LIghtweight Swarm Attestation: a Tale of Two LISA-s

Xavier Carpent
Computer Science Department
University of California, Irvine
xcarpent@uci.edu

Karim ElDefrawy*
Computer Science Lab
SRI International
karim@csl.sri.com

Norrathep Rattanavipanon
Computer Science Department
University of California, Irvine
nrattana@uci.edu

Gene Tsudik
Computer Science Department
University of California, Irvine
gene.tsudik@uci.edu

ABSTRACT

In the last decade, Remote Attestation (RA) emerged as a distinct security service for detecting attacks on embedded devices, cyber-physical systems (CPS) and Internet of Things (IoT) devices. RA involves verification of current internal state of an untrusted remote hardware platform (prover) by a trusted entity (verifier). RA can help the latter establish a static or dynamic root of trust in the prover and can also be used to construct other security services, such as software updates and secure deletion. Various RA techniques with different assumptions, security features and complexities, have been proposed for the single-prover scenario. However, the advent of IoT brought about the paradigm of many interconnected devices, thus triggering the need for efficient collective attestation of a (possibly mobile) group or swarm of provers. Though recent work has yielded some initial concepts for swarm attestation, several key issues remain unaddressed, and practical realizations have not been explored.

This paper's main goal is to advance swarm attestation by bringing it closer to reality. To this end, it makes two contributions: (1) a new metric, called QoSA: Quality of Swarm Attestation, that captures the information offered by a swarm attestation technique; this allows comparing efficacy of multiple protocols, and (2) two practical attestation protocols – called ***LISA***α and ***LISA***s – for mobile swarms, with different QoSA features and communication and computation complexities. Security of proposed protocols is analyzed and their performance is assessed based on experiments with prototype implementations.

1. INTRODUCTION

The number of so-called Internet of Things (IoT) devices is expected to soon [8] exceed that of traditional computing devices, i.e., PCs, laptops, tablets and smartphones. IoT can be loosely defined as a set of interconnected embedded devices, each with a various blend of sensing, actuating and computing capabilities. In many IoT settings and use-cases, devices operate collectively as part of a group or swarm, in order to efficiently exchange information and/or collaborate on common tasks. Examples of IoT swarms include multitudes of interconnected devices in smart environments, such as a *smart* households, factories, and buildings. Actual devices might include home theater sound systems, home camera and surveillance systems, electrical outlets, light fixtures, sprinklers, smoke/CO_2 detectors, faucets, appliances, assembly-line components as well as drones. Device swarms also appear in agriculture, e.g., livestock monitoring [16], as well as other research areas, e.g., swarm robotics and swarm intelligence [19]. As IoT swarms become increasingly realistic, their security and overall well-being becomes both apparent and important. Specifically, it is necessary to periodically (or on demand) ensure collective integrity of software running on swarm devices.

The formidable impact of large-scale remote malware infestations has been initially demonstrated by the Stuxnet incident in 2011 and the most recent Dyn denial-of-service (DoS) attack in 2016. This attack type aims to compromise as many devices as possible, without physical access, or close proximity, to any victim device. Compromise of "smart" household devices may also have significant privacy ramifications. In one recent incident, cameras in compromised smart TVs were used to record private activities of their owners [26]. It is not hard to imagine other such attacks, e.g., malware that performs physical DoS by activating smart door locks, sprinklers, or light-bulbs.

1.1 Remote Attestation

In the last decade, Remote Attestation (RA) emerged as a distinct security service with the main goal of detecting malware presence on embedded systems and IoT devices. Essentially, RA is the process where a trusted entity (verifier or $\mathcal{V}$rf) securely verifies current internal state of an untrusted and possibly compromised remote device (prover or $\mathcal{P}rv$). RA can help establish a static or dynamic root of trust in the prover. It can also be used as a building block for constructing more specialized security services, such as software updates as well as secure deletion and device resetting.

RA techniques generally fall into three categories: *hardware*, *software* and *hybrid*. The first relies on secure hardware, e.g., a Trusted Platform Module (TPM) [24, 20], often present in relatively powerful devices such as desktops, laptops and smartphones. It is impractical for medium- and low-end IoT devices due to costs

*Work conducted while at HRL Laboratories.

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates

DOI: http://dx.doi.org/10.1145/3052973.3053010

and complexity. Software RA techniques [23, 21] provide security guarantees based on strong assumptions about adversarial behavior. They are generally applicable only to legacy devices that lack hardware security features, and to settings where the prover is physically close to the verifier such that round-trip delay is fixed and/or negligible. Finally, hybrid techniques (e.g., SMART [9] and TrustLite [13]) are based on hardware/software co-design; they require the prover to have a minimal set of hardware features and thus target cost-sensitive medium- and low-end devices. Hybrid techniques include a software component implementing the attestation protocol as well as hardware components that ensure certain guarantees, e.g., non-interruptibility, memory isolation and exclusive access to secret keys.

A more detailed overview of related work can be found in Appendix A.

1.2 Swarm Attestation

Both feasibility and efficacy of hybrid RA approaches[1] have been demonstrated in the single-prover scenario. Nonetheless, new issues emerge when it is necessary to attest a potentially large number (group or swarm) of devices. First, it is inefficient and sometimes impractical to naïvely apply single-prover RA techniques to each device in a large swarm that might cover a large geographical area. Second, swarm RA needs to take into account topology discovery, key management and routing. This can be further complicated by mobility (i.e., dynamic topology) and device heterogeneity, in terms of computing and communication resources.

A recently proposed scheme, Scalable Embedded Device Attestation (SEDA) [4], represents the first step towards practical swarm RA. It builds upon aforementioned hybrid SMART and TrustLite techniques. It combines them with a flooding-like protocol that propagates attestation requests and gathers corresponding replies. According to simulations in [4], SEDA performs significantly better than individually attesting each device in a swarm. Despite its viability as a paper design, SEDA is not a practical technique, for several reasons. First, it is under-specified in terms of:

- **Architectural Impact:** What is the impact of swarm RA on the underlying hardware and security architecture (which suffices for single-prover settings), in terms of: (a) additional required features, as well as (b) increased size and complexity of current features?
- **Timing:** How to determine overall attestation timeout for the verifier? This issue is not as trivial as it might seem, as we discuss later in the paper.
- **Initiator Selection:** How to select the device(s) that start(s) the attestation process in order to construct a spanning tree over the swarm topology?

Second, as we discuss later, SEDA has some gratuitous (unnecessary) features, such as the use of public key cryptography, which are unjustified by the assumed attack model. Third, it is unclear whether SEDA handles device (node) mobility. This is an important issue: some swarm settings are static in nature, while others involve node mobility and dynamic topologies.

Finally, SEDA does not capture or specify the exact quality of the overall attestation outcome and thus provides no means to compare its security guarantees to other swarm RA techniques. We believe that it is important to define a qualitative (and whenever possible, quantitative) measure for swarm RA, i.e., *Quality of Swarm Attestation* (QoSA). This measure should reflect verifier's information requirements and allow comparisons across swarm RA techniques.

[1] In the RA context, we use the following terms interchangeably throughout the paper: protocols, techniques, methods and approaches.

1.3 Contributions

In order to bring swarm attestation closer to practice, issues discussed above need to be addressed. To this end, after defining the notion of QoSA, we design and evaluate two practical swarm RA protocols (***LISA***α and ***LISA***s, with different QoSA-s) that narrow the gap between paper-design techniques such as SEDA and realistic performance assessment and practical deployment. We also carefully investigate their impact on the underlying security architecture. Performance of proposed protocols is assessed using the open-source Common Open Research Emulator (CORE) [2].

1.4 Outline

Section 2 describes assumptions, preliminaries and the network model. Section 3 presents the design of the two LIghtweight Swarm Attestation (***LISA***) protocols. Section 4 contains the security analysis of our protocols. Section 5 presents implementation details and performance assessment, while Section 6 discusses additional cryptographic considerations. Section 7 concludes the paper and discusses future work.

2. PRELIMINARIES

We now delineate this paper's scope and outline our assumptions.

2.1 Scope

This paper focuses on swarm RA in the presence of *limited* device mobility, which means that swarm topology is assumed to be connected and quasi-static during each RA session. The latter means that the swarm connectivity graph can change as long as changes do not influence message propagation during an attestation session.

Similar to prior results in the single-prover RA setting, proposed protocols are not resistant to physical attacks. Other than imposing ubiquitous tamper-resistant hardware, the only practical means of mitigating physical attacks is by heartbeat-based absence detection [10]. We consider this to be an orthogonal direction and focus on remote malware attacks. Also, low-level denial-of-service (DoS) attacks that focus on preventing communication (e.g., physical-layer jamming) are beyond the scope of this paper. However, we do take into account DoS attacks that try to force devices to perform a lot of computation and/or communication.

Since we build upon state-of-the-art hybrid techniques for single-prover RA, our protocols assume that each device adheres to the minimal requirements specified in [9] and [7]. Our protocols attest only static or predictable segments of memory, e.g., code and data. It is very difficult, perhaps even infeasible, to perform an integrity check over dynamic memory segments due to the combinatorial explosion of possible outcomes.

Even though practicality, i.e., suitability for real-world deployment, is the ultimate goal of this work, we do not actually deploy proposed techniques in real-world swarm settings. Nevertheless, we achieve the next best thing by implementing and evaluating them via emulation, which effectively replicates the behaviors of physical, link and network layers (by virtualizing them on top of Linux) in a virtual environment which takes into account wireless channel interference, noise and loss. Emulation allows us to easily experiment with multiple deployment configurations (varying number of devices, their wireless capabilities and environments) and swarm topologies – something not easily doable in an actual deployed swarm. We claim that, though not the same as actual deployment, emulation is much more realistic than simulation. Since the latter completely abstracts away the protocol stack, it can miss some practical performance issues and artifacts that arise in using

the actual stack, the medium access protocol (at the data link layer) and characteristics of the wireless channel (at the physical layer).

2.2 Network & Device Assumptions

Devices: We assume that each swarm device (prover):

- Adheres to ***SMART+*** architecture, as discussed in Section 2.3 below.
- Has at least one network interface and ability to send/receive both unicast and broadcast packets.
- The second protocol (***LISA***s) in Section 3.2, requires each device to have a clock in order to implement a timer and to know the total number of devices in the swarm – n.

In general, devices can vary along three dimensions: (1) attestation architecture, (2) computational power, and (3) installed code-base. As mentioned above, we assume uniform adherence to ***SMART+*** architecture. Our first protocol, ***LISA***α, makes no other assumptions. The second, ***LISA***s, also assumes homogeneity in terms of computational power.

Connectivity & Topology: The verifier ($\mathcal{V}$rf) is assumed to be unaware of the current swarm topology. The topology (connectivity graph) of the swarm can change arbitrarily between any two attestation instances. It might change for a number of reasons, e.g., physical movement of devices, foreign objects impeding or enabling connections between devices, hibernating devices, or devices entering or leaving the network. However, during each attestation instance, the swarm is assumed to be: (1) connected, i.e., there is a path between any pair of devices, and (2) quasi-static. The latter means that the swarm connectivity graph **can** actually change during an attestation session, as long as changes do not influence message propagation, e.g., if a link disappears after one device finishes sending a message, and re-appears before any other message is exchanged between the same pair of devices. See Sections 3.1.3 and 3.2.3 for details.

If either condition does not hold, protocols discussed in Section 3 still provide best-effort attestation, i.e., if a change of connectivity occurs, some healthy devices might end up not being attested, which would result in a false-negative outcome. Nonetheless, infected devices are never positively attested, regardless of any connectivity changes during attestation.

Adversary Type: Based on the recently proposed taxonomy [1], adversaries in the context of RA can be categorized as follows:

- *Remote:* exploits vulnerabilities in prover's software to remotely inject malware. In particular, it can modify any existing code, introduce new malicious code, or read any unprotected memory location.
- *Local:* located sufficiently near the prover to eavesdrop on, and manipulate, prover's communication channels.
- *Physical Non-Intrusive:* located physically near the prover; can perform side-channel attacks in order to learn secrets.
- *Stealthy Physical Intrusive:* can physically compromise a prover; however, it leaves no trace, i.e., can read any (even protected) memory and extract secrets.
- *Physical Intrusive:* has full (local) physical access to the prover; can learn or modify any state of hardware or software components.

Our protocols take into account remote and local adversary flavors. However, as with most prior work, all types of physical attacks are out of scope.

Since all hybrid RA schemes involve hardware-protected key storage, we assume (for now) a trivial master key approach, whereby all swarm devices have the same secret, shared with $\mathcal{V}$rf. While this might seem silly, it is sufficient in a setting without physical attacks; see Section 3 for more details. However, for the sake of completeness, counter-measures to physical attacks and additional cryptographic considerations are discussed in Section 6.

2.3 Security Architecture

Architectural minimality is a key goal of this work; hence, our protocols require minimal hardware support. Specifically, we assume that each device adheres to the SMART architecture [9], augmented with $\mathcal{V}$rf authentication (aka DoS mitigation) features identified in [7]. We refer to this combination as ***SMART+*** and its key aspects are:

- All attestation code (***AttCode***) resides in ROM. ***AttCode*** is safe, i.e., it always terminates and leaks no information beyond the attestation result, i.e., a token.
- Execution of ***AttCode*** is atomic and complete, which means: (a) it can not be interrupted, and (b) it starts and ends at official entry and exit points, respectively.[2] This feature is generally enforced by a Memory Protection Unit (MPU) using a set of static rules.
- At least one secret key stored in ROM, which can only be read from within ***AttCode***. For now, we remain agnostic as far as what type of cryptography is being used.
- A fixed-size block of secure RAM that stores the counter and/or a timestamp of the last executed attestation instance. (This is needed to prevent replay attacks). This memory can only be modified from within ***AttCode***. [7] offers an alternative in the form of a reliable real-time clock that can not be modified by non-physical means. However, we opt for a secure counter since it is a cheaper feature.

SMART+ operates as follows:

1. $\mathcal{V}$rf generates an authenticated attestation request. Authentication is achieved either via a signature or a message authentication code (MAC), depending on the type of cryptography used.
2. On $\mathcal{P}rv$, the attestation request is received by untrusted code outside ***AttCode*** and passed on to ***AttCode***.
3. ***AttCode*** disables interrupts and checks whether the sequence number of the request is greater than current counter value. If not, request is ignored.
4. ***AttCode*** authenticates – using either symmetric or public key – the attestation request. If authentication fails, request is ignored.
5. ***AttCode*** computes the authenticated integrity check of its memory (i.e., the result), stores it in a publicly accessible location, cleans up after itself, enables interrupts and terminates.
6. Untrusted code on $\mathcal{P}rv$ (outside of ***AttCode***) returns the result to $\mathcal{V}$rf.
7. $\mathcal{V}$rf authenticates the result and decides whether $\mathcal{P}rv$ is in a secure state.

Memory organization and memory access rules for ***SMART+*** and ***LISA*** are summarized in Figure 1.

2.4 Quality of Swarm Attestation (QoSA)

The main goal of swarm RA is to verify collective integrity of the swarm, i.e., all devices at once. However, in some settings, e.g., when a swarm covers a large physical area, the granularity of a simple binary outcome is not enough. Instead, it might be more useful to learn which devices are potentially infected, so that quick action can be taken to fix them. By the same token, it could be also useful to learn the topology. To this end, we introduce a notion that tries to capture the information provided by swarm RA, called *Quality of Swarm Attestation (QoSA)*. It also enables comparing

[2] There might be multiple legal exit points.

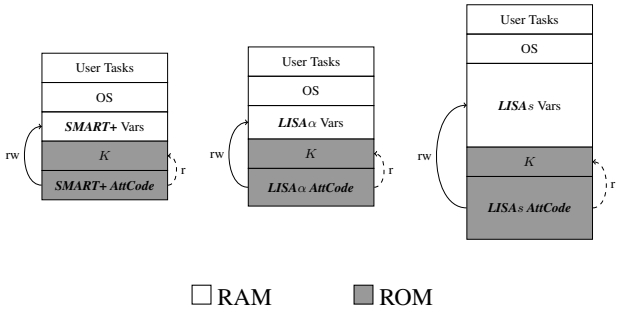


Figure 1: Memory organization and access rules in ***SMART+*** and LISA; r denotes read, w denotes write.

multiple swarm attestation protocols. We consider the following types of QoSA:

- *Binary QoSA (B-QoSA):* a single bit indicating success or failure of attestation of the entire swarm.
- *List QoSA (L-QoSA):* a list of identifiers (e.g., link-layer and/or network-layer addresses) of devices that have successfully attested.
- *Intermediate QoSA (I-QoSA):* information that falls between B-QoSA and L-QoSA, e.g., a count of successfully attested devices.
- *Full QoSA (F-QoSA):* a list of attested devices along with their connectivity, i.e., swarm topology.

This is not an exhaustive list. Although we view these four types as fairly natural, other QoSA-s can be envisioned. We also note that, in a single-prover setting which applies to most prior attestation literature, QoSA is irrelevant, since $\mathcal{V}$rf communicates directly with one $\mathcal{P}rv$, and there is no additional information beyond the attestation result itself. In contrast, in a multi-prover setting, QoSA is both natural and useful. It can be tailored to the specific application's needs, as described below in Section 3.

2.5 Attestation Timeouts

Since envisaged swarm attestation is mostly autonomous and $\mathcal{V}$rf is initially unaware of the current topology, there needs to be an overall timeout value. As in any one-to-many reliable delivery protocol, timeouts are necessary to account for possible losses of connectivity during attestation, caused by mobility, noisy channels, or excessive collisions, all of which might occur naturally, or be caused by DoS attacks. As usual, the timeout parameter must be selected carefully, since an overly low value would result in frequent false positives, while an overly high one would cause unnecessary delays. In any case, we assume that the timeout is dependent on n – the number of devices in the swarm.

2.6 Initiator Selection

To minimize its burden, $\mathcal{V}$rf can initiate the process by directly sending an attestation request to one device in the swarm. We call this device an "initiator". There are several ways to select it, e.g., based on physical proximity, and/or computation power. If $\mathcal{V}$rf has no knowledge about nearby devices, it first needs to perform neighbor discovery (e.g., [12] or [14]) which introduces an extra step in the overall process. Alternatively, $\mathcal{V}$rf can use multiple initiators and skip neighbor discovery by simply broadcasting an attestation request to whichever device(s) can hear it. In that case, all $\mathcal{V}$rf's immediate neighbors become initiators, in parallel. Our protocols are agnostic to this choice and work regardless of how initiators are selected, as long as at least one is picked.

2.7 Verifier Assumptions

Following prior work, we assume an honest $\mathcal{V}$rf. In particular, it is not compromised and is trusted to correctly generate all attestation requests, as well as to correctly process all received attestation reports (replies). Also, $\mathcal{V}$rf is assumed to know n.

3. NEW SWARM RA PROTOCOLS

We now describe two lightweight swarm RA protocols, ***LISA***α and ***LISA***s, including their design rationale, details and complexities. Similar to ***SMART+***, either symmetric or public key cryptography can be used to provide authenticated integrity of protocol messages. However, for the sake of simplicity and efficiency, we describe ***LISA***α and ***LISA***s assuming a single swarm-wide symmetric master key. This master key can be pre-installed into all swarm devices at manufacture or deployment time. Although this might seem naïve, recall that, in the absence of physical attacks, there is no difference between having: (1) one swarm-wide master key shared with $\mathcal{V}$rf, (2) a symmetric unique key each device shares with $\mathcal{V}$rf, or (3) a device unique public/private key-pair for each device. This is because malware that infects any number of devices still can not access a device's secret key due to ***SMART+***'s MPU access rules. However, if physical attacks are considered, Section 6 discusses the use of device-specific symmetric keys and public key cryptography.

3.1 Asynchronous Version: *LISA*α

LISAα stands for: **Li**ghtweight **S**warm **A**ttestation, **a**synchronous version. Its goal is to provide efficient swarm RA while incurring minimal changes over ***SMART+***. Before describing ***LISA***α, we can imagine a very intuitive approach, whereby $\mathcal{V}$rf, relying strictly on ***SMART+***, runs an individual attestation protocol directly with each swarm device. This would require no extra support in terms of software or hardware features. Nonetheless, this naïve approach does not scale, since it requires $\mathcal{V}$rf to either: (1) attest each device in sequence, which can be very time-consuming, or (2) broadcast to all devices and maintain state for each, while waiting for replies. This scalability issue motivates device collaboration for propagating attestation requests and reports. ***LISA***α adopts this approach and involves very low computational overhead, while being resistant to computational denial-of-service (DoS) attacks. Devices act independently and asynchronously, relying on each other only for forwarding attestation requests and reports.

3.1.1 *LISA*α Protocol Details

LISAα's pseudo-code and finite state machine (FSM) for a prover device ($\mathcal{D}ev$) are illustrated in Algorithm 1 and an upper figure of Figure 2, respectively. ***LISA***α's FSM for $\mathcal{V}$rf is illustrated in a lower figure of Figure 2 and the pseudo-code is described in Algorithm 2. The protocol involves two message types:
(1) *request*: $\mathcal{A}tt_{req} = [\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{A}uth_{req}]$ and
(2) *report*: $\mathcal{A}tt_{rep} = [\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{P}ar, \mathcal{S}eq, H(\mathcal{M}em), \mathcal{A}uth_{rep}]$
where:

- $\mathcal{S}nd$ – identifier of the sending device; this field is not authenticated
- $\mathcal{S}eq$ – sequence number and/or timestamp of the present attestation instance; set by $\mathcal{V}$rf
- $\mathcal{A}uth_{req}$ – authentication token for the attestation request message; computed by $\mathcal{V}$rf as: $\mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$
- $\mathcal{D}evID$ – identifier of $\mathcal{D}ev$; stored in ROM, along with ***AttCode***
- $\mathcal{P}ar$ – identifier of the reporting device's parent in the spanning tree; copied from $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$
- $\mathcal{A}uth_{rep}$ – authentication token for the attestation reply message; computed by $\mathcal{D}ev$ as:

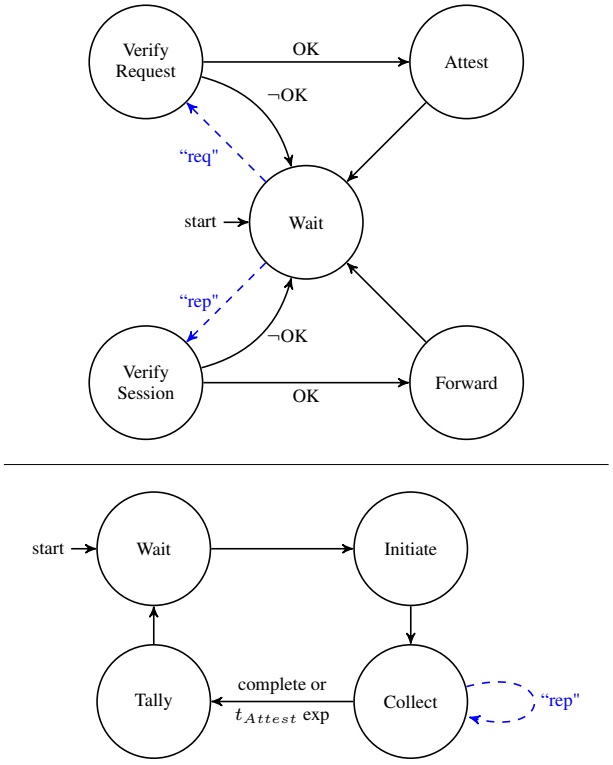


Figure 2: ***LISA***α FSM-s for $\mathcal{D}ev$ (top) and $\mathcal{V}$rf (bottom)

MAC$(K, \text{“}rep\text{”}||\mathcal{D}evID||\mathcal{S}eq||H(\mathcal{M}em))$, where $H()$ is a suitable cryptographic hash function and $\mathcal{M}em$ denotes device memory that is subject to attestation[3].

LISAα *Prover.*

From the perspective of a prover $\mathcal{D}ev$, ***LISA***α has five states:

1. Wait: $\mathcal{D}ev$ waits for an attestation-relevant packet. In case of $\mathcal{A}tt_{req}$, $\mathcal{D}ev$ proceeds to VerifyRequest and in case of $\mathcal{A}tt_{rep}$, it jumps to VerifySession.

2. VerifyRequest: $\mathcal{D}ev$ first checks validity of $\mathcal{S}eq$, which must be strictly greater than the previous stored value; otherwise, it discards $\mathcal{A}tt_{req}$ and returns to Wait. Next, $\mathcal{D}ev$ validates $\mathcal{A}uth_{req}$ by recomputing MAC. If verification fails, $\mathcal{D}ev$ discards $\mathcal{A}tt_{req}$ and returns to Wait. Otherwise, $\mathcal{D}ev$ saves $\mathcal{S}eq$ as the current session number $\mathcal{C}urSeq$, stores $\mathcal{S}nd$ as its parent device $\mathcal{P}ar$ for this session, and transitions to Attest.

3. Attest: $\mathcal{D}ev$ sets $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$ to $\mathcal{D}evID$ and broadcasts the modified $\mathcal{A}tt_{req}$. Next, $\mathcal{D}ev$ computes $\mathcal{A}uth_{rep}$ and composes $\mathcal{A}tt_{rep}$, as defined above. Note that $\mathcal{A}uth_{rep}$ authenticates $\mathcal{P}ar$ by virtue of covering $\mathcal{A}tt_{req}$. Finally, $\mathcal{D}ev$ unicasts $\mathcal{A}tt_{rep}$ to $\mathcal{P}ar$ and transitions to Wait.

4. VerifySession: $\mathcal{D}ev$ receives $\mathcal{A}tt_{rep}$ from one of its descendants. If the $\mathcal{S}eq$ in $\mathcal{A}tt_{rep}$ does not match its stored counterpart $\mathcal{C}urSeq$, $\mathcal{D}ev$ discards $\mathcal{A}tt_{rep}$ and returns to Wait. Otherwise, it proceeds to Forward.

5. Forward: $\mathcal{D}ev$ unicasts $\mathcal{A}tt_{rep}$ received in VerifySession, to its stored $\mathcal{P}ar$ and returns to Wait.

[3]Note that $H(\mathcal{M}em)$ is part of $\mathcal{A}tt_{rep}$. We can omit it to save space, and have $\mathcal{V}$rf keep a mapping of $(\mathcal{D}evID, H(\mathcal{M}em))$. However, this would take away $\mathcal{V}$rf's ability to make decisions based on actual device signatures.

Algorithm 1: Pseudo-code of ***LISA***α for $\mathcal{D}ev$

Write-Protected Vars: $\mathcal{D}evID$ – id of $\mathcal{D}ev$
$\mathcal{C}urSeq$ – current sequence #
$\mathcal{P}ar$ – $\mathcal{D}ev$'s parent id

```
1  while True do
2      m = RECEIVE();
3      if TYPE(m) = "req" then
4          [Snd, Auth_req, Seq] ← DECOMPOSE(m);
5          if Seq < CurSeq then
6              CONTINUE();
7          end
8          if Auth_req ≠ MAC(K, "req"||Seq) then
9              CONTINUE();
10         end
11         CurSeq ← Seq; Par ← Snd;
12         BROADCAST("req"||DevID||CurSeq||Auth_req);
13         Auth_rep ← MAC(K, "rep"||DevID||CurSeq||H(Mem));
14         Att_rep ←
           "rep"||DevID||Par||CurSeq||H(Mem)||Auth_rep;
15         UNICAST(Par, Att_rep);
16     else if TYPE(m) = "Rep" then
17         Seq ← GETSEQ(m);
18         if Seq = CurSeq then
19             UNICAST(Par, m);
20         end
21     end
22 end
```

LISAα *Verifier.*

From $\mathcal{V}$rf's perspective, ***LISA***α is simpler, with four states:

1. Wait: $\mathcal{V}$rf waits for external signal (e.g., from a user) to start a new attestation session. When it arrives, $\mathcal{V}$rf moves to Initiate.

2. Initiate: $\mathcal{V}$rf sets the overall timeout and selects the initiator(s), as discussed earlier. It then initializes $\mathcal{A}ttest = \mathcal{F}ail = \emptyset$, $\mathcal{N}orep = \{all\ \mathcal{D}evID\}$. Next, $\mathcal{V}$rf sets $\mathcal{S}nd$ =$\mathcal{V}$rf, composes $\mathcal{A}tt_{req}$, sends it (via unicast) to the initiator(s), and moves to Collect.

3. Collect: $\mathcal{V}$rf waits for $\mathcal{A}tt_{rep}$ messages from the initiator(s) or an overall timeout. If a timeout occurs, $\mathcal{V}$rf transitions to Tally. Upon receipt of $\mathcal{A}tt_{rep}$, $\mathcal{V}$rf extracts and validates $\mathcal{A}uth_{rep}$ by recomputing MAC. (Note that duplicate $\mathcal{A}tt_{rep}$ messages are assumed to be automatically detected and suppressed). There are three possible outcomes:
   i. Validation fails: $\mathcal{A}tt_{rep}$ is discarded,
   ii. $\mathcal{A}uth_{rep}$ is authentic and $H(\mathcal{M}em)$ corresponds to an expected (legal) state of $\mathcal{D}evID$'s attested memory: $\mathcal{V}$rf adds $\mathcal{D}evID$ to $\mathcal{A}ttest$, and removes it from $\mathcal{N}orep$,
   iii. $\mathcal{A}uth_{rep}$ is authentic and $H(\mathcal{M}em)$ does not match any expected state of $\mathcal{D}evID$'s attested memory: $\mathcal{V}$rf adds $\mathcal{D}evID$ to $\mathcal{F}ail$ and removes it from $\mathcal{N}orep$

If $|\mathcal{A}ttest| + |\mathcal{F}ail| = n$, $\mathcal{V}$rf moves to Tally; otherwise it remains in Collect.

4. Tally: $\mathcal{V}$rf outputs $\mathcal{A}ttest$, $\mathcal{F}ail$ and $\mathcal{N}orep$ as sets of devices that passed, failed and didn't reply, respectively. Finally, $\mathcal{V}$rf returns to Wait.

3.1.2 $\mathcal{V}$rf *Timeout in* ***LISA***α

As follows from the protocol description (or, equivalently, from FSMs and pseudocode), devices do not require a timeout. For its part, $\mathcal{V}$rf sets the overall attestation timeout to $t_{Attest} = t_a + n \cdot t_{MAC} + 2 \cdot n \cdot t_t + t_s$, **where:**

- t_a – time for $\mathcal{D}ev$ to perform self-attestation[4]
- t_{MAC} – time for $\mathcal{D}ev$ to compute a MAC (to verify or generate) over a short message
- t_t – time for $\mathcal{D}ev$ to transmit a message to another device

[4]In case of heterogeneous devices, t_a represents the maximum self-attestation time across all devices. The same applies to t_{MAC} and t_t.

Algorithm 2: Pseudo-code of ***LISA***α for $\mathcal{V}$rf

```
1  t_Attest ← t_a + t_MAC + 2 · n · t_t + t_s;
2  while True do
3      wait();
4      InitID ← getInitId();
5      CurSeq ← getSeq();
6      Att_req ← "req"||Vrf||CurSeq||Auth_req;
7      unicast(InitID, Att_req);
8      Attest ← ∅; Fail ← ∅;
9      Norep ← {allDevID};
10     T ← getTimer();
11     while T < t_Attest do
12         Att_rep ← receive();
13         [DevID, Par, Seq, H(Mem), Auth_rep]
           ← decompose(Att_rep);
14         if Seq = CurSeq ∧ Auth_rep =
           MAC(K, "rep"||DevID||CurSeq||H(Mem)) then
15             if H(Mem) ⊂ expectedHash(DevID) then
16                 Attest ← Attest ∪ {DevID};
17             else
18                 Fail ← Fail ∪ {DevID};
19             end
20             Norep ← Norep \ {DevID};
21         end
22         if |Attest| + |Fail| = n then
23             break();
24         end
25     end
26     output(Attest, Fail, Norep);
27 end
```

- t_s – slack time, which accounts for variabilities, i.e., possible deviations

t_{Attest} represents the time corresponding to running ***LISA***α over a n-device swarm with the worst-case topology scenario, i.e., a realistic upper bound. The worst-case is a **line topology** where $\mathcal{A}tt_{req}$ processing is done in sequence, taking $n \cdot t_{MAC}$. Only one t_a needs to be included in t_{Attest} since the last device (the only leaf in the tree) finishes its attestation after all others. Also, since there are at most n hops between $\mathcal{V}$rf and the last device, it takes $n \cdot t_t$ to transmit $\mathcal{A}tt_{req}$ to that device and the same amount of time to transmit the last $\mathcal{A}tt_{rep}$ to $\mathcal{V}$rf.

3.1.3 Connectivity in *LISA*α

Let t_0 denote the time when $\mathcal{D}ev$ receives $\mathcal{A}uth_{req}$ from $\mathcal{P}ar$, $t_{rep,i}$ denote the time when $\mathcal{P}ar$ receives the i^{th} $\mathcal{A}uth_{rep}$ from $\mathcal{D}ev$ and z denote the number of $\mathcal{D}ev$'s descendants. The connectivity assumption of ***LISA***α can be formally stated as follows:

LISAα produces a correct swarm attestation result, i.e. no false positives and no false negatives, if a link between every $\mathcal{D}ev$ and its $\mathcal{P}ar$ exists during their t_0, $t_{rep,1}$, $t_{rep,2}$,...,$t_{rep,z+1}$.

3.1.4 QoSA of *LISA*α

At the end, $\mathcal{V}$rf collects a set of $\mathcal{A}tt_{rep}$ messages, one from each device. After verifying all $\mathcal{A}tt_{rep}$-s, $\mathcal{V}$rf learns the list of successfully attested devices, thus achieving L-QoSA. It is easy to augment the protocol to collect topology information along with attestation results. This can be performed by simply including $\mathcal{P}ar$ in each $\mathcal{A}tt_{rep}$. $\mathcal{V}$rf then can thus reconstruct the topology based on verified reports. Specifically, line 15 in Algorithm 1 would become:
$\mathcal{A}uth_{rep} \leftarrow \mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{P}ar||H(\mathcal{M}em))$;
However, topology information obtained by $\mathcal{V}$rf is not reliable, since $\mathcal{P}ar$ is not authenticated upon receiving $\mathcal{A}tt_{req}$. Fixing this is not hard; it would require each device to: (1) compute and attach an extra MAC, at least over $\mathcal{P}ar$ and $\mathcal{A}uth_{req}$ fields, at $\mathcal{A}tt_{req}$ forwarding time, and (2) verify the $\mathcal{P}ar$'s MAC upon receiving $\mathcal{A}tt_{req}$.

3.1.5 Complexity of *LISA*α

Complexity is discussed in Appendix B.1. In brief, ***LISA***α is very simple in terms of software complexity and impose no additional features on the underlying attestation architecture. However, it requires a larger ROM and additional static MPU rules. Also, high communication overhead is ***LISA***α's biggest drawback, since $\mathcal{D}ev$ transmits n reports in the worst case. This motivates the design of ***LISA***s, which aims to reduce communication overhead by aggregating multiple $\mathcal{A}tt_{rep}$-s.

3.2 Synchronous Version: *LISA*s

The main idea in ***LISA***s is to let devices authenticate and attest each other. When one device is attested by another, only the identifier of the former needs to be securely forwarded to $\mathcal{V}$rf, instead of the entire $\mathcal{A}tt_{rep}$. This translates into considerable bandwidth savings and lower $\mathcal{V}$rf workload. Also, $\mathcal{A}tt_{rep}$-s can be aggregated, which decreases the number of packets sent and received. It also allows more flexibility in terms of QoSA: from B-QoSA to F-QoSA. Finally, malformed or fake $\mathcal{A}tt_{rep}$-s are detected in the network and not propagated to $\mathcal{V}$rf, as in ***LISA***α. However, these benefits are traded off for increased protocol (and code) complexity, as described below.

LISAs's main distinctive feature is that each $\mathcal{D}ev$ waits for all of its children'$\mathcal{A}tt_{rep}$-s before submitting its own. This makes the protocol synchronous. Each $\mathcal{D}ev$ keeps track of its parent and children during an attestation session. Once $\mathcal{A}tt_{req}$ is processed and propagated, $\mathcal{D}ev$ waits for each child to complete attestation by submitting a $\mathcal{A}tt_{rep}$. Then, $\mathcal{D}ev$ verifies each $\mathcal{A}tt_{rep}$, aggregates a list of children as well as descendants they attested, attests itself, and finally sends its authenticated $\mathcal{A}tt_{rep}$ (which contains the list of attested descendants) to its $\mathcal{P}ar$.

3.2.1 *LISA*s Protocol Details

The FSM and pseudo-code for $\mathcal{D}ev$ are shown in an upper part of Figure 3 and Algorithm 3, respectively. Design choices are discussed in Appendix C. ***LISA***s is constructed such that $\mathcal{D}ev$ can receive a new $\mathcal{A}tt_{req}$ in any state, even while waiting for children's $\mathcal{A}tt_{rep}$-s. Besides $\mathcal{A}tt_{req}$ and $\mathcal{A}tt_{rep}$, ***LISA***s involves one extra message type:
(1) *request*: $\mathcal{A}tt_{req} = [\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{D}epth, \mathcal{A}uth_{req}]$,
(2) *report*: $\mathcal{A}tt_{rep} = [\text{“}rep\text{”}, \mathcal{S}eq, \mathcal{D}evID, \mathcal{D}esc, \mathcal{A}uth_{rep}]$, and
(3) *acknowledgment*: $\mathcal{A}tt_{ack} = [\text{“}ack\text{”}, \mathcal{S}eq, \mathcal{D}evID, \mathcal{P}ar]$, where:

- $\mathcal{S}nd$– identifier of the sending device; this field is not authenticated
- $\mathcal{S}eq$– sequence number and/or timestamp of the present attestation instance; set by $\mathcal{V}$rf
- $\mathcal{D}epth$– depth of the sending device in the spanning tree
- $\mathcal{A}uth_{req}$– authentication token for the attestation request message; computed by $\mathcal{V}$rf as: $\mathsf{MAC}(K, \text{“}req\text{”}||\mathcal{S}eq)$
- $\mathcal{D}evID$– identifier of $\mathcal{D}ev$ (stored in ROM, along with ***AttCode***)
- $\mathcal{D}esc$– list of $\mathcal{D}ev$'s descendants; populated when $\mathcal{D}ev$ receives an authentic report
- $\mathcal{A}uth_{rep}$– authentication token for the attestation reply message; computed by $\mathcal{D}ev$ as: $\mathsf{MAC}(K, \text{“}rep\text{”}||\mathcal{S}eq||\mathcal{D}evID||\mathcal{D}esc)$
- $\mathcal{P}ar$– identifier of reporting device's parent in the spanning tree; copied from $\mathcal{S}nd$ field in $\mathcal{A}tt_{req}$

Prover in ***LISA***s .

From the perspective of a prover $\mathcal{D}ev$, ***LISA***s consists of eight states:
1. Wait: the initial state where $\mathcal{D}ev$ waits for an attestation-relevant packet. $\mathcal{D}ev$ transitions to VerifyRequest if it is $\mathcal{A}tt_{req}$, Veri-

Algorithm 3: $\mathcal{D}ev$ pseudo-code in ***LISA***s.

Write-Protected Vars: $\mathcal{C}urSeq$ – current sequence number
$\mathcal{D}evID$ – id of Device $\mathcal{D}ev$
$\mathcal{P}ar$ – id of current $\mathcal{D}ev$'s parent
$\mathcal{C}$ – pre-installed hash of $\mathcal{D}ev$'s memory
$\mathcal{D}esc$ – list of id-s of $\mathcal{D}ev$'s descendants

$t_{ACK} \leftarrow t_{MAC} + 2t_t + t_s$;
while $True$ **do**
  $m \leftarrow$ NONBLOCKRECEIVE();
  $T \leftarrow$ GETTIME();
  if TYPE$(m) =$ "req" **then**
    $[\mathcal{A}uth_{req}, \mathcal{S}eq, \mathcal{S}nd, \mathcal{D}epth\ ] \leftarrow$ DECOMPOSE$(m,$ "req"$)$;
    if $\mathcal{S}eq < \mathcal{C}urSeq$ **then**
      CONTINUE();
    end
    if $\mathcal{A}uth_{req} \neq \mathsf{MAC}_K($"$req$"$\|\mathcal{S}eq)$ **then**
      CONTINUE();
    end
    $\mathcal{C}urSeq \leftarrow \mathcal{S}eq$; $\mathcal{P}ar \leftarrow \mathcal{S}nd$;
    $\mathcal{A}tt_{ack} \leftarrow$ "ack"$||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{P}ar$;
    UNICAST$(\mathcal{P}ar, \mathcal{A}tt_{ack})$;
    $t_{REP} \leftarrow (n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$;
    $\mathcal{A}tt_{req} \leftarrow$ "req"$||\mathcal{C}urSeq||\mathcal{D}evID||(\mathcal{D}epth{+}1)||\mathcal{A}uth_{req}$;
    BROADCAST$(\mathcal{A}tt_{req})$;
    $\mathcal{D}esc \leftarrow \emptyset$;
    $\mathcal{C}hildren \leftarrow \emptyset$;
    $T \leftarrow$ RESTARTTIMER();
  else if TYPE$(m) =$ "ack" **then**
    if $T > t_{ACK}$ **then**
      CONTINUE();
    end
    $[\mathcal{S}eq, \mathcal{S}nd, \mathcal{S}ndPar\ ] \leftarrow$ DECOMPOSE$(m,$ "ack"$)$;
    if $\mathcal{S}eq = \mathcal{C}urSeq$ **then**
      $\mathcal{C}hildren = \mathcal{C}hildren \cup \{\mathcal{S}nd\}$;
    end
  else if TYPE$(m) =$ "rep" **then**
    if $T \leq t_{ACK} \vee T \geq t_{REP}$ **then**
      CONTINUE();
    end
    $[\mathcal{S}eq, \mathcal{S}nd, \mathcal{S}ndDesc, \mathcal{A}uth_{rep}\ ] \leftarrow$ DECOMPOSE$(m,$ "rep"$)$;
    if $\mathcal{S}eq \neq \mathcal{C}urSeq$ **then**
      CONTINUE();
    end
    if $\mathcal{A}uth_{rep} = \mathsf{MAC}(K,$ "rep"$||\mathcal{C}urSeq||\mathcal{S}nd||\mathcal{S}ndDesc)$ **then**
      $\mathcal{D}esc \leftarrow \mathcal{D}esc \cup \{\mathcal{S}nd\} \cup \mathcal{S}ndDesc$;
      $\mathcal{C}hildren \leftarrow \mathcal{C}hildren \setminus \{\mathcal{S}nd\}$;
    end
  end
  if $(\mathcal{C}hildren = \emptyset \wedge T \geq t_{ACK}) \vee (T \geq t_{REP})$ **then**
    if $H(\mathcal{M}em) \neq \mathcal{C}$ **then**
      ABORT();
    end
    $\mathcal{A}uth_{rep} \leftarrow \mathsf{MAC}(K,$ "rep"$||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{D}esc)$;
    $\mathcal{A}tt_{rep} \leftarrow$ "rep"$||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{D}esc||\mathcal{A}uth_{rep}$;
    UNICAST$(\mathcal{P}ar, \mathcal{A}tt_{rep})$;
    $T \leftarrow$ RESETANDSTOPTIMER();
  end
end

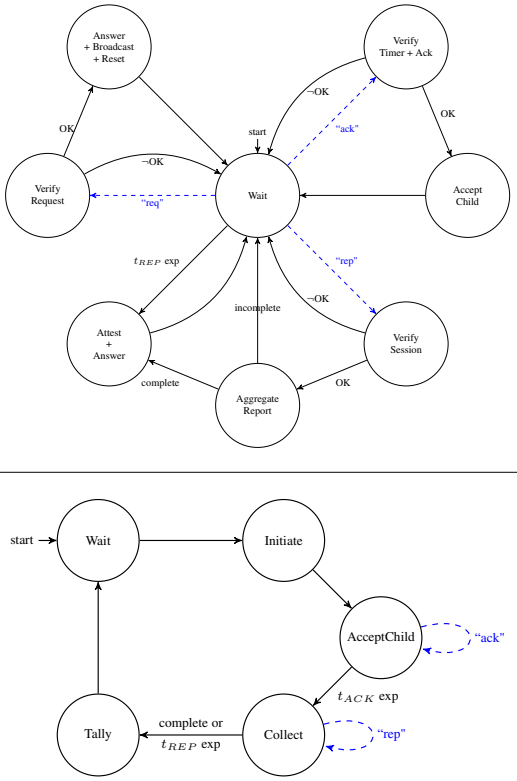


Figure 3: FSM of ***LISA***s: Dev (top) and Vrf (bottom)

fySession if it is $\mathcal{A}tt_{rep}$ and VerifyTimer+Ack if it is $\mathcal{A}tt_{ack}$. Also, if a timeout occurs during this state, $\mathcal{D}ev$ transitions to Attest+Answer. This timeout is set in Answer+Broadcast+Reset below.

2. VerifyRequest: This state is similar to VerifyRequest in ***LISA***s. If verification of $\mathcal{A}tt_{req}$ fails, $\mathcal{D}ev$ discards $\mathcal{A}tt_{req}$ and goes back to Wait. Otherwise, $\mathcal{D}ev$ realizes its depth in the spanning tree through $\mathcal{D}epth$ field in $\mathcal{A}tt_{req}$ and saves $\mathcal{S}eq$ as $\mathcal{C}urSeq$ and $\mathcal{S}nd$ as $\mathcal{P}ar$. Finally, $\mathcal{D}ev$ transitions to Answer+Broadcast+Reset.

3. Answer+Broadcast+Reset: $\mathcal{D}ev$ sends $\mathcal{A}tt_{ack}$ back to $\mathcal{P}ar$, copies its $\mathcal{D}evID$ into $\mathcal{S}nd$ field of $\mathcal{A}tt_{req}$ and broadcasts the modified $\mathcal{A}tt_{req}$. Next, $\mathcal{D}ev$ computes a timeout t_{REP}. This timeout is used to determine when to stop receiving $\mathcal{A}tt_{rep}$ during Wait. $\mathcal{D}ev$ then initializes a list of its children ($\mathcal{C}hildren$) and a list of its descendants ($\mathcal{D}esc$) to empty sets, starts a timer, and returns to Wait.

4. VerifyTimer+Ack: $\mathcal{D}ev$ receives $\mathcal{A}tt_{ack}$ from a device that wants to be its child. First, $\mathcal{D}ev$ checks with an acknowledgment timeout (t_{ACK}), which is a global constant. If the current time is later than t_{ACK}, $\mathcal{D}ev$ discards $\mathcal{A}tt_{ack}$ and returns to Wait. If the $\mathcal{S}eq$ in $\mathcal{A}tt_{ack}$ does not match $\mathcal{C}urSeq$, $\mathcal{D}ev$ also discards $\mathcal{A}tt_{ack}$ and goes back to Wait. Otherwise, $\mathcal{D}ev$ transitions to AcceptChild.

5. AcceptChild: $\mathcal{D}ev$ accepts $\mathcal{A}tt_{ack}$ and stores $\mathcal{S}nd$ into $\mathcal{C}hildren$. Then, $\mathcal{D}ev$ returns to Wait.

6. VerifySession: This state is also similar to VerifySession in ***LISA***α. $\mathcal{D}ev$ discards $\mathcal{A}tt_{rep}$ and return to Wait if $\mathcal{S}eq$ in $\mathcal{A}tt_{rep}$ does not match $\mathcal{C}urSeq$. Otherwise, it transitions to AggregateReport

7. AggregateReport: $\mathcal{D}ev$ accepts $\mathcal{A}tt_{rep}$ and aggregates it with other received reports in the same session. The aggregation is done by adding $\mathcal{S}nd$ and $\mathcal{D}esc$ fields in $\mathcal{A}tt_{rep}$ into its $\mathcal{D}esc$ and removing $\mathcal{S}nd$ from $\mathcal{C}hildren$ since $\mathcal{S}nd$ has replied. If all of its

children have already replied (or $\mathcal{C}hildren = \emptyset$), $\mathcal{D}ev$ transitions to Attest+Answer. Otherwise, $\mathcal{D}ev$ returns to Wait.

8. Attest+Answer: $\mathcal{D}ev$ computes a hash of its attestable memory. If the resulting digest does not match with the pre-installed hash value ($\mathcal{C}$), $\mathcal{D}ev$ outputs an error and acts accordingly (e.g., hardware reset and memory wipe-out). Otherwise, $\mathcal{D}ev$ constructs $\mathcal{A}uth_{rep}$ and $\mathcal{A}tt_{rep}$ as defined earlier and unicasts $\mathcal{A}tt_{rep}$ to $\mathcal{P}ar$. Finally, the timer is reset and stopped and $\mathcal{D}ev$ returns to Wait.

*Verifier in **LISA**s .*

The $\mathcal{V}$rf in ***LISA***s has one additional state – AcceptChild – while the rest of the states remain similar or the same as the ones in ***LISA***α. $\mathcal{V}$rf's pseudo-code is illustrated below and its finite state machine is in the lower part of Figure 3.

Algorithm 4: $\mathcal{V}$rf pseudo-code in ***LISA***s.

```
1  t_ACK ← t_MAC + 2t_t + t_s;
2  t_Attest ← n · (t_ACK + t_a + t_MAC + t_t + t_s);
3  while True do
4      wait();
5      InitID ← getInitId();
6      CurSeq ← getSeq();
7      Att_req ← "req"||Vrf||CurSeq||Auth_req;
8      unicast(InitID, Att_req);
9      Attest ← ∅; Children ← ∅;
10     Norep ← {allDevID};
11     T ← getTimer();
12     while T < t_ACK do
13         Att_ack ← receive();
14         [Seq, DevID, Par] ← decompose(Att_ack);
15         if Seq = CurSeq then
16             Children ← Children ∪ {DevID};
17         end
18     end
19     while T < t_REP do
20         Att_rep ← receive();
21         [Snd, Par, Seq, SndDesc, Auth_rep]
           ← decompose(Att_rep);
22         if Seq = CurSeq ∧ Auth_rep =
           MAC(K, "rep"||Seq||DevID||SndDesc) then
23             Attest ← (Attest ∪ SndDesc) ∪ {Snd};
24             Norep ← (Norep \ SndDesc) \ {Snd};
25             Children ← Children \ {Snd};
26         end
27         if Children = ∅ ∨ |Attest| = n then
28             break();
29         end
30     end
31     output(Attest, Norep);
32 end
```

1&2. Wait and Initiate: These two state are identical to their counterparts in ***LISA***α.

3. AcceptChild: $\mathcal{V}$rf waits for $\mathcal{A}tt_{ack}$-s from the initiator(s), which are used to determine the completion of Collect. After a timeout occurs, $\mathcal{V}$rf stops receiving $\mathcal{A}tt_{ack}$-s and transitions to Collect.

4. Collect: This state is also similar to Collect in ***LISA***α except the following three behaviors: One is $\mathcal{V}$rf does not need to check $H(\mathcal{M}em)$ since it is not include in $\mathcal{A}tt_{rep}$. Secondly, $\mathcal{V}$rf does not need to maintain a list of unsuccessfully attested devices ($Fail$) since software-infected devices cannot output an authentic $\mathcal{A}uth_{rep}$. Lastly, $\mathcal{V}$rf transitions to Tally if the initiator(s) (realized in AcceptChild) has sent its reports. The rest of its behaviors remains the same as in ***LISA***α.

5. Tally: $\mathcal{V}$rf outputs $Attest$ and $Norep$ and returns to Wait.

3.2.2 *Timeouts in **LISA**s*

$\mathcal{D}ev$ requires two timeouts: an acknowledgment timeout (t_{ACK}) and a report timeout (t_{REP}).

t_{ACK} is the amount of time that $\mathcal{D}ev$ waits after having broadcast a request for children acknowledgments. It is set to the constant value of $t_{MAC}+2t_t+t_s$, that is time for the broadcast to reach a neighbor (t_t), for the neighbor to verify the request (t_{MAC}), and then for the answer to be received by $\mathcal{D}ev$ (another t_t), plus some slack t_s (global parameter). This gives all neighbors an opportunity to send $\mathcal{A}tt_{ack}$.

t_{REP} is the amount of time, after the children have been determined, that $\mathcal{D}ev$ will wait for reports before performing its own attestation. It is set to the value $(n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$. This represents the time the descendants would take to answer back to $\mathcal{D}ev$ in the absolute worst scenario. This scenario is when the descendants are in a line topology and each has only one child (except the last one). It is indeed the worst case because no work can be done in parallel. Each node in the line (except the leaf) will perform the following: forward a request to and wait for the answer from its (only) child (t_{ACK}), and then, after receiving the answer, verify its child's report (t_{MAC}), attest itself (t_a), and finally answer back to the parent (t_t) and some slack t_s for variability considerations. In this scenario, all of $\mathcal{D}ev$'s descendants will take this time (except the leaf which takes slightly less). If $\mathcal{D}ev$ has $\mathcal{D}epth$ ancestors, it has at most $(n - \mathcal{D}epth)$ descendants. Hence, time for attestation of descendants is bounded, even in the worst-case scenario, by: $(n - \mathcal{D}epth)(t_{ACK} + t_a + t_{MAC} + t_t + t_s)$. Note that the timeouts are needed to detect errors and DoS attacks. In most topologies, the delay in gathering reports will be much shorter. Finally, $\mathcal{D}epth$ is important: without it, if a node times out, its parent (and all ancestors) will also time out. Then, $\mathcal{V}$rf would have no idea about what happened. Instead, if a node times out, it sends what it has thus far to its parent, which does not time out itself.

Since $\mathcal{V}$rf can be viewed as the root of the spanning tree, t_{ACK} and t_{REP} are applicable to it. $\mathcal{V}$rf's $t_{REP} = n \cdot (t_{ACK} + t_a + t_{MAC} + t_t + t_s)$.

3.2.3 *Connectivity in **LISA**s*

Let t_0 denote the time that $\mathcal{D}ev$ receives $\mathcal{A}uth_{req}$ from $\mathcal{P}ar$, t_1 denote the time that $\mathcal{P}ar$ receives $\mathcal{A}tt_{ack}$ from $\mathcal{D}ev$ and t_2 denote the time that $\mathcal{P}ar$ receives $\mathcal{A}uth_{rep}$ from $\mathcal{D}ev$. Then, we can formally state the connectivity assumption in ***LISA***s as follows:

LISAs provides correct swarm attestation result, i.e. no false-positive and false-negative, if a link between every $\mathcal{D}ev$ and its $\mathcal{P}ar$ exists during their t_0, t_1 and t_2.

Note that this assumption is more relaxed than the one in ***LISA***α since each link has to appear during those three times while in ***LISA***α, $\mathcal{D}ev$ and $\mathcal{P}ar$ have to be connected for transmitting $z + 1$ messages where z is a number of $\mathcal{D}ev$'s descendants. The downside, however, is that when the assumption does not hold, $\mathcal{V}$rf will lose all $\mathcal{A}uth_{rep}$-s of $\mathcal{D}ev$ and its descendants while in ***LISA***α, some of those $\mathcal{A}uth_{rep}$-s could still arrive to $\mathcal{V}$rf.

3.2.4 *QoSA of **LISA**s*

As presented in Algorithm 3, ***LISA***s offers L-QoSA. By changing information contained in the reports (line 48), QoSA can be amended to:

- *Binary:* Instead of $\mathcal{D}esc$, $\mathcal{A}tt_{rep}$ contains a single bit indicating whether all descendants have been successfully attested. This saves bandwidth over L-QoSA since reports are smaller (the MAC is also faster to compute). The obvious downside is that $\mathcal{V}$rf is only learns the result of swarm attestation as a whole, and can not identify missing devices. One option is to use ***LISA***s with B-QoSA until failure is encoun-

tered and then re-run ***LISA***s with higher QoSA to identify devices that failed attestation.

- *Counter:* Using a counter allows $\mathcal{V}\mathsf{rf}$ to learn how many devices failed attestation. This comes at a marginal increase is bandwidth consumption.
- *List Complement:* Instead of composing a list of attested descendants, each device can build a list of unattested ones. In a mostly healthy swarm, this is cheaper in terms of bandwidth than list QoSA.
- *Topology:* By representing the list of descendants as a tree instead of a set in the report, ***LISA***s can provide topology information to $\mathcal{V}\mathsf{rf}$. Specifically, line 39 is replaced by: $\mathcal{D}esc \leftarrow \mathcal{D}esc \cup (\mathcal{S}nd : \mathcal{S}ndDesc)$. This recursively creates a subtree rooted at each node. The only drawback is a small increase in bandwidth usage.

4. SECURITY ANALYSIS

We now describe possible attacks and then (informally) assess security of ***LISA***α and ***LISA***s.

4.1 Attack Vectors

Recall that our adversarial model only considers *remote and local adversaries*, while physical attacks are out of scope. An adversary $\mathcal{A}dv$ can remotely modify the software and/or the state of any device. It also has full control of all communication channels, i.e., can eavesdrop on, inject, delete, delay or modify any messages between devices, as well as between any device and $\mathcal{V}\mathsf{rf}$. In the context of ***LISA***, the following attacks are possible:

1. *Report Forgery:* Forging a $\mathcal{A}tt_{rep}$ would allow a device to evade detection of malware, or allow $\mathcal{A}dv$ to impersonate a device.
2. *Request Forgery:* Forging a $\mathcal{A}tt_{req}$ would trigger unnecessary swarm attestation and result in denial-of-service (DoS) for the entire swarm.
3. *Application Layer DoS:* $\mathcal{A}dv$ can launch a DoS attack abusing the attestation protocol itself. This type of attack can vary, depending on the protocol version. One example is flooding the swarm with fake $\mathcal{A}tt_{rep}$-s.
4. *DoS on Network Layer and Lower Layers:* $\mathcal{A}dv$ can launch a DoS attacks that target network, link and physical layers of devices. This includes radio jamming, random packet flooding, packet dropping, etc. We do not consider such attacks since they are not specific to swarm attestation.

4.2 Security of ***LISA***α

Report Forgery: Recall that $\mathcal{A}tt_{rep}$ in ***LISA***α is $[\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{P}ar, \mathcal{S}eq, H(\mathcal{M}em), \mathcal{A}uth_{rep}]$ where $\mathcal{A}uth_{rep} = \mathsf{MAC}(K, \text{“}rep\text{”} || \mathcal{D}evID || \mathcal{S}eq || H(\mathcal{M}em))$. If $\mathcal{A}dv$ produces an authentic $\mathcal{A}tt_{rep}$ for some device then one of the following must hold:

- $\mathcal{A}dv$ forges $\mathcal{A}uth_{rep}$ without knowing K: this requires $\mathcal{A}dv$ to succeed in a MAC forgery, which is infeasible, with overwhelming probability, given a secure MAC function.
- $\mathcal{A}dv$ knows K and constructs its own $\mathcal{A}uth_{rep}$: this is not possible, since only ***AttCode*** can read K, ***AttCode*** leaks no information about K beyond $\mathcal{A}uth_{rep}$, and $\mathcal{A}dv$ can not tamper with hardware.
- $\mathcal{A}dv$ modifies a compromised device's $\mathcal{D}evID$ which results in producing $\mathcal{A}uth_{rep}$ for another $\mathcal{D}evID$: since $\mathcal{D}evID$ “lives” in ROM, it can not be modified.

We note that replay attacks are trivially detected by $\mathcal{V}\mathsf{rf}$ since each $\mathcal{A}tt_{req}$ includes a unique monotonically increasing $\mathcal{S}eq$, which is authenticated by every $\mathcal{D}ev$ and included in $\mathcal{A}tt_{rep}$.

Request Forgery: Recall that $\mathcal{A}tt_{req}$ in ***LISA***α is $[\text{“}req\text{”}, \mathcal{S}nd, \mathcal{S}eq, \mathcal{A}uth_{req}]$ where $\mathcal{A}uth_{req} = \mathsf{MAC}(K, \text{“}req\text{”} || \mathcal{S}eq)$. An $\mathcal{A}dv$ that fakes an $\mathcal{A}tt_{req}$ must either create a forged $\mathcal{A}uth_{req}$ without K or somehow know K. Similar to report forgery above, neither case is possible due to our assumptions about the MAC function and inaccessibility of K.

Application Layer DoS: An $\mathcal{A}dv$ flooding the swarm with fake $\mathcal{A}tt_{rep}$-s and/or $\mathcal{A}tt_{req}$-s can result in an effective attack if it triggers a lot of computation on, and/or communication between, devices. Fake $\mathcal{A}tt_{req}$ flooding to a device is not very effective since it causes DoS for only that device which authenticates an $\mathcal{A}tt_{req}$ before doing further work and forwarding it. On the other hand, a fake $\mathcal{A}tt_{rep}$ sent to a single device can result in several devices forwarding garbage. This is because a device forwards a report to its parent (and further up the spanning tree) without any authentication. We consider this attack not to be severe because it does not trigger any other computation (only communication).

4.3 Security Analysis of ***LISA***s

Report Forgery: Analysis of this attack in ***LISA***s is similar to that in ***LISA***α. $\mathcal{A}tt_{rep}$ in ***LISA***s is $[\text{“}rep\text{”}, \mathcal{D}evID, \mathcal{C}urSeq, \mathcal{D}esc, \mathcal{A}uth_{rep}]$, where $\mathcal{A}uth_{rep} = \mathsf{MAC}(K, \text{“}rep\text{”} || \mathcal{D}evID || \mathcal{C}urSeq || \mathcal{D}esc)$. If $\mathcal{A}dv$ successfully forges a $\mathcal{A}tt_{rep}$ for one of the swarm devices such that $\mathcal{V}\mathsf{rf}$ accepts it, then one of the following must have occurred: (1) $\mathcal{A}dv$ forged $\mathcal{A}uth_{rep}$ violating security of the underlying MAC; (2) $\mathcal{A}dv$ learned K which is in ROM and only accessible from ***AttCode*** which is leak-proof; or (3) $\mathcal{A}dv$ was able to modify variables that “live" in write-protected memory (i.e., $\mathcal{C}urSeq$, $\mathcal{D}evID$, $\mathcal{P}ar$, $\mathcal{D}esc$ and $\mathcal{C}hildren$). This is also not possible due to the guaranteed write-protection from MPU access rules and ROM.

Request Forgery: $\mathcal{A}tt_{req}$ in ***LISA***s is similar to $\mathcal{A}tt_{req}$ in ***LISA***α except the additional field – $\mathcal{D}epth$. This field is, however, not utilized when checking integrity of $\mathcal{A}tt_{req}$. Thus, the analysis of this attack is similar to that in the ***LISA***α case above.

Application-Layer DoS: Fake request flooding in ***LISA***s has the same effect as that in ***LISA***α since the request of both protocols has similar format and is handled similarly. Fake report flooding, nonetheless, does not result in significant communication overhead since a device in ***LISA***s verifies all reports before aggregating them. In addition, ***LISA***s involves one additional type of message: $\mathcal{A}tt_{ack}$. Recall that $\mathcal{D}ev$ in ***LISA***s constructs $\mathcal{C}hildren$ based on $\mathcal{A}tt_{ack}$-s and then waits for reports until $\mathcal{C}hildren$ is empty or a timeout t_{REP} occurs. A fake $\mathcal{A}tt_{ack}$ causes devices to wait longer than necessary. However, such waiting time is still bounded by t_{REP} and thus in the worst case this attack will result in timeout of all of its ancestor devices. This attack is not severe since it does not incur extra computation on any devices, and produces effects similar to DoS attacks on lower layers. Fake $\mathcal{A}tt_{ack}$ flooding is also possible in ***LISA***s, though it results in DoS for targeted devices and not the entire swarm.

5. EXPERIMENTAL ASSESSMENT

We implemented ***LISA***α and ***LISA***s in Python, and assessed their performance by emulating device swarms using the open-source Common Open Research Emulator (CORE) [2].

5.1 Experimental Setup and Parameters

CORE Emulator: CORE is a framework for emulating networks. It allows defining network topologies with lightweight virtual machines (VMs). CORE contains Python modules for scripting network emulations at different layers of the TCP/IP stack and allows defining mobility scenario for nodes, configuring routing and

switching, network traffic and applications running inside emulated VMs. One key advantage of CORE over other simulation frameworks, such as ns2 or ns3, is that it utilizes the actual network stack in Linux and instantiates each emulated node using a lightweight VM with its own file system running its own processes. Using the actual networking stack results in performance estimates very close to reality, since it does not abstract away any implementation details or issues at the data link, network and transport layers.

Experimental Setup & Scenarios: We generated several CORE scenarios with n nodes each. In every scenario, the positions of the n nodes are chosen uniformly at random in an area of $1,500 \times 800$ units, e.g., meters. A pair of nodes is connected if the distance between them is smaller then a threshold of 200 units, corresponding roughly to the coverage range of 802.11/WiFi. If the resulting network is not connected, the above process is repeated until a connected network is generated. $\mathcal{V}\mathsf{rf}$ is also randomly positioned within the area, and connected to the generated network. Figure 4 in shows a sample configuration.

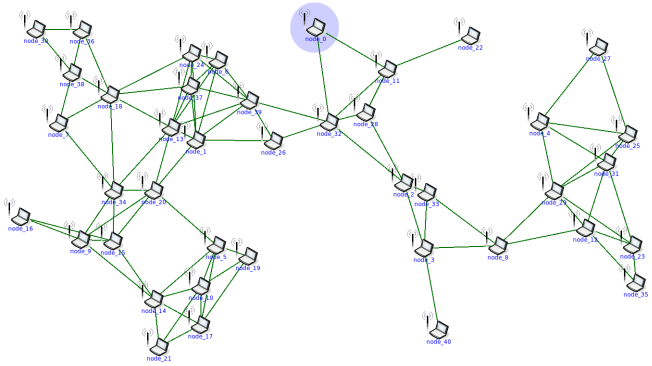


Figure 4: Example scenario generated in CORE (40 nodes). $\mathcal{V}\mathsf{rf}$ is highlighted.

The link-layer medium access control protocol running between neighboring nodes is 802.11. Network layer (IP) routing tables are automatically populated via the Optimized Link State Routing (OLSR) protocol, an IP-based routing protocol optimized for MANETs. OLSR uses proactive link-state routing with “Hello" and “Topology Control" messages that discover and disseminate link state information throughout the network. Each node uses topology information to compute next-hop destinations for all other nodes using the shortest path algorithm. Each node then runs our swarm attestation protocol, though instead of actually performing cryptographic operations, we insert delays (specified below) corresponding to time to perform such operations on low-end devices. At the beginning of each experiment, $\mathcal{V}\mathsf{rf}$ broadcasts a new $\mathcal{A}tt_{req}$ that is propagated throughout the network according to ***LISA***α or ***LISA***s.

Timings of Cryptographic Operations: Delays used to mimic cryptographic operations on low-end devices are as follows:

- $\mathcal{V}\mathsf{rf}$ signature computation/verification: $0.0001s$
- Node signature computation/verification: $0.001s$
- Node hash speed (for attestation): $0.0429s$/MB

The scheme used to sign messages can be implemented by a MAC or a public key signature scheme such as ECDSA (see Section 6). These timings are based on using ECDSA-160 and SHA-256. Using a MAC would reduce the time for small memory sizes. However, as discussed in Sections 5.2 and 6, computation is generally dominated by hashing. Numbers for $\mathcal{V}\mathsf{rf}$ are derived from a typical laptop, and those for $\mathcal{D}ev$ nodes come from a Raspberry Pi-2.

5.2 Experimental Results

In each experiment, we measured: (a) total time to perform swarm attestation: from $\mathcal{V}\mathsf{rf}$ sending $\mathcal{A}tt_{req}$, until $\mathcal{V}\mathsf{rf}$ finishes verification of the last $\mathcal{A}tt_{rep}$; (b) average CPU time for $\mathcal{D}ev$ to performing attestation; and (c) average number of bytes transmitted per $\mathcal{D}ev$. Figure 5 shows the results for both protocols with various amounts of attested memory and different swarm sizes. Each data point is obtained as an average over 30 randomly generated scenarios for that setup.

Total time (Figure 5a) varies significantly between ***LISA***α and ***LISA***s. This is because in ***LISA***s nodes spend a lot of time waiting for external input, without computing anything. In these results, the factor varies between 2 (for 1MB) to 8 (for 100MB). This time is also heavily influenced by the size of the attested memory, as shown in Figure 5b. Finally, total attestation time depends (roughly logarithmically) on n, since nodes are explored in a tree fashion. Although random, the tree is expected to be relatively well-balanced; see Figure 4.

Average CPU time (Figure 5b) for $\mathcal{D}ev$ is roughly equivalent in both protocols. This might seem counterintuitive, since in ***LISA***s nodes verify $\mathcal{A}tt_{rep}$-s of their children. However, verification is much cheaper than attestation, in particular, if attested memory size is large. This is discussed in Section 6. The number of devices (n) also has little effect on computation costs. On the other hand, the amount of attested memory has a strong impact on $\mathcal{D}ev$'s CPU usage. This shows that both protocols impose negligible extra overhead (over single-prover attestation) in terms of CPU usage.

Bandwidth usage (Figure 5c) is, as expected, higher in ***LISA***α than in ***LISA***s. The exact difference factor depends on n, ranging from negligible (5 nodes) to 3 (40 nodes). This only represents payloads size. Nodes in ***LISA***α also send more packets[5], compared to only 3 in ***LISA***s: $\mathcal{A}tt_{req}$, $\mathcal{A}tt_{ack}$, and $\mathcal{A}tt_{rep}$. Bandwidth usage is also roughly linear in terms of n. The size of the memory does not affect bandwidth usage, since data transmitted by nodes is independent to it.

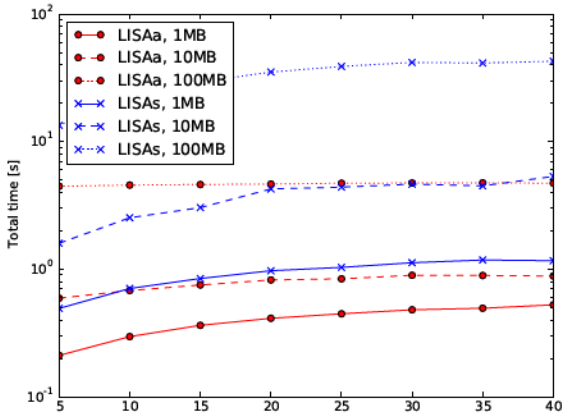
6. CRYPTOGRAPHIC CHOICES

As described above, both ***LISA***α and ***LISA***s assume that symmetric cryptography is used for constructing the MAC primitive, i.e., a keyed cryptographic hash function [5] or a CBC-based MAC [11]. Key management is trivial: a single master key shared between $\mathcal{V}\mathsf{rf}$ and all devices is used for computing and verifying all attestation reports. However, under some conditions, it might be desirable or even preferable to apply less naïve key management techniques and/or take advantage of public-key cryptography.

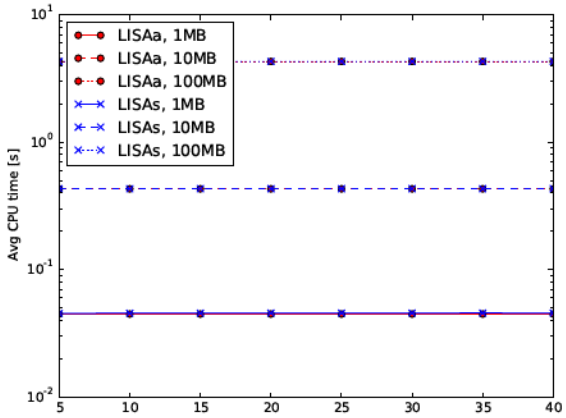
Physical Attacks: As stated earlier, ***LISA***α and ***LISA***s consider physical attacks to be out of scope, following most prior literature on this topic. Thus, ***SMART+*** architecture, coupled with a single shared symmetric master key, is sufficient for attesting the entire swarm in the presence of *Remote* and *Local* adversaries, as defined in Section 2.2. However, in the presence of physical adversaries, neither scheme is secure. A physical attack on a single device exposes the master key, which allows the adversary to impersonate all other devices as well as $\mathcal{V}\mathsf{rf}$.

Device-Specific Keying: One natural mitigation approach is to impose a unique key that each device shares with $\mathcal{V}\mathsf{rf}$. That way, the adversary learns only one key upon compromising a single device. Although this approach would work well with ***LISA***α, it requires changes to ***LISA***s to support key establishment among neighboring

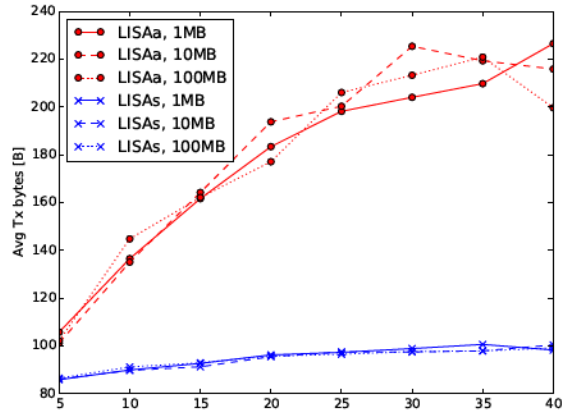
[5] The number of packets sent by both protocols, not depicted here, follows a behavior very similar to Figure 5c.



(a) Total time [s] for swarm attestation in ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (log y-scale).



(b) Average CPU time [s] per device for ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (log y-scale).



(c) Average # bytes transmitted per device for ***LISA***α and ***LISA***s, for different memory sizes, as a function of n (linear y-scale).

Figure 5: Experimental Results for ***LISA***α and ***LISA***s

devices; this would likely entail the use of public key cryptography, e.g., using the Diffie-Hellman key establishment protocol.

$\mathcal{A}tt_{req}$ Authentication: Device-specific symmetric keying also does not address the issue of $\mathcal{V}$rf impersonation. To allow devices to authenticate each $\mathcal{A}tt_{req}$ individually, $\mathcal{V}$rf would need to compute n distinct $\mathcal{A}uth_{req}$ tags, one for each device. This might incur significant computational and bandwidth overheads, if n is large. For small swarms, the tradeoff could be acceptable. Regardless of whether a single master key or device-specific keys are used, one simple step towards preventing $\mathcal{V}$rf impersonation and consequent DoS attacks is to impose a public key on $\mathcal{V}$rf only. In other words, $\mathcal{V}$rf would sign each $\mathcal{A}tt_{req}$, thus changing the format of $\mathcal{A}uth_{req}$ to: $SIGN(SK_{\mathcal{V}\text{rf}}, \text{“}req\text{”}||\mathcal{S}eq)$ where $SK_{\mathcal{V}\text{rf}}$ is $\mathcal{V}$rf's private key and $PK_{\mathcal{V}\text{rf}}$ is its public counterpart, assumed to be known to all devices. One obvious downside of this simple method is the extra computational overhead of verifying $\mathcal{A}uth_{req}$. We note that the combination of: (1) public key-based $\mathcal{A}tt_{req}$ authentication, and (2) per device symmetric keys, is both appropriate and efficient for ***LISA***α, which does not require devices to authenticate each other's $\mathcal{A}tt_{rep}$-s. It makes less sense for ***LISA***s, due to the need for a means to authenticate one's neighbors' $\mathcal{A}tt_{rep}$-s.

Public Keys for All: Predictably, we now consider imposing a unique public/private key-pair for each device. Admittedly, this only mitigates the effects of physical attacks and clearly does not prevent them. However, a successful physical attack on a single device yields knowledge of that device's secret key and does not lead to impersonation, or easier compromise, of other devices. For ***LISA***α, there is almost no difference between the full-blown public key approach and device-specific symmetric keying, as long as either is coupled with public key-based $\mathcal{A}tt_{req}$ authentication. The only issue arises if $\mathcal{V}$rf is not fully trusted; in that case, the former is preferable since $\mathcal{V}$rf would not be able to create fake $\mathcal{A}tt_{rep}$-s. For ***LISA***s, using device-specific public keys is conceptually simpler as there would be no need to establish pairwise keys between neighbors.

Attested Memory Size: An orthogonal (non-security) issue influencing cryptographic choices is the size of attested memory. Considering relatively weak low-end embedded devices, the cost of computing a symmetric MAC (dominated by computing a hash) over a large segment of memory might exceed that of computing a single public key signature. In that case, it makes sense to employ a digital signature in both ***LISA***α and ***LISA***s. To illustrate this point, Figure 6 compares performance of SHA-256 with several signature algorithms on Raspberry Pi-2. When attested memory size reaches 45KB, the run-time of Elliptic Curve Digital Signature Algorithm (ECDSA) with a 256-bit public key catches up to that of SHA-256. Hence, at least with Raspberry Pi-2, it makes sense to switch to ECDSA-256 for memory sizes exceeding 4.5MB – at that point, ECDSA-256 consumes less than 1% of total attestation runtime.

7. CONCLUSIONS & FUTURE WORK

This paper brings swarm RA closer to reality by designing two simple and practical protocols: ***LISA***α and ***LISA***s. To analyze and compare across protocols we introduced a new metric: Quality of Swarm Attestation (QoSA) which captures the information offered by a specific swarm RA protocol. Issues for future work include: (i) formally proving security for swarm protocols, and (ii) trial deployment of proposed protocols on device swarms.

Acknowledgments

The authors are grateful to AsiaCCS'17 anonymous reviewers for helpful comments and improvement suggestions. UCI authors were

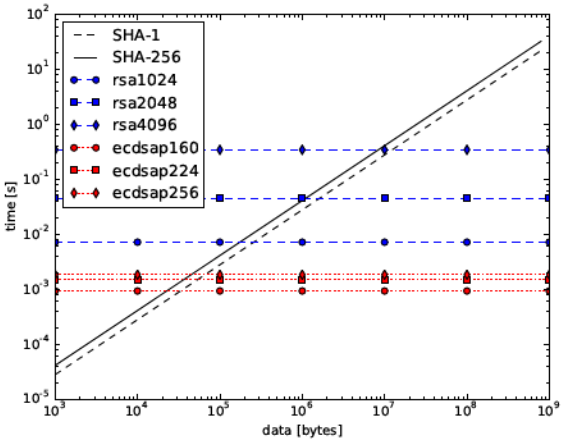


Figure 6: Performance Comparison: Hash & Signature on Raspberry Pi-2@900MHz [17].

supported, in part, by funding from: (1) the National Security Agency (NSA) under contract H98230-15-1-0276, (2) the Department of Homeland Security, under subcontract from the HRL Laboratories, (3) the Army Research Office (ARO) under contract W911NF-16-1-0536, and (4) a fellowship of the Belgian American Educational Foundation.

8. REFERENCES

[1] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik. Invited: Things, trouble, trust: on building trust in IoT systems. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[2] J. Ahrenholz. Comparison of CORE network emulation platforms. In *IEEE Military Communications Conference (MILCOM)*, 2010.

[3] Apple Computer, Inc. LibOrange, 2006.

[4] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. SEDA: Scalable embedded device attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[5] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *IACR Crypto*, 1996.

[6] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: tiny trust anchor for tiny devices. In *ACM/IEEE Design Automation Conference (DAC)*, 2015.

[7] F. Brasser, A.-R. Sadeghi, and G. Tsudik. Remote attestation for low-end embedded devices: the prover's perspective. In *ACM/IEEE Design Automation Conference (DAC)*, 2016.

[8] J. Camhi. BI Intelligence projects 34 billion devices will be connected by 2020.

[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[10] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni. DARPA: Device attestation resilient to physical attacks. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[11] Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher. Standard, ISO.

[12] A. Kandhalu, K. Lakshmanan, and R. R. Rajkumar. U-connect: a low-latency energy-efficient asynchronous neighbor discovery protocol. In *ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN)*, 2010.

[13] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *ACM European Conference on Computer Systems (EuroSys)*, 2014.

[14] M. Kohvakka, J. Suhonen, M. Kuorilehto, V. Kaseva, M. Hännikäinen, and T. D. Hämäläinen. Energy-efficient neighbor discovery protocol for mobile wireless sensor networks. *Ad Hoc Networks*, 7(1):24–41, 2009.

[15] B. Parno. Bootstrapping trust in a Trusted Platform. In *3rd USENIX Conference on Hot Topics in Security (HotSec)*, 2008.

[16] D. Puri. Got milk? IoT and LoRaWAN modernize livestock monitoring.

[17] RASPBERRY PI FOUNDATION. RASPBERRY PI 2 MODEL B, 2015.

[18] A.-R. Sadeghi, M. Schunter, A. Ibrahim, M. Conti, and G. Neven. SANA: Secure and scalable aggregate network attestation. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[19] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In *International workshop on swarm robotics*, pages 10–20. Springer, 2004.

[20] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1):13 – 22, 2008.

[21] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *ACM Workshop on Wireless Security (WiSe)*, 2006.

[22] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, December 2005.

[23] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Research in Security and Privacy (S&P)*, 2004.

[24] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert. A robust integrity reporting protocol for remote attestation. In *Workshop on Advances in Trusted Computing (WATC)*, 2006.

[25] Trusted Computing Group. Trusted platform module (tpm).

[26] R. Waugh. Smart TV hackers are filming people having sex on their sofas.

APPENDIX

A. RELATED WORK

Initial RA efforts relied on secure hardware exemplified by the Trusted Platform Module (TPM) [25]. (A TPM is a secure co-processor, designed for protecting a secret key as well as performing cryptographic operations using that key. Attestation evidence can be securely created inside a TPM by computing an unforgeable keyed hash over the hardware and software states.) Such techniques assume existence of a TPM on $\mathcal{P}rv$. However, medium- and low-end (as well as legacy) devices generally can not accommodate a TPM. This motivated the development of software-only (i.e., no hardware support) RA techniques, such as [23] and [22]. The main idea is that, for each attestation instance, $\mathcal{V}\mathsf{rf}$ sends $\mathcal{P}rv$ a custom and randomized checksum function with run-time side-effects that the latter uses to compute the attestation token. Any attempt by $\mathcal{P}rv$-resident malware to evade the checksum (e.g., by copying memory during attestation) will result in a noticeable delay which is then detected by $\mathcal{V}\mathsf{rf}$. However, this approach makes a strong assumption that $\mathcal{V}\mathsf{rf}$ and $\mathcal{P}rv$ are one hop apart, i.e., a round-trip delay is either negligible or fixed. While this works in some scenarios (notably, attestation of peripherals or legacy devices), it is not suitable for RA performed over a network. The previous assumption is sufficient for mitigating a remote adversary. To protect against a local adversary, an additional assumption is needed: any message sent or received by $\mathcal{P}rv$ during attestation must be overhear-able by $\mathcal{V}\mathsf{rf}$. This is needed to protect against a cuckoo attack [15].

A.1 Hybrid Single-Prover RA

Hybrid techniques provide RA while aiming to minimize hardware features. This approach was first explored in SMART [9], where attestation software (ROM code) resides in immutable storage to prevent it from being modified by malware. SMART also requires a hard-wired MPU to ensure exclusive access (by ROM code) to the attestation key. Furthermore, all interrupts are disabled during execution of ROM code in order to ensure atomicity. Plus, MPU controls assure that ROM code starts execution only at its legal entry point and similarly exits only at legal exit point(s).

TrustLite [13] extends SMART by supporting isolation of software modules, called trustlets. One of its distinguishing features is that an interrupts do not need to be disabled during attestation. TrustLite modifies the CPU Exception Engine to support interrupt handling in a special trustlet. Similar to SMART, access to the attestation key is guarded by hardware-enforced MPU in ROM. TyTAN [6] adopts a similar RA technique by combining hardware-assisted isolation of software components with real-time execution.

SMART, TrustLite and TyTAN assume that $\mathcal{V}\mathsf{rf}$ is always trusted. However, in practice, an adversary can launch DoS attacks by impersonating $\mathcal{V}\mathsf{rf}$ and deluging $\mathcal{P}rv$ with with fake requests. To remedy this issue, [7] focused on $\mathcal{P}rv$'s perspective and concluded that an attestation request must have authenticated integrity (via symmetric MAC or a public-key signature) provided by $\mathcal{V}\mathsf{rf}$ and checked by $\mathcal{P}rv$. Also, to protect against replay and re-ordering attacks, $\mathcal{P}rv$ needs either: (1) a secure writeable memory location to store a counter, or (2) a reliable read-only clock.

A.2 Swarm RA

Although much research effort has been invested into single-prover RA techniques, RA of swarms or networks of interconnected devices is a relatively new topic.

SEDA [4] is one of the few concrete and relevant results. In it, $\mathcal{V}\mathsf{rf}$ starts the swarm attestation protocol by sending a request to an initiator device, selection of which is flexible. Having received a request, a device accepts the sender as its parent and forwards that request to its neighbors. An attestation report of any device is created – and protected using a secret key (distributed as part of the off-line phase) shared between that device and its parent – and sent to its parent device. Once it receives all of its children's reports, a device aggregates them into a single report. This process is recursive until the initiator receives all reports from its children. The initiator then combines those reports, signs them using its private key and returns the final result to $\mathcal{V}\mathsf{rf}$.

Secure and Scalable Aggregate Network Attestation (SANA) [18] extends SEDA [4] by providing a more efficient and scalable swarm attestation scheme based upon Optimistic Aggregate Signatures. OAS allows many individual signatures to be efficiently combined into a single aggregated signature, which can be verified much faster than all individual signatures. SANA's scalability is demonstrated via simulation showing that it can attest a million devices in 2.5 seconds.

Device Attestation Resilient to Physical Attacks (DARPA) [10] provides a way to mitigate physical attacks in a network of devices. The rationale behind DARPA is that, an adversary that performs a physical attack on a single device needs to spend a non-negligible amount of time to physically compromise that device. Hence, in order to detect device absence, DARPA requires each device to periodically monitor other devices by recording their *heartbeat*, at regular time intervals. $\mathcal{V}\mathsf{rf}$ can then detect any absent device when collecting these heartbeats. DARPA can also be used in a conjunction with any swarm attestation scheme (e.g. ***LISA***, SEDA or SANA) to provide protection in the presence of a physical adversary.

B. COMPLEXITY CONSIDERATIONS

B.1 Complexity of *LISA*α

Architectural Impact: Roughly speaking, the ***LISA***α protocol adheres to ***SMART+*** security architecture, i.e., it does not impose any additional features. However, it clearly requires a larger ROM to house additional code, and a more complex MPU to implement access rules. Compared to ***SMART+***, ROM size is expected to grow by just 30 bytes, as shown in Table 1. Also, ***LISA***α introduces two extra write-protected variables: $\mathcal{S}eq$ and $\mathcal{P}ar$. We assume that each can be a 32-bit value, i.e., only 8 extra bytes need to be write-protected. Finally, MPU needs to support two additional access rules to protect these two variables. The resulting increase in hardware complexity is shown in Figure 1 and Table 1.

Table 1: Estimated code complexity; all code in "C".

	METHOD:			
	SMART+ w/o MAC[6]	***SMART+***	***LISA***α	***LISA***s
Lines of Code	43	262	269	321
Executable Size (bytes)	8,565	17,896	17,928	18,128
Write-Protected Memory Size (bytes)	n/a	4	12	40 + 4n

Software Complexity: ***LISA***α needs only one simple extra operation: message ($\mathcal{A}tt_{req}$) broadcast. This operation is usually straight-forward in practice if a device is already capable of unicasting. Moreover, ***LISA***α is nearly stateless: only $\mathcal{S}eq$ and $\mathcal{P}ar$ need to persist between attestation sessions. Table 1 shows that ***LISA***α is only about 2% higher than single-prover ***SMART+*** in terms of lines-of-code (LoC-s).

[6]MAC is implemented as HMAC-SHA-256 from [3]

Communication Overhead: We assume an SHA-256-based hash and MAC constructs, each yielding a 32-byte output. Overall size of $\mathcal{A}tt_{req}$ is thus 43 bytes: 3 – message tag, 4 – $\mathcal{S}nd$, 4 – $\mathcal{S}eq$, and 32 – $\mathcal{A}uth_{req}$. Meanwhile, $\mathcal{A}tt_{rep}$ is 79 bytes: 3 – message tag, 4 – $\mathcal{D}evID$, 4 – $\mathcal{P}ar$, 4 – $\mathcal{S}eq$, 32 – $\mathcal{A}uth_{rep}$, and 32 – $H(\mathcal{M}em)$. We also assume that $\mathcal{D}ev$ has z descendants and w neighbors in the swarm spanning tree, and there are n devices total.

In each session, $\mathcal{D}ev$ receives up to w $\mathcal{A}tt_{req}$-s and exactly z $\mathcal{A}tt_{rep}$-s. Thus, depending on topology, $\mathcal{D}ev$ might receive anywhere between $(43 + 79z)$ and $(43w + 79z)$ bytes. Also, $\mathcal{D}ev$ broadcasts one $\mathcal{A}tt_{req}$ to neighbors and unicasts $(z + 1)$ $\mathcal{A}tt_{rep}$-s to $\mathcal{P}ar$. Thus, overall transmission cost for each $\mathcal{D}ev$ is: $43 + 79(z + 1)$.

Clearly, potentially high communication overhead is ***LISA***α's biggest drawback, since a device – in the worst case – transmits n reports. This motivated us to design ***LISA***s which reduces communication overhead by aggregating $\mathcal{A}tt_{rep}$-s.

B.2 Complexity of *LISA*s

Architectural Impact: ***LISA***s does not require any additional secure hardware features, and, in coarse terms, adheres to ***SMART+***. However, ROM needs to be expanded by around 200 bytes to support larger code. Also, ***LISA***s has 5 write-protected values (while ***SMART+*** has one): $\mathcal{S}eq$, $\mathcal{P}ar$, $\mathcal{C}$ and $\mathcal{D}esc$. To guard them, MPU needs to include at least as many memory access control rules. Each of the first two is a 4-byte integer, while $\mathcal{C}$ is 32 bytes, while the size of $\mathcal{D}esc$ is proportional to n. In total, $\mathcal{D}ev$ needs $40 + 4n$ bytes of write-protected memory, which is $O(n)$. Protecting a fixed-size value is clearly easier than a variable-size one. However, Appendix E illustrates a simple mechanism that accommodates variable-size data with implicit write-protection with minimal (constant) added space complexity for write-protected memory.

Software Complexity: ***LISA***s's software is much more complex than that of ***SMART+*** or ***LISA***α. Compared to ***LISA***α, ***LISA***s has three extra states, needed to: (1) determine timeouts, (2) establish parent-child relationship, and (3) handle report aggregation. For that, $\mathcal{D}ev$ needs to store additional variables(two of which are arrays): $\mathcal{D}epth$, $\mathcal{P}ar$, $\mathcal{D}esc$ and $\mathcal{C}hildren$, in each attestation session. This makes ***LISA***s a stateful protocol. In terms of LoC-s, ***LISA***s is approximately 22% and 19% over ***SMART+*** and ***LISA***α, respectively.

Bandwidth Usage: Suppose $\mathcal{D}ev$ has q children, z descendants and w neighbors. Compared to ***LISA***α, $\mathcal{A}tt_{req}$ includes one extra field: $\mathcal{D}epth$– 4 bytes. Thus, the size of $\mathcal{A}tt_{req}$ in ***LISA***s is 47 bytes. $\mathcal{A}tt_{rep}$ does not include $H(\mathcal{M}em)$ and $\mathcal{P}ar$. However, it contains additional variable-size data, $\mathcal{D}esc$, which can be as long as $4z$. Thus, the size of $\mathcal{A}tt_{rep}$ is $47 + 4z$ bytes. Finally, $\mathcal{A}tt_{ack}$ size is 15 bytes: 3 – message tag, 4 – $\mathcal{S}eq$, 4 – $\mathcal{D}evID$ field and 4 – $\mathcal{P}ar$.

In each session, $\mathcal{D}ev$ broadcasts one $\mathcal{A}tt_{req}$ to its neighbors, plus unicasts one $\mathcal{A}tt_{rep}$ and one $\mathcal{A}tt_{ack}$ to $\mathcal{P}ar$. Thus, the overall transmission cost for $\mathcal{D}ev$ is $47+47+4z+15 = 109+4z$. In the same session, $\mathcal{D}ev$ receives up to w $\mathcal{A}tt_{req}$-s, exactly q $\mathcal{A}tt_{rep}$-s and q $\mathcal{A}tt_{ack}$-s. Hence, in the worst case, $\mathcal{D}ev$ receives (in bytes):

$$47w+\sum_{i=1}^{q}(47+4z_i)+15q = 47w+47q+4(z-q)+15q = 47w+58q+4z$$

where z_i is the number of descendants of i^{th} child of $\mathcal{D}ev$.

Overall, ***LISA***s reduces the number of messages, compared to ***LISA***α. Each $\mathcal{D}ev$ transmits a fixed number of messages while, in ***LISA***α, this depends on the number of descendants and neighbors.

C. *LISA*s: DESIGN CHOICES

We now consider some details of Algorithm 3.

- Line 3: Reception of messages should be non-blocking, such that the timer can be checked even when no message is received[7].
- Line 7: Freshness of $\mathcal{S}eq$ in $\mathcal{A}tt_{req}$ is established by comparing it to $\mathcal{C}urSeq$, as in ***LISA***α. During any given session, a node acknowledges to the first neighbor that broadcasts an attestation request with $\mathcal{C}urSeq$. Subsequent broadcasts with the same $\mathcal{C}urSeq$ are ignored.
- Line 14: $\mathcal{A}tt_{ack}$ to $\mathcal{P}ar$ is constructed, consisting of: $\mathcal{S}eq$, $\mathcal{D}evID$ and $\mathcal{P}ar$. These values are not authenticated since $\mathcal{A}tt_{ack}$ is only used for determining timeouts. An adversary can send fake $\mathcal{A}tt_{ack}$-s to $\mathcal{D}ev$ which would only cause longer timeouts.
- Line 17: The request contains: $\mathcal{S}eq$, $\mathcal{S}nd$ and $\mathcal{D}epth$. Authentication of $\mathcal{S}eq$ is required to prevent replay attacks while $\mathcal{S}nd$ and $\mathcal{D}epth$ do not need to be authenticated. If either or both of the latter are modified by a local adversary, the result would be a DoS attack.
- Lines 19 and 20: The sets $\mathcal{D}esc$ and $\mathcal{C}hildren$ are re-initialized, i.e., set to empty. The former represents $\mathcal{D}evID$'s of $\mathcal{D}ev$'s descendants, which is populated when reports from children are verified (line 39). The latter represents the set of $\mathcal{D}ev$'s children. It is populated whenever a neighbor acknowledgment is verified (line 28) and de-populated when a child's report is verified (line 40). If $\mathcal{C}hildren$ is empty at any time after t_{ACK}, it means all the reports of $\mathcal{D}ev$'s children have been attested and $\mathcal{D}ev$ may proceed with self-attestation (line 43).
- Line 21: The timer is reset whenever a request is verified and re-broadcast.
- Lines 27 and 35: The received sequence number $\mathcal{S}eq$ is compared to the one stored for the last accepted request – $\mathcal{C}urSeq$. If they differ, the message (acknowledgment or report) is discarded. This comparison incurs a negligible cost while preventing acknowledgments and reports from older sessions being verified when a new attestation session has started. It also mitigates DoS attacks whereby a remote adversary (unaware of the current $\mathcal{S}eq$) sends fake acknowledgments or reports.
- Lines 44 and 45: A hash of specified memory range of $\mathcal{D}ev$ is computed and compared to its reference value – $\mathcal{C}$. If they do not match, ***LISA***s returns an error, performs a hardware re-set and cleans up its memory. $\mathcal{C}$ needs to be write-protected. This can be enforced by a static MPU rule.
- Line 48: $\mathcal{A}uth_{rep}$ contains authenticated fields: $\mathcal{S}eq$, $\mathcal{S}nd$ and $\mathcal{S}ndDesc$. $\mathcal{S}eq$ and $\mathcal{S}nd$ need to be validated to ensure authenticity and prevent replay attacks. $\mathcal{S}ndDesc$ is also authenticated to prevent a man-in-the-middle attacks that might overwrite attestation status of some descendants. Self-attestation must be performed last – after verifying reports of all children. If performed earlier, the protocol becomes vulnerable to a sort of a time-of-check-to-time-of-use (TOCTOU) attack where $\mathcal{D}ev$ gets corrupted after performing self-attestation and before sending out the aggregated report.
- Line 50: The timer is reset and stopped when attestation is completed. It is stopped so the condition at line 43 does not hold until a new $\mathcal{A}uth_{req}$ is received.

D. *LISA*s: IMPLICIT ACKS

[7]Note that, in loops with only non-blocking operations, it is necessary to avoid busy waiting; this is usually done by adding a short sleep timer at each iteration.

All $\mathcal{A}tt_{ack}$-s in ***LISA**s* are immediately followed by an attestation request broadcast. This means that $\mathcal{A}tt_{ack}$-s are, in principle, redundant, since an attestation request broadcast received from a neighbor can be viewed as an implicit acknowledgment. For that, the $\mathcal{P}ar$ has to be added in the request broadcast ($\mathcal{A}tt_{req}$ on line 17 becomes "req"$||\mathcal{A}uth_{rep}||\mathcal{C}urSeq||\mathcal{D}evID||\mathcal{P}ar||(\mathcal{D}epth+1)$), since a device must be able to distinguish between a broadcast of a child (implicit acknowledgment) and that of a non-child neighbor (concurrent broadcast) which should be ignored. This would make the protocol slightly more efficient since one less message has to be computed and transmitted, though less intuitive.

E. IMPLICIT WRITE PROTECTION FOR VARIABLE-SIZE DATA

Let x be a variable-sized data that needs to be write-protected. Let h_x be a fixed-size memory location that stores $H(x)$. Instead of enforcing access rules for x, the MPU ensures that only h_x is write-protected. Whenever x is modified to x', MPU stores $H(x')$ at h_x. Whenever x (as a whole or any part thereof) needs to be read, MPU first checks whether $h_x = H(x)$. This does not prevent malware from modifying x. However, any unauthorized change is detected upon the next read, which is sufficient for our purposes.

We note that SEDA already implicitly requires write-protected variable-sized data even though [4] does not discuss how this can be achieved in practice.