CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Secure Attestation Framework for Intelligent Transportation Systems

Advancing Swarm Attestation Through the Application of Homomorphic Hashing

Master's Thesis in Computer Science and Engineering

Imad Alihodžić & Abirami Rengaraj

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Secure Attestation Framework for Intelligent Transportation Systems

Advancing Swarm Attestation Through the Application of Homomorphic Hashing

Imad Alihodžić & Abirami Rengaraj

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Secure Attestation Framework for Intelligent Transportation Systems
Advancing Swarm Attestation Through the Application of Homomorphic Hashing
Imad Alihodzic & Abirami Rengaraj

Supervisor: Md Masoom Rabbani, Computer and Network Systems
Advisor: Francisco Blas Izquierdo, KITS AB & Wouter Hellemans, KU Leuven
Examiner: Ahmed Ali-Eldin Hassan, Computer and Network Systems

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Secure Attestation Framework for Intelligent Transportation Systems
Advancing Swarm Attestation Through the Application of Homomorphic Hashing
Imad Alihodžić & Abirami Rengaraj
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Internet of Thing (IoT) systems, and in particular intelligent transportation systems, are becoming ever increasingly large and complex. Simultaneously, such systems increasingly coming under attack from malicious parties.

To aid in monitoring, and ensuring, the security of such system, one prominent solution is remote attestation (RA) - a process by which trusted entities can determine, and attest, the integrity of un-trusted devices. Whilst contemporary research has lead to the creation of numerous RA schemes, each have their own shortcomings. To address these shortcomings, this thesis proposes a new attestation scheme - *FLASH: Fast Lightweight Attestation using Scalable Homomorphic Hashing*. As the name suggests, this lightweight attestation scheme utilizes homomorphic hashing to aid in the aggregation of attestation results, reducing computational costs, and enabling efficient scaling across large systems.

FLASH is comprised of three separate algorithms, namely two separate on-demand algorithms, and one self-attestation algorithm, all built using a common homomorphic hashing library. To verify the performance and usability of FLASH, proof of concept implementations of the on-demand algorithms have been developed and tested using real-world hardware, with the timing results then subsequently being applied to large-scale self-attestation simulations.

Furthermore, result analysis of FLASH testing allowed for the evaluation of both worst-case behavior for both the PoC and simulation, as well as the creation of performance estimates in large scale networks. Together, these findings confirm that FLASH achieves its design goal of being fast, lightweight, and scalable, providing an efficient attestation framework suitable for large-scale IoT and intelligent transportation systems.

Keywords: attestation, swarm attestation, homomorphic hashing.

Acknowledgements

Isaac Newton, though not coining the phrase, once remarked in a letter *"If I have seen further it is by standing on the shoulders of Giants"*. We, likewise, feel the same, and therefore would (in no order) like to take this opportunity to not just acknowledge, but wholeheartedly thank a number of people without whom this thesis would frankly not have been possible.

Firstly, we cannot express enough gratefulness for the support that we have received from Md Masoom Rabbani, and Francisco Blas Izquierdo Riera. Not only was this thesis inspired largely by their previous work and research, but without their infinite patience, willingness to answer our questions on weekends and late into the nights, and continuous feedback and help in revising our algorithms, this thesis would frankly not have been possible. Genuinely, we can't thank you enough.
We must also, in the same vein, thank Wouter Hellemans - for his support and guidance in developing our algorithms, and helping us write the best report possible. There are but a few people who would provide their time and expertise purely to help two (sometimes very confused!) students.

Next, we would like to thank the whole of KITS AB, and in particular Dennis Dubrefjord. Not only did KITS graciously provide us access to their office space, but they also financed the hardware which made collecting data possible, and kept our energy up through copious amounts of snacks and soft drinks!

Last, but certainly not least, no good acknowledgment section is complete without a thank you to family. Not only have our families supported us throughout this time (even though we had promised to be done months earlier), but they've sat countless times and listened to us talk about this thesis, despite it being completely beyond their realm of knowledge. Thank you.

Imad Alihodžić & Abirami Rengaraj, Gothenburg, 2026-01-18

Declaration of Generative AI

AI tools were used to improve grammar and clarity in passages of the text. All changes were reviewed, and the authors take full responsibility for the final content.

Imad Alihodžić & Abirami Rengaraj, Gothenburg, 2026-01-18

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

AdHash	Additive Hash
AES	Advanced Encryption Scheme
ECU	Electronic Control Unit
EPID	Enhanced Privacy ID
EV	Edge Verifier
FLASH	Fast Lightweight Attestation using Scalable Homomorphic Hashing.
HH	Homomorphic Hashing
HMAC	Hash-Based Message Authentication Code
HSM	Hardware Security Module
IoT	Internet of Things
ITS	Intelligent Transportation System
IV	Initialization Vector
LtHash	Lattice Hash
MitM	Man-in-the-Middle
MuHash	Multiplicative Hash
PoC	Proof of Concept
PUF	Physical Unclonable Function
PSP	Platform Security Processor
RA	Remote Attestation
RISC-V	Reduced Instruction Set Computing Version Five
RV	Root Verifier
SA	Swarm Attestation
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extensions
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module

Contents

List of Figures

List of Tables

1

Introduction

The world is now more connected than ever. Some 100 years ago, when data was exchanged through letters and telegrams, the modern public transportation systems of today were barely in their infancy. As communication and transport systems both evolved and progressed, so did many persons' reliance on them. During 2024 in Sweden, public transport held an approximate 27% market share compared to all transportation means, with this number rising to 47% in Stockholm [1].

The in-tandem rise of both communication and transportation technologies has ultimately culminated in the existence of large, intelligent transportation systems (ITS) - constructed and employed for a large number of purposes, including (public) transit management, and vehicle control [2]. As ITS have evolved and grown in scope and capabilities, our reliance on them to effectively, and safely, control the transportation that we take for granted has in turn grown, too.

However, this development has not come without risk. Cybersecurity threats are becoming not just a growing concern [3], but also a growing occurrence. Indeed, attacks against ITS have in recent years been on the rise around the world, with examples including a 2023 attack on Polish Railways which caused the emergency stoppage of trains [4], and a 2024 cyberattack on the Transportation for London which resulted in the shutdown of multiple transport services, and millions in damages [5]. Such attacks have illustrated that malicious parties could not just compromise data integrity, and system stability, but even human safety.

A major challenge in securing ITS is the fact that they are not monolithic systems, but rather large, dynamic, connected networks, with hundreds of thousands of nodes all working together. Many such systems rely on the continuous exchange of data with central aggregators, which means that cyber-threats can quickly propagate within networks [6]. Combined with the fact that ensuring the security of ITS devices remains difficult due to scalability issues, these systems are like dry tinder awaiting a spark - highly vulnerable to exploitation and fast cascading failures [7], [8].

One prominent solution for identifying malicious/infected nodes is remote attestation (RA). RA mechanisms typically operate on a challenge-response basis, in which a trusted entity (verifier) issues a challenge to an un-trusted (prover) node. The verifier can then check the prover's response, and compare it to a known safe

state; thus determining the integrity of the prover. To address scenarios involving large-scale (distributed) systems, wherein attestation is required for many provers and verifiers, RA schemes can be extended into swarm attestation (SA) schemes [9], [10].

1.1 Motivation

However, RA and SA schemes do not come without drawbacks (see: section 2.4). In the case of SA schemes for instance, one of the primary weaknesses is that solutions largely struggle in dynamic environments where devices constantly join or leave a network.

To address the common challenges within RA and SA schemes, this thesis aims to explore attestation through homomorphic hashing - a technique that allows for the computation of new hash values without knowing individual inputs. As highlighted in recent literature, it enables the aggregation of data in a manner that reduces computational complexity and avoids revealing the original values [11]. Several homomorphic hash functions have been developed through prior research, including Multiplicative Hash (MuHash), Lattice Hash (LtHash), and Additive Hash (AdHash) [12].

Whilst each mechanism offers distinct trade-offs between efficiency, security, and computational complexity (see: section 2.1), they all, nonetheless, have the potential to be leveraged to address the key challenges in swarm attestation.

More specifically, their unique property of data aggregation has the promise to be particularly suitable for applications like swarm attestation, where multiple devices must prove their integrity in a resource-constrained, distributed setting. By utilizing homomorphic hashes, a solution could be constructed wherein, instead of requiring each node's state to be individually verified and transmitted, attestation results could be combined into a single aggregate hash.

In essence, the motivation behind this thesis is to understand, and evaluate, how homomorphic hashing can be integrated to address traditional attestation challenges related to scalability, distribution, and dynamism.

1.2 Research Questions

To structure the goals of this thesis, and ultimately produce a more robust, and secure attestation mechanism, this thesis is centered around three primary research questions:

- What are the common challenges facing current attestation mechanisms, and how can the implementation of homomorphic hashing aid in their resolution?
- How can a hierarchical structure of attestation, with cluster heads and central

hubs, be securely managed using homomorphic hashing?

- What are the performance implications of introducing homomorphic hashing into a real-world hierarchical network structure?

1.3 Aim

In order to answer the previously outlined research questions, verify the validity of our proposed solution(s), and therein develop a new attestation framework that is applicable to a range of dynamic SA problems, the thesis will be based on an experiment-driven approach.

Firstly, the current challenges and issues within swarm attestation will be explored, based on current literature and studies. This lays the groundwork for the identification of the primary challenges to be resolved, as well as for the development of an adversary model.

Secondly, based on these identified constraints and adversary model, this thesis will investigate several ways in which homomorphic hashing can be implemented for secure attestation. More specifically, this will be done through the development of several attestation algorithms collectively named FLASH - Fast Lightweight Attestation using Scalable Homomorphic Hashing.

Thirdly, to verify FLASH, compare, and evaluate it depending on system constraints, this thesis will additionally develop both a physical proof of concept (PoC) implementation, and large-scale network simulation based on real-world data obtained from the PoC.

Finally, the results of the PoC and simulation will be used to extrapolate performance estimates for large networks, therein allowing for an assessment of FLASH in large scale, real world, contexts.

1.4 Limitations

To limit the scope of this thesis project, a number of assumptions will be made, as well as research limitations.

Firstly, this thesis will not explore the manner in which intra-vehicle/device communication is implemented. More specifically, the project will presume that the devices within each vehicle have already been configured such that they can function as a leaf node or edge verifier.
The reasoning behind this is that this thesis aims at developing a broad framework applicable to many different ITS, as opposed to purely implementation-specific networks, such as vehicle-to-vehicle.

A further limitation to this thesis is that physical adversaries will not be considered. Included within this limitation is consideration towards side channel attacks, and denial of service attacks on the network.

Lastly, two assumptions are made in regards to the bootstrapping process of devices to the swarm network. Firstly, devices are presumed to be secure and operational when they are first configured and connected to the network. Secondly, this thesis will not explicitly develop a secure mechanism for devices to dynamically join a swarm after an initial common bootstrapping phase (though the solutions proposed in chapter 3 can be extended to support this). This limitation will be further discussed in section 8.1.

2

Background

The following chapter presents the background theory and concepts necessary to understand this thesis, including an overview of attestation and its variants, homomorphic hashing, and the fundamentals of security theory.

2.1 Homomorphic Hashing

Homomorphic hashing was first pioneered by Bellare et. al., under the name "incremental" cryptography [13]. Put simply, homomorphic hashing provides a mechanism such that when one has the hash of an input and the input is changed, the hash of the new input can be computed using the original hash, without the need for the entire hash to be recomputed entirely. Figure 2.1 illustrates these capability differences.

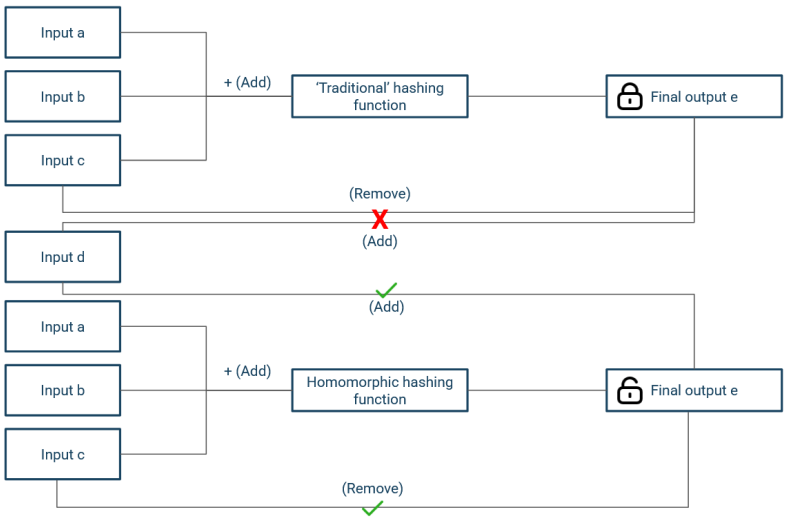


Figure 2.1: Traditional vs. Homomorphic hashing as illustrated by the possible operations on a hash output e created using inputs a, b, and c. With traditional hashing (seen top) no further operations are possible on e, e.g. the later addition of an input d to the hash, or the removal of input c. Using homomorphic hashing (bottom), however, enables those operations, effectively creating a hash that isnt locked to one final output.

Several homomorphic hashing algorithms exist, including MuHash, AdHash and

LtHash, first designed by Bellare and Micciancio, all utilizing standard hash functions along with algebraic operations [12]. These are covered in more detail in subsection 2.4.3.

2.2 Security

While security is a broad and multifaceted concept, and sub-concepts evolve and take on different meanings as technology progresses, there are several foundational principles that any effective solution must address. This section outlines the key theoretical constructs underpinning security.

2.2.1 CIA Triad

The CIA triad, standing for confidentiality, integrity, and availability, is a widely accepted foundation that defines what it means for a system to be secure [1].

- Confidentiality is the notion that any information exchanged, or stored, within a system is restricted such that only authorized parties may access (in other words, read) it. In simpler terms, data must be kept private.
- Integrity, meanwhile, refers to the fact that the data contained/exchanged within a system can be trusted to be complete, and accurate (free from any tampering).
- In regards to availability, this concept refers to the fact that systems should be able to provide data/information in a timely, and reliable, manner, without interruptions, or delay.

2.2.2 Freshness

The CIA triad, whilst fundamental to developing a secure service, does not provide a complete security guarantee. Indeed, even if data is unreadable, available upon request, and transmitted securely, an attacker could for example perform a man-in-the-middle attack, capturing data and re-sending, thus providing the appearance of a well functioning service.

Thus, an important concept to consider is freshness. Within the context at hand, data freshness is the verification, and assurance, that the data transmitted within a system is up to date/current. Freshness mechanisms are often implemented through timestamps, providing the ability for services to verify that the data they receive is not just correct, but also up to date.

[1] Note, the CIA triad does not have a single origin or attributable creator(s), but it is a common term referred to in literature.

2.3 Attestation

Attestation, at a simple level, is a catch-all phrase for any mechanism that can be used to verify a certain property about a system. As such, there exists a wide range of definitions for attestation and, likewise, the attestation of devices can be achieved through a number of means. However, the attestation of computer devices follows a number of fundamentals.

Firstly, it builds on the fact that one (or more) devices, so called verifiers, can verify that one or more provers are behaving as expected, without deviation. Although the exact nature of this exchange can take on many forms, all attestation solutions involve an exchange of information where provers send information to verifiers who, through computation, comparison, or any other means, infer the integrity of provers [14].

2.3.1 Remote Attestation (RA)

One form of attestation that is of particular interest (especially in today's age of widespread IoT device adoption) is remote attestation (RA). RA mechanisms allow for attestation to occur when prover and verifiers are either logically or physically separated, and instead connected over a network.

RA ensures that only an honest (not compromised) prover can convince the verifier that it is in an acceptable state. This is crucial as the prover and verifier communicate over an open network, in which attackers can attempt to evade detection through various means such as eavesdropping and forgery [15]. Based on the type of RA implementation, such solutions can generally be classified into three categories (though solutions can overlap), namely: software-based attestation, hardware-based attestation, and hybrid-based attestation.

2.3.1.1 Software-based Attestation

Software-based attestation schemes are ones which do not rely on hardware to perform remote attestation. The attestation program is run from memory to validate the state of the system's software. These schemes are attractive for low-cost devices because they require no dedicated Hardware Security Modules (HSMs) or Trusted Platform Modules (TPMs).

As highlighted in literature [16], [17], software based approaches provide a lightweight and cost effective means of establishing device trust, especially in commodity IoT and cyber-physical system environments. However, these approaches often rely on timing and behavioral assumptions that may not hold in adversarial network settings - making them less resilient to physical or low level software attacks. Recent advancements, such as delegated or non-interactive attestation, have improved the scalability of software-based schemes by distributing verification responsibilities and reducing per-device computational overhead.

2.3.1.2 Hardware-based Attestation

Hardware-based attestation schemes, meanwhile, perform integrity verification using dedicated hardware components that act as a root of trust. In this model, a trusted hardware element such as TPM, Trusted Execution Environments (TEE) or a Physical Unclonable Function (PUF) is responsible for securely measuring, storing and reporting the system's integrity state. The attestation process typically involves cryptographically signing measurements of firmware or runtime states, and sending these to a remote verifier for validation.

Hardware-based schemes provide strong assurance by protecting attestation logic and secret keys from software compromise [17], [18]. Because the measurement and signing operations are executed within isolated hardware, adversaries with control over the main processor or operating system cannot easily forge attestation evidence.

These approaches offer robust defense against low-level attacks, including firmware modification and runtime code injection, making them suitable for critical infrastructures. However, they also introduce higher implementation costs and hardware dependencies, which can limit scalability in resource-constrained IoT environments.

2.3.1.3 Hybrid Attestation

The purpose of hybrid attestation, meanwhile, is to combine the security guarantees of hardware-based attestation with the lower cost of software-based attestation. The key idea is to embed a minimal hardware root of trust, such as a secure boot-loader or lightweight co-processor, that anchors the attestation process, while most of the attestation logic is executed in software. This design allows verification to remain secure while avoiding the full hardware overhead.

Hybrid approaches mitigate the limitations of purely software-based schemes, which rely on strong timing and adversary assumptions, while maintaining lower complexity than fully hardware-based designs [19], [20]. As a result, several hybrid software-hardware co-designs have been proposed to overcome this limitation [17], [19].

2.3.2 Swarm Attestation (SA)

Although traditional RA focuses on verifying the integrity of individual devices, modern systems are increasingly composed of large numbers of interconnected and autonomous nodes. As these environments grow in scale and complexity, attestation mechanisms must evolve from verifying a single prover, to verifying the large numbers of provers,

To this end, SA schemes are designed with the explicit purpose of attesting large-scale networks of devices (swarms), where devices cooperate to collectively prove the integrity of the overall network.

As with attestation, in general there is no single way to implement SA. Researchers have proposed a wide range of different solutions (see section 2.4). Furthermore, solutions target static or dynamic swarms. Static swarms are those in which the

network topology does not change, whereas dynamic swarms consistently change, for example, when devices join or leave the network.

Table 2.1 summarizes how SA inherits the core principle of RA while adapting them to handle the scalability and heterogeneity of swarm systems.

Property	Remote Attestation	Swarm Attestation
Scope	Focuses on verifying a single prover integrity remotely.	Extends RA to verify multiple provers collaboratively within a swarm of interconnected devices.
Authenticity & Freshness	Verifier checks the current state of a single devices to ensure recent and accurate measurements.	Ensures collective responses from a group of devices are fresh and un-tampered.
Scalability	Typically designed for one-to-one or small-scale systems.	Must support large-scale, IoT swarms with minimal overhead.
Heterogeneity	Less emphasis and usually assumes consistent hardware/software.	Must support diverse devices with varying hardware and configurations.
Unpredictability & Unforgeability	Challenges must be unpredictable to prevent precomputed responses.	The same property must hold collectively, even under coordination among multiple devices.
Determinism	Repeated challenges should yield consistent and verifiable results.	Responses across the swarm must be consistent, without relying on centralized synchronization.
Topology Awareness	Often assumes a static topology or direct verifier-prover link.	May support dynamic, mobile or ad hoc topologies depending on the schemes.

Table 2.1: Comparison of Key Properties: Remote Attestation vs. Swarm Attestation [14], [21]

2.3.3 Attestation Models (On-demand & Self-attestation)

Building on swarm attestation frameworks, this thesis adopts two complementary attestation models. The first model, on-demand attestation, follows a verifier-initiated challenge-response process that ensures synchronized verification across all leaf nodes. The second, self-attestation, enables each leaf node to autonomously perform attestation upon receiving a uniform trigger signal, reducing communication overhead and enabling periodic verification. The detailed design and comparative evaluation of these two models are presented in chapter 3 and chapter 4.

2.3.3.1 On-Demand Attestation

In on-demand attestation, the verifier initiates the process by issuing a challenge to (one or several) prover devices. Provers then compute an integrity measurement, respond with an attestation report, and the verifier validates the response. This scheme is reactive, and is well suited in a range of scenarios, such as when:

- The network is small or tightly controlled,
- Message timing and clocks are inaccurate or unreliable,
- Communication overhead is acceptable because the number of nodes is modest.

2.3.3.2 Self-Attestation

By contrast, self-attestation is a proactive scheme in which each node periodically triggers its own attestation and reports its results without waiting for a verifier challenge. In many swarm or large-scale contexts, this approach is beneficial because nodes can operate autonomously. Literature [17] describes such models as non-interactive attestation models: where nodes send integrity reports at regular intervals rather than waiting for a challenge. This scheme is well suited in scenarios when:

- Scalability is a higher priority than centralized control,
- Continuous and proactive integrity assurance is desired rather than reactive, or
- The Verifier cannot continuously challenge all child prover devices due to scale or intermittent connectivity.

In summary, on-demand attestation provides high-assurance, verifier-driven validation suitable for smaller or critical systems, whereas self-attestation offers scalable and autonomous integrity verification ideal for large, distributed networks [17].

2.4 Related Works

RA, and particularly SA, has received serious attention in the context of securing large-scale IoT networks. Many research works have been conducted, each differing in their architecture, assumptions, scalability, and adversary model. This section presents a high-level review of state-of-the-art papers and their efforts to address these problems. The objective is to highlight key contributions, identify limitations, and highlight the research gap(s) that this thesis aims to address.

2.4.1 Overview of Traditional Attestation Approaches

Banks et al. [17] conducted a comprehensive literature review that evaluates RA schemes in multiple dimensions of threat and scalability. Their analysis highlights that many traditional protocols are not adequately equipped to handle the dynamic and distributed nature of networks, and notably identified a lack of robustness when dealing with mobility, intermittent connectivity, and coordinated multinode attacks.

Ménétrey et al. [22] analyzed attestation mechanisms leveraging TEE, such as Intel Software Guard Extensions (SGX), Arm TrustZone, AMD Secure Encrypted Virtualization (SEV), and Reduced Instruction Set Computing-Version Five (RISC-V) platforms, to enhance reliability in untrusted execution environments. Their study illustrates how TEEs can enforce mutual attestation, isolate sensitive operations from host operating systems, thereby strengthening the trust anchor in heterogeneous IoT deployments.

In addition, several independent works have conducted in-depth analyses of specific TEE platform:

- Intel SGX provides enclave-based execution to ensure confidentiality and integrity even when the operating system or hypervisor is untrusted. Swami [23] offers an extensive analysis of the SGX remote attestation framework and its limitations in practice. The study emphasizes that while SGX remote attestation, built on the Enhanced Privacy ID (EPID) group-signature protocol, can prove that code executes within genuine Intel hardware, this guarantee alone is not sufficient for end-to-end security. Overall, Swami concludes that although SGX provides strong hardware-based isolation and a well-founded attestation model based on cryptographically verifiable enclave measurements, its reliance on centralized trust and its susceptibility to composition and side-channel risks must be carefully considered.
- AMD SEV extends hardware-based protection to virtualized cloud environments by encrypting the memory of virtual machines with dedicated, per-virtual machine keys. This design prevents a privileged hypervisor from inspecting guest memory, thereby providing confidentiality for workloads running in potentially untrusted cloud infrastructures. Buhren et al. [24] conducted a comprehensive analysis of the SEV remote attestation protocol implemented on AMD Epyc processors and revealed critical weaknesses in its firmware-based trust model. Their study demonstrated that it is possible to

extract platform-specific private keys from the Platform Security Processor (PSP) firmware, enabling attackers to impersonate SEV-enabled platforms or decrypt protected VM memory. Overall, the analysis highlights that while SEV provides strong confidentiality guarantees at runtime, its remote attestation mechanism remains fragile when faced with firmware compromise or centralized key trust, factors that are particularly critical in large-scale distributed environments.

2.4.2 Overview of Swarm Attestation Approaches

As explained previously, SA protocols aim to extend the remote attestation model to large and dynamic networks of interconnected devices known as swarms. Several approaches have been proposed to achieve scalability robustness, and efficiency under diverse threat assumptions.

Asokan et al. [25] proposed Scalable Efficient Device Attestation (SEDA) which allows multiple devices in a swarm to be attested simultaneously. This enhances security across the network by verifying device integrity in bulk. It also scales well. However, it has limitation in terms of performance, particularly regarding the number of devices in the swarm and the number of devices in the spanning tree as the network grows. Additionally, the attestation process fully depends on neighboring devices to propagate the attestation, which could become a bottleneck if some devices are unreliable. Most importantly, SEDA produces only a boolean attestation outcome and assumes a largely static network topology.

To overcome the above limitations of SEDA, Ambrosin et al. [8] introduced Secure and Scalable Aggregate Network Attestation (SANA) - the first attestation scheme for large collections of devices that is scalable, verifiable, and supports untrusted aggregation during the attestation process. However, SANA does not solve all issues related to SA, as it relies on a single verifier that must be inherently trusted. In addition, it involves a higher computational cost and employs an optimistic aggregation scheme, leading to greater implementation complexity.

Identifying the single-verifier issue within SANA, Rabbani et al. [26] introduced Scalable Heterogeneous Layered Attestation (SHeLA). As the name suggests, SHeLA is an attestation mechanism that instead builds a layered, hierarchical tree structure with multiple edges which, in turn, attest provers and report results to a root verifier. While, SHeLA resolves many of the issues with SANA (particularly the difficulty of adding new edge devices), it does not consider physical adversaries and still presumes the authenticity of the root and edge verifiers.

To further decentralize attestation, Kuang et al. [27] presented Automatic Federated Swarm Attestation (FESA), which allows multiple verifiers to cooperate in swarm-wide integrity assessments. Although FESA reduces reliance on centralized infrastructure, it introduces synchronization and trust management complexities. In a complementary effort, their Efficient and Secure Distributed Attetsation Scheme for IoT Swarms (ESDRA) [21] protocol distributes the attestation responsibilities across cluster heads using a reputation-based parallel verification system. This ap-

proach enhances scalability and enables finer-grained trust judgments but assumes static clusters and does not consider physical adversaries.

Carpent et al. [28], meanwhile, introduced Lightweight Swarm Attestation (LISA-s) - two lightweight swarm attestation protocols designed for mobile IoT swarms with varying Quality of Swarm Attestation (QoSA) capabilities. LISAα prioritizes simplicity and low architectural overhead, but suffers from high communication cost due to lack of aggregation of reports. LISAs improves scalability and bandwidth efficiency through synchronous operation and aggregation.

Ammar et al. [29] introduced Lightweight Intelligent Swarm Attestation (WISE), which uses a Hidden Markov Model to intelligently select and attest devices that are most likely to be compromised based on their attestation history and network characteristics. Unlike deterministic approaches, it follows a probabilistic model and attests only a subset of devices in each iteration, so delayed detection of compromised devices may occur if they are not selected in the given iteration. Moreover, WISE is designed for static networks, which limits its applicability in dynamic environments.

Hellemans et al. [30] proposed a Secure Privacy-Preserving Anonymous Swarm Attestation protocol (SPARK), designed for in-vehicular networks following zonal architecture. Unlike traditional swarm attestation schemes that assume a trusted verifier and expose device configurations during verification, SPARK achieves privacy, anonymity, and traceability using a TPM 2.0-backed group signature scheme. The protocol binds each Electronic Control Unit (ECU) to its Zonal Gateway through a hierarchical key structure, preventing change-of-path attacks and enabling privacy-preserving evidence aggregation. Overall, SPARK represents one of the first swarm attestation frameworks to offer public verifiability and TPM-backed anonymity, marking an important advancement toward scalable and privacy-preserving attestation in vehicular and IoT swarm environments.

El Kasem et al. [31] introduced PRIVÉ, a privacy preserving and accountable swarm attestation protocol that extends traditional Direct Anonymous Attestation (DAA) with traceability and evidence privacy. In PRIVÉ, evidence privacy ensures that a device can prove its integrity without revealing any sensitive information about its internal state. Unlike prior swarm attestation schemes that assume a fully trusted verifier, PRIVÉ removes this assumption by allowing devices to produce verifiable yet anonymous attestation evidence. PRIVÉ represents a significant step toward zero-trust, privacy-preserving, and accountable swarm attestation in dynamic IoT and edge environments.

Ambrosin et al. [32] developed Practical Attestation for Highly Dynamic Swarm Topology (PADS), a lightweight, consensus-based swarm attestation protocol designed for highly dynamic and unstructured networks. Although PADS eliminates the need for centralized coordination and remains efficient in resource-constrained devices, it cannot guarantee complete 100 percent attestation coverage of all nodes in the swarm. In addition, PADS lacks robustness against physical attacks and does not support dynamic node enrollment, limiting its applicability in evolving IoT environments.

Ambrosin et al. [33] contribute a systematic survey of Collective Remote Attestation (CRA) Schemes. Their work classifies CRA protocols by system architecture, adversary model, and aggregation techniques. They particularly emphasize scalability issues in the deployments, the need for secure result aggregations in the presence of compromised nodes, and the limited defenses against physical adversaries. These insights help define the broader research and motivate the need for protocols with improved decentralization and resilience to physical attack.

2.4.2.1 Publish/Subscribe-Based Attestation

Conventional attestation protocols generally assume direct interactions between verifiers and provers. In contrast, event-driven communication models, such as publish/subscribe require decentralized and asynchronous attestation techniques.

Dushku et al. [34] introduced Provable Remote Attestation for Public Verifiability (PROVE) IoT environments. It supports public verifiability and operates without the need for pre-shared keys or public-key infrastructure. PROVE enables untrusted subscribers to attest the integrity of publishers using lightweight, symmetric cryptographic operations. Hardware validation confirms its feasibility on constrained devices. PROVE represents a significant step toward enabling secure attestation in decentralized, event-driven IoT networks.

2.4.3 Homomorphic Hashing

Bellare and Micciancio [12] introduced a class of incremental and collision-resistant hash functions built on the randomize-then-combine paradigm. This approach supports incrementality, enabling the hash output to be updated in proportion to the size of the change instead of requiring recomputation from scratch. Their work introduced three main types, namely LtHash, MuHash, and AdHash:

- **LtHash:** LtHash is a popular homomorphic hashing algorithm based on lattice cryptography. It takes a set of arbitrarily long elements as input and produces a 2KB (2048 byte) hash value. Unlike traditional hash functions such as SHA-256 which produce a fixed-length 256-bit digest, Lthash generates a much larger output. This gives it two important properties: set homomorphism and collision resistance.

  Set Homomorphism: For any two disjoint sets, S and T, the hash of their union can be computed as the sum of their individual hashes. This property enables efficient computation of hashes for larger sets by combining hashes of smaller ones.

  $$\mathbf{LtHash(S \cup T) = LtHash(S) + LtHash(T)}$$

  Collision Resistance: It is computationally infeasible to find different sets whose hashes collide (i.e. produce the same hash value), and security is proven based on the hardness of the shortest lattice vector approximation [12], [35].

- **MuHash:** Unlike LtHash, MuHash is faster and more efficient because it requires only one modular multiplication per message block. Like LtHash, MuHash also has the properties of set homomorphism and collision resistance, but its approach differs.

  Set Homomorphism: MuHash is homomorphic under multiplication (i.e., for any two disjoint sets S and T, the hash of their union can be computed as the product of their individual hashes in a given group G).

$$\mathbf{MuHash(S \cup T) = MuHash(S) \cdot MuHash(T) \mod p}$$

  Collision Resistance: The security of MuHash is provable based on the hardness of the discrete logarithm problem (DLP) in the chosen group G. If someone can find a collision in MuHash, they could also effectively compute discrete logarithms, which is considered computationally infeasible in a well-chosen group. Since the DLP also underlies widely deployed public key cryptography primitives, such as Diffie-Hellman key exchange used in Transport Layer Security (TLS), a collision attack on MuHash would imply a break of these cryptographic assumptions rather than a weakness in MuHash [12], [35].

- **AdHash:** AdHash is an incremental and additive homomorphic hash function that combines randomized message blocks through modular addition. Unlike MuHash, which operates multiplicatively in a group, AdHash performs this by combining operations using addition modulo a large integer M, achieving improved efficiency while retaining provable collision resistance.

  Set Homomorphism: For any two disjoint sets S and T, the hash of their union can be efficiently obtained as the sum of their individual hashes modulo M.

$$\mathbf{AdHash(S \cup T) = (AdHash(S) + AdHash(T)) \mod M}$$

  Collision Resistance: The security of AdHash is based on the hardness of the weighted subset-sum knapsack problem, which is provably hard under the assumption that approximating the length of the shortest lattice vector is computationally infeasible [12], [35].

The following Table 2.2 provides a comparison of these aforementioned homomorphic hash.

Property	MuHash	AdHash	LtHash
Operation Type	Modular multiplication in a group.	Modular addition over integers.	Vector addition in a lattice space.
Security Basis	Based on discrete logarithm problem.	Based on subset sum (knapsack) problem.	Based on lattice problems (e.g., SVP).
Efficiency	High efficiency with moderate cost; supports incremental updates.	Lightweight, insecure, and fast, offering high performance for simple operations.	Secure but moderately efficient due to higher computational overhead.
Incrementality	Efficiently supports update via multiplication/division of elements.	Supports simple additive update of hash values.	Allows incremental updates using vector operations.
Use Case Suitability	Ideal for IoT swarm attestation and real-time aggregation.	Suitable for low-power or embedded systems.	Best for large scale file system integrity.

Table 2.2: Comparison of Homomorphic Hash Functions: MuHash, AdHash, and LtHash [12], [35]

In regards to the usage of homomorphic hashing within distributed systems, Lewi et al. [35] formalized the use of homomorphic hashing for securing large-scale update propagation in distributed systems. Their work addresses the problem of ensuring integrity during database replication across untrusted peers while minimizing verification overhead. Building on the lattice-based LtHash construction of Bellare and Micciancio, they introduced a complete security model for secure update propagation, defining the correctness and robustness of propagation algorithms under adversarial modification. The key contribution is an efficient update-signing scheme in which each new state can be verified using only the previous hash and the update itself, rather than recomputing or storing large Merkle trees. This deployment demonstrates how homomorphic hashing can replace Merkle trees in scalability-critical systems such as swarm attestation, where large groups of nodes must periodically prove data integrity without recomputing or transmitting full state hashes.

2.5 Threat/Adversary Models

Whilst the focus of this thesis is to develop a framework which can attest that devices have not been undermined by attackers, the mechanism(s) by which this is done must naturally, be secure too. The following is therefore a holistic overview of adversary models, which are discussed in relation to FLASH in subsection 3.1.6.

An adversary, also known as an attacker, is an entity whose goal is to undermine the privacy, integrity, and availability of data, often with malicious intent [26]. We divide the adversary model to be considered into three categories - remote, local, and physical adversaries. However, in line with other attestation mechanisms, only the former two adversary types are considered.

2.5.1 Local Adversaries

Local adversaries are those which are physically present within the communication range of devices within the swarm, but which do not have direct physical access to the devices. Such an adversary can intercept, replay or inject messages in the communication channels (e.g., replay attacks, man-in-the-middle attacks).

2.5.2 Remote Adversaries

Remote adversaries are those which are able to compromise devices without any physical presence in the network. Typically, such an attacker exploits software vulnerabilities or bugs in the device firmware to inject malicious code remotely.

2.5.3 Physical Adversaries

Physical adversaries, meanwhile, are those which have complete physical access to devices. These adversaries can be divided into several sub-categories, including non-intrusive adversaries capable of performing side-channel attacks, and intrusive adversaries capable of manipulating device hardware/software states, as well physically extracting information.

These attacks can include hardware tampering, side-channel analysis (such as power analysis or electromagnetic emanation analysis), fault injection, or direct extraction of data from memory or storage components. Physical attackers can potentially bypass software-based security measures by directly manipulating hardware components or extracting cryptographic keys from physical memory.

3

FLASH

In order to address the shortcomings of other studies within swarm attestation, and satisfy the proposed threat model, this thesis introduces FLASH - Fast Lightweight Attestation using Scalable Homomorphic Hashing.

First, the requirements will be introduced, followed by an overview and explanation of FLASH. Subsequently, the commonalities, notation, and the algorithms themselves will be presented.

3.1 Requirements

Based on the challenges and drawbacks identified in the literature review in section 2.4, and the established threat model presented in section 2.5, FLASH was developed under the following set of requirements/guidelines.

3.1.1 Completeness

The first, and most fundamental requirement for all variants of FLASH, is completeness. Namely, the algorithms must ensure that the integrity of each device in the network is successfully evaluated (attested), and that all attestation results are aggregated up to the root.

3.1.2 Efficiency

Beyond completeness, all of FLASH's algorithms also need to be efficient in terms of computational, and message transport, cost. The reasoning behind this requirement is that FLASH's primary area of potential application is one where a swarm consists of many low power IoT devices. Thus, for it to be applicable in real-world environments, it needs to be as lightweight as possible.

3.1.3 Scalability

Another requirement that goes hand-in-hand with efficiency is scalability. Namely, even if the algorithms are lightweight and can be performed on one device, this is no guarantee that the same applies when increasing the number of devices in a swam.

To this end, the use of homomorphic hashes allows for the aggregation of hash values in the hierarchy presented in subsection 3.2.1, meaning that individual reports do not need to be inspected, and therein reducing computational and bandwidth overhead.

3.1.4 Heterogeneity and Dynamism

Whilst scalability and low computing overhead allow for the creation of large networks of devices, it does not however guarantee a solution that is robust in respect to networks wherein devices differ in architecture, capabilities, and reliability. This is an issue as existing attestation mechanisms often make assumptions about either uniformity, fixed topology, or both, therein making them unsuitable for highly dynamic environments where devices frequently join, leave, or move within the network.

Thus, the algorithms developed must allow for dynamism, both in regards to what devices are in the network, and the type of network. This further motivates the development of multiple algorithms as they will allow for greater understanding of performance impacts of different security solutions.

3.1.5 Resilience

Another issue, as discussed, that many attestation solutions face (e.g., SANA [8]) is the fact that their architectures rely on a central verifier, introducing a single point of failure. This presents a dual-sided risk, in that if this node is compromised, the entire system's integrity evaluation collapses and, likewise, if disabled prevents the network from being attested. Therefore, none of the proposed algorithms in this thesis rely on a single source of truth (which itself cannot be verified).

3.1.6 Security

There are, likewise, a number of security requirements for FLASH, in line with the established threat model. Whilst section 3.2 explains the structure of FLASH, and subsection 3.2.2 expands upon the specific use-cases and security guarantees provided, a number of requirements apply in all scenarios.

3.1.6.1 Local Adversary Mitigation

In regards to local adversaries, FLASH must guarantee that each message is not only valid, but also authentic (in that it provably comes from a certain origin, and the content has not been modified). To this end, one particular area of focus is freshness. More specifically, each attestation result includes or utilizes information to prevent the replaying of previously valid responses, ensuring that the message was both generated by a trusted party, and is current.

3.1.7 Remote Adversaries

As for remote adversaries, protection against remote compromise is implemented in several ways. However, one particular cornerstone of FLASH, that allows for the detection of remote adversary threats, is it's ability to allow for the localization of faults. Namely, since every device in the network is attested by a higher parent, FLASH can detect signs of the remote compromise of any specific node (and its corresponding edge verifiers).

Additionally, at the hardware level, FLASH fundamentally assumes that all cryptographic keys are stored on device-level root of trusts, therein preventing malicious entities from impersonating devices in the network.

3.2 Overview of FLASH

To satisfy the requirements whilst also exploring different ways in which homomorphic hashing can be applied, FLASH is divided into two distinct algorithms (types of attestation), namely on-demand attestation algorithm, and self-attestation algorithm (see Figure 3.1).

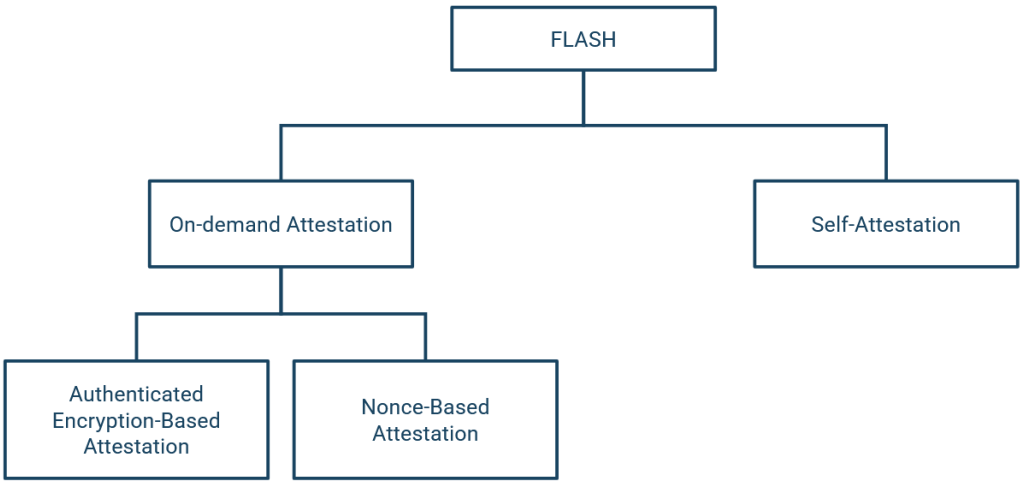


Figure 3.1: Composition of FLASH

In the case of the on-demand attestation algorithm, attestation is performed on a challenge-response basis, using two different variations wherein the attestation requirements are upheld using either authenticated encryption, or nonces.
The self-attestation algorithm meanwhile, uses a different approach, omitting attestation challenges and instead relying on leaf nodes periodically triggering their own attestation and sending the result to their parent edge verifier.

3.2.1 Network Model

An important factor for satisfying the aforementioned requirements is FLASH's network model, which consists of a tree-based network hierarchy with three conceptual

layers, as seen in Figure 3.2.

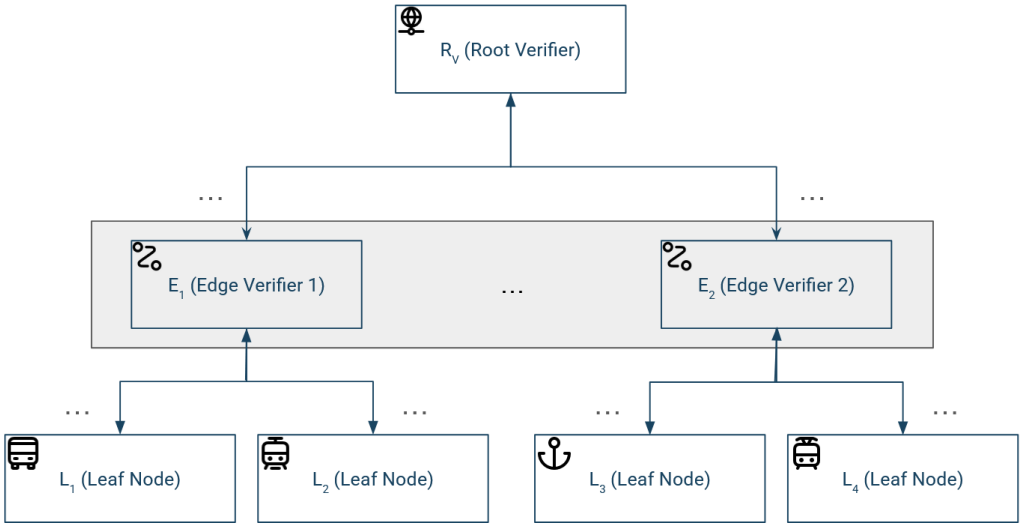


Figure 3.2: Network Hierarchy of FLASH

At the top is the root verifier, which serves as the central anchor for the entire network. The root verifier does not perform direct validation of the leaf nodes; rather, it's responsible for receiving aggregated hashes from the intermediate layer, and making the final integrity assessments for the network as a whole by comparing finalized hash tables.

Beneath the root verifier lies one or more layers of one or more intermediate child edge-verifier nodes that act as both provers and verifiers. These edge-verifiers occupy a dual role within the system. They function as verifiers for the devices directly beneath them validating and aggregating the hashes received from their child nodes and as provers to the root, generating their own attestation reports to be submitted upward. This layered design supports distributed validation, and the addition of as many node-clusters as desired, while ensuring that every device, including those performing verification tasks, is itself subject to attestation.

At the bottom of the hierarchy is the final layer, consisting exclusively of leaf nodes (provers). Each leaf prover is assigned to a single prover-verifier, to which it sends attestation reports to. Leaf nodes do not perform any verification themselves and are the terminating points in the attestation chain.

This hierarchical structure enables efficient scaling of the attestation process. By delegating verification tasks to the intermediate layer, the root verifier is insulated from the overhead of handling a large number of nodes directly. At the same time, it ensures that every node in the network, regardless of role, is incorporated into the trust evaluation process. The design remains identical across both attestation algorithms, providing a consistent and predictable topology that simplifies integration and analysis.

The roles and responsibilities outlined above are formalized in Table 3.1.

Symbol	Entity	Description
R_V	Root Verifier	Central authority responsible for final integrity evaluation. Aggregates and verifies top-level attestation data submitted by edge verifiers.
E_V	Edge Verifier (Prover-Verifier)	Intermediate node serving dual roles. Validates attestation data from its assigned leaf nodes and generates its own report to be verified by R_V.
L_{ID}	Leaf Prover Node	Terminal node in the hierarchy. Generates attestation reports (periodically or upon challenge) and sends them to its corresponding E_V.

Table 3.1: Formal Representation of Attestation Entities

Regardless of the algorithm or variant, FLASH implements mechanisms to ensure data freshness, and prevent replay attacks. In the case of periodic attestation, each node sends its report on a fixed schedule, and the verifier validates the received timestamp against the expected interval with a tolerance window. For the on-demand algorithm, freshness is enforced through the generation of unique challenges, which are shared with the L_{ID} and used to create responses.

3.2.2 Uses Cases & Motivation

The reasoning behind the design, development, and testing of two classes of algorithms, as opposed to one, is multi-factored. Firstly, this approach allows for the support of different operational contexts, providing a more robust framework capable of being used in a wider array of situations.

Secondly, it allows for the implementation and comparison of several different security features and checks, with different security assumptions, allowing not just more in depth research into how homomorphic hashing can be used effectively, but also supporting the aforementioned operational differences. A detailed comparison of on-demand and self-attestation is provided in Table 3.2, whilst Table 3.3 provides a further comparison of the on-demand algorithm variations.

	On-Demand	Self-Attestation
Key Characteristics & Differences	• Attestation performed on-demand, with R_V initiating attestation • Simpler mechanism requiring storage and computation of fewer entities • No need for swarm synchronization since challenges are sent synchronously	• Attestation performed using UAT signals • Attestation data only sent one-way, reducing message overhead
Use-Case	• Swarms where implementation simplicity is important	• Swarms where more precise control over attestation timing is required • Swarms with limited bandwidth
Commonalities	• Hashes are aggregated at E_V layers • Monotonic clock-based values used to create symmetric keys • Same bootstrapping behavior at L_{IDs}	

Table 3.2: Self vs. On-Demand Algorithm Comparison

	Nonce-Based	Encryption-Based
Key Characteristics & Differences	• Does not require encryption. • Lower overhead than encryption-based attestation. • Confidentiality ensured using K_{Secret} and $Challenge$ values.	• Confidentiality ensured using encryption of attestation values. • Data kept confidential in both directions.
Use-Case	• CPU limited swarms where on-device encryption is restricted.	• Memory-limited swarms where on-device storage is restricted. • Swarms where sharing configuration can cause privacy issues
Commonalities	• Attestation message integrity ensured using signatures of H_{att}. • Hash aggregation through multiplication of responses. • L_{ID} Hashes stored to enable future L_{ID} (and corresponding R) removal, addition, and updates.	

Table 3.3: On-Demand Algorithm Comparison

3.3 Algorithm Notation

The following section provides all notation used within the on-demand and self-attestation algorithms.

3.3.1 Common Notation for all Algorithms

Entities	
K_{Secret}	Symmetric key established during bootstrapping
H_{ATT}	Attestation hash generated at L_{ID}
R	Canonicalized H_{ATT}
S	Hash signature of H_{ATT}
T_H	Combined hash of all values of R recieved in a cycle
$bootCtr$	Monotonic counter for each node, incremented on every reboot
F_B	Canonicalized hash of T_H combined after bootstrapping
F_R	Canonicalized hash of T_H after attestation cycle
M_L	Map of all L_{ID} bootstrapped, and corresponding data
M_{RL}	Map of L_{ID} that have responded to a challenge, and corresponding data
LTS	Indicates an entity is stored across bootstrapping/attestation rounds
F_S	Copy of F_R sent by E_V to parent

Table 3.4: Common Entities Across All Algorithms

Functions	
authDecrypt(a,b,c)	Decrypts string a using key b and nonce c
authEncrypt(a,b,c)	Encrypts string a using key b and nonce c
verifyHMAC(a,b,c)	Verifies signature of hash c using HMAC value b, and key a
generateHMAC(a,b)	Creates a HMAC of string b using key a
muHash(a)	Create a MuHash value for string a
att()	Generates a device attestation value
muHash(att())	Creates a homomorphic hash of the leaf attestation data
reset()	Resets entities M_{RL}, F_R, T_H
flag()	Compares M_L to M_{RL} to identify mismatched nodes.
uptime()	Returns monotonic device local up-time counter
attTime()	Atomic reference timer at the edge verifier

Table 3.5: Common Functions

3.3.2 Notation for On-Demand Algorithms

Entities	
E_C	Encrypted challenge message generated at E_V for a L_{ID}
E_R	Encrypted response message generated at L_{ID} for a E_V
E_R	Encrypted reboot message generated at L_{ID} for a E_V
Q	Result of decrypting a challenge at a L_{ID} during attestation
$nonceCounter$	Counter for challenge/response messages
τ	Random nonce used for encrypted messages during reboot
K_T	Random key generated at L_{ID} for use by E_V during reboot
K_C	HKDF derived encryption-based algorithm challenge key
K_R	HKDF derived encryption-based algorithm response key
K_B	HKDF derived encryption-based algorithm reboot key

Table 3.6: Entities Used for On-Demand Attestation

3.3.3 Notation for Self-Attestation Algorithm

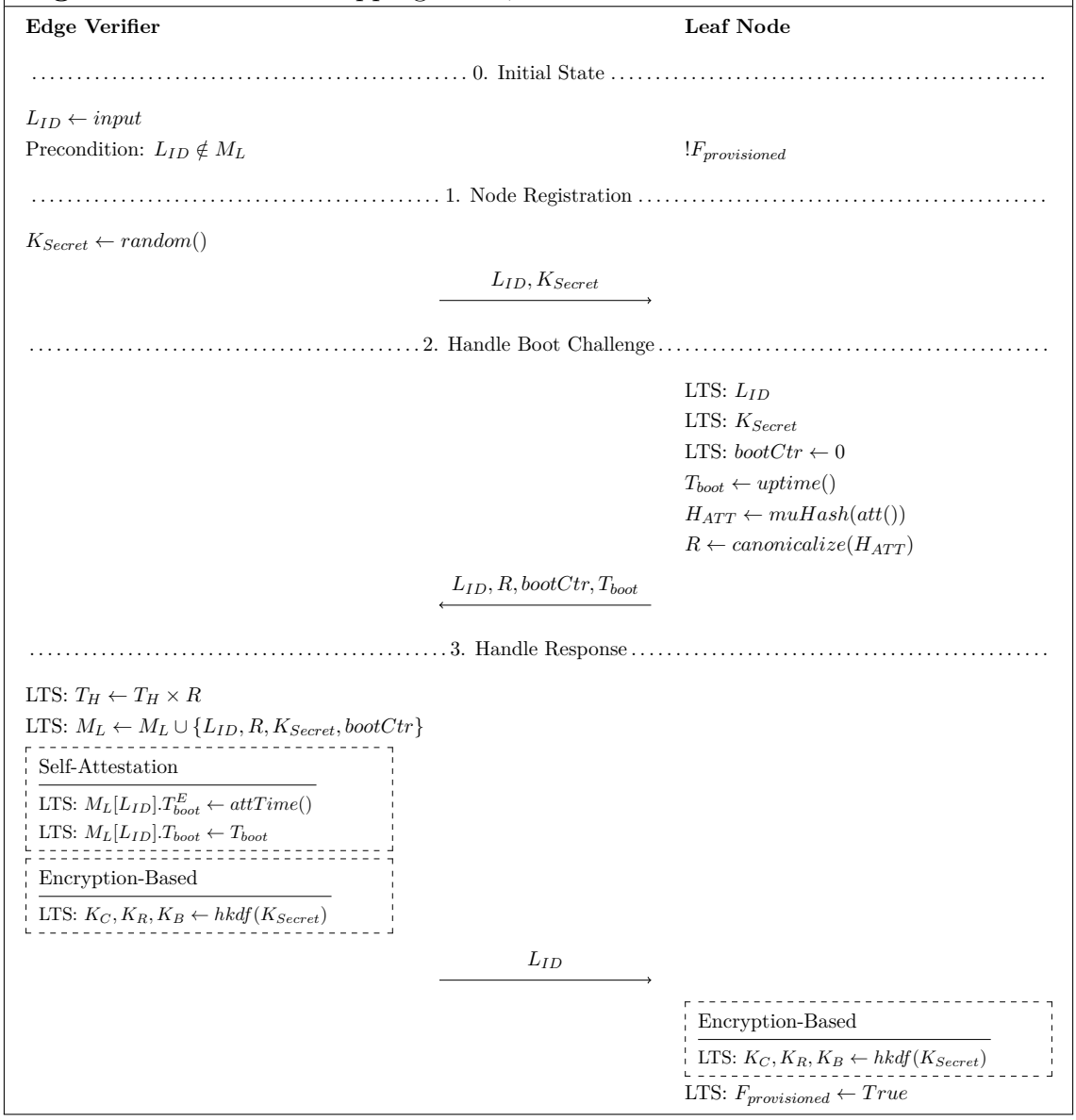
Reboot Phase Entities	
T_{boot}	Boot timestamp of the node
M_{reboot}	Reboot message sent from node to edge
T^E_{boot}	Boot timestamp recorded at the edge verifier
S_R	Reboot authentication tag
Attestation and Verification Entities	
UAT signal	Uniform Attestation Trigger for self-attestation
T_{up}	Uptime of node since boot
K_i	Subkey derived from the master secret K_{secret}
i	Fixed identifier used for key separation
T_{now}	Current timestamp recorded at the verifier during verification
Δ	General drift tolerance window used in self-attestation
$elapsed$	Elapsed time between two consecutive attestations
$last\_Tup[L_{ID}]$	Last observed uptime value of leaf L_{ID} stored by the verifier
T^E_{seen}	Verifiers local timestamp
nonce	Monotonic nonce constructed from boot counter and uptime

Table 3.7: Entities Used for Self-Based Attestation

3.4 Bootstrapping

The bootstrap phase establishes a secure baseline for all L_{ID} in the network, as described in algorithm 3.1. With the exception of part 3, 'Handle Response', the behavior is shared across all algorithms. Because this phase is presumed to be secure, no encryption is used, and the data is presumed to be legitimate, correct, and fresh.

Algorithm 3.1: Bootstrapping for E_V and L_{ID}



3.4.1 Initial State

Bootstrapping begins with an initial state wherein each edge verifier is provided a list of the leaf nodes (L_{ID}) it is responsible for. Leaf nodes, meanwhile, have a firmware flag variable $F_{provisioned}$ set to false, therein ensuring the node provides a bootstrap response, as opposed to a runtime response.

Bootstrapping then proceeds to L_{ID} registration under the precondition that an L_{ID} has not already been bootstrapped (added to M_L).

3.4.2 Node Registration

After the initial state is set, the E_V then sends a plain-text challenge to the L_{ID} containing its L_{ID} value, and K_{Secret} - a unique value generated using the $random()$ function to retrieve the E_{Vs} up-time since boot.

3.4.3 Handle Boot Challenge

When the L_{ID} receives the message, it first stores the received L_{ID}, as well as K_{Secret}. It then computes a MuHash H_{ATT} of its $att()$ function return value. As explained previously, owing to the properties of MuHash, H_{ATT} can be canonicalized, which is what the L_{ID} then does - therein creating the response R. Finally, the leaf updates its firmware flag to mark that bootstrapping is over, before sending R and its L_{ID} back to the parent E_V.

3.4.4 Handle Response

Upon receiving the response, the E_V first multiplies the canonicalized hash R into the running T_H accumulator hash, before storing the L_{ID}, R, K_{Secret}, and $bootCtr$ in M_L. Next, it performs slightly different operations depending on the exact algorithm to be used, namely:

- If self-attestation is used, the current time (calculated using $attTime()$) is stored, as well as the leaf's boot time T_{boot}.
- If encryption-based attestation is used, the E_V derives and stores K_C, K_R, and K_B using the $hkdf()$ function, with K_{Secret} as the input key.

Lastly, the E_V sends a confirmation message containing L_{ID} to the leaf therein signaling to the L_{ID} that bootstrapping is over. If encryption-based attestation is used, the leaf also performs the same aforementioned key derivation behavior. Finally, the leaf updates its $F_{Provisioned}$ to true.

3.4.5 Bootstrap Finalization

The prior defined bootstrapping phase repeats until all input L_{ID} have been bootstrapped at each E_V. Prior to the attestation phase, however, the state must be updated and finalized at the E_V, as described in algorithm 3.2.

Algorithm 3.2: Bootstrap Finalization for E_V
Precondition: M_L contains all L_{ID} else bootstrap() $F_B \leftarrow canonicalize(T_H)$ LTS: F_B

More specifically, the E_V first ensures that M_L contains all child L_{IDs}. If the check fails, the E_V restarts the bootstrap phase for the missing L_{IDs}. If this check passes, however, the hash accumulator T_H is then canonicalized into the boot hash F_B, which is placed into long term storage.

3.4.6 Bootstrap Aggregation

Whilst the outlined behavior above defines the behavior in which a cluster, consisting of one edge and its L_{ID} is bootstrapped, data must also be aggregated further up the network hierarchy to the root. This can involve multiple additional levels of clusters (as E_{Vs} can also be L_{ID} to devices higher up the tree).

Therein, bootstrapping must, in-effect, be performed synchronously, wherein:

- Challenges are propagated down the network hierarchy to the final, bottom, layer of L_{ID},
- The L_{ID} perform bootstrapping with a parent E_V,
- The E_V itself performs the same behavior as it's L_{ID}, attaching its H_{att} to the F_B, and sending the result up to a higher parent.

This repeats on all levels until the root is reached, at which point the golden hash value is established, and (with the exception of self-attestation), the attestation challenges are propagated down the network in the same way.

3.5 On-Demand Attestation

Once bootstrapping is over and the E_V has switched phase, E_{Vs} performs periodic attestation cycles to continuously monitor child L_{ID}'s integrity. The following section therefore defines the attestation behavior for each of the proposed on-demand algorithms.

3.5.1 Encryption-Based Attestation

The behavior for encryption-based attestation is defined in algorithm 3.3.

Algorithm 3.3: On-Demand Encryption-Based Attestation for E_V and L_{ID}

Edge Verifier		**Leaf Node**
0. Initial State		
Precondition: $L_{ID} \notin M_{RL} \wedge L_{ID} \in M_L$		Precondition: $F_{provisioned}$
1. Create Challenge		
$Challenge \leftarrow (bootCtr,\ nonceCounter)$ $E_C \leftarrow authEncrypt(\text{“”}, K_C, Challenge)$		
	$L_{ID},\ E_C \longrightarrow$	
2. Handle Challenge		

		if $LTS(L_{ID}) \neq L_{ID}$ **then** $abort()$
		$Q \leftarrow authDecrypt(E_C, K_C, Challenge)$
		if $Q = \text{“”}$ **then**
		$H_{ATT} \leftarrow muHash(att())$
		$R \leftarrow canonicalize(H_{ATT})$
		$E_R \leftarrow \text{authEncrypt}(R, K_R, Challenge)$
		$nonceCounter \leftarrow nonceCounter + 1$
		$Challenge \leftarrow (bootCtr,\ nonceCounter)$
		else // go to unexpected challenge
		// retry, if decryption fails again, abort

	$\longleftarrow L_{ID}, E_R$	
3. Handle Response		
$R \leftarrow authDecrypt(E_R, K_R, Challenge)$ **if** $R \neq \perp$ **then** $T_H \leftarrow T_H \times R$ $M_{RL} \leftarrow M_{RL} \cup \{L_{ID}, R\}$ $nonceCounter \leftarrow nonceCounter + 1$		

3.5.1.1 Initial State

Before any attestation behavior is initiated at the E_V, a precondition is checked to ensure that the L_{ID} has not already undergone attestation during the current round, and that it is in fact a leaf that the E_V is responsible for.
Likewise, at the L_{ID}, a flag ($F_{provisioned}$) is checked to ensure that it has undergone bootstrapping.

3.5.1.2 Create Challenge

If the preconditions are passed, attestation first starts with the E_V generating a unique *Challenge* composed of two 6-byte counters (*bootCtr*, *nonceCounter*).

The former counter, *bootCtr*, is incremented upon the edge rebooting, and subsequently the boot/unexpected challenge protocol being triggered as defined in subsubsection 3.5.1.5. The *nonceCounter*, meanwhile, is incremented for each new challenge/response round. This process enables the creation of unique challenges for each attestation iteration, mitigating the effects of an attacker replaying a past challenge/response message.

Next, an empty string is encrypted into E_C with *authEncrypt*(), using the challenge key K_C, and *Challenge* as the nonce. Subsequently, the E_C and L_{ID} are sent to the L_{ID}.

3.5.1.3 Handle Challenge

When a challenge arrives at the L_{ID}, it first checks that its internal L_{ID} value matches the received L_{ID} value. If this check does not pass, the leaf aborts.

Next, it uses the *authDecrypt*() function to decrypt E_C into Q, using it's locally stored copy of K_C and *Challenge* which were first established during bootstrapping. If this is successful, and the decrypted output Q is an empty string, the leaf then computes the hash H_{ATT} of its *att*() function return value, and canonicalizes it into R. Following this, R is encrypted into E_R using the response key K_R, and the *Challenge* nonce.

Whilst the same nonce used for challenging the leaf can be reused for its response (since it has not been used with K_R yet), the *nonceCounter* must be incremented, and the *Challenge* recomputed, in anticipation of the next challenge-response round. With these operations complete, the leaf can then send a response to the edge, consisting of E_R and L_{ID}.

If, however, the decryption of E_C fails, or the "Q" value is not an empty string, the reboot protocol is instead triggered, and the leaf renegotiates a challenge with the E_V using the reboot protocol defined in algorithm 3.4. After the renegotiation decryption is attempted once more, if decryption returns now the empty string the protocol proceeds with the new value of Q. If decryption does not, the client aborts the protocol before calling the reboot protocol again.

3.5.1.4 Handle Response

When the response arrives back at the E_V, it decrypts E_R using it's locally stored copy of *Challenge* and K_R into R. If decryption is successful, R is then multiplied into the running T_H accumulator MuHash hash, and the L_{ID} as well as R are added to the map M_{RL} of all responded nodes and their hashes. Finally, the E_V updates the nonceCounter in anticipation of the next round.

3.5.1.5 Reboot/Unexpected Challenge

As mentioned previously, a reboot protocol is required in case the edge reboots or the leaf receives an unexpected challenge. This process described in algorithm 3.4.

Algorithm 3.4: On-Demand Encryption-Based Reboot Protocol for E_V and L_{ID}

Edge Verifier		**Leaf Node**
..... 0. Initial State		
Precondition: $L_{ID} \in M_L$		Precondition: $F_{provisioned}$
..... 1. Boot/Unexpected Challenge		
		$K_T \leftarrow random()$ $\tau \leftarrow random()$ $E_B \leftarrow authEncrypt(K_T, K_B, \tau)$
	$\xleftarrow{L_{ID}, E_B, \tau}$	

$K_T \leftarrow authDecrypt(E_B, K_B, \tau)$
if $K_T \neq R \wedge K_T \neq \perp$ **then**
  $Challenge \leftarrow (bootCtr,\ nonceCounter)$
  $\tau \leftarrow random()$
  $E_C \leftarrow authEncrypt(Challenge, K_T, \tau)$		

	$\xrightarrow{L_{ID}, E_C, \tau}$	

		$Challenge \leftarrow authDecrypt(E_C, K_T, \tau)$
		if $LTS(L_{ID}) = L_{ID}$ **then**
		  // Update local copy of Challenge

Like with attestation, the reboot protocol has an initial state precondition. Namely, the L_{ID} must have already been bootstrapped to the E_V, and both devices must be aware of this.

Rebooting starts with the leaf generating a single-use random key K_T and random nonce τ. K_T is then encrypted using τ as the nonce, and the shared reboot key K_B that was previously established during initial bootstrapping. Once this is complete, the encrypted value E_B is sent to the E_V, together with the plaintext τ and L_{ID}.

Upon receiving the message, the E_V first decrypts E_B using its locally stored K_B value and the received τ nonce. Next, it verifies that the resulting K_T value is both correctly decrypted, and that it is different from the standard shared response key K_R. If these checks pass, the edge creates a new *Challenge* value using its local *bootCtr* and *nonceCounter*, and creates a new random single-use nonce τ. It then generates E_C by encrypting *Challenge* using the received key K_T and newly generated τ value, before sending E_C, τ, and the L_{ID} to the leaf.

When the leaf receives this message, it decrypts E_C, thus getting a new *Challenge* value. If the received L_{ID} value matches its own stored value, the local copy of *Challenge* is then updated to the received value.

3.5.2 Nonce-Based Attestation

The following section presents nonce-based attestation, with the pseudocode provided in algorithm 3.5.

Algorithm 3.5: On-Demand Nonce-Based Attestation for E_V and L_{ID}

Edge Verifier		**Leaf Node**
	0. Initial State	
Precondition: $L_{ID} \notin M_{RL} \wedge L_{ID} \in M_L$		Precondition: $F_{provisioned}$
	1. Create Challenge	
Challenge $\leftarrow random()$		
	L_{ID}, $Challenge$ $\longrightarrow$	
	2. Handle Challenge	
		if $LTS(L_{ID}) \neq L_{ID}$ **then** $abort()$
		$H_{ATT} \leftarrow muHash(att())$
		$R \leftarrow canonicalize(H_{ATT})$
		$S \leftarrow generateHMAC((Challenge \| \| K_{Secret}), R)$
	$\longleftarrow$ L_{ID}, S, R	
	3. Handle Response	
if $!verifyHMAC((Challenge \| \| K_{Secret}), R, S)$		
then $abort()$		
$T_H \leftarrow T_H \times R$		
$M_{RL} \leftarrow M_{RL} \cup \{L_{ID}, R\}$		

3.5.2.1 Initial State

As with encryption-based attestation, the E_V and L_{ID} undergo the same preconditions. Namely, the edge verifying the leaf has not already been attested, and that the leaf is a child of it, and the L_{ID} verifying it has previously been bootstrapped.

3.5.2.2 Create Challenge

Subsequently, the E_V first generates a $Challenge$ using the $random()$ function, before sending the $Challenge$ and L_{ID} to the L_{ID}.

3.5.2.3 Handle Challenge

When a challenge arrives at an L_{ID}, it verifies that the received L_{ID} matches its internal L_{ID} value, before computing the hash H_{ATT} of the $att()$ function return value, and canonicalizing it into R.

Unlike in encryption-based attestation, however the L_{ID} then creates an authentication tag of R, by using a concatenation of K_{Secret} and the $Challenge$ as the signing key.

Lastly, the L_{ID} concludes by sending the canonicalized hash R, signature S, and L_{ID} back to the parent E_V.

3.5.2.4 Handle Response

When a response is received at the E_V, it then verifies the received authentication tag S by using the same process for establishing a signing key as the L_{ID} did, and then comparing the generated signature to the received signature. If the check passes, R is multiplied into the running T_H accumulator hash, and the L_{ID} as well as R are added to the map M_{RL} of all responded nodes and their hashes.

3.6 Self Attestation

Once the bootstrapping phase is over, the self-attestation design enables each leaf node L_{ID} to perform self-attestation upon receiving the Uniform Attestation Trigger (UAT) signal. Each node then reports the resulting attestation message to its assigned edge verifier E_V. Subsequently, the edge verifier performs self-attestation to prove its own integrity to the root verifier R_V, following the same procedure used by the leaf nodes L_{ID}. Finally, the overall integrity of the swarm is verified at the root verifier R_V based on the aggregated results received from all edges.

3.6.1 Self Attestation Phase

The Self-Attestation phase begins when the UAT occurs, which is a locally generated, periodically scheduled trigger at the node. In the simulation, the UAT interval is drawn from a uniform distribution within a predefined range, ensuring that attestations occur at randomized but bounded times. Once the UAT is received, the node first measures its current uptime as $T_{up} \leftarrow uptime()$. This value represents the time elapsed since the most recent reboot and provides a trusted reference for ensuring freshness. In parallel, the node maintains a persistent $bootCtr$, which remains constant throughout a single boot session and increments only when the device reboots. The combination of $bootCtr$ and T_{up} is then used to construct a monotonic nonce. This ensures that uptime values are always bound to the correct boot session, guaranteeing nonce uniqueness both within a single boot session and across reboots, thereby enabling effective replay protection. The node then computes its runtime attestation value as $H_{ATT} = muHash(att())$, and canonicalizes the hash into R. Then the node authenticates itself using a domain separated subkey derived from its provisioned symmetric key K_{Secret}. Specifically, it computes a message authentication code as

$$H_M \leftarrow HMAC(K_i \parallel nonce, R),$$

where

$$K_i = K_{secret} \parallel i,$$

For example, given K_{Secret_1}, which is used for self-attestation messages, and K_{Secret_2}, which is used for reboot authentication. This binding ensures that messages of other types cannot be replayed in other parts of the protocol, making attacks more difficult. The node then transmits the attestation reports as

$$(L_{ID}, T_{up}, R, S)$$

to its managing edge verifier.

3.6.2 Verification Phase

Once the attestation report is received, the edge verifier reconstructs the same nonce and validates the signature to confirm message integrity and authenticity. After successful verification, the verifier performs a drift window check to ensure that the reported uptime value T_{up} aligns with the expected time since the last accepted report. This check prevents replay attacks by confirming that each node's local clock remains consistent with the verifier atomic reference time. The verifier computes the expected uptime as

$$elapsed \leftarrow T_{now} - T^{E}_{seen}; Tup^{exp} = last\_Tup[L_{ID}] + elapsed,$$

where T_{now} is measured using an atomic reference timer at the edge. The variables $last\_Tup[L_{ID}]$ and T^{E}_{seen} are initialized from the first accepted attestation report, ensuring that subsequent rounds have a consistent baseline. The received T_{up} is accepted only if it lies within a permissible drift window:

$$(T^{exp}_{up} - \Delta \leq T_{up} \leq T^{exp}_{up} + \Delta),$$

where Δ defines the maximum allowable deviation between the leaf and edge clocks. If the drift check fails, the E_V notifies the leaf to reboot and re-enter the reboot phase. Through this approach, the protocol accurately tracks drift between nodes and the verifier. If all the checks pass, R is multiplied into the running T_H accumulator hash, and the L_{ID} as well as R are added to the map M_{RL} of all responded nodes and their hashes.

3.6.3 Resynchronization Phase

This phase is triggered when the edge verifier detects that the reported uptime T_{up} deviates from the expected value beyond the allowed tolerance window. Such a deviation indicates that the L_{ID} clock has drifted away significantly. Rather than immediately rejecting the attestation, the E_V initiates the reboot request. To authenticate the reboot request, the E_V derives the reboot-domain key

$$K_{Secret_2} = K_{Secret} \parallel 2,$$

and computes a reboot tag as

$$S_R = HMAC(K_{Secret}, L_{ID} \parallel S).$$

This binds the reboot request to both the L_{ID} identity and the attestation value S, ensuring that the request cannot be forged or replayed. The L_{ID} verifies this tag using the same-domain separated key. Only if the verification succeeds, the leaf enters the reboot phase; otherwise, the request is ignored.

Algorithm 3.6: Self Attestation and Verification for L_{ID} and E_V

Edge Verifier		**Leaf Node**
..... 0. Initial State		
bootCtr Synchronized see subsection 3.6.4		*Precondition: UAT signal received*
..... 1. Self Attestation Phase		
		$T_{up} \leftarrow uptime()$ $H_{ATT} \leftarrow muHash(att())$ $R \leftarrow canonicalize(H_{ATT})$ $nonce \leftarrow bootCtr \parallel T_{up}$ $K_1 \leftarrow K_{Secret} \parallel 1 \parallel nonce$ $S \leftarrow generateHMAC((K_1, R)$
	$\xleftarrow{(L_{ID}, T_{up}, R, S)}$	
..... 2. Verification Phase		

$nonce \leftarrow bootCtr \parallel T_{up}$
$K_1 \leftarrow K_{Secret} \parallel 1 \parallel nonce$
if $!verifyHMAC((K_1, R, S)$
then $abort()$
$T_{now} \leftarrow attTime()$
$elapsed \leftarrow T_{now} - T^E_{seen}$
$T^{exp}_{up} \leftarrow last\_Tup[L_{ID}] + elapsed$
if $(T^{exp}_{up} - \Delta \leq T_{up} \leq T^{exp}_{up} + \Delta)$ **then**
$last\_Tup[L_{ID}] \leftarrow T_{up}$; $T^E_{seen} \leftarrow T_{now}$
$T_H \leftarrow T_H \times R$
$M_{RL} \leftarrow M_{RL} \cup (L_{ID}, R)$
else		

..... 3. Resynchronization Phase (if times are out of range)		
$K_2 \leftarrow K_{Secret} \parallel 2$ $S_R \leftarrow generateHMAC(K_2, L_{ID} \parallel S)$		
	$\xrightarrow{reboot\_request, S_R}$	
		$K_2 \leftarrow K_{Secret} \parallel 2$ **if** $!verifyHMAC(K_2, L_{ID} \parallel S, S_R)$ **then** $abort()$ *enters reboot phase see subsection 3.6.4*

3.6.4 Reboot Update Phase

Before each attestation cycle, the leaf and the edge verifier must have a synchronized view of the leaf boot state. The Reboot Update Phase handles this scenario. As shown in algorithm 3.7, when a leaf node detects a reboot, it increments its local boot counter and records the corresponding boot timestamp. The node forms a message M_{reboot} containing its identifier, updated boot counter, and boot time, and signs it using an HMAC derived from its local symmetric key and nonce. The edge verifier validates this signature and ensures that the received counter is strictly larger than the previously stored one. Upon successful verification, the edge updates its local record $M_L[L_{ID}]$ with the new boot information, thereby synchronizing the node's freshness state before the self-attestation cycle begins.

Algorithm 3.7: Reboot Update between L_{ID} and E_V

Edge Verifier		**Leaf Node**
	0. Initial State	
		Precondition: Detects reboot
	1. Reboot (triggered on leaf reboot)	
		LTS: $bootCtr \leftarrow bootCtr + 1$;
		$T_{boot} \leftarrow uptime()$
		$nonce \leftarrow bootCtr \parallel T_{boot}$
		$M_{reboot} \leftarrow (L_{ID}, bootCtr, T_{boot})$
		$K_2 \leftarrow K_{Secret} \parallel 3 \parallel nonce$
		$S \leftarrow generateHMAC((K_2, M_{reboot})$
	$\xleftarrow{M_{reboot}, S}$	
$K_2 \leftarrow K_{Secret} \parallel 3 \parallel nonce$		
if $!verifyHMAC((K_2, M_{reboot}, S)$ **then**		
$abort()$		
if $bootCtr \leq M_L[L_{ID}].bootCtr$ **then**		
$abort()$		
LTS: $M_L[L_{ID}].T^E_{boot} \leftarrow attTime()$		
LTS: $M_L[L_{ID}] \leftarrow (bootCtr, T_{boot})$		
		continue to Self-Attestation Phase

3.6.5 Granularity Depth

- **Level 0:** The root verifier validates swarm integrity by comparing the aggregated cluster hash (MuHash) received from all edge verifiers with the expected cluster hash computed during bootstrapping.
- **Level 1&2:** When a mismatch is detected, the root requests a full report from the E_V until all registered nodes are covered. Each full report contains the fields: *nodeID, y_runtime, T_up, sourceEvId, status.* These details allow the root to determine which edge verifier cluster is faulty and which specific nodes are compromised.

3.7 Verification

Algorithm 3.8: Attestation at E_V

Precondition: $M_{RL} = M_L$ **else** $bootstrap()$
$F_R \leftarrow canonicalize(T_H)$
if $F_R \neq F_B$ **then**
  $flag()$
LTS: $F_S \leftarrow F_R$
$reset()$

Once the attestation is done, regardless of the chosen algorithm, the attestation results are then verified. As laid out in algorithm 3.8, the edge verifier E_V first checks completeness by confirming that all expected leaves $L_1, L_2 ... L_n$ appear in both maps M_L and M_{RL}, ensuring that every assigned leaf has responded. Upon successful verification, the running hash T_H is canonicalized into a final runtime hash F_R, which is then copied into F_S for future reference.

The verifier then compares the runtime hash F_R against the baseline hash F_B established during the boot phase. In case F_R does not match F_B, the function $flag()$ checks M_L and M_{RL} to identify which exact node is incorrect (by identifying which L_{ID} does not have the same response as it did during bootstrapping). Otherwise, the E_V's state is reset and a new attestation cycle is initiated.

4

Evaluation

4.1 Introduction

In order to evaluate the two classes of FLASH algorithms, experiments were conducted using two different implementations, namely: Hardware-based on-demand attestation using a Raspberry Pi and Arduinos, and simulated self-attestation using OMNeT++. As shown in Figure 4.1, the simulation utilized the timing measurments obtained from the proof of concept (PoC).

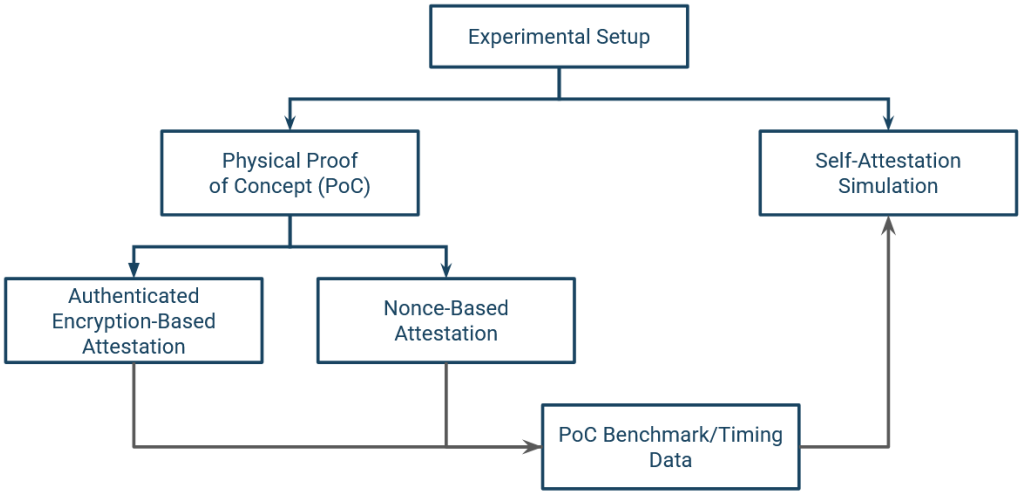


Figure 4.1: Illustration of Implementations

4.2 Proof of Concept Implementation

To evaluate the proposed algorithms under realistic conditions, two PoC implementations were created. The first implementation applied the encryption-based algorithm variation, whilst the second applied the nonce-based variation. The setup for the two experiments was largely the same, with both experiments sharing as many commonalities as possible, therein enabling comprehensive assessment of performance in a real-world environment.

4.2.1 Hardware Setup

The edge verifier in both cases was a Raspberry Pi Model 5 featuring a Broadcom BCM2712 Arm Cortex A76 processor running at 2.4GHz. The Raspberry Pi was provisioned with Raspberry Pi OS Lite running kernel version 6.12 - a headless OS based on Debian 12. Only the necessary software packages needed to run the code were utilized, and the unit was accessed via SSH.

At the leaf node layer, ESP32 Wroom-based Arduino micro-controllers were employed as leaf provers. These were responsible for computing preliminary hashes.

Communication, meanwhile, was handled over a wireless access point which was configured to broadcast in the 2.4GHz range, creating a local access network for devices to communicate over. This was selected since network delays were excluded from the data, and therefore no special considerations were required.

4.2.2 Codebase Setup

The core codebases for the PoCs were developed such that it was both easily modifiable to suit both algorithm variations, and portable - ensuring that it could in the future be tested on a wide range of systems utilizing different hardware configurations. Additionally, particular care was taken when developing the structure of the code, such that data could be collected for how long different parts of the algorithms required (see: subsection 4.2.4).

In regards to edge devices (and the root device, though this was not tested during data collection), a shared (and configurable) codebase was written in Golang. To aid in the comparison of the two algorithm variations, the codebase was (as much as possible) kept the same between the two - with only the necessary differences being implemented to suit each form of attestation.

For the ESP32 leaf provers, the codebase was developed in C++ using the PlatformIO development environment, using the Arduino core - chosen for its efficiency in managing embedded system tool-chains and libraries. As was the case for the edge verifier code, this codebase was to a large extent kept the same between variations.

4.2.2.1 Chosen Codebase Libraries

Whilst the algorithms proposed in this thesis have been created such that they do not rely on any one specific crytographic or code library, the PoC implementations were developed using just one library/implementation per required feature.

In the case of encryption-based attestation, authenticated encryption was implemented using the ChaCha20-Poly1305 algorithm [36], primarily due to its security, ease of implementation, and high performance.
The HKDF implementation, meanwhile, (required for encryption-based attestation) utilized the SHA256 hashing algorithm. Whilst other algorithms, such as BLAKE2s,

could provide greater performance, this was not required since key-derivation occurs during bootstrapping, and therefore does not affect attestation performance. Furthermore, SHA256 is included in the Arduino core framework, and therefore no additional on-device storage was consumed at the leafs.

For nonce-based attestation, the only required library was BLAKE2s, used in HMAC generation and verification. Since this behavior occurs during attestation, performance was the greatest consideration, therein the selection of BLAKE2s.

A key component shared across all devices, codebases, and algorithms, is the MuHash algorithm library, originally implemented in C++. To maintain consistency in cryptographic operations, the same MuHash library was deployed across all cases. Interoperability between C++ and Go was achieved through the use of the 'Simplified Wrapper and Interface Generator' (SWIG), which generates the necessary bindings to enable the C++ MuHash library to be invoked directly within Golang.

4.2.3 Testing Procedure

To efficiently test the developed PoCs, a minimal network consisting of one edge verifier was set up. For this configuration, separate tests were run to gather data for scenarios wherein the edge communicated with one, two, four, and six child leaf nodes, respectively. For each test, the devices progressed through one bootstrapping cycle, and 200 runtime attestation and verification cycles.

Note that the edge verifiers did not attest themselves and, additionally, no root verifier was present. The reasoning behind the decision to implement a simplified network was that it allowed for precise data collection from each device, and each of its operations, while providing us with a greater understanding of the effects of scaling the number of leaf nodes for each edge.

4.2.4 Data Collection & Analysis

In the case of the edge verifier, data collection was accomplished using Golang's inbuilt profilers. For both on-demand algorithms, data was collected for several different points in the code, corresponding to different parts of the algorithms. More specifically, data was collected for the:

- Time required for attestation challenge creation at the E_V,
- Time required for challenge handling at a L_{ID},
- Time required for response handling at the E_V,
- Time required for attestation verification at the E_V,
- Memory statistics before an attestation round was started and,
- Memory statistics after attestation and verification were complete.

All measurements at the E_V were collected using Golang's inbuilt time package. This enabled the placement of monotonic clocks in the code, which were used to measure the elapsed time between two points (start and end).

An important factor to note in regards to data collection is, however, the following: Whilst FLASH has been created such that edge verifiers can create challenges, serialize/send them, and receive responses in parallel for leaf node counts greater than one, the timings that will be collected are instead the summed times of each parallel execution (i.e, the sum of times required for each leaf node).

This decision has been chosen such that the PoC demonstrates an upper bound, worst case scenario, rather than a best case. This is applicable for instance in implementation scenarios where the edge verifier is very resource constrained, and thus can't handle parallel execution. However, to allow for an understanding of FLASH in its intended use case, section 7.1 will additionally extrapolate and prove the parallel performance capabilities.

In the case of leaf provers meanwhile, data was only collected on how much time each challenge-handling cycle required.

4.3 Simulation

To complement the PoC implementation, large-scale evaluations were conducted through simulation. Although the PoC demonstrates feasibility on real hardware, it is inherently limited in terms of the number of devices that can be deployed. In contrast, simulation makes it possible to model the full hierarchical architecture and observe how communication flows across the systems. This enables controlled experimentation at scale, allowing us to study scenarios with hundreds or thousands of nodes with varying patterns. The following subsection describes the simulation setup used to evaluate the self-attestation algorithm under such large-scale conditions.

4.3.1 Simulation Environment

The simulation experiments were conducted using the OMNeT++ simulator, which was extended with custom C++ modules to implement the proposed attestation protocol. For large-scale evaluations, the simulations were transitioned to Cmdenv to avoid the overhead of the graphical interface and to ensure scalability. Cryptographic functions such as MuHash, and HMAC were modeled using delay parameters derived from benchmark measurements on Raspberry Pi and Arduino devices, enabling realistic timing behavior within the simulation.

4.3.2 Network Topology

The topology was implemented as a tree in OMNeT++, where nodes connect to their designated edge verifier, and all edge verifiers are connected to the root. In

addition, peer-to-peer communication between edge verifiers was enabled to support the report synchronization.

Level 0 - Leaf Nodes: Acts solely as provers that must undergo attestation.
Level 1 - EdgeVerifier: Acts as verifiers for the lower level by verifying their attestation, and as provers for the upper level by sending their own attestation.
Level 2 - RootVerifier: Acts solely as a verifier that verifies the lower level, aggregates report from all provers, and determines the final integrity of the entire architecture.

4.3.3 Simulation Parameters

Parameter	**Value**
Leaf Node (Arduino)	
MuHash Delay (D_{muhash})	341.24 μs
HMAC Delay (D_{hmac})	363.38 μs
Edge (Raspberry Pi)	
MuHash Delay (D_{muhash})	341.24 μs
HMAC Delay (D_{hmac})	363.38 μs
HMAC Verify Delay	7462.75 μs
MuHash Insertion Cost (per node)	26.90 μs
Root (Raspberry Pi)	
HMAC Verify Delay	7462.75 μs
MuHash Insertion Cost (per node)	26.90 μs

Table 4.1: Simulation parameters used in the evaluation. The HMAC verification delay appears larger because, in the PoC implementation, verification includes the challenge handling and additional processing

Table 4.1 lists the cryptographic delays used in the simulation. These values are derived directly from PoC measurements.

4.3.4 Simulation Setup and Assumptions

4.3.4.1 Code Setup

The implementation was designed to closely replicate the attestation workflow used in the PoC experiments while enabling scalable evaluation across large network configurations. The core of the simulation consists of three primary modules representing the hierarchical layers of the architecture: Leaf Node, Edge Verifier, and Root Verifier. Each module is implemented in a dedicated source file and paired with its corresponding header file, which defines the communication interfaces and data structures utilized during simulation. The behavioral logic of each module reflects the attestation phase defined in the proposed algorithm.

The overall network structure and interconnections between modules are defined using OMNeT++ Network Description (NED) language and the network configuration parameters such as the number of nodes per edge verifier, attestation intervals, delay models are specified within the main configuration file `omnetpp.ini`. This file also defines the simulation environment mode (Qtenv or Cmdenv), random seed settings, and output statistics.

The key components shared across all modules is the implementation of MuHash algorithm, located within the dedicated MuHash directory. This library provides the homomorphic hashing functionality used for aggregating attestation values at the edge and root levels. The entire project is compiled using OMNeT++ built in `opp_makemake` system, producing two primary executables: `swarmattestation.exe` and `swarmattestation_dbg.exe`. These binaries can be executed in both Qtenv and Cmdenv simulation environments. The `results` directory stores all generated output data, including scalar and vector files used for later analysis

By structuring the simulation in this modular fashion, the codebase enables controlled and repeatable experiments over a wide range of parameters. Furthermore, by integrating benchmark from the PoC implementation, the simulated behavior closely approximates that of the real-world hardware, thereby ensuring consistency between physical and simulated evaluations of the proposed self-attestation algorithm.

4.3.4.2 Simulation Model

The simulation models only the cryptographic processing delays associated with attestation as mentioned in the Table 4.1, specifically self-attestation, verification and aggregation at the L_{ID}, E_V and R_V. This design isolates the cryptographic bottleneck, which is the dominant cost in resource-constrained systems, and allows the evaluation to focus on the scalability of the attestation itself. Network delays are not explicitly modeled in the simulation.

4.3.4.3 Assumption

The following assumptions are made:

- **Trusted bootstrapping:** The bootstrapping phase is considered trusted, with values generated on trusted hardware that cannot be forged.
- **Clock reference:** The simulation uses a global clock, while each leaf node keeps a local uptime counter that resets on reboot. Edge verifier use an atomic reference from the global clock for drift checks, enabling consistent comparisons across nodes.
- **Static topology:** Nodes are assumed to be stationary. The mobility of the nodes and the changes in dynamic topology are not simulated [25][29].
- **Delay calibration:** Delay values were determined on the basis of original hardware measurements conducted on Arduino and Raspberry Pi devices.

- **Network model:** The network follows a tree-based topology, with nodes organized hierarchically from leaf nodes to edge verifiers to a root verifier [32].

4.3.5 Execution Procedure

A small-scale validation was first conducted in Qtenv to visually confirm event ordering and message flow. The configuration instantiated 10 leaf nodes, 2 edge verifier (5 leaf per edge) connected to a single root verifier. The run exercised one boot phase and several periodic self-attestation cycles. During this stage, successful drift-window freshness checks and consistency of the recomputed MuHash at the root were verified, confirming that the final integrity decision matched the expected aggregate. After correctness was established in Qtenv, experiments were transitioned to Cmdenv for scalability evaluation. In the large-scale setting, the full hierarchy was instantiated with 5,000 leaf nodes, with 4 edge verifier simulated under benchmark-derived cryptographic delays. Each configuration was repeated ten times with different random seeds to assess variabilities, and OMNeT++ scalar and vector outputs were collected for post-processing.

4.3.6 Data Collection and Analysis

Data was collected using OMNeT++ built in scalar and vector recording mechanisms, which capture event-driven performance statistics for each module in the simulation. During each execution, timing and computation related variables were recorded for all entities in the hierarchy. The resulting data was automatically stored as .sca (scalar) and .vec (vector) files within the results directory for subsequent analysis.

- Time required for self-attestation at the node L_{ID}
- Time required for verification and aggregation at the edge E_V
- Time required for final verification at the root R_V
- End-to-end cryptographic attestation latency

An important aspect of the simulation during data collection is that all timing measurements are obtained using OMNeT++'s built in simulation clock (simTime()), which provides a deterministic notion of time progression within the discrete event environment. Another key consideration is that all the cryptographic attestation verification and insertion operations at the E_V and R_V are serialized, ensuring that each operation is scheduled only after the previous one has completed. This simulation modeling choice is consistent with the PoC implementation and reflects realistic scenarios in which edge devices are resource constrained.

5

Results

This chapter presents the results of both the hardware PoC experiments and simulations conducted to evaluate the performance of FLASH.
More specifically, the chapter is divided such that the results of the on-demand PoC and self-attestation simulation are divided into three main sections, namely: section 5.1 presents the on-demand encryption-based PoC results, section 5.2 the on-demand nonce-based PoC results, and section 5.3 the self-attestation simulation results.

5.1 Encryption-Based PoC Results

The following section presents the on-demand encryption-based attestation algorithm data, with Table 5.1 presenting an overview of averages, medians, and standard deviations. For a more detailed understanding of the performance, the results are then subsequently divided into attestation phase timings (itself further divided into challenge creation, challenge handling, and response handling), verification phase timings, and combined attestation and verification phase memory statistics.

Number of Leaves	**1 Leaf**	**2 leaf**	**4 Leaf**	**6 Leaf**
Average Create Challenge Time (ms)	0.009	0.019	0.045	0.051
Median Create Challenge Time (ms)	0.009	0.018	0.038	0.041
Std Dev (ms)	0.006	0.007	0.036	0.038
Average Handle Response Time (ms)	0.042	0.085	0.172	0.258
Median Handle Response Time (ms)	0.040	0.080	0.161	0.243
Std Dev (ms)	0.006	0.013	0.033	0.047
Average Verification Time (ms)	36.075	35.495	35.699	35.637
Median Verification Time (ms)	33.736	33.760	33.850	33.781
Std Dev (ms)	4.096	3.811	5.648	5.656
Average System Memory Usage (MB)	13.51	13.49	13.56	13.77
Std Dev (MB)	2.77	2.73	2.38	2.21

Table 5.1: Overall Averages, Medians, and Standard Deviations for Encryption-Based Attestation & Verification

5.1.1 Attestation Timing Results

As mentioned previously, the attestation timings are divided into separate sections, corresponding to the three phases of the attestation algorithms provided in subsection 3.5.1. The reasoning for this is twofold: not only does this allow for data collection which excludes network delays or other outside factors, but it also provides a better understanding of the performance of the algorithm(s), in turn allowing for more comprehensive analysis in the discussion (see chapter 7).

5.1.1.1 Challenge Creation Timing Results

The first step of the overall attestation is the creation of challenges. The unfiltered per-round time data for this process can be seen in Figure 5.1. Note that this data presents the summed challenge creation time required for all nodes. Whilst this data has a large number of time spikes, owing to the data collection process measuring wall time as opposed to CPU time, the time-scale for challenge creation is predominantly below 0.10 milliseconds across all configurations.

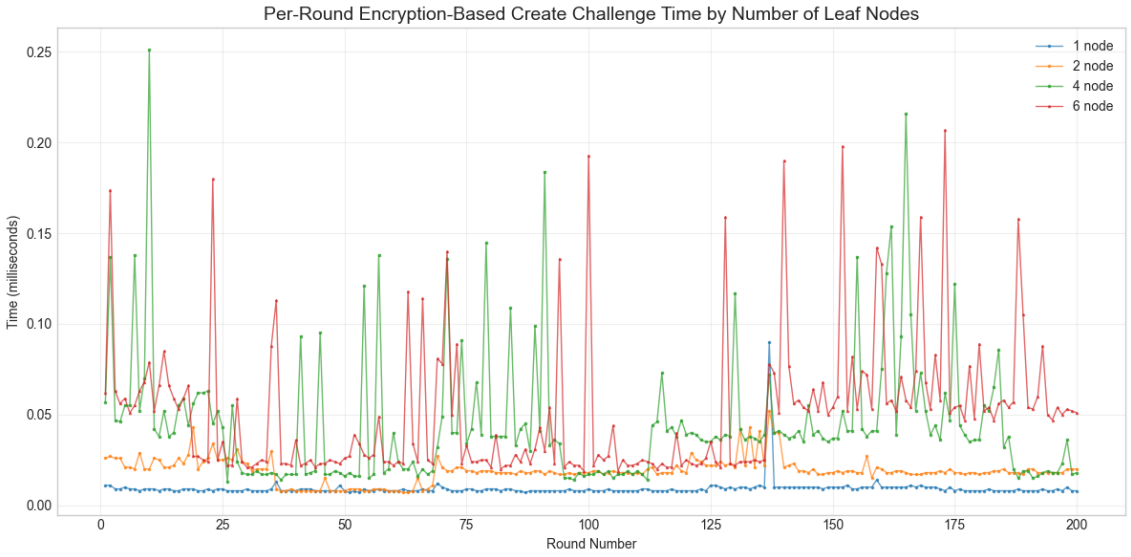


Figure 5.1: Challenge Creation Time (ms) required at E_V Across 200 rounds for 1, 2, 4, and 6 Leaf Nodes

The box and whiskers plots for for the create challenge portion of the encryption-based algorithm are presented in Figure 5.2.

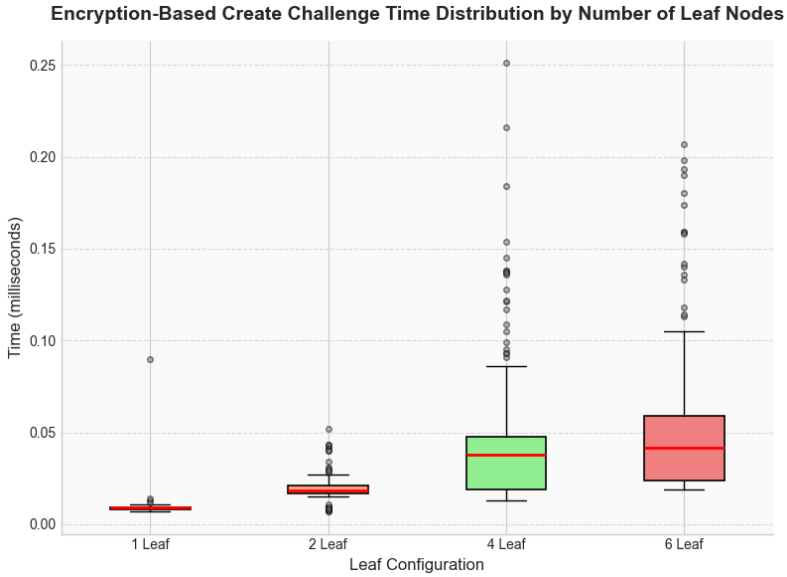


Figure 5.2: Box and Whiskers Plot for Challenge Creation Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

5.1.1.2 Challenge Handling Timing Results

Upon having been sent a challenge by an E_V, L_{IDs} must then generate a response, and send it back. As this behavior is the same across all nodes, and is only done once per round and node, the results presented in Figure 5.3 are therefore taken only for a single iteration, at a random node.

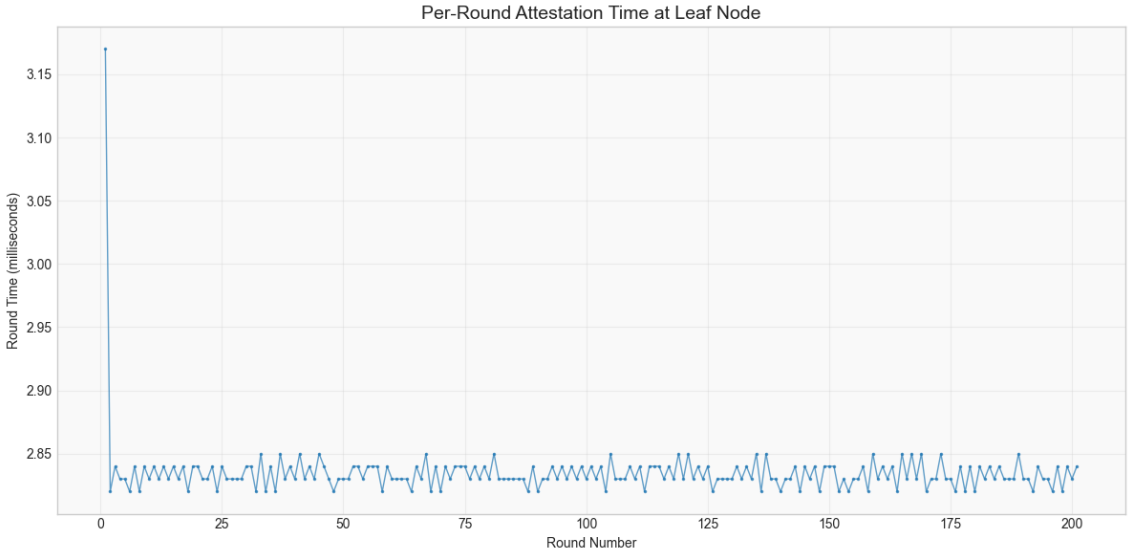


Figure 5.3: Attestation Time (ms) at L_{ID} Across 200 rounds

On average, the leaf required 2.835 ms to process the challenge and return a request, with a median time of 2.830 ms and standard deviation of 0.025 ms.

5.1.1.3 Response Handling Timing Results

The next operation of the attestation sequence is the handling of responses. As with challenge creation, the summed per-round data is presented in Figure 5.4.

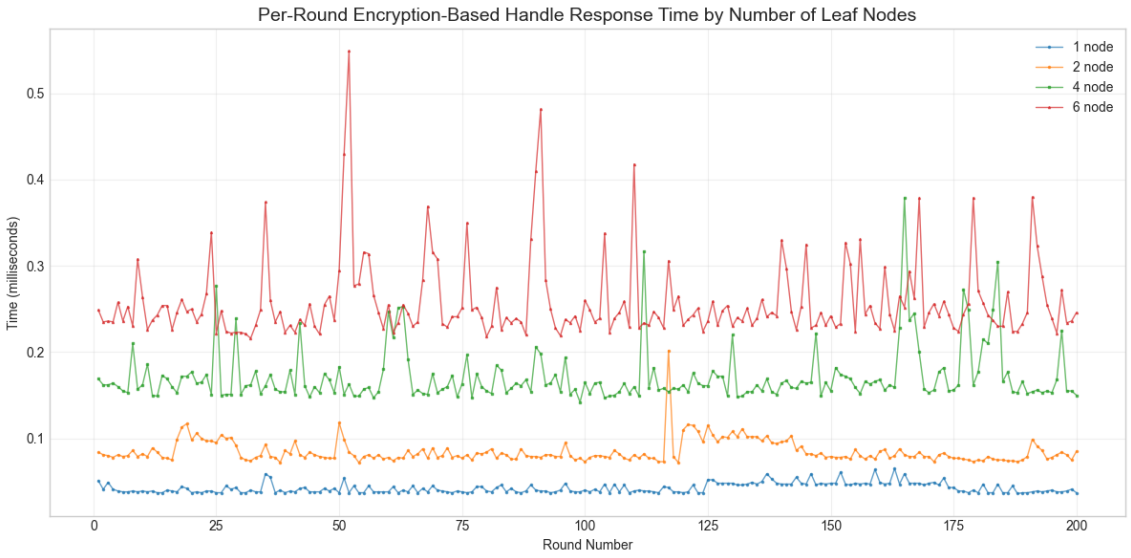


Figure 5.4: Response Handling Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

Notably, the data shows a clear and consistent distinction between all 4 configurations, despite the spikes in data. This is visible by the flier points in the box and whiskers plot presented in Figure 5.5.

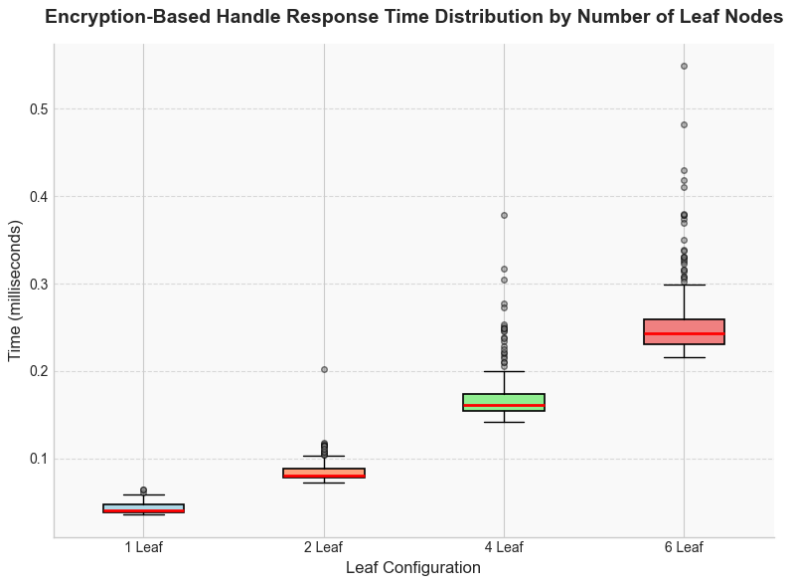


Figure 5.5: Box and Whiskers Plot for Response Handling Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

5.1.1.4 Verification Results

Finally for verification, which occurs after all nodes have responded and their data has been attested, the per-round data is presented in Figure 5.6. As is the case with the recorded attestation data, the data contains a significant amount of outliers, reflected in the visible spikes in the figure.

Due to the particularly noticeable 'bump' in verification time for the single-leaf configuration for rounds 125-170, the box and whiskers plot shown in Figure 5.7 presents only the data for the first 100 rounds.

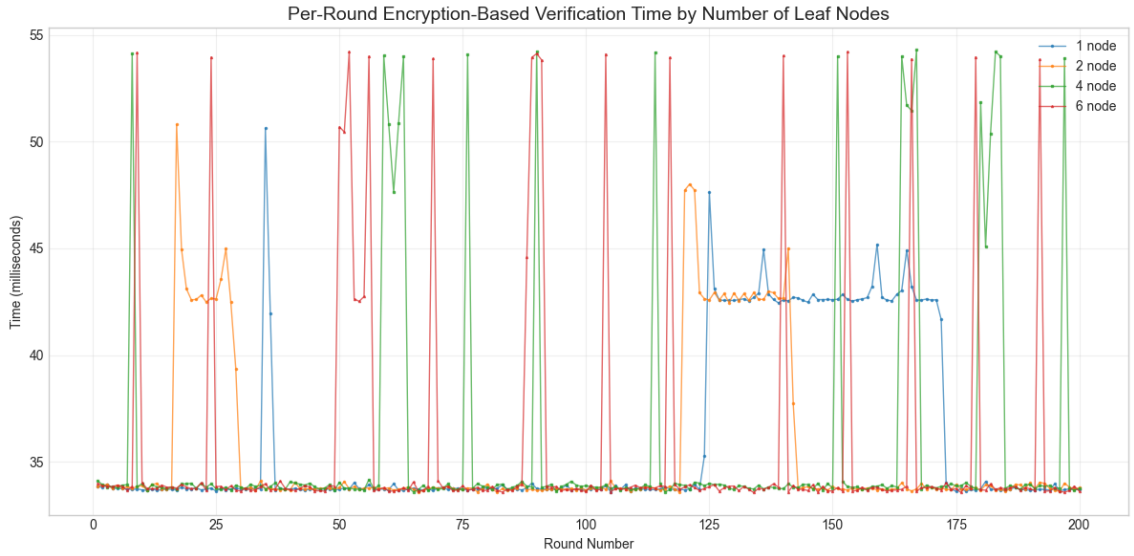


Figure 5.6: Verification Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

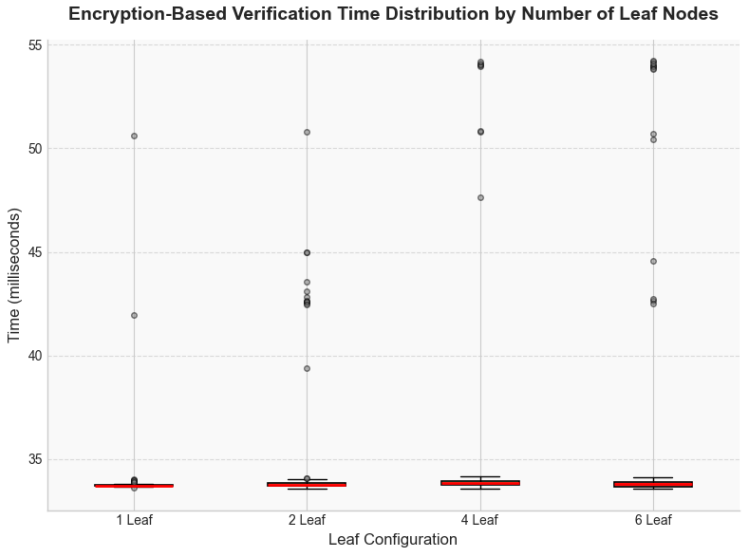


Figure 5.7: Box and Whiskers Plot for Verification Time (ms) at E_V Across 100 Rounds for 1, 2, 4, and 6 Leaf Nodes

5.1.1.5 Total Per-Round Memory Results

The following section provides memory usage statistics and data collected at the E_V over the course of 200 runs, at leaf node counts 1,2,4 and 6.

The average in-use allocated and reserved system memory required by the E_V are presented in Figure 5.8. Memory allocation was in the 3.33 to 3.61 MB range, with reserved system memory in the 13.49 to 14.77 MB range.

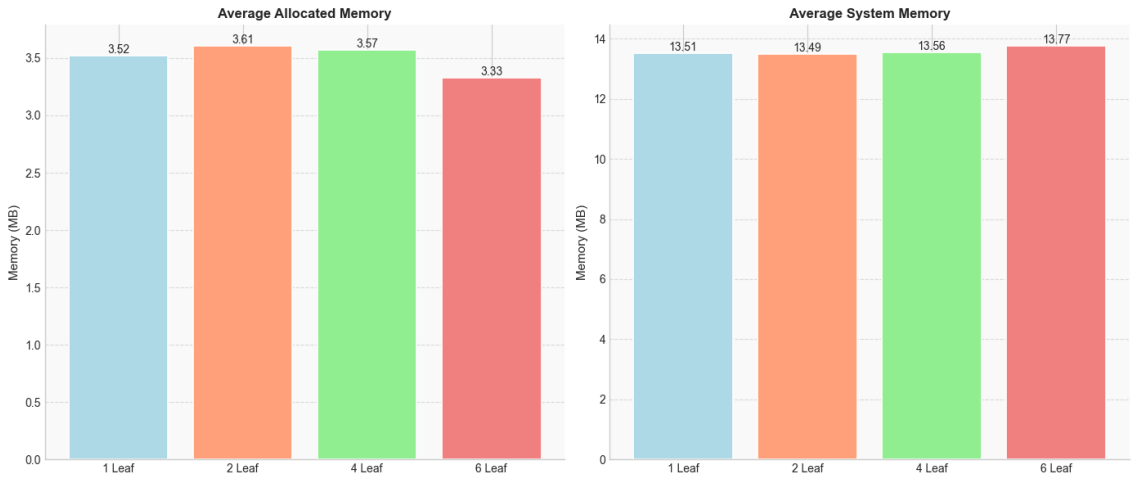


Figure 5.8: Average Allocated and System Memory Utilization (MB) During Attestation and Verification at E_V across 200 rounds for 1, 2, 4, and 6 Leaf Nodes

However, noteworthy is the fact that both system and allocated memory usage rose to far greater amounts during the 200 iterations, as seen in Figure 5.9, and Figure 5.10, respectively[1].

[1]This behavior is due to the way in which Golang handles and returns system memory. Upon testing at higher round counts (2000 rounds) for 1 node with stricter Golang memory management/garbage collection, system memory stayed at a consistent level once the enforced limit was reached.

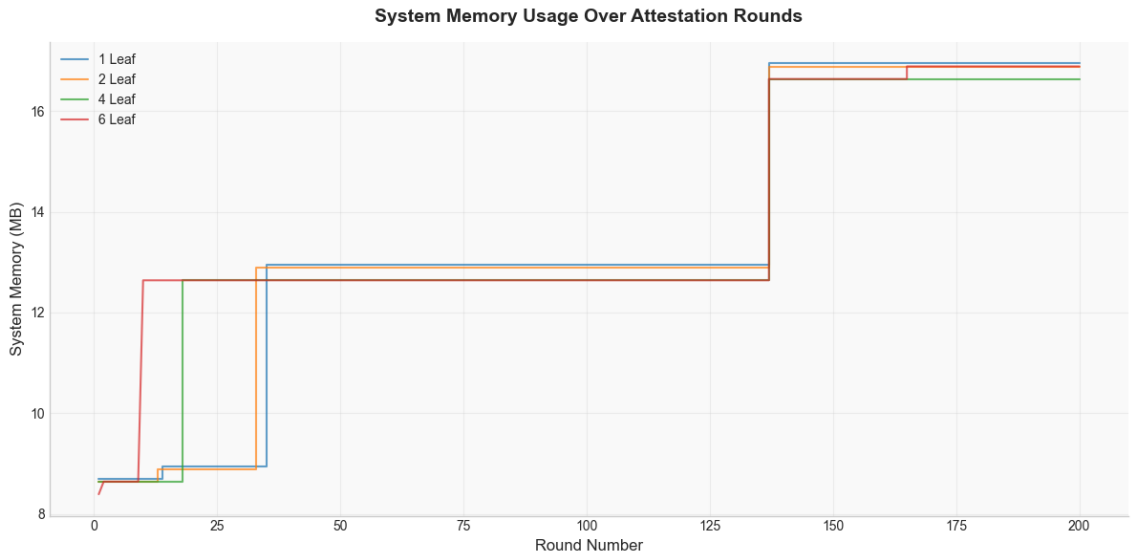


Figure 5.9: System Memory (MB) Utilized During Attestation and Verification at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

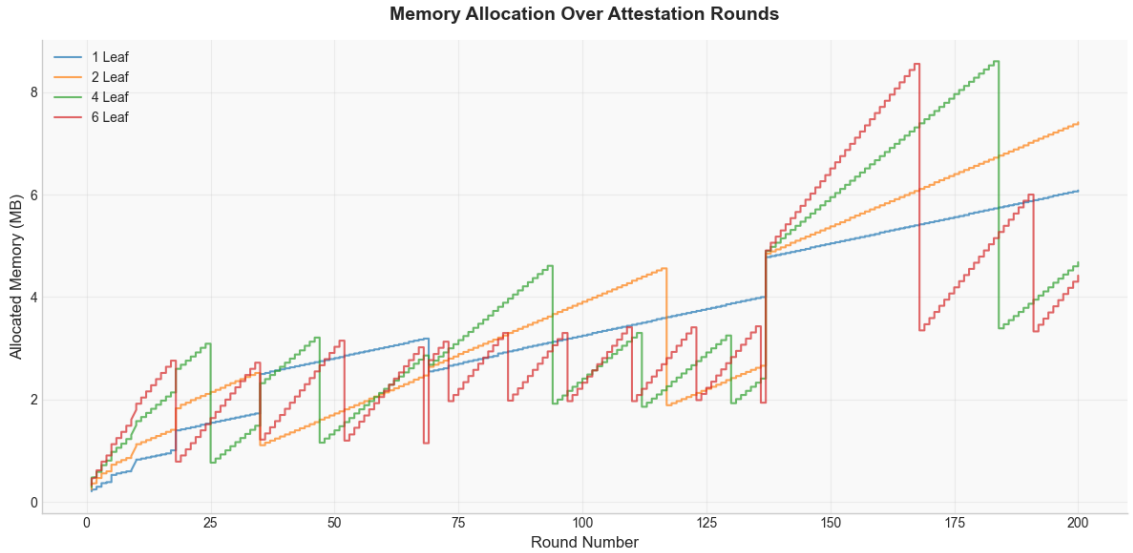


Figure 5.10: Allocated Memory (MB) Utilized During Attestation and Verification at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes

5.2 Nonce-Based PoC Results

The following section provides the results collected for nonce-based attestation, with Table 5.2 presenting an overview.

Number of Leaves	1 Leaf	2 leaf	4 Leaf	6 Leaf
Average Create Challenge Time (ms)	0.008	0.021	0.052	0.044
Median Create Challenge Time (ms)	0.008	0.018	0.039	0.036
Std Dev (ms)	0.002	0.011	0.036	0.039
Average Handle Response Time (ms)	0.051	0.100	0.197	0.309
Median Handle Response Time (ms)	0.050	0.094	0.184	0.280
Std Dev (ms)	0.010	0.020	0.043	0.080
Average Verification Time (ms)	36.271	34.954	34.745	35.936
Median Verification Time (ms)	33.777	33.680	33.709	33.837
Std Dev (ms)	4.276	4.471	4.022	5.832
Average System Memory Usage (MB)	13.22	13.36	13.87	13.76
Std Dev (MB)	2.74	2.66	2.30	2.11

Table 5.2: Overall Averages, Medians, and Standard Deviations for Nonce-Based Attestation & Verification

5.2.1 Attestation Timing Results

As discussed previously, and as done for encryption-based attestation, attestation time measurements were divided into challenge creation and response handling measurements to avoid network conditions and delays affecting the results.

5.2.1.1 Challenge Creation Timing Results

In regards to challenge creation, the unfiltered per-round time data can be seen in Figure 5.11. Notably, there is a consistent and visible separation of time required across the different leaf node configurations. This is despite the, like seen for encryption-based attestation, spikes that occur due to the measurement of wall time.

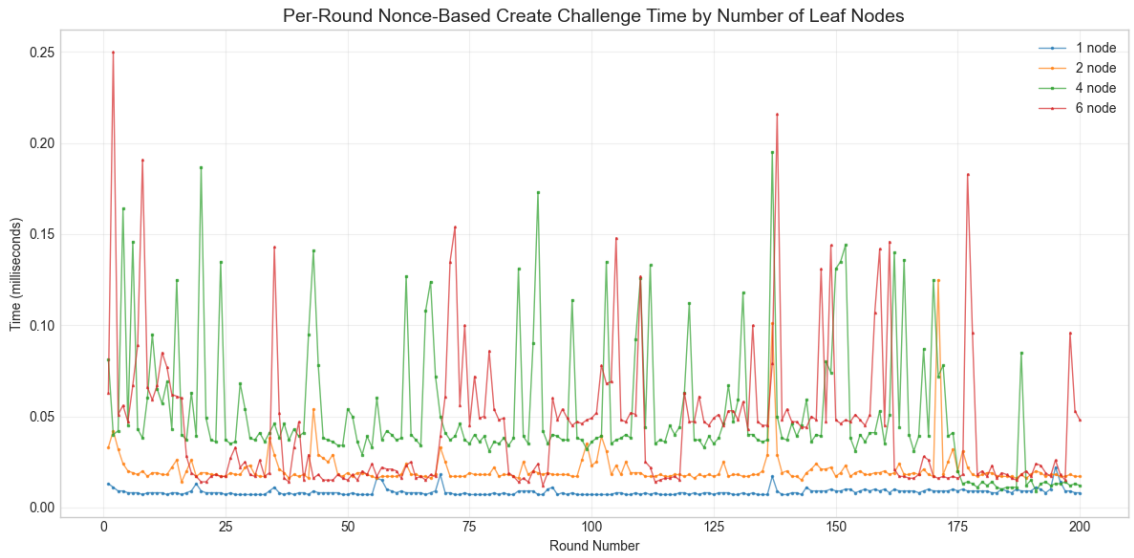


Figure 5.11: Challenge Creation Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes.

The box and whiskers plot in Figure 5.12 provides an overview of the distribution, inter-quartile range, and averages of the challenge creations times. Notably, the 4 leaf configuration has a greater average time (51.83 ms) than the 6-leaf configuration (44.48 ms).

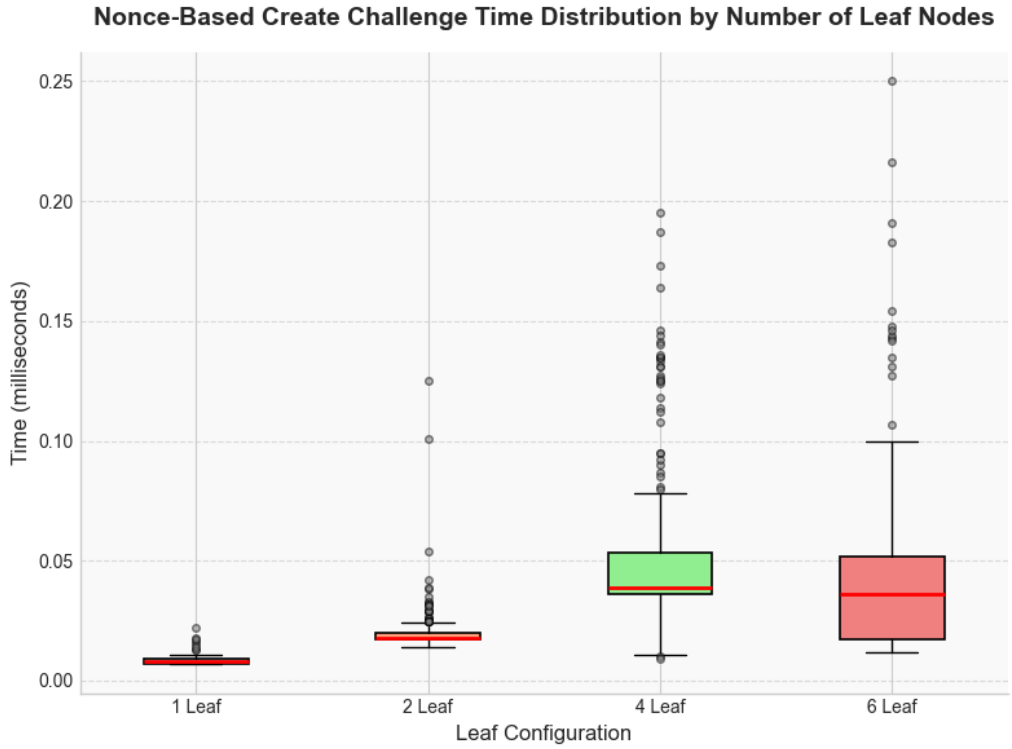


Figure 5.12: Box and Whiskers Plot for Challenge Creation Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes.

5.2.1.2 Challenge Handling Timing Results

Upon being challenged by an E_V, L_{IDs} must then generate a response, and send it back. As with encryption-based attestation, this data is displayed in Figure 5.13 for a randomly selected leaf.

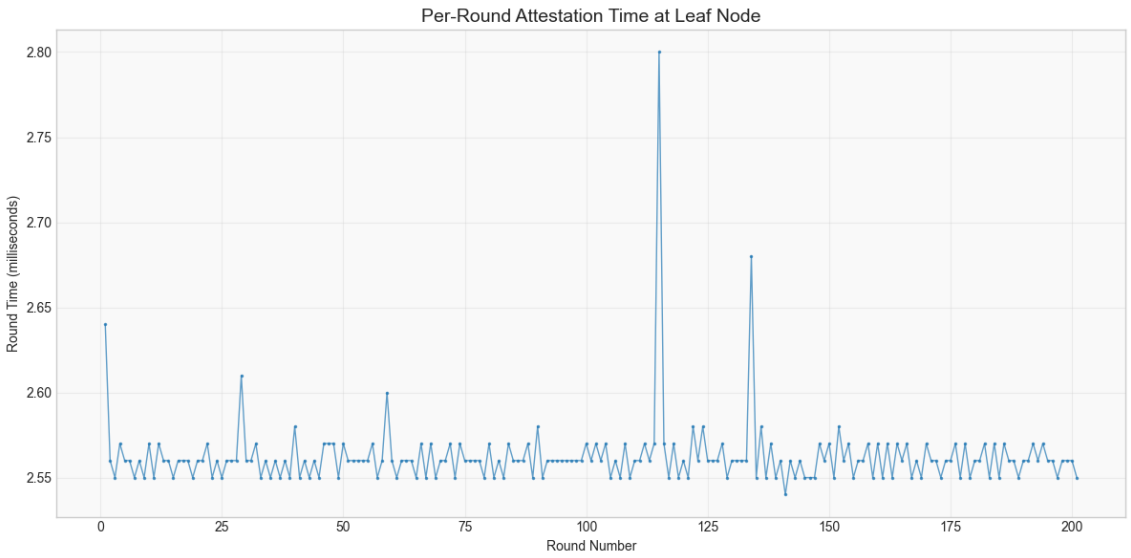


Figure 5.13: Attestation Time (ms) at L_{ID} Across 200 rounds

On average, the leaf required 2.562 ms to handle the challenge, build a response, and send it, with the data having a median value of 2.560 ms. The standard deviation, meanwhile, was 0.022 ms.

5.2.1.3 Response Handling Timing Results

In regards to response handling times, meanwhile, the per-round data is presented in figure Figure 5.14. As seen in the challenge response data, there is a consistent distinction between all configurations. However, this data shows far less spiking behavior than either the overall attestation data, or the create challenge data.

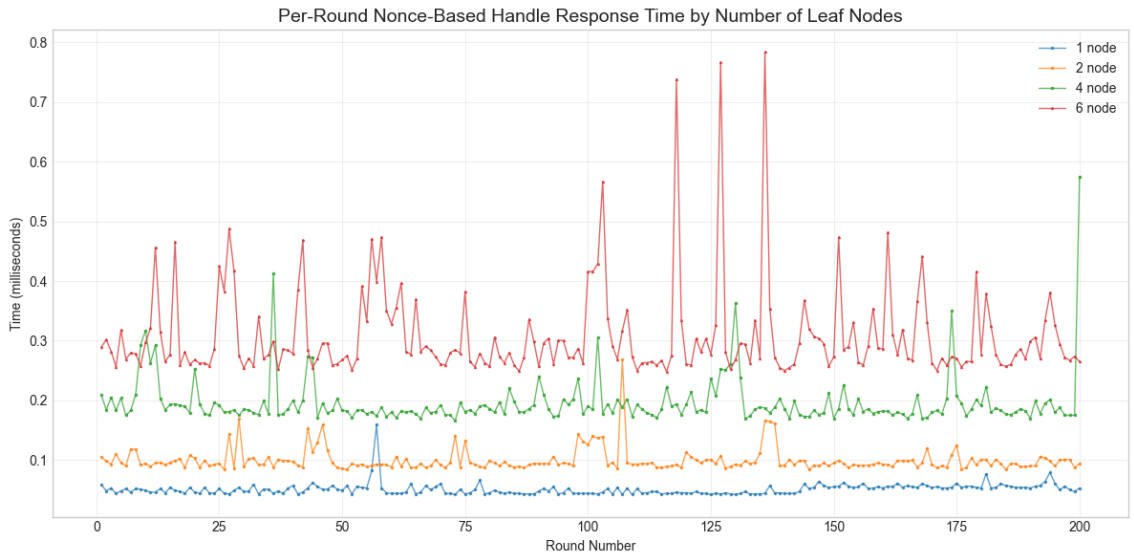


Figure 5.14: Response Handling Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes.

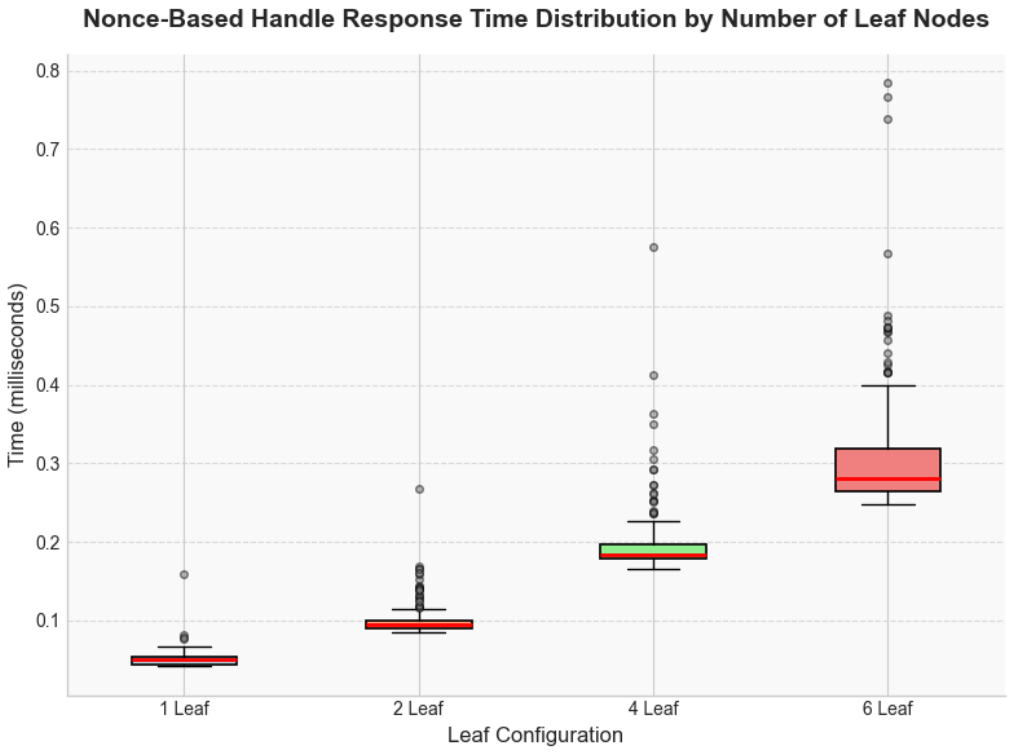


Figure 5.15: Box and Whiskers Plot for Response Handling Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes.

When looking at the box and whiskers plot in Figure 5.15, it is notable that the average time increases in a near linear fashion across the leaf configurations.

5.2.1.4 Verification Results

In regards to verification, the raw data per-round is presented in figure Figure 5.16. Whilst the averages for all leaf node counts are with a +/- 3.6% range, the data contains a significant amount of outliers, particularly for the 1 leaf node configuration from approximately round 140. For this purpose, the box and whiskers plot (Figure 5.17) presents the data for the first 100 rounds only, like for encryption-based verification.

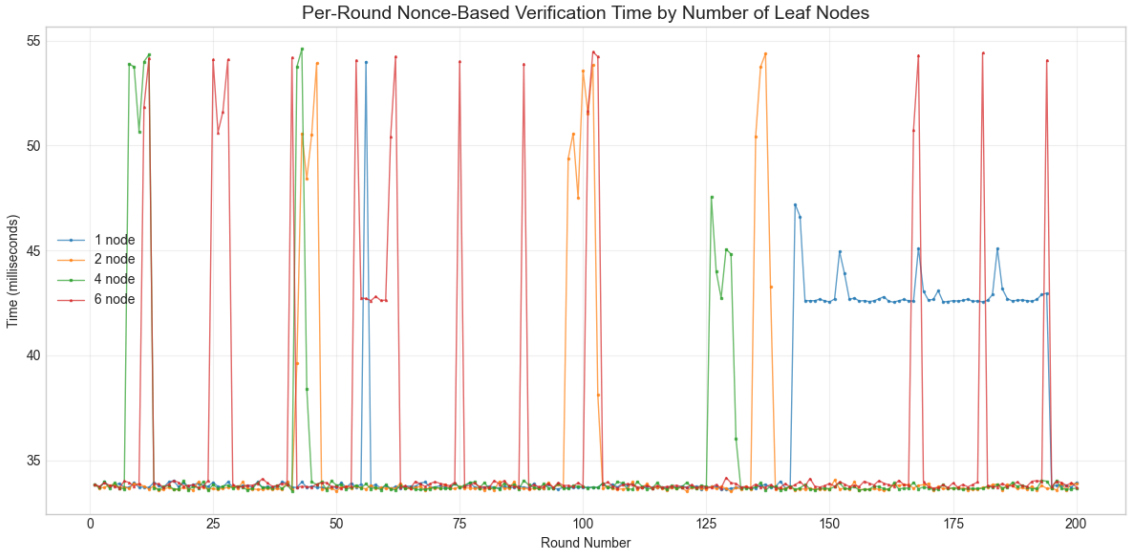


Figure 5.16: Verification Time (ms) at E_V Across 200 Rounds for 1, 2, 4, and 6 Leaf Nodes.

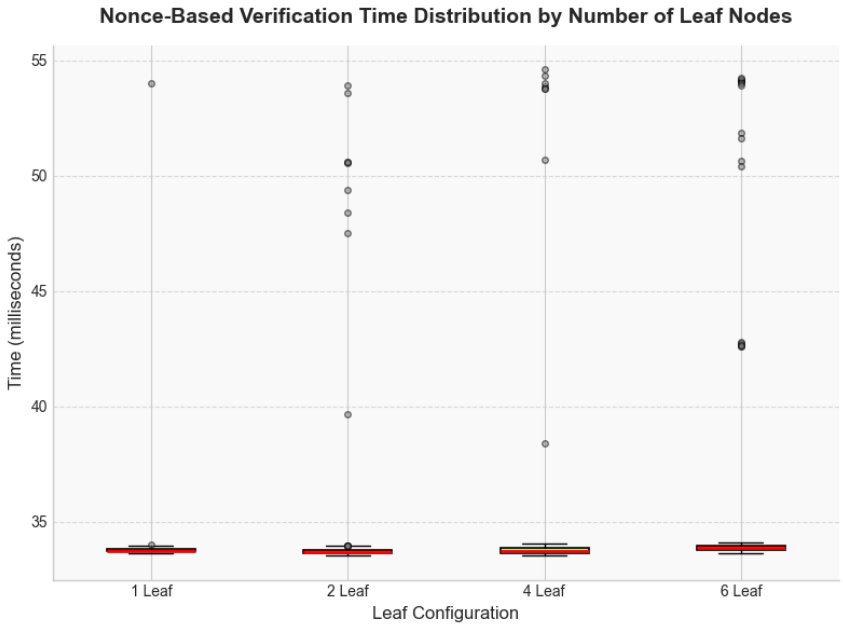


Figure 5.17: Box and Whiskers Plot for Verification Time (ms) at E_V Across 100 Rounds for 1, 2, 4, and 6 Leaf Nodes.

5.2.1.5 Total Per-Round Memory Results

Lastly, this section provides an overview of memory consumption and usage collected at the E_V.

Figure 5.18 displays the averages for in-use allocated and reserved system memory required by the E_V. The results were very even across all node counts, with memory allocation in the 3.29 to 3.87 MB range, and reserved system memory in the 13.22 to 13.87 MB range.

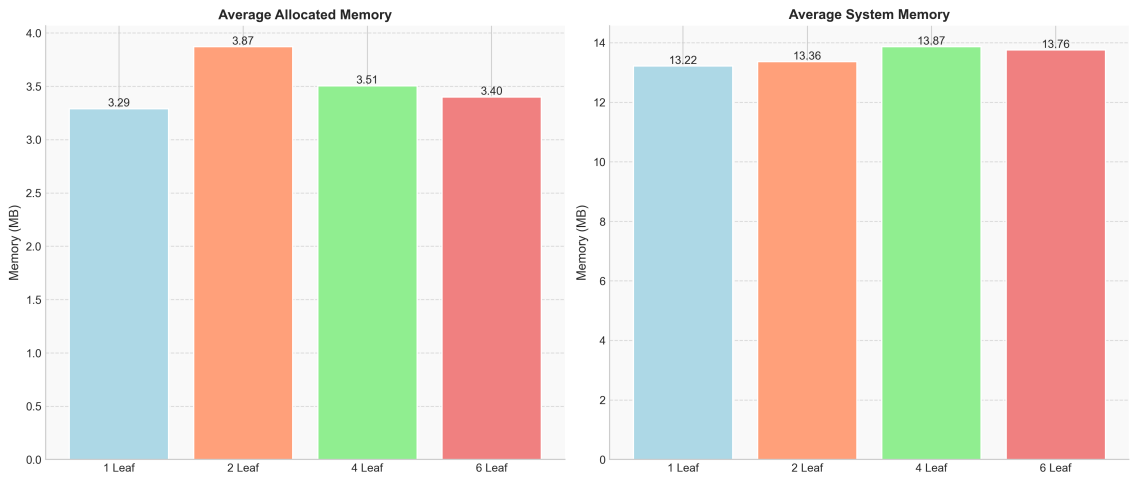


Figure 5.18: Average Allocated and System Memory Utilization (MB) During Attestation and Verification at E_V for 1, 2, 4, and 6 Leaf Nodes.

In regards to the allocated memory used by the E_V, a particular thing to note is that both system and allocated memory usage did rise to far higher amounts during the 200 iterations, as seen in Figure 5.19, and Figure 5.20, respectively. However, as explained previously, this is due to Golang's memory management.

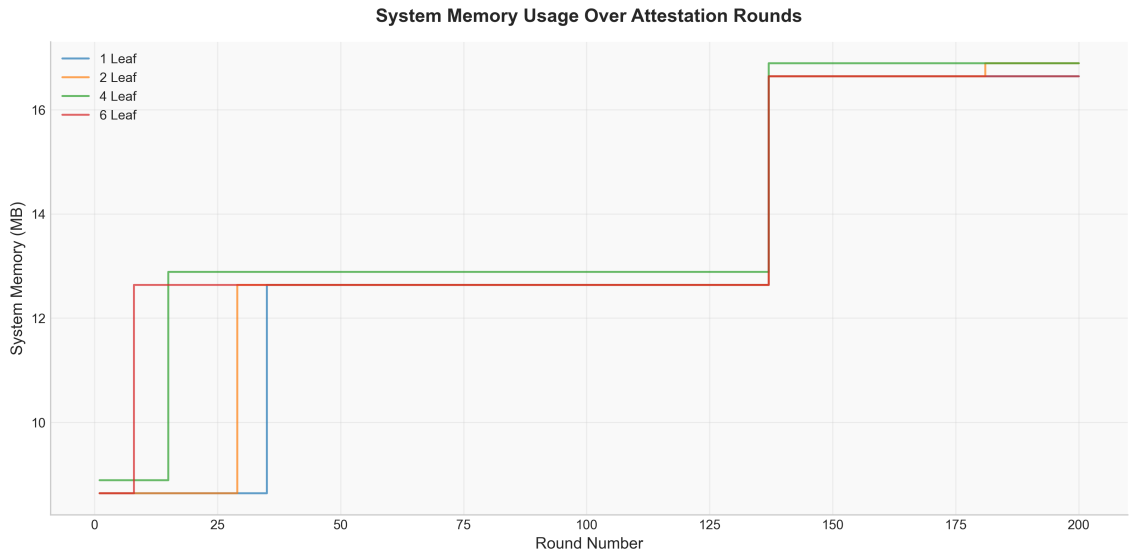


Figure 5.19: System Memory (MB) Utilized During Attestation and Verification at E_V for 1, 2, 4, and 6 Leaf Nodes.

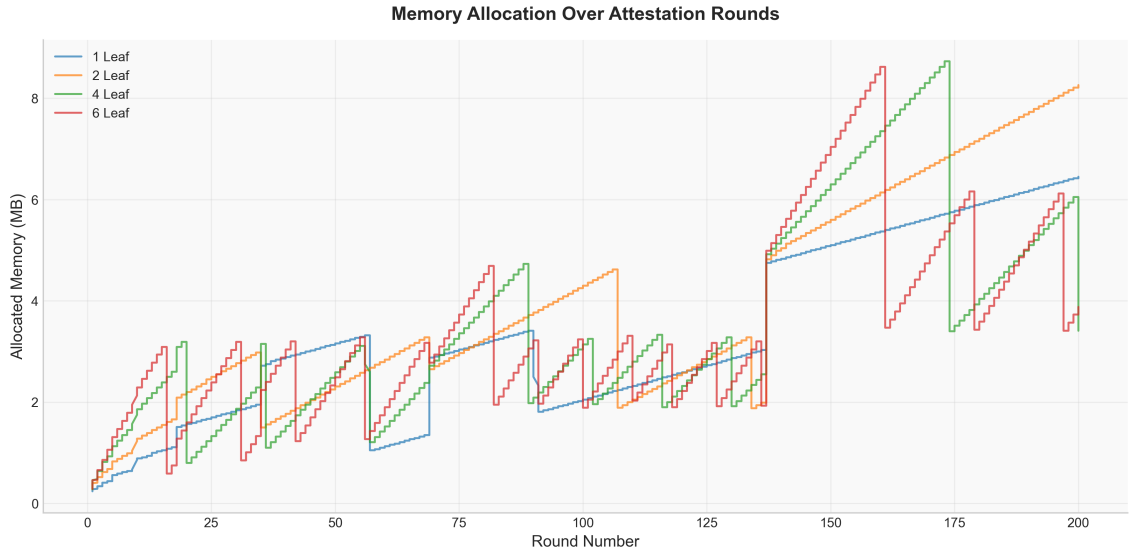


Figure 5.20: Allocated Memory (MB) Utilized During Attestation and Verification at E_V for 1, 2, 4, and 6 Leaf Nodes.

5.3 Self-Attestation Simulation Results

This section presents the simulation results that analyze the cryptographic processing cost and scalability of the attestation protocol under varying network sizes. The evaluation was conducted across multiple seed sets to account for variability and ensure statistical reliability. The following subsections elaborate on these findings in detail.

5.3.1 End-to-End Cryptographic Attestation Latency

Figure 5.21 presents the end-to-end cryptographic attestation latency across attestation rounds for varying network sizes of 3000, 4000, and 5000 nodes. Each curve shows the mean latency computed over multiple simulation runs, while the error bars indicate the standard deviation, capturing variability across ten repetitions.

This metric defines the time required to complete one full attestation round, measured from the moment an E_V receives the first attestation message from any leaf node to the point at which the R_V finalizes the attestation result. This timing window captures the cryptographic costs at the E_V and R_V. Note: Although individual nodes may initiate self-attestation asynchronously, this metric captures only the cryptographic verification and aggregation cost incurred at the E_V and R_V and excludes local self-attestation at the L_{ID}, which is consistent with the key performance indicators used in prior attestation work.

One important observation from the figure is that, across all network size, the end-to-end latency increases during the initial rounds and stabilizes after approximately the fifth round.

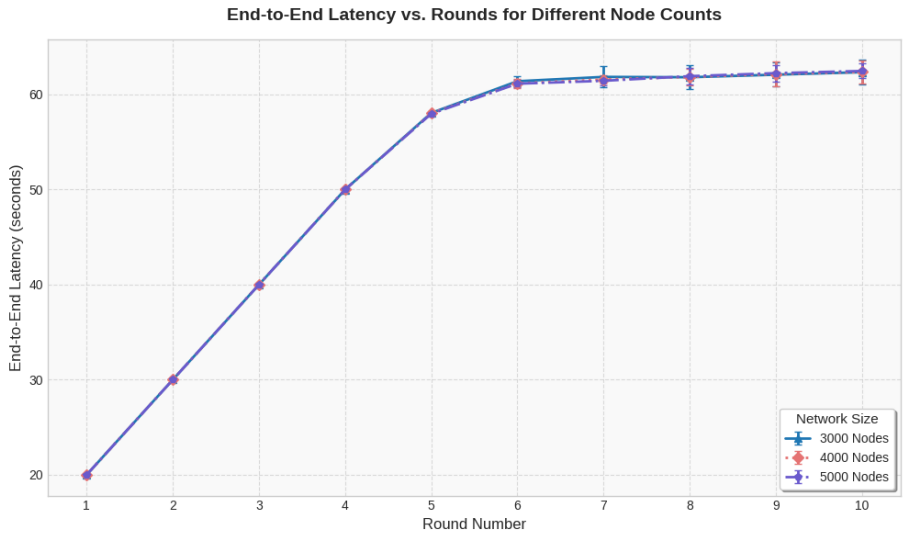


Figure 5.21: End-to-End Cryptographic Attestation Latency

Leaf Count	Min (s)	Median (s)	Max (s)	Mean (s)	Std. Dev (s)
3000	19.99	59.44	65.57	50.75	0.56
4000	19.99	59.41	64.39	50.71	0.44
5000	19.99	59.38	64.05	50.72	0.34

Table 5.3: Statistical Summary Across 10 Rounds

Table 5.3 presents the corresponding statistical summary across 10 runs for different network sizes. Mean, median, min, and max are computed over all rounds, while the standard deviation reflects steady-state variation across runs

5.3.2 Throughput

Figure 5.22 shows the cryptographic attestation throughput as a function of network size for 3000, 4000 and 5000 nodes. Throughput is measured after the system reaches steady state, where end-to-end cryptographic latency has stabilized.

Throughput is computed as:

$$\text{Throughput} = \text{Number of Nodes} \div \text{Maximum Latency (s)}$$

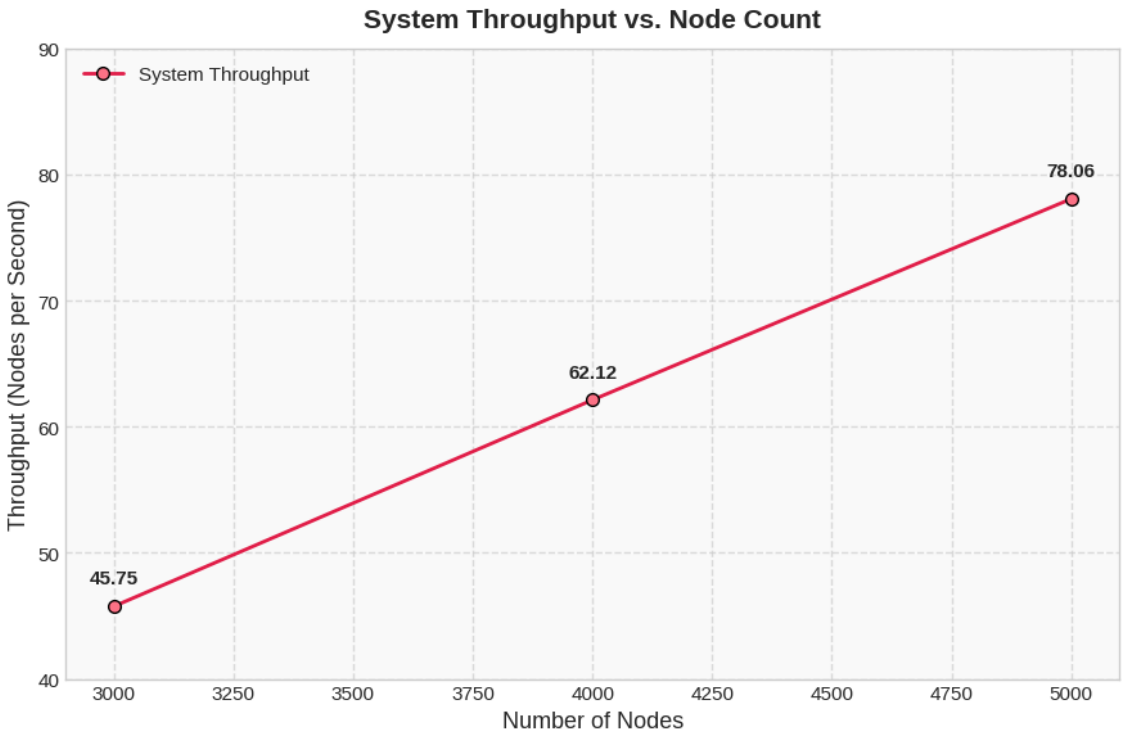


Figure 5.22: Throughput Analysis across 5000 nodes

5.3.3 Layer-wise Cryptographic Processing Cost

Layer	Per-node Cryptographic Cost (ms)	Scaling Behavior	Cost of Scaling
Leaf (L_{ID})	0.70	Constant (parallel across L_{ID})	0.70
Edge Verifier (E_V)	0.73	Linear in L_{ID} (serialized per E_V, parallel across E_V)	$0.73 + 7.46 \times L_{ID}$
Root Verifier (R_V)	7.49	Linear in E_V	$7.49 \times E_V$

Table 5.4: Layer-wise Cryptographic Processing Cost Under Steady-State Operation

Table 5.4 summarizes the steady-state cryptographic processing cost measured at each hierarchy layer. The table reports the measured per-node cost and the corresponding scaling behavior.

6

Security Analysis

In this section, we analyze how the proposed protocol FLASH withstands the attacks defined for our adversary model in section 2.5 and how it achieves the security outlined in subsection 3.1.6. As mentioned in subsection 4.3.2, the root verifiers are assumed to be trusted, edge verifiers are assumed to be trusted or untrusted based on the hierarchy level, and only the swarm nodes are untrusted.

6.1 Security Against Remote Adversaries

A remote adversary may compromise the software state of the node, thereby altering its attestation hash. In FLASH, an edge verifier compares the node's reported MuHash with a trusted reference bootstrap value for verification. Any mismatch directly reveals tampering. At the network level, the root verifier validates the aggregated hashes from all edges and if inconsistencies arise, it requests full reports to identify the compromised nodes.

6.2 Security Against Local Adversaries

A local adversary may eavesdrop or manipulate communication between nodes and edge verifiers. While such interception cannot be completely prevented at the protocol level, FLASH ensures both integrity and authenticity through keyed authentication using HMAC or through the implementation of authenticated encryption. Each message is bound to a challenge or uptime value, thereby preventing the reuse of previously valid responses. Consequently, any attempt to modify, forge, or replay messages is detected during verification, ensuring that even if local traffic is intercepted, it cannot be altered.

6.3 Security Against Replay Attacks

Replay attacks involve reusing previously valid attestation responses. In all attestation modes, this is overcome by temporal uniqueness and cryptographic binding. In the on-demand algorithms, each attestation employs a freshly generated challenge that is embedded into the node response via an HMAC or authenticated encryption, ensuring that responses cannot be reused. In the self-attestation scheme, every

report includes the monotonic uptime bound to a strictly increasing boot counter which the edge verifier cross-checks against its expected window. So, any message replayed from a previous boot session is immediately rejected, providing protection against replay attacks across both variants.

6.4 Security Against Hardware Attacks

Physical attackers can bypass software protections by manipulating or using side-channel attacks. Our protocol does not address such threats. We assume a trusted execution environment and secure key storage, focusing instead on software and communication level compromises.

6.5 Security Against Malicious Edge Verifiers

In FLASH, edge verifiers are assumed to possess sufficient computational resources and secure key storage to perform attestation related operations. However, they are not inherently trusted with respect to correctness, as they may themselves be compromised. Edge verifiers are responsible for validating node reports, aggregating the results, and forwarding them to the root verifier. So before accepting any lower level reports, the root verifier must first attest the corresponding edge verifier. Only after the edge verifier is successfully validated, the root accepts its aggregated report which then verifies the overall network. By doing so, FLASH protects against malicious edge verifiers.

We now discuss how FLASH achieves the security goals mentioned in subsection 3.1.6;

- **Successful Attestation:** This goal is achieved by validating aggregated attestation results at the root/edge. Each edge verifier collects the reports from its assigned nodes, verifies them against the trusted boot data, and aggregates them into a MuHash accumulator. At runtime, the root/edge combines the MuHash values received from all edges into a cluster hash and compares it against the expected cluster hash precomputed during bootstrapping. A match confirms system integrity, while a mismatch triggers a deeper inspection.
- **Freshness:** Freshness is guaranteed through complementary mechanisms in both attestation modes. In the on-demand variant, each attestation begins with the edge verifier generating a new, unique challenge derived from its current uptime. This challenge is cryptographically bound to the node's response using an HMAC, ensuring that any replayed report containing an old challenge is immediately rejected. In contrast, the self-attestation variant guarantees freshness through time consistency, each node reports its monotonic uptime T_{up} bound to the current boot counter $bootCtr$, while the edge verifier verifies that the received T_{up} falls within an expected drift window Δ, relative to its last recorded value. Together, these mechanisms provide end-to-end assurance that every accepted attestation reflects a recent and valid one preventing

replays.

- **Localization of Faults:** As described in Section 3.6.5, FLASH supports multiple levels of granularity. During normal operation, only cluster-level aggregates are sent. If a mismatch occurs, the root triggers a full-report fallback, collecting detailed records from all edge verifiers. This allows the root to localize integrity failures down to specific nodes and their associated edge verifier.
- **Aggregation and Low Communication Overhead:** Aggregation is implemented at the edge verifier level using MuHash. Instead of forwarding every node's attestation to the root, each edge combines verified reports into a single aggregated value. This compact representation significantly reduces communication costs, since the root only receives cluster-level reports during normal operation. Full node-level reports are requested only in the event of mismatches, ensuring that the protocol remains lightweight in typical cases while retaining the ability to perform detailed checks when needed.

7

Discussion

The following chapter presents an analysis and discussion of the results, as well as provides a comparison to other attestation algorithms - therein allowing for a holistic understanding of the effects of homomorphic hashing on swarm attestation. First, the PoC (on-demand algorithm) results are discussed, followed then by the simulation (self-attestation algorithm) results.

7.1 On-demand PoC Results

To understand the implication of the PoC results provided in section 5.1 and section 5.2, and therein create an understanding of FLASH's overall performance, the results must be put into context.

This will be done in two separate steps. Firstly, the PoC results will analyzed separately. To this effect, owing to the data collection procedure outlined in subsection 4.2.4, the analysis will look at the averages and medians of both combined attestation/verification times, and times normalized by number of nodes.

Subsequently, the results of the data analysis will be used to create estimates for performance at higher node and hierarchy cluster level counts than implemented in the PoCs implemented.

7.1.1 Encryption-Based PoC Result Comparison

The following section provides a comparison between the timing data of different node counts for encryption-based attestation and verification, with a summary of the overall data provided in Table 7.1. Upon examining the data, two things stand out in particular.

Leaves	1	2	4	6
Create Challenge				
Average	9	19	45	51
Node-Normalized	9	10	11	9
Median	9	18	38	41
Node-Normalized	9	9	9.5	6.8
Increase(%)	—	0	5.5	-24.1
Handle Response				
Average	42	85	172	258
Node-Normalized	42	43	43	43
Median	40	80	161	243
Node-Normalized	40	40	40.3	40.5
Increase(%)	—	0	0.63	1.25
Verification Time				
Average	36 075	35 495	35 699	35 637
Median	33 736	33 760	33 850	33 781

Table 7.1: Summary and Comparison of Time Averages and Medians in µs for Encryption-Based Attestation & Verification

Firstly, and as expected based on the behavior of the verification algorithm, the average and median verification times across all node counts are consistent, with fluctuations of less than 1 ms across all node counts.
Secondly, by looking at the node-normalized averages and medians, it's clear that the performance of FLASH (when algorithm execution is performed sequentially for each leaf) is linear, with each leaf adding an equal amount of time to the attestation phase.

However, as the actual operation of FLASH presumes parallel computation, taking an average of the node-normalized medians (and in the case of challenge-handling just the average of edge leaf computation time), we can therefore estimate times for a cluster to be:

- 8.58 µs for challenge creation,
- 40.23 µs for response handling,
- 2835 µs for challenge-handling at the leaf (as laid out in subsubsection 5.1.1.2)

Additionally a constant verification time of 33781.75 µs is required, regardless of the number of child nodes added, obtained by averaging the median of the verification times across all node counts.

7.1.2 Nonce-Based PoC Result Comparison

With a preliminary analysis of the encryption-based PoC results performed, the same can be done for the nonce-based PoC, with the summary likewise provided in Table 7.2.

Leaves	1	2	4	6
Create Challenge				
Average	8	21	52	44
Node-Normalized	8	10.5	13	7.3
Median	8	18	39	36
Node-Normalized	8	9	9.8	6
Increase(%)	—	12.5	21.9	-25
Handle Response				
Average	51	100	197	309
Node-Normalized	51	50	49.3	51.5
Median	50	94	184	280
Node-Normalized	50	47	46	46.7
Increase(%)	—	-6	-8	-6.7
Verification Time				
Average	36 271	34 954	34 745	35 936
Median	33 777	33 680	33 709	33 837

Table 7.2: Summary and Comparison of Time Averages and Medians in µs for Nonce-Based Attestation & Verification

Overall, the data exhibits in many ways the same trends as the encryption-based approach. Namely, the nonce-based approach has consistent verification times both between node counts, and as compared to encryption-based verification. This is an expected outcome as the code and algorithm are the same in all cases. In regards to the challenge creation and handling times, these likewise show fairly consistent values when looking at the node-normalized medians/averages.

As with encryption-based attestation, the average of the node-normalized medians can be estimated to:

- 8.20 µs for challenge creation,
- 47.43 µs for response handling,
- 2562 µs for challenge-handling at the leaf

The constant average of median verification times was instead 33750.75 µs.

7.1.3 Generalized On-Demand Performance Estimates

By using the previously concluded node-normalized timing averages for each of the on-demand variants' attestation and verification step(s), further extrapolation of performance is possible for networks of several clusters, one edge and several child leafs, at one or more network hierarchy levels.

Note that other attestation algorithms either exclude network delays completely (for example SHeLA [26]), approximate/simulate network delays either using time (see SEDA [25] which uses 20ms as the average end-to-end delay estimation) or based on calculated message frame sizes (e.g,: PADS [32] and SANA [8]). For the purposes of this analysis, we presume use the same 20ms per-direction time as SEDA.

7.1.3.1 Single Edge-Multiple Leaf Network

At the smallest possible configuration, that being one edge capable of executing p threads simultaneously and n leafs, the time required for one attestation and verification cycle is a sum of how long it takes for:

- The edge to create a challenge for each leaf,
- the leaf(s) to handle the challenge,
- the edge to handle all the responses (perform attestation),
- the edge to verify the performed attestation,
- sending and receiving challenge/response messages.

In this scenario, this would be performed under the following rules:

- The E_V does challenge creation and response handling with a parallelism of p,
- Networking and challenge handling at the leaves is performed in parallel,
- Attestation verification is performed only once per edge regardless of the number of leaves.

This can be formalized using the formula below, where m is the number of leaves in the cluster, and therefore $m = n$:

$$t_{cluster} = \left\lceil \frac{m}{p} \right\rceil \cdot (\text{CreateChal} + \text{HandleResponse}) + \text{HandleChal} + \text{Verify} + (\text{NetworkDelay} \cdot 2)$$

Consequently, we can establish formulas for both the best-case and worst-case performance based on the level of parallelism. In the best-case, with an infinite number of processors, $p = m$ and thus the formula becomes:

$$t_{cluster} = CreateChal + \text{HandleResponse} + \text{HandleChal} + \text{Verify} + (\text{NetworkDelay} \cdot 2)$$

Conversely, the worst case scenario (with fully sequential operations and thus $p = 1$) can be found using the formula:

$$t_{cluster} = m \cdot (\text{CreateChal} + \text{HandleResponse}) + \text{HandleChal} + \text{Verify} + (\text{NetworkDelay} \cdot 2)$$

Thus, the formulas for encryption and nonce-based attestation and verification with one cluster are therefore, respectively:

$$t_{encCluster} = \left\lceil \frac{m}{p} \right\rceil \cdot (8.58 + 40.23) + 2835 + 33781.75 + 40000 = (\left\lceil \frac{m}{p} \right\rceil \cdot 48.81) + 76616.75\mu s$$

$$t_{nonceCluster} = \left\lceil \frac{m}{p} \right\rceil \cdot (8.20 + 47.43) + 2562 + 33750.75 + 40000 = (\left\lceil \frac{m}{p} \right\rceil \cdot 55.63) + 76312.75\mu s$$

7.1.3.2 Multiple Edge-Multiple Leaf Network

In a more realistic implementation of FLASH, attestation would be implemented using many intermediate edges. These edges would be distributed across many levels in a network tree hierarchy. Additionally we presume $p = 1$ so as to demonstrate the capabilities on FLASH even when using single-threaded processors.

In such a scenario, its important to note that each edge would also need to operate as a leaf: receiving a challenge from a parent, computing it's own MuHash attestation value, adding it to the total round hash obtained from its child leafs, and sending the hash to its parent.

Therefore, the time required to attest a network can be attained by first calculating the number of hierarchy levels in a tree with n total devices and m leaves per edge using:

$$\text{Hierarchy levels } (H_L) = \lceil \log_m(n) \rceil$$

H_L can subsequently be applied to the prior defined cluster formulas using:

$$\text{Round Time} = H_L \cdot t_{Cluster}$$

By applying these formulas, we can therein estimate performance for networks of any size and configuration. Figure 7.1 presents extrapolated results for both nonce-based and encryption-based FLASH when using 2, 4, 8, 12, and 16-ary tree formations.

As made obvious by the cluster formulas provided earlier, networking and verification are the two primary factors that contribute to overall round times for large swarms. To this end, serialized FLASH offers good performance as the serialized operations are all very low cost. Notably, the algorithms yield sub-second runtimes even for 4-ary tree networks composed of a million devices.

However, performance can be improved even further. More specifically, this can be done by computing the optimal fan-out (m value) for each of the on-demand variants - in effect minimizing the effects of the expensive verification step and network delays. The effects of fan-out on a network of 2^{16} devices can be seen in Figure 7.2.

As seen in the above figure, runtimes for both algorithms drop significantly at specific points, which we will refer as break points, and then increasing slowly until reaching another break point. These break points correspond to the value of m where the tree increases the number of levels (called l) by one. Analyzing the distribution of these "BreakPoints" we have realized that the breakpoint where $H_L = l$ depends only on the number of leaves n and is defined by the formula:

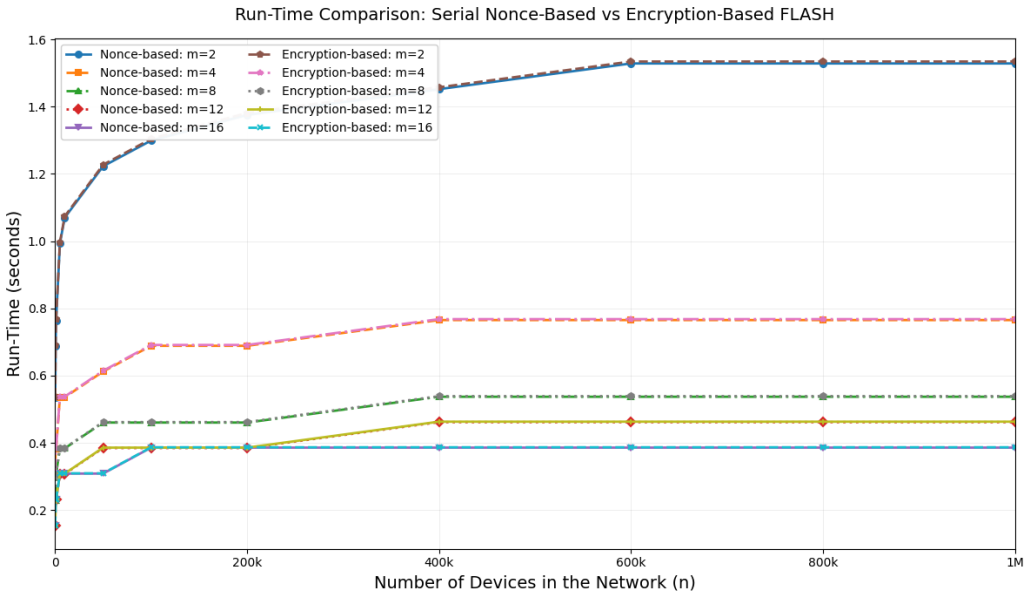


Figure 7.1: Run-Time vs Network Size for Serial On-Demand FLASH Applied to 2, 4, 8, 12, and 16-ary Tree Structures

$$\text{BreakPoint}(l) = \left\lceil \sqrt[l]{n} \right\rceil$$

We can exclude the special case where $H_L = 1$ and thus $m = n$, and know that the maximum, meaningful, value for H_L is obtained when $m = 2$, since $m = 1$ would result in an infinite tree. Thus, to find the optimal value for m we can iterate l between 1 and H_{L_2} (the number of levels when the tree is binary) to find the value of m that minimizes $t_{cluster}$. Hence:

$$m_{opt} = \max_{1 \le l \le H_{L_2}} (t_{cluster}/m = \text{BreakPoint}(l)) = \max_{1 \le l \le H_{L_2}} \left(t_{cluster}/m = \left\lceil \sqrt[l]{n} \right\rceil \right)$$

In this equation, we assume that $\sqrt[1]{n} = n$.

The procedure above can be optimized by noticing that H_L decreases quickly for small values of m, the real implementation we used took that into account this fact by generating candidates for m as follows: start with $m = 2$, then present m as a candidate and increase m by one until $H_{L_m} = H_{L_{m-1}}$. When we reach this point, calculate the rest of the candidates for m given $1 \le l \le H_{L_m} - 1$ as $m = \sqrt[l]{n}$.

Using this procedure for finding the optimal fan-out, times can also be estimated for far larger swarms. To this end, Figure 7.3 presents a graph of optimal m values for networks of up to 2^{20}.

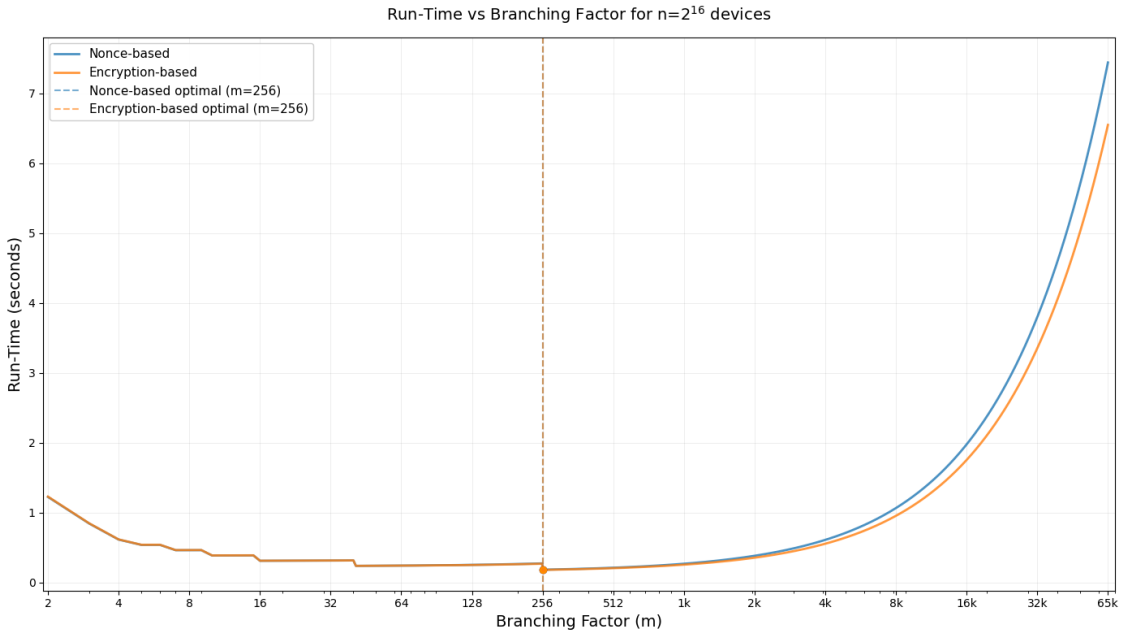


Figure 7.2: Log Scale Run-Time vs Branching Factor for Serial On-Demand FLASH

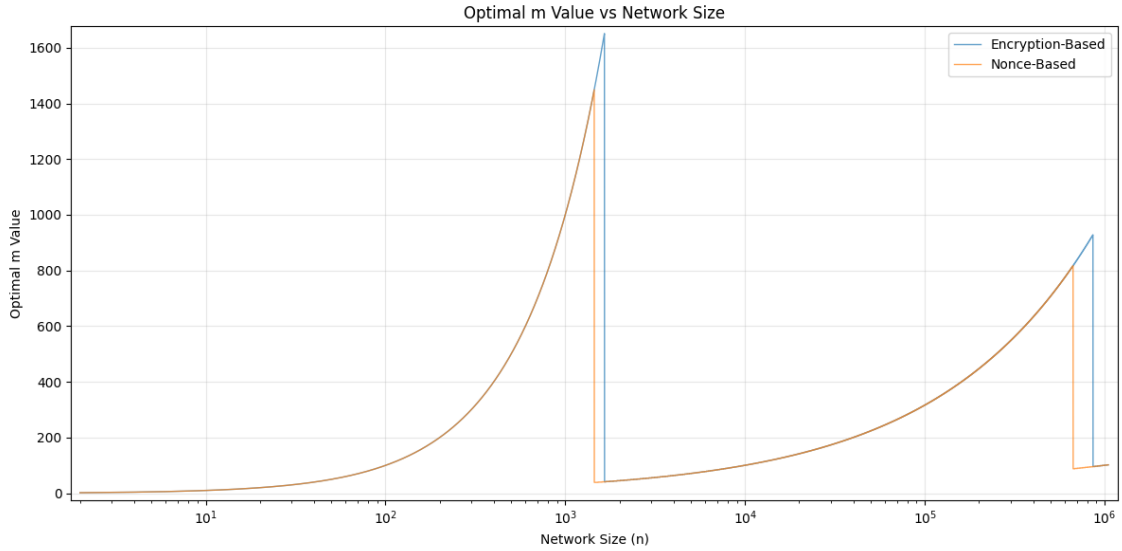


Figure 7.3: Optimal m Value vs Network Size for Serial On-Demand FLASH

To put the performance of serialized FLASH in context, the optimal m values, per-round attestation & verification time, and required hierarchy levels, are calculated for FLASH networks of the sizes presented in Table 7.3.

Scale	Network/Address Space
2^8	IPv4 Class C network
2^{16}	IPv4 Class B network
2^{24}	IPv4 Class A network
2^{32}	Total IPv4 address space
2^{48}	MAC address space
2^{64}	IPv6 network
2^{128}	Total IPv6 address space
10^{100}	A Googol more than atoms in the known universe

Table 7.3: Comparison of Network Address Spaces and Their Relative Scales

Table 7.4 presents estimated times for encryption-based swarms, whilst Table 7.5 presents the data for nonce-based swarms[1].

Number of Devices	Fan-Out	Hierarchy Levels	Time (μs)
2^8	256	1	89112
2^{16}	256	2	178224
2^{24}	256	3	271662
2^{32}	256	4	356448
2^{48}	256	6	534672
2^{64}	256	8	712897
2^{128}	371	15	1420879
10^{100}	317	40	3683581

Table 7.4: Runtime, Optimal Fan-out, and Number of Hierarchy Level for Very Large Encryption-Based FLASH Swarms

Number of Devices	Fan-Out	Hierarchy Levels	Time (μs)
2^8	256	1	90554
2^{16}	256	2	181108
2^{24}	256	3	543324
2^{32}	256	4	362216
2^{48}	256	6	543324
2^{64}	256	8	724432
2^{128}	371	16	1448864
10^{100}	275	41	3756051

Table 7.5: Runtime, Optimal Fan-out, and Number of Hierarchy Level for Very Large Nonce-Based FLASH Swarms

[1]Note that since this data was computed with Python, which has numerical instability, these numbers are an estimate.

7.1.4 Comparison To Other Algorithms

Whilst performance appears to be good, it's important to put it into context as compared to other SA schemes. To this end, a comparison to other schemes is presented in Table 7.6. Despite each of the attestation models presented below having their own benefits, drawbacks, security models, and ways of presenting/measuring data, it is evident that, generally, FLASH offers far greater performance in most cases - especially when networks are composed of very high numbers of devices.

SEDA	Logarithmic growth like serialized FLASH, but has run times of over 2 seconds at high (>100,000 node counts) in a realistic setting.
SANA	Likewise has logarithmic growth, but requires over 9 seconds to attest 200,000 devices or more in the best case scenario (using a 4- or 8-ary tree).
SHeLA	Doesn't present results/estimates for time as a function of swarm size, but presents greater attestation/verification times even for one device.
FESA	Roughly comparable to FLASH estimates, owing to its constant time in static swarms.
LISA-s	Difficult to draw a direct comparison as results are presented for a maximum of 40 devices, but also appears comparable to FLASH if behavior is same at high device counts.
WISE	Computation time of measured in minutes (<6 minutes for 10,000 devices).
SPARK	Computation times are not only greater at very small node counts, but also grow linearly with the number of networked devices.
PRIVÉ	Same issue as SPARK, with linear growth of time as more devices are added to a network.
PADS	Has time that, in the best case, linearly increases for single-thousand digit numbers of devices, time measured in 10s of seconds.

Table 7.6: Performance Characteristics of On-Demand FLASH vs. Competing Swarm Attestation Schemes

7.2 Self-Attestation Simulation Results

Similarly to the PoC evaluation, to understand FLASH performance in the self-attestation-based simulation, we first discuss the self-attestation results presented in section 5.3, followed by a comparative analysis with other swarm attestation algorithms discussed in section 2.4.

7.2.1 Self-Attestation Simulation Results Comparison

This section discusses the simulation results presented in section 5.3.

Firstly, and as expected due to the serialized processing at the E_V and R_V, the measured end-to-end cryptographic attestation latency converges to a stable value across all evaluated network sizes.

The trend observed in the initial attestation rounds arises from the absence of queued cryptographic operations at the beginning of the simulation. During these early rounds, E_V and R_V are idle and can immediately process the attestation workloads. As the simulation progresses, cryptographic operations are serialized and accumulate in a processing queue, introducing waiting time and increasing end to end latency until a steady state is reached.

Secondly, despite the increases in per-round latency during the early rounds, the system exhibits an improvement in throughput as the network size increases. As shown in Table 7.7, the completion time of an attestation round remains nearly constant in steady state. The completion time values correspond to the end-to-end attestation latency per round obtained from earlier latency evaluation as presented in Table 5.3, while the number of nodes processed per unit of time increases significantly. This results in efficiency gains of up to 70.6 percent when scaling from 3000 to 5000 nodes. The key reason is that the hierarchical structure ensures that the critical path depends on the number of E_V instances, rather than the number of leaf nodes. Because the number of E_V remains fixed in these experiments, the completion time remains stable, enabling FLASH to achieve stable steady state latency and scalable attestation throughput under serialized cryptographic execution.

Leaf Count	Completion Time (s)	Throughput (nodes/s)	Efficiency Gain
3000	65.57	45.75	-
4000	64.39	62.12	35.7%
5000	64.05	78.06	70.6%

Table 7.7: System Throughput and Scaling Efficiency Under Steady-State Operation

Thirdly, as shown in Table 5.4, the analysis assumes that each hierarchical layer operates in parallel, while cryptographic operations executed on an individual node are serialized. Under this model, the reported values represent the isolated per-node processing costs at each hierarchy layer.

At the leaf layer, the cryptographic processing cost remains constant, since each leaf device performs only its own self attestation. Because leaf attestations are

independent and execute in parallel across the leaf layer, this cost does not scale with the number of n

At the edge layer, the reported workload corresponds to the per-node post-node-collection processing cost of an individual E_V, which includes only the edge verifier's own self-attestation. Consequently, the edge workload appears constant in the table. However, each edge verifier additionally verifies attestation reports received from multiple L_{ID}. These verification operations are serialized within each edge verifier and therefore scale linearly with the number of L_{ID}, as reflected in the table. Likewise, incremental costs incurred during report reception (such as per-report insertion operations) are excluded from the reported edge and root workloads, as these operations overlap with arrival processing. These insertion costs are negligible compared to the verification cost and are captured in the end-to-end latency.

In contrast, the reported root side workload includes the cost of verifying attestation reports received from each edge verifier, (i.e., attesting the edges). Since this verification is performed once per edge verifier, the total root-side processing cost scales linearly with the number of m.

7.2.2 Generalized Serialized Computational Cost for FLASH Self-Attestation

We model the serialized computational cost per attestation round of the FLASH architecture to multi-level hierarchies, as considered in the PoC evaluation. The analysis assumes that all operations performed by an individual verifier are serialized, while verifiers operating at the same hierarchy level execute in parallel.

$$
\begin{aligned}
t = \; & \underbrace{n \cdot \text{HandleChal}}_{\text{Leaf level}} \\
& + \underbrace{\sum_{i=1}^{H_L-1} \left\lceil \frac{n}{m^i} \right\rceil \left((m \cdot \text{HandleResponse}) + \text{Verify} + \text{HandleChal}\right)}_{\text{Intermediate verifier levels}} \\
& + \underbrace{\left((m \cdot \text{HandleResponse}) + \text{Verify}\right)}_{\text{Root verifier}}
\end{aligned}
$$

In this generalized hierarchy as mentioned in subsubsection 7.1.3.2, n denotes the total number of leaf devices in the system, m denotes the fan-out of the hierarchy (i.e., the number of child nodes handled by each verifier), and H_L denotes the number of hierarchy levels above the leaves, including the root. Consequently, there are $H_L - 1$ intermediate verifier levels between the leaves and the root.

At the leaf level, all n leaf devices perform exactly one handle challenge per round. Since leaf devices operate independently and in parallel, the serialized cost incurred at the leaf level is equal to the cost of a single handle challenge, denoted by HandleChal.

At each intermediate verifier level $i \in \{1, \ldots, H_L - 1\}$, each verifier processes at-

testation responses from exactly m child nodes. For a single verifier, this entails m response handling operations denoted as HandleResponse, one verification operation per round, denoted as Verify, and one handle challenge. The serialized computational cost incurred by a single intermediate verifier is therefore given by

$$m \cdot \text{HandleResponse} + \text{Verify} + \text{HandleChal}.$$

Although multiple verifiers may exist at a given intermediate level, all verifiers at the same level operate in parallel. As a result, the serialized cost contributed by each intermediate level is equal to the cost incurred by a single verifier at that level.

Finally, the root verifier processes attestation reports from its m immediate children and performs a single aggregation operation per round. The root does not perform self-attestation. Accordingly, the serialized computational cost incurred at the root level is

$$m \cdot \text{HandleResponse} + \text{Verify}.$$

7.2.3 Single Root - Multiple edge and Leaf Network

This configuration considers a hierarchical topology consisting of multiple leaf nodes, multiple edge verifiers, and a single root verifier. The time required for one full round, i.e., one attestation and verification cycle, is the sum of the time required for:

- Each leaf to perform handle challenge and forward its attestation report to its assigned edge.
- Each edge to verify the handled the attestation reports and incrementally aggregate/verify the verified reports.
- Each edge to perform its own handle challenge, as it acts as a leaf node with respect to the higher hierarchy level
- The root to verify each edge, that is, its attestation, and then receive the edge-level attestation reports for each leaf and perform the final attestation

In this scenario, the attestation process is performed under the following execution rules:

- Each verifier processes attestation reports from its child nodes using a parallelism of p, allowing up to p reports to be handled concurrently.
- Networking and handle challenge operations at the leaf nodes are performed fully in parallel.
- Response handling at each verifier are performed once per child node and may be parallelized up to p.
- Verification at each verifier is performed once per attestation round, independent of the number of child nodes.

This execution behavior can be formalized using the model described below for $t(p)$, where m denotes the number of child nodes per verifier (with $m = n$ at the leaf level).

We consider a generalized execution model in which each verifier is equipped with p parallel processing units. At a given hierarchy level, a verifier processes attestation reports from m child nodes. Since at most p reports can be processed in parallel, the verifier requires $\lceil m/p \rceil$ serialized processing batches per attestation round.

Consistent with the PoC evaluation, a network latency of 20 ms per communication hop is assumed, as described in SeDA [25]. For a hierarchy consisting of H_L verifier levels above the leaves, an attestation message traverses H_L communication hops along with critical path from a leaf to the root. Since only one message is transmitted upward at any time, the total network delay along with the critical path is given by: $H_L \cdot \text{NetworkDelay}$.

The resulting end-to-end attestation latency is given by:

$$
\begin{aligned}
t(p) = \; & \underbrace{\text{HandleChal}}_{\text{Leaf level}} \\
& + \underbrace{\sum_{i=1}^{H_L-1} \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Intermediate verifier levels}} \\
& + \underbrace{\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify}}_{\text{Root verifier}} \\
& + \underbrace{H_L \cdot \text{NetworkDelay}}_{\text{Network delay}} .
\end{aligned}
$$

To simplify the formula, we can start by grouping together the cost of leaf and root term. Thus we obtain:

$$
\begin{aligned}
t(p) = \; & \underbrace{\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal}}_{\text{Root verifier and Leaf level}} \\
& + \underbrace{\sum_{i=1}^{H_L-1} \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Intermediate verifier levels}} \\
& + \underbrace{H_L \cdot \text{NetworkDelay}}_{\text{Network delay}} .
\end{aligned}
$$

Since the summation does not depend on the index i, the summation is replaced by

multiplication so we obtain instead:

$$t(p) = \underbrace{1 \cdot \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Root verifier and Leaf level}}$$

$$+ \underbrace{(H_L - 1) \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Intermediate verifier levels}}$$

$$+ \underbrace{H_L \cdot \text{NetworkDelay}}_{\text{Network delay}}.$$

Since this equation has the same terms on the right side of two added multiplications we add them together as follows:

$$t(p) = \underbrace{(H_L - 1 + 1) \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Intermediate verifier levels and Root verifier and Leaf level}}$$

$$+ \underbrace{H_L \cdot \text{NetworkDelay}}_{\text{Network delay}}.$$

We can then simplify the left side of the first multiplication which gives us:

$$t(p) = \underbrace{H_L \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} \right)}_{\text{Intermediate verifier levels and Root verifier and Leaf level}}$$

$$+ \underbrace{H_L \cdot \text{NetworkDelay}}_{\text{Network delay}}.$$

And if we group in the right side terms of H_L we end up with the simplified formula:

$$t(p) = H_L \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay} \right)$$

In line with the PoC evaluation, we derive expressions for both best-case and worst-case performance as a function of the available parallelism. The best case corresponds to full parallelism, where a verifier is provisioned with sufficient processing resources to handle all m incoming attestation reports concurrently. This execution model corresponds to $p = m$, yielding $\lceil m/p \rceil = 1$.

$$t_{min} = H_L \left(\text{HandleResponse} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay} \right) \tag{7.1}$$

Substituting the measured values from the Table 7.1.2 into the above expression yields:

$$t_{min} = H_L\,(47.43\ \mu s + 33750.75\ \mu s + 2562\ \mu s + 20000\ \mu s) = H_L(56360.18\ \mu s) \tag{7.2}$$

The worst-case execution model corresponds to fully serialized processing at each verifier, i.e., $p = 1$, for which $\lceil m/p \rceil = m$.

$$t_{max} = H_L\,(m \cdot \text{HandleResponse} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay}) \tag{7.3}$$

Substituting the measured values from the Table 7.1.2 into the above expression yields:

$$\begin{aligned} t_{max} &= H_L\,(m \cdot (47.43\ \mu s + 33750.75\ \mu s + 2562\ \mu s + 20000\ \mu s) \\ &= H_L\,(m \cdot 47.43\ \mu s + 56312.75\ \mu s) \end{aligned} \tag{7.4}$$

Although, the parallel and serialized execution models above capture the best-case and worst-case attestation latencies, real deployments typically operate between these two extremes. Attestation requests arrive periodically, and each verifier processes a steady stream of reports over time. To capture this more realistic operating scenario, we model the leaf, edge, and root as periodic real-time tasks. The system operates in periodic attestation rounds with an end-to-end deadline of T = 1 s. Within each round, every leaf node generates exactly one attestation report, which is forwarded to its assigned edge verifier. Consequently, each edge verifier receives m leaf attestation reports per period T, while the root verifier receives m edge-level attestation reports per period. The m reports may arrive at different times and should be modeled as schedulable tasks.

For a period real-time task with execution cost C and period T, a standard schedulability condition as given by [37] is: $U = \frac{C}{T}$ This condition requires that the processor utilization remains strictly below one. To ensure conservativeness and guarantee schedulability under worst-case conditions, we enforce a conservative utilization bound of $U_{\max} = \frac{2}{3}$ as recommended by [37]. While utilization bounds provide general scheduability intuition, in our hierarchical approach, feasibility is determined by the serialized critical-path cost within the end-to-end deadline.

In each attestation period, every leaf node performs exactly one handle challenge and forwards its attestation report to its assigned edge verifier. The edge verifier processes the received report from its m child nodes, performs verification and handle challenge, and forwards a single verification report to the root. Finally, the root verifier processes the received reports from its m child verifiers and performs the final verification. Since these stages lie on the critical path, their execution must be completed sequentially within the same attestation period. Under this execution

model, the end-to-end real-time constraint is directly given by the serialized cost along the critical path.

$$C = t(p) = H_L \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay} \right)$$

Consequently, we can now calculate the minimum period T of our system as:

$$T \leq \frac{C}{U} = \frac{H_L \left(\left\lceil \frac{m}{p} \right\rceil \text{HandleResponse} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay} \right)}{U}$$

We need to consider now that while we have control on when the Verify and HandleChal executions will be executed and that the NetworkDelays will not depend on our processing speed (but still subtract from the time we have available in a period). Nevertheless, the $\left\lceil \frac{m}{p} \right\rceil$ attestation requests with a cost of HandleResponse will arrive at different times and will need to be scheduled correctly. Therefore, we need to consider the system utilization U at these points to ensure we can meet the scheduling deadlines. Hence the equation becomes instead:

$$T \leq H_L \left(\left\lceil \frac{m}{p} \right\rceil \frac{\text{HandleResponse}}{U} + \text{Verify} + \text{HandleChal} + \text{NetworkDelay} \right)$$

Replacing the terms with the values we used before and with our $U_{\max}$ we get the following equation:

$$\begin{aligned} T &\leq H_L \left(\left\lceil \frac{m}{p} \right\rceil \frac{47.43\ \mu s}{\frac{2}{3}} + 33750.75\ \mu s + 2562\ \mu s + 20000\ \mu s \right) \\ &= H_L \left(\left\lceil \frac{m}{p} \right\rceil \cdot 71.145\ \mu s + 56312.75\ \mu s \right) \end{aligned}$$

These terms still depend on the values of H_L, m and p But allow us to estimate the maximal acceptable size of a network with fan-out value m. For example, with uniprocessor edges $p = 1$ and using a quaternary tree $m = 4$, we could calculate the maximum size of n as follows:

$$T \leq H_L \left(4 \cdot 71.145\ \mu s + 56312.75\ \mu s \right)$$

$$T \leq H_L \left(284.58\ \mu s + 56312.75\ \mu s \right)$$

$$T = 1000000 \leq \lceil \log_4(n) \rceil \cdot 56597.33\ \mu s$$

$$\lceil \log_4(n) \rceil \leq \frac{1000000}{56597.33}$$

$$4^{\lceil \log_4(n) \rceil} \leq 4^{\frac{1000000}{56597.33}}$$

$$4^{\log_4(n)} \leq 4^{\lfloor \frac{1000000}{56597.33} \rfloor}$$

$$n \leq 4^{\lfloor \frac{1000000}{56597.33} \rfloor} = 4^{17} = 17179869184$$

Thus, under steady-state real time operation with a 1 s end-to-end deadline, a quaternary tree network can sustain up to $4^{17} = 17179869184$ child nodes.

8

Conclusion

This thesis presented FLASH, a lightweight and scalable protocol that efficiently incorporates homomorphic hashing into swarm attestation. The performance of FLASH was evaluated through both a PoC implementation on Arduino and Raspberry Pi platforms and a simulation-based analysis using OMNeT++. In addition, a comprehensive adversary model was defined, and the security of FLASH was analyzed to demonstrate its resilience against adversaries.

Overall, the results show that swarm attestation, when implemented using homomorphic hashing, can lead to very fast attestation, even in large scale networks. However, there are naturally a number of ways in which this would could be expanded in the future.

8.1 Future Work

In regards to the core FLASH algorithms themselves, one area of future work is developing the algorithms to explicitly allow for dynamic swarm topologies. Whilst the core algorithms allow this to be added without significant changes, due to the fact that individual leaf node attestation reports are stored on parent edges, this is an area of great interest. In particular, creating an understanding of the performance effects of leaf nodes joining/existing swarms would be of great interest to real world implementations.

Another area of future work is the implementation of mechanisms for detecting physical attacks, and providing a secure way for edge nodes to exchange/replicate child attestation, allowing for more resilient swarms which could withstand the loss of one or more edge devices.

Beyond expanding the FLASH algorithms, there are likewise a number different aspects related to testing FLASH which could be performed in the future to allow for a greater understanding of the effects of homomorphic hashing on swarm attestation.

Notably, one such extension would be conducting large-scale PoC deployments, with multiple edge-leaf clusters reporting to a central root. This would allow for validation of the performance models, and strengthen the validity of the current findings.

Furthermore, it would likewise be interesting to explore the impact of different network hierarchy topologies on attestation performance and scalability through extended simulation studies. In particular, this would allow for the establishment of lower bound, worst-case, performance estimates.

Lastly, we aim to prepare and submit FLASH for an academic publication.

Bibliography

[1] Svensk Kollektivtrafik, *Kollektivtrafikbarometern årsrapport 2024*, 2024. [Online]. Available: https://svenskkollektivtrafik.se/app/uploads/2025/02/Arsrapport-Kollektivtrafikbarometern-2024.pdf.

[2] S. A. Shaheen and R. Finson, "Intelligent transportation systems," in *Encyclopedia of Energy*, C. J. Cleveland, Ed., New York: Elsevier, 2004, pp. 487–496, ISBN: 978-0-12-176480-7. DOI: https://doi.org/10.1016/B0-12-176480-X/00191-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B012176480X001911.

[3] E. Fok, "An introduction to cybersecurity issues in modern transportation systems," *ITE Journal*, vol. 3, no. 19, pp. 19–24, 2013. DOI: N/A. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fc389b5b8c6d4e760a614ad3300b390e277234da.

[4] "Poland investigates cyber-attack on rail network," *British Broadcasting Corporation*, Aug. 26, 2023. [Online]. Available: https://www.bbc.com/news/world-europe-66630260.

[5] T. Edwards, "TfL cyber attack: What you need to know," *British Broadcasting Corporation*, Sep. 28, 2023. [Online]. Available: https://www.bbc.com/news/articles/ceqn7xng7lpo.

[6] C. Xu, H. Liu, P. Li, and P. Wang, "A remote attestation security model based on privacy-preserving blockchain for V2X," *IEEE Access*, vol. 6, pp. 67 809–67 818, 2018. DOI: 10.1109/ACCESS.2018.2878995.

[7] D. Oladimeji, K. Gupta, N. A. Kose, K. Gundogan, L. Ge, and F. Liang, "Smart transportation: An overview of technologies and applications," *Sensors*, vol. 23, no. 8, 2023, ISSN: 1424-8220. DOI: 10.3390/s23083880. [Online]. Available: https://www.mdpi.com/1424-8220/23/8/3880.

[8] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "SANA: Secure and scalable aggregate network attestation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 731–742, ISBN: 9781450341394. DOI: 10.1145/2976749.2978335. [Online]. Available: https://doi.org/10.1145/2976749.2978335.

[9] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Toward secure and efficient attestation for highly dynamic swarms: Poster," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '17, Boston, Massachusetts: Association for Computing Machinery, 2017, pp. 281–282, ISBN: 9781450350846. DOI:

10.1145/3098243.3106026. [Online]. Available: https://doi.org/10.1145/3098243.3106026.

[10] E. Dushku and N. Dragoni, "Swarm attestation," in *Encyclopedia of Cryptography, Security and Privacy*, S. Jajodia, P. Samarati, and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019, pp. 1–5, ISBN: 978-3-642-27739-9. DOI: 10.1007/978-3-642-27739-9_1783-1. [Online]. Available: https://doi.org/10.1007/978-3-642-27739-9_1783-1.

[11] B. Buchanan, *Homomorphic hashes - efficient trust and avoiding complex hashing operations*, https://medium.com/asecuritysite-when-bob-met-alice/homomophic-hashes-efficient-trust-and-avoiding-complex-hashing-operations-1b288a17f7b1, Accessed: 2025-05-19, 2023.

[12] M. Bellare and D. Micciancio, "A new paradigm for collision-free hashing: Incrementality at reduced cost," in *Advances in Cryptology - EUROCRYPT '97*, W. Fumy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 163–192, ISBN: 978-3-540-69053-5. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-69053-0_13.

[13] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '94, Berlin, Heidelberg: Springer-Verlag, 1994, pp. 216–233, ISBN: 3540583335. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-48658-5_22.

[14] G. Coker, J. Guttman, P. Loscocco, *et al.*, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011. DOI: 10.1007/s10207-011-0124-7. [Online]. Available: https://doi.org/10.1007/s10207-011-0124-7.

[15] B. Kuang, A. Fu, W. Susilo, S. Yu, and Y. Gao, "A survey of remote attestation in internet of things: Attacks, countermeasures, and prospects," *Computers & Security*, vol. 112, p. 102 498, 2022, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2021.102498. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821003229.

[16] M. Ammar, B. Crispo, I. De Oliveira Nunes, and G. Tsudik, "Delegated attestation: Scalable remote attestation of commodity CPS by blending proofs of execution with software attestation," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '21, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2021, pp. 37–47, ISBN: 9781450383493. DOI: 10.1145/3448300.3467818. [Online]. Available: https://doi.org/10.1145/3448300.3467818.

[17] A. S. Banks, M. Kisiel, and P. Korsholm, *Remote attestation: A literature review*, 2021. arXiv: 2105.02466. [Online]. Available: https://arxiv.org/abs/2105.02466.

[18] I. Sfyrakis and T. Gross, *A survey on hardware approaches for remote attestation in network infrastructures*, 2020. arXiv: 2005.12453 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2005.12453.

[19] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: Hybrid design for remote attestation (using a formally verified microkernel)," in *Proceedings*

of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks, ser. WiSec '17, Boston, Massachusetts: Association for Computing Machinery, 2017, pp. 99–110, ISBN: 9781450350846. DOI: 10.1145/3098243.3098261. [Online]. Available: https://doi.org/10.1145/3098243.3098261.

[20] M. N. Aman, M. H. Basheer, S. Dash, *et al.*, "HAtt: Hybrid remote attestation for the internet of things with high availability," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7220–7233, 2020. DOI: 10.1109/JIOT.2020.2983655.

[21] B. Kuang, A. Fu, S. Yu, G. Yang, M. Su, and Y. Zhang, "ESDRA: An efficient and secure distributed remote attestation scheme for IoT swarms," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8372–8383, 2019. DOI: 10.1109/JIOT.2019.2917223.

[22] J. Ménétrey, C. Göttel, M. Pasin, P. Felber, and V. Schiavoni, *An exploratory study of attestation mechanisms for trusted execution environments*, 2022. arXiv: 2204.06790 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2204.06790.

[23] Y. Swami, *SGX remote attestation is not sufficient*, Cryptology ePrint Archive, Paper 2017/736, 2017. [Online]. Available: https://eprint.iacr.org/2017/736.

[24] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure until proven updated: Analyzing AMD SEV's remote attestation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 1087–1099, ISBN: 9781450367479. DOI: 10.1145/3319535.3354216. [Online]. Available: https://doi.org/10.1145/3319535.3354216.

[25] N. Asokan, F. Brasser, A. Ibrahim, *et al.*, "SEDA: Scalable embedded device attestation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 964–975.

[26] M. M. Rabbani, J. Vliegen, J. Winderickx, M. Conti, and N. Mentens, "SHeLA: Scalable heterogeneous layered attestation," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 240–10 250, 2019. DOI: 10.1109/JIOT.2019.2936988.

[27] B. Kuang, A. Fu, Y. Gao, Y. Zhang, J. Zhou, and R. H. Deng, "FeSA: Automatic federated swarm attestation on dynamic large-scale iot devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 4, pp. 2954–2969, 2023. DOI: 10.1109/TDSC.2022.3193106.

[28] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik, "Lightweight swarm attestation: A tale of two LISA-s," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, pp. 86–100, ISBN: 9781450349444. DOI: 10.1145/3052973.3053010. [Online]. Available: https://doi.org/10.1145/3052973.3053010.

[29] M. Ammar, M. Washha, and B. Crispo, "WISE: Lightweight intelligent swarm attestation scheme for iot (the verifiers perspective)," in *2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2018, pp. 1–8. DOI: 10.1109/WiMOB.2018.8589107.

[30] W. Hellemans, N. El Kassem, M. Masoom Rabbani, *et al.*, "SPARK: Secure privacy-preserving anonymous swarm attestation for in-vehicle networks," in

2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P), 2025, pp. 903–922. DOI: 10.1109/EuroSP63326.2025.00056.

[31] N. {El Kassem}, W. Hellemans, I. Siachos, *et al.*, "Privé: Towards privacy-preserving swarm attestation," English, in *Proceedings of the 22nd International Conference on Security and Cryptography*, 22nd International Conference on Security and Cryptography, SECRYPT 2025, Science and Technology Publications, Lda, Jun. 2025, pp. 247–262, ISBN: 978-989-758-760-3. DOI: 10.5220/0013629000003979. [Online]. Available: https://secrypt.scitevents.org/?y=2025.

[32] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "PADS: Practical attestation for highly dynamic swarm topologies," in *2018 International Workshop on Secure Internet of Things (SIoT)*, 2018, pp. 18–27. DOI: 10.1109/SIoT.2018.00009.

[33] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Collective remote attestation at the Internet of things scale: State-of-the-art and future challenges," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 4, pp. 2447–2461, 2020. DOI: 10.1109/COMST.2020.3008879.

[34] E. Dushku, M. M. Rabbani, J. Vliegen, A. Braeken, and N. Mentens, "PROVE: Provable remote attestation for public verifiability," *Journal of Information Security and Applications*, vol. 75, p. 103 448, 2023. DOI: 10.1016/j.jisa.2023.103448. [Online]. Available: https://doi.org/10.1016/j.jisa.2023.103448.

[35] K. Lewi, W. Kim, I. Maykov, and S. Weis, "Securing update propagation with homomorphic hashing," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 227, 2019. [Online]. Available: https://eprint.iacr.org/2019/227.

[36] F. De Santis, A. Schauer, and G. Sigl, "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 692–697. DOI: 10.23919/DATE.2017.7927078.

[37] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973, ISSN: 0004-5411. DOI: 10.1145/321738.321743. [Online]. Available: https://doi.org/10.1145/321738.321743.