Lightweight and High-Throughput Secure Logging for Internet of Things and Cold Cloud Continuum

SAIF E. NOUMA and ATTILA A. YAVUZ, University of South Florida, USA

The growing deployment of resource-limited Internet of Things (IoT) devices and their expanding attack surfaces demand efficient and scalable security mechanisms. System logs are vital for the trust and auditability of IoT, and offloading their maintenance to a Cold Storage-as-a-Service (Cold-STaaS) enhances cost-effectiveness and reliability. However, existing cryptographic logging solutions either burden low-end IoT devices with heavy computation or create verification delays and storage inefficiencies at Cold-STaaS. There is a pressing need for cryptographic primitives that balance security, performance, and scalability across IoT–Cold-STaaS continuum.

In this work, we present *Parallel Optimal Signatures for Secure Logging* (POSLO), a novel digital signature framework that, to our knowledge, is the first to offer constant-size signatures and public keys, near-optimal signing efficiency, and tunable fine-to-coarse-grained verification for log auditing. POSLO achieves these properties through efficient randomness management, flexible aggregation, and multiple algorithmic instantiations. It also introduces a GPU-accelerated batch verification framework that exploits homomorphic signature aggregation to deliver ultra-fast performance. For example, POSLO can verify 2^{31} log entries per second on a mid-range consumer GPU (NVIDIA GTX 3060) while being significantly more compact than state-of-the-art. POSLO also preserves signer-side efficiency, offering substantial battery savings for IoT devices, and is well-suited for the IoT–Cold-STaaS ecosystem.

CCS Concepts: • **Security and Privacy → Cryptography**.

Additional Key Words and Phrases: Authentication, secure logs, cold storage, digital signatures, parallel computing, CUDA.

1 INTRODUCTION

Internet of Things (IoT) refers to a large-scale ecosystem of heterogeneous, Internet-connected devices [38]. Cyber-Physical Systems (CPS) harness IoT devices (e.g., sensors) to monitor the physical environment and construct Digital Twins (DT) for autonomous decision-making and real-time actuation [36]. Despite their proliferation, IoT devices remain intrinsically vulnerable due to stringent constraints in computation, bandwidth, and memory. These limitations impede the deployment of advanced security mechanisms. Moreover, their exposure to open and untrusted networks further amplifies their susceptibility to diverse attacks (e.g., tampering [43]).

System auditing is a critical security measure for early detection of malware and intrusions [1, 30, 53]. It involves maintaining system logs that record security-relevant events (e.g., user activity, errors, breaches), serving as a foundational element for post-incident investigation, attack reconstruction, and forensic analysis. However, modern cyberattacks employ anti-forensics techniques, such as log deletion or manipulation, to hide evidence [37], thereby hindering investigators and system administrators from tracing the origin of security incidents. As a result, the importance of ensuring the trustworthiness of logs is vital for both authorities[1] and practitioners [8, 34, 55].

IoT devices (e.g., smartwatches, pacemakers) often lack the storage capacity to retain log streams locally. Their vulnerabilities (e.g., cyber-physical attacks) further increase the risk of log tampering. A common approach is to securely offload log data to cloud storage for secure archival and forensic analysis [31, 39]. While Storage-as-a-Service (STaaS)[2] offers scalable infrastructure, retaining

[1] https://bidenwhitehouse.archives.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[2] https://www.intel.com/content/www/us/en/cloud-computing/storage-as-a-service.html

Authors' address: Saif E. Nouma, saifeddinenouma@usf.edu; Attila A. Yavuz, attilaayavuz@usf.edu, University of South Florida, 3720 Spectrum Blvd, Interdisciplinary Research Building (IDR)-400, Tampa, Florida, USA, 33612.

append-only log files on cloud servers is prohibitively expensive. A *cold storage solution* [10] is a type of cost-effective data warehouse designed for large-scale archives. Cold-STaaS, therefore, becomes a suitable choice to store rarely accessed yet valuable system logs. Secure offloading requires integrity protection, authentication, and confidentiality to prevent tampering and unauthorized access [52]. In this work, we focus on achieving data authentication and integrity. To this end, an ideal secure log-authentication scheme for IoT-STaaS must, at a minimum, offer the following properties:

1) SCALABILITY, PUBLIC VERIFIABILITY, AND NON-REPUDIATION. *(i)* The cryptographic solution should be scalable to large IoT networks. *(ii)* It should allow any authorized entity to publicly verify and attest the trustworthiness of information (e.g., metadata, logs) when requested by external parties. *(iii)* It should provide the non-repudiation property, in which the signer cannot later deny that they signed the message. This is an is essential feature for digital forensics and legal dispute resolution (e.g., financial, health). These features are usually offered by digital signatures [20, 55].

2) LOGGER EFFICIENCY. Cryptographic mechanisms must efficiently manage the limited resources (e.g., battery, memory) of low-end IoT loggers, which are expected to operate unattended for extended periods. *(i) Efficient Authentication*: Authentication must impose minimal computational overhead to preserve energy. *(ii) Compact Signatures*: Signatures should be compact to reduce transmission and storage overhead. *(iii) Optimized Memory*: A lightweight cryptographic code with minimal memory footprint is useful to extend the life of the device (e.g., 8-bit microcontroller) [42].

3) VERIFICATION EFFICIENCY AND MINIMAL CLOUD STORAGE. Cold-STaaS platforms manage big data that requires efficient verification and cryptographic compression for secure and scalable archival. *(i) Real-time Security Audits*: Regular integrity checks are essential to mitigate undetected data tampering and for early breach detection [1]. *(ii) Regulatory Compliance*: An efficient verification complies with strict integrity requirements (e.g., General Data Protection Regulation (GDPR) [46]).

4) FLEXIBLE VERIFICATION GRANULARITY. There is a performance and precision trade-off for secure log verification. Authenticating the entire log stream with a single authentication tag offers minimal storage and fast batch verification. However, having a single altered log entry (e.g., attack, error) voids the authentication of the entire log stream. Conversely, per-entry signatures enables the highest precision (i.e., the maximum granularity), but with a high storage overhead. Hence, the authentication scheme should permit for both logger and cold storage to adjust the storage granularity and verification precision based on the application requirements [19, 33].

5) CONFIGURABILITY FOR DIFFERENT SECURITY AND PERFORMANCE DEMANDS. An effective authentication primitive must support tunable parameters to align with various security requirements and resource constraints across diverse IoT environments. These tunable parameters include, for example, cryptographic primitives (e.g., standard cryptographic or lighweight non-cryptographic hash function [9]). As such, it enables system designers to tailor latency and memory usage based on the device constraints and adversarial models, enabling practical deployment.

Overall, it is a highly challenging task to devise a digital signature scheme that meets the stringent performance and security requirements of both IoT devices and STaaS simultaneously. The current state-of-the-art techniques prioritize the needs of either the logger or verifier side while omitting performance and security features for the other side. In the following, we outline the research gap in existing secure logging schemes with a focus on digital signature schemes.

1.1 Related Work and Research Gap

We first discuss the most closely related works to our solutions with a focus on digital signature-based secure logging approaches. We then discuss other relevant and complementary works.

Related Work in our Scope: The proposed signature framework, *Parallel Optimal Signatures for Secure Logging* (POSLO), leverages aggregate digital signature schemes, specialized seed management, and GPU-accelerated batch verification. The relevant research is outlined below.

Aggregate Signatures. POSLO follows prominent Aggregate Signature (AS)-based secure logging schemes (e.g., [19, 20, 25, 33, 55]), wherein the logger computes a compact aggregate signature over log entries to enable post hoc attestation. Digital signatures provide data integrity, authentication, public verifiability, and non-repudiation through Public Key Infrastructures (PKI), making them ideal for scalable authentication in IoT and Cold-STaaS.

Conventional digital signatures (e.g., Ed25519 [4]) incur expensive operations (e.g., modular exponentiation, Elliptic Curve (EC) scalar multiplication), which are costly for resource-limited IoTs. Moreover, they lack signature aggregation, resulting in $O(n)$ signature overhead for n log entries, which poses a heavy storage burden on cold storage servers. Finally, most fail to support batch verification, an important property for efficient large-scale log authentication.

AS schemes [5, 54] enable the aggregation of multiple distinct signatures into a single compact tag, with some schemes also supporting batch verification [14]. Therefore, they are instrumental tools for building cryptographic forensic schemes [19, 33, 34, 55]. Condensed-RSA (C-RSA) [54] and BLS [5] are two essential AS schemes but with a costly computation in both signing and verification. BLS requires highly expensive pairings and EC scalar multiplication with a heavy special hash function on the verifier and signer sides, respectively. Conversely, C-RSA relies on expensive modular exponentiation during signing with large key sizes. As demonstrated in our evaluations (see Table 1), these schemes are not suitable for our envisioned IoT-STaaS applications.

Forward-secure and Aggregate Signatures (FAS) [25, 55] offer both key-compromise resiliency and signature aggregation. Despite their merits, most FASs introduce high computational and storage overhead either at the signer and/or verifier side. BAF signatures [55] are tailored for signer-efficient signatures, but at the cost of a linear public key size. Our experiments demonstrate that large keys introduce substantial storage overhead at Cold-STaaS. Moreover, they cannot offer storage at different granularities due to fixed public key sizes. Hence, they are not suitable for cold storage applications.

Recent AS schemes with extended properties for IoTs (e.g., [28, 49, 50]) are either based on BLS [5] or Schnorr [11], inheriting their expensive operations overhead (e.g., pairing, EC scalar multiplication) at the signer. Our empirical analysis confirms that these operations impose high overhead for ultra low-end IoT devices. Moreover, a critical gap in the literature is the lack of performance benchmarks on low-end devices (e.g., 8-bit ATMega2560). In our evaluations, we focus on Ed25519 [4], SchnorrQ [11], and BLS [5] to represent the signer overhead of these signature primitives, with the note that they do not offer a (full) signature aggregation property.

Hardware-Accelerated Signatures. POSLO parallel batch verification follows a prominent line of research [12, 13, 23, 24, 27, 41] that accelerates digital signature primitives by exploiting the massive parallelism and computational throughput of modern GPU architectures. For example, Dong et al. [12] improve the RSA signing and verification throughputs, but RSA remains prohibitively expensive for low-end IoT signers, rendering it impractical for IoT-STaaS. Other works (e.g., [13, 24]) demonstrate high-throughput GPU implementations of conventional ECDSA and NIST post-quantum signature standards, but they focus solely on optimizing independent signature operations without tackling structural efficiencies of mutable AS schemes and batch verification algorithms.

Complementary Related Work: POSLO is a special class of ASs, and therefore does not offer data confidentiality. POSLO can be complemented by privacy services: (i) data encryption on the logger [16], (ii) private auditing on the STaaS side, and (iii) privacy enhancement tools [51].

There is a line of work focusing on Proof of Data Possession (PDP) [3] and Proof of Retrievability (PoR) [2] on the outsourced user data. Some efforts also address privacy-preserving public auditing

[52]. However, these approaches differ from our system model and primary performance objectives. Specifically, PoR/PDP schemes allow IoT devices to offload log files to STaaS providers without generating data signatures or performing authentication checks. Instead, integrity verification is usually initiated by administrators (or STaaS) via interactive audit protocols, whereas AS-based schemes are generally non-interactive. PoR/PDP schemes offer fast audit time that is achieved by Homomorphic Linear Authenticators (HLA) [52]. These enable an external entity to audit the data without retrieving the entire set. However, it comes at the cost of a very high computational overhead on IoT devices since most deployed HLAs (i.e., BLS, RSA) suffer from expensive signing (see Table 1 and Fig. 7). In a different line, Li et al. [29] proposed a public auditing scheme with data sampling.

Herein, *our goal is to achieve optimal signing and small cryptographic payload for IoT devices, while offering high verification efficiency and compact storage at STaaS. By doing so, we permit low-end IoT to actively compute signatures, thereby ensuring public verifiability and non-repudiation.*

1.2 Our Contribution

In this work, *we create Parallel Optimal Signatures for secure Logging (`POSLO`), a novel AS-based secure logging framework. To the best of our knowledge, `POSLO` is the first to achieve simultaneously small constant-size tags and public key sizes while enabling near-optimal signing and high-throughput parallel batch verification at multiple granularities. These features make `POSLO` ideal for IoT-STaaS, where lightweight signing on constrained IoTs and scalable verification on Cold-STaaS are essential.*

1.2.1 Main Idea. Our key observation is that EC-based signatures (e.g., Ed25519 [4], SchnorrQ [11]) provide compact signature sizes and superior signing efficiency compared to RSA [32, 54] and pairing-based schemes [5]. However, they still incur the cost of a full EC scalar multiplication for each signature generation. Seed-based signing methods [40, 55] improve signing efficiency by employing commitment separation and precomputation during key generation. This gain comes at the expense of significant verifier-side storage and high verification costs. For instance, FI-BAF [55] demands 2TB of storage and 264 hours of verification time on commodity hardware for 2^{35} log entries, each 32 bytes in size. As detailed in Section 1.1 and Section 6, existing AS schemes fall short of jointly minimizing signer and verifier overheads, and do not sufficiently address the challenge of efficient verification at scale for large-volume logs.

`POSLO` addresses these bottlenecks through novel design strategies: *(i) Efficient Seed Management*: A tree-based randomness for seed management in constrained signers reduces signer-side seed storage from $O(n)$ to intermediate $O(\log_2 n)$ and final $O(1)$ overhead. *(ii) Flexible Tag Aggregation*: Our scheme can aggregate additive or multiplicative signature components at arbitrary granularity. This allows for signature aggregation either at the signer per epoch or at the verifier. *(iii) Tailored Variants*: We propose two `POSLO` variants: Coarse-grained signer-optimal `POSLO` (`POSLO-C`) optimized for minimal computational, bandwidth, and memory overhead; and Fine-grained public-key `POSLO` (`POSLO-F`) provides fine-grained auditing with a constant-size public key and efficient signing. *(iii) High-Throughput Parallel Batch Verification*: `POSLO` introduces, to the best of our knowledge, the first GPU-accelerated verification (`POSLO.PAVer`), with orders-of-magnitude speedup compared to AS-based secure logging schemes.

1.2.2 Improvements over Preliminary Version. This article is the extended version of initial `OSLO` signatures appeared in [39] (IEEE/ACM IoTDI'23). Our current article makes substantial contributions over initial `OSLO` [39] with new algorithmic approaches and experimental optimizations:

1) Enhanced Seed Management. In `POSLO`, we introduce a new seed management algorithm that replaces `OSLO`'s hash-table-based structure with a stack-based tree structure [34]. This reduces the signing overhead and memory footprint on resource-constrained IoT devices.

Fig. 1. A high-level illustration of POSLO system model

2) New Instantiations (POSLO$^+$ and POSLO^{++}). We instantiate the message processing and enhanced seed manager of POSLO with symmetric/arithmetic primitives [15] beyond cryptographic hash functions: (i) POSLO$^+$: is an AES-based instantiation, optimized for energy efficiency and parallelism on low-end IoT [42] and Cold-STaaS [48]. (ii) POSLO^{++}: combines AES and modular arithmetic to offer superior performance for small inputs but with a reduced security level.

3) Parallel Batch Verification (POSLO.PAVer) and Expanded Performance Analysis. Most notably, we created POSLO.PAVer, the first GPU-accelerated batch verification system for AS schemes, optimized for POSLO scheme. POSLO.PAVer achieves high-throughput verification while maintaining lightweight signing. It leverages POSLO's commitment separation and the additive homomorphism of its signature aggregation to offload hashing operations to GPUs and perform parallel reduction for computing sub-aggregate ephemeral keys. Our extended performance analysis includes a detailed evaluation of signer-side efficiency with reduced energy consumption and verifier-side scalability with several orders of magnitude speedup from GPU-parallel batch processing.

These novel features combine to significantly improve the efficiency of POSLO, surpassing not only its previous version OSLO but also current state-of-the-art secure logging schemes. In Figure 1 and Table 1, we present a high-level architectural and comparative evaluation of POSLO against state-of-the-art cryptographic secure logging schemes (details are discussed in Section 6).

1.2.3 Desirable Properties. The security and efficiency properties of POSLO are as follows:

• **Highly Efficient Verification and Minimal Cold Cryptographic Storage.** Table 1 provides a comparative analysis of verification time and cryptographic storage requirements for a 1TB log dataset (2^{35} entries, 32 bytes each). (i) *Verification Time:* Our most efficient instantiation, the AES-based POSLO$^+$, achieves the lowest verification latency in the Cold-STaaS setting, outperforming C-RSA and BLS by factors of 4.8× and 70×, respectively, on commodity CPUs. Its GPU-accelerated version, POSLO$^+$.PAVer, completes verification of 1TB of log data in just 24.8 seconds, achieving a throughput of 2^{31} log entries per second, and significantly outpaces the GPU-accelerated baseline SchnorrQ by several orders of magnitude. (ii) *Minimal Cold Storage:* POSLO reduces cryptographic storage to only 0.06 KB, representing a maximum compression compared to EC-based schemes such as Ed25519 and FI-BAF, which require several TBs of storage.

• **Granular and Adaptive Verification Architectures.** POSLO supports flexible verification architectures tailored for constrained signers and cold storage environments. (i) *Coarse-Grained Signer-Optimal POSLO−C:* This variant authenticates each log entry and aggregates them into a single epoch-level umbrella signature. POSLO−C produces the most compact signature and enables the most efficient verification, outperforming BLS by several orders of magnitude for an epoch of size $n_1 = 256$. Although it initially requires $O(n_1)$ public key storage at the verifier, it ultimately compresses to a constant-size $O(1)$ public key in Cold-STaaS. (ii) *Fine-Grained Public-Key POSLO−F:* This variant transmits individual signatures and enables on-the-fly aggregation at the verifier (distiller), allowing maximum granularity while maintaining a constant-size public key. (iii) *Configurable Distillation:* POSLO introduces a configurable distillation process, enabling entries to be verified and aggregated

Table 1. Performance of POSLO and its counterparts on embedded IoT and cold storage servers

Scheme	Logger (Signer) IoT Device: AtMega2560 (8-bit)			Edge Cloud (Distiller) Commodity Hardware		Cold Storage Server Commodity Hardware (Desktop)				Dynamic Granularity	Granul. Level (C: Coarse F: Fine)	Initial/Final Public Key
	Signing (sec) (per item)	Crypto. Payload (KB)	Priv Key Size (KB)	Ver time (ms)	Distill & Agg (μs)	Cold Cryptographic Data		Verification time				
						Entire Sig/PK Set (for 2^{35} entries)	One Sig (KB)	AVer (hours)	PAVer (seconds)			
Ed25519 [4]	0.869	8	0.03	91.38	-	1 TB	0.03	3,406.89	-	×	F	$\mathcal{O}(1)$ / $\mathcal{O}(1)$
SchnorrQ [11]	0.323	8	0.03	28.02	-	1 TB	0.03	1,044.66	6h26m	×	F	$\mathcal{O}(1)$ / $\mathcal{O}(1)$
FI-BAF [55]	0.004	0.05	0.10	56.04	0.02	2 TB	0.77	2,089.32	-	×	C	$\mathcal{O}(n)$ / $\mathcal{O}(n)$
C-RSA [54]	35.828	0.25	0.51	1.48	5.27	0.77 KB	0.25	55.18	-	✓	C/F	$\mathcal{O}(1)$ / $\mathcal{O}(1)$
BLS [5]	4.08	0.05	0.03	77.31	0.02	0.1 KB	0.05	2,882.33	-	✓	C/F	$\mathcal{O}(1)$ / $\mathcal{O}(1)$
SOCOSLO [39]	0.005	0.03	0.06	1.27	1.45	0.06 KB	0.05	45.35	-	✓	**C**	$\mathcal{O}(n/n_1)/\mathcal{O}(1)$
FIPOSLO [39]	0.016	8	65.6	28.38	0.37					✓	**F**	$\mathcal{O}(1)/\mathcal{O}(1)$
POSLO-F$^+$	**0.002**	**0.03**	**0.06**	**1.92**	**1.45**	**0.06** KB	**0.05**	**71.58**	**24.8**	✓	**C**	$\mathcal{O}(n/n_1)/\mathcal{O}(1)$
POSLO-F$^+$	**0.014**	**8**	**65.6**	**28.38**	**0.37**					✓	**F**	$\mathcal{O}(1)/\mathcal{O}(1)$

The experiment settings, hardware/software configurations, and cryptographic parameters are given in Section 6. We chose our counterparts to cover the primary signature schemes deployed for secure logging in IoT environments (see Section 6 for selection rationale). The total number of entries and the epoch size are $n = 2^{35}$ and $n_2 = 2^8$, respectively. The input message is of size 32 bytes. The overall data is of size 1TB. At the logger (signer), the signature is measured per epoch, and signing time (in seconds) is given for a single entry. At the distiller, verification time (in ms) is for all entries within an epoch. At the cold storage server, the cryptographic storage is the total size of signatures and public keys needed to verify n entries. The verification time (AVer on x86/64 and PAVer on GTX 3060) is the total runtime for batch verifying n items.

with flexible granularity. Distillation can be performed by an intermediate verifier (e.g., edge cloud) or directly at the cold storage server.

• **Near-Optimal Logging Efficiency.** POSLO schemes are highly efficient for signing operations, making them particularly well suited for secure logging in constrained IoT environments. (i) *POSLO-C:* By eliminating EC scalar multiplications, POSLO-C achieves one to several orders of magnitude faster signing compared to the most compact traditional and aggregate schemes, namely SchnorrQ and BLS. It is 2× faster than FI-BAF and significantly more compact in Cold-STaaS, with a 34× verification speedup at the distiller. (ii) *POSLO-F:* It offers fine-grained auditability and a constant-size public key. It uses a larger private key for precomputation, which can be optionally replaced with EC scalar multiplication to reduce its size.

• **Full-Fledged Implementation.** We implement the POSLO schemes on both a low-end IoT device and commodity hardware, benchmarking them against existing alternatives. Our experiments confirm that the theoretical advantages of POSLO yield practical performance benefits. The implementation is open-sourced for public testing and further adaptation: https://github.com/SaifNOUMA/POSLO

2 PRELIMINARIES

Notations and Algorithmic Definitions. $||$ and $|x|$ denote concatenation and the bit length of variable x, respectively. $x \xleftarrow{\$} \mathcal{S}$ means that the variable x is randomly selected from the finite set $\mathcal{S}$ using a uniform distribution. $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. $\{0,1\}^*$ denotes a set of binary strings of any finite length. For q a prime power, $\mathbb{Z}_q$ denotes a finite field of order q and $\mathbb{Z}_q^*$ denotes $\mathbb{Z}_q \backslash \{0\}$. $\boldsymbol{x} = \{x_i\}_{i=1}^n$ denotes the set of items $(x_1, x_2, \ldots, x_n)$. $\log x$ denotes $\log_2 x$. $H : \{0,1\}^* \rightarrow \{0,1\}^h$ is a cryptographic hash function [35], where h is the output size. $\mathrm{PRF}_{0,1} : \{0,1\}^\kappa \rightarrow \{0,1\}^\kappa$ are two pseudorandom functions. n denotes the maximum number of items to be signed in a given signature scheme. Our schemes operate in epochs in which n_2 items are processed, with a total of n_1 epochs available such that $n = n_1 \cdot n_2$. The variable $\tilde{x}$ denotes an aggregate value. The variable $\tilde{x_i}$ denotes the aggregate value for an epoch i. $m_i^j \in \boldsymbol{m_i}$ means that m_i^j belongs to a vector of n_2 items $\boldsymbol{m_i} = \{m_i^j\}_{j=1}^{n_2}$. $\vec{\boldsymbol{m}} = \{\boldsymbol{m_i}\}_{i \in I}$ denotes a super vector where each $\boldsymbol{m_i}$ contains n_2 items and $\boldsymbol{I}$ are epoch indexes of $\vec{\boldsymbol{m}}$.

Definition 2.1. Matyas-Meyer-Oseas (MMO) [35] consists of a generic single-length cryptographic hash algorithm MMO that generates an ℓ-bit digest given a pre-defined ℓ-bit block cipher (E) and its initialization vector h_0. MMO is defined as follows:

- $h \leftarrow \texttt{MMO}(m)$: Given a k-bit string m, it splits m into $t = \lfloor \frac{k+1}{\ell} \rfloor$ ℓ-bit blocks $\{m_i\}_{i=1}^{t}$ (padding the last block). Then, it computes $h_i = E_{h_{i-1}}(m_i) \oplus m_i, \forall i = 1, \ldots, t$. Finally, it outputs $h \leftarrow h_t$.

Definition 2.2. MDC-2 [35] consists of a generic double-length cryptographic hash algorithm MDC−2 that generates a 2ℓ-bit digest using a pre-defined ℓ-bit block cipher (E) and two initialization vectors $\{h_0, h'_0\}$. MDC−2 is defined as follows:

- $h \leftarrow \texttt{MDC-2}(m)$: Given a k-bit string m, it splits m into $t = \lfloor \frac{k+1}{\ell} \rfloor$ ℓ-bit blocks $\{m_i\}_{i=1}^{t}$ (padding the last block). Then, it computes $h_i = E_{h_{i-1}}(m_i) \oplus m_i$, and $h'_i = E_{h'_{i-1}}(m_i) \oplus x_i, \forall i = 1, \ldots, t$, interprets $h_i \leftarrow h_i^1 \| h_i^2, h'_i \leftarrow h_i^{1\prime} \| h_i^{2\prime}$, and updates $h_i \leftarrow h_i^1 \| h_i^{2\prime}, h'_i \leftarrow h_i^{1\prime} \| h_i^2$. Finally, it outputs $h \leftarrow h_t \| h'_t$.

Definition 2.3. A single-signer multiple-time aggregate signature scheme ASGN consists of four algorithms (Kg, ASig, Agg, AVer) defined as follows:

- $(I, sk, PK) \leftarrow \texttt{ASGN.Kg}(1^\kappa, n)$: Given the security parameter κ and maximum number of generated signatures n, it returns a private/public key pair (sk, PK) with a public system-wide parameter I.
- $\sigma_i \leftarrow \texttt{ASGN.ASig}(sk, m_i)$: Given sk and a message m_i, it returns a signature σ_i as output.
- $\tilde{\sigma} \leftarrow \texttt{ASGN.Agg}(\sigma_1, \ldots, \sigma_\ell)$: Given ℓ signatures $\{\sigma_i\}_{i=1}^{\ell}$, it returns an aggregate signature $\tilde{\sigma}$.
- $b \leftarrow \texttt{ASGN.AVer}(PK, \{m_i\}_{i=1}^{\ell}, \tilde{\sigma})$: Given PK, a vector of messages $\{m_i\}_{i=1}^{\ell}$ and their corresponding aggregated signature $\tilde{\sigma}$, it outputs $b = 1$ if $\tilde{\sigma}$ is valid or $b = 0$ otherwise.

POSLO schemes rely on the intractability of *Discrete Logarithm Problem (DLP)* [35]. Remark that, while our notations are based on DLP, our implementation is based on *Elliptic Curves (EC)* for efficiency, and *the definitions also hold under ECDLP [11].*

Definition 2.4. Let q and p two prime numbers where $p > q$, $\mathbb{G}$ be a cyclic group of order q, α be a generator of $\mathbb{G}$, and DLP attacker $\mathcal{A}$ be an algorithm that returns an integer in $\mathbb{Z}_q$. We consider the following experiment:

Experiment $Expt^{DL}_{\mathbb{G},\alpha}(\mathcal{A})$:

$y \xleftarrow{\$} \mathbb{Z}_q^*, Y \leftarrow \alpha^y \mod p, y' \leftarrow \mathcal{A}(Y)$

If $\alpha^{y'} \mod p = Y$ then return 1, else return 0

The DL-advantage of $\mathcal{A}$ in this experiment is defined as: $Adv^{DL}_{\mathbb{G}}(\mathcal{A}) = Pr[Expt^{DL}_{\mathbb{G},\alpha}(\mathcal{A}) = 1]$.

The DL-advantage of $(\mathbb{G}, \alpha)$ in this experiment is defined as follows: $Adv^{DL}_{G}(t) = \max_{\mathcal{A}}\{Adv^{DL}_{\mathbb{G}}(\mathcal{A})\}$, where the maximum is over all $\mathcal{A}$ having time complexity t.

POSLO−F uses Boyko-Peinado-Venkatesan (BPV) generator [6]. It reduces the computational cost of expensive operations (e.g., EC scalar multiplication) via a pre-computation technique.

Definition 2.5. BPV generator consists of two algorithms (BPV.Offline, BPV.Online):

- $(\Gamma, v, k) \leftarrow \texttt{BPV.Offline}(1^\kappa, p, q, \alpha)$: It chooses BPV parameters (v, k) as the size of the pre-computed table and number of randomly selected elements, respectively. Then, it generates the pre-computed table $\Gamma = \{r_i, R_i\}_{i=1}^{v}$.
- $(r, R) \leftarrow \texttt{BPV.Online}(\Gamma)$: Generate a random set $S \in \{1, \ldots, v\}$ of size $|S| = k$ and compute a one-time commitment pair $(r \leftarrow \sum_{i \in S} r_i \mod q\,,\ R \leftarrow \prod_{i \in S} R_i \bmod p)$.

Parallel Computing on GPU. A Graphical Processing Unit (GPU) is an electronic circuit that aims to accelerate computer computations by leveraging its parallel structure. It consists of a set of Stream Multiprocessors (SMs). Each SM contains a set of relatively constrained cores (e.g., 1320 MHz base clock frequency). The GPU provides different memory types *(i) global memory (≈GBs)*: can be accessed by all threads in SMs. It is an off-chip memory and represents the data transfer medium

between system memory and GPU with high access throughput (≈100 GB/s via PCI-Express link). *(ii) shared memory (≈KBs)*: is an on-chip memory shared among cores in a single SM and provide a much faster memory access compared to (i). *(iii) register file*: is an on-chip memory that resides in each core. It has the fastest memory access to hold the frequently accessed and local data.

CUDA. It is a program computing platform and application programming interface developed by NVIDIA [26]. It allows programmers to define and execute operations on GPUs. A CUDA program launches a *kernel* function that executes multiple *threads* in parallel. The fundamental execution unit is a *warp*, consisting of 32 threads, and multiple warps form a *block*. Blocks are organized into a *grid*, containing all the threads. Before executing the kernel, (i) the programmer must specify the number of both threads and blocks (ii) the input data is copied from system memory to global (device) memory. After kernel execution, the output data is copied from global memory to system memory to resume CPU processes. Note that CUDA follows Single Instruction Multiple Threads (SIMT) model. Running different tasks on threads causes warp divergence, forcing sequential execution.

3 MODELS

3.1 System Model

Our system model follows a well-established AS-based secure logging framework (e.g., [19, 20, 33, 55]), in which the logger (i.e., IoT device) generates authentication tags for its log entries to enable future public verification. We consider an *IoT–Cloud continuum*, where numerous IoT devices produce log streams and report them to an edge or core cloud for processing, verification, and archival. As illustrated in Figure 1, our model consists of three main entities:

(i) Logger (Signer): This component represents resource-constrained end-user IoT devices (e.g., medical sensors) that collect sensitive data such as health or personal information. These devices periodically send the data along with their corresponding log entries to a nearby edge server. They are limited in computational power, memory, energy, and communication bandwidth.

(ii) Distiller: This is an authorized entity responsible for verifying log entries using associated public keys. For example, in a smart-building scenario, IoT sensors may periodically transmit sensing reports to a local edge cloud. Before transferring logs to cold storage, the edge cloud performs a distillation process that generates *Cold Cryptographic Data (CCD)*, a curated dataset containing valid log batches with compressed, adjustable-granularity cryptographic tags. Invalid (flagged) entry-signature pairs are stored individually. In most real-world applications, such invalid entries are rare, so the storage of *CCD* is dominated by valid entries. After distillation, the edge cloud uploads *CCD* to cold storage servers for long-term archival and audit readiness.

(iii) Cold Storage Server (`CSS`): This server provides a Cold-STaaS platform within the IoT–STaaS continuum. As discussed in Section 1, these systems require periodic audits to demonstrate the trustworthiness of archived digital content [1, 46]. Verifiers regularly check the authenticity and integrity of log data maintained in `CSS`. For simplicity, we assume verifiers are integrated within `CSS` in our system model, and `CSS` is equipped with a GPU hardware card.

3.2 Threat and Security Model

We follow the threat model of cryptographic audit log techniques originally introduced by Schneier et al. in [44] and then improved in various subsequent cryptographic works [19, 55]. In this model, the adversary $\mathcal{A}$ is an active attacker that aims to forge and/or tamper with audit logs to implicate other users. The state-of-the-art cryptographic secure logging schemes rely on digital signatures to thwart such attacks with public verifiability and non-repudiation. As stated in Section 1, we focus on signer-efficient (EC-based) AS-based approaches due to their compactness and fast batch verification.

We follow the *Aggregate Existential Unforgeability Under Chosen Message Attack* (A−EU−CMA) [5] security model that captures our threat model. A−EU−CMA considers the homomorphic properties of aggregate signatures and can offer desirable features such as log order preservation (if enforced) and truncation detection for signature batches [55]. POSLO schemes are single-signer aggregate signatures, and therefore we do not consider inter-signer aggregations.

Definition 3.1. A−EU−CMA experiment for ASGN is as follows:
Experiment $Expt^{\text{A-EU-CMA}}_{\text{ASGN}}(\mathcal{A})$
$(I, sk, PK) \leftarrow \text{ASGN.Kg}(1^{\kappa}, n)$,
$(\boldsymbol{m}^*, \sigma^*) \leftarrow \mathcal{A}^{RO(.),\, \text{ASGN.ASig}_{sk}(.)}(PK)$,
If $\text{ASGN.AVer}(PK, \boldsymbol{m}^*, \sigma^*) = 1$ and $\boldsymbol{m}^* \notin \{\boldsymbol{m}_j\}_{j=1}^{n_1}$, return 1 otherwise return 0.

The A−EU−CMA of $\mathcal{A}$ is defined as

$$Adv^{\text{A-EU-CMA}}_{\text{ASGN}}(\mathcal{A}) = Pr[Expt^{\text{A-EU-CMA}}_{\text{ASGN}}(\mathcal{A}) = 1].$$

The A−EU−CMA advantage of ASGN is defined as

$$Adv^{\text{EU-CMA}}_{\text{SGN}}(t, q_H, q_s) = \max_{\mathcal{A}}\{Adv^{\text{A-EU-CMA}}_{\text{ASGN}}(\mathcal{A})\},$$

where the maximum is over $\mathcal{A}$ having time complexity t, with at most n' queries to the random oracle $RO(.)$ and n queries to ASGN.ASig(.).

The oracles reflect how POSLO works as an ASGN scheme. The signing oracle ASGN.ASig(.) returns an aggregate signature $\tilde{\sigma}$ on a batch of messages $\vec{\boldsymbol{m}} = (\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{n_1})$ computed under sk. ASGN.Agg(.) aggregates the individual (or batch) signatures of these messages. ASGN.Agg(.) can be performed during the signing or before verification (e.g., in the distillation). It can aggregate additive or multiplicative components $\delta_i \in \sigma_i$. $RO(.)$ is a random oracle from which $\mathcal{A}$ can request the hash of any message of her choice up to n' messages. In our proofs (see Section 5), the cryptographic hash function H is modeled as a random oracle [35] via $RO(.)$.

4 PROPOSED SCHEMES

Our goal is to design novel cryptographic secure logging schemes that satisfy the stringent requirements of low-end IoT devices by enabling efficient signing and compact signatures, while also ensuring fast verification and optimal storage at the Cold-StaaS. Specifically, we aim to achieve: *(i)* Near-optimal signer efficiency without relying on costly operations such as EC scalar multiplication, *(ii)* Compact aggregate tag storage and communication overhead, *(iii)* $O(1)$ final cryptographic storage at cold storage, i.e., constant-size public key and signature for maximal compression, *(iv)* Fast batch verification over large message sets, and *(v)* Flexible aggregation support at any desired granularity, either at the signer or verifier side.

EC-based signatures (e.g., Ed25519 [4], SchnorrQ [11]) provide compact signature sizes and improved signing efficiency over RSA [32, 54] and pairing-based schemes [5]. However, they still require at least one EC scalar multiplication per signing operation, which limits their applicability in constrained environments. Prior efforts to address this include: (i) *Precomputation*, which shifts signing costs to key generation through commitment precomputation. While this reduces signing time, it incurs linear storage at the signer, making it impractical for resource-limited devices. (ii) *Seed-based signing*, which replaces precomputed public commitments with one-time seeds [40, 55]. This transforms Schnorr signatures [45] into one-time aggregate signatures, moving the generation and storage of expensive commitments ($R \leftarrow \alpha^r \bmod q, r \xleftarrow{\$} \mathbb{Z}_q^*$) to key generation and the verifier, respectively. The signing process decouples the message m from the commitment by substituting $H(m||R)$ with $H(m||x)$, where x is a one-time randomness. Since x cannot be revealed before signing

and is non-aggregatable, this leads to $O(n)$ storage at Cold-STaaS (e.g., 2TB for 2^{35} log entries of 32 bytes each) and expensive batch verification (hundreds of hours to verify 1TB), as detailed in Section 6. In Sections 1.1 and 6, we analyze AS-based signatures and their limitations in detail. Overall, existing AS-based secure logging solutions either favor signer efficiency at the cost of excessive verifier-side overhead or vice versa, failing to provide a scalable and efficient path for verifying large volumes of log data in Cold-STaaS environments.

In POSLO, we address the persistent *signer-versus-verifier bottleneck* through a set of novel and synergistic design strategies: *(i) Efficient Seed Management:* We introduce a tree-based randomness structure that reduces signer-side seed storage from $O(n)$ to intermediate $O(\log_2 n)$ and ultimately to $O(1)$ in the final form. This design eliminates the need for linear signer storage while preserving deterministic and secure EC-based signing. *(ii) Flexible Tag Aggregation:* POSLO supports adaptive aggregation of additive and multiplicative signature components with configurable granularity. This allows aggregation either at the signer per epoch or on-demand by the verifier. *(iii) Tailored Variants:* We design two core POSLO variants to serve distinct application needs: the *Coarse-grained signer-optimal* POSLO−C, which minimizes signer-side overhead and maximizes compression, and the Fine-grained public-key POSLO−F, which enables precise auditing with constant-size public key and efficient signing. *(iv) High-Throughput Parallel Batch Verification:* We introduce POSLO.PAVer, the first GPU-accelerated batch verification framework (to our knowledge) for mutable AS schemes. POSLO eliminates sequential bottlenecks such as repeated hashing, enabling order-of-magnitude speedups over traditional digital signature schemes. *(v) Multiple Instantiations*: We provide two distinct instantiations: POSLO^{+}, and POSLO^{++}, each offering unique performance-security trade-offs. POSLO+ utilizes MMO and MDC-2 constructions using AES as a block cipher to replace SHA-256 in the original POSLO for the key derivation PRF and message hashing H, respectively. POSLO^{++} replaces MDC-2 with modular addition [9], enabling reduced overhead in low-end signers.

We first describe our seed management architecture and associated data structures that addresses the signer storage challenge in Section 4.1. We then present the POSLO−C and POSLO−F schemes, detailing their efficiency in signing, storage compression, and batch verification at configurable granularity. Lastly, we introduce POSLO.PAVer, our parallelized batch verification engine that achieves significant throughput gains, facilitating scalable and efficient log auditing at Cold-STaaS.

4.1 POSLO Data Types and Seed Management

POSLO Data Types: POSLO Tree-based structure (POSLOT) is designed for seed storage and management, in which the leaves represent one-time seeds x, and the left and right children of each node are computed using $\mathrm{PRF}_{0,1}$, respectively. Let n_1 and $D = \log n_1$ be the total number of leaves and tree height, respectively. The POSLOT root node, $x_D[0]$, serves as the source of randomness in order to derive 2^D one-time seeds $x_0[i], i = 0, \dots 2^D - 1$. The inner nodes of POSLOT $x_d[i]$, $0 \leq i < 2^{D-d}$, where $d = 0, \dots, D-1$, are computed as follows:

$$x_d[i] = \begin{cases} \mathrm{PRF}_0\left(x_{d+1}\left[\left\lfloor \frac{i}{2} \right\rfloor\right]\right), & \text{if } i \equiv 0 \pmod 2 \\ \mathrm{PRF}_1\left(x_{d+1}\left[\left\lfloor \frac{i}{2} \right\rfloor\right]\right), & \text{if } i \equiv 1 \pmod 2 \end{cases} \tag{1}$$

Disclosed Seeds (ds) is a stack data structure with the following operations: Init(ds) initializes the stack ds to an empty stack. $x \leftarrow$ Top(ds) returns the top element of the stack. Push(ds, x) pushes x onto the top of ds. $x \leftarrow$ Pop(ds) removes and return the top element of ds.

ds maintains the disclosed one-time seeds $\{x_0[i]\}_{i=0}^{n_1-1}$ in a $O(\log(n_1))$ compact storage by replacing leaves with their parent nodes whenever it is possible.

Seed Manager: The Seed Management Functions (SMF) are described in Figure 2a: *(i) Seed Computation (SC)* computes the node $x_{d_1}[i_1]$ from the source node $x_{d_0}[i_0]$ by traversing the POSLOT tree.

$x_{d_1}[i_1] \leftarrow \texttt{SC}(x_{d_0}[i_0], d_1, i_1)$: require $d_1 < d_0$ and $i_0 \le i_1 \cdot 2^{d_1-d_0} < i_0 + 1$

1: $x_p \leftarrow x_{d_0}[i_0]$, $i \leftarrow i_1 - i_0 \cdot 2^{d_0-d_1}$
2: **for** $j = d_0 - d_1 - 1, \ldots, 0$ **do**
3: $\quad x_p = \begin{cases} \texttt{PRF}_0(x_p), & \text{if } \lfloor i/2^j \rfloor \equiv 0 \mod 2 \\ \texttt{PRF}_1(x_p), & \text{if } \lfloor i/2^j \rfloor \equiv 1 \mod 2 \end{cases}$
4: **return** $x_{d_1}[i_1] \leftarrow x_p$

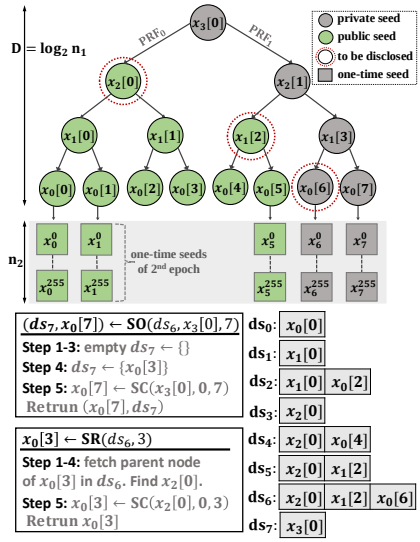
$(ds_i, x_0[i]) \leftarrow \texttt{SO}(ds_{i-1}, x_D[0], i)$: require $0 \le i < 2^D$

1: $d_1 \leftarrow 0$, $i_1 \leftarrow i$, $ds_i \leftarrow ds_{i-1}$
2: **while** Depth($x \leftarrow \texttt{Top}(ds_i)$) $= d_1$ **do**
3: $\quad x \leftarrow \texttt{Pop}(ds_i)$, $d_1 \leftarrow d_1 + 1$, $i_1 \leftarrow \lfloor i_1/2 \rfloor$
4: $\texttt{Push}(ds_i, x_{d_1}[i_1])$, where $x_{d_1}[i_1] \leftarrow \texttt{SC}(x_D[0], d_1, i_1)$
5: **return** $(ds_i, x_0[i] \leftarrow \texttt{SC}(x_{d_1}[i_1], 0, i))$

$x_0[i'] \leftarrow \texttt{SR}(ds, i')$:

1: $x_d[i] \leftarrow \texttt{Pop}(ds)$
2: **if** $i' \ge (i+1) \cdot 2^d$ **then return** $\perp$
3: **while** $i' < i \cdot 2^d$ **do**
4: $\quad x_d[i] \leftarrow \texttt{Pop}(ds)$
5: **return** $x_0[i'] \leftarrow \texttt{SC}(x_d[i], 0, i')$

(a) Seed Management Functions (SMF)



(b) Example of SMF execution ($n_1 = 2^3, n_2 = 2^8$)

(ii) Seed storage Optimizer (SO) progressively discloses ancestor nodes as the signer completes epochs. For a given leaf index i, root $x_D[0]$, and prior ds_{i-1}, it outputs ds_i that represents disclosed nodes during the epoch i. *(iii) Seed Retrieval (SR)* takes a ds instance and leaf index i as inputs. It returns the seed $x_0[i]$ if ds contains an ancestor node for leaf index i.

An instance of POSLOT is provided in Figure 2b, where $(n_1 = 2^3, n_2 = 2^8)$. It shows the POSLOT status after completing the 6th epoch and the execution of SMF algorithms. The seeds, to be disclosed, are highlighted. They can be determined by running SO algorithm. The SO output is: $(ds_6, x_0[6]) \leftarrow \texttt{SO}(ds_5, x_3[0], 6)$ where $ds_5 = \{x_2[0], x_1[2]\}$. The output ds_6 is equal $\{x_2[0], x_1[2], x_0[6]\}$.

The advantage of POSLOT seed management is apparent over the linear disclosure of one-time commitments in Schnorr-like schemes. It reduces both $O(n)$ signer transmission and verifier storage into $O(\log n_1)$. Upon finishing all epochs, the signer discloses the POSLOT root $x_D[0]$, enabling $O(1)$ storage at verifiers.

4.2 Coarse-grained signer-optimal POSLO (POSLO-C)

POSLO-C offers a near-optimal signing efficiency in terms of both computational and storage overhead. It offloads an aggregate tag upon signing an epoch of individual log entries. Unlike previous EC-based signature designs, POSLO-C pre-stores a $O(n_1)$ sublinear number of public commitments (R) at the verifier side and compacts them after receiving the authenticated logs from IoT devices. In the following, we give a detailed description of POSLO-C algorithms.

4.2.1 *POSLO-C Digital Signature Algorithms.*

We give the aggregate signature functions of POSLO-C in Fig. 3a.

In POSLO-C.Kg(.), for a given n, we first select the number of epochs (n_1) and items to be signed per epoch (n_2) (Step 1). We generate EC-based parameters (p, q, α) and private/public key pair (y, Y) (Steps 2-3). We then generate the initial ephemeral randomness r and the root of POSLOT tree $x_D[0]$ (Step 4). These values will be used to generate ephemeral public commitments (R) and one-time randomness (x) for a given epoch state St. POSLO-C is coarse-grained, thus we combine the

$(I, sk, PK) \leftarrow$ POSLO-C.Kg$(1^\kappa, n)$:

1: Choose (n_1, n_2) s.t. $n = n_1 \cdot n_2$ and $n_1 = 2^D$ where $D \in \mathbb{N}^*$.
2: Generate large primes q and $p > q$ such that $(p-1)$ divides q. Select a generator α of the subgroup $\mathbb{G}$ of order q in $\mathbb{Z}_q^*$.
3: $y \xleftarrow{\$} \mathbb{Z}_q^*$, $Y \leftarrow \alpha^y \bmod p$
4: $x_D[0] \xleftarrow{\$} \{0,1\}^\kappa$, $r \xleftarrow{\$} \mathbb{Z}_q^*$, Init(ds_0)
5: **for** $i = 0, \ldots, n_1 - 1$ **do**
6: $\quad \tilde{r}_i \leftarrow \sum_{j=0}^{n_2-1} r_i^j \bmod q$, where $r_i^j \leftarrow \mathrm{PRF}_0(r \,\|\, i \,\|\, j)$
7: $\quad \tilde{R}_i \leftarrow \alpha^{\tilde{r}_i} \bmod p$
8: $sk \leftarrow (y, r, x_D[0])$, $PK \leftarrow (Y, \boldsymbol{R})$, where $\boldsymbol{R} \leftarrow \{\tilde{R}_i\}_{i=0}^{n_1-1}$
9: The public parameter $I \leftarrow (p, q, \alpha, n_1, n_2, St := (i = 1, ds_0))$
10: **return** (I, sk, PK)

$\tilde{\delta} \leftarrow$ POSLO-C.Agg$(\{\delta_j \in \sigma_j\}_{j=1}^{\ell})$:

1: **if** $\delta_1 \in \mathbb{Z}_q^*$ **then** $\tilde{\delta} \leftarrow \sum_{j=1}^{\ell} \delta_j \bmod q$ **else** $\tilde{\delta} \leftarrow \prod_{j=1}^{\ell} \delta_j \bmod p$
2: **return** $\tilde{\delta}$

$\tilde{\sigma}_i \leftarrow$ POSLO-C.Sig$(sk, \boldsymbol{m_i})$:

1: **if** $i > n_1$ and $\boldsymbol{m_i} \neq \{m_i^j\}_{j=0}^{n_2-1}$ **then return** $\perp$
2: $(ds_i, x_0[i]) \leftarrow \mathrm{SO}(ds_{i-1}, x_D[i], i)$
3: **for** $j = 0, \ldots, n_2 - 1$ **do**
4: $\quad e_i^j \leftarrow H(m_i^j \,\|\, x_i^j) \bmod q$, where $x_i^j \leftarrow \mathrm{PRF}_0(x_D[i] \,\|\, j)$
5: $\quad s_i^j \leftarrow r_i^j - e_i^j \cdot y \bmod q$, where $r_i^j \leftarrow \mathrm{PRF}_0(r \,\|\, i \,\|\, j) \bmod q$
6: $\tilde{s}_i \leftarrow$ POSLO-C.Agg$(\{s_i^j\}_{j=0}^{n_2-1})$, $St \leftarrow (i+1, ds_i)$
7: **return** $\tilde{\sigma}_i \leftarrow (\tilde{s}_i,\ ds_i)$

$b \leftarrow$ POSLO-C.AVer$(PK, \vec{\boldsymbol{m}}, \tilde{\sigma})$: $\vec{\boldsymbol{m}} = \{\boldsymbol{m_i}\}_{i \in \boldsymbol{I}}$

1: **if** $|\boldsymbol{m_i}| \not\equiv 0 \bmod n_2, \forall i \in \boldsymbol{I}$ **then return** $\perp$
2: **if** $\tilde{R} \notin \tilde{\sigma}$ **then** $\tilde{R} \leftarrow$ POSLO-C.Agg$(\{\tilde{R}_i \in PK\}_{i \in \boldsymbol{I}})$
3: $\tilde{e} \leftarrow 0$
4: **for** $i \in \boldsymbol{I}$ **do**
5: $\quad x_0[i] \leftarrow \mathrm{SR}(ds, i)$
6: $\quad$ **for** $j = 0, \ldots, n_2 - 1$ **do**
7: $\quad\quad x_i^j \leftarrow \mathrm{PRF}_0(x_0[i] \,\|\, j)$
8: $\quad\quad \tilde{e} \leftarrow \tilde{e} + H(m_i^j \,\|\, x_i^j) \bmod q$
9: **if** $\tilde{R} = Y^{\tilde{e}} \cdot \alpha^{\tilde{s}} \bmod p$ **then return** $b = 1$ **else return** $b = 0$

(a) Digital signature algorithms

$CCD_i \leftarrow$ POSLO-C.Distill$(PK, CCD_{i-1}, \boldsymbol{m_i}, \tilde{\sigma}_i, n^u)$: require $n_1 \equiv 0 \bmod n^u$

Init $\tilde{s}^u \leftarrow 0$, $\tilde{R}^u \leftarrow 1$, $\tilde{s} \leftarrow 0$, $\tilde{R} \leftarrow 1$
1: $b_i \leftarrow$ POSLO-C.AVer$(PK, \boldsymbol{m_i}, \tilde{\sigma}_i)$
2: **if** $b_i = 1$ **then**
3: $\quad \tilde{s}_i^v \leftarrow$ POSLO-C.Agg$(\tilde{s}_{i-1}^v, \tilde{s}_i \in \tilde{\sigma}_i)$
4: $\quad \tilde{R}_i^v \leftarrow$ POSLO-C.Agg$(\tilde{R}_{i-1}^v, \tilde{R}_i \in \tilde{\sigma}_i)$
5: $\quad \tilde{\sigma}_i^v \leftarrow (\tilde{s}_i^v, \tilde{R}_i^v)$ ▷ valid signature
6: $\quad CCD_i^v \leftarrow CCD_{i-1}^v \cup \{\tilde{\sigma}_i^v\}$
7: $\quad \tilde{s}^u \leftarrow$ POSLO-C.Agg$(\tilde{s}^u, \tilde{s}_i)$
8: $\quad \tilde{R}^u \leftarrow$ POSLO-C.Agg$(\tilde{R}^u, \tilde{R}_i)$
9: $\quad$ **if** $i \equiv 0 \mod \lfloor n1/n^u \rfloor$ **then**
10: $\quad\quad \tilde{\sigma}_i^u \leftarrow (\tilde{s}^u, \tilde{R}^u)$ ▷ umbrella signature
11: $\quad\quad CCD_i^u \leftarrow CCD_{i-1}^u \cup \{\tilde{\sigma}_i^u\ , \lfloor \frac{i}{\lfloor n_1/n^u \rfloor} \rfloor\}$
12: $\quad\quad$ reset $\tilde{s}^u = 0$, $\tilde{R}^u = 1$
13: **else**
14: $\quad \tilde{\sigma}_i{}^i \leftarrow (\tilde{s}_i, \tilde{R}_i)$, where $\tilde{s}_i \in \tilde{\sigma}_i$ and $\tilde{R}_i \in PK$ ▷ invalid signature
15: $\quad CCD_i^i \leftarrow CCD_{i-1}^i \cup \{\tilde{\sigma}_i{}^i, i\}$
16: remove $\tilde{R}_i$ from $\boldsymbol{R} \in PK$
17: $CCD_i \leftarrow (CCD_i^v, CCD_i^u, CCD_i^i, ds_i \in \tilde{\sigma}_i)$
18: **return** CCD_i

$\boldsymbol{b} \leftarrow$ POSLO-C.SeBVer$(PK, \vec{\boldsymbol{m}}, CCD, \mu)$: require $|\vec{\boldsymbol{m}}| \equiv 0 \bmod n_2$

1: **switch** (μ)
2: $\quad$ **case** "V":
3: $\quad\quad \vec{\boldsymbol{m}} \leftarrow \{\boldsymbol{m_i}\}_{i \in \boldsymbol{I} \setminus CCD^i}$
4: $\quad\quad b^v \leftarrow$ POSLO-C.AVer$(PK, \vec{\boldsymbol{m}}, \tilde{\sigma}^v \in CCD^v)$
5: $\quad\quad \boldsymbol{b} = \{b^v\}$
6: $\quad$ **case** "U":
7: $\quad\quad$ **for** $(\tilde{\sigma}_\ell, i_\ell) \in CCD^u$ **do**
8: $\quad\quad\quad \vec{\boldsymbol{m}}' \leftarrow \{\boldsymbol{m_i}\}_{i \in \{i_\ell \cdot \lfloor n^u/n \rfloor, \ldots, (i_\ell+1) \cdot \lfloor n^u/n \rfloor - 1\} - CCD^i}$
9: $\quad\quad\quad b_\ell^u \leftarrow$ POSLO-C.AVer$(PK, \vec{\boldsymbol{m}}', \tilde{\sigma}_\ell)$
10: $\quad\quad \boldsymbol{b} \leftarrow \{b_\ell^u\}_{\ell \in |CCD^u|}$
11: $\quad$ **case** "I":
12: $\quad\quad$ **for** $(\tilde{\sigma}_\ell, i_\ell) \in CCD^i$ **do**
13: $\quad\quad\quad b_\ell^i \leftarrow$ POSLO-C.AVer$(PK, \boldsymbol{m_{i_\ell}}, \tilde{\sigma}_\ell)$
14: $\quad\quad \boldsymbol{b} \leftarrow \{b_\ell^i\}_{\ell \in |CCD^i|}$
15: **return** $\boldsymbol{b}$

(b) Distillation and selective batch verification

Fig. 3. Coarse-grained signer-optimal POSLO (POSLO-C)

commitments for each epoch as in Steps 5-7, which results in initial $O(n_1)$ and final $O(1)$ storage at the verifier via aggregation. The private/public keys and public parameters are as in Steps 8-9.

POSLO-C.Agg(.) is a keyless signature aggregate function with dual signature combination mode. That is, given a signature element $s \in \sigma$ or $R \in \sigma$, it performs an additive or multiplicative aggregation, respectively. This generic construction enables the aggregation of different keys during the signing and/or batch verification algorithms.

POSLO-C.Sig(.) is an aggregate signature generation that signs each entry and sequentially aggregates into a single umbrella signature (i.e., the tag authenticates all items in the epoch). The signer first checks if the message set complies with the epoch size n_2 (Step 1). The seed $x_0[i]$ is then computed via SO once per epoch i (Step 2) and is used to derive one-time seeds x_i^j (Step 4). The aggregate signature $\tilde{s}_i$ is computed with only a few hashing and modular additions plus a modular multiplication (Steps 3-6). This makes POSLO-C the most signer efficient POSLO scheme. At the end of epoch i, the signer updates its internal state and outputs the aggregate signature $\tilde{\sigma}_i$ (Steps 6-7).

POSLO-C.AVer(.) accepts as input the public key PK, a set of messages $\vec{\boldsymbol{m}}$, and an aggregate signature $\tilde{\sigma}$. The verifier checks if messages comply with the epoch size (Step 1), and then identifies the format of the aggregate signature to choose a component R (Step 2). POSLO-C.AVer(.) can be invoked by the edge cloud or CSS as the final verifier. This difference dictates if the aggregate commitment R is included in the initial public key PK or the aggregate signature σ. Below, we will elaborate further that the POSLO-C.Distill(.) function can be used to verify the entries and then compact them according to a granularity parameter ρ. Hence, if verification is done during the distillation, the verifier already has $\tilde{R}_i \in \boldsymbol{R}$ as part of PK and this value is used during verification (Step 9). Otherwise, if the verification is run by the CSS, then $\tilde{R}$ can be found as a part of the signature in CCD. The verifier retrieves the seeds in the given epoch via SR (Step 5) and then computes the aggregate hash component $\tilde{e}$ (Steps 6-8). Finally, the aggregate signature is verified (Step 9). Figure 2b depicts the mechanism for seed retrieval. It consists of the verifier's view after finishing 6$^{\text{th}}$ epoch. It illustrates the request to retrieve the seed of the 3$^{\text{rd}}$ epoch (i.e., $x_0[3] \leftarrow \text{SR}(ds_6, 3)$).

4.2.2 POSLO-C Distillation and Selective Batch Verification. The verification involves two entities from our system model (as in Section 3): *(i) Distiller* and *(ii) Cold Storage Server* (CSS).

The CSS maintains the cryptographic cold data (CCD), which is updated by the distiller. Figure 3b formally describes the distillation and batch verification algorithms. Initially, both entities initialize the CCD as empty sets of signatures. POSLO-C.Distill(.) updates the CCD structure by aggregating valid signatures and adding them to CCD^v (Steps 2-6). The CSS stores valid signatures according to the granularity level ρ, resulting in: (i) an overall valid tag CCD^v, (ii) a set of umbrella valid signatures CCD^u (Steps 7-12), and (iii) individual invalid signatures CCD^i (Step 15).

By storing sub-aggregates (i.e., umbrella aggregates) in CCD^v, umbrella signatures enable localization of invalid log data chunks when the batch verification of the overall tag fails.

POSLO-C.SeBVer(.) is a selective batch verification algorithm that operates in three modes: *(i)* Mode "V" verifies CCD^v, which consists of one aggregate signature for all valid messages. *(ii)* Mode "U" checks partial umbrella signatures if the overall verification (mode "V") fails. The storage overhead for CCD can be adjusted according to the granularity parameter ρ. *(iii)* Mode "I" verifies the invalid set by checking each entry individually.

The generic batch verification POSLO-C.AVer(.) supports both the edge server and CSS in the distillation and verification processes, respectively.

4.3 Fine-grained public-key POSLO (POSLO-F)

Our fine-grained variant POSLO-F, as shown in Figure 4, utilizes BPV [6] (see Definition 2.5) to pre-store a constant size of one-time commitments at the signer. Prior works [40] have shown that this storage overhead is tolerable for some low-end IoT devices. The pre-computation is important for immediate and fine-grained verification at the edge server and allows CSS to authenticate log entries individually, enabling precise investigation and optimal recovery.

POSLO-F provides several performance advantages on the distiller side. It enables immediate verification of each message within an epoch by attaching the seed x_i^j to the signature, as shown in POSLO-F.Sig(.) (Step 6). Unlike POSLO-C, POSLO-F allows $O(1)$ public-key storage at the distiller. The signer generates a commitment R_t using BPV (Step 3) and includes it in the signature (Step 6). This eliminates the initial $O(n_1)$ public key storage (as in POSLO-C) and allows the highest granularity by enabling individual signature verification.

The distillation and selective batch verification functionalities are similar to those of POSLO-C with minor differences. The edge server aggregates each signature separately. The invalid set CCD^i contains individual signatures (at the highest granularity), requiring CSS to verify each

invalid entry separately. Thus, POSLO−F offers better verification precision than POSLO−C, albeit with slightly slower verification and higher signer communication overhead.

$(I, sk, PK) \leftarrow$ POSLO-F.Kg$(1^\kappa, n)$: Steps 1-4 are identical to POSLO−C.Kg(.), the rest is as follows:

1: $(\Gamma, v, k) \leftarrow$ BPV.Offline$(1^\kappa, p, q, \alpha)$
2: $sk \leftarrow (y, r, x_D[0], \Gamma)$, $PK \leftarrow Y$
3: The public parameters $I \leftarrow (p, q, \alpha, v, k, n_1, n_2, St := (t = 0, ds_0))$
4: **return** (I, sk, PK)

$\sigma_t \leftarrow$ POSLO-F.Sig(sk, m_t): Given $t \leq n$

1: $i \leftarrow \lfloor \frac{t}{n_2} \rfloor$; $j \leftarrow i \bmod n_2$
2: **if** $j = 1$ **then** $(ds_i, x_0[i]) \leftarrow$ SO$(ds_{i-1}, x_D[0])$
3: $(r_t, R_t) \leftarrow$ BPV.Online(Γ, v, k)
4: $e_t \leftarrow H(m_t \parallel x_t) \bmod q$, where $x_t \leftarrow \mathrm{PRF}_0(x_0[i] \parallel j)$
5: $s_t \leftarrow r_t - e_t \cdot y \bmod q$, $St := (t + 1, ds_i)$
6: **if** $j = n_2$ **then return** $\sigma_t \leftarrow (s_t, R_t, ds_i)$ **else return** $\sigma_t \leftarrow (s_t, R_t, x_t)$

$b \leftarrow$ POSLO-F.AVer$(PK, \vec{m}, \sigma)$:

1: **if** $|\vec{m}| = 1$ **then** $e \leftarrow H(m \parallel x) \bmod q$, where $\vec{m} = \{m\}$ and $x \in \sigma$
2: **else** execute POSLO−C.AVer Steps 2-8
3: **if** $R = Y^e \cdot \alpha^s \bmod p$ **then return** $b = 1$ **else return** $b = 0$

Fig. 4. Fine-grained public-key POSLO (POSLO−F)

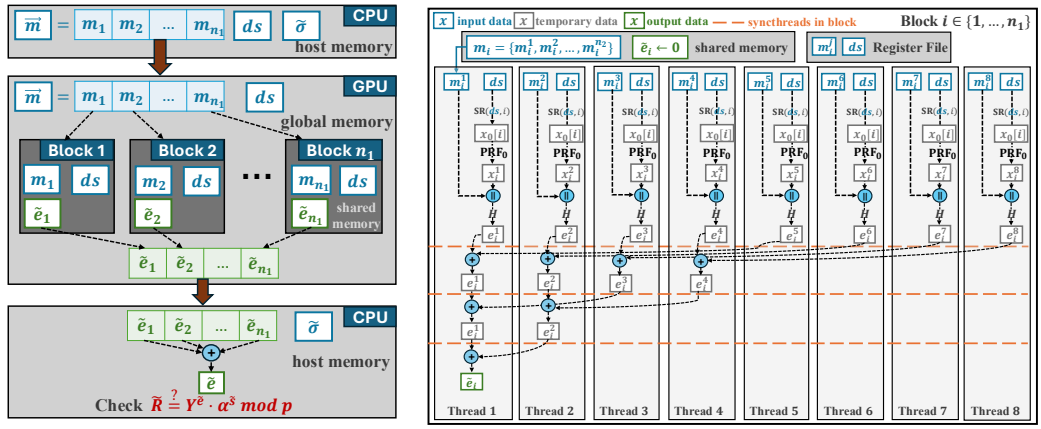
4.4 POSLO Parallel Batch Verification (POSLO.PAVer)

POSLO signature verification, as well as other batch verification algorithms (e.g., [14]), offers significantly increased computational efficiency compared to traditional per-message verification. This improvement stems primarily from reducing the number of expensive operations, such as EC scalar multiplications in ECDLP-based signatures. Nonetheless, batch verification can remain costly for large datasets due to sequential hashing operations needed to derive one-time ephemeral keys e_i^j (see Step 8 in POSLO−C.AVer, Fig. 3a). For example, without hardware acceleration, verifying 1 GB of log data (with 32-byte entries) using POSLO.AVer takes approximately 30 seconds on commodity hardware. In contrast, the BLS aggregate signature scheme requires roughly 2.55 hours due to costly map-to-point hashing. However, as the size of dataset grows, the cumulative overhead of hashing becomes a bottleneck for real-time log verification, even for efficient primitives like POSLO.

To address this gap, we propose *Parallel batch verification* (POSLO.PAVer), which exploits two key properties: (i) the independence of per-signature hashing (e.g., in POSLO−C.AVer), and (ii) the additive homomorphism of the aggregation function (i.e., modular addition) used to compute the aggregate ephemeral key $\tilde{e}$. These properties allow the hashing and aggregation steps to be parallelized or shuffled without compromising correctness. This observation motivates the design of POSLO.PAVer for efficient batch verification at the CSS. While the edge server (i.e., the distiller) can also leverage POSLO verification, the benefits of parallelization are most pronounced at the CSS, which archives the entire log stream.

POSLO.PAVer is a CUDA-based parallel verification algorithm designed to exploit the inherent parallelism in mutable aggregate signatures, focusing on POSLO batch verification. The architecture of POSLO.PAVer is shown in Figure 5, with a formal description in Figure 6.

Verification begins by initializing the aggregate ephemeral key $\tilde{e}$, then executing the CUDA kernel POSLO.Agg−eKeys, which computes the individual ephemeral keys e_i^j in parallel and aggregates them into per-epoch subaggregates $\{\tilde{e}_i\}_{i \in I}$ using a tree-based parallel reduction (Step 8). CUDA blocks are mapped to batches in $\vec{m} = \{m_i\}_{i \in I}$, and the threads per block correspond to the number of items to be processed per epoch n_1. Figure 5b shows the kernel execution for a single block with



(a) CPU-GPU communication workflow (b) CUDA kernel execution within a block ($n_2 = 8$)

Fig. 5. High-level illustration of the parallel batch verification algorithm (POSLO.PAVer)

$\tilde{e} \leftarrow$ POSLO.Agg-eKeys$(\vec{m}, ds)$: $\vec{m} = \{\boldsymbol{m_i}\}_{i \in \boldsymbol{I}}$, where $\boldsymbol{m_i} = \{m_i^j\}_{j=1}^{n_2}$

CUDA kernel

Ensure: $\tilde{e}_i$ (global memory) $\leftarrow 0, \forall \tilde{e}_i \in \tilde{e}$

1: **For each** block $i \in \boldsymbol{I}$ **in parallel do**
2: **For each** thread $j = 1, \ldots, n_2$ **in parallel do**
3: m_i^j (register) $\leftarrow m_i^j$ (global memory)
4: $x_0[i] \leftarrow \mathrm{SR}(ds, i)$
5: $x_i^j \leftarrow \mathrm{PRF}_0(x_0[i] \parallel j)$
6: $e_i^j \leftarrow H(m_i^j \| x_i^j) \mod q$
7: synchronize threads in the block i
8: $\tilde{e}_i \leftarrow \sum_{j=1}^{n_2} e_i^j \mod q$
9: **return** $\tilde{e} \leftarrow \{\tilde{e}_i\}_{i=1}^{n_1}$

$b \leftarrow$ POSLO.PAVer$(PK, \vec{m}, \tilde{\sigma})$: Given $\tilde{\sigma} = \{\tilde{s}, \tilde{R}\}$

1: $\tilde{e} \leftarrow 0$
2: $\tilde{e} \leftarrow$ POSLO.Agg-eKeys$(\vec{m}, ds)$
3: **For** $i \in \boldsymbol{I}$ **omp parallel do**
4: $\tilde{e} \leftarrow \tilde{e} + \tilde{e}_i \mod q$
5: **if** $\tilde{R} = Y^{\tilde{e}} \cdot \alpha^{\tilde{s}} \bmod p$ **then return** $b = 1$ **else return** $b = 0$

Fig. 6. Parallel POSLO signature verification algorithm (POSLO.PAVer)

$n_2 = 8$. Each batch $\boldsymbol{m_i}$ is first loaded from global memory into shared memory, and each log entry m_i^j is placed into the register space of thread j in block $i \in \boldsymbol{I}$. Each thread $j \in \{1, \ldots, n_2\}$ computes the seed $x_0[i]$ using the SR algorithm (Step 4), and then computes its own ephemeral key e_i^j (Steps 5-6), and a synchronization is performed to ensure readiness for aggregation. A tree-based parallel reduction is performed within each block (see Figure 5) to compute an aggregate $\tilde{e}_i$, which is then written back to global memory for final aggregation on the CPU.

The final aggregation of $\tilde{e}_i$ values into $\tilde{e}$ can be done using another CUDA kernel, OpenMP (OMP) [7], or sequentially on the CPU. Because aggregation involves $O(n_1)$ modular addition and $\tilde{e}_i \in \mathbb{Z}_q$ for all $i \in \boldsymbol{I}$, this step is lightweight even when done sequentially.

POSLO.PAVer is applied to the valid set CCD^v , which contains the overall aggregate tag. It can also be applied to the umbrella set CCD^u by re-aggregating $\{\tilde{e}_i\}_{i \in \boldsymbol{I}}$ into coarser aggregates according to the selected granularity parameter ρ.

Overall, POSLO.PAVer significantly reduces CPU load by offloading expensive hashing and PRF operations to the GPU. Section 6 details the speedups obtained with various instantiations of H and PRF, and compares them with both baseline and state-of-the-art digital signature schemes.

5 SECURITY ANALYSIS

We formally prove that POSLO−C and POSLO−F schemes are A−EU−CMA-secure AS schemes in the Random Oracle Model (ROM) in [35] Theorem 5.1 and Lemma 5.1. We ignore terms that are negligible in terms of κ. We prove that POSLO^{+} and POSLO^{++} instantiations are as secure as POSLO in Corollary 5.3. The full security proofs are given in Appendix A.

THEOREM 5.1. *$Adv^{A-EU-CMA}_{POSLO-C(p,q,\alpha)}(t, n', n) \leq Adv^{DL}_{\mathbb{G},\alpha}(t')$, where $t' = O(t) + O(n \cdot (\kappa^3 + RNG))$.*

LEMMA 5.2. *POSLO−F is as secure as POSLO−C.*

COROLLARY 5.3. *POSLO^{+} and POSLO^{++} instantiations are as secure as POSLO.*

6 PERFORMANCE ANALYSIS

This section presents a detailed performance comparison of POSLO schemes and their counterparts.

6.1 Evaluation metrics

Our evaluation considers the following metrics: *(i)* signer's execution time and energy consumption, *(ii)* private/public key sizes, *(iii)* signature size, *(iv)* individual or sub-aggregate signature verification and batch verification execution times at CSS, and *(v)* cryptographic cold storage at CSS.

We select the main counterparts that represent key families of AS schemes: *(i)* Factorization-based: C-RSA [54] is an AS scheme offering near-optimal signature verification. *(ii)* ECDLP-based: SchnorrQ [11] is one of the fastest EC-based signature schemes (compared to ECDSA or Ed25519 [4]), with high efficiency on embedded devices. FI-BAF [55] is a signer-optimal FAS scheme, and is our closest logger-efficient comparator. *(iii)* Pairing-based: BLS [5] is a multi-user AS scheme based on bilinear maps, offering the most compact storage among the alternatives. Moreover, BLS is widely adopted in recent AS schemes with enhanced properties (e.g., [28, 50]) in IoT networks, making it a natural benchmark to highlight the efficiency advantage of POSLO.

In the remainder of this section, we present the parameters and hardware/software setup used in our evaluation. We then provide an in-depth performance analysis on the resource-constrained logger (signer), edge cloud (distiller), and cold storage server (final log repository and auditor).

6.2 Instantiations and Configurations

We set κ to 128-bit security. We used FourQ curve [11] and set $|q| = 256$ bits for the EC-based schemes. The BPV parameters are $(v, k) = (1024, 16)$. The composite modulus in C-RSA is $|n| = 2048$.

Table 2. Instantiations of POSLO schemes using different PRF and H functions

Scheme	$\mathbf{PRF}_{0,1}$	H	Standard compliance	Hardware Support	High Efficiency	Large Input Sizes
POSLO	SHA-256	SHA-256	✓	✗	✗	✓
POSLO^{+}	MMO-AES-128	MDC-AES-128	✗	✓	✓	✓
POSLO^{++}	MMO-AES-128	Add_q	✗	✓	✓	✗

Modular addition (Add_q) cannot evaluate inputs larger than the modulus q.

6.2.1 Instantiations of $PRF_{0,1}$. We derive the pseudo-random functions $\texttt{PRF}_{0,1}$ from: (i) Standard cryptographic hash function SHA-256. (ii) MMO construction with AES-128 as a block cipher (E).

$$\texttt{PRF}_j(x) = F(x\|j), \forall F \in \{\text{MMO-AES-128}, \text{SHA-256}\}, j = 0, 1, x \in \{0,1\}^\kappa$$

$\texttt{PRF}_{0,1}$ are used in POSLO signature algorithms to derive one-time keys (i.e., x_i^j and r_i^j) and in SMF algorithms to derive the left and right child nodes, respectively. Each node $\{x_d[i]\}_{d,i}$ in the POSLOT tree has a bit-length of κ. For $\kappa = 128$, $|x_d[i]\|j|$ is 129 bits, where $j \in \{0,1\}$, requiring two AES evaluations per PRF call. However, if $\kappa = 127$, the concatenated input $|x_d[i]\|j|$ becomes exactly 128 bits, matching the AES block size and allowing a single AES call, reducing the computational overhead by half. This optimization makes the $\kappa = 127$ configuration the lightest and fastest variant, with slightly relaxed security guarantees.

6.2.2 Instantiations of H. We use multiple instantiations of H in POSLO signature schemes:

(i) *Standard cryptographic hash function.* SHA-256 is used to ensure standard compliance. This mode incurs more computational overhead since cryptographic hash functions are sequential and do not provide high parallelism and efficiency.

(ii) *MDC-2 construction with AES-128.* Given a 128-bit input, MDC-2 [35] outputs a 256-bit digest using two iterations of the MMO construction (see Definition 2.2). When E is instantiated with AES-128, MDC-2 delivers efficient performance on both constrained and commodity devices, benefiting from highly optimized hardware and software AES implementations.

(iii) *Modular addition.* For small input messages (i.e., $|m| < q$), Chen et al. [9] demonstrate that modular addition can be securely and efficiently used in Schnorr-based digital signatures. Given a message m, private key (y, r), and one-time seed x, the POSLO signature becomes $\sigma \leftarrow (s, x)$ where $s \leftarrow r + (m + x) \cdot y \mod q$. This variant enables ultra-lightweight signature generation that is especially well-suited for resource-constrained devices.

6.2.3 Instantiations of POSLO signatures. Table 2 summarizes our POSLO signature instantiations, each defined by a particular combination of PRF and H functions. The POSLO variant ensures standard compliance with cryptographic standards by exclusively using SHA-256. $\texttt{POSLO}^+$ and $\texttt{POSLO}^{++}$ variants prioritize efficiency through the reliance on AES-based constructions. These benefit from the hardware acceleration available across platforms, including AES-NI [21] and GPU-accelerated AES [18, 48]. We demonstrate the high efficiency of $\texttt{POSLO}^+$ in subsequent sections on both resource-constrained IoT devices and commodity platforms.

6.2.4 Hardware and Software Configurations. We evaluate POSLO on the following testbeds:
(i) Signing/Verification on x86/64. A desktop with an Intel i9-11900K@3.5GHz and 64 GB of RAM.
(ii) Signing on 8-bit. A low-end 8-bit AVR ATmega2560 microcontroller, due to its low energy consumption and extensive use in practice. It is equipped with 256KB flash memory, 8KB SRAM, and 4KB EEPROM, with a clock frequency of 16MHz.
(iii) Verification on GPUs. A commodity desktop with an Intel i9-11900K@3.5GHz processor and 64 GB of RAM. It also includes an NVIDIA GTX 3060 GPU, which provides CUDA 3584 cores, 12GB GDDR6-based memory, and 360GB/s memory bandwidth. In our benchmarks, we include the memory communication overhead between the CPU's main memory and the GPU's global memory.

6.3 Performance Analysis on Signer (Logger)

6.3.1 Analytical Evaluation. Table 3 illustrates the efficiency of POSLO-C signature generation, which only requires 2 PRF and one H calls (on average), two and one modular additions and multiplication, respectively. This makes it as lightweight as its most signer-efficient counterpart FI-BAF, but with a vastly superior performance at CSS. POSLO-C is significantly more signer

Table 3. Private/public key and signature sizes, and signature generation/verification costs of POSLO and its counterparts

Scheme	Logger (Signer)			Edge Server (Distiller)		
	Signature Generation	$\lvert sk\rvert$	$\lvert\sigma\rvert$	$\lvert PK\rvert$	Signature Verification ($\times n_2$)	Distill & Agg ($\times \tau_S \cdot n_2$)
SchnorrQ [11]	$2H + Add_q + Mul_q + EMul$	$\lvert q\rvert$	$2\lvert q\rvert$	$\lvert q\rvert$	$H + 1.3 \cdot EMul$	N/A
FI-BAF [55]	$3H + 2Add_q + Mul_q$	$2 \cdot (\lvert q\rvert + \kappa)$	$\lvert q\rvert + \kappa$	$2n \cdot (\lvert q\rvert + \kappa)$	$2 \cdot (H + Add_q) + 2.3 \cdot EMul$	Add_q
C-RSA [54]	$H + Exp\vert^{d_{RSA}\vert}_{n_{RSA}\vert}$	$2\lvert n_{RSA}\rvert$	$\lvert n_{RSA}\rvert$	$2\lvert n_{RSA}\rvert$	$H + Exp\vert^{e_{RSA}\vert}_{n_{RSA}\vert}$	$Mul_{n_{RSA}}$
BLS [5]	$MtP + EMul'$	$\lvert q\rvert$	$\lvert q\rvert$	$2\lvert q\rvert$	$MtP + Pr$	Mul_q
POSLO-C	$3 \cdot PRF + H + 2Add_q + Mul_q$	$\lvert q\rvert + 2\kappa$	$\lvert q\rvert$	$n_1 \cdot \lvert q\rvert$	$PRF + H + Add_q + (PRF + 1.3 \cdot EMul)/n_2$	$(Add_q + EAdd)/n_2$
POSLO-F	$2 \cdot PRF + H + Add_q + Mul_q$ $+k \cdot (Add_q + EAdd)$	$2 \cdot v \cdot \lvert q\rvert + \kappa$	$2\lvert q\rvert + \kappa$	$\lvert q\rvert$	$H + 1.3 \cdot EMul$	$Add_q + EAdd$

Add_q and Mul_q denote modular addition and multiplication, respectively, with modulus q. $EMul$, $EMul'$ are EC scalar multiplication on Four$\mathbb{Q}$ and pairing-based curves, respectively. We used double-point scalar multiplication (e.g., $1.3EMul$ instead of $2EMul$ for Four$\mathbb{Q}$). Pr is a pairing operation. $Exp\vert^{\lvert x\rvert}_{y\rvert}$ denotes modular exponentiation with exponent x and modulus y. τ_S denotes the success verification rate.

efficient than all other alternatives in terms of runtime, with a highly compact signature and small key sizes. POSLO-F is the second most signer efficient alternative that requires constant number (e.g., 16) of *EAdd* operations. It relies on a pre-computed BPV table, which increases its private key size in exchange for better signing efficiency. Note that the use of BPV can be avoided by accepting a single *EMul*, which makes POSLO-F signing cost equal to that of SchnorrQ. We remind that POSLO-F accepts extra signing/verification cost over POSLO-C in exchange for finer granularity.

Seed Management Overhead Analysis. The amortized seed management computational overhead of POSLO signing algorithms across n messages is on average one PRF call based on the derivation and disclosure of seeds by SC and SO algorithms, respectively. The average amortized cost is $(1 + \frac{\log n_1}{n_2}) \cdot$ PRF, which corresponds to less than two PRF calls, therefore we conservatively accept it as two PRF calls.

The storage overhead of POSLOT stack structure (i.e., *ds*) is $\log(n_1) \cdot \kappa$ bytes. We used an array-based implementation of the stack (i.e., *ds*) [22] since the maximum number of elements in *ds* is known and limited to $D = \log(n_1)$ (height of POSLOT) nodes. At the end of the last epoch (i.e., $i = n_1$), the signer discloses the POSLOT root $x_D[0]$, enabling the CSS to verify any previous message-signature pair with $O(1)$ final storage.

6.3.2 Experimental Evaluation on 8-bit AVR. Figure 7 showcases the energy usage of POSLO schemes and their counterparts compared to that of sensors typically found in IoT devices. Specifically, we compared the energy usage of a single signature generation with that of sampling via pulse[3] and pressure[4] sensors (10s per sampling time with 1ms reading time).

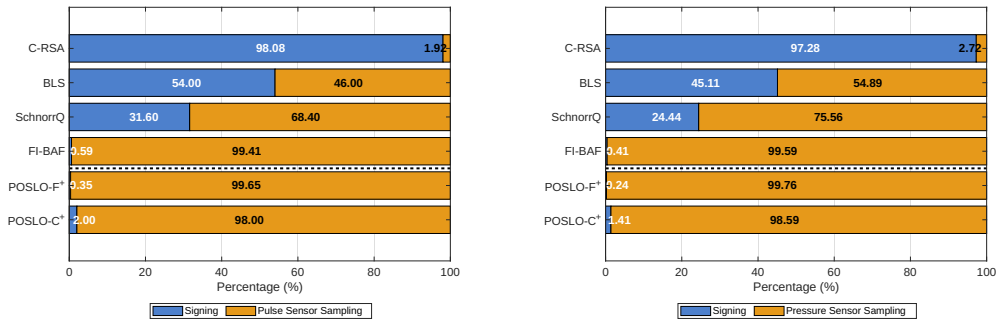


Fig. 7. Energy consumption of POSLO schemes and their counterparts at the logger side

[3]https://pulsesensor.com/

[4]https://cdn-shop.adafruit.com/datasheets/1900_BMP183.pdf

Our most efficient instantiations using AES-based PRF and hashing functions, POSLO−C$^+$ and POSLO−F$^+$, have remarkably low energy usage with 0.35% and 2%, respectively, compared to that of the pulse sensor. For POSLO−C$^+$, this translates into 90× and 154× lower energy usage than the most efficient standard SchnorrQ and verifier compact BLS, respectively. POSLO−C$^+$ is 1.65× lower energy than FI-BAF, but with substantial gains in cold storage, which will be discussed in further detail in the following. POSLO−F$^+$ is the third most energy efficient alternative, while offering fine granularity and higher verification efficiency.

Traffic Variation and Bandwidth Usage. Table 1 depicts the signer cryptographic payload, by enabling full aggregation (per epoch). The signer-efficient variant, POSLO−C, has equal bandwidth overhead compared to both of the short signature scheme BLS and the most signer-efficient counterpart FI-BAF. POSLO−C is the most suitable during a low-frequency upload since it has a lightweight signature generation with a compact signature size. For a high-frequency upload and/or more available battery lifetime, POSLO−F offer higher precision by uploading individual signatures to a nearby edge cloud, to be verified and distilled separately. Table 4 depicts the variation of the signer's cryptographic payload under different epoch sizes. Recall that the epoch size represents the number of individual tags to be aggregated. That is, the low-end devices can increase the epoch size when low bandwidth or battery is observed. POSLO−F have equal cryptographic payload compared to SchnorrQ, while having 3× faster signature generation time. Similarly, POSLO−C have equal bandwidth overhead compared to the most signer-efficient counterpart FI-BAF but with constant and flexible storage at the distiller and CSS sides. POSLO−C is considered the best scheme to offer both low bandwidth overhead and fast signature generation on the signer side.

The sign-aggregate-forward approach can be adopted in a hop-by-hop setting, wherein each IoT device signs a set of log entries, aggregates the individual signatures, and forwards the resulting tag to the next IoT device. Another possible design is to employ a clustering approach [17] wherein IoT devices elect a cluster leader to communicate authenticated log entries to the distiller. The leader adjusts the cryptographic payload based on network conditions. For instance, for a set of 2^{10} loggers and 2^8 of epoch size, the bandwidth overhead for a maximum compression across multiple signers is 16.03KB, which is 3× and 171× smaller than single-signer aggregate and non-aggregate approaches.

6.4 Performance Analysis on Edge Cloud (Distiller)

6.4.1 Analytical Evaluation. The edge cloud (or distiller) performs the signature verification on the received log entries. While POSLO−C enjoys an efficient batch verification with one single expensive EC scalar multiplication per epoch of size n_2, POSLO−F validates each individual log entry-signature pair individually and ensure a fine-grained auditing on the cloud storage server. Therefore, POSLO schemes offer different granularity levels depending on the application context. We further add a set of *umbrella* signatures for the valid set to avoid binary epoch verification and bit-flipping events. The verification of POSLO schemes is more computationally efficient compared to the pairing-based BLS by replacing expensive map-to-point and pairing operations with hashing and EC scalar multiplication, respectively. Moreover, the distillation process for POSLO schemes only requires modular addition and EC point addition, which are computationally efficient.

6.4.2 Experimental Evaluation on x86/64. The distiller storage overhead is more cumbersome than the cold storage server, especially when the hardware is not a resourceful device (e.g., hotspot). The latter receives sets of authenticated logs, from a large number of IoT devices, to be verified and aggregated following the pre-determined policy. Therefore, the authentication mechanism must have a low-cost verification algorithm and a flexible aggregation capability. By introducing the valid and invalid cryptographic sets, along with the umbrella valid signatures, the distiller can adjust the signature sizes depending on its resource capabilities and/or the network conditions at

Table 4. Bandwidth overhead and signature generation time of POSLO and its counterparts at signer

Scheme	Analytical complexity	Cryptographic payload (KB)					Signing (s) (per item)
		16	32	64	128	256	
SchnorrQ [11]	$2 \cdot n_2 \cdot \|q\|$	0.5	1	2	4	8	0.323
FI-BAF [55]	$\|q\| + \kappa$	0.05	0.05	0.05	0.05	0.05	0.004
C-RSA [54]	$\|n_{RSA}\|$	0.25	0.25	0.25	0.25	0.25	35.828
BLS [5]	$\|q\|$	0.05	0.05	0.05	0.05	0.05	4.080
POSLO-C	$\|q\| + 2 \cdot \kappa$	0.05	0.05	0.05	0.05	0.05	0.005
POSLO-C$^+$							0.002
POSLO-C^{++}							0.002
POSLO-F	$n_2 \cdot (\|q\| + \kappa)$	0.5	1	2	4	8	0.016
POSLO-F$^+$							0.014
POSLO-F^{++}							0.014

The cryptographic payload is under various epochs (i.e., n_2) to illustrate the impact on signer bandwidth.

the cost of auditing precision. In table 1, the verification time per epoch is performed by the edge cloud. POSLO-C verification is 89× and 26× much faster than the short signature BLS and most signer-efficient counterpart FI-BAF, respectively. While POSLO-F offer a finer granularity, it still outperforms BLS and FI-BAF by a factor of 5.4 and 1.6, respectively.

6.5 Performance Analysis on Cold Storage Server

6.5.1 Analytical Evaluation. Table 5 shows the overall performance of POSLO schemes and their counterparts at the server side. The aggregate signatures offer batch verification for all valid entries (i.e., $\tau = \tau_S$ and $\lambda = 1$) and umbrella signatures. The batch verification of all valid entries requires only one $EMul$, while that of umbrella tags requires $n^u \cdot EMul$ where n^u is the number of umbrella signatures. In terms of storage, the final public key and aggregate signature sizes of POSLO schemes are as efficient as the most compact alternative BLS, but with a faster running time since they do not require expensive pairing (Pr) operation. POSLO schemes have more efficient verification compared to the most signer-efficient counterpart, FI-BAF, which suffer from a linear public key size at CSS.

Table 5. Storage and computation costs of POSLO and their counterparts at cold storage server (CSS)

Scheme	Cold Cryptographic Data (CCD)		Verification Overhead
	$\|PK\|$	$\|\sigma\|$	
SchnorrQ [11]	$\|q\|$	$2 \cdot \tau \cdot n \cdot \|q\|$	$\tau \cdot n \cdot (H + 1.3 \cdot EMul)$
FI-BAF [55]	$2 \cdot \tau \cdot n \cdot (\|q\| + \kappa)$	$\lambda \cdot (\|q\| + \kappa)$	$\tau \cdot n \cdot (2H + 2Add_q + 1.3 \cdot EMul)$ $+\lambda \cdot 1.3 \cdot EMul$
C-RSA [54]	$2\|n_{RSA}\|$	$\lambda \cdot \|n_{RSA}\|$	$\tau \cdot n \cdot (H + Mul_{n_{RSA}}) + \lambda \cdot Exp_{\|n_{RSA}\|}^{\|e_{RSA}\|}$
BLS [5]	$2\|q\|$	$\lambda \cdot \|q\|$	$\tau \cdot n \cdot (MtP + Mul_q) + \lambda \cdot Pr$
POSLO-C	$2\|q\|$	$\lambda \cdot \|q\|$	$\tau \cdot n \cdot (2 \cdot PRF/n_2 + H + Add_q)$ $+\lambda \cdot 1.3 \cdot EMul$
POSLO-F	$\|q\|$	$\lambda \cdot 2\|q\|$	

The notations are as in Table 3. The CCD storage and ver. overhead of valid set (CCD^v), umbrella valid set (CCD^u), and invalid set (CCD^i) are displayed when ($\tau = \tau_S, \lambda = 1$), ($\tau = \tau_S, \lambda = n^u$), and ($\tau = \tau_F, \lambda = \tau_F \cdot n$), respectively.

6.5.2 Experimental Evaluation on x86/64. Figure 8 presents the verification time and the storage overhead for different log entry set sizes (with each entry being 32 bytes) and failure rates τ_F. Recall that τ_F represents the proportion of entries marked as "invalid" during distillation. As discussed in Section 3, in the vast majority of real-world applications, flagged events ("invalid" logs) constitute

only a small fraction of the overall log. Therefore, it is preferable to avoid compressing invalid tags, allowing them to be attested individually.

In the case of full signature aggregation (i.e., $\tau_F = 0$), we refer the reader to Table 1 that summarizes the verification time and storage advantages of our schemes. In Fig. 8, we further investigate the efficiency of compared schemes for varying failure rate (τ_F) and umbrella signature (n^u).

Specifically, we vary $\tau_F = 0, 1, 5\%$ to observe storage overhead and verification time in Fig. 8-(1^{st} row) and Fig. 8-(2^{nd} row), respectively. We increase the size of log entries from 64 GB (2^{31} entries) to 2 TB (2^{36} entries). We crop the y-axis to ensure better visibility of schemes with lower overhead and to prevent schemes having linear storage (e.g., SchnorrQ) and/or computational (e.g., Fi-BAF) from overshadowing the comparison. In our experiments, for large logs, we processed them in small batches and included repeated disk I/O time in our results. We experimented with ρ from $10^{-7}\%$ to 1% and we observed that it has a minimal impact on performance in these margins. Further increase mainly impacts storage with only a slight increase in verification time.

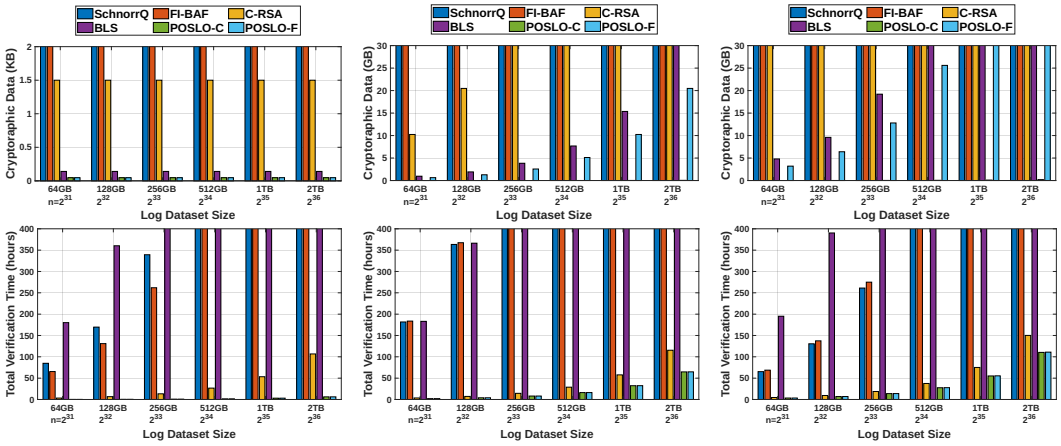
Disk I/O and Cold Storage Cost. Consider a large IoT network where several low-end IoT devices are offloading their authenticated log entries to a remote edge cloud, and ultimately to the cold storage server (CSS). The overall storage at both the edge clouds and CSS becomes exponentially large and costly. Recall that log files are infrequently accessed data, and therefore it is preferred to store them at a cold line solution (e.g., Google cloud[5]), which is relatively low-priced (i.e., \$49.15/year for each TB). However, we argue that POSLO is able to offer the best trade-off between low-cost compact server storage, low disk I/O, and fast verification. According to Table 1, POSLO−C's cryptographic storage overhead is only 0.05KB for 1TB of log entries, whereas it is 2TB for the most signer-efficient counterpart FI-BAF. As a result, POSLO−C have lower disk I/O time and cheaper storage cost since both metrics are directly proportional to the storage overhead. Additionally, POSLO optimizes the disk memory access time by only loading the overall aggregate tag to verify the set of log files. In case the verification fails, the partially condensed signatures are loaded to locate the flagged log. The storage cost at the distiller is more expensive than that of the cold storage server. As the distiller represents the medium between IoT devices and CSS, its stored data is frequently accessed since it receives the authenticated log entries, and distills them after performing the verification. Afterward, it offloads the logs along with the associated cryptographic payload upon finishing a pre-defined set of epochs. This fits the standard storage for data stored within only brief periods of time. Based on the Google cloud solution, the storage cost of one Terabyte is equal to \$245.76/year. Similarly, the disk I/O becomes a key metric at the distiller side.

6.5.3 Experimental Evaluation on GPUs.

In the following, we describe the implementation details and optimization techniques used to achieve an improved speedup compared to CPU version.

Finite Field Arithmetic with PTX Instructions. The proposed digital signature, POSLO, is instantiated with elliptic curves. The basic operations are performed in $\mathbb{Z}_q^*$, where q is a 256-bit prime. A number $a \in \mathbb{Z}_q^*$ can be decomposed into 4 64-bit words $\{a_i\}_{i=1}^4$. We use PTX-based arithmetic over 64-bit words, halving the number of PTX instructions per operation compared to the 32-bit approach in [23]. Multi-precision addition and subtraction of two 256-bit numbers are performed using PTX 64-bit instructions, with carry and borrow propagated using the single carry flag bit, CC.CF. The implementation of Add_q is illustrated as follows:

[5]https://cloud.google.com/storage



(a) $\tau_F = 0\%$ (b) $\tau_F = 1\%$ (c) $\tau_F = 5\%$

Fig. 8. Storage and verification time (on x86/64) comparison of POSLO schemes and their counterparts at the cold storage server (CSS) under different failure rates ($|m|$=256-bit, n_2=2^8)

```
__device__ uint64_t add(uint64_t a[4], uint64_t b[4], uint64_t r[4]) {
    uint64_t carry;
    asm volatile("add.cc.u64 %0,%1,%2;" :"=l"(r[0]):"l"(a[0]),"l"(b[0]));
    asm volatile("addc.cc.u64 %0,%1,%2;" :"=l"(r[1]):"l"(a[1]),"l"(b[1]));
    asm volatile("addc.cc.u64 %0,%1,%2;" :"=l"(r[2]):"l"(a[2]),"l"(b[2]));
    asm volatile("addc.cc.u64 %0,%1,%2;" :"=l"(r[3]):"l"(a[3]),"l"(b[3]));
    asm volatile("addc.u64 %0, 0, 0;" :"=l"(carry));
    return carry;
}
```

AES with T-Table Approach. The primary computational overhead of POSLO signing and verification are hash function (H) and pseudo-random function (PRF) invocations. To optimize signing on resource-constrained devices, we instantiate H and PRF using AES-based constructions for POSLO+. It also provides an advantage on verification for distiller and CSS due to hardware-accelerated implementations on commodity hardware (e.g., x86/64) and GPU architectures. Numerous hardware-accelerated GPU-based techniques (e.g., T-table approach [48], bitsliced approach [18]) have been proposed for AES block cipher on GPUs. We adopt the T-table approach by storing only a single T-table in shared memory. In [48], 32 copies of the T-table are used to avoid bank conflicts, but this creates warp divergence and consumes 32KB of shared memory. The high shared memory usage and warp divergence hinder its deployment as a building block in digital signature primitives (e.g., POSLO), which rely on shared memory to store input messages, auxiliary data, and output results. Moreover, prior GPU-based AES implementations target encryption and decryption applications, where the user key is not updated, thus, the round key is precomputed before kernel launch. In contrast, the MMO and MDC-2 constructions using AES-128 execute the key expansion for each 128-bit chunk of the input data. Therefore, we implemented the AES key expansion algorithm on the device, using PTX instructions similar to the cipher computation. Copying 32 duplicates (as in [48]) of T-tables of both key expansion and cipher algorithms would exceed the shared memory limit (74KB out of a maximum of 64KB). In our implementation, the total shared memory usage for AES computations in the CUDA kernel is only 2.3KB per block, which includes two T-tables, the S-box table, and the round constant word array, while the round keys are stored in the register space.

Inline Macro and Loop Unrolling. (i) Device functions called from the main kernel introduce additional overhead due to the need for the program to jump to the function code. To mitigate this, CUDA provides the __forceinline__ macro, which forces the compiler to inline device functions. This optimization technique reduces the function call overhead, ultimately lowering the overall verification time. *(ii)* Loop unrolling enhances performance by increasing register usage and eliminating loop control overhead. However, when loops already consume a large number of registers, unrolling may lead to register saturation, reducing verification efficiency. To balance performance and resource constraints, we apply loop unrolling selectively for finite field operations but omit it in the implementation of AES-128 and SHA-256.

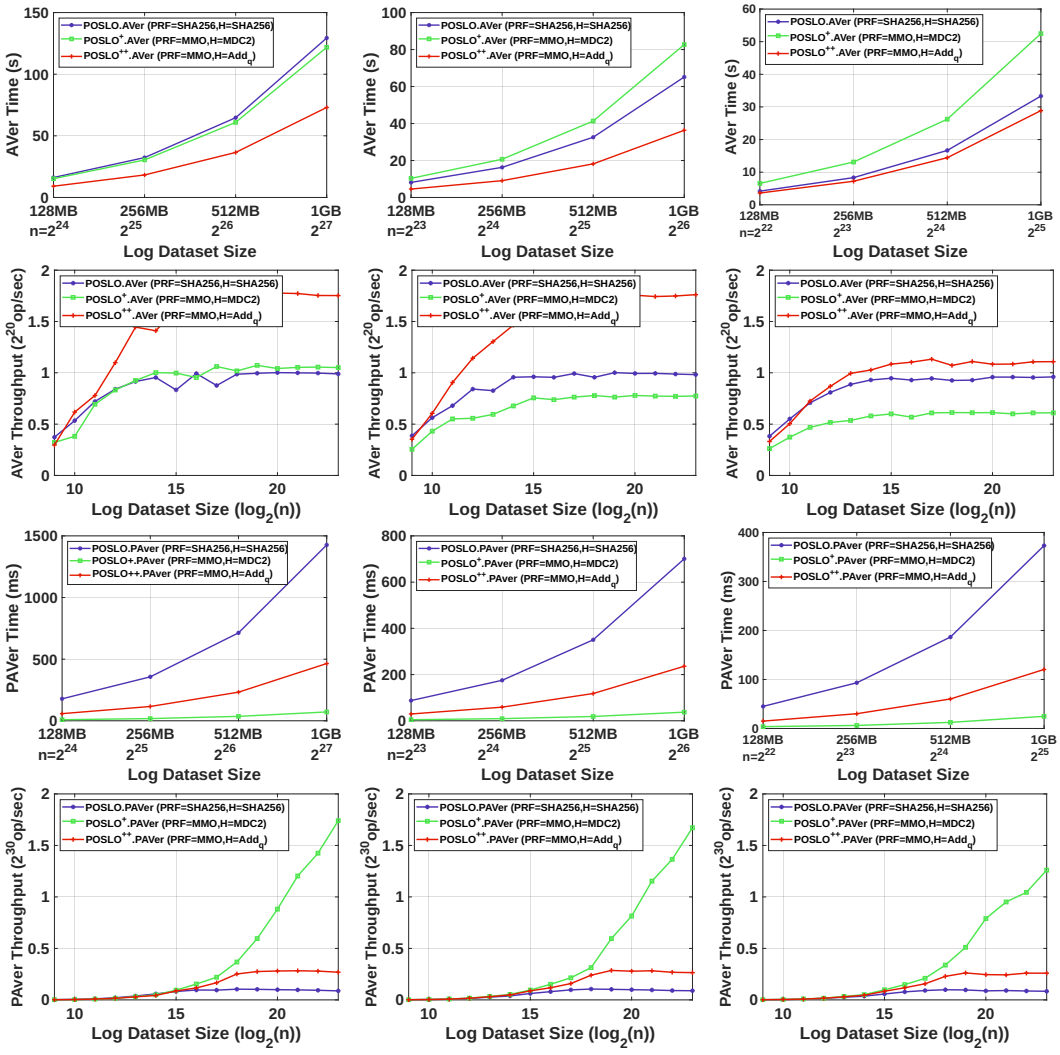
Table 6. Verification peak throughput and speedup of AVer and PAVer for POSLO and counterparts

Scheme	$\|m\|$ = 64-bit			$\|m\|$ = 128-bit			$\|m\|$ = 256-bit		
	AVer(x86/64)	PAVer(GPU)	AVer / PAVer Ratio	AVer(x86/64)	PAVer(GPU)	AVer / PAVer Ratio	AVer(x86/64)	PAVer(GPU)	AVer / PAVer Ratio
SchnorrQ [11]	3.4×10^{-5}	0.0037	108.82	3.4×10^{-5}	0.0037	108.82	3.4×10^{-5}	0.0037	108.82
BLS [5]	1.23×10^{-5}	N/A	N/A	1.23×10^{-5}	N/A	N/A	1.23×10^{-5}	N/A	N/A
POSLO	0.99	89.83	90.83	0.98	91.33	93.20	0.96	85.76	89.33
POSLO^{+}	1.05	1782.13	1697.27	0.77	1711.60	2222.86	0.6	1290.32	2150.53
POSLO^{++}	1.75	275.62	157.50	1.76	271.23	154.21	1.1	265.68	241.53

AVer and PAVer denote the throughput of 2^{20} log entry verifications per second. No GPU implementation is available for BLS.

• *Performance Results*: Figure 9 and Table 6 highlight the verification efficiency on the GPU of POSLO variants and counterparts, compared to the CPU-bounded x86/64 baseline implementation. Our evaluation is performed across different input and batch sizes, with three different combinations of H and PRF (as detailed in Table 2). POSLO+, uses AES-based MMO and MDC-2 constructions, yields superior verification throughput up to $\approx 2^{31}$ log entry verification per second and consists of $2,222\times$ speedup over x86/64 for 128-bit messages. This is attributed to the GPU optimizations, specifically leveraging the T-Table AES optimization and PTX instructions. Considering a large log data (e.g., 1TB), POSLO+ batch verification time is only 24.8 seconds, including the disk I/O overhead. In contrast, POSLO, instantiated with SHA256 for both H and PRF, offers stronger security against collision attacks but achieves a $18\times$ lesser throughput compared to POSLO^{+}. POSLO^{++} variant, which uses modular arithmetic hashing for small inputs, achieves a throughput of $\approx 2^{28}$ log entry verification per second, which is $2.97\times$ and $6.3\times$ faster and slower than POSLO and AES-based POSLO^{+}, respectively. However, POSLO^{++} outperforms both POSLO and POSLO^{+} in terms of verification time on x86/64 architecture due to the low overhead of modular addition compared to sequential SHA-256 and MDC-2 with AES-128. Therefore, POSLO^{++} is the best candidate when efficient distillation and batch verification on x86/64 is desirable but only for small input sizes, whereas POSLO^{+} supports large input sizes and is the best candidate for both low-end devices and Cold-StaaS, especially when GPU acceleration is available at the Cold-StaaS.

POSLO instantiations performs several orders of magnitude better compared to the baseline non-aggregate SchnorrQ signature scheme, which uses identical EC-based operations. Note that our peak performance is observed on 1GB log data. The verification of larger log data (e.g., 1TB) can be performed via (1) sequential execution of POSLO.Agg-eKeys per 1GB chunk to obtain the sub-aggregate of ephemeral keys (i.e., $\tilde{e}$). (2) running in parallel POSLO.Agg-eKeys on 1GB chunks across multiple GPUs, to further reduce the audit time. Moreover, the umbrella signature verification can be performed simultaneously during the final aggregation of sub-aggregate ephemeral keys. Overall, our performance results confirm the efficiency of our parallel POSLO design that harnesses the epoch processing and mutual aggregation property, and further improves on the AES-based variant using T-Table implementation. This reaffirms that POSLO^{+} is the most suitable candidate for the IoT-Cloud continuum by achieving the lowest energy consumption and the highest verification throughput on low-end IoT devices and cold storage servers, respectively.



(a) $|m| = 64$-bit (b) $|m| = 128$-bit (c) $|m| = 256$-bit

Fig. 9. Performance of (parallel) batch verification (P)AVer of POSLO instantiations on x86/64 (POSLO.AVer: 1st-2nd rows) and GPU (POSLO.PAVer: 3rd-4th rows)

7 CONCLUSION

In this work, we introduced a new family of aggregate signatures, called POSLO, designed for secure logging in resource-constrained IoT networks. To the best of our knowledge, POSLO achieves the strongest trade-off among provable security guarantees, computational efficiency, and cryptographic storage compactness, surpassing existing secure logging schemes across diverse performance metrics.

At its core, POSLO incorporates a novel tree-based seed management strategy and a post-signature disclosure of one-time separated commitments. This combination enables a highly compact cryptographic state while supporting secure and efficient signing and verification. To preserve fine-grained verifiability, POSLO introduces a tunable granularity parameter that allows selective retention of compact tags after signature distillation. Beyond the traditional CPU-bound implementations, we

developed POSLO.PAVer, a GPU-accelerated batch verification algorithm that exploits the homomorphic and decoupled-commitment structure of POSLO to deliver ultra-efficient throughput. POSLO.PAVer achieves up to 2^{31} log verifications per second, and demonstrates multiple orders of magnitude speedup over GPU-accelerated non-aggregate EC-based SchnorrQ (as our baseline).

Our comprehensive experimental evaluation on resource-constrained devices, general-purpose CPUs, and GPU platforms confirms that POSLO and its variants offer strong configurability, exceptional efficiency, and minimal cryptographic storage. We further implement three optimized instantiations—POSLO, POSLO^{+}, and POSLO^{++}, each leveraging different cryptographic primitives (e.g., hash-based, AES-based, or modular arithmetic) to provide flexible performance-security trade-offs. We formally prove that POSLO achieves A-EU-CMA security in the random oracle model. Our full-fledged implementation is open source and publicly available to support reproducibility, third-party evaluation, and future deployment.

ACKNOWLEDGMENTS

This research is partially supported by the National Science Foundation (NSF) grant CNS-2350213.

REFERENCES

[1] Adil Ahmad, Sangho Lee, and Marcus Peinado. 2022. Hardlog: Practical tamper-proof system auditing using a novel audit device. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1791–1807.

[2] Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, Clément Pernet, Michael Hanling, and Daniel S Roche. 2021. Dynamic proofs of retrievability with low server storage. In *30th USENIX Sec. Symp.* 537–554.

[3] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. 2008. Scalable and efficient provable data possession. In *Proc. of the 4th international conference on Security and privacy in communication netowrks*. 1–10.

[4] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of Cryptographic Engineering* 2, 2 (01 Sep 2012), 77–89.

[5] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short Signatures from the Weil Pairing. *J. Cryptol.* 17, 4 (2004), 297–319.

[6] Victor Boyko, Marcus Peinado, and Ramarathnam Venkatesan. 1998. Speeding up Discrete Log and Factoring Based Schemes via Precomputations. In *EUROCRYPT '98* (eurocrypt '98 ed.). 221–235.

[7] Rohit Chandra. 2001. *Parallel programming in OpenMP*. Morgan kaufmann.

[8] Changhua Chen, Tingzhen Yan, Chenxuan Shi, Hao Xi, Zhirui Fan, Hai Wan, and Xibin Zhao. 2024. The Last Mile of Attack Investigation: Audit Log Analysis towards Software Vulnerability Location. *IEEE Transactions on Information Forensics and Security* (2024).

[9] Yilei Chen, Alex Lombardi, Fermi Ma, and Willy Quach. 2021. Does Fiat-Shamir require a cryptographic hash function?. In *Annual International Cryptology Conference*. Springer, 334–363.

[10] Mucong Chi, Jun Liu, and Jie Yang. 2020. ColdStore: a storage system for archival data. *Wireless Personal Communications* 111, 4 (2020), 2325–2351.

[11] Craig Costello and Patrick Longa. 2016. Schnorrq: Schnorr signatures on fourq. *MSR Tech Report* (2016).

[12] Jiankuo Dong, Fangyu Zheng, Niall Emmart, Jingqiang Lin, and Charles Weems. 2018. sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 599–609.

[13] Zonghao Feng, Qipeng Xie, Qiong Luo, Yujie Chen, Haoxuan Li, Huizhong Li, and Qiang Yan. 2022. Accelerating elliptic curve digital signature algorithms on GPUs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[14] Anna Lisa Ferrara, Matthew Green, Susan Hohenberger, and Michael Østergaard Pedersen. 2009. Practical short signature batch verification. In *Cryptographers' Track at the RSA Conference*. Springer, 309–324.

[15] Benjamin Glas, Jorge Guajardo, Hamit Hacioglu, Markus Ihle, Karsten Wehefritz, and Attila A. Yavuz. 2012. Signal-based Automotive Communication Security and Its Interplay with Safety Requirements. ESCAR, Embedded Security in Cars Conference, Germany, November 2012.

[16] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc of the 13th ACM conference on Computer and communications security*. 89–98.

[17] Mohamed Grissa, Attila A Yavuz, and Bechir Hamdaoui. 2019. TrustSAS: A trustworthy spectrum access system for the 3.5 GHz CBRS band. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1495–1503.

[18] Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin. 2019. Fast AES implementation: A high-throughput bitsliced approach. *IEEE Transactions on parallel and distributed systems* 30, 10 (2019), 2211–2222.

[19] Gunnar Hartung. 2016. Secure Audit Logs with Verifiable Excerpts. In *Topics in Cryptology - CT-RSA 2016*, Kazue Sako (Ed.). Springer International Publishing, Cham, 183–199.

[20] Gunnar Hartung. 2017. Attacks on Secure Logging Schemes. In *Financial Cryptography and Data Security*. Springer International Publishing, Cham, 268–284.

[21] Gael Hofemeier and Robert Chesebrough. 2012. Introduction to intel aes-ni and intel secure key instructions. *Intel, White Paper* 62 (2012), 6.

[22] John E Hopcroft, Jeffrey D Ullman, and Alfred Vaino Aho. 1983. *Data structures and algorithms*. Vol. 175. Addison-wesley Boston, MA, USA:.

[23] Xinyi Hu, Debiao He, Min Luo, Cong Peng, Qi Feng, and Xinyi Huang. 2023. High-performance implementation of the identity-based signature scheme in IEEE P1363 on GPU. *ACM Transactions on Embedded Computing Systems* 22, 2 (2023), 1–35.

[24] DongCheon Kim, HoJin Choi, and Seog Chung Seo. 2024. Parallel Implementation of SPHINCS + With GPUs. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2024).

[25] Jihye Kim and Hyunok Oh. 2019. FAS: Forward secure sequential aggregate signatures for secure logging. *Information Sciences* 471 (2019), 115 – 131.

[26] David Kirk et al. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, Vol. 7. 103–104.

[27] Sokjoon Lee, Hwajeong Seo, Hyeokchan Kwon, and Hyunsoo Yoon. 2019. Hybrid approach of parallel implementation on CPU–GPU for high-speed ECDSA verification. *The Journal of Supercomputing* 75 (2019), 4329–4349.

[28] Tian Li, Huaqun Wang, Debiao He, and Jia Yu. 2020. Permissioned blockchain-based anonymous and traceable aggregate signature scheme for Industrial Internet of Things. *IEEE Internet of Things Journal* 8, 10 (2020), 8387–8398.

[29] Xin Li, Huazhe Wang, Ye Yu, and Chen Qian. 2017. An IoT data communication framework for authenticity and integrity. In *2017 IEEE/ACM 2nd International Conf. on Internet-of-Things Design and Implementation (IoTDI)*. 159–170.

[30] Zhenyuan Li, Qi Alfred Chen, Runqing Yang, Yan Chen, and Wei Ruan. 2021. Threat detection and investigation with system-level provenance graphs: A survey. *Computers & Security* 106 (2021), 102282.

[31] Wenhao Liao, Jia Sun, Haiyan Wang, Zhaoquan Gu, and Jianye Yang. 2024. Semantic-Integrated Online Audit Log Reduction for Efficient Forensic Analysis. In *International Conf. on Advanced Data Mining and Applications*. 318–333.

[32] Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. 2010. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Workshop on the Security of the Internet of Things-SOCIOT*, Vol. 10.

[33] Di Ma and Gene Tsudik. 2009. A New Approach to Secure Logging. *Trans. Storage* 5, 1, Article 2 (2009), 21 pages.

[34] Giorgia Azzurra Marson and Bertram Poettering. 2014. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *Computer Security - ESORICS 2014*. Cham, 37–54.

[35] A.J. Menezes, P. C. van Oorschot, and S.A. Vanstone. 1996. *Handbook of Applied Cryptography*. CRC Press.

[36] Roberto Minerva, Gyu Myoung Lee, and Noel Crespi. 2020. Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models. *Proc. IEEE* 108, 10 (2020), 1785–1824.

[37] MITRE. [n. d.]. Indicator Removal: Clear Linux or Mac System Logs . https://attack.mitre.org/techniques/T1070/002/. Accessed: April 5, 2025.

[38] Arsalan Mosenia and Niraj K Jha. 2016. A comprehensive study of security of internet-of-things. *IEEE Transactions on emerging topics in computing* 5, 4 (2016), 586–602.

[39] Saif E Nouma and Attila A Yavuz. 2023. Practical Cryptographic Forensic Tools for Lightweight Internet of Things and Cold Storage Systems. In *Proc. of the 8th ACM/IEEE Conf. on Internet of Things Design and Implementation*. 340–353.

[40] Muslum Ozgur Ozmen, Rouzbeh Behnia, and Attila A. Yavuz. 2019. Energy-Aware Digital Signatures for Embedded Medical Devices. In *7th IEEE Conf. on Communications and Network Security (CNS), June*.

[41] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. 2016. An efficient elliptic curve cryptography signature server with GPU acceleration. *IEEE Trans. on Information Forensics and Security* 12, 1 (2016), 111–122.

[42] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. 2008. Fast hash-based signatures on constrained devices. In *Smart Card Research and Advanced Applications: 8th IFIP WG 8.8/11.2 International Conference, CARDIS 2008, London, UK, September 8-11, 2008. Proceedings 8*. Springer, 104–117.

[43] Tinshu Sasi, Arash Habibi Lashkari, Rongxing Lu, Pulei Xiong, and Shahrear Iqbal. 2024. A comprehensive survey on IoT attacks: Taxonomy, detection mechanisms and challenges. *J. of Information and intelligence* 2, 6 (2024), 455–513.

[44] B. Schneier and J. Kelsey. 1999. Secure audit logs to support computer forensics. *ACM Transaction on Information System Security* 2, 2 (1999), 159–176.

[45] Claus-Peter Schnorr. 1991. Efficient signature generation by smart cards. *Journal of cryptology* 4, 3 (1991), 161–174.

[46] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. 2019. Analyzing the impact of {GDPR} on storage systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems*.

[47] John P Steinberger. 2007. The collision intractability of MDC-2 in the ideal-cipher model. In *Advances in Cryptology-EUROCRYPT: 26th Annual International Conf. on the Theory and Applications of Cryptographic Techniques*. 34–51.
[48] Cihangir Tezcan. 2021. Optimization of advanced encryption standard on graphics processing units. *IEEE Access* 9 (2021), 67315–67326.
[49] Thokozani F. Vallent, Damien Hanyurwimfura, and Chomora Mikeka. 2021. Efficient certificate-less aggregate signature scheme with conditional privacy-preservation for vehicular adhoc networks enhanced smart grid system. *Sensors* 21, 9 (2021).
[50] Girraj Kumar Verma, Neeraj Kumar, Prosanta Gope, BB Singh, and Harendra Singh. 2021. SCBS: a short certificate-based signature scheme with efficient aggregation for industrial-internet-of-things environment. *IEEE Internet of Things Journal* 8, 11 (2021), 9305–9316.
[51] Cong Wang, Ning Cao, Jin Li, Kui Ren, and Wenjing Lou. 2010. Secure ranked keyword search over encrypted cloud data. In *2010 IEEE 30th international conference on distributed computing systems*. 253–262.
[52] Cong Wang, Sherman SM Chow, Qian Wang, Kui Ren, and Wenjing Lou. 2011. Privacy-preserving public auditing for secure cloud storage. *IEEE transactions on computers* 62, 2 (2011), 362–375.
[53] Attila A. Yavuz. [n. d.]. System and method for secure review of audit logs. Robert Bosch, Provisional Application No. 62/006,476, Filing Date: June 2, 2014, PCT Application: June 2, 2015.
[54] Attila A. Yavuz. 2018. Immutable Authentication and Integrity Schemes for Outsourced Databases. *IEEE Trans. Dependable Sec. Comput.* 15, 1 (2018), 69–82.
[55] A. A. Yavuz, Peng Ning, and Michael K. Reiter. 2012. BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems. *ACM Trans. on Inf. System Sec.* 15, 2 (2012), 28 pages.

APPENDIX A

We provide the security proof of POSLO−C scheme as below:

THEOREM 5.1. $Adv^{A-EU-CMA}_{POSLO-C(p,q,\alpha)}(t, n', n) \leq Adv^{DL}_{\mathbb{G},\alpha}(t')$, *where* $t' = O(t) + O(n \cdot (\kappa^3 + RNG))$.

Proof: Let $\mathcal{A}$ be a POSLO−C attacker. We construct a *DL-attacker* $\mathcal{F}$ that uses $\mathcal{A}$ as a sub-routine. That is, we set ($b \xleftarrow{\$} \mathbb{Z}^*_q$, $B \leftarrow \alpha^b \bmod p$) as defined in *DL*-experiment (i.e., Definition 2.4) and then run the simulator $\mathcal{F}$ by Definition 3.1 (i.e., A−EU−CMA experiment) as follows:

Algorithm $F(B)$

Setup: $\mathcal{F}$ maintains $\mathcal{LH}$, $\mathcal{LM}$, and $\mathcal{LS}$ to keep track of query results in the duration of the experiment. $\mathcal{LH}$ is a hash list in form of tuples (M_l, h_l), where M_l and h_l denote the l^{th} data item queried to $RO(.)$ and its corresponding $RO(.)$ answer, respectively. $\mathcal{LH}[l, 0]$ and $\mathcal{LH}[l, 1]$ denote the access to the element M_l, h_l via the hash function H, respectively. $\mathcal{LM}$ is a list of messages, in which each of its elements $\mathcal{LM}[i]$ is a message set $\boldsymbol{m}_i$ (i.e., the i^{th} batch query). $\mathcal{LS}$ is a signature list that is used to record answers given by POSLO−C.Sig_{sk}.

- $\mathcal{F}$ creates a simulated POSLO−C public key PK:
  a) Set (n_1, n_2) as in POSLO−C.Kg(.)
  b) $Y \leftarrow B$ and $x_D[0] \xleftarrow{\$} \{0,1\}^\kappa$
  c) **for** $l = 1, \ldots, n$ **do**
     i) $R_l \leftarrow Y^{e_l} \cdot \alpha^{s_l} \bmod p$ where $(s_l, e_l) \xleftarrow{\$} \mathbb{Z}^*_q$
  d) **for** $i = 1, \ldots, n_1$ **do**
     i) $\tilde{R}_i \leftarrow \prod_{j=1}^{n_2} R_{i \cdot n_1 + j} \bmod p$
  e) $PK \leftarrow (Y, \boldsymbol{R})$, where $\boldsymbol{R} \leftarrow \{\tilde{R}_i\}_{i=1}^{n_1}$
  f) $I \leftarrow (p, q, \alpha, n_1, n_2)$ and initialize $l \leftarrow 0$, $i \leftarrow 0$

Execute $\mathcal{A}^{RO(.),\text{POSLO−C.Sig}_{sk}(.)}(PK)$:

- Queries: $\mathcal{A}$ queries POSLO−C.$\text{Sig}_{sk}(.)$ oracle on n messages of her choice. It also queries $RO(.)$ oracle on up to n' messages of her choice. These queries are as follows:

• *How to Handle RO(.) Queries*: $\mathcal{F}$ implements a function *H-Sim*(δ, k) that works as *RO*(.) as follows: If $\exists l' : \delta \in \mathcal{LH}[l']$ then return $\mathcal{LH}[l']$. Otherwise, return $h \xleftarrow{\$} \mathbb{Z}_q^*$ as the answer for H, insert new tuple (δ, h) to $\mathcal{LH}$ as $(\mathcal{LH}[l,0] \leftarrow \delta, \mathcal{LH}[l,1] \leftarrow h)$ and then update $l \leftarrow l+1$. That is, the cryptographic hash function H used in `POSLO-C` is modeled as random oracle.
When $\mathcal{A}$ queries *RO*(.) on a message m_l, $\mathcal{F}$ returns $h_l \leftarrow$ *H-Sim*(m_l) as described above.
• *How to respond to* `POSLO-C.Sig`$_{sk}$(.) *queries:*
- For each batch query $\boldsymbol{m_i}$, $\mathcal{A}$ queries `POSLO-C.Sig`(.) on $\boldsymbol{m_i}$ of her choice. If $i \geq n_1$, $\mathcal{F}$ rejects the query (i.e., the query limit is exceeded), else $\mathcal{F}$ continues as follows:
  a) $\mathcal{F}$ computes $x_0[i] \leftarrow \texttt{SC}(x_D[0], D, 0, 0, i)$
  b) Initialize $\tilde{s}_i \leftarrow 0$
  c) **for** $j = 1, \ldots, n_2$ **do**
    i) $\mathcal{F}$ sets $x_i^j \leftarrow \texttt{PRF}_0(x_0[i] \| j)$. If $(m_i^j \| x_i^j) \in \mathcal{LH}$ then $\mathcal{F}$ *aborts*, else inserts $(m_i^j \| x_i^j)$ to $\mathcal{LH}$.
    ii) $\mathcal{F}$ computes $\tilde{s}_i \leftarrow \texttt{POSLO-C.Agg}(\tilde{s}_i, s_i^j)$
  d) $\mathcal{F}$ sets $\tilde{\sigma}_i \leftarrow (\tilde{s}_i, ds_i \leftarrow \texttt{SO}(x_0[0], i))$, inserts $(\boldsymbol{m_i}, \tilde{\sigma}_i)$ to $(\mathcal{LM}, \mathcal{LS})$ and $i \leftarrow i+1$.
- Forgery of $\mathcal{A}$: Eventually, $\mathcal{A}$ outputs a forgery on *PK* as $(\vec{\boldsymbol{m}}^*, \tilde{\sigma}^*)$, where $\vec{\boldsymbol{m}}^* = \{\boldsymbol{m_i^*}\}_{i \in \boldsymbol{I}}$ and $\tilde{\sigma}^* = (\tilde{s}^*, ds^*)$. By Definition 3.1, $\mathcal{A}$ wins `A-EU-CMA`-experiment if `POSLO-C.AVer`$(PK, \vec{\boldsymbol{m}}^*, \tilde{\sigma}^*) = 1$ and $\vec{\boldsymbol{m}}^* \notin \mathcal{LM}$ hold. If these conditions hold, $\mathcal{A}$ returns 1, else, returns 0.

- Forgery of $\mathcal{F}$: If $\mathcal{A}$ loses the `A-EU-CMA` experiment for `POSLO-C`, $\mathcal{F}$ also loses in the *DL*-experiment, and therefore $\mathcal{F}$ *aborts* and returns 0. Else, if $\boldsymbol{m^*} \in \mathcal{LH}$ then $\mathcal{F}$ *aborts* and returns 0 (i.e., $\mathcal{A}$ wins the experiment without querying *RO*(.) oracle). Otherwise, $\mathcal{F}$ continues:

$\tilde{R} \equiv Y^{\tilde{e}} \cdot \alpha^{\tilde{s}} \bmod p$ holds for the aggregated variables $(\tilde{R}, \tilde{e}, \tilde{s})$. That is, given the indices of corresponding previous messages $\boldsymbol{I}$, $\mathcal{F}$ retrieves (s_i, r_i) from $(\mathcal{LS}, \mathcal{LH})$, and then computes $\tilde{e} \leftarrow \sum_{i \in \boldsymbol{I}} \sum_{j=1}^{n_2} e_{i \cdot n_1 + j} \bmod q$ and $\tilde{s} \leftarrow \texttt{POSLO-C.Agg}(\{\tilde{s}_i\}_{i \in \boldsymbol{I}})$. Moreover, `POSLO-C.AVer`$(PK, \boldsymbol{m^*}, \sigma^*) = 1$ holds, and therefore $R \equiv Y^{e^*} \cdot \alpha^{s^*} \bmod p$ also holds. Note that $\mathcal{A}$ queries $\mathcal{F}$ on n_1 batches and n messages in total. Hence, $\mathcal{F}$ disclosed the root of `OSLOT` tree, from which required seeds can be derived. $\mathcal{F}$ calls $x_0[i] \leftarrow \texttt{SR}(ds^*, i)$, $\forall i \in \boldsymbol{I}$. It then computes $\tilde{e}^* = \sum_{i \in \boldsymbol{I}} \sum_{j=1}^{n_2} H\text{-}Sim(m_i^{j^*} \| x_i^{j^*}, 0)$ where $x_i^j \leftarrow \texttt{PRF}_0(x_0[i] \| j)$. Thus, the following equations hold:

$$\tilde{R} \equiv Y^{\tilde{e}} \cdot \alpha^{\tilde{s}} \bmod p, \quad \tilde{R} \equiv Y^{\tilde{e}^*} \cdot \alpha^{\tilde{s}^*} \bmod p,$$

$\mathcal{F}$ then extracts $y' = b$ by solving the below modular linear equations (note that only unknowns are y and r), where $Y = B$ as defined in the public key simulation:

$$r \equiv y' \cdot e + s \bmod q, \; r \equiv y' \cdot e^* + s^* \bmod q$$

$B' \equiv \alpha^b \bmod p$ holds, since $\mathcal{A}$'s forgery is valid and non-trivial on $B' = B$. By Definition 2.4, $\mathcal{F}$ wins the *DL-experiment*.

The execution time and probability analysis are as follows:
Execution Time Analysis: In this experiment, the runtime of $\mathcal{F}$ is that of $\mathcal{A}$ plus the time it takes to respond *RO*(.) queries.

- *Setup phase:* $\mathcal{F}$ draws $2n+1$ random numbers, performs $2n$ modular exponentiations and multiplications. Hence, the total cost of this phase is $2n \cdot O(\kappa^3 + \kappa^2) + (2n+1) \cdot \texttt{RNG}$, where

$O(\kappa^3)$ and $O(\kappa^2)$ denote the cost of modular exponentiation and modular multiplication, respectively. RNG denotes the cost of drawing a random number.

- *Query phase:* $\mathcal{F}$ draws $n_1 \cdot \log n_1 \cdot$ RNG to compute the epoch seeds and $n \cdot$ RNG to derive one-time random keys. It also draws $n \cdot$ RNG to handle $\mathcal{A}$'s $RO(.)$ queries. The cost of query phase is bounded as $O(T) \cdot$ RNG.

Therefore, the approximate total running time of $\mathcal{F}$ is $t' = O(t) + O(n \cdot (\kappa^3 + RNG))$.

Success Probability Analysis: $\mathcal{F}$ succeeds if all below events occur.

- $\overline{E1}$: $\mathcal{F}$ does not abort during the query phase.
- $E2$: $\mathcal{A}$ wins the A−EU−CMA experiment for POSLO−C.
- $\overline{E3}$: $\mathcal{F}$ does not abort after $\mathcal{A}$'s forgery.
- *Win*: $\mathcal{F}$ wins the A−EU−CMA experiment for *DL-experiment*.
- $Pr[Win] = Pr[\overline{E1}] \cdot Pr[E2|\overline{E1}] \cdot Pr[\overline{E3}|\overline{E1} \wedge E2]$

• *The probability that event $\overline{E1}$ occurs*: During the query phase, $\mathcal{F}$ aborts if $(m_i^j||x_i^j) \in \mathcal{LH},\ 1 \leq i \leq n_1,\ 1 \leq j \leq n_2$ holds, *before* $\mathcal{F}$ inserts $(m_i^j\|x_i^j)$ into $\mathcal{LH}$. This occurs if $\mathcal{A}$ guesses x_i^j (before it is released) and then queries $(m_i^j\|x_i^j)$ to $RO(.)$ *before* querying it to POSLO−C.Sig(.). The probability that this occurs is $\frac{1}{2^\kappa}$, which is negligible in terms of κ. Hence, $Pr[\overline{E1}] = (1 - \frac{1}{2^\kappa}) \approx 1$.

• *The probability that event E2 occurs*: If $\mathcal{F}$ does not abort, $\mathcal{A}$ also does not abort since the $\mathcal{A}$'s simulated view is *indistinguishable* from $\mathcal{A}$'s real view (see the indistinguishability analysis). Thus, $Pr[E2|\overline{E1}] = Adv^{\text{A−EU−CMA}}_{\text{POSLO−C}(p,q,\alpha)}(t, n', n)$.

• *The probability that event $\overline{E3}$ occurs*: $\mathcal{F}$ does not abort if the following conditions are satisfied: (i) $\mathcal{A}$ wins the A−EU−CMA experiment for POSLO−C on a message m^* by querying it to $RO(.)$. The probability that $\mathcal{A}$ wins without querying m^* to $RO(.)$ is as difficult as a random guess. (ii) After $\mathcal{F}$ extracts $y' = b$ by solving modular linear equations, the probability that $Y' \not\equiv \alpha^{y'} \bmod p$ is negligible in terms κ, since $(Y = B) \in PK$ and POSLO−C.AVer$(PK, m^*, \sigma^*) = 1$. Hence, $Pr[\overline{E3}|\overline{E1} \wedge E2] = Adv^{\text{A−EU−CMA}}_{\text{POSLO−C}(p,q,\alpha)}(t, n', n)$. Omitting the terms that are negligible in terms of κ, the upper bound on *A−EU−CMA-advantage of POSLO−C* is as follows:

$$Adv^{\text{A−EU−CMA}}_{\text{POSLO−C}(p,q,\alpha)}(t, n', n) \leq Adv^{DL}_{\mathbb{G},\alpha}(t'),$$

Indistinguishability Argument: The real-view of $\overrightarrow{A}_{real}$ is comprised of public key PK, parameters I, the answers of POSLO−C.Sig$_{sk}$(.) and $RO(.)$, recorded by $\mathcal{F}$ in $\mathcal{LS}$ and $\mathcal{LM}$, respectively. These values are generated by POSLO−C algorithms as in the real system, where $sk = (y, r, x_D[0])$ serves as initial randomness. The joint probability distribution of $\overrightarrow{A}_{real}$ is random uniform as that of sk.

The simulated view of $\mathcal{A}$ is as $\overrightarrow{A}_{sim}$, and it is equivalent to $\overrightarrow{A}_{real}$ except that in the simulation, values (s_l, e_l) for $l = 1, \ldots, n$ are randomly selected from $\mathbb{Z}_q^*$. This then dictates the selection of R_l for $l = 1, \ldots, n$ as random via the public key simulation (step c)-ii). Note that the joint probability distribution of these variables is also random uniformly distributed and is identical to the original signature and hash outputs (since H is modeled as $RO(.)$ via *H-Sim*). POSLO−C.Distill(.) and POSLO−C.SeBVer(.) use POSLO−C.Agg(.) and POSLO−C.AVer(.), which are invoked in signature simulation and forgery/extraction phases. Since *CCD* only contains the values produced in the simulation, $\overrightarrow{A}_{sim}$ for POSLO−C.Distill(.) and POSLO−C.SeBVer(.) are indistinguishable from that of $\overrightarrow{A}_{real}$. □

We provide the security proof of POSLO−F scheme as below:

LEMMA 5.2. *POSLO−F is as secure as POSLO−C.*

Proof: In the sketch proof, we first show that POSLO−F public key and signature simulations produce random uniformly distributed values as in POSLO−C. We then show that the forgery and extraction phases in A−EU−CMA experiment for both variants are identical. Finally, we provide an indistinguishability argument for the A−EU−CMA for POSLO−F.

• *Public Key Simulation*: POSLO−F.Kg(.) Step 1-4 are identical to that of POSLO−C, except commitment values R are generated via BPV generator. Therefore, $\mathcal{F}$ runs the public key simulation as in POSLO−C, expect $\boldsymbol{R}$ is not pre-stored as a part of the public key. All the values $\{s_l, e_l, R_l\}_{l=1}^{n}$ are as in the POSLO−C simulation.

• *Signature Simulation*: $\mathcal{F}$ sets $\sigma_l = (s_l, R_l, x_l)$, where (s_l, R_l) are as defined above, and (e_l, x_l) are obtained through $RO(.)$ as in POSLO−C via *H-Sim* function. POSLO−F.Sig(.) queries are individual, and therefore σ_l is not aggregated via POSLO−C.Agg(.). The abort conditions in both POSLO−C and POSLO−F are the same.

• *Forgery and Extraction*: POSLO−C and POSLO−F verifications are identical except for the first step, which identifies if the signature is on a single or batch of messages. If the forgery is an aggregate signature on a batch message, POSLO−F.AVer(.) verifies it by performing aggregation as in POSLO−C.AVer(.). Hence, the forgery and extraction are identical, wherein $\mathcal{A}$ might return a batch or individual forgery (σ^*, M^*). $\mathcal{F}$ retrieves (s, R, e) from $\mathcal{LS}$ since R components are the part of signatures but not *PK* (unlike POSLO−C).

• *Indistinguishability Argument*: $\overrightarrow{A}_{real}$ of POSLO−F is as in POSLO−C except that $\{R_\ell\}_{l=1}^{n}$ (generated via BPV) are not part of *PK* but in individual signatures $\{\sigma_l = (s_l, R_l, ds_l)\}_{l=1}^{n}$. The joint probability distribution of the values in $\overrightarrow{A}_{real}$ are random uniformly distributed as all derived from sk (as in POSLO−C). Remark that each R_l is also random uniform because the distribution of BPV output r_l is statistically close to the uniform random distribution with an appropriate choice of parameters (v, k) [6]. $\overrightarrow{A}_{sim}$ is identical to $\overrightarrow{A}_{real}$ since public key and signature simulations produce random uniformly distributed values of equal size to $\overrightarrow{A}_{real}$. As in POSLO−C, POSLO−F.Distill(.) and POSLO−F.SeBVer(.) call POSLO−F.Agg(.) and POSLO−F.AVer(.), in which *CCD* values are produced by POSLO−F.Sig(.) and *H-Sim*. □

COROLLARY 5.3. *POSLO$^+$ and POSLO^{++} instantiations are as secure as POSLO.*

PROOF. In POSLO$^+$, the cryptographic hash function (SHA-256) used in POSLO is replaced with block-cipher-based constructions: MMO for the PRF primitive and MDC-2 for message hashing H. Their security is based on the pseudorandomness of the underlying block cipher (i.e., AES-128) [35]. MMO's output is indistinguishable from a random string provided that AES-128 is a random permutation. As a key derivation function $\mathrm{PRF}_{0,1}$, AES-based MMO achieves preimage resistance with probability at most $\frac{1}{2^{128}}$ [35]. As for MDC-2, Steinberger et al. [47] shows that the best theoretical collision attack in the ideal cipher model requires $2^{3\ell/5}$ queries, where ℓ is the block size. However, collision attacks still require close to 2^{ℓ} queries ($\ell = 128$ for AES-128), offering comparable security to SHA-256. In POSLO^{++}, the MDC-2-based hash function H is replaced by modular addition over a large prime field $Add_q(x, y) = x + y \mod q$. Chen el al. [9] prove that such a non-cryptographic hash function (i.e., Add_q) is sufficient for Schnorr-based digital signatures despite missing full collision resistance, given that inputs are unpredictable and of size smaller than q (i.e., lesser than 32-byte). Therefore, under standard assumtpions on AES-128 security and following the results of [9], POSLO$^+$ and POSLO^{++} achieve equivalent security to the original POSLO. □