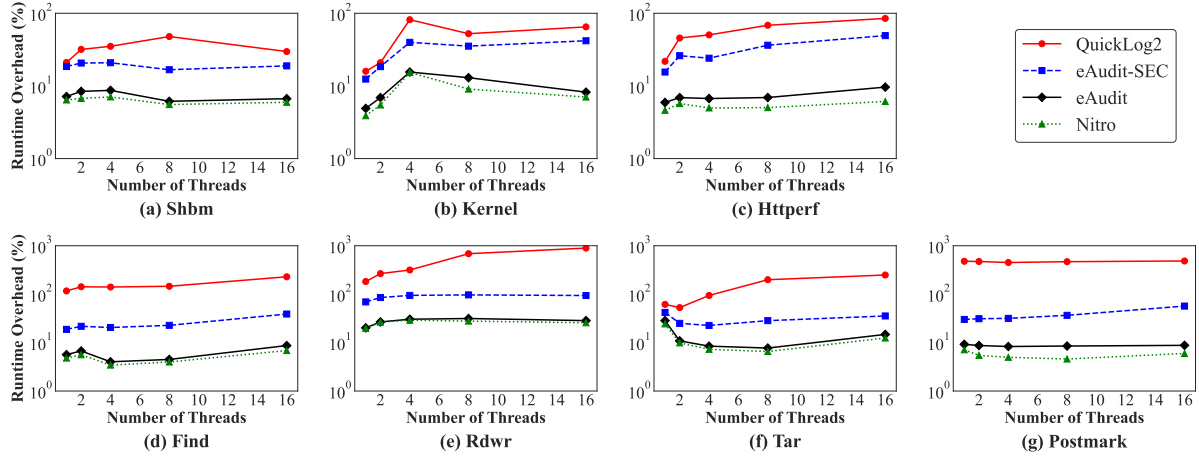
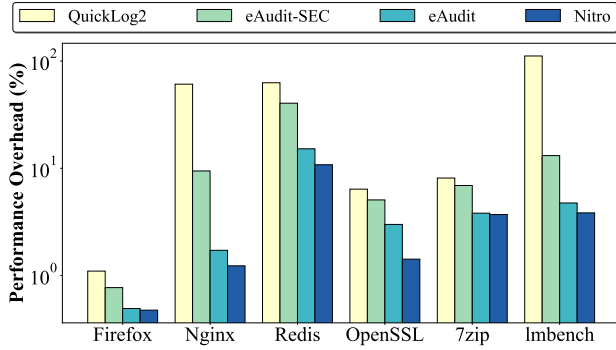**Figure 8: Runtime overhead comparison under the stress-test benchmark.**

Figure 9: Runtime overheads for the real-world benchmark.

stems from modifications to the core logging system (Linux Auditd [10]), which drops logs under load, leading to conflicts with the MAC logic. In contrast, NITRO avoids these issues entirely and offers superior stability. For I/O-intensive tasks such as `find` and `rdwr`, NITRO avoids overhead spikes and outperforms QuickLog2 by up to 30× while maintaining consistent performance. NITRO also achieves runtime overhead comparable to or lower than eAudit, owing to its optimized XLog and reduced kernel I/O. This confirms that strong security and low overhead can be achieved together. Results further show that simply adding cryptographic protections to eAudit (eAudit-SEC) is insufficient. We applied MACs to eAudit using Chaskey [61], as in NITRO. Although eAudit-SEC benefits from eBPF efficiency and occasionally outperforms QuickLog2, it fails to achieve both speed and security.

REAL-WORLD BENCHMARKS. Figure 9 shows that NITRO consistently achieves lower runtime overhead than all other systems across a diverse set of real-world benchmarks, demonstrating its superior runtime efficiency. This result highlights NITRO 's ability to sustain high performance under practical conditions where system demands vary. While performance differences are generally less pronounced in lower-load scenarios, NITRO still maintains an average performance advantage exceeding 4× over QuickLog2, reflecting the benefits of its architecture and MAC optimizations. These findings confirm NITRO's ability to deliver minimal overhead under a

wide range of workloads and establish it as a high-performance, secure logging solution that addresses key limitations of legacy logging system-based architectures.

These results underscore a central insight: retrofitting cryptographic protections onto legacy loggers like Auditd, as in Quick-Log2, leads to instability and inefficiency due to design mismatches. In contrast, NITRO co-designs logging and cryptographic components from the ground up. Compact encoding, eBPF-aware padding, verifier-safe key management, and per-CPU MACs ensure compatibility with eBPF while reducing log size, CPU usage, and I/O pressure. These choices enable NITRO to achieve fast logging.

6.2 RQ2: Data Loss

In each experiment, data loss is calculated as the ratio of the average number of lost logs (across $n = 30$ independent runs) to the average total number of logs. The ratio of two normal variables is not normally distributed, and thus one cannot use the standard way to compute confidence intervals. Instead, we employ Fieller's method [40] to compute confidence intervals for the ratio of two normal variables.

In particular, consider two normal variables X and Y in n observations with sample mean μ_X and μ_Y respectively; in our case X is the number of lost logs and Y is the total number of logs. Let s_X, s_Y, and $\sigma_{X,Y}$ be the standard deviation of X, the standard deviation of Y, and the covariance of X and Y respectively. Then the confidence interval of X/Y can be (approximately) computed via

$$\rho \pm \rho z \cdot \sqrt{\left(\frac{s_X}{\mu_X}\right)^2 + \left(\frac{s_Y}{\mu_Y}\right)^2 - \frac{2\sigma_{X,Y}}{\mu_X \cdot \mu_Y}} \ ,$$

where $\rho = \mu_X/\mu_Y$ is the ratio of the two means, and z is the Student t-value threshold with degree of freedom $n - 1$ for the desired confidence level. As reported in Table 1, the 90% confidence intervals in our case are consistently small across benchmarks, indicating reliable measurements and supporting the following key observations:

- **NITRO vs. QuickLog2:** NITRO consistently outperforms Quick-Log2 [45] in data loss, with QuickLog2 exceeding 90% loss in most benchmarks due to reliance on Auditd [10], which discards

Table 1: Data loss comparison using stress-test benchmarks.

System	Benchmarks (%)						
	Postmark	**shbm**	**tar**	**find**	**httperf**	**rdwr**	**kernel**
QuickLog2	93.92 ± 3.44	90.04 ± 2.18	96.41 ± 1.79	97.96 ± 1.54	95.20 ± 3.61	98.03 ± 1.05	30.92 ± 2.35
eAudit-SEC	35.67 ± 2.77	29.58 ± 2.51	40.17 ± 3.23	48.90 ± 4.10	34.64 ± 2.45	52.11 ± 3.85	9.75 ± 0.94
eAudit	1.94 ± 0.18	0.16 ± 0.03	2.41 ± 0.18	2.65 ± 0.26	0.87 ± 0.07	4.94 ± 0.46	0.00 ± 0
Nitro	0.37 ± 0.03	0.14 ± 0.01	1.52 ± 0.12	1.91 ± 0.17	0.52 ± 0.06	2.17 ± 0.12	0.00 ± 0

logs under load [51, 67]. When XMAC blocks logs in the kernel, Auditd cannot flush them to user space, leading to near-complete data loss in repeated high-load tests.

- **Nitro vs. eAudit-SEC:** Simply integrating cryptographic protections into eAudit, as done in eAudit-SEC, results in high data loss and overhead. This shows that cryptographic features must be co-designed with system architecture. Nitro achieves this through tight integration with kernel memory management and efficient log handling.
- **Nitro vs. eAudit:** While eAudit uses weighted scheduling and a two-level cache to limit data loss, Nitro goes further by adding a two-level time controller, eBPF-aware padding, and compact encoding. This design maintains data integrity and low loss even under extreme load.

To evaluate resilience under peak load, we use only stress-test benchmarks for this RQ, as they offer controlled conditions to measure data loss during extreme system pressure. Real-world application benchmarks, though valuable for general performance, impose lighter workloads and are less effective at exposing data retention limits. At the same time, all results are reported with the thread count set to 16, as this represents the heaviest load. The trend is similar for all other settings.

6.3 RQ3: Optimization Breakdown

There are two major architecture optimizations in our design: (i) using per-core tags instead of a single tag, and (ii) using a two-level time controller in the log buffering. To quantify their contributions in improving performance, we use a full factorial design analysis. Below, we briefly describe the procedure of the analysis; one can see, for example, [50, Chapter 18] for further details.

Full factorial design. Here we have two factors in the design, each of two levels: signing (single tag or per-core tags) and buffering (frequent versus occasional pushes). The eAudit-SEC system, for example, uses the single-tag signing and frequent pushes, whereas Nitro uses per-core tags and occasional pushes. To understand the effect of these two factors and their interaction, one would need to compare performance of all possible $2 \times 2 = 4$ systems. We will use the Postmark benchmark where each experiment has $r = 30$ replications. This is known as a full $2^2 \times r$ factorial design. The benchmark has 5 setting for 1, 2, 4, 8, 16 threads. Below, we describe how to analyze data for each setting.

Let $x_s = -1$ if one signs with per-core tags, and $x_s = 1$ otherwise. Let $x_b = -1$ if one uses occasional-push caching, and $x_b = 1$ otherwise. The running time of each system is modeled via a linear regression $y = Q_0 + Q_s x_s + Q_b x_b + Q_i x_s x_b$, where Q_0 is the mean running time of the four systems, Q_s is the effect of signing, Q_b is

Table 2: Allocation of variation on running time. Each reported number is a percentage. The unexplained fraction F_e is negligible in all settings.

	1 thread	**2 threads**	**4 threads**	**8 threads**	**16 threads**
F_s	52.49	56.84	61.75	72.17	80.15
F_b	43.92	41.13	36.62	26.92	19.24
F_i	3.59	2.03	1.63	0.91	0.61

the effect of buffering, and Q_i is the effect of the interaction of the two factors.

The benchmarks will provide the average running time of each of the four systems, meaning that we have a system of four linear equations, which is enough to determine the variables Q_0, Q_s, Q_b, Q_i. To determine the allocation of variation, we compare each actual running time Y_k and the predicted time y_k of the linear regression. Let $S_e = \sum_{k=1}^{4r} (Y_k - y_k)^2$ be the total square error. The signing, buffering, and their interaction help explain a fraction F_s, F_b, F_i respectively in the variation, where for each $k \in \{s, b, i\}$,

$$ F_k = \frac{Q_k^2}{S_e/4r + Q_s^2 + Q_b^2 + Q_i^2} \ . $$

The remaining (unexplained) fraction $F_e = 1 - (F_s + F_b + F_i)$ of variation is due to errors in the experiments.

6.3.1 Runtime Overhead. The results are reported in Table 2. The unexplained fraction F_e is negligible, suggesting a strong model fitness. The two factors have essentially no interaction because signing improves CPU operations, whereas buffering improves I/O ones. Signing is the dominant factor; for example it explains nearly 80% of the variation for the setting of 16 threads. As expected, the relative contribution of signing increases when the level of concurrency increases. Notably, the value of F_i remains averagely 1.75%, suggesting that the combination of the two factors has a positive effect, but not a significantly large one. Even if there is only one thread, F_s still plays a dominant role, because having a single thread does not necessarily mean that only one CPU core is used. Although only one core is responsible for running the process at any given moment, it may switch between cores based on system scheduling. In per-core tag versus single-tag systems, the data structures for storing system secrets differ fundamentally: one utilizes a Per-CPU Array, while the other relies on a global structure in kernel space (e.g., BPF_MAP). Even with thread=1, this structural difference leads to performance variation, making F_s still dominant under this setting.

6.3.2 Data Loss. Here we give a breakdown of the contributions of the two optimization factors (signing and buffering) on data loss via a full factorial design with the Postmark benchmark. However,

Table 3: Allocation of variation on the number of lost logs. Each reported number is a percentage. The unexplained fraction F_e is negligible in all settings.

	1 thread	2 threads	4 threads	8 threads	16 threads
F_s	26.94	29.85	31.24	34.95	42.92
F_b	72.53	69.10	66.31	62.28	53.45
F_i	0.53	1.05	2.45	2.77	3.63

(a) PostMark   (b) shbm
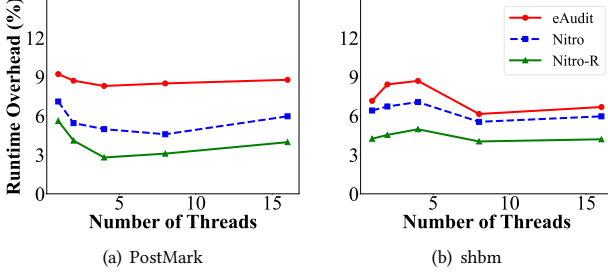
Figure 10: Comparison of Nitro-R, Nitro, and eAudit on two benchmarks. The remaining results are provided in Appendix J.

one can't directly do a linear regression on the data loss ratio, because this involves adding ratios of different denominators, which is meaningless. Instead, we do a linear regression on the number of lost logs. At the same time, although fast push clearly increases overhead, it is beneficial for reducing data loss, making it unnecessary to analyze the impact of fast push versus occasional push on data loss in this context. The key factor we designed to address data loss under high loads, the two-level time controller, essentially serves as an accelerator for cache pushes. Thus, we further refine the definition of x_b: Let $x_b = -1$ if using occasional-push caching with a two-level time controller, and $x_b = 1$ if using standard occasional-push caching.

The results are given in Table 3. Again, the interaction is small. It is evident that the two-level time controller plays a major role in reducing data loss, with its contribution showing a consistent pattern as the thread count changes. When the number of threads increases, the influence of per-core tags also grows; however, the two-level time controller remains dominant, accounting for nearly 71% when the thread count is only 1. Notably, as the load continues to increase, the combined effect of per-core tags and the two-level time controller also becomes more pronounced, indicating that their combined impact is more effective under high-load conditions.

6.4 RQ4: Overhead & Data Loss using Nitro-R

To thoroughly evaluate Nitro-R, we conducted comparative tests on all stress-test and real-world benchmarks, using Nitro and eAudit [67] as baselines. Due to space constraints, we present results for shbm and PostMark, with the remaining results included in Appendix J. The additional results exhibit trends consistent with those shown here. Figure 10 shows that Nitro-R achieves an average runtime overhead reduction of 24% on stress-test benchmarks and 11% on real-world benchmarks, relative to Nitro, which already maintains low overhead. It also reduces redundant logs by 87.91% on stress-test workloads and 49.76% on real-world ones. With these improvements, Nitro-R surpasses the performance of eAudit [67]. Its use of an in-kernel LRU hash table enables efficient log reduction

with minimal overhead, maintaining high performance under load. By removing duplicate logs before they reach user space, Nitro-R reduces cache contention and pressure on downstream components. This results in lower runtime overhead and improved resilience to data loss, without affecting data integrity. Under an expanded definition of data loss that considers the proportion of effective log data lost, Nitro-R is the only system to achieve zero loss across all benchmarks. These results were consistent across repeated trials and are summarized in Table 1.

7 DISCUSSION AND FUTURE DIRECTIONS

Combining QuickLog2 and eAudit. A natural question that arises is whether the security benefits of QuickLog2 can be combined with the performance advantages of eAudit. However, our experiments (Section 6) show that such integration is practically prohibitive due to conflicting design principles: 1) eAudit uses small buffers and frequent log pushes to reduce the tamper window, leading to high I/O overhead, while QuickLog2 relies on large buffers and batch processing for cryptographic efficiency; combining them forces frequent signing, negating batching benefits. 2) QuickLog2's AES-based MAC depends on AES-NI hardware acceleration, which is incompatible with eBPF. 3) eAudit supports multi-core logging, but QuickLog2 performs sequential cryptographic processing, creating bottlenecks when combined.

Cross-Platform Portability. Although designed for Linux, Nitro could be adapted for Windows, as the Windows platform now supports eBPF [26]. At the same time, Nitro's core design, including Per-CPU Array and ring buffer, is also supported by eBPF for Windows [29, 30]. Therefore, porting Nitro to other systems does not require major design changes. However, due to fundamental differences between Windows and Linux, the migrated Nitro requires modifications to its event capture hooks, such as the names of entry events [31]. The closed-source nature of Windows limits low-level kernel event monitoring, with most monitoring capabilities focusing on network traffic. Users can integrate Event Tracing for Windows (ETW) to assist in analyzing the runtime behavior of eBPF programs on Windows [28]. Secondly, the log encoding scheme also requires adaptation. Since Nitro's padding strategy relies on analyzing parameters of the monitored targets, the encoding rules must be redefined for the new platform. Specifically, known-length parameters need to be combined to meet the input length requirement of XLog's MAC, which represents an engineering effort. Lastly, the memory usage constraints differ across platforms. With the recent update to eBPF for Windows, which relaxes the size limitations [26], the optimal configuration proposed in this paper is now supported. While outside the focus of this paper, adapting Nitro for Windows environments is a practical future direction that could broaden its applicability in diverse security contexts.

Choosing between Nitro and Nitro-R. End users can choose Nitro or Nitro-R based on application needs. Nitro-R performs log reduction in the kernel. Compared to Nitro, it has lower runtime overhead and less data loss. It also removes redundant logs, such as repeated file reads, reducing storage use. Nitro-R fits scenarios with lower log sensitivity and stronger performance or storage demands, such as large-scale cloud services, CDNs, or internal

testing. For use cases needing higher log integrity, such as enterprise systems, financial platforms, government facilities, or critical infrastructure, NITRO is preferred.

LIMITATIONS. While NITRO and NITRO-R offer efficient, tamper-evident logging, some limitations remain. First, both systems support auditing and post-incident investigation, not runtime attack prevention, which aligns with other forensic tools [10, 45, 63, 67]. Second, NITRO-R boosts performance and cuts storage by filtering high-frequency, low-value events. This may reduce temporal precision, but prior work [68] shows limited impact on forensic usefulness. Third, NITRO depends on stable BPF tracepoints and CO-RE for cross-version compatibility. This lowers maintenance in most cases, but major kernel changes may still require BPF updates [72].

8 RELATED WORK

HARDWARE-ASSISTED SECURE LOGGING. While software-only solutions like NITRO can detect attacks, they cannot prevent an adversary from wiping log data from disk to erase evidence. A complementary approach involves using special-purpose hardware for secure log storage. For instance, HardLog [33] uses a RockPro64 development board to synchronously record predefined critical system calls. OmniLog [41] employs an NXP IMX8MQ-EVK board for more comprehensive system call logging and ensures synchronous log availability. SGX-Log [52] leverages Intel SGX to protect code and data in memory and on disk from tampering, even by privileged software. Custos [62] also uses Intel SGX, integrating cryptographic mechanisms to further enhance logging security. More recently, HitchHiker [71] leverages EL3-based memory permission switching on ARM platforms to improve log protection assurance. While it significantly reduces the TCB and protection delay, its reliance on periodic permission switching introduces non-negligible runtime overhead under log-intensive workloads.

EBPF-BASED LOGGING SYSTEM. The ease of deployment offered by eBPF has made it increasingly popular for building modern logging systems. Notable examples include tracee [16], Tetragon [4], Sysdig [3], eAudit [67], and Falco [2], all of which are either built from scratch or significantly enhanced using eBPF. However, none of these systems is tamper-evident or fully eliminates the window for log tampering. With NITRO, we introduce the first tamper-evident logging system based on eBPF, enhancing the security guarantees of prior systems without sacrificing performance.

LOG REDUCTION SCHEMES. Numerous log reduction schemes, such as those in [38, 43, 47, 49, 68–70], focus on minimizing audit log volume by offloading processing to user space, which introduces transfer overhead. In contrast, NITRO leverages in-kernel log reduction to eliminate the overhead of log transfers. This paper specifically implements Auditrim's [68] log reduction strategy within the kernel to minimize performance costs, achieving a log reduction ratio similar to AudiTrim's. Future adaptations of other log reduction schemes into the kernel could further enhance NITRO's efficiency in managing log data with reduced overhead.

9 CONCLUSION

This paper presents NITRO, the first tamper-evident logging system fully implemented in eBPF. By co-designing the cryptographic and systems architecture, NITRO addresses longstanding challenges in logging performance, granularity, and maintainability. Our enhanced variant, NITRO-R, further reduces overhead by incorporating in-kernel log reduction, demonstrating that practical, high-integrity logging is achievable even under high system loads.

10 ACKNOWLEDGMENT

REFERENCES

[1] 7zip Homepage. https://www.7-zip.org/.
[2] Falco: cloud native runtime security tool for Linux operating systems. https://github.com/falcosecurity/falco.
[3] Aaron Kili. Sysdig – a powerful system monitoring and troubleshooting tool for linux. https://www.tecmint.com/sysdig-systemmonitoring-and-troubleshooting-tool-for-linux/.
[4] Cilium. Cilium/tetragon: eBPF-based security observability and runtime enforcement. https://github.com/cilium/tetragon.
[5] The Apache Software Foundation, "ab - Apache HTTP server benchmark tool,". https://httpd.apache.org/docs/2.4/programs/ab.html.
[6] DARPA OPTC. https://github.com/FiveDirections/OpTC-data, .
[7] DARPA TC. https://github.com/darpa-i2o/Transparent-Computing, .
[8] What is eBPF? https://ebpf.io/what-is-ebpf/.
[9] Firefox. https://www.mozilla.org/en-US/firefox/.
[10] The Linux audit daemon. https://linux.die.net/man/8/auditd.
[11] Advanced Load Balancer, Web Server, Reverse Proxy - NGINX. https://www.nginx.com/.
[12] OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.
[13] Phoronix test suite. https://www.phoronix-test-suite.com/.
[14] Redis Ltd., "Redis benchmark". https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/.
[15] BPF ring buffer. https://www.kernel.org/doc/html/next/bpf/ringbuf.html.
[16] Aqua Security. Tracee: Runtime security and forensics using eBPF. https://github.com/aquasecurity/tracee.
[17] Cve-2017-16995. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16995, 2017.
[18] BSidesSF 2017: Security Monitoring with eBPF. https://www.brendangregg.com/Slides/BSidesSF2017_BPF_security_monitoring/, 2017.
[19] Falco vs. AuditD from the HIDS perspective. https://sysdig.com/blog/falco-vs-auditd-hids/, 2021.
[20] Speedometer 2.0. https://browserbench.org/Speedometer2.0/, 2023.
[21] Redis in-memory data structure store. https://redis.io/, 2023.
[22] Transparent computing engagement 3 data. https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md, 2024.
[23] Intel® Advanced Encryption Standard Instructions (AES-NI). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html, 2024.
[24] BPF Compiler Collection (BCC). https://github.com/iovisor/bcc, 2024.
[25] BPF Compiler Collection (BCC) Reference Guide. https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md, 2024.
[26] ebpf for windows. https://github.com/microsoft/ebpf-for-windows, 2024.
[27] Intel® 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2025.
[28] Towards debuggability and secure deployments of ebpf programs on windows. https://opensource.microsoft.com/blog/2022/10/25/towards-debuggability-

and-secure-deployments-of-ebpf-programs-on-windows/, 2025.

[29] ebpf for windows file reference. https://microsoft.github.io/ebpf-for-windows/ebpf__structs_8h.html, 2025.

[30] Getting linux based ebpf programs to run with ebpf for windows. https://opensource.microsoft.com/blog/2022/02/22/getting-linux-based-ebpf-programs-to-run-with-ebpf-for-windows/, 2025.

[31] Tutorial of ebpf for windows. https://github.com/microsoft/ebpf-for-windows/blob/main/docs/tutorial.md, 2025.

[32] Linux bpf documentation. https://docs.kernel.org/bpf/, 2025.

[33] A. Ahmad, S. Lee, and M. Peinado. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[34] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu. Atlas: A sequence-based learning approach for attack investigation. In *USENIX Security Symposium*, 2021.

[35] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *IN-DOCRYPT*, 2012.

[36] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *USENIX Security Symposium*, 2015.

[37] M. Bellare and P. Rogaway. The security oftriple encryption and a framework for code-based gameplaying proofs. In *EUROCRYPT 2006*, pages 409–426, 2006.

[38] H. Ding, S. Yan, J. Zhai, and S. Ma. Elise: A storage efficient logging system powered by redundancy reduction and representation learning. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[39] H. Ding, J. Zhai, D. Deng, and S. Ma. The case for learned provenance graph storage systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3277–3294, 2023.

[40] E. C. Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, 16(2):175–185, 1954. ISSN 00359246. URL http://www.jstor.org/stable/2984043.

[41] V. Gandhi, S. Banerjee, A. Agrawal, A. Ahmad, S. Lee, and M. Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023.

[42] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012.

[43] W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, D. Wang, Z. Chen, Z. Li, J. Rhee, J. Gui, et al. This is why we can't cache nice things: Lightning-fast threat hunting using suspicion-based hierarchical storage. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[44] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security (NDSS)*, 2020.

[45] V. T. Hoang, C. Wu, and X. Yuan. Faster yet safer: Logging system via Fixed-Key blockcipher. In *USENIX Security*, 2022.

[46] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan. SLEUTH: Real-time attack scenario reconstruction from COTS audit data. In *USENIX Security Symposium*, 2017.

[47] M. N. Hossain, J. Wang, R. Sekar, and S. D. Stoller. Dependence-preserving data compaction for scalable forensic analysis. In *USENIX Security Symposium*, 2018.

[48] M. A. Inam, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan. Faust: Striking a bargain between forensic auditing's security and throughput. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 813–826, 2022.

[49] M. A. Inam, Y. Chen, A. Goyal, J. Liu, J. Mink, N. Michael, S. Gaur, A. Bates, and W. U. Hassan. Sok: History is a vast early warning system: Auditing the provenance of system intrusions. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[50] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

[51] P. Jiang, R. Huang, D. Li, Y. Guo, X. Chen, J. Luan, Y. Ren, and X. Hu. Auditing frameworks need resource isolation: A systematic study on the super producer threat to system auditing and its mitigation. In *USENIX Security Symposium*, 2023.

[52] V. Karande, E. Bauman, Z. Lin, and L. Khan. SGX-Log: Securing System Logs With SGX. In *Asia CCS*, 2017.

[53] J. Katcher. Postmark: A new file system benchmark. 1997. URL https://api.semanticscholar.org/CorpusID:59816446.

[54] J. Katz and Y. Lindell. Aggregate message authentication codes. pages 155–169, 01 2008. ISBN 978-3-540-79262-8. doi: 10.1007/978-3-540-79263-5_10.

[55] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2003.

[56] K. H. Lee, X. Zhang, and D. Xu. Loggc: garbage collecting audit log. In *CCS*, 2013.

[57] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal. Towards a timely causality analysis for enterprise security. In *Network and Distributed System Security (NDSS)*, 2018.

[58] S. Ma, X. Zhang, and D. Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *Network and Distributed System Security (NDSS)*, 2016.

[59] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha. Kernel-supported cost-effective audit logging for causality tracking. In *USENIX Annual Technical Conference (ATC)*, 2018.

[60] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference (ATC)*, 1996.

[61] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede. Chaskey: An efficient mac algorithm for 32-bit microcontrollers. In *Selected Areas in Cryptography – SAC 2014*, pages 306–323, Cham, 2014. Springer International Publishing.

[62] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. W. Fletcher, A. Miller, and D. Tian. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *NDSS*, 2020.

[63] R. Paccagnella, K. Liao, D. Tian, and A. Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[64] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. Camflow: Managed data-sharing for cloud services. In *TCC*, 2015.

[65] T. Pasquier, X. Han, M. Goldstein, T. Moyer, D. Eyers, M. Seltzer, and J. Bacon. Practical whole-system provenance capture. In *Symposium on Cloud Computing*, 2017.

[66] L. Rice. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, 2023. ISBN 9781098135089. URL https://books.google.com/books?id=OqCyEAAAQBAJ.

[67] R. Sekar, H. Kimm, and R. Aich. eAudit: A fast, scalable and deployable audit data collection system. In *IEEE Symposium on Security and Privacy (SP)*, 2024.

[68] H. Sun, S. Wang, Z. Wang, Z. Jiang, D. Han, and J. Yang. Auditrim: A real-time, general, efficient, and low-overhead data compaction system for intrusion detection. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM, 2024. ISBN 9798400709593.

[69] Y. Tang, D. Li, Z. Li, M. Zhang, K. Jee, X. Xiao, Z. Wu, J. Rhee, F. Xu, and Q. Li. Nodemerge: Template based efficient data reduction for big-data causality analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[70] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[71] C. Zhang, J. Zeng, Y. Zhang, A. Ahmad, F. Zhang, H. Jin, and Z. Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2024.

[72] S. W. Zhong, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Revealing the unstable foundations of ebpf-based kernel extensions. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 21–41, 2025.

A PROOF OF PROPOSITION 1

Consider the following games P_1 and P_2. Game P_1 corresponds to game $\mathbf{G}_{\mathrm{MC}}^{\mathrm{forge}}(\mathcal{A})$. Game P_2 is similar to game P_1, but each call to $G(K_i, \cdot)$ is replaced by the corresponding call to a truly random function $f_i : \{0,1\}^* \rightarrow \{0,1\}^\tau$. Note that in game P_2, the correct aggregate tag T_r^* of the forgery messages is uniformly random over $\{0,1\}^\tau$, independent of whatever the adversary $\mathcal{A}$ receives. Thus the chance that it can guess T_r^* correctly is $2^{-\tau}$. In other words, $\Pr[P_2(\mathcal{A})] = 2^{-\tau}$.

To bound the gap between the two games above, consider the following adversary $\mathcal{B}$ attacking the (multi-user) PRF security of G.

Adversary $\mathcal{D}()$

$(M_1, \ldots, M_q, \sigma) \leftarrow\!\!\$ \, \mathcal{A}$
Return $(M_1, \ldots, M_q, \sigma)$

Adversary $\mathcal{D}(\sigma, T)$

$X_1, \ldots, X_q \leftarrow\!\!\$ \, \{0,1\}^n; S, K \leftarrow\!\!\$ \, \{0,1\}^n$
$(M_1', \ldots, M_r', T') \leftarrow\!\!\$ \, \mathcal{A}(S, K, X_1, \ldots, X_q, T, \sigma)$
Return $(M_1', \ldots, M_r', T')$

Figure 11: Constructed adversary $\mathcal{D}$ in the proof of Theorem 2.

It runs $\mathcal{A}$ and simulates game P_1. However, for each call to $G(K_i, \cdot)$, adversary $\mathcal{B}$ instead makes the corresponding oracle query $\textsc{Fn}(i, \cdot)$. Hence game $P_1(\mathcal{A})$ corresponds to game $\mathbf{G}_G^{\text{mu-prf}}(\mathcal{B})$ with challenge bit 1, whereas game $P_2(\mathcal{A})$ corresponds to game $\mathbf{G}_G^{\text{mu-prf}}(\mathcal{B})$ with challenge bit 0. Hence

$$\Pr[P_1(\mathcal{A})] - \Pr[P_2(\mathcal{A})] = \Pr[\mathbf{G}_G^{\text{mu-prf}}(\mathcal{B})] = \mathbf{Adv}_G^{\text{mu-prf}}(\mathcal{B}) \ .$$

Hence,

$$\begin{aligned}
\mathbf{Adv}_{MC}^{\text{forge}}(\mathcal{A}) &= \Pr[P_1(\mathcal{A})] \\
&= \Pr[P_1(\mathcal{A})] - \Pr[P_2(\mathcal{A})] + \Pr[P_2(\mathcal{A})] \\
&= \mathbf{Adv}_G^{\text{mu-prf}}(\mathcal{B}) + \Pr[P_2(\mathcal{A})] \\
&\leq \mathbf{Adv}_G^{\text{mu-prf}}(\mathcal{B}) + 2^{-\tau} \ .
\end{aligned}$$

B PROOF OF THEOREM 2

Without loss of generality, assume that $\mathcal{A}$ alters the logs. Since the adversary $\mathcal{A}$ receives the state S_q and key K_q, it can compute any key K_i (with $i > q$). Thus without loss of generality, we assume that $\mathcal{A}$ only generates at most q forgery messages. Indeed, suppose that $\mathcal{A}$ outputs $(M_1', \ldots, M_r', T')$ with $r > q$. Then it could instead produce $(M_1', \ldots, M_q', T_q')$ to win with the same advantage, where T_q' is computed as follows. Let $T_r' = T'$, and for each i from r down to $q + 1$, let $T_{i-1}' = T_i' \boxminus G(K_{i-1}, M_i')$. Hence from now on, we assume that $r \leq q$.

Consider the following games G_1 and G_2. Game G_1 corresponds to game $\mathbf{G}_{\text{XLog}[MC,F]}^{\text{fa}}(\mathcal{A})$. Game G_2 is similar to game G_1, but all the keys and states and tag masks are sampled uniformly at random. To bound the gap between the two games, consider the following adversary $\mathcal{B}$ attacking the (multi-user) PRF security of F. It runs $\mathcal{A}$ and simulates game G_1. However, for each call to $F(S_{i-1}, \cdot)$, adversary $\mathcal{B}$ instead makes the corresponding oracle query $\textsc{Fn}(i, \cdot)$. Hence game $G_1(\mathcal{A})$ corresponds to game $\mathbf{G}_F^{\text{mu-prf}}(\mathcal{B})$ with challenge bit 1, whereas game $G_2(\mathcal{A})$ corresponds to game $\mathbf{G}_F^{\text{mu-prf}}(\mathcal{B})$ with challenge bit 0. Hence

$$\Pr[G_1(\mathcal{A})] - \Pr[G_2(\mathcal{A})] = \Pr[\mathbf{G}_F^{\text{mu-prf}}(\mathcal{B})] = \mathbf{Adv}_F^{\text{mu-prf}}(\mathcal{B}) \ .$$

To bound $\Pr[G_2(\mathcal{A})]$, consider the adversary $\mathcal{D}$ attacking MC as described in Figure 11. It perfectly simulates game $G_2(\mathcal{A})$, and thus

$$\mathbf{Adv}_{MC}^{\text{forge}}(\mathcal{D}) = \Pr[G_2(\mathcal{A})] \ .$$

Summing up,

$$\begin{aligned}
\mathbf{Adv}_{\text{XLog}[MC,F]}^{\text{fa}}(\mathcal{A}) &= \Pr[G_1(\mathcal{A})] \\
&= \Pr[G_1(\mathcal{A})] - \Pr[G_2(\mathcal{A})] + \Pr[G_2(\mathcal{A})] \\
&= \mathbf{Adv}_F^{\text{mu-prf}}(\mathcal{B}) + \Pr[G_2(\mathcal{A})] \\
&= \mathbf{Adv}_F^{\text{mu-prf}}(\mathcal{B}) + \mathbf{Adv}_{MC}^{\text{forge}}(\mathcal{D}) \ .
\end{aligned}$$

C DESCRIPTION OF BENCHMARKS

We evaluate our system using two categories of workloads: *Stress-test Benchmarks* and *Real-world Benchmarks*. The *Stress-test Benchmarks* in Table 4 are adapted from prior work [67] and are designed to evaluate system behavior under high-load conditions. These benchmarks target specific subsystems such as file system metadata operations (e.g., postmark, find), process management (shbm), system call throughput (rdwr), network stack performance (httperf), and large-scale build workloads (kernel). Together, they provide a controlled environment to stress the system and measure overhead, data loss, and scalability under pressure. The *Real-world Benchmarks.* in Table 5 capture common workloads in modern systems, including web browsing, web serving, caching, cryptographic operations, file compression, and file system activity. They reflect practical deployment scenarios for logging and auditing systems, and help evaluate the system's effectiveness and overhead under realistic conditions.

Table 4: Stress-test benchmarks in this paper.

Name	Description
Postmark [53]	Simulation of a mail server.
tar [67]	Using tar to archive /usr/lib.
shbm [67]	A shell script executing echo repeatedly.
find [67]	Using find to print file names in /usr .
httperf [67]	A benchmark for web servers.
rdwr [67]	A C-program calling read and write.
kernel [67]	Compiling the Linux kernel.

Table 5: Real-world benchmarks in this paper.

Application	Benchmarks
Firefox [9]	Speedometer benchmark for web browser [20].
Nginx [11]	Benchmark for a Nginx web server [5] (with 10,000 requests for a 1KB file under 12 concurent threads in a local environment).
Redis [21]	Benchmark for a Redis server [14].
OpenSSL [12]	Benchmark for cryptographic operations [13].
7zip [1]	Benchmark for file compression [13].
lmbench	Benchmark for file creation and deletion (0K and 10K) in the lmbench benchmark suite [60].

D LIST OF SYSCALLS TO MONITOR

Below is the list of syscalls that we monitor:

- *Privilege escalation and tampering:* execve, execveat, setresuid, setuid, setfsuid, setfsgid, setgid, setregid, setresgid, kill, tgkill, tkill, ptrace.
- *Process provenance:* clone, clone3, exit, exit_group, mmap, mprotect, vfork.
- *File name and attribute change:* unlink, unlinkat, chmod, fchmod, fchmodat, chdir, fchdir.

Algorithm 1: In-Kernel Log Reduction

Input : Raw Log: R_L, Time Window: T_W
Output : Reduced Log: R_D

1   $H \leftarrow$ LRU Hash Table // Initialize or reload LRU Hash Table H
2   **foreach** e in R_L **do**
     /* Map log entry to a structured format   */
3      $\alpha \leftarrow$ (PID, Name, Args) $\leftarrow e$
     /* Check if entry is a duplicate within the specified time window */
4   **if** $\alpha \in H$ **then**
5   **if** $(Time_{current} - H[\alpha]) \leq T_G$ **then**
6   Drop e // Duplicate entry found, drop it
7   Continue
8   **else**
9              $T_c \leftarrow$ getCurrentTime()
10   Update $(\alpha : T_c)$ in H
11   **else**
12          $T_c \leftarrow$ getCurrentTime()
13   Insert $(\alpha : T_c)$ into H
14      $R_D \leftarrow R_D \cup$ process(e) // Add unique entry to reduced log
15   **return** R_D

- *Data endpoint creation:* connect, accept, accept4, open, openat, creat, socket.
- *Sends and receives:* sendto, recvfrom, sendmmsg, sendmsg, recvmsg, recvmmsg.
- *File descriptor operations:* dup, dup2, dup3, pipe, pipe2, tee, rmdir, truncate, ftruncate, mkdir, mkdirat, mknod, mknodat, socketpair.
- *Reads and writes:* read, readv, pread64, preadv, preadv2, write, writev, pwrite64, pwritev, pwritev2, splice, vmsplice.
- *Others:* getpeername.

E   DETAILED ALGORITHM OF NITRO-R

Nitro-R implements a verifier-safe, in-kernel log reduction algorithm using eBPF to eliminate redundant system log entries in real time. Algorithm 1 presents the detailed workflow. For each incoming log entry e from the raw log stream R_L, Nitro-R extracts a structured semantic key α, which consists of the process ID, system call name, and its arguments. This key represents the semantic identity of the event.

To detect duplicates efficiently, Nitro-R maintains an in-kernel Least-Recently-Used (LRU) hash table H, which maps each key α to the timestamp of its most recent occurrence. If the same key appears again within a defined time window T_W, the event is considered redundant and dropped. Otherwise, the current timestamp is stored or updated in H, and the entry is added to the reduced log R_D.

The algorithm ensures real-time performance and verifier compatibility by avoiding dynamic memory allocation and unbounded loops. Instead, it uses fixed-size structures, bounded iteration, and simple branching logic. Additionally, the LRU eviction policy of H prevents memory overflow, enabling scalable, long-term operation without kernel modification. The log reduction window T_W is configurable and can be dynamically adapted to system conditions if needed, although a static value (e.g., 1 second) is used in the default configuration. This design ensures that only unique entries are

retained and passed forward for cryptographic processing, minimizing both computational and I/O overhead without sacrificing log integrity or completeness.

F   RQ5: MEMORY USAGE

We analyze the memory consumption of Nitro from two perspectives: kernel space and user space, as shown in Figure 12. In the kernel, memory allocation and actual memory usage are distinct concepts. Nitro relies on eBPF-compliant data structures, and their sizes must be specified at compile time. As a result, regardless of how much memory is actually used at runtime, these structures are preallocated and persist in memory throughout execution. Specifically, Nitro allocates a Per-CPU Array of size S_p for each logical core and a shared ring buffer of size S_r. Given our configuration with $N = 36$, $S_p = 32$ KB, and $S_r = 64$ MB, the total allocated kernel memory can be computed as $N \times S_p + S_r$, which amounts to approximately 65 MB. This value represents the upper bound on memory that can be consumed, but not the actual memory usage at runtime. In Figure 12, the actual kernel memory usage of Nitro increases sharply after the stress-test benchmarks begin. It then fluctuates for a period before gradually decreasing to a lower level and stabilizing. In user space, we track the Resident Set Size (RSS) to reflect the memory usage of Nitro. Nitro relies on the Python runtime and BCC bindings to receive and process logs [24]. This component is initialized once and maintains stable memory usage throughout execution. Our evaluation on stress-test benchmarks shows that the user space memory consumption for Nitro consistently remains around 345MB, regardless of workload intensity or logging frequency.
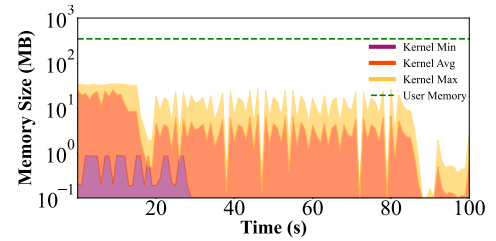
Figure 12: Kernel and user space memory usage of Nitro across all stress-test benchmarks. Kernel memory tracks in-kernel data structures; user memory is Nitro's Resident Set Size (RSS).

G   RQ6: COMPARISON WITH NODROP

To evaluate the efficiency of Nitro against NoDrop [51], a state-of-the-art tamper-proof logging system, we ran all seven stress-test benchmarks on an Intel-based machine, as NoDrop requires Intel MPK support. Other system configurations were kept identical to our main evaluation setup.

Figure 13 presents the complete comparison results across all benchmarks. On average, Nitro incurs over 10× lower runtime overhead than NoDrop across all benchmarks. In I/O-heavy scenarios like rdwr and find, the overhead reduction reaches up to 25×. This performance gap stems from key architectural differences. NoDrop uses Intel MPK for kernel-user isolation and relies on synchronized logging, resulting in frequent control transfers and coordination overhead. In contrast, Nitro employs a fully
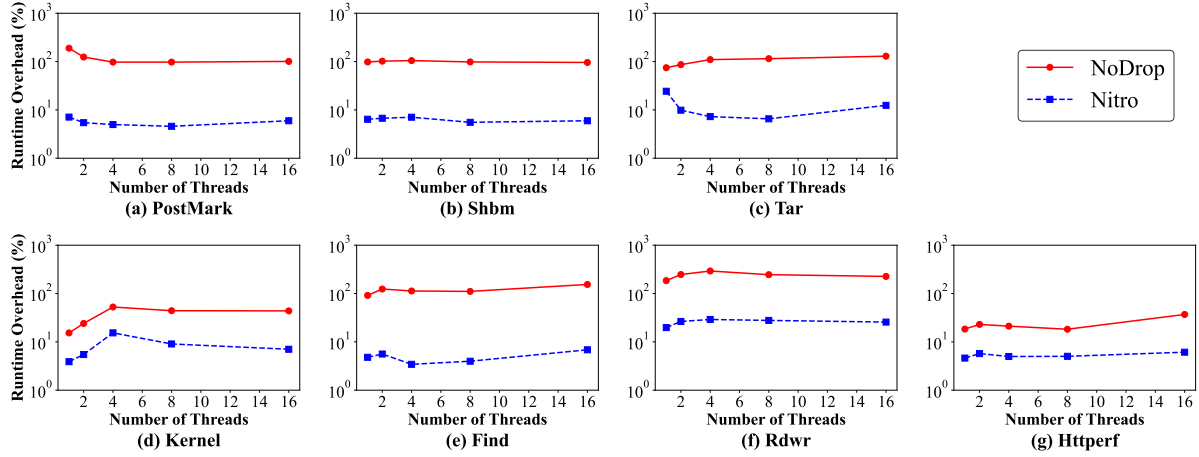
Figure 13: Runtime overhead of NoDrop and NITRO.

asynchronous, eBPF-based design, leveraging per-core logging and decoupled flushing to minimize syscall blocking and user space transitions.

While NoDrop guarantees zero data loss, we do not conduct a data loss comparison here, as that is its primary design goal. However, NITRO achieves near-zero loss (see Section 6) while offering significantly better performance. This tradeoff makes NITRO a more practical choice for high-throughput production environments.

H RQ7: LOG TAMPERING DETECTION AGAINST REAL-WORLD ATTACKS

Theoretically, Section 3 shows that it's impossible to alter logs in NITRO without being detected, unless one can break the PRF security of Chaskey. Still, it would be interesting to evaluate this on prior attacks. (This evaluation was in fact requested by reviewers of a prior submission to IEEE Security & Privacy 2025.) To showcase NITRO's ability on detecting log tampering, we consider attack scenarios from [62, 63]. All experiments are conducted in a controlled virtualized environment running Ubuntu 22.04 (with kernel version 6.5.0) and repeated 20 times. We generate encrypted tags for every log to test NITRO's ability to extract unmodified logs.

THE ATTACKS. The first attack is based on the race attack in [63] that led to CVE-2017-16995 [17]. The original attacks generates 25 forensic-relevant syscalls but contains hardcoded memory offsets that cause errors on modern Ubuntu kernels. We modify it to fix this issue, ensuring that all syscalls execute in order. To simulate the truncation attack, in each run, we sample a number $V \leftarrow^\$ \{1, \ldots, 25\}$ and only use the logs of the first V syscalls.

For the second attack, we adopt the forensic tampering scenario from Custos [62], using the dataset from its attack evaluation section. This dataset originates from the DARPA TC dataset [22] and is labeled as "Firefox Backdoor w/ Drakon In-Memory". It consists of sequential log events mapping to specific kernel syscalls, including file writes, process execution, and network communications.

EVALUATION RESULTS. In every case, NITRO points out that the logs are altered, and correctly extracts the list of unmodified logs.

I RQ8: LOG CORRECTNESS ANALYSIS

Following prior work [44, 49, 58], we validate the correctness of logs produced by NITRO by checking whether they preserve accurate causal relationships. Our goal is to confirm that asynchronous per-CPU logging does not compromise provenance integrity, and that the combination of timestamps and CPU core indexes suffices to reconstruct the original partial order of events. We adopt a methodology that combines online log collection with offline causality reconstruction, introduce formally defined reference baselines for comparison, and provide theoretical justification, as described below.

BASELINES. We define one baseline: B_1, corresponding to a centralized logging configuration for NITRO. The baseline replaces the per-core asynchronous buffer with a centralized `perf_buffer` that synchronously records all events in a globally ordered stream. All other aspects of the system remain unchanged. Let $\mathcal{L}_{B_1}$ denote the logs generated under this baseline, and let $\mathcal{L}_{\text{NITRO}}$ denote the logs produced by the default NITRO configuration with asynchronous per-CPU logging. This centralized baseline preserves a total order of system events and serves as ground truth for verifying the correctness of event relationships reconstructed from $\mathcal{L}_{\text{NITRO}}$.

ONLINE: LOG COLLECTION. The baseline configuration (B_1) logs all events into a single, centrally ordered stream, ensuring global temporal alignment. In contrast, NITRO buffers events independently on each CPU core and flushes them asynchronously to user space.

OFFLINE: CAUSAL RELATIONSHIP RECONSTRUCTION. We collect logs from both configurations while running the full suite of stress-test benchmarks (Section 6), which naturally exercise a wide range of causal relationships, including process creation, inter-thread communication, I/O callbacks, and signal delivery. From these logs, we analyze whether the expected causal relationships are preserved. In $\mathcal{L}_{B_1}$, relationships are derived directly from the globally ordered stream. For $\mathcal{L}_{\text{NITRO}}$, we merge per-core logs using timestamps and CPU core indexes to reconstruct the same set of causal dependencies. Specifically, we evaluate:

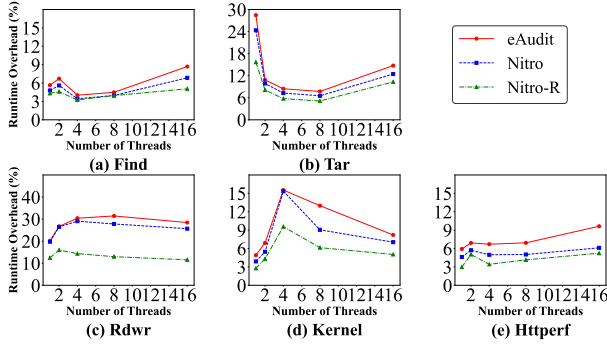- Whether the same causal dependencies appear in $\mathcal{L}_{\text{NITRO}}$ as in $\mathcal{L}_{B_1}$;

Figure 15: Remaining results for runtime overhead of eAudit [67], Nitro, and Nitro-R.
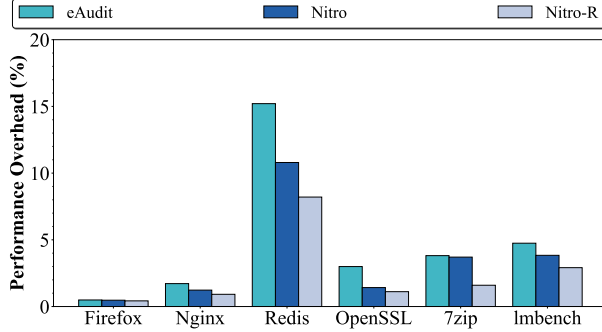

Figure 14: Runtime overhead comparison of eAudit, Nitro, and Nitro-R under real-world benchmarks.

- Whether any spurious, missing, or misordered relationships are introduced;
- Whether asynchronous logging leads to inconsistencies under concurrency.

THEORETICAL JUSTIFICATION. Let e_1 and e_2 denote two system events such that $e_1 \prec e_2$ in the baseline log $\mathcal{L}_{B_1}$. Each event corresponds to a syscall record captured by the logger, and includes metadata such as the syscall name, related parameters, timestamp t_i, and CPU core index c_i.

Assuming (1) clocks are synchronized across cores (or clock skew is bounded and known), and (2) intra-core event order is preserved, we claim that the ordering of events in the reconstructed logs from Nitro, denoted as $\mathcal{L}_{\text{Nitro}}$, can be recovered by lexicographic sort over (t, c):

$$e_1 \prec e_2 \Rightarrow (t_1 < t_2) \lor (t_1 = t_2 \land c_1 < c_2)$$

That is, if e_1 occurred before e_2 in the baseline, then the lexicographically ordered logs from Nitro will reflect the same precedence. Note that the secondary comparison on CPU core index does not imply a causal dependency, but rather serves as a deterministic tie-breaker to resolve simultaneous events with identical timestamps. In practice, such ties typically arise from concurrent, independent activities on different cores.

Across all stress-test benchmarks, the reconstructed event relationships from $\mathcal{L}_{\text{Nitro}}$ are consistent with those from $\mathcal{L}_{B_1}$. All tested interaction patterns preserve their expected ordering. No violations were observed, such as reordered, missing, or spurious dependencies. Additionally, we performed similar experiments with Nitro-R and we observed consistent results. These results confirm that timestamps and CPU core indexes provide sufficient information to reconstruct correct event relationships, even under asynchronous per-core logging. By validating against well-defined baselines, we demonstrate that Nitro and Nitro-R maintain log correctness and provenance fidelity across diverse workloads.

J REMANING EXPERIMENT RESULTS OF NITRO-R

Figure 15 shows the remainder of the seven stress-test benchmarks for comparing for comparing Nitro-R with Nitro and eAudit [67]. Figure 14 presents the results of real-world benchmarks. The trend here is the same as reported in Section 6. Across both stress-test and real-world workloads, the overhead of Nitro-R remains within 1.2× of Nitro on average, and is significantly lower than that of eAudit. For example, in the tar and find benchmarks, Nitro-R incurs only marginal overhead beyond Nitro despite additional signature generation, while still outperforming eAudit by a wide margin.

This efficiency stems from Nitro-R's use of a kernel-level redundant log reduction mechanism implemented using eBPF's built-in LRU map. Although the log reduction process introduces modest overhead during logging, it eliminates substantial downstream cost by suppressing unnecessary log entries before they reach the MAC and storage stages. In effect, this tradeoff results in net performance gains. In particular, log reduction reduces the volume of logs transmitted to both the Per-CPU Array and the ring buffer, thereby lowering I/O pressure and minimizing memory copy overhead. It also decreases the size of the MAC input, leading to reduced cryptographic computation during signature generation. As a result, Nitro-R achieves strong runtime integrity protection while maintaining overhead even lower than Nitro.