Proteus: A Practical Framework for Privacy-Preserving Device Logs

Sanket Goutam
Stony Brook University

Hunter Kippen
Samsung Research America

Michael Grace
Unaffiliated

Amir Rahmati
Stony Brook University

Abstract

Device logs are essential for forensic investigations, enterprise monitoring, and fraud detection; however, they often leak personally identifiable information (PII) when exported for third-party analysis. Existing approaches either fail to minimize PII exposure across all stages of log collection and analysis or sacrifice data fidelity, resulting in less effective analysis. We present Proteus, a privacy-preserving device logging framework that enables forensic analysis without disclosing plaintext PII or compromising fidelity, even when facing adversaries with access to multiple snapshots of the log files. To achieve this, Proteus proposes a two-layer scheme that employs keyed-hash pseudonymization of PII fields and time-rotating encryption with ratcheted ephemeral keys to prevent multi-snapshot correlation. For controlled sharing, clients export ratchet states that grant time-bounded access, permitting decryption of pseudonymized tokens that enable linkage and timeline reconstruction without exposing the underlying PII. Subsequent ratchet rotations ensure forward secrecy, while DICE-based attestation authenticates device provenance. We implement Proteus as a transparent extension to Android's logcat and evaluate it across three generations of hardware. Our results demonstrate a median latency of 0.2 ms per message and an average per-PII-field size overhead of only 97.1 bytes.

Keywords

Forensic logging, Mobile log privacy, PII leakage

1 Introduction

Security and operational analytics have long relied on collecting device logs, but the source of these logs is undergoing a fundamental shift. While telemetry was historically sourced from corporate-controlled workstations within a clear security perimeter, it is now increasingly gathered from a diverse ecosystem of user-centric devices. This trend is prominent in enterprise settings through Bring-Your-Own-Device (BYOD) policies, which integrate personal smartphones into corporate monitoring. Telemetry collection has now extended into the consumer space, where services collect logs from smart TVs, IoT hubs, medical wearables, and streaming clients for fraud detection and abuse prevention [12, 15, 26]. The expansion of log collection to personally owned devices creates a privacy dilemma. Unlike their corporate counterparts, these endpoints are deeply embedded in users' private lives, capturing sensitive data from communications, location history, and health applications. Consequently, granular logs essential for security and analytics inevitably expose users' personally identifiable information (PII) when exfiltrated, pitting the need for observability against users' right to privacy and often violating regulatory requirements [6, 16, 31].

This privacy dilemma persists because existing data protection techniques are fundamentally mismatched with the mobile endpoint threat model. Solutions like post-hoc redaction [32, 35], client-side taint tracking [2, 13, 19, 56], and encrypted auditing [7, 20] were designed for traditional enterprise environments with clear trust boundaries. As we detail in §3, they fail in the mobile context: they either expose sensitive data during collection, compromise the event-level fidelity required for forensics, or encounter critical deployment friction. Traditional data protection approaches are ill-suited to a world of user-owned devices, with continuous data exfiltration to honest-but-curious cloud platforms.

In this paper, we first formalize the requirements for privacy-preserving logging under a mobile endpoint threat model (§4). We consider an adversary who can capture multiple log snapshots over time (a multi-snapshot on-device observer) or receive exported logs and perform large-scale correlation (an honest-but-curious server). These assumptions reflect real-world threat vectors in which a malicious app is installed on a device, a privileged user has access to device logs, or an adversary gains access to cloud storage through a breach.

To resolve this tension, we present Proteus, the first logging framework that provides in-situ privacy protection without compromising forensic utility. Our design is built on a key insight: effective forensic analysis requires **correlation** (linking related events), not the **disclosure** of raw PII values. Proteus materializes this insight through a novel, two-layer cryptographic protocol applied during log generation. Proteus operates on any log fields designated as sensitive. At log emission, these fields are first pseudonymized with a keyed hash, creating stable tokens that enable correlation. Second, it encrypts these tokens with daily-rotated keys derived from a hardware-rooted hierarchical ratchet, defending against correlation attacks by multi-snapshot adversaries. When analysis is needed, a controlled sharing protocol grants time-bounded access to decrypt the pseudonyms without exposing the underlying PII. This entire process is anchored in DICE attestation, which cryptographically binds logs to an attested device state [51].

We demonstrate that Proteus provides robust protection through both theoretical analysis and deployment. We implemented and

Proceedings on Privacy Enhancing Technologies 2026(4), 891–905

https://doi.org/10.56553/popets-2026-0150

evaluated Proteus as both an end-to-end prototype and an Android library on three generations of hardware (details in Appendix A). We evaluate our deployment against the 30.3 million log entries in the LogHub dataset [60]. Our results show that Proteus's in-situ protection is highly efficient, adding only 2.41% in storage overhead and a median latency of 0.2 ms per message on real devices.

Contributions. In summary, this paper makes the following contributions:

- **First in-situ privacy-preserving framework for mobile logging**: Proteus protects sensitive data at the point of emission, preventing plaintext PII from ever leaving the device while preserving full forensic utility.
- **Formal threat model for mobile endpoint forensics**: We define the first threat model and security requirements for privacy-preserving logging systems centered around mobile devices with multi-snapshot adversaries and honest-but-curious cloud platforms.
- **Practical implementation and evaluation**: Android library deployed across three device generations with demonstrated practical overhead (median 0.2 ms latency, 2.41% storage) suitable for production deployment.

2 Background

Enterprise endpoint-monitoring solutions are expanding their coverage beyond traditional desktops and servers to mobile devices and user-personal endpoints. Unified Endpoint Management (UEM) platforms today support everything from basic mobile device management to comprehensive endpoint telemetry, policy enforcement, and security monitoring across smartphones and tablets in BYOD settings [11, 12, 23, 24, 45]. This shift reflects a fundamental change: organizations now treat mobile endpoints as integral to their security posture rather than peripheral assets.

Mobile devices, unlike their traditional counterparts, pose unique challenges with respect to data extraction policies. Unlike enterprise-owned desktops and servers, smartphones and tablets operate in deeply personal spaces and serve as repositories for sensitive user data. They contain private communications (messaging, email), precise location traces (GPS coordinates, visited venues), health information (fitness tracking, medical apps), and behavioral patterns that extend far beyond work-related activities. Yet modern UEM and SIEM (Security Information and Event Management) solutions require continuous log exfiltration from these devices to feed AI-driven analytics pipelines for anomaly detection, fraud prevention, and behavioral analysis [33]. This creates an inherent tension: the same granular logs that enable effective security monitoring inevitably expose PII when collected and analyzed by third-party platforms, even when the original purpose is purely security-focused.

Beyond enterprise BYOD scenarios, consumer-facing setups such as streaming platforms, smart TVs, and IoT hubs deploy similar endpoint data-collection pipelines for fraud detection and device-integrity monitoring. Modern SIEM solutions collect device logs to enable unified visibility across device, user, and network events [15, 33]. These logs capture fine-grained events (timestamps, process identifiers, network interactions, sensor readings, and runtime states) that allow security analysts to reconstruct timelines, build attack graphs, perform provenance analysis, and trace incident root causes [21, 55, 57]. In Table 1 we contrast traditional workstation logging with modern mobile endpoint logging, highlighting the fundamental differences in data ownership, sensitivity, and collection models. Thus, the challenge for mobile endpoint integration into SIEM is threefold: ensuring event-level fidelity (*i.e.,* capturing what, when, and where for each event), managing privacy (limiting PII leakage), and handling scale (volume of mobile logs). Unlike enterprise setups, where corporate ownership permits broad data collection, consumer-grade mobile devices require carefully designed telemetry pipelines that respect user privacy and uphold data compliance standards while providing sufficient granularity for forensic analysis.

The shift from traditional enterprise workstations to mobile and IoT endpoints fundamentally changes the privacy calculus for security monitoring. While enterprise-owned systems operate within clear trust boundaries and organizational policies, mobile devices cross into personal user space, where privacy expectations are high and regulatory protections apply. Current logging practices, however, were designed for the traditional model and fail to account for the heightened PII sensitivity and continuous exfiltration requirements of modern mobile SIEM workflows. This mismatch underscores the need for privacy-preserving logging approaches that operate effectively in mobile contexts.

3 Limitations of Current Approaches

Most existing mobile logging frameworks provide no protection for PII at the point of emission, relying instead on downstream controls. Current approaches primarily employ post-hoc sanitization [35], enterprise DLP (Data Loss Prevention) [36], and cryptographic controls over stored audit data [7]. While these methods address specific risks, they rely on trust assumptions that fail in mobile contexts, leaving residual PII exposure on devices, during extraction, or in post-processing stages. We examine each approach and identify why it fails to meet the requirements for privacy-preserving mobile logging.

A dominant industrial practice is the application of automated PII detection and redaction tools such as Microsoft Presidio [35], which employ rule-based recognizers and statistical models to identify sensitive entities (*e.g.,* names, email addresses, device identifiers) in textual data and redact them before further processing. Enterprise DLP and UEM platforms [18, 23, 30, 52] integrate similar detection mechanisms to enforce centralized inspection of data-in-transit and apply policy-based filters. Although effective within controlled enterprise pipelines, these approaches share a fundamental limitation: they embody a post-hoc privacy model. Raw logs are extracted from user devices and ingested by the service infrastructure prior to sanitization, thereby placing trust in the collection and staging servers. Even when protected in transit (*e.g.,* TLS), logs are commonly stored in cleartext within processing backends for indexing and analysis, extending broad access to service operators and conflicting with least-privilege principles [37]. In large-scale deployments, these systems also face operational complexity, high false-positive rates, and inconsistent enforcement across diverse endpoints. While valuable for compliance and policy governance,

Table 1: Comparison of Traditional vs. Modern SIEM Workflows

Aspect	Traditional SIEM (Enterprise Systems)	Modern SIEM (Mobile / Personal Devices)
Data Ownership	Owned and managed by the enterprise. Logs originate from corporate-controlled infrastructure and hosts. Centralized IT policies dictate collection, retention, and access control.	Data resides on user-owned mobile or IoT endpoints. Collection often depends on user consent or application permissions. Ownership is fragmented between user, OS vendor, and service provider.
Data Context for Analysis	Context primarily involves enterprise applications, network flows, and access control events. Analysis relies on structured corporate assets and identity systems (*e.g.,* Active Directory).	Context includes diverse app ecosystems, sensors, connectivity states, and background system events. Analysis requires correlating heterogeneous data sources with dynamic user activity.
PII Sensitivity	Limited; logs mostly contain operational or authentication data tied to employee identifiers within enterprise boundaries.	High; mobile logs often include location traces, user identifiers, contact information, and behavioral signals. Privacy-preserving collection is essential to mitigate the exposure of user information.
Collection Scope	Within a single enterprise trust boundary. Data aggregation occurs in a secured network perimeter managed by corporate SOCs (Security Operations Center).	Crosses user trust boundaries: data may traverse from user space to cloud SIEM platforms or third-party analytics engines. Requires federated trust and data minimization guarantees.
Privacy Risks	Risks are limited to insider misuse or over-retention of audit data.	Risk of re-identification, inference of private attributes, and regulatory non-compliance (*e.g.,* GDPR, CCPA) if logs are not anonymized or filtered.

post-hoc redaction and DLP frameworks fail to reconcile the asymmetry between device-side data generation and remote data control, leaving privacy enforcement external to the source.

Client-side analysis frameworks such as TaintDroid [13] and FlowDroid [2] attempt to detect privacy violations before data leaves the device by tracking the flow of sensitive information from sources to sinks. Although effective as research prototypes, these systems incur measurable runtime overheads, lack comprehensive coverage of proprietary binaries and third-party SDKs, and struggle to remain compatible across modern Android versions. Moreover, static taint policies cannot dynamically adapt to evolving data types or complex logging paths, resulting in both false alarms and missed leaks. Consequently, these tools have seen little adoption in production-grade telemetry or forensic pipelines.

Differential privacy mechanisms, such as Google's RAPPOR [14], take a fundamentally different approach by applying client-side aggregation and randomized response to provide formal privacy guarantees. However, these techniques are designed for statistical analysis of large populations and inherently compromise per-event fidelity. Forensic investigations, fraud detection, and security incident response all require precise event reconstruction and timeline correlation, all of which are explicitly sacrificed by differential privacy.

Cryptographic approaches to audit log protection have evolved from early integrity-focused designs [47] to more recent systems that support selective decryption. FA-SEAL [7], for instance, encrypts audit logs symmetrically and enables selective decryption of relevant segments through clustering and segmentation. While this addresses the wholesale-decryption problem, it does not solve the PII exposure issue: any decrypted fragment may still contain embedded identifiers that reveal sensitive information.

Collectively, these approaches reveal a persistent gap. Post hoc redaction and DLP frameworks can expose PII during collection and staging. Client-side taint tracking suffers from coverage gaps and deployment friction. Differential privacy sacrifices per-event fidelity. Encrypted audit systems either require wholesale decryption for analysis or impose prohibitive performance costs. No existing approach enforces in-situ privacy protection to prevent plaintext PII from leaving the device, while preserving the correlation, timeline reconstruction, and linkage analysis required by forensic investigations.

Design goals. To address these limitations, we outline the following design goals for an in-situ privacy-preserving logging framework:

- G1 **In-situ protection**: Protect sensitive fields at the point of generation so that PII never exists in exportable plaintext on or off the device.
- G2 **Utility preservation**: Preserve non-PII context and event structure to enable indexing, filtering, and timeline reconstruction.
- G3 **Linkage preservation**: Enable correlation and timeline reconstruction analysis over privacy-preserving derivatives (*e.g.,* keyed hashes) of PII without disclosing original values.
- G4 **Forward secrecy and device binding**: Defend against multi-snapshot adversaries who capture logs over time. Bind logs to attested device identity to prevent tampering and injection attacks.
- G5 **Post compromise security**: Provide a mechanism to restore secrecy of future logs after compromise of device state.
- G6 **Deployability**: Integrate with production logging frameworks (*e.g.,* Android `logcat`) without specialized hardware or application modifications, imposing minimal overhead.

Problem Statement. We seek to prevent PII leakage in device logs while preserving the forensic utility required for incident response,

fraud detection, and security analysis. Specifically, logs must support correlation and timeline reconstruction without ever exposing plaintext PII, even to honest-but-curious servers or adversaries who capture multiple log snapshots over time.

The limitations outlined above stem from a common root cause: existing approaches were designed for traditional enterprise threat models in which trust boundaries are well defined, and devices are corporate-owned. They fail in the mobile context because they do not account for multi-snapshot adversaries who can observe logs over extended periods, honest-but-curious cloud platforms that perform AI-driven analytics, or the deeply personal nature of data on user-owned devices. To address this gap, we must first formalize the requirements for a privacy-preserving logging system within a threat model appropriate for mobile endpoints. We establish this foundation in §4.

```
# Plaintext PII in logs (Status Quo)
10-15 14:23:47.821  2341  2341 I AuthService: Login attempt for user
↪  alice@example.com from device IMEI:352099001761481

# Microsoft Presidio: Redaction destroys linkage
10-15 14:23:47.821  2341  2341 I AuthService: Login attempt for user
↪  <EMAIL_ADDRESS> from device <IMEI>

# Simple Pseudonymized hash
10-15 14:23:47.821  2341  2341 I AuthService: Login attempt for user <PII
↪  type="EMAIL">BASE64-HASH-EMAIL</PII> from device <PII
↪  type="IMEI">BASE64-HASH-IMEI</PII>

# Our System: Encrypted PII (on-device and in-transit)
10-15 14:23:47.821  2341  2341 I AuthService: Login attempt for user <PII
↪  type="EMAIL">CIPHERTEXT-EMAIL</PII> from device <PII
↪  type="IMEI">CIPHERTEXT-IMEI</PII>
```

Figure 1: Simple pseudonymization preserves linkage but exposes correlation patterns to multi-snapshot adversaries. Proteus's two-layer protection scheme protects against unauthorized correlation while enabling controlled, user-authorized decryption for forensic analysis.

4 Threat Model and System Formalization

In this section, we formalize the notion of a privacy-preserving logging system and define the threat model it must defend against.

System Model. We consider endpoint logging pipelines in which consumer devices emit events via OS logging subsystems (*e.g.,* Android `logcat`). Log records may contain PII alongside non-PII context and are exported over authenticated channels to service-side storage and analytics. This establishes clear trust boundaries among the device, the transport, and the backend services. Throughout, we assume standard transport-layer protections (TLS/QUIC) and focus on exposures arising from log content at rest and during processing.

Adversary Model. The goal of the adversary is to recover plaintext PII, link identities across epochs, or infer sensitive attributes from logs. We model two complementary vantage points. First, a multi-snapshot on-device observer (a malicious app, privileged logger, or an attacker with repeated physical access) can obtain successive log snapshots via legitimate APIs or undetected extraction and attempt cross-time correlation, as observed in high-risk surveillance settings [28]. Second, a server-side adversary (honest-but-curious operators or breach adversaries) can access stored logs, associated metadata, previously shared decryption keys, and analytics-derived tokens across many devices and epochs, enabling linkage at scale. Neither adversary controls the OS kernel or device trust anchors.

Trust Assumptions. We assume the device OS and the logging modules execute correctly and are not compromised; cryptographic primitives and transport protections are implemented correctly with high-quality randomness; initial device identity, attestation, and key provisioning are trustworthy; and PII detection (*e.g.,* Regex) attains high recall for common sensitive types, leaving a residual risk from undetected fields that deployment policy must mitigate.

Security Objectives. Under this threat model, a solution must satisfy the design goals articulated in §3. Specifically, it must ensure that PII remains confidential against both on-device observers (who capture multiple snapshots) and honest-but-curious servers (who receive exported logs), while preserving the forensic utility required for correlation and timeline reconstruction. Both forward secrecy and post-compromise security are essential: compromise of the current device state must not expose prior logs or allow for recovering the security of future logs. Finally, logs must be cryptographically bound to attested device identity to detect tampering and prevent injection attacks.

Out of Scope. Physical attacks, firmware modification, side-channel extraction, and denial-of-service are considered out of scope. Metadata privacy (timestamps, sizes) and traffic-analysis resistance (against side-channel attacks) are considered orthogonal to PII protection.

5 Proteus

Proteus is the first in-situ logging framework that prevents the disclosure of PII while preserving full forensic utility. Our design rests on a key insight: forensic analysis requires *correlation*—creating linkage between related events—but not *disclosure* of the actual PII values. This suggests that simple pseudonymization of PII fields could protect against most privacy attacks that involve harmful PII disclosure. However, simple pseudonymization is insufficient against the multi-snapshot adversary model of §4. An adversary who captures logs over time can correlate pseudonymized tokens to infer linkage between user activity and events, revealing associations even without recovering the plaintext PII. For example, observing that `BASE64-HASH-EMAIL` appears in logs during office hours, alongside work-related activity, reveals behavioral patterns and enables profiling attacks.

A simpler approach would be to obfuscate all PII. Microsoft Presidio [35], the current state-of-the-art system, employs this strategy to manage PII-injected logs used in analytics tools. However, as Figure 1 illustrates, this post-hoc redaction not only lacks protection at rest or during collection and analysis, but also sacrifices the forensic fidelity needed for correlation. Proteus resolves this trade-off by operationalizing the key insight that analysis requires linkage, not recovery, through a two-layer protection protocol that enables user-authorized, controlled disclosure of data for forensic analysis without sacrificing utility for privacy.

Consider four approaches to handling a typical Android log entry containing an email address and device identifier (Figure 1):

The status quo exposes plaintext PII to all parties. Microsoft Presidio's post-hoc redaction replaces PII with generic type labels (<EMAIL_ADDRESS>, <IMEI>), preventing disclosure but destroying forensic utility. All users become indistinguishable, making it impossible to track repeated login attempts or correlate suspicious activity across sessions. Simple pseudonymization addresses both problems by replacing PII with stable keyed hashes (BASE64-HASH-EMAIL), enabling linkage while preventing plaintext recovery. However, this approach fails against multi-snapshot adversaries who can observe correlation patterns over time—*e.g.,* inferring that BASE64-HASH-EMAIL corresponds to a work email address based on temporal activity patterns.

With Proteus, we introduce an additional layer of protection. At log emission, PII fields are first pseudonymized via keyed hashing, then encrypted with daily-rotated keys. The resulting CIPHERTEXT EMAIL value is a base64-encoded ciphertext that prevents unauthorized correlation: even adversaries who capture multiple log snapshots cannot link events without decryption keys. This encrypted form is stored on-device and transmitted to servers, ensuring that plaintext PII never exists in an exportable form. When forensic analysis is required, servers receive user-authorized controlled access (via the protocol in §4) to decrypt ciphertexts and recover the underlying pseudonymized hashes (BASE64-HASH-EMAIL). These stable, base64-encoded 16-byte hashes enable correlation of all events involving the same email address across epochs originating from the same device. Crucially, even the forensic server with decryption keys cannot reverse the keyed hash to obtain alice@example.com—analysts can determine *whether* two events involve the same user, but not *who* that user is. This two-layer protection scheme ensures confidentiality against unauthorized observers (via encryption) and unlinkability to real identities (via pseudonymization) while preserving full forensic utility through controlled access.

We designed Proteus to enable remote device log collection and analysis for forensic investigations. The end-to-end architecture is shown in Figure 2. We will now detail the cryptographic designs behind the client and server components.

5.1 Client-Side Components

Key Derivation and Initialization. We choose DICE to be at the heart of our framework because it addresses three problems that privacy-preserving logging requires but that software-only approaches cannot reliably address. First, DICE yields a hardware-rooted CDI derived from a device-unique secret and measured software state. Keys derived from the CDI are, by construction, bound to both the physical device and its integrity measurements [51, 53]. Second, this binding naturally supports device attestation: the measurements that define the CDI also produce evidence that a verifier can check before trusting exported material. Third, DICE has been standardized by the Trusted Computing Group (TCG) and has seen increased adoption across manufacturers as a foundation for device-bound identifiers and attestation, making it a pragmatic choice for large-scale deployment [53]. In this model, the hardware's role is to securely produce the CDI based on software measurements. Proteus, operating in user-space, then uses this CDI as the root of its key hierarchy. Our implementation models this interface by taking a simulated CDI as input.

Algorithm 1: Client initialization

Input : CDI, Server public key B
Output : Initial keys and state
// Derive root keys from hardware-rooted CDI and Server pk
$K_{\text{hash}} \leftarrow \text{KDF}(\text{CDI}, \text{“hmac-key”}, 32)$
$\text{nonce}_{\text{init}} \leftarrow \text{random}(32)$
$a \leftarrow \text{DeriveX25519Priv}(\text{nonce}_{\text{init}}, \text{“dh-init”})$
$R_0 \leftarrow \text{KDF}(\text{CDI}, \text{ECDH}(a, B), \text{“root-key”}, 32)$
// Initialize chain key for current date
$d_0 \leftarrow \text{current\_date}()$
$ck_0 \leftarrow \text{KDF}(R_0, \text{“ck-init”} \,\|\, d_0, 32)$
store $(R_0, K_{\text{hash}}, a, ck_0, d_0)$

Proteus anchors all long-term secrets in the CDI so that they are usable only on a correctly measured device state. The client derives two keys from the CDI: (i) a per-system hash key K_{hash} used exclusively on-device for PII pseudonymization; and (ii) the root ratchet key R_0. R_0 is then used to initialize the symmetric ratchet, which generates daily encryption keys, thereby ensuring forward secrecy against multi-snapshot adversaries. Together, these realize *attestation* (binding logs to attested device state), *integrity* (measured-boot binding of derived keys), and *authentication* (mutual authentication and export encryption during sharing), addressing design goals G1 (in-situ protection), G3 (linkage preservation), and G4 (forward secrecy and device binding). Algorithm 1 details the initialization process. Notice that the derivation of R_0 is identical to the RatchetInitAlice procedure in §3.3 of the original description of the double ratchet algorithm [42], where the CDI takes the place of SK. Because the CDI is withheld from the server, it cannot derive R_0 (and the subsequent chain keys) without additional interaction with the client.

Two Layer PII Protection. The Proteus architecture is designed to protect any data field that is identified as PII, whether by an automated scanner or, more robustly, by explicit developer annotation at the logging call site. For any such identified field, the client enforces in-situ protection through a two-layer design that separates pseudonymization from encryption. When a log message containing PII is emitted, Proteus first pseudonymizes detected PII fields via keyed hashing, then encrypts the resulting tokens with daily-rotated ephemeral keys. This design ensures that even if encryption keys are compromised, plaintext PII is never recovered—only pseudonymized tokens suitable for correlation analysis.

Daily Key Ratcheting. To defend against multi-snapshot adversaries and provide forward secrecy, the client derives a fresh encryption key for each day using a ratcheting mechanism (Algorithm 2). Each day, the chain key ck_t is used to derive both the next chain key ck_{t+1} and the daily message key K_t via a one-way key derivation function. Once ck_{t+1} is derived, the old chain key is deleted, preventing backward derivation and ensuring that compromise of the current state does not expose prior logs.

Hierarchical Ratchet for Post-Compromise Security. Proteus employs a hierarchical ratcheting mechanism to protect against advanced adversaries who compromise a short-term chain key and attempt to recover all future logs. While daily ratcheting provides

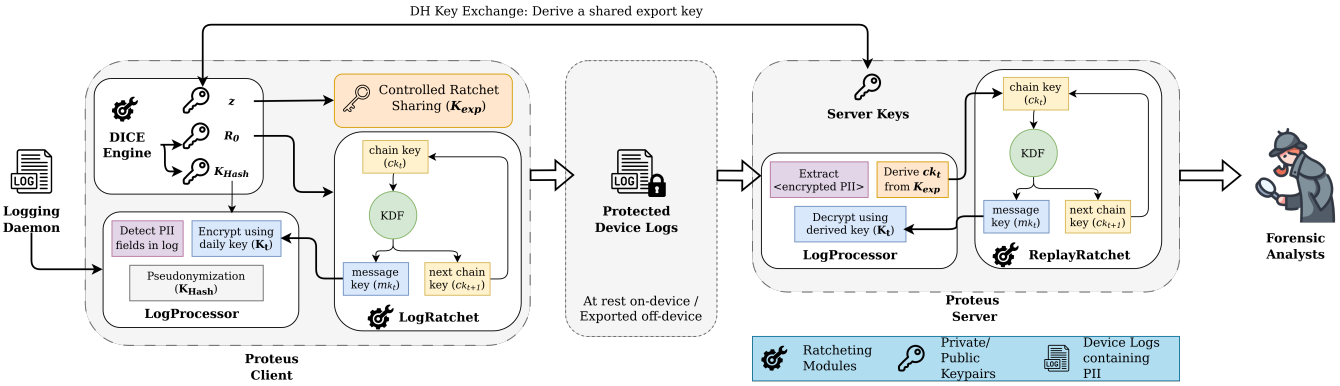


Figure 2: Proteus is designed as a client-server system geared towards forensic data logging and collection, introducing an in-situ framework for sensitive data protection.

Algorithm 2: Daily key ratcheting

Input : Current chain key ck_t
Output : Next chain key ck_{t+1} and message key K_t

Function KDF_CK(ck):
  $x \leftarrow \text{KDF}(ck, \text{“ratchet”}, 64)$
  $ck' \leftarrow x[0{:}32]$
  $mk \leftarrow x[32{:}64]$
  return (ck', mk)

// Daily key derivation
$(ck_{t+1}, mk_t) \leftarrow \text{KDF\_CK}(ck_t)$
mk_t *// Field encryption key for day t*
store ck_{t+1}, mk_{t+1} *// Update chain state*
delete ck_t, mk_t

Algorithm 3: PII field protection (log emission)

Input : PII plaintext field p, K_{hash}, encryption key K_t for day t, server_id, date t
Output : Protected field ciphertext c

// Layer 1: Pseudonymization via keyed hashing
token $\leftarrow \text{HMAC}(K_{\text{hash}}, p)$ *// Hash plaintext PII*
// Layer 2: Encryption with daily-rotated key
nonce $\leftarrow$ random(12)
$c \leftarrow \text{AE.Enc}(mk_t, \text{nonce}, \text{token}, \emptyset)$
return c *// Ciphertext stored in log*

forward secrecy for past logs, it does not protect future logs if the current chain key is compromised. To counter this threat and preserve secrecy guarantees throughout the system's lifetime, Proteus enforces a root key rotation whenever logs are exported to a server for analysis. Note in the sharing protocol described in 4, providing the chain key ck_{t^*} is an intentional compromise of the client's internal state. Viewed in a secure messaging context, all encrypted logs can be thought of as undelivered messages (as the client has not informed the server of an outer ratchet update). Once the client reveals any chain key via the controlled sharing protocol, it must rotate the root key R_0 by mixing in fresh entropy from the ephemeral Diffie–Hellman exchange performed during the sharing protocol. This ensures the new root key is unpredictable and severs the cryptographic chain from the old state, guaranteeing post-compromise security (G5).

PII field protection during log emission. Algorithm 3 details the two-layer protection applied to each detected PII field. In the first layer, the plaintext PII field p is pseudonymized by computing a keyed HMAC using a server-specific hash key $K_{\text{hash},t}$ derived from the long-term hash key K_{hash} (which is never shared). This produces a correlation token that enables linkage analysis without revealing the original value. In the second layer, the token is encrypted with the daily encryption key K_t using authenticated encryption. The resulting ciphertext is stored in the log, ensuring that on-device observers, network adversaries, and multi-snapshot attackers cannot correlate PII across log dumps. While Proteus uses keyed HMAC for pseudonymization, distance-sensitive hashing [3] could be substituted for PII types where similarity-based correlation is valuable—for example, hashing geolocation coordinates such that nearby locations produce similar tokens, enabling proximity-based forensic analysis without exposing precise coordinates.

Controlled Sharing Protocol (Client Side). To grant a forensic server time-windowed access to logs, the client initiates a controlled sharing protocol. The client exports the ratchet state for a specified interval, from a chosen start date t^* to the current day, enabling the server to derive daily decryption keys for that period only. The client never shares the hash key K_{hash}, ensuring that decryption yields only correlation tokens rather than plaintext PII. Immediately after export, the client rotates its root key R_0 (as described in §7), ensuring the server cannot decrypt logs generated after the current day. This provides post-compromise security and strictly bounds the decryption window. Algorithm 4 presents the client-side protocol.

5.2 Server-Side Design

Controlled Sharing Protocol (Server Side). The server receives the encrypted grant from the client and uses its long-term private key to derive the shared export key. After decrypting and verifying the grant, the server obtains the intermediate chain key ck_{t^*} for the start date t^*. The server then employs a *replay ratchet* mechanism:

Algorithm 4: Controlled sharing protocol

CLIENT SIDE:

Input : Ephemeral server public key B', start date t^*, current date d
Output : Grant sent to server

// Generate ephemeral ECDH keypair
$\text{nonce}_{\text{grant}} \leftarrow \text{random}(32)$
$a' \leftarrow \text{DeriveX25519Priv}(\text{nonce}_{\text{grant}}, \text{“export-keygen”})$
$A' \leftarrow a' \cdot G$

// Derive export encryption key
$z \leftarrow \text{ECDH}(a', B')$
$K_{\text{exp}} \leftarrow \text{KDF}(z, \text{“export-kdf”}, 32)$

// Re-derive ck_{t^*}
if $t^* < d$ **then**
  $ck_0 \leftarrow \text{KDF}(R_0, \text{“ck-init”} \parallel d_0, 32)$
  for $t \leftarrow d_0$ **to** $t^* - 1$ **do**
    $(ck_{t+1}, \_) \leftarrow \text{KDF\_CK}(ck_t)$ *// mk_t unused*

// Package and encrypt grant
$\text{AAD} \leftarrow \text{server\_id} \parallel \text{device\_id} \parallel \text{attest\_digest} \parallel \text{grant\_id} \parallel d$
$M \leftarrow \text{AE.Enc}(K_{\text{exp}}, \{ck_{t^*}, t^*\}, \text{AAD})$
send (A', M, AAD) to server

// Post-grant rotation for forward secrecy
$R'_0 \leftarrow \text{KDF}(R_0, z, \text{“root-key-rotate”}, 32)$
$ck'_0 \leftarrow \text{KDF}(R'_0, \text{“ck-init”} \parallel (d{+}1), 32)$
delete $\{R_0, ck_d, a, d_0\}$
store (R'_0, ck'_0, a', d)

SERVER SIDE:

Input : Server ephemeral keypair (b', B'), grant date d, received $(A', M, \text{nonce}, \text{AAD})$ from client
Output : Daily keys $\{K_t\}$ for $t \in [t^*, d]$

$z \leftarrow \text{ECDH}(b', A')$
$K_{\text{exp}} \leftarrow \text{KDF}(z, \text{“export”}, 32)$

// Decrypt and verify grant
$\{ck_{t^*}, t^*\} \leftarrow \text{AE.Dec}(K_{\text{exp}}, \text{nonce}, M, \text{AAD})$
if verification fails **then** abort

// Forward-only key derivation
for $t \leftarrow t^*$ **to** d **do**
  $(ck_{t+1}, mk_t) \leftarrow \text{KDF\_CK}(ck_t)$
  $K_t \leftarrow mk_t[0{:}32]$
  store K_t for day t

it initializes its local ratchet state with ck_{t^*} and iteratively applies the same one-way key derivation function used by the client (Algorithm 2) to derive daily decryption keys for each day in the authorized time window $[t^*, \text{current}]$. This forward-only replay ensures that the server can decrypt logs only within the granted interval, without the ability to derive keys for dates before t^* or after the current day (due to the client's post-export root key rotation). The server-side protocol is shown in the right column of Algorithm 4.

Token Recovery and Forensic Analysis. After receiving daily decryption keys via the controlled sharing protocol, the forensic server decrypts protected log fields to recover correlation tokens. Algorithm 5 details the token recovery process. For each ciphertext

Algorithm 5: Pseudonymized token recovery (server side)

Input : Protected field ciphertext c from log, decryption key K_t for day t
Output : Pseudonymized token for PII correlation

// Decrypt to recover pseudonymized token
$\text{token} \leftarrow \text{AE.Dec}(K_t, \text{nonce}, c, \emptyset)$
if decryption fails **then** abort
return token *// Used for correlation*

c in the logs, the server applies the corresponding daily key K_t to obtain the pseudonymized token. These tokens enable linkage analysis, timeline reconstruction, and cross-device correlation without exposing plaintext PII. Crucially, since the server never receives the hash key K_{hash}, it cannot reverse the pseudonymization to recover the original sensitive values. The semantic structure of logs remains intact. The server can observe event sequences, construct attack graphs, and reconstruct timelines, while preserving privacy even under an honest-but-curious server model.

This design naturally conforms to the workflow of real-world forensic pipelines, where initial investigations do not require full PII disclosure. Analysts first perform correlation analysis over pseudonymized tokens to identify patterns of suspicious activity and narrow the suspect set. Only once a specific device or user is flagged as suspicious does the investigation escalate to PII recovery: the analyst requests the hash key K_{hash} from the device owner (subject to appropriate legal authorization), enabling reversal of the pseudonymization and recovery of plaintext PII exclusively for the identified suspect's logs. This two-phase model—correlation first, targeted disclosure second—minimizes unnecessary PII exposure across the broader user population while preserving the investigative capability needed to act on confirmed suspects.

We implemented Proteus in two configurations: an end-to-end prototype in Python for evaluating the cryptographic protocols and macro-level performance on bulk log data, and an Android logging library for micro-benchmarking on real-world devices. The Android library provides an API that transparently integrates with existing applications, performing the client-side PII protection pipeline in user space. A detailed description of both implementations is provided in Appendix A.

6 Evaluation

We evaluate Proteus using two complementary approaches: an end-to-end prototype for macro-level measurements (storage overhead, bulk processing latency) and an Android library deployment for micro-level performance analysis (per-message latency, throughput). Both implementations are detailed in Appendix A. Our evaluation addresses the following research questions:

- RQ1 **Feasibility:** Can Proteus operate within practical performance boundaries for production logging systems across diverse hardware generations?
- RQ2 **Storage Efficiency:** What storage overhead does Proteus's two-layer protection introduce across different PII types, and how does it compare to plaintext logging?

Table 2: Test devices used for stress testing Proteus client performance. SD: SnapDragon, GB6-MC: Geekbench 6 multi-core scores [38–40].

Device	SoC	RAM	GB6-MC	OS
Pixel 2 (2017)	SD 835	4GB	1508	Android 11
Tab S6 (2019)	SD 855	8GB	2834	Android 12
Pixel 6a (2022)	Tensor G1	6GB	3208	Android 13

RQ3 **Performance Scalability:** How do per-message latency and sustained throughput scale across devices with varying capabilities, and how does Proteus compare to native Android logging?

RQ4 **Processing Bottlenecks:** Which pipeline components (key derivation, PII detection, encryption) dominate processing time, and where should optimization efforts focus?

RQ5 **Real-World Applicability:** Can Proteus handle realistic log workloads at scale, processing log entries under high message rates with acceptable latency for both client and server operations?

RQ6 **Security Guarantees:** Does Proteus achieve confidentiality of PII, forward secrecy against multi-snapshot adversaries, and integrity of exported logs under the threat model in §4?

6.1 Experimental Setup

Test Devices. Most modern smartphones ship with the requisite hardware support for DICE attestation. To demonstrate Proteus's feasibility and performance across a wide range of hardware, including legacy devices, we selected three generations of Android devices spanning 2017–2022. All three of our test devices (Table 2) possess the necessary hardware capabilities to support a DICE implementation [17]. However, at the time of our evaluation, a stable, publicly available Android API that enables third-party applications to access the DICE framework was not readily available. This ecosystem limitation, rather than a hardware constraint, necessitated simulating the presence of a DICE provider to supply the initial hardware-rooted keys. This methodology allows us to precisely measure the performance overhead of the Proteus cryptographic pipeline itself, which is the primary focus of our evaluation. Our results, therefore, demonstrate the practicality of the logging framework across a range of capable hardware, assuming that OS-level DICE APIs become available.

End-to-End Prototype. For the end-to-end prototype evaluation, we use an Ubuntu 20.04 system with 32 GB RAM. The evaluations are not memory- or computationally intensive, so the prototype runs readily on any commodity PC. We open-source our implementation for the broader research community (§**??**).

6.2 Macro Evaluation

We evaluated the end-to-end Proteus prototype (described in §A) using the LogHub dataset [60], a large collection of system logs spanning diverse applications, servers, and devices. We focused on Android device logs comprising 25 files with 30.3 million log entries, totaling 3.37 GB of raw data. Using this dataset, we simulated a realistic log streaming, collection, and analysis pipeline to measure Proteus's macro-level performance characteristics.

Size Overhead (RQ2). Figure 3 presents a granular breakdown of Proteus's storage efficiency across different PII types. Figure 3a shows the distribution of PII occurrences by type in the dataset. Passing the logs through Proteus's pipeline introduced 83.2 MB of additional overhead on top of the original 3.37 GB, resulting in a 2.41% increase in size. Figure 3b shows the relative contribution of each PII type to this total overhead. Figure 3c reveals an interesting property: Proteus actually reduces storage for long PII types such as URLs. Many developers log complete URLs with query parameters (which often contain sensitive tokens) as single strings. Because Proteus encrypts each URL as a fixed-size token, it reduces the per-occurrence overhead for URLs by 2.8 bytes on average compared to the original plaintext.

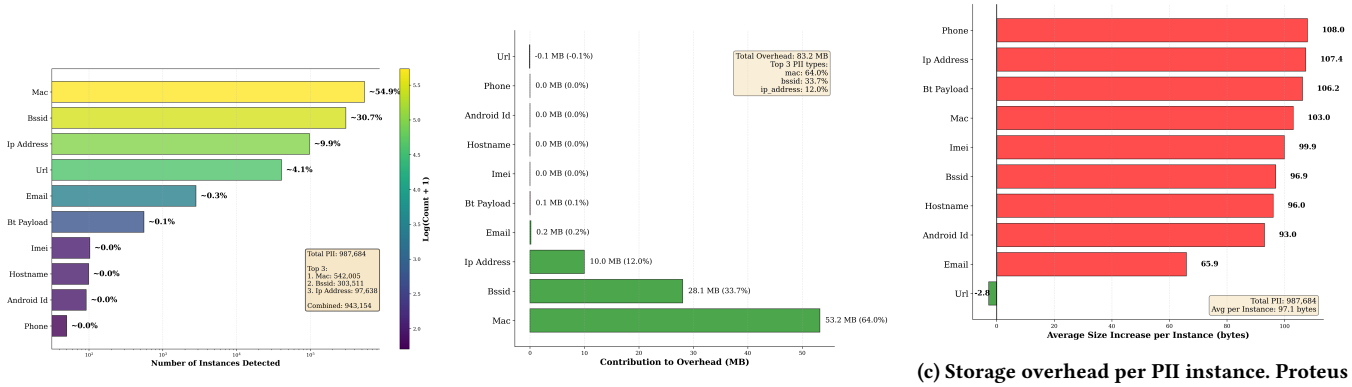
Bulk Processing Latency (RQ4, RQ5). We evaluate the overall latency of Proteus's pipeline for bulk processing of the 30.3 million log entries in the dataset, addressing real-world applicability at scale (RQ5). Figure 4 breaks down the total time consumed by each component on both client and server sides. We categorize operations into four stages: (i) PII Detection, (ii) Date Extraction, (iii) Encryption/Decryption, and (iv) Key Derivation. The Date Extraction module parses timestamps from each log line to determine the appropriate daily key K_t for encryption or decryption. PII Detection uses regex-based pattern matching to identify sensitive fields, and Encryption/Decryption follows the two-layer protection scheme described in §5.1.

Client-side PII Detection step dominates the time budget (RQ4) because the client must apply all PII detection patterns (10 regex searches per log entry, corresponding to the PII types in Figure 3a) to identify sensitive fields. In contrast, the server only needs to locate pre-tagged `<PII>...</PII>` markers, which is significantly faster. Similarly, client-side encryption is slower than server-side decryption because of the additional overhead of in-place token replacement: after detecting PII, the client must locate and replace each plaintext occurrence with its encrypted token. When multiple PII fields appear in a single log line, this search-and-replace operation compounds. Despite these asymmetries, the total processing time remains practical for offline batch analysis of large log corpora.

6.3 Micro Evaluation

We deployed Proteus as an Android library integrated with a sample application to conduct micro-level performance evaluation. We instrumented a stress-testing harness to exercise the logging pipeline under varying workloads, PII densities, and device capabilities. Figure 5 presents the results across all three test devices.

Latency and Throughput (RQ3). Figure 5a and Figure 5b present per-message latency and sustained throughput as a function of log rate, thereby assessing performance scalability across hardware generations. Proteus incurs higher latency and lower throughput than the native Android `Log()` API, as expected, given that our implementation operates entirely in user space and performs additional cryptographic operations. Notably, latency decreases with increasing message rate, indicating that the overhead is likely dominated by context switching and garbage collection rather than cryptographic operations themselves. This suggests that a kernel-space implementation would significantly reduce per-message overhead.



(a) Distribution of PII occurrences in the dataset, as identified using standard Regex patterns.

(b) Relative contribution of each PII type introduced by Proteus processing over the entire dataset.

(c) Storage overhead per PII instance. Proteus reduces size for long PII types (*e.g.,* URLs with query parameters) due to fixed-size encrypted tokens.

Figure 3: PII analysis and storage overhead measurements using the end-to-end Proteus prototype.

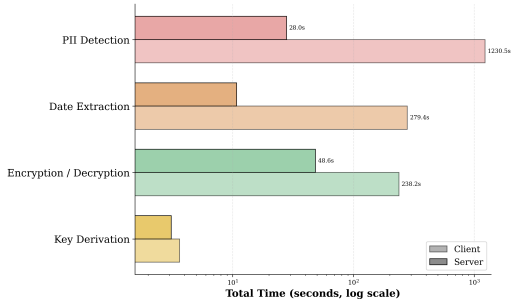


Figure 4: Latency budget observed by Proteus client and server for each component during bulk processing of logs.

Component-Level Breakdown (RQ4). Figure 5c shows the latency breakdown by component across all devices under varying PII densities (low, medium, high), identifying processing bottlenecks. Figure 5d presents a waterfall view of the same breakdown on the Pixel 6a, the most performant device in our testbed. Two components dominate the latency budget: `keyDerivation` and `formatProcessing`. Key derivation computes the daily encryption key K_t from the current date and chain key ck_t for each logging operation. Format processing performs PII detection and in-place token replacement, which compounds when multiple PII fields appear in a single log line. Consistent with the bulk processing results in Figure 4, PII detection and formatting consume the majority of processing time due to regex-based pattern matching and string manipulation. Optimization efforts should focus on these two stages: key derivation overhead could be amortized by caching daily keys or by implementing Proteus within AOSP to perform key derivation once per day in kernel space.

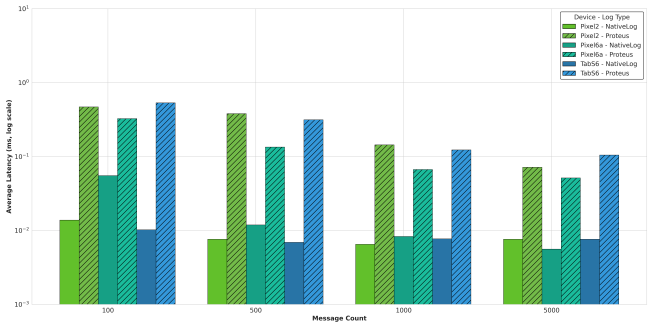
Summary (RQ1, RQ5). Across our comprehensive evaluation suite, Proteus demonstrates practical performance suitable for production logging systems. We establish feasibility by showing that Proteus operates well within acceptable bounds for production systems: sub-millisecond per-message latency (0.2 ms median) and low single-digit storage overhead (2.41%). These metrics are practical because they preserve the responsiveness of application logging (logging operations remain non-blocking) while incurring minimal storage costs that scale linearly with the number of PII occurrences, rather than with the total log volume (RQ1). The end-to-end prototype processed 30.3 million log entries, with bulk processing latency dominated by regex-based PII detection rather than cryptographic operations, demonstrating that the protection mechanisms are efficient. These results confirm that Proteus can handle realistic log workloads at scale (RQ5), processing millions of log entries with acceptable latency for both client and server operations. While our user-space implementation incurs overhead from context switching and garbage collection, the results demonstrate that Proteus's two-layer protection scheme (*i.e.,* pseudonymization plus encryption with daily key ratcheting) operates within performant logging system boundaries. A kernel-space implementation would further reduce overhead by amortizing key derivation and eliminating user-space context switches, making Proteus suitable for deployment in production logging frameworks. Security guarantees (RQ6) are formally established in §7.

7 Security Analysis

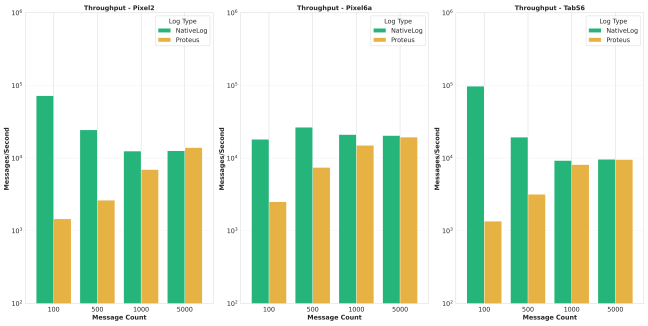
Having presented Proteus's design (§5) and empirical performance (§6), we now analyze its security properties, addressing RQ6. We establish that Proteus's hierarchical ratcheting construction provides confidentiality and forward secrecy by comparing its properties to Signal's messaging protocol, which has been formally analyzed under multi-snapshot adversaries. This section formalizes the security guarantees, identifies the cryptographic assumptions, and discusses limitations.

7.1 Security Objectives

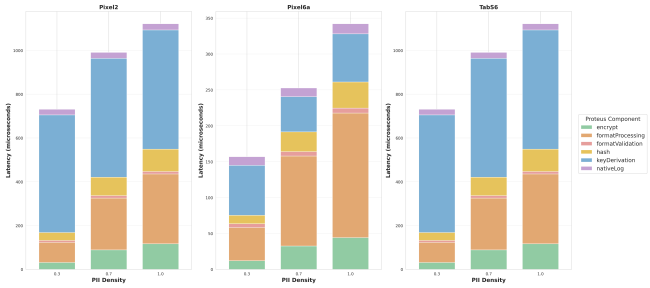
We analyze Proteus under the threat model defined in §4: (i) a multi-snapshot on-device observer who captures repeated log dumps and internal client state, and (ii) an honest-but-curious server that receives exported logs and cryptographic grants. Building on the design in §5, we establish three security objectives: (1) *confidentiality of PII at rest and in transit*, ensuring that plaintext PII is never exposed in logs; (2) *forward secrecy across time windows and access grants*, protecting past and future logs from compromise of current



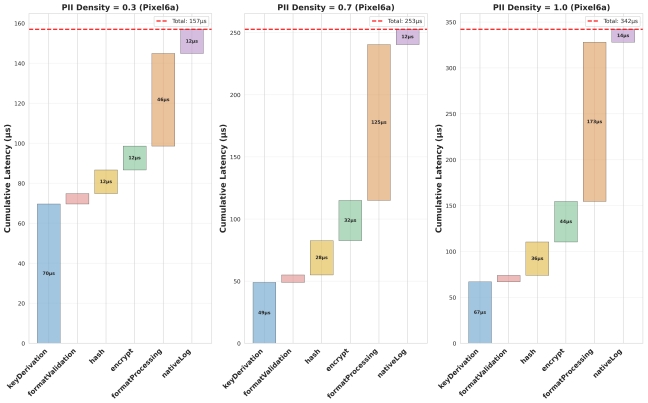
(a) Per-message latency decreases with increasing message rate for Proteus, while native Log latency stays constant.



(b) Observed throughput across devices under sustained workload shows that Proteus can handle high message rates with equal throughput to native Log.



(c) Component latency breakdowns across devices show that format-Processing (Regex) and KeyDerivation (Proteus) dominate the latency budget.



(d) Varying the PII density per log line reveals FormatProcessing (Regex) as the actual bottleneck. Detection is supplementary to Proteus(§8).

Figure 5: Comprehensive on-device performance evaluation of Proteus client.

state; and (3) *integrity and provenance of exported material*, binding logs to attested device identity.

The analysis assumes standard security notions for the underlying primitives: KDF is a pseudorandom function family, AE provides IND-CCA [4] confidentiality and integrity, and HMAC is a keyed-pseudorandom function. We further assume that DICE attestation produces CDIs that are computationally indistinguishable from random, given device measurements [51, 53].

7.2 Comparison with Signal's Double Ratchet

The design in §5 employs a hierarchical ratcheting construction. We now argue that it provides security guarantees analogous to Signal's well-analyzed double ratchet protocol [1, 5, 9, 10, 42]. Proteus uses two nested ratchets: an *outer ratchet* realized by the DICE-derived root key R_0, and an *inner ratchet* realized by the daily chain key ck_t. The outer ratchet is explicitly rotated when an access grant is fulfilled (Algorithm 4, lines 13–15), and implicitly rotated whenever DICE re-measures the device firmware or boot state (since the CDI, and thus R_0, changes with different measurements). The inner ratchet advances daily via one-way key derivation (Algorithm 2).

The key difference from Signal lies in the rotation triggers. Signal rotates its outer (Diffie–Hellman) ratchet whenever a party receives a fresh DH public key from its peer, starting a new sending/receiving chain with each message exchange. In contrast, Proteus rotates its outer ratchet only on hardware-attested events (log export grants and firmware changes) and advances the inner ratchet on a deterministic daily schedule. A second structural difference lies in how the chain key is re-initialized after outer ratchet rotation. In Signal, a single KDF call over (R_0, z) simultaneously produces both the new root key R'_0 and the new chain key ck'_0, making them independent outputs of the same input. In Proteus, the rotation proceeds in two steps: first $R'_0 \leftarrow \mathrm{KDF}(R_0, z, \text{“root-key-rotate”}, 32)$, then $ck'_0 \leftarrow \mathrm{KDF}(R'_0, \text{“ck-init”} \,\|\, (d{+}1), 32)$ (Algorithm 4, lines 13–15). This creates a hierarchical dependency: ck'_0 is derived *from* R'_0, not independently of it. The security implications of this distinction are discussed in the forward secrecy argument below. We argue that each epoch in Proteus is functionally equivalent to a single (long) sending epoch in the double ratchet algorithm that omits the immediate decryption property. As noted in [5] (Appendix A.2.6), immediate decryption is a fundamental property for the practical usability of a secure messaging protocol and is required in their security model, but it is not necessary to achieve forward secrecy

or post-compromise security. Proteus thusly inherits the expected security properties analyzed in [1, 5, 10].

Security Argument via Structural Correspondence. The structural correspondence described above allows us to argue that Proteus inherits the security properties of Signal's double ratchet. The following discussion traces how each property carries over through the precise mapping between the two constructions.

Forward secrecy : The inner ratchet's one-way key derivation (Algorithm 2) ensures that, given the current chain key ck_t, no prior chain key $ck_{t'}$ for $t' < t$ can be computed, under the PRF assumption for KDF. Consequently, the daily message keys $mk_{t'}$ for $t' < t$ are computationally inaccessible to an adversary who compromises state at day t. This provides forward secrecy against an adversary who compromises only the inner chain state (ck_t, mk_t) at day t—a weaker guarantee than Signal's, where forward secrecy holds even under full state compromise. The difference stems from the structural distinction noted above: since ck'_0 is derived from R'_0 rather than independently, an adversary who also obtains R_0 could re-derive all future chain keys. Signal avoids this because its single-step KDF produces R'_0 and ck'_0 independently from the same (R_0, z) input. In Proteus, this assumption is justified by the hardware binding of R_0 to the DICE-derived CDI, which is protected by the device TEE; extracting R_0 requires physical compromise of the hardware root of trust, which lies outside the threat model (§4). Proteus's forward secrecy is therefore the product of two complementary mechanisms: (i) the one-way chaining of the inner ratchet protects past logs against chain-key compromise, and (ii) the TEE-backed protection of R_0 ensures that an adversary cannot re-enter the key hierarchy from the root.

Post-compromise security : Post-compromise security in Proteus addresses a distinct compromise scenario from forward secrecy. Whereas forward secrecy models the exposure of inner chain state (ck_t, mk_t) with R_0 remaining intact in the TEE, post-compromise security addresses the case where chain key state ck_{t^*} has been intentionally disclosed via the controlled sharing protocol—an authorized release that permits the server to derive daily keys for the interval $[t^*, d]$. Unlike Signal, which analyzes both properties against the same uniform full-state compromise, Proteus separates the two: forward secrecy is enforced by the inner ratchet under TEE-backed root key protection, while post-compromise security is enforced by the outer ratchet, which injects fresh DH entropy after each grant to sever the cryptographic link between the shared chain and all subsequent keys.

After each access grant, the client re-derives its root key as $R'_0 \leftarrow \mathsf{KDF}(R_0, z, \text{“root-key-rotate”})$, where $z = \mathsf{ECDH}(a', B')$ is the shared secret from a freshly generated ephemeral DH keypair (Algorithm 4, lines 13–15). An adversary who has compromised the current root key R_0 cannot predict R'_0 without knowledge of the ephemeral private key a', which is generated after the compromise event. This is structurally identical to the outer DH ratchet step in Signal, which injects fresh DH entropy to sever the cryptographic link to the prior compromised state [1, 42]. Future chain keys and message keys derived from R'_0 are therefore independent of any state obtained by the adversary prior to the grant.

Epoch independence : Each daily epoch in Proteus corresponds to a single sending epoch in Signal's double ratchet under a relaxed rotation schedule. Within each epoch, all PII ciphertexts are produced under the same daily message key mk_t, derived from the chain key ck_t via KDF_CK. Chain keys across epochs are linked only by the forward-only one-way derivation, so compromise of ck_t reveals nothing other than $ck_{t+1}, ck_{t+2}, \ldots$ until the outer ratchet fires at the next grant event. The Decisional Diffie–Hellman (DDH) assumption over Curve25519 is required to bound the adversary's advantage in computing the ephemeral shared secret z, precisely as in the analyses of Signal [1, 5, 10].

In summary, Proteus's hierarchical ratchet is a valid instantiation of the double ratchet construction whose outer rotation is triggered by hardware-attested grant events (rather than peer message exchanges) and whose inner rotation is daily (rather than per-message). Both ratchets inject fresh entropy into the key hierarchy via the same mechanisms as Signal—ephemeral DH for the outer ratchet and one-way KDF for the inner ratchet—and the security argument for forward secrecy and post-compromise security follows the same structure as the analyses in [1, 5, 9, 10]. The coarser rotation schedule means exposure windows are longer (at most one day for the inner ratchet), but the post-grant root rotation in Algorithm 4 ensures that any granted decryption window has cryptographically independent keys from future logs, satisfying the security objectives in §4.

7.3 Security Guarantees

Confidentiality of PII. Crucially, Proteus's two-layer design provides confidentiality even against the honest-but-curious server. The server never receives K_{hash} or the device CDI (only the ratchet state used to derive the decryption keys mk_t). Therefore, decryption of $c^\star$ reveals only the pseudonymized token defined as $\text{token}^\star = \mathsf{HMAC}(K_{\text{hash}}, p_b)$, which is pseudorandom under the PRF assumption for HMAC. Both the on-device adversary (who sees only encrypted ciphertexts without K_t) and the honest-but-curious server (who obtains K_t but not K_{hash}) observe only pseudorandom values. The only leakage is token equality (*i.e.,* identical PII values produce identical tokens), which is necessary for forensic correlation and explicitly permitted by design goal G3.

Forward Secrecy and Break-In Recovery. Forward secrecy follows directly from the ratchet construction. Consider an adversary who compromises the client at day t and obtains the current state (ck_t, K_t). Due to the one-way property of the inner ratchet (Algorithm 2), recovering any prior chain key $ck_{t'}$ for $t' < t$ is computationally infeasible under the PRF assumption for KDF. Thus, logs encrypted with keys $K_{t'}$ for $t' < t$ remain confidential even after state compromise at day t.

For post-compromise security (break-in recovery), Proteus employs the hierarchical ratcheting mechanism described in §7. After each access grant, the client rotates the outer ratchet by re-deriving a fresh root key R'_0 from the current R_0 and the shared secret z from the ephemeral DH exchange for that grant. This ensures that an adversary who has compromised the device state cannot predict future root keys, as they do not know the ephemeral private keys used in subsequent sharing protocols. This provides strong break-in recovery.

Integrity and Attestation Guarantees. The integrity of exported material follows from three mechanisms. First, the controlled sharing protocol (Algorithm 4) includes comprehensive context in the

AEAD associated data: server identifier, device identifier, attestation digest, grant identifier, and grant date (lines 7–9). This binds the grant to the specific attested device state, server, and time window. Any tampering with the grant or its context is detected by AEAD verification (line 19). Second, since all symmetric keys are derived from the hardware-rooted CDI via KDF, a server successfully decrypting a grant verifies that the client possesses the correct CDI and executed on a measured state authorized by the attestation policy. Third, each protected PII field carries an AEAD tag (Algorithm 3) that cannot be forged without knowledge of the daily key mk_t. Consequently, any alteration to protected log fields is detected during token recovery (Algorithm 5), and a malicious party cannot inject fabricated PII tokens that appear authentic.

Limitations. The analysis inherits the standard assumptions of the underlying cryptographic primitives [9]. We identify three limitations. First, if an adversary compromises K_{hash} (which remains on-device for the lifetime of the CDI), they can compute $\mathsf{HMAC}(K_{\text{hash}}, \cdot)$ for candidate PII values and link tokens to plaintext through offline dictionary attacks. This is mitigated by the hardware binding of K_{hash} to the CDI, requiring physical device access and extraction. Second, Proteus's coarser rotation granularity (daily for inner ratchet, per-grant for outer ratchet) creates longer exposure windows than Signal's per-message rotation. A compromise at day t exposes all logs encrypted with K_t on that day. However, this granularity is appropriate for logging workloads, where sub-second key rotation would impose prohibitive overhead. Third, metadata leakage, including timestamps, log categories, message sizes, and ciphertext lengths, remains observable to all parties and can reveal behavioral patterns. As stated in §4, metadata privacy and traffic analysis resistance are explicitly out of scope for Proteus.

8 Discussion

PII Detection Strategy. Our prototype implements regex-based PII detection as a stand-in for other production-grade inference-based identification mechanisms [32, 35, 46]. While this is sufficient to evaluate the performance of our cryptographic pipeline, we do not claim it is a complete solution to the PII detection problem. Indeed, our work highlights that the most secure and performant path forward is to shift responsibility from brittle, post hoc scanning to the developer. We advocate for the adoption of privacy-aware logging APIs, where sensitive data is explicitly tagged at the source (*e.g.,* `Log.safe("Login for %s", pii(user_email))`). Proteus provides the ideal backend architecture to support such an API, guaranteeing protection once a field is marked, thereby solving the enforcement problem while leaving the (orthogonal) identification problem to a more appropriate layer.

DICE Attestation Practicality. Our evaluation focuses on the performance of the Proteus cryptographic pipeline, which operates on keys *provided by* an attestation framework. By simulating this interface, we accurately benchmark the overhead of our contribution. While the overall end-to-end security depends on the hardware's correct implementation of DICE, our work demonstrates that the software layer built on it is practical and efficient.

Server-Side Scalability. A potential concern is the complexity of managing decryption keys and ratchet states for thousands or millions of devices on the server side. However, the threat model for forensic analysis clarifies the scope of this challenge. PII token decryption is not intended for bulk, real-time analytics across all devices. Instead, it is a targeted forensic tool used on a small handful of devices that have already been identified as compromised or suspicious through other means. The initial stages of an investigation, such as timeline reconstruction and provenance analysis, can proceed on the encrypted logs without requiring PII correlation. The resource-intensive decryption and correlation process is only invoked selectively, ensuring the server-side workload remains manageable.

9 Related Work

Tamper-Evident and Secure Logging. A significant body of work focuses on log integrity, ensuring logs cannot be altered post-facto. These systems provide tamper-evidence through techniques like cryptographic chaining [47], Trusted Execution Environments (TEEs) [27, 41], or eBPF [59]. While essential for establishing log immutability, they protect the container rather than its contents. Proteus provides a complementary guarantee of log *provenance* via DICE attestation, binding logs to a device's state at creation. By protecting both the content and its origin, Proteus offers an integrated solution for provenance, integrity, and confidentiality.

Log Reduction. A parallel line of research focuses on log reduction and compression to manage high data volumes [22, 29, 49, 50]. These systems use techniques like heuristic pruning and in-kernel filtering to reduce storage overhead. However, this provides data minimization, not content protection; any remaining log entries still contain plaintext PII. Proteus's in-situ protection is therefore complementary, ensuring confidentiality even within a reduced log stream.

Privacy-Preserving Data Collection. Prior work on privacy preserving data collection fails to reconcile utility and confidentiality. Post-hoc redaction [35] exposes PII during collection and transit. Client-side taint analysis [2, 13, 19, 56] incurs high overhead and has coverage gaps. Differential privacy [14, 34, 44, 54] destroys the event-level fidelity required for forensics. Even selective decryption of encrypted logs [7] still exposes plaintext PII. Proteus resolves this trade-off by separating correlation from disclosure. It enables linkage analysis on pseudonymized tokens without ever exposing the sensitive plaintext values, providing an essential privacy layer for modern forensic systems [8, 25, 43, 48, 58].

10 Conclusion

This paper presents Proteus, the first in-situ logging framework that prevents PII disclosure while preserving full forensic utility for mobile endpoints. By exploiting the insight that forensic analysis requires correlation rather than the recovery of PII, Proteus addresses longstanding challenges in privacy-preserving logging: protecting data at emission, defending against multi-snapshot adversaries, and enabling controlled server-side correlation without exposing plaintext. Our hardware-rooted double-ratchet construction with DICE-anchored key derivation provides forward secrecy, while the controlled-sharing protocol enables time-windowed forensic access to pseudonymized tokens. Evaluation across three Android device generations demonstrates practical overhead (median 0.2 ms latency, 2.41% storage) suitable for production deployment,

showing that privacy-preserving logging is achievable even under continuous exfiltration to honest-but-curious cloud platforms.

Ethics Considerations

Our work relies on the publicly available LogHub dataset of Android logs [60] and does not involve human subjects. We did not collect, process, or release any PII from actual users. All experiments on physical devices were performed in a controlled environment for performance benchmarking purposes only, ensuring no risk of data exposure or privacy violations to third parties.

Generative AI Usage

We used publicly available GenAI tools such as ChatGPT, Gemini, and Claude to assist in code prototyping and to refine the manuscript's prose. All AI-assisted code was manually verified via compilation and testing. The authors authored the original draft of this paper; LLMs were used only for grammatical and quality improvements. Every AI-generated suggestion was reviewed by the authors to ensure technical accuracy and the absence of hallucinations.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback that significantly improved the presentation and scope of this work. We thank Hayawardh Vijayakumar, Lee Harrison, and Luke Deshotels from Samsung Research America for helpful discussions and suggestions during various stages of this work.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 129–158.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices* 49, 6 (2014), 259–269.

[3] Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Francesco Silvestri. 2018. Distance-sensitive hashing. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 89–104.

[4] Mihir Bellare, Anupam Desai, David Pointcheval, and Phillip Rogaway. 1998. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Advances in Cryptology – CRYPTO '98*. Springer, 26–45. https://doi.org/10.1007/BFb0055346

[5] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. 2022. A more complete analysis of the signal double ratchet algorithm. In *Annual International Cryptology Conference*. Springer, 784–813.

[6] California Legislature. 2018. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa Accessed 2025-10-30.

[7] Basanta Chaulagain and Kyu Hyung Lee. 2024. FA-SEAL: Forensically Analyzable Symmetric Encryption for Audit Logs. In *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 716–732.

[8] Zijun Cheng, Qiujian Lv, Jinyuan Liang, Yan Wang, Degang Sun, Thomas Pasquier, and Xueyuan Han. 2024. Kairos: Practical Intrusion Detection and Investigation Using Whole-System Provenance. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3533–3551.

[9] Katriel Cohn-Gordon, Cas Cremers, Luke Dowling, Benjamin Garratt, and Douglas Stebila. 2017. A Formal Security Analysis of the Signal Messaging Protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 451–466. https://doi.org/10.1109/EuroSP.2017.27

[10] Daniel Collins, Doreen Riepel, and Si An Oliver Tran. 2024. On the Tight Security of the Double Ratchet. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. 4747–4761.

[11] IBM Corporation. 2025. IBM MaaS360 Unified Endpoint Management. https://www.ibm.com/cloud/maas360. Accessed 2025-11-13.

[12] Microsoft Corporation. 2025. Microsoft Intune: Managing Devices and Apps in the Cloud. https://learn.microsoft.com/en-us/intune/intune-service/fundamentals/what-is-intune. Accessed 2025-11-13.

[13] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1054–1067.

[15] Soheil Esmaeilzadeh, Negin Salajegheh, Amir Ziai, and Jeff Boote. 2022. Abuse and Fraud Detection in Streaming Services Using Heuristic-Aware Machine Learning. *arXiv preprint arXiv:2203.02124* (2022).

[16] European Union. 2016. Regulation (EU) 2016/679 (General Data Protection Regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj Accessed 2025-10-30.

[17] Google LLC. 2025. Verify Hardware-Backed Key Pairs with Key Attestation. https://developer.android.com/privacy-and-security/security-key-attestation. Accessed 2025-11-12.

[18] Digital Guardian. n.d.. The Ultimate Guide to BYOD Security: Overcoming Challenges, Creating Effective Policies and Mitigating Risk. https://www.digitalguardian.com/blog/ultimate-guide-byod-security-overcoming-challenges-creating-effective-policies-and-mitigating Last accessed: August 29, 2025.

[19] Guangwu Hu, Bin Zhang, Xi Xiao, Weizhe Zhang, Long Liao, Ying Zhou, and Xia Yan. 2021. SAMLDroid: a static taint analysis and machine learning combined high-accuracy method for identifying android apps with location privacy leakage risks. *Entropy* 23, 11 (2021), 1489.

[20] Daniel Hugenroth, Alberto Sonnino, Sam Cutler, and Alastair R Beresford. 2024. Sloth: Key Stretching and Deniable Encryption using Secure Elements on Smartphones. *Proceedings on Privacy Enhancing Technologies* (2024).

[21] Muhammad Adil Inam, Yinfang Chen, Akul Goyal, Jason Liu, Jaron Mink, Noor Michael, Sneha Gaur, Adam Bates, and Wajih Ul Hassan. 2023. SoK: History is a Vast Early Warning System: Auditing the Provenance of System Intrusions. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2620–2638.

[22] Muhammad Azeem Inam, Akshay Goyal, Junjie Liu, James Mink, Nathan Michael, Siddhant Gaur, Adam Bates, and Wajih Ul Hassan. 2022. Faust: Striking a Bargain Between Forensic Auditing's Security and Throughput. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*. ACM, 813–826. https://doi.org/10.1145/3564625.3568005

[23] Citrix Systems Inc. 2025. Citrix Endpoint Management. https://www.citrix.com/products/citrix-endpoint-management.html. Accessed 2025-11-13.

[24] VMware Inc. 2025. VMWare Workspace ONE Unified Endpoint Management. https://www.vmware.com/products/workspace-one.html. Accessed 2025-11-13.

[25] Zian Jia, Yun Xiong, Yuhong Nan, Yao Zhang, Jinjing Zhao, and Mi Wen. 2024. MAGIC: Detecting Advanced Persistent Threats via Masked Graph Representation Learning. In *33rd USENIX Security Symposium (USENIX Security 24)*. 5197–5214.

[26] S. Karagiannis, L. L. Ribeiro, C. Ntantogian, E. Magkos, and L. M. Campos. 2023. Chidroid: A Mobile Android Application for Log Collection and Security Analysis in Healthcare and IoMT. *Applied Sciences* 13, 5 (2023), 3061.

[27] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing System Logs with SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 19–30.

[28] Karen Kent and Murugiah Souppaya. 2006. *Guide to Computer Security Log Management*. Technical Report NIST Special Publication 800-92. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-92

[29] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: Garbage Collecting Audit Log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*. ACM, 1005–1016. https://doi.org/10.1145/2508859.2516731

[30] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a Timely Causality Analysis for Enterprise Security. In *Network and Distributed System Security Symposium (NDSS)*.

[31] Allan Lyons, Julien Gamba, Austin Shawaga, Joel Reardon, Juan Tapiador, Serge Egelman, and Narseo Vallina-Rodríguez. 2023. Log: It's Big, It's Heavy, It's Filled with Personal Data! Measuring the Logging of Sensitive Information in the Android Ecosystem. In *32nd USENIX Security Symposium (USENIX Security '23)*. 2115–2132.

[32] Luca Mainetti and Andrea Elia. 2025. Detecting Personally Identifiable Information Through Natural Language Processing: A Step Forward. *Applied System Innovation* 8, 2 (2025), 55.

[33] ManageEngine (Zoho Corp.). 2025. Different Types of Logs in SIEM and Their Log Formats. https://www.manageengine.com/log-management/siem/collecting-and-analysing-different-log-types.html. Accessed 2025-11-13.

[34] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. 2019. Privacy-preserving process mining: Differential privacy for event logs. *Business & Information Systems Engineering* 61, 5 (2019), 595–614.
[35] Microsoft Corporation. 2025. Microsoft Presidio: Data Protection and Anonymization SDK. https://microsoft.github.io/presidio/ Last accessed: October 30, 2025.
[36] Microsoft Corporation. 2025. What Is Data Loss Prevention (DLP)? https://www.microsoft.com/en-us/security/business/security-101/what-is-data-loss-prevention-dlp. Accessed 2025-11-13.
[37] MITRE Corporation. 2025. CWE-359: Exposure of Private Personal Information to an Unauthorized Actor. https://cwe.mitre.org/data/definitions/359.html. Accessed 2025-11-10.
[38] NanoReview. 2025. Google Tensor (G1) – Geekbench 6: Single-Core 1317 / Multi-Core 3208. https://nanoreview.net/en/soc/google-tensor. Accessed 2025-11-11.
[39] NanoReview. 2025. Qualcomm Snapdragon 835 – GeekBench 6 Scores: Single-Core 415 / Multi-Core 1508. https://nanoreview.net/en/soc/qualcomm-snapdragon-835. Accessed 2025-11-11.
[40] NanoReview. 2025. Qualcomm Snapdragon 855 – GeekBench 6 Scores: Single-Core 930 / Multi-Core 2834. https://nanoreview.net/en/soc/qualcomm-snapdragon-855. Accessed 2025-11-11.
[41] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical Tamper-Evident Auditing of Operating Systems Using Trusted Execution. In *Network and Distributed System Security Symposium (NDSS)*.
[42] Trevor Perrin and Moxie Marlinspike. 2016. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/. Accessed 2025-11-10.
[43] Mati Ur Rehman, Hadi Ahmadi, and Wajih Ul Hassan. 2024. Flash: A Comprehensive Approach to Intrusion Detection via Provenance Graph Representation Learning. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3552–3570.
[44] Xuebin Ren, Liang Shi, Weiren Yu, Shusen Yang, Cong Zhao, and Zongben Xu. 2022. LDP-IDS: Local differential privacy for infinite data streams. In *Proceedings of the 2022 international conference on management of data*. 1064–1077.
[45] Samsung Electronics. 2023. Samsung and Microsoft unveil first on-device attestation solution for enterprise. https://news.samsung.com/global/samsung-and-microsoft-unveil-first-on-device-attestation-solution-for-enterprise Last accessed: August 29, 2025.
[46] Maksim Savkin, Timur Ionov, and Vasily Konovalov. 2025. SPY: Enhancing Privacy with Synthetic PII Detection Dataset. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 4: Student Research Workshop)*. 236–246.
[47] Bruce Schneier and John Kelsey. 1999. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)* 2, 2 (1999), 159–176.
[48] R Sekar, Hanke Kimm, and Rohit Aich. 2024. eAudit: A Fast, Scalable and Deployable Audit Data Collection System. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3571–3589.
[49] Hongbin Sun, Su Wang, Zhiliang Wang, Zheyu Jiang, Dongqi Han, and Jiahai Yang. 2024. 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 (RAID)*. ACM, 263–277. https://doi.org/10.1145/3678890.3679048
[50] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. 2018. NodeMerge: Template-Based Efficient Data Reduction for Big-Data Causality Analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer & Communications Security (CCS)*. ACM, 1324–1337. https://doi.org/10.1145/3243734.3243763
[51] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V Thakur. 2021. {DICE*}: A formally verified implementation of {DICE} measured boot. In *30th USENIX Security Symposium (USENIX Security 21)*. 1091–1107.
[52] TechTarget. n.d.. What is Unified Endpoint Management (UEM)? https://www.techtarget.com/searchenterprisedesktop/definition/unified-endpoint-management-UEM Last accessed: August 29, 2025.
[53] Trusted Computing Group. 2021. *Device Identifier Composition Engine (DICE) Attestation Architecture*. Technical Report. Trusted Computing Group. https://trustedcomputinggroup.org/resource/dice-attestation-architecture/ Accessed 2025-11-07.
[54] Gatha Varma, Ritu Chauhan, and Dhananjay Singh. 2022. Sarve: synthetic data and local differential privacy for private frequency estimation. *Cybersecurity* 5, 1 (2022), 26.
[55] Zhiqiang Xu, Pengcheng Fang, Changlin Liu, Xusheng Xiao, Yu Wen, and Dan Meng. 2022. DepComm: Graph Summarization on System Audit Logs for Attack Investigation. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 540–557.
[56] Zhi Yang, Zhanhui Yuan, Shuyuan Jin, Xingyuan Chen, Lei Sun, Xuehui Du, Wenfa Li, and Hongqi Zhang. 2022. FSAFlow: Lightweight and fast dynamic path tracking and control for privacy protection on Android using hybrid analysis with state-reduction strategy. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2114–2129.
[57] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. 2021. WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics. In *Network and Distributed System Security Symposium (NDSS)*.
[58] Jun Zeng, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. 2022. ShadeWatcher: Recommendation-Guided Cyber Threat Analysis Using System Audit Records. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 489–506.
[59] Rui Zhao, Muhammad Shoaib, Viet Tung Hoang, and Wajih Ul Hassan. 2025. Rethinking Tamper-Evident Logging: A High-Performance, Co-Designed Auditing System. *arXiv preprint arXiv:2509.03821* (2025).
[60] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. 2023. LogHub: A Large Collection of System Log Datasets for AI-Driven Log Analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 355–366.

A Implementation

We implemented Proteus in two configurations to enable comprehensive evaluation of both cryptographic correctness and real-world performance.

End-to-End Prototype. Our first implementation is an end-to-end prototype (client and server) in Python that evaluates the complete cryptographic primitives and logging utility construction. We used the LogHub dataset [60], the largest publicly available collection of system logs at the time of writing, to simulate a realistic log streaming, collection, and analysis pipeline built for Android devices. Figure 2 shows the end-to-end architecture of this prototype.

The client component simulates DICE to derive a CDI and device-bound identity, from which it derives the three long-term keys (R_0, K_{hash}, s_{DH}) as specified in Algorithm 1. The client then issues an Ed25519-signed X.509 certificate embedding its X25519 public key and performs Diffie–Hellman-based key agreement to establish a shared export key K_{exp}. Log protection operates as a streaming pipeline: per-line date extraction, regex-based PII detection, per-day symmetric ratcheting (HKDF) to derive daily encryption keys K_t from the chain key ck_t, HMAC-based pseudonym tokenization using K_{hash}, and AEAD encryption (Fernet) with K_t. The client emits tagged protected lines, records timing and space metrics, and exports time-bounded capabilities (t^*, ck_{t^*}) using the controlled sharing protocol for server-side replay. The server component loads a pinned client certificate, reconstructs the export key K_{exp} via Diffie–Hellman, and replays the symmetric ratchet forward from the provided chain key ck_{t^*} to derive day-specific decryption keys $\{K_t\}$. For each protected log line, the server locates PII tags, derives the correct per-day key from the ratchet, decrypts to recover stable pseudonym tokens, and collects data for analysis.

The purpose of this prototype is to evaluate macro-level measurements using real-world logs: overall size increase, per-PII-type overhead, and end-to-end pipeline performance. These measurements, presented in §6, provide insight into the storage and bandwidth costs of Proteus across diverse log workloads.

Android Logging Library. Our Android implementation focuses on Proteus's client-side features for micro-evaluation of performance on real-world devices. We deployed the prototype across three hardware generations with varying capabilities: Pixel 2 (2017), Pixel 6a (2022), and Samsung Galaxy Tab S6 (2019). To ensure modularity and ease of integration with any Android application, we designed a user-space library `loglib` that can be imported into any app to leverage Proteus's logging framework. Proteus as a logging library introduces a `Log.safe()` API that mimics the functionality of the native `Log()` API in AOSP. `Log.safe()` implements all the

client-side functionality described in §5.1 using standard Android and Java cryptographic imports: DICE-based key derivation to obtain R_0, K_{hash}, and s_{DH}; regex-based PII detection; HMAC-based pseudonym tokenization with K_{hash}; per-day symmetric ratcheting (HKDF) to derive daily encryption keys K_t from chain key ck_t; and AEAD encryption with K_t.

The prototype instruments timing metrics for each pipeline stage (key derivation, pattern scan, hashing, encryption) at nanosecond granularity to enable fine-grained performance analysis. We include a stress-testing harness that exercises throughput, concurrency, latency, memory, and PII-density scenarios, with on-device result collection to app-scoped storage. We rely on conservative implementation patterns, avoiding reflection or native code paths in the core pipeline, to ensure reproducibility across devices.

Due to development constraints—specifically, limitations in modifying AOSP where the native `Log()` API is implemented—our solution is entirely implemented in user space. This design limitation introduces additional overhead in our latency experiments: garbage collection and context switching by user-space applications increase the measured latency compared to the AOSP-native `Log()` API. Despite these constraints, our measurements demonstrate that Proteus imposes practical overhead suitable for production deployment. We discuss the performance results in detail in §6.