This is a full version of the paper which appeared in:
Computational Science and Its Applications - ICCSA 2007, International Conference, Kuala Lumpur, Malaysia, August 26-29, 2007
O. Gervasi and M. Gavrilova (Eds.), Springer-Verlag, LNCS 4705, Part I, pp. 603-621

Provably Secure Framework for Information Aggregation in Sensor Networks

(Full version)

Mark Manulis and Jörg Schwenk

Horst-Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{mark.manulis|joerg.schwenk}@nds.rub.de

Abstract. Information aggregation is an important operation in wireless sensor networks executed for the purpose of monitoring and reporting of the environmental data. Due to the performance constraints of sensor nodes the in-network form of the aggregation is especially attractive since it allows to save expensive resources during the frequent network queries. Easy accessibility of networks and nodes and almost no physical protection against corruptions arise high challenges on the security of the aggregation process. Especially, protection against attacks aiming to falsify the aggregated result is considered to be of prime importance.
In this paper we propose a novel security model for the aggregation process based on the well-established cryptographic techniques, focusing on the scenario with the single aggregator node. In order to show soundness and feasibility of our definitions we describe a *generic* practical approach that achieves security against node corruptions during the aggregation process in a provable cryptographic way based solely on the symmetric cryptographic primitives. To the best of our knowledge this is the first paper which aims to combine the paradigm of *provable security* in the cryptographic sense with the task of information aggregation in WSNs.

1 Introduction

Monitoring and reporting of the physically measured data to some querying device represented by a sink, base station, or mobile reader is surely one of the main goals for the deployment of wireless sensor networks (WSNs). This task is especially important in scenarios where high confidence on the integrity of the reported information becomes an indispensable part of the application security. For the purpose of performance optimization the reporting phase is frequently combined with the in-network processing resulting in the *in-network information aggregation*. The following two aggregation scenarios have been established throughout the literature. The *single* aggregator scenario is usually applied in cases where the aggregation process is independent of the network topology. In such scenarios the aggregator role is typically assigned to one of the nodes based on the execution of the underlying aggregator election protocol (e.g. [13]). Moreover, this role is usually temporary and changed (randomly) between nodes in order to distribute the increasing costs for the aggregation operation over the whole lifetime of the WSN. On the other hand, *hierarchical* aggregation scenarios usually assume certain aggregation topology computed in the underlying protocol (e.g. [8]). In such scenarios nodes located closest to the query device form the highest level of the aggregation hierarchy. Both scenarios are useful and may have own advantages in terms of efficiency, security, and administration. In this paper we focus on the single aggregator scenario and address one of the most important issues – the security.

Due to the loose infrastructure deployed sensor nodes may easily become subject of an adversarial attack. Surely, node corruptions in addition to active network attacks state one of the highest security threats. Especially, assuming that all nodes have equal physical properties, without any rigorous protection mechanisms such as tamper-resistance, designing secure information aggregation solutions becomes even more challenging. Surely, designing and adequate *formal* security model together with some *generic* (in the cryptographic sense) provably secure practical solution appears to be an interesting task.

1.1 Related Work

Currently, there exist only few scientific results in the area of secure information aggregation dealing with security of the aggregation process in the presence of corrupted nodes. In [5], Hu and Evans designed a protocol for hierarchical information aggregation between a set of nodes and the sink. This was the first solution based on symmetric cryptography that considered active attacks by compromised sensor nodes. Remarkable that previous solutions like [3, 4, 6, 8] addressed the scenario with honest communication participants only and are therefore not of much interest in the context of this work. The protocol in [5] requires an underlying protocol for the construction of the aggregation tree (e.g. [8]), as well as shared individual keys possibly pre-deployed in sensor nodes, and an authentic unidirectional communication channel between the sink and the involved nodes (e.g. [7, 10]). As for the corruption of nodes, we observe that if a node and a parent node in the aggregation tree are compromised then the adversary can significantly modify the aggregated result. For instance, corruption of the root node and its both children would allow complete falsification of the final aggregated value. Przydatek, Song, and Perrig [11] proposed a Secure Information Aggregation (SIA) framework for sensor networks which provides better resilience against malicious sensor nodes than the process in [5]. SIA addresses the single aggregator scenario. The main drawback of SIA (in the cryptographic sense) is its probabilistic security. In general the probability of the query device (sink) accepting some falsified aggregation result can be minimized by increasing the communication (and computation) between the sink and the aggregator node which constructs a Merkle hash commitment tree [9] for the received individual inputs and proves correctness of some parts of this tree during the subsequent interaction with the sink. Przydatek et al.'s protocol considers aggregation functions whose outputs can be approximated by the uniform sampling of the input values, e.g., computation of the MIN/MAX, AVERAGE, and MEDIAN. Recently, Chan, Perrig, and Song [2] described a solution for the hierarchical in-network aggregation which prevents active attacks aiming to modify and falsify the aggregation result. One of the core requirements in their approach builds the notion of *optimal security* – a property that no adversary can induce the sink to accept any aggregation result which is not already achievable by the so-called *direct data injection*, i.e., when the attacker reports biased data on behalf of nodes under its control. Their approach extends the previous one mainly by a fully distributed result-checking phase without relying on probabilistic security. Similar to [5] it requires the construction of the aggregation tree structure (e.g. [8]). Optimal security is achieved via interactive computation of the Merkle hash commitment trees. Chan et al. focus on the function SUM and show how to use it for the computation of AVERAGE and Φ-QUANTILE (the value at the Φn-th position in a sorted list).

1.2 Contributions and Organization

One general remark on the aforementioned solutions is that specified definitions of security are rather intuitive than formal. Therefore, proposing a formal security model to allow cryptographically sound security proofs seems to be an interesting extensional work in this research field. Another remark is that previous solutions cannot be really called generic since they have been designed with respect to some concrete aggregation functions (e.g. SUM) and then extended to deal with further functions. A more generic approach would be to give an abstract definition of the aggregation function and its security relevant properties. In the light of these remarks we contribute in this paper in two different ways: in Section 2 we design a formal security model for the in-network aggregation process and formalize for the first time the aggregation function in a very general way, and in Section 4 we design a concrete framework and prove its security according to our formal definitions using a well-known cryptographic proving technique after providing the required building blocks in Section 3. In terms of performance our framework relies on the primitives of symmetric cryptography without any costly public-key operations.

2 Formal Model for In-Network Aggregation in WSNs

In the following we propose an end-to-end model for in-network aggregation in WSNs. We focus on the single aggregator scenario, however, remark that the model is modular and, thus, extendable.

2.1 Communication Model and Participants

Protocol Participants By $\mathcal{S} := \{S_1, \ldots, S_n\}$, $n \in \mathbb{N}$ we denote the set of all sensor nodes in the network. We assume that all nodes have identical physical properties. By $\mathrm{A} \in \mathcal{S}$ we denote the role of the *aggregator*. This role is temporary and assigned by an underlying random aggregator election protocol. By R we denote a digital device which is assumed to be more powerful than any node in $\mathcal{S}$. R is usually represented by a sink, base station, or some mobile reader, and is assumed to be the party which is supposed to obtain the aggregated result.

Protocol Sessions and Participating Instances In order to distinguish between different protocol executions we use the notion of a *session*, that is every execution results in a new session identified by some value s, which is *unique* for each new session. In order to model entities $S_i \in \mathcal{S}$ resp. R as participants of some session s we consider that each entity may have an unlimited number of *instances* denoted S_i^s resp. R^s.

Secret Keys For the purpose of authentication we consider that every sensor node S_i resp. R is in possession of some secret key denoted k_i resp. k_R (notation k is used in case of generality). This key should be seen as a place holder, that is any k can in practice consist of several secret values, e.g., R may possess k_R composed of the secret key for broadcast authentication and secret keys shared between R and S_i. By 1^κ, $\kappa \in \mathbb{N}$ we denote the *security parameter* of the protocol, assuming that all security relevant parameters are polynomially related to 1^κ. In this work we apply *symmetric* secret keys aiming to avoid the use of the costly asymmetric cryptography.

2.2 Aggregation Function

In the following we abstractly define the aggregation function agg operating on real numbers in $\mathbb{R}$, however, extension to other domains is straightforward. We define agg with two inputs and consider its symmetry and associativity to deal with multiple inputs. We also allow one of the inputs to be empty (ε); then agg is the identity function. For the purpose of generality we require an additional auxiliary input space $\mathbb{A}$.

Definition 1 (Aggregation Function). *Let* $agg : \mathbb{R}\cup\{\varepsilon\}\times\mathbb{R}\cup\{\varepsilon\}\times\mathbb{A}\cup\{\varepsilon\} \rightarrow \mathbb{R}\cup\{\varepsilon\}$ *be an* aggregation function*,* ε *an* empty element*, and* $\mathbb{A}$ *some* auxiliary information space*. By convention* $agg(\varepsilon, \varepsilon; \mathtt{aux}) = \varepsilon$ *for any* $\mathtt{aux} \in \mathbb{A}$*. For any* $v_1, v_2, v_3 \in \mathbb{R}$ *and* specific $\mathtt{aux} \in \mathbb{A}$ *the aggregation function should satisfy:*

Identity: $agg(v_1, \varepsilon; \mathtt{aux}) = v_1$
Symmetry: $agg(v_1, v_2; \mathtt{aux}) = agg(v_2, v_1; \mathtt{aux})$
Associativity: $agg(agg(v_1, v_2; \mathtt{aux}), v_3; \mathtt{aux}) = agg(v_1, agg(v_2, v_3; \mathtt{aux}); \mathtt{aux})$

Let $\mathbf{v} := \{v_1, \ldots, v_n\}$*,* $n > 2$*. By* $agg(\mathbf{v}; \mathtt{aux})$ *we mean the output*

$$a_{i+1} := agg(a_i, v_{i+2}; \mathtt{aux})$$

after $i = 1, \ldots, n-2$ *iterations where* $a_1 := agg(v_1, v_2; \mathtt{aux})$*. For simplicity we will omit the indication of* $\mathtt{aux}$ *as one of the inputs.*

Many thinkable and widely used aggregation functions such as SUM, PRODUCT, additive/multiplicative AVERAGE, MIN/MAX, etc. satisfy the above properties of identity, symmetry and associativity. Note, that in case of AVERAGE $agg(\mathbf{v}; \mathtt{aux})$ can be computed correctly only if n is known during each iteration (as part of $\mathtt{aux}$); otherwise the associativity may not always hold. This emphasizes the need of $\mathbb{A}$ in the abstract definition of agg.

Additionally, we define boolean predicates B_v and B_a for the inputs and outputs of agg, respectively. These predicates will be used in our definition of security in order to handle node corruptions in a reasonable way.

Definition 2 (Aggregation Input/Output Predicates). *By* $B_v(v; \mathtt{aux}_v)$ *resp.* $B_a(a; \mathtt{aux}_a)$ *we denote a* boolean predicate for any input $v \in \mathbb{R}$ *resp.* output $a \in \mathbb{R}$ *of agg, where* $\mathtt{aux}_v$ *resp.* $\mathtt{aux}_a$ *is some* auxiliary information.

Let $\mathbf{v}$ *and* $\mathbf{a}$ *be sets/lists of possible inputs and outputs of agg. By* $B_v(\mathbf{v}; \mathtt{aux}_v)$ *we mean* $B_v(\mathbf{v}[1]; \mathtt{aux}_v) \wedge \ldots \wedge B_v(\mathbf{v}[n]; \mathtt{aux}_v)$*. By* $B_a(\mathbf{a}; \mathtt{aux}_a)$ *we mean* $B_a(\mathbf{a}[1]; \mathtt{aux}_a) \wedge \ldots \wedge B_a(\mathbf{a}[n]; \mathtt{aux}_a)$*.*

Additionally, we require that agg *with corresponding predicates* B_v *and* B_a *satisfies the following properties for any* $\mathbf{v}$*:*

Correctness: *if* $B_v(v; \mathtt{aux}_v) = true$ *for all* $v \in \mathbf{v}$ *then* $B_a(agg(\mathbf{v}); \mathtt{aux}_a) = true$*, and*
if $B_v(v; \mathtt{aux}_v) = false$ *for all* $v \in \mathbf{v}$ *then* $B_a(agg(\mathbf{v}); \mathtt{aux}_a) = false$
Consistency: *if* $B_a(agg(\mathbf{v}); \mathtt{aux}_a) = false$ *then there exists NO* $\mathbf{v}$ *with* $B_v(\mathbf{v}) = true$

For simplicity we will use $B_v(v)$ *instead of* $B_v(v; \mathtt{aux}_v)$ *and* $B_a(a)$ *instead of* $B_a(a; \mathtt{aux}_a)$.

Abstractly defined boolean predicates B_v and B_a can be used to restrict inputs and outputs of agg, e.g., for the SUM function one can require that every input $v \in \mathbb{R}$ is within a certain bound $[v_{\mathtt{min}}, v_{\mathtt{max}}]$ (whereby $v_{\mathtt{min}}$ and $v_{\mathtt{max}}$ become part of $\mathtt{aux}_v$ and $\mathtt{aux}_a$). Then, one would typically require that every output a should be in the interval between $nv_{\mathtt{min}}$ and $nv_{\mathtt{min}}$ where n (as part of $\mathtt{aux}_a$) is the maximal number of inputs to be aggregated (added) at once. It is easy to see that in this case the above defined properties of correctness and consistency are satisfied for any $\mathbf{v}$ of size n. At this point we remark that B_v and B_a play an essential role in our security definition and their correct specification for a particular aggregation function is necessary. Finally, one important observation is that we do NOT assume that if a *strict* subset of inputs does not satisfy B_v then the output does not satisfy B_a either. This opens doors for the actual attacks. For example, let agg be the SUM function, $[0, 10]$ the allowed interval for its inputs, and number 3 the total allowed number of inputs for a single aggregation. Consequently the output should lie in the interval $[0, 30]$. Assume, that two inputs are 5 and 8. Obviously, it is possible to choose the third input as 15 (which is not in the input interval) and still satisfy the output interval, namely $5 + 8 + 15 = 28 < 30$.

2.3 Definition of In-Network Aggregation and its Correctness

In the following we provide an abstract definition of the in-network aggregation protocol $\mathtt{InAP1}_{agg}$ focusing on the single aggregator scenario.

Definition 3 (In-Network Aggregation Protocol $\mathtt{InAP1}_{agg}$). *In session* s *of the* in-network aggregation protocol $\mathtt{InAP1}_{agg}$ *each sensor node instance* $S_i^s \in \mathcal{S}^s \setminus \mathrm{A}^s$*,* $|\mathcal{S}^s| = n_s$ *communicates to* A^s *own* aggregation input $v_i \in \mathbb{R}$. A^s *computes the* aggregation result $a^* := agg(v_1, \ldots, v_{n_s})$ *and communicates it to the instance* R^s *which* terminates *either with or without* accepting a^* *(possibly after additional interaction with the instances in* $\mathcal{S}^s$*).*

We say that an in-network aggregation protocol $\mathtt{InAP1}_{agg}$ is *correct* if R^s accepts $a^* := agg(v_1, \ldots, v_{n_s})$ where each v_i, $i \in [1, n_s]$ is the original input of $S_i^s \in \mathcal{S}^s$ such that $B_v(v_i) = true$.

2.4 Adversarial Model

As next we specify the adversarial setting for the in-network aggregation protocols. We assume that the whole communication is controlled by the probabilistic polynomial-time (PPT) adversary $\mathcal{I}$, i.e., $\mathcal{I}$ is able to replay, modify, delay, drop, and deliver protocol messages out of order as well as inject own messages. Note that since $\mathcal{I}$ can always refuse to deliver protocol messages our model does not address any denial-of-service attacks (similar to [2,11]) which aim to prevent R from obtaining any result at all. Note that in WSNs such attacks would normally be recognized and reveal the information about the presence of $\mathcal{I}$. Thus, our security model aims to recognize an occurring attack and prevent R from accepting a "biased" value.

Adversarial Queries The protocol execution in the presence of $\mathcal{I}$ is modeled based on *queries* to the instances of the participants. By $Send$ we denote a query type which allows $\mathcal{I}$ to send a message m to any instance involved in the protocol execution. This query can be used by $\mathcal{I}$ not only to inject own messages but also to replay or modify those sent by the instances, or simply forward them honestly without any changes.

$Send(S_i, S_j^s, m)$**:** $\mathcal{I}$ sends m to the node instance S_j^s (claiming that it is from some instance of S_i).
$Send(S_i, R^s, m)$**:** $\mathcal{I}$ sends m to the sink instance R^s (claiming that it is from some instance of S_i).
$Send(R, S_i^s, m)$**:** $\mathcal{I}$ sends m to the node instance S_i^s (claiming that it is from some instance of R).

In response to a $Send$ query $\mathcal{I}$ receives the outgoing message which the receiving instance would generate after processing m. This outgoing message might be an empty string in case that m is unexpected or a failure occurred. Further, there are two special $Send$ queries of the form $Send(\mathrm{S}_i^s, \texttt{'start'}, \mathcal{S}^s, \mathrm{A}^s, R^s)$ and $Send(R^s, \texttt{'start'}, \mathcal{S}^s, \mathrm{A}^s)$. The first query allows $\mathcal{I}$ to invoke the protocol execution at instance S_i^s. It contains instances of other participating sensor nodes in $\mathcal{S}^s \setminus \mathrm{S}_i^s$, reference on the aggregator instance A^s (note that $\mathrm{A}^s \in \mathcal{S}^s$), and the sink instance R^s. Similarly, the second query invokes the protocol execution at R^s. In response to these queries $\mathcal{I}$ receives the first message generated by the asked instance according to the protocol specification.

In addition to the active protocol participation of $\mathcal{I}$ we consider node corruptions. We do not assume any tamper-resistance property. Upon corrupting S_i the adversary obtains full control over S_i and reveals all information kept in S_i including its secret key k_i. We also allow corruptions of R. However, our security definition will exclude the meaningless case where R is corrupted during the session in which $\mathcal{I}$ wishes to falsify the aggregation result. Using queries $Corrupt(S_i)$ resp. $Corrupt(R)$ the adversary can obtain the secret key k_i resp. k_R.

Definition 4 (Strong Corruption Model). *For any PPT adversary $\mathcal{I}$ we say that $\mathcal{I}$* operates in the strong corruption model *if it is given access to the queries $Send$ and $Corrupt$.*

Protocol Execution in the Presence of $\mathcal{I}$ We assume that each secret key is generated during the initialization phase and is implicitly known to all instances of the entity. The protocol execution for one particular session s in the presence of the adversary $\mathcal{I}$ proceeds as follows. After $\mathcal{I}$ operating in the strong corruption model invokes the protocol execution for the session s all its queries are answered until R^s terminates either with or without having accepted the aggregation result. If R^s terminates without having accepted then a failure has been occurred (or an attack has been recognized). Consequently, the goal of $\mathcal{I}$ is to influence R^s accepting some "biased" aggregation result. Note that after the instance terminates it cannot be invoked for a new session so that a new instance (with new s) should be invoked instead.

2.5 Definition of (Optimal) Security

Prior to the definition of security of $\mathtt{InAP1}_{agg}$ we need to exclude the case where R is controlled by $\mathcal{I}$ in the attacked session. This is done by the following definition of freshness.

Definition 5 (Freshness of R). *Let R^s be the instance that has accepted in session s of $\mathtt{InAP1}_{agg}$, and $\mathcal{I}$ a PPT adversary operating in the strong corruption model. We say that R^s is* fresh *if no $Corrupt(R)$ queries have been previously asked.*

Note that whenever $\mathcal{I}$ corrupts R all its instances which have not terminated yet can be controlled by $\mathcal{I}$. As already mentioned any sensor node including the aggregator node can be corrupted. Hence, we can even consider the case where all sensor nodes are corrupted and R is the only honest party. There is one general remark on consideration of corrupted sensor nodes which equally holds for our protocol and the protocols in [2, 11]. Namely, corrupted nodes can report data which (strongly) deviates from the real one. Even, restricting input intervals would not provide security against such attacks. For example, if nodes measure temperature and reported values should lie between 5 and 100 degrees then any corrupted node can report 100 degrees although the real measured value is 30. It is clear that such attacks, denoted in [2] as *direct data injection*, cannot be prevented unless one completely disallows node corruptions in the adversarial setting, but then this setting would be weak. Nevertheless, damage of such attacks can be decreased if one ensures the overwhelming majority of uncorrupted nodes at any time during the network lifetime. Our security definition, similar to the informal definition of *optimal security* in [2], does not aim to detect such attacks. Instead, it focuses on the modification of the aggregated result with respect to the attacks in which corrupted nodes try to report semantically incorrect inputs to the aggregation function, that is inputs v_i with $B_v(v_i) = false$. Note that in the single aggregator scenario such *stealthy attacks* [11] are possible only if A is corrupted (unless A does not check predicates for all received original inputs). Obviously, verification of the input predicates by A is indispensable part of any secure protocol in the strong corruption model.

Definition 6 ((Optimal) Security of $\mathtt{InAP1}_{agg}$). *Let $\mathcal{I}$ be a PPT adversary operating in the strong corruption model and $\mathsf{Game}^{\mathtt{opt-sec}}_{\mathtt{InAP1}_{agg}}(\mathcal{I}, \kappa)$ denote the interaction where $\mathcal{I}$ interacts via queries with instances of parties in $\mathcal{S}$, $|\mathcal{S}| = n_s$ and instances of* R *participating in the in-network aggregation protocol $\mathtt{InAP1}_{agg}$ such that at the end of this interaction there is a* **fresh** *instance R^s which has accepted with the aggregation result a^*. Let $\mathcal{S}^s_{\mathtt{h}} \subseteq \mathcal{S}^s$ be a subset of sensor node instances for which no $Corrupt$ queries have been asked prior to the acceptance of a^* by R^s. Let $\mathbf{v}_{\mathtt{h}}$ be a set/list of size $n_{\mathtt{h}} \in [1, n_s]$ containing original inputs of instances in $\mathcal{S}^s_{\mathtt{h}}$ and $a_{\mathtt{h}} := agg(\mathbf{v}_{\mathtt{h}})$.*

We say that $\mathcal{I}$ wins *in* $\mathsf{Game}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\mathcal{I}, \kappa)$ *if there exists NO set/list* $\mathbf{v}_{\mathtt{c}}$ *of size* $n_{\mathtt{c}} = n_s - n_{\mathtt{h}}$ *with* $B_v(\mathbf{v}_{\mathtt{c}}) = true$ *such that* $a^* = agg(a_{\mathtt{h}}, \mathbf{v}_{\mathtt{c}})$*.*

Let $\mathsf{Succ}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\kappa)$ *denote the maximal probability (over all adversaries $\mathcal{I}$ running within time κ) of winning in the above interaction, i.e.*

$$\mathsf{Succ}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\kappa) = \max_{\mathcal{I}} \left| \Pr[\mathcal{I} \text{ wins in } \mathsf{Game}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\mathcal{I}, \kappa)] \right| .$$

We say that $\texttt{InAP1}_{agg}$ *is* (optimally) secure *if* $\mathsf{Succ}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\kappa)$ *is negligible.*

In the following we provide some explanations. The main goal is to require that $\mathcal{I}$ should be unable to exclude contributions (inputs) of uncorrupted nodes from the aggregated result. For example, if agg is SUM then the aggregated result should be at least the sum of inputs of uncorrupted nodes (denoted by $a_{\mathtt{h}}$). On the other hand, falsification of the input data by corrupted nodes is not considered as an attack as long as their aggregation result, say some $a_{\mathtt{c}}$, satisfies the boolean predicate B_a (in spirit of direct data injection), note that in this case the result $a^* := agg(a_{\mathtt{h}}, a_{\mathtt{c}})$ would also satisfy B_a due to the correctness of agg. Therefore, as an attack we consider the opposite case, i.e., where the receiver instance accepts a^* such that $a_{\mathtt{c}}$ does not satisfy B_a. The only general condition for $B_a(a_{\mathtt{c}}) = false$ is when all inputs $\mathbf{v}_{\mathtt{c}}$ with $a_{\mathtt{c}} := agg(\mathbf{v}_{\mathtt{c}})$ do not satisfy B_v, i.e., if $B_v(\mathbf{v}_{\mathtt{c}}) = false$ (due to the correctness of agg). Hence, in our definition we require that there exists NO set/list of possible inputs $\mathbf{v}_{\mathtt{c}}$ with $B_v(\mathbf{v}_{\mathtt{c}}) = true$, in addition to the inputs of uncorrupted users $\mathbf{v}_{\mathtt{h}}$ (that is why $n_{\mathtt{c}} = n_s - n_{\mathtt{h}}$ should hold).

3 Building Blocks

In this section we describe main building blocks of our framework distinguishing between cryptographic primitives and technical constructions.

3.1 Background on used Symmetric Cryptographic Primitives

Definition 7 (Hash Function). *By* $\mathtt{H} : \{0,1\}^{\kappa_1} \rightarrow \{0,1\}^{\kappa_2}$*,* $\kappa_1, \kappa_2 \in \mathbb{N}$ *we denote a* collision-resistant hash function*, i.e., for every PPT algorithm $\mathcal{I}$ the probability that $\mathcal{I}$ finds* x_1*,* $x_2 \in \{0,1\}^{\kappa_1}$ *such that* $x_1 \neq x_2$ *and* $\mathtt{H}(x_1) = \mathtt{H}(x_2)$ *is upper-bounded by a negligible fraction* $\epsilon_{\mathtt{H}}$*.*

Definition 8 (Message Authentication Code). *By* $\mathtt{MAC} := (\mathtt{Gen}, \mathtt{Sign}, \mathtt{Verify})$ *we define a* message authentication code *with the algorithms:*

$\mathtt{Gen}$*: A probabilistic algorithm that on input a security parameter* 1^κ *outputs a secret key* $k \in \{0,1\}^\kappa$*.*

$\mathtt{Sign}$*: A deterministic algorithm that on input k and a message* $m \in \{0,1\}^*$ *outputs a* MAC value μ*.*

$\mathtt{Verify}$*: A deterministic algorithm that on input* k*,* $m \in \{0,1\}^*$ *and a candidate MAC value μ outputs 1 or 0, indicating whether μ is valid or not.*

$\mathtt{MAC}$ *is* secure *if for any PPT algorithm $\mathcal{I}$ which obtains polynomially bounded number of MAC values on any messages of its choice the probability that $\mathcal{I}$ outputs* (m, μ) *such that* $\mathtt{Verify}(k, m, \mu) = 1$ *and no MAC value for m has been previously asked by $\mathcal{I}$ is upper-bounded by a negligible fraction* $\epsilon_{\mathtt{MAC}}$*.*

3.2 List Structures

In the following we define lists, their operations, and further notations used in the description of our protocol.

Definition 9 (Lists and Operations). *By convention we use bold letters to denote lists. For any* list $\mathbf{x}$ *by* $|\mathbf{x}|$ *we denote its* size. *By* $\mathbf{x}[i]$, $i \in [1, |\mathbf{x}|]$ *we denote the* element at its i-th position. *An empty element is denoted* ε. *Upon initialisation each list* $\mathbf{x}$ *is empty, that is* $\mathbf{x} = \{\varepsilon\}$ *and by convention* $|\mathbf{x}| = 0$. *Let* y *be an element to be inserted into* $\mathbf{x}$. *We use* $y.\mathbf{x}$ *to say that* y *is* pre-pended *to* $\mathbf{x}$ *resulting in* $\mathbf{x}[1] = y$. *Similarly, we use* $\mathbf{x}.y$ *to say that* y *is* appended *to* $\mathbf{x}$ *resulting in* $\mathbf{x}[|\mathbf{x}|] = y$.

Note that lists can be represented via binary trees and vice versa, e.g., using the *pre-oder* notation, that is the root vertex of the tree followed by its child vertices is recursively appended to the empty list. In general lists reduce implementation overhead compared to binary trees.

Definition 10 (Paths, Siblings, Co-Paths, Child and Parent Elements). *Let* $\mathbf{x} := \{x_1, \ldots, x_n\}$ *be a list and* $p \in [2, n]$ *any position within it. By*

$$\left\{\mathbf{x}\left[\frac{p}{2}\right], \ldots, \mathbf{x}\left[\frac{p}{2^{\lfloor \log_2 p \rfloor}} = 1\right]\right\}$$

we denote the path of $\mathbf{x}[p]$ *(note that* $\mathbf{x}[p]$ *does not belong to its path). If* p *is even then* $\mathbf{x}[p+1]$, *otherwise* $\mathbf{x}[p-1]$, *is said to be the* sibling *of* $\mathbf{x}[p]$. *By* co-path of $\mathbf{x}[p]$ *we denote the list consisting of its sibling and of siblings of all elements in the path of* $\mathbf{x}[p]$ *except for* $\mathbf{x}[1]$. *For any* $p \in [1, n]$ *by* $\mathbf{x}[2p]$ *and* $\mathbf{x}[2p+1]$ *we denote the* first *and* second child element *of* $\mathbf{x}[p]$, *respectively. Consequently,* $\mathbf{x}[p]$ *is the* parent element *of* $\mathbf{x}[2p]$ *and* $\mathbf{x}[2p+1]$.

4 Specification of the `InAP1`$_{agg}$ Framework

Our `InAP1`$_{agg}$ framework consists of the protocol which proceeds in three stages (`UP-FLOW`, `DOWNFLOW`, `VERIFICATION`) described in the following. For simplicity we assume that the received messages reveal unique identities of their senders. Since we describe one particular protocol execution we use entities and not their instances.

4.1 The `UPFLOW` Stage

In the `UPFLOW` stage every S_i after having received the authenticated sink's query containing a *random nonce* r and the expected number of nodes n_s sends own *initial data value* v_i to A. At the same time A initializes the *node counter* denoted c, the *timer* t, the *list of sensor node's identities* $\mathbf{id}$ and the *list of sensor nodes' initial data values* $\mathbf{v}$ and assigns own identity id_{A} and own data value v_{A} to their first positions, respectively. The formal specification of the aggregator's calculations is given in Figure 1. Whenever A receives a new message it extends both lists by corresponding identities and data values. This extension is performed until A obtains messages from all $n_s - 1$ nodes; otherwise it sends a negative acknowledgement `ERR` to R indicating that a failure has occurred.

```
On input $r$, $n_s$, and $k_{\mathrm{A}}$ the aggregator proceeds as follows:
    initialize $\mathbf{id}$, $\mathbf{v}$, $\mathbf{a}$, $\mathbf{h}$, timer $t$, $c := 1$, compute $\mathbf{id} := \mathbf{id}.id_{\mathrm{A}}$, $\mathbf{v} := \mathbf{v}.v_{\mathrm{A}}$
    while $c \leq n_s$ or $t$ is not expired do
        if new $v_i$ received and $B_v(v_i) = true$ then $c := c + 1$, $\mathbf{id} := \mathbf{id}.id_i$, $\mathbf{v} := \mathbf{v}.v_i$
    if $c \neq n_s$ and $t$ is expired then $\mu_{\mathrm{A}} := \mathtt{MAC.Sign}(k_{\mathrm{A}}, (r, \mathtt{ERR}))$, send $(\mathtt{ERR}, \mu_{\mathrm{A}})$ to $R$
    else $(\mathbf{a}, \mathbf{h}) := \mathtt{Commit}(r, \mathbf{v}, \mathbf{a}, \mathbf{h})$, send $(r, \mathbf{a}[1], \mathbf{h}[1])$ to $R$
```

Fig. 1. UPFLOW stage specification for the aggregator A

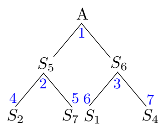
Note that under the assumption that messages arrive in the order which is correlated with their "physical" distance to A the identities and initial data values of "closer" nodes would appear in the beginning of both lists. This will be of advantage wrt. the communication efficiency in the DOWNFLOW stage.

Starting with nodes whose identities and initial data values are assigned to the later positions in both lists A computes the *list of intermediate aggregation values* $\mathbf{a}$ and the *list of intermediate commitment values* $\mathbf{h}$ using the auxiliary Commit function specified in Figure 2.We remark that the same function will be used by other nodes in the DOWNFLOW stage. Let $\mathbf{id}[i]$ be a sensor node's identity. Then, $\mathbf{a}[i]$ is the output of

```
$\mathtt{Commit}(r, \mathbf{v}, \mathbf{a}, \mathbf{h})$ :
    $c_v := |\mathbf{v}|$, $n := |\mathbf{v}|$
    while $c_v \geq 1$ do
        if $2c_v \geq n$ then $\mathbf{a} := \mathbf{v}[c_v].\mathbf{a}$, $\mathbf{h} := \mathtt{H}(r, \mathbf{a}[1]).\mathbf{h}$
        else if $2c_v \leq n$ and $2c_v + 1 > n$ then
            $\mathbf{a} := agg(\mathbf{v}[c_v], \mathbf{a}[c_v]).\mathbf{a}$, $\mathbf{h} := \mathtt{H}(r, \mathbf{a}[1], \mathbf{h}[c_v]).\mathbf{h}$
        else $\mathbf{a} := agg(\mathbf{v}[c_v], \mathbf{a}[c_v], \mathbf{a}[c_v + 1]).\mathbf{a}$, $\mathbf{h} := \mathtt{H}(r, \mathbf{a}[1], \mathbf{h}[c_v], \mathbf{h}[c_v + 1]).\mathbf{h}$
        $c_v := c_v - 1$
    return $(\mathbf{a}, \mathbf{h})$
```

Fig. 2. Function Commit

the aggregation function agg on inputs $\mathbf{v}[i]$ and every data value $\mathbf{v}[j]$ of node $\mathbf{id}[j]$ which has $\mathbf{id}[i]$ in its path. Further, $\mathbf{h}[i]$ is a hash commitment computed on r, $\mathbf{a}[i]$, $\mathbf{h}[2i]$, and $\mathbf{h}[2i + 1]$. Note $\mathbf{h}[2i]$ and $\mathbf{h}[2i + 1]$ are included into the hash commitment only if these values really exist; otherwise missing hash commitments are treated as empty elements. The construction of $\mathbf{a}$ ensures that $\mathbf{a}[1]$ gives the aggregation result $agg(\mathbf{v}[1], \ldots, \mathbf{v}[n_s])$. Similarly, the construction of $\mathbf{h}$ ensures that $\mathbf{h}[1]$ is the final hash commitment value which depends on all intermediate commitments. At the end of the UPFLOW stage A forwards $(r, \mathbf{a}[1], \mathbf{h}[1])$ to R which verifies that r is correct and checks whether $B_a(\mathbf{a}[1]) = true$. R terminates if ERR is received or if $B_a(\mathbf{a}[1]) = false$. Otherwise, R broadcasts authenticated (r, a^*, h^*) with $a^* = \mathbf{a}[1]$ and $h^* = \mathbf{h}[1]$ to all nodes in the network initiating the DOWNFLOW stage. Figure 3 shows an example of computed lists for the scenario with seven sensor nodes $\mathcal{S} := \{S_1, \ldots, S_7\}$ where S_3 plays the role of A.



		1	2	3	4	5	6	7
$\mathbf{id}$	=	id_3	id_5	id_6	id_2	id_7	id_1	id_4
$\mathbf{v}$	=	v_3	v_5	v_6	v_2	v_7	v_1	v_4
$\mathbf{a}$	=	$agg(v_3, \mathbf{a}[2], \mathbf{a}[3])$	$agg(v_5, v_2, v_7)$	$agg(v_6, v_1, v_4)$	v_2	v_7	v_1	v_4
$\mathbf{h}$	=	$\mathtt{H}(r, \mathbf{a}[1], \mathbf{h}[2], \mathbf{h}[3])$	$\mathtt{H}(r, \mathbf{a}[2], \mathbf{h}[4], \mathbf{h}[5])$	$\mathtt{H}(r, \mathbf{a}[3], \mathbf{h}[6], \mathbf{h}[7])$	$\mathtt{H}(r, v_2)$	$\mathtt{H}(r, v_7)$	$\mathtt{H}(r, v_1)$	$\mathtt{H}(r, v_4)$

Fig. 3. List Structures in the UPFLOW Stage for $\mathcal{S} := \{S_1, \ldots, S_7\}$, $\mathrm{A} = S_3$. Left side: Visualisation of node assignments in a binary tree structure. Right side: Reference lists $\mathbf{id}$, $\mathbf{v}$, $\mathbf{a}$, and $\mathbf{h}$ computed by A. Some exemplary notations: sibling of S_7 ($\mathbf{id}[5]$) is S_2 ($\mathbf{id}[4]$); path of S_7 consists of S_5 and A ($\mathbf{id}[2]$ and $\mathbf{id}[1]$); co-path of S_7 consists of S_2 and S_6 ($\mathbf{id}[4]$ and $\mathbf{id}[3]$); first child of S_6 is S_1 ($\mathbf{id}[6]$); second child of S_6 is S_4 ($\mathbf{id}[7]$); parent of S_2 is S_5.

4.2 The DOWNFLOW Stage

The DOWNFLOW stage of our protocol is a distributed process requiring communication between the sensor nodes. Its goal is to provide every node with sufficient information which will be used during the VERIFICATION stage to recompute the intermediate aggregation values and hash commitments along the path (in spirit of [2]). However, (unlike the tree structure in [2]) all lists computed during the UPFLOW stage are first known to A, but not to the other nodes. Therefore, A is the first to start the dissemination process which is specified in Figure 4. First, (honest) A must check that the

```
On input $r$, $n_s$, $a^*$, $h^*$, $\mathbf{id}$, $\mathbf{v}$, $\mathbf{a}$, $\mathbf{h}$ the aggregator proceeds as follows:
    $\mathtt{acc} := true$
    if $n_s \neq c$ or $a^* \neq \mathbf{a}[1]$ or $h^* \neq \mathbf{h}[1]$ then $\mathtt{acc} := false$
    else if $n_s \geq 2$ then
        initialize $\mathbf{id}_\mathtt{L}$, $\mathbf{v}_\mathtt{L}$, $\mathbf{id}_\mathtt{R}$, $\mathbf{v}_\mathtt{R}$, $\mathbf{a}^{\mathtt{co}}_\mathtt{L}$, $\mathbf{a}^{\mathtt{co}}_\mathtt{R}$, $\mathbf{h}^{\mathtt{co}}_\mathtt{L}$, $\mathbf{h}^{\mathtt{co}}_\mathtt{R}$, $\mathbf{v}_\mathtt{P}$
        $p_\mathtt{L} := 2$
        if $n_s \geq 3$ then $p_\mathtt{R} := 3$
        if $n_s \geq 4$ then $(\mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, \mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R}) := \mathtt{SplitIdV}(\mathbf{id}, \mathbf{v}, \mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, \mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R})$
        $(\mathbf{a}^{\mathtt{co}}_\mathtt{L}, \mathbf{h}^{\mathtt{co}}_\mathtt{L}, \mathbf{a}^{\mathtt{co}}_\mathtt{R}, \mathbf{h}^{\mathtt{co}}_\mathtt{R}) := \mathtt{SplitAH}(n, \mathbf{a}, \mathbf{h}, \mathbf{a}^{\mathtt{co}}_\mathtt{L}, \mathbf{h}^{\mathtt{co}}_\mathtt{L}, \mathbf{a}^{\mathtt{co}}_\mathtt{R}, \mathbf{h}^{\mathtt{co}}_\mathtt{R})$
        $\mathbf{v}_\mathtt{P} := \mathbf{v}_\mathtt{P}.v_\mathrm{A}$
        send $(\mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, p_\mathtt{L}, \mathbf{v}_\mathtt{P}, \mathbf{a}^{\mathtt{co}}_\mathtt{L}, \mathbf{h}^{\mathtt{co}}_\mathtt{L})$ to $S_{\mathbf{id}[2]}$
        if $n_s \geq 3$ then send $(\mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R}, p_\mathtt{R}, \mathbf{v}_\mathtt{P}, \mathbf{a}^{\mathtt{co}}_\mathtt{R}, \mathbf{h}^{\mathtt{co}}_\mathtt{R})$ to $S_{\mathbf{id}[3]}$
```

Fig. 4. DOWNFLOW stage specification for the aggregator A

message received from R contains the same values that have been sent by A in the UPFLOW stage; otherwise the verification process would fail. Therefore, if A notices the mismatch then it sets its boolean variable $\mathtt{acc} := false$ and turns immediately into the VERIFICATION stage where it will send its negative acknowledgement to R. If no mismatch is found then A whose identity is assigned to $\mathbf{id}[1]$ sends one message to each of its child nodes $\mathbf{id}[2]$ and $\mathbf{id}[3]$. Note that via $n_s \geq 2$ it can easily check whether any child nodes exist. The message addressed to $\mathbf{id}[2]$ ($\mathbf{id}[3]$) contains: (1) a list of identities $\mathbf{id}_\mathtt{L}$ ($\mathbf{id}_\mathtt{R}$) which consists of elements from $\mathbf{id}$ which have $\mathbf{id}[2]$ ($\mathbf{id}[3]$) in their paths, (2) a list of initial data values $\mathbf{v}_{\mathrm{L}}$ ($\mathbf{v}_{\mathrm{R}}$) which consists of elements from $\mathbf{v}$ which have $\mathbf{v}[2]$ ($\mathbf{v}[3]$) in their paths, (3) position value $p = 2$ ($p = 3$), (4) a list of initial data values $\mathbf{v}_{\mathrm{P}}$ consisting of v_{A}, (5) a list of intermediate aggregation values $\mathbf{a}^{\mathrm{co}}_{\mathrm{L}}$ ($\mathbf{a}^{\mathrm{co}}_{\mathrm{R}}$) which contains $\mathbf{a}[3]$ ($\mathbf{a}[2]$), and (6) a list of intermediate hash commitments $\mathbf{h}^{\mathrm{co}}_{\mathrm{L}}$ ($\mathbf{h}^{\mathrm{co}}_{\mathrm{R}}$) which contains $\mathbf{h}[6]$ and $\mathbf{h}[7]$ ($\mathbf{h}[4]$ and $\mathbf{h}[5]$), if such values exist.

The auxiliary function SplitIdV (Figure 5) is used by A to build the corresponding sets $(\mathbf{id}_{\mathrm{L}}, \mathbf{v}_{\mathrm{L}})$ resp. $(\mathbf{id}_{\mathrm{R}}, \mathbf{v}_{\mathrm{R}})$. Of course, SplitIdV is executed only if $n_s \geq 4$, that is

```
SplitIdV($\mathbf{id}, \mathbf{v}, \mathbf{id}_{\mathrm{L}}, \mathbf{v}_{\mathrm{L}}, \mathbf{id}_{\mathrm{R}}, \mathbf{v}_{\mathrm{R}}$) :
    $x := 4, y := 0$
    while $(x + y) \leq |\mathbf{id}|$ do
        if $y < \frac{x}{2}$ then $\mathbf{id}_{\mathrm{L}} := \mathbf{id}_{\mathrm{L}}.\mathbf{id}[x + y]$, $\mathbf{v}_{\mathrm{L}} := \mathbf{v}_{\mathrm{L}}.\mathbf{v}[x + y]$
        else $\mathbf{id}_{\mathrm{R}} := \mathbf{id}_{\mathrm{R}}.\mathbf{id}[x + y]$, $\mathbf{v}_{\mathrm{R}} := \mathbf{v}_{\mathrm{R}}.\mathbf{v}[x + y]$
        if $y < x - 1$ then $y := y + 1$
        else $x := 2x, y := 0$
    return $(\mathbf{id}_{\mathrm{L}}, \mathbf{v}_{\mathrm{L}}, \mathbf{id}_{\mathrm{R}}, \mathbf{v}_{\mathrm{R}})$
```

Fig. 5. Function SplitIdV

if $\mathbf{id}[2]$ and $\mathbf{id}[3]$ have in turn further child nodes. SplitIdV function splits the initial sets $\mathbf{id}$ resp. $\mathbf{v}$ into the sublists $\mathbf{id}_{\mathrm{L}}$ and $\mathbf{id}_{\mathrm{R}}$ resp. $\mathbf{v}_{\mathrm{L}}$ and $\mathbf{v}_{\mathrm{R}}$ containing identities resp. original data values of sensor nodes that have $\mathbf{id}[2]$ and $\mathbf{id}[3]$ resp. $\mathbf{v}[2]$ and $\mathbf{v}[3]$ in their paths. The idea behind the SplitIdV function is to move along the initial $\mathbf{id}$ resp. $\mathbf{v}$ lists and insert their elements into either $\mathbf{id}_{\mathrm{L}}$ or $\mathbf{id}_{\mathrm{R}}$ resp. $\mathbf{v}_{\mathrm{L}}$ or $\mathbf{v}_{\mathrm{R}}$ lists based on the condition $y < \frac{x}{2}$, which identifies whether $\mathbf{id}[x + y]$ has $\mathbf{id}[2]$ or $\mathbf{id}[3]$ in its path.

Another auxiliary function called SplitAH (Figure 6) is used by A to compute lists of intermediate aggregation values $\mathbf{a}^{\mathrm{co}}_{\mathrm{L}}$ resp. $\mathbf{a}^{\mathrm{co}}_{\mathrm{R}}$ and hash commitments $\mathbf{h}^{\mathrm{co}}_{\mathrm{L}}$ resp. $\mathbf{h}^{\mathrm{co}}_{\mathrm{R}}$ in the co-paths of its first and second child nodes.

```
SplitAH($c, \mathbf{a}, \mathbf{h}, \mathbf{a}^{\mathrm{co}}_{\mathrm{L}}, \mathbf{h}^{\mathrm{co}}_{\mathrm{L}}, \mathbf{a}^{\mathrm{co}}_{\mathrm{R}}, \mathbf{h}^{\mathrm{co}}_{\mathrm{R}}$) :
    if $c \geq 3$ then $\mathbf{a}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{a}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{a}[2]$, $\mathbf{a}^{\mathrm{co}}_{\mathrm{L}} := \mathbf{a}^{\mathrm{co}}_{\mathrm{L}}.\mathbf{a}[3]$
    else $\mathbf{a}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{a}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{a}[2]$, $\mathbf{a}^{\mathrm{co}}_{\mathrm{L}} := \mathbf{a}^{\mathrm{co}}_{\mathrm{L}}.\varepsilon$
    if $c \geq 7$ then
        $\mathbf{h}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{h}[4].\mathbf{h}[5]$, $\mathbf{h}^{\mathrm{co}}_{\mathrm{L}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{L}}.\mathbf{h}[6].\mathbf{h}[7]$
    else if $c \geq 6$ then
        $\mathbf{h}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{h}[4].\mathbf{h}[5]$, $\mathbf{h}^{\mathrm{co}}_{\mathrm{L}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{L}}.\mathbf{h}[6]$
    else if $c \geq 5$ then $\mathbf{h}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{h}[4].\mathbf{h}[5]$
    else if $c \geq 4$ then $\mathbf{h}^{\mathrm{co}}_{\mathrm{R}} := \mathbf{h}^{\mathrm{co}}_{\mathrm{R}}.\mathbf{h}[4]$
    return $(\mathbf{a}^{\mathrm{co}}_{\mathrm{L}}, \mathbf{h}^{\mathrm{co}}_{\mathrm{L}}, \mathbf{a}^{\mathrm{co}}_{\mathrm{R}}, \mathbf{h}^{\mathrm{co}}_{\mathrm{R}})$
```

Fig. 6. Function SplitAH

For example, according to Figure 3 the aggregator $\mathrm{A} = S_3$ sends to its first child node S_5 the following contents: $\mathbf{id}_{\mathrm{L}} := \{id_2, id_7\}$, $\mathbf{v}_{\mathrm{L}} := \{v_2, v_7\}$, $p = 2$, $\mathbf{v}_{\mathrm{P}} := \{v_3\}$, $\mathbf{a}^{\mathrm{co}}_{\mathrm{L}} := \{agg(v_6, v_1, v_4)\}$, and $\mathbf{h}^{\mathrm{co}}_{\mathrm{L}} := \{\mathtt{H}(r, v_1), \mathtt{H}(r, v_4)\}$.

Calculations performed by any other S_i during the DOWNFLOW stage (Figure 7) are similar to that of A, except that S_i has to wait for the message containing $(\mathbf{id}, \mathbf{v}, p, \mathbf{v}_\mathtt{P}, \mathbf{a}^{\mathrm{co}}, \mathbf{h}^{\mathrm{co}})$. Before, S_i performs computations of the DOWNFLOW stage it pre-pends own

```
On input $r$, $n_s$, $a^*$, $h^*$, $\mathbf{id}$, $\mathbf{v}$, $p$, $\mathbf{v}_\mathtt{P}$, $\mathbf{a}^{\mathrm{co}}$, $\mathbf{h}^{\mathrm{co}}$ every sensor node $S_i$ proceeds as follows:
    $\mathbf{id} := id_i.\mathbf{id}$, $\mathbf{v} := v_i.\mathbf{v}$, $c := |\mathbf{id}|$, $c_p := |\mathbf{v}_\mathtt{P}|$, $\mathtt{acc} := true$
    if $c \neq |\mathbf{v}|$ or $c_p \neq |\mathbf{a}^{\mathrm{co}}|$ or $c_p \neq \lfloor \log_2 p \rfloor$ or $B_v(\mathbf{v}) = false$ or $B_v(\mathbf{v}_\mathtt{P}) = false$
      or $B_a(a^*) = false$ or $B_a(\mathbf{a}) = false$ then $\mathtt{acc} := false$
    else
        initialize $\mathbf{a}$, $\mathbf{h}$
        $(\mathbf{a}, \mathbf{h}) := \mathtt{Commit}(r, \mathbf{v}, \mathbf{a}, \mathbf{h})$
        if $c \geq 2$ then
            initialize $\mathbf{id}_\mathtt{L}$, $\mathbf{v}_\mathtt{L}$, $\mathbf{id}_\mathtt{R}$, $\mathbf{v}_\mathtt{R}$, $\mathbf{a}^{\mathrm{co}}_\mathtt{L}$, $\mathbf{a}^{\mathrm{co}}_\mathtt{R}$, $\mathbf{h}^{\mathrm{co}}_\mathtt{L}$, $\mathbf{h}^{\mathrm{co}}_\mathtt{R}$, $\mathbf{v}'_\mathtt{P}$
            $\mathbf{a}^{\mathrm{co}}_\mathtt{L} := \mathbf{a}^{\mathrm{co}}$, $\mathbf{a}^{\mathrm{co}}_\mathtt{R} := \mathbf{a}^{\mathrm{co}}$, $\mathbf{h}^{\mathrm{co}}_\mathtt{L} := \mathbf{h}^{\mathrm{co}}$, $\mathbf{h}^{\mathrm{co}}_\mathtt{R} := \mathbf{h}^{\mathrm{co}}$, $p_\mathtt{L} := 2p$
        if $c \geq 3$ then $p_\mathtt{R} := 2p + 1$
        if $c \geq 4$ then $(\mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, \mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R}) := \mathtt{SplitIdV}(\mathbf{id}, \mathbf{v}, \mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, \mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R})$
        $(\mathbf{a}^{\mathrm{co}}_\mathtt{L}, \mathbf{h}^{\mathrm{co}}_\mathtt{L}, \mathbf{a}^{\mathrm{co}}_\mathtt{R}, \mathbf{h}^{\mathrm{co}}_\mathtt{R}) := \mathtt{SplitAH}(c, \mathbf{a}, \mathbf{h}, \mathbf{a}^{\mathrm{co}}_\mathtt{L}, \mathbf{h}^{\mathrm{co}}_\mathtt{L}, \mathbf{a}^{\mathrm{co}}_\mathtt{R}, \mathbf{h}^{\mathrm{co}}_\mathtt{R})$, $\mathbf{v}'_\mathtt{P} := \mathbf{v}_\mathtt{P}.v_i$
        send $(\mathbf{id}_\mathtt{L}, \mathbf{v}_\mathtt{L}, p_\mathtt{L}, \mathbf{v}'_\mathtt{P}, \mathbf{a}^{\mathrm{co}}_\mathtt{L}, \mathbf{h}^{\mathrm{co}}_\mathtt{L})$ to $S_{\mathbf{id}[2]}$
        if $c \geq 3$ then send $(\mathbf{id}_\mathtt{R}, \mathbf{v}_\mathtt{R}, p_\mathtt{R}, \mathbf{v}'_\mathtt{P}, \mathbf{a}^{\mathrm{co}}_\mathtt{R}, \mathbf{h}^{\mathrm{co}}_\mathtt{R})$ to $S_{\mathbf{id}[3]}$
```

Fig. 7. DOWNFLOW stage specification for the sensor node S_i

identity id_i and data value v_i to $\mathbf{id}$ and $\mathbf{v}$, respectively. Note that this results in $\mathbf{id}[1] = id_i$ and $\mathbf{v}[1] = v_i$. Before S_i proceeds with the computation it checks whether the received parameters are well-formed. Note that the equality $c_\mathtt{P} = \lfloor \log_2 p \rfloor$ ensures the consistency between the node's position p and the number of nodes in its path. If any of these verifications fails then S_i sets its boolean variable acc to false and turns directly into the VERIFICATION stage. Note that in this case child nodes of S_i will not receive any messages. Thus, a negative acknowledgement will be sent to A and then forwarded to R. Otherwise, S_i (with $\mathbf{id}[1]$) invokes the Commit function which outputs intermediate aggregation values $\mathbf{a}$ and hash commitments $\mathbf{h}$. Then S_i checks whether there are any further child nodes via the condition $c \geq 2$. If so, S_i splits $\mathbf{id}$ resp. $\mathbf{v}$ into $\mathbf{id}_\mathtt{L}$ and $\mathbf{id}_\mathtt{R}$ resp. $\mathbf{v}_\mathtt{L}$ and $\mathbf{v}_\mathtt{R}$ using the SplitIdV function, updates $\mathbf{a}^{\mathrm{co}}_\mathtt{L}$ and $\mathbf{a}^{\mathrm{co}}_\mathtt{R}$ resp. $\mathbf{h}^{\mathrm{co}}_\mathtt{L}$ and $\mathbf{h}^{\mathrm{co}}_\mathtt{R}$ based on the previously computed lists $\mathbf{a}$ and $\mathbf{h}$ using the SplitAH function, extends $\mathbf{v}'_\mathtt{P} := \mathbf{v}_\mathtt{P}.v_i$ (note that the received $\mathbf{v}_\mathtt{P}$ remains unchanged since it will be needed in the VERIFICATION stage), and sends appropriate messages to its existing child node(s).

According to the example in Figure 3 node S_5 sends to S_2 the following contents: $\mathbf{id}_\mathtt{L} := \{\varepsilon\}$, $\mathbf{v}_\mathtt{L} := \{\varepsilon\}$, $p = 4$, $\mathbf{v}_\mathtt{P} := \{v_3, v_5\}$, $\mathbf{a}^{\mathrm{co}}_\mathtt{L} := \{agg(v_6, v_1, v_4), v_7\}$, and $\mathbf{h}^{\mathrm{co}}_\mathtt{L} := \{\mathtt{H}(r, v_1), \mathtt{H}(r, v_4)\}$; and to S_7: $\mathbf{id}_\mathtt{R} := \{\varepsilon\}$, $\mathbf{v}_\mathtt{R} := \{\varepsilon\}$, $p = 5$, $\mathbf{v}_\mathtt{P} := \{v_3, v_5\}$, $\mathbf{a}^{\mathrm{co}}_\mathtt{R} := \{agg(v_6, v_1, v_4), v_2\}$, and $\mathbf{h}^{\mathrm{co}}_\mathtt{R} := \{\mathtt{H}(r, v_1), \mathtt{H}(r, v_4)\}$. The dissemination process of the DOWNFLOW stage is executed until every of $n_s - 1$ nodes obtains the required information and turns into the VERIFICATION stage.

4.3 The VERIFICATION stage

In the VERIFICATION stage every S_i recomputes a^* and h^* and checks whether these values match those received from R. Every S_i is in possession of the own intermediate aggregation value $\mathbf{a}[1]$ and its corresponding hash commitment $\mathbf{h}[1]$. Furthermore, every S_i (and A) knows own data value v_i (and v_A), data values in its path given by $\mathbf{v}_\mathtt{P}$, intermediate aggregation values in its co-path given by $\mathbf{a}^\mathrm{co}$, hash commitments in its co-path given by $\mathbf{h}^\mathrm{co}$, as well as the aggregation result a^* and hash commitment h^* from the broadcast message of R. Additionally, every S_i knows own position p which it can use to recognize whether it is the first (p is even) or the second (p is odd) child node. Beside that every S_i maintains a boolean variable `acc` indicating whether the node will confirm the obtained final values or not. Note that during the `DOWNFLOW` stage `acc` could possibly be changed to $false$. Figure 8 describes calculations of S_i. According to the construction of **id** by A in the `UPFLOW` stage for every node $\mathbf{id}[p]$

```
On input $r$, $n_s$, $a^*$, $h^*$, $p$, $\mathbf{a}$, $\mathbf{h}$, $\mathbf{v}_\mathtt{P}$, $\mathbf{a}^\mathrm{co}$, $\mathbf{h}^\mathrm{co}$, acc, $k_i$ every sensor node $S_i$ proceeds as follows:
    if acc $= true$ then
        $a := \mathbf{a}[1], h := \mathbf{h}[1], c_p := |\mathbf{v}_\mathtt{P}|, c_h := |\mathbf{h}^\mathrm{co}|$
        while $c_p \geq 1$ do
            if $p$ even then
                if $p + 1 \leq n_s$ then
                    $a := agg(\mathbf{v}_\mathtt{P}[c_p], a, \mathbf{a}^\mathrm{co}[c_p])$
                    if $2(p+1)+1 \leq n_s$ then
                        $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p], \mathbf{h}^\mathrm{co}[c_h - 1], \mathbf{h}^\mathrm{co}[c_h]), c_h := c_h - 2$
                    else if $2(p+1) \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p], \mathbf{h}^\mathrm{co}[c_h - 1]), c_h := c_h - 1$
                    else $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p])$
                    $h := \mathtt{H}(r, a, h, \bar{h})$
                else $a := agg(\mathbf{v}_\mathtt{P}[c_p], a), h := \mathtt{H}(r, a, h)$
            else
                $a := agg(\mathbf{v}_\mathtt{P}[c_p], \mathbf{a}^\mathrm{co}[c_p], a)$
                if $2(p-1)+1 \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p], \mathbf{h}^\mathrm{co}[c_h - 1], \mathbf{h}^\mathrm{co}[c_h]), c_h := c_h - 2$
                else if $2(p-1) \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p], \mathbf{h}^\mathrm{co}[c_h - 1]), c_h := c_h - 1$
                else $\bar{h} := \mathtt{H}(r, \mathbf{a}^\mathrm{co}[c_p])$
                $h := \mathtt{H}(r, a, \bar{h}, h)$
            $c_p := c_p - 1, p := \lfloor \frac{p}{2} \rfloor$
        if $a \neq a^*$ or $h \neq h^*$ then acc $= false$
    if acc $= false$ then $\mu_i := \mathtt{MAC.Sign}(k_i, (r, \mathtt{ERR}))$, send $(\mathtt{ERR}, \mu_i)$ to A
    else $\mu_i := \mathtt{MAC.Sign}(k_i, (r, \mathtt{OK}))$, send $(\mathtt{OK}, \mu_i)$ to A
```

Fig. 8. `VERIFICATION` stage specification for the sensor node S_i

with odd position $p > 1$ there exists a sibling node $\mathbf{id}[p-1]$. However, if p is even then the additional verification via $p + 1 \leq n_s$ becomes necessary to ensure that $\mathbf{id}[p+1]$ exists. Note that iterative division $\lfloor p/2 \rfloor$ can further be used to find out whether $\mathbf{id}[p]$ is the first or the second child node of $\mathbf{id}[\lfloor p/2 \rfloor]$. In case that `acc` is already set to $false$ no further checks are necessary and S_i replies to A with a negative acknowledgement in form of an error message `ERR` which it authenticates using a MAC value μ_i computed with k_i which is shared with R. Otherwise, S_i recomputes the aggregation result a and the hash commitment value h and compares them to a^* and h^* received from R. To perform these computations S_i sets initially $a := \mathbf{a}[1]$ and $h := \mathbf{h}[1]$. Note that $\mathbf{a}$ and $\mathbf{h}$ have been computed by S_i via the `Commit` function during the `DOWNFLOW` stage. In each iteration S_i updates a resp. h to the aggregation value resp. hash commitment corresponding to the next position in its path using the auxiliary aggregation value $\mathbf{a}^{\text{co}}[c_p]$ and hash commitment $\bar{h}$ from its co-path. $\bar{h}$ is computed by S_i from the received commitments and $\mathbf{a}^{\text{co}}[c_p]$, whereas $\mathbf{a}^{\text{co}}[c_p]$ is taken directly from the parent node's message. It is easy to check that after the final iteration a resp. h should (ideally) match a^* resp. h^*. If these values match then S_i sends a positive acknowledgement `OK` to A together with the MAC value μ_i.

Figure 9 specifies operations of A. Note that for A it is not necessary to recompute

```
On input $r$, $n_s$, acc, $k_A$ aggregator A proceeds as follows:
    if acc $= true$ and $n_s = 1$ then $\mu_A :=$ MAC.Sign($k_A, (r, $OK$)$), send (OK, $\mu_A$) to $R$
    else if acc $= true$ and $n_s > 1$ then
        $\mu :=$ MAC.Sign($k_A, (r, $OK$)$), $c := 1$, nxt $= true$, initialize timer $t$
        while $c < n_s$ and nxt $= true$ and $t$ is not expired do
            receive new $(m, \mu_i)$
            if $m =$ OK then $\mu := \mu \oplus \mu_i$, $c := c + 1$
            else if $m =$ ERR then send (ERR, $\mu_i$) to $R$, nxt $= false$
        if nxt $= true$ and $c = n_s$ then send (OK, $\mu$) to $R$
        else if nxt $= true$ and $c < n_s$ then
            $\mu_A :=$ MAC.Sign($k_A, (r, $ERR$)$), send (ERR, $\mu_A$) to $R$
    else if acc $= false$ then $\mu_A :=$ MAC.Sign($k_A, (r, $ERR$)$), send (ERR, $\mu_A$) to $R$
```

Fig. 9. `VERIFICATION` stage specification for the aggregator A

the final aggregation result and hash commitment since it knows them already after the `UPFLOW` stage, and has already compared them to the values received from R during the `DOWNFLOW` stage. In case of mismatch `acc` is already set to $false$. In this case A sends a negative acknowledgement `ERR` to R together with the own MAC value μ_A. If `acc` is $true$ at the beginning of the stage then A checks whether it is the only node participating in the protocol. In this case it simply replies with the positive acknowledgement `OK` and its MAC value μ_A. Otherwise, A initializes timer t and starts waiting for the acknowledgements of other nodes. A counts the number of the received acknowledgements until every node has replied. In our protocol (unlike [2]) any node S_i can reply with the negative acknowledgement. In this case A simply aborts and forwards this negative acknowledgement and the MAC value μ_i to R. Otherwise, A aggregates MAC values from all positive acknowledgements using the XOR function as in [2] and sends the result to R. On the other hand, the case where some acknowledgements are still missing is considered as a failure so that A replies to R with its own negative acknowledgement.

Finally, we provide description of the operations performed by R upon receiving the verification result (m, μ) from A. R accepts the aggregation result a^* only if $m =$ `OK` and the received value μ is valid, i.e., it matches the value recomputed by R using individual keys of all n_s nodes. In all other cases (including the case where R receives any authenticated negative acknowledgement $m =$ `ERR`) R terminates without accepting. Note that at the end of the `UPFLOW` stage R has already verified that $B_a(a^*) = true$.

Remark 1. Note that in [2] a node replies either with a positive acknowledgement or does not reply at all. Obviously, in this case A would need some timer; otherwise it would not know whether it still needs to wait for further acknowledgements or not. Furthermore, the solution in [2] does not explicitly abort further protocol execution in case where failures are identified before all nodes receive the required information and recompute the final hash value. By introducing negative acknowledgements we can abort the protocol execution at any time (also during the DOWNFLOW process) saving further processing costs. Any node which identifies a failure aborts and reports a negative acknowledgement to A. Note that if a failure is identified and reported by some parent node before sending required information to its child node(s) then sending this information becomes obsolete.

4.4 Security of $\mathtt{InAP1}_{agg}$

In the following we prove security of our framework in the formal model from Section 2 using the meanwhile classical cryptographic proving technique called *sequence of games* [12].

Theorem 1. *Let* $\mathtt{H}$ *be collision-resistant and* $\mathtt{MAC}$ *secure in the sense of Definitions 7 and 8. Assuming the existence of an authentication broadcast channel between* R *and the sensor nodes in* $\mathcal{S}$ *and individual secret keys* k_i *shared between each* $S_i \in \mathcal{S}$ *and* R*, the* $\mathtt{InAP1}_{agg}$ *framework from Section 4 is (optimally) secure in the sense of Definition 6.*

Proof (Sketch). We define a sequence of games $\mathbf{G}_i$, $i = 0, \ldots, 7$ with the adversary $\mathcal{I}$ against the (optimal) security of $\mathtt{InAP1}_{agg}$. In each game we denote Win_i the event that $\mathcal{I}$ breaks the (optimal) security of $\mathtt{InAP1}_{agg}$ (wins in $\mathsf{Game}^{\mathtt{opt-sec}}_{\mathtt{InAP1}_{agg}}(\mathcal{I}, \kappa)$), that is there exists session s in which R^s accepts the aggregation result a^* and there exists NO list $\mathbf{v}_\mathtt{c}$ of size $n_\mathtt{c} = n_s - n_\mathtt{h}$ with $B_v(\mathbf{v}_\mathtt{c}) = true$ such that $a^* = agg(a_\mathtt{h}, \mathbf{v}_\mathtt{c})$. Note that in our framework the unique session id s is given by the random nonce r chosen by R. The classical idea behind the *sequence of games* technique is to start with the adversarial game (interaction) described in the original security definition (here Definition 6) and construct subsequent games via small incremental changes until the resulting adversarial probability matches the desired value (in our case 0). Upon estimating the probability difference between two consecutive games in the sequence (using the *Difference Lemma* [12, Lemma 1]) one can upper-bound the total probability of a successful attack.

Game $\mathbf{G}_0$. This game is the real interaction between $\mathcal{I}$ and instances of R and of sensor nodes in $\mathcal{S}$ according to the description of $\mathsf{Game}^{\mathtt{opt-sec}}_{\mathtt{InAP1}_{agg}}(\mathcal{I}, \kappa)$ within Definition 6 where instances of all uncorrupted parties are replaced by the simulator Δ. Note that Δ has a view on all computations which it simulates.

Game $\mathbf{G}_1$. This game is identical to Game $\mathbf{G}_0$ with the only exception that the simulation fails if an equal nonce r is generated by R in two different sessions. Considering q_s as the total number of protocol sessions, the probability that a randomly chosen nonce appears twice is bound by $q_\mathsf{s}^2/2^\kappa$. Hence,

$$|\Pr[\mathsf{Win}_1] - \Pr[\mathsf{Win}_0]| \leq \frac{q_\mathsf{s}^2}{2^\kappa}. \tag{1}$$

Game $\mathbf{G}_2$. This game is identical to Game $\mathbf{G}_1$ with the only exception that the simulation fails if any instance S_i^s successfully verifies any broadcast message which has not been previously output by the corresponding instance R_i^s. Since Δ simulates all uncorrupted protocol parties it can easily detect this event. Let $\epsilon_{\mathtt{BC}}$ denote the probability of the successful attack on the applied broadcast authentication mechanism. By assumption $\epsilon_{\mathtt{BC}}$ is negligible. Considering two broadcast messages in each session we get

$$|\Pr[\mathsf{Win}_2] - \Pr[\mathsf{Win}_1]| \leq 2q_{\mathsf{s}}\epsilon_{\mathtt{BC}}. \tag{2}$$

Having excluded collisions of random nonces and attacks against the broadcast messages of R we remark that this game excludes any forgeries and replay attacks on the messages of R.

Game $\mathbf{G}_3$. This game is identical to Game $\mathbf{G}_2$ with the only difference that the simulation fails if there exists an instance S_i^s of an uncorrupted node S_i which has not output its positive acknowledgement $(\mathtt{OK}, \mu_i)$ but R^s has accepted. The only condition for the acceptance of the aggregation result by R^s is a correct verification of the received acknowledgement μ by recomputing individual μ_i and aggregating them using the XOR function. Since S_i and R are uncorrupted the individual key k_i remains unknown to $\mathcal{I}$. Let $\epsilon_{\mathtt{MAC}}$ be the probability of a successful attack against $\mathtt{MAC}$. By assumption $\epsilon_{\mathtt{MAC}}$ is negligible. Since there are at most n_s nodes and q_{s} protocol sessions we obtain

$$|\Pr[\mathsf{Win}_3] - \Pr[\mathsf{Win}_2]| \leq n_s q_{\mathsf{s}}\epsilon_{\mathtt{MAC}}. \tag{3}$$

Similar to Game $\mathbf{G}_2$ this game excludes any forgeries and replay attacks on the acknowledgements of sensor nodes.

Game $\mathbf{G}_4$. This game is identical to Game $\mathbf{G}_3$ with the only exception that the simulation fails immediately after computing any hash commitment collision on behalf of uncorrupted parties. The simulator is easily able to detect this event since it computes hash commitments for all uncorrupted parties. Note that computation of equal hash commitments on equal data values (e.g., two or more sensors report equal data) does not count as a collision. Considering $\epsilon_{\mathtt{H}}$ as the probability of finding a successful hash collision for $\mathtt{H}$ and at most n_s computed hash commitments for each executed protocol session, we obtain

$$|\Pr[\mathsf{Win}_4] - \Pr[\mathsf{Win}_3]| \leq n_s q_{\mathsf{s}}\epsilon_{\mathtt{H}}. \tag{4}$$

Having excluded collisions of hash commitments and due to the fact that every sensor node verifies predicates B_v and B_a for every received value in $\mathbf{v}$, $\mathbf{v}_{\mathsf{P}}$ and $\mathbf{a}^{\mathsf{co}}$ during the protocol execution we follow that in this game every uncorrupted node outputs own positive acknowledgement only if its contribution has been correctly included into the aggregation result a^* and all checked predicates are $true$. Successful verification of predicates implies that for a_{c} corresponding to the aggregation value of all adversarial inputs $B_a(a_{\mathsf{c}}) = true$ should hold. Hence, due to the correctness property of agg there exists a tuple $\mathbf{v}_{\mathsf{c}}$ of size $n_s - n_{\mathtt{h}}$ such that $B_v(\mathbf{v}_{\mathsf{c}}) = true$. Therefore,

$$\Pr[\mathsf{Win}_4] = 0. \tag{5}$$

Considering, Equations (1) to (5) we can upper-bound the total probability of a successful attack as follows:

$$\mathsf{Succ}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\kappa) \leq \frac{q_{\texttt{S}}^2}{2^\kappa} + 2q_{\texttt{S}}\epsilon_{\texttt{BC}} + n_s q_{\texttt{S}}\epsilon_{\texttt{MAC}} + n_s q_{\texttt{S}}\epsilon_{\texttt{H}},$$

which is negligible according to the assumptions made in the theorem.

5 Boolean Predicate Examples for Various Aggregation Functions

In the following we give practical examples that illustrate specification of reasonable input/output predicates B_v/B_a for some aggregation functions. Note that in order to achieve reasonable setting one usually needs to restrict possible input intervals (otherwise any $\mathcal{I}$ can provide any input value of its choice and would still satisfy the requirement of optimal security (as also mentioned in [2])). Note also that, if required, any node S_i is able to identify the number of original data values used to compute the intermediate aggregation result at some position p of the reference list $\mathbf{a}$ computed by A. Let $n_v \in [1, n]$ denote this total number. Given the total number of nodes n and *any* position $p \in [1, n]$ every S_i (not necessary assigned to p) can compute the *relative distance*[1] $\delta := \lfloor \log_2 n \rfloor - \lfloor \log_2 p \rfloor$. Let $p_r := (p+1)2^\delta - 1$ and $p_\ell := p2^\delta$. S_i estimates n_v as follows:

if $p_r \leq n$ then $n_v := 2^{\delta+1} - 1$
else if $p_\ell \leq n < p_r$ then $n_v := 2^{\delta+1} - (p_r - n)$
else $n_v := 2^\delta$

For example, in Figure 3 given $n = 7$ and $p = 2$ we obtain $n_v = 3$, that is the intermediate aggregation value $\mathbf{a}[2]$ is the output of agg on 3 inputs. Assuming that the tree is incomplete such that $n = 4$ and $p = 2$ we obtain $n_v = 2$.

5.1 MIN, MAX

Let agg be a MIN (or MAX) function, i.e., on input $\mathbf{v} := \{v_1, \dots, v_n\}$, $v_i \in \mathbb{R}$, $i \in [1, n]$, $n \in \mathbb{N}$ the aggregated result $agg(\mathbf{v})$ corresponds to the minimal (or maximal) value in $\mathbf{v}$. Restricting each v_i to a value in the interval between $[v_{\texttt{min}}, v_{\texttt{max}}]$ (with $v_{\texttt{min}} \leq v_{\texttt{max}}$) we obtain $B_v(v) = true$ if and only if $v_{\texttt{min}} \leq v \leq v_{\texttt{max}}$ whereby $v_{\texttt{min}}$ and $v_{\texttt{max}}$ are part of $\texttt{aux}_v$. Consequently, $B_a(a) = true$ if and only if $v_{\texttt{min}} \leq a \leq v_{\texttt{max}}$ whereby $\texttt{aux}_v = \texttt{aux}_v$.

5.2 SUM, COUNT, *Φ*-QUANTILE

Let agg be a SUM function, i.e., on input $\mathbf{v} := \{v_1, \dots, v_n\}$, $v_i \in \mathbb{R}$, $i \in [1, n]$, $n \in \mathbb{N}$ the aggregated result $agg(\mathbf{v})$ corresponds to $\sum_{i=1}^n v_i$. Assuming that each v_i is restricted to $[v_{\texttt{min}}, v_{\texttt{max}}]$ as in MIN/MAX $B_a(a) = true$ if and only if $nv_{\texttt{min}} \leq a \leq nv_{\texttt{max}}$ whereby n, $v_{\texttt{min}}$, and $v_{\texttt{max}}$ are part of $\texttt{aux}_a$. If agg is COUNT then $v_i \in [0, 1]$, $v_i \in \mathbb{N}$. Chan *et al.* [2] show how to implement ϕ-QUANTILE based on COUNT.

[1] Visualizing the list as a binary tree (e.g. Figure 3) the relative distance between two vertices equals to the difference between levels to which these vertices are assigned, e.g., if the relative distance is 0 then both vertices are located at the same level in the tree.

5.3 PRODUCT

Let agg be a PRODUCT function, i.e., on input $\mathbf{v} := \{v_1, \ldots, v_n\}$, $v_i \in \mathbb{R}$, $i \in [1, n]$, $n \in \mathbb{N}$ the aggregated result $agg(\mathbf{v})$ corresponds to $\prod_{i=1}^{n} v_i$. Let v_i be restricted to the interval $[v_{\mathtt{min}}, v_{\mathtt{max}}]$ as in MIN/MAX. For the specification of the output predicate we need to take into account that $v_{\mathtt{min}}$ and $v_{\mathtt{max}}$ may have different signs and that the number of inputs for the single aggregation can be even or odd. Let $|v|$ denote the absolute value of v. It is easy to check that the following specification of B_a provides the required consistency:

```
if $v_{\mathtt{max}} \le 0$ then
    if $n$ *even* then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{max}}^n \le a \le v_{\mathtt{min}}^n$
    if $n$ *odd* then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{min}}^n \le a \le v_{\mathtt{max}}^n$
if $v_{\mathtt{min}} < 0$ and $v_{\mathtt{max}} > 0$ then
    if $|v_{\mathtt{min}}| \le |v_{\mathtt{max}}|$ then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{min}} v_{\mathtt{max}}^{n-1} \le a \le v_{\mathtt{max}}^n$
    if $|v_{\mathtt{min}}| > |v_{\mathtt{max}}|$ then
        if $n$ *even* then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{min}}^{n-1} v_{\mathtt{max}} \le a \le v_{\mathtt{min}}^n$
        if $n$ *odd* then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{min}}^n \le a \le v_{\mathtt{min}}^{n-1} v_{\mathtt{max}}$
if $v_{\mathtt{min}} \ge 0$ then $B_a(\mathbf{v}) = true$ if and only if $v_{\mathtt{min}}^n \le a \le v_{\mathtt{max}}^n$
```

5.4 Additive and Multiplicative AVERAGE

Let agg be an *additive* AVERAGE function, i.e., on input $\mathbf{v} := \{v_1, \ldots, v_n\}$, $v_i \in \mathbb{R}$, $i \in [1, n]$, $n \in \mathbb{N}$ the aggregated result $agg(\mathbf{v})$ corresponds to $(\sum_{i=1}^{n} v_i)/n$. Assuming that $v_i \in [v_{\mathtt{min}}, v_{\mathtt{max}}]$ as in MIN/MAX $B_a(a) = true$ if and only if $v_{\mathtt{min}} \le a \le v_{\mathtt{max}}$.

Let agg be a *multiplicative* AVERAGE function, i.e., on input $\mathbf{v} := \{v_1, \ldots, v_n\}$, $v_i \in \mathbb{R}$, $i \in [1, n]$, $n \in \mathbb{N}$ the aggregated result $agg(\mathbf{v})$ corresponds to $(\prod_{i=1}^{n} v_i)/n$. Again, we assume that $v_i \in [v_{\mathtt{min}}, v_{\mathtt{max}}]$ as in MIN/MAX. The output predicate can then be defined exactly as in PRODUCT for the difference that all bounds should be divided by n, e.g., if $v_{\mathtt{min}} \ge 0$ then $B_a(\mathbf{v}) = true$ if and only if $\frac{v_{\mathtt{min}}^n}{n} \le a \le \frac{v_{\mathtt{max}}^n}{n}$.

6 Conclusions and Future Work

Along the lines of this paper we have presented a formal communication and security model and a novel framework for the in-network aggregation in WSNs, focusing on the single aggregator scenario. Our framework is both, practical and provably secure (in the cryptographic sense). The modularity of our model provides basis for further extensions (e.g. towards a hierarchical scenario [2] or concealed data aggregation processes [1, 14]). The abstract definition of the aggregation function agg and its input resp. output predicates B_v resp. B_a provides basis for the specification of the integrity checks that are necessary for the optimal security of the aggregation process.

Acknowledgements

This work was supported by the European Commission through STREP UbiSec&Sens (`http://www.ist-ubisecsens.org`). The authors are also thankful to Benedikt Driessen and Axel Poschmann for the (prototypical) implementation of the framework within the UbiSec&Sens project.

References

1. C. Castelluccia, E. Mykletun, and G. Tsudik. Efficient Aggregation of Encrypted Data in Wireless Sensor Networks. In *International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2005)*, pages 109–117. IEEE CS, 2005.
2. H. Chan, A. Perrig, and D. Song. Secure Hierarchical In-Network Aggregation in Sensor Networks. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006*, pages 278–287. ACM, 2006.
3. A. Deshpande, S. K. Nath, P. B. Gibbons, and S. Seshan. Cache-and-Query for Wide Area Sensor Databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 503–514. ACM, 2003.
4. D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *MOBICOM*, pages 263–270, 1999.
5. L. Hu and D. Evans. Secure Aggregation for Wireless Network. In *2003 Symposium on Applications and the Internet Workshops (SAINT 2003)*, pages 384–394. IEEE Computer Society, 2003.
6. C. Intanagonwiwat, D. Estrin, R. Govindan, and J. S. Heidemann. Impact of Network Density on Data Aggregation in Wireless Sensor Networks. In *ICDCS*, pages 457–458, 2002.
7. D. Liu and P. Ning. Multilevel μTESLA: Broadcast Authentication for Distributed Sensor Networks. *ACM Transactions in Embedded Computing Systems*, 3(4):800–836, 2004.
8. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.
9. R. C. Merkle. A Certified Digital Signature. In *Advances in Cryptology - CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1990.
10. A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Netowrks. In *MOBICOM*, pages 189–199, 2001.
11. B. Przydatek, D. X. Song, and A. Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems, SenSys 2003*, pages 255–265. ACM, 2003.
12. V. Shoup. Sequences of Games: A Tool for Taming Complexity in Security Proofs. Cryptology ePrint Archive, Report 2004/332, 2006. `http://eprint.iacr.org/2004/332.pdf`.
13. M. Sirivianos, D. Westhoff, F. Armknecht, and J. Girao. Non-Manipulable Aggregator Node Election Protocols for Wireless Sensor Networks. In *International Symposium on Modeling and Optimization in Mobile, Ad-Hoc and Wireless Networks (WiOpt 2007)*. IEEE Computer Society, 2007. to appear, also available at `http://www.ics.uci.edu/~msirivia/publications/sane-fullpaper.pdf`.
14. D. Westhoff, J. Girao, and M. Acharya. Concealed Data Aggregation for Reverse Multicast Traffic in Sensor Networks: Encryption, Key Distribution, and Routing Adaptation. *IEEE Transactions on Mobile Computing*, 05(10):1417–1431, 2006.