Security Model and Framework for Information Aggregation in Sensor Networks

MARK MANULIS
Technische Universität Darmstadt
JÖRG SCHWENK
Ruhr-Universität Bochum

Information aggregation is an important operation in wireless sensor networks (WSNs) executed for the purpose of monitoring and reporting environmental data. Due to the performance constraints of sensor nodes the in-network form of the aggregation is especially attractive since it allows saving expensive resources during frequent network queries. Easy accessibility of networks and nodes and almost no physical protection against corruptions raise high security challenges. Protection against attacks aiming to falsify the aggregated result is considered to be of prime importance.

In this article we design the first general framework for secure information aggregation in WSNs focusing on scenarios where aggregation is performed by one of its nodes. The framework achieves security against node corruptions and is based solely on the symmetric cryptographic primitives that are more suitable for WSNs in terms of efficiency. We analyze performance of the framework and unlike many previous approaches increase confidence in it by a rigorous proof of security within the specially designed *formal security model*.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Security, Algorithms, Design

Additional Key Words and Phrases: Aggregation, framework, in-network processing, provable security, wireless sensor network

ACM Reference Format:

Manulis, M. and Schwenk, J. 2009. Security model and framework for information aggregation in sensor networks. ACM Trans. Sensor Netw. 5, 2, Article 13 (March 2009), 28 pages. DOI = 10.1145/1498915.1498919 http://doi.acm.org/10.1145/ 1498915.1498919

A preliminary version of this paper appeared in O. Gervasi and M. Gavrilova (Eds.) *Proceedings of the 2007 International Conference on Computational Science and its Applications (ICCSA'07)*.
Authors' addresses: M. Manulis, Cryptographic Protocols Group, Department of Computer Science, TU Darmstadt + CASED, Mornewegstr. 30, 64293, Darmstadt, Germany; email: mark@manulis.eu; J. Schwenk, Ruhr-Universität Bochum, Universitätsstr. 150, 44780 Bochum, Germany; email: joerg.schwenk@nds.rub.de.

DOI 10.1145/1498915.1498919 http://doi.acm.org/10.1145/1498915.1498919

1. INTRODUCTION

Monitoring and reporting of the physically measured data to some querying device represented by a sink, base station, or mobile reader is one of the main goals for the deployment of wireless sensor networks (WSNs). This task is especially important in scenarios where high confidence in 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 described in 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. Sirivianos et al. [2007]). Moreover, this role is usually temporary and randomly changed 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 a certain aggregation topology computed in the underlying protocol (e.g. Madden et al. [2002]). 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 their own advantages in terms of efficiency, security, and administration. In this article we focus on the single aggregator scenario and address one of the most important issues—security.

Due to the usually missing physical protection, deployed sensor nodes may easily become subject to an adversarial attack. In addition to active network attacks, node corruptions are 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. Thus, designing and adequate formal security model together with some general (in the cryptographic sense) provably secure practical solution appears to be an interesting task.

1.1 Related Work

Currently, there exist few scientific results in the area of secure information aggregation dealing with security of the aggregation process in the presence of corrupted nodes. Hu and Evans [2003] 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. It is remarkable that previous solutions like Estrin et al. [1999], Madden et al. [2002], Intanagonwiwat et al. [2002], and Deshpande et al. [2003] addressed the scenario with only honest communication participants and are therefore not of much interest in the context of this work. The protocol in Hu and Evans [2003] requires an underlying protocol for the construction of the aggregation tree (e.g. Madden et al. [2002]), as well as shared individual keys possibly predeployed in sensor nodes, and an authentic

unidirectional communication channel between the sink and the involved nodes (e.g. Perrig et al. [2001, 2002] and Liu and Ning [2004]). 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 both of its children would allow complete falsification of the final aggregated value. Przydatek et al. [2003] proposed a Secure Information Aggregation (SIA) framework for sensor networks, which provides better resilience against malicious sensor nodes than the process in Hu and Evans [2003]. 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 [Merkle 1990] for the received individual inputs and proves the 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 uniform sampling of the input values, for example, computation of the MIN/MAX, AVERAGE, and MEDIAN. Recently, Chan et al. [2006] described a solution for the hierarchical in-network aggregation that 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 achievable by so-called *direct data injection*, wherein 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 Hu and Evans [2003] it requires the construction of the aggregation tree structure (e.g. Madden et al. [2002]). Optimal security is achieved via interactive computation of 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

Definitions of security in the aforementioned solutions are intuitive rather than formal. Therefore designing a formal security model to allow cryptographically sound security proofs is an important work. In addition, previous solutions are not general since they have been designed for some concrete aggregation functions (e.g. SUM) and then extended to deal with further functions. A more general approach would be to give an abstract definition of the aggregation function and its security relevant properties. In this article we contribute in two different ways; in Section 2 we develop a formal security model for the in-network aggregation process and formalize for the first time, the aggregation function in a very abstract way. In Section 3 we show how this abstraction can be used in practice. After the introduction of the required building blocks in Section 4, we focus on the description of our general aggregation framework $\texttt{InAP1}_{agg}$ in Section 5, where we also evaluate its performance and prove its security in

our formal model with well-known cryptographic proving techniques. In terms of performance our framework relies on the efficient 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. We start by describing the communication model, continue with the formal definition of an abstract aggregation function, and conclude with the adversarial model and security definitions.

2.1 Communication Model and Participants

2.1.1 *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 $A \in \mathcal{S}$ 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 that 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 that is supposed to obtain the aggregated result.

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

2.1.3 *Secret Keys.* For the purpose of authentication we consider that every sensor node S_i and R is in possession of some secret key denoted k_i and k_R, respectively, whereby the notation k is used for generality. This key should be seen as a place holder, that is any k can in practice consist of several secret values, for example, 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 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 2.1 (*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 aux as one of the inputs.

Many widely used aggregation functions such as SUM, PRODUCT, additive/multiplicative AVERAGE, MIN/MAX, and so forth satisfy the properties of identity, symmetry and associativity. Note that AVERAGE $agg(\mathbf{v}; \mathtt{aux})$ can be computed correctly only if n is known during each iteration (as part of 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 for the inputs and B_a for outputs of the aggregation function *agg*. Briefly speaking, B_v evaluated on the inputs of *agg* and auxiliary information $\mathtt{aux}_v$ returns *true* only if these inputs are legal values. On the other hand, B_a evaluated on some aggregation result of *agg* and auxiliary information $\mathtt{aux}_a$ should return *false* if no legal inputs for the computation of this result exist. In both cases, the auxiliary information, $\mathtt{aux}_v$ and $\mathtt{aux}_a$, may contain further, not necessarily equal parameters needed to perform the verification. Both predicates, B_v and B_a, will be used in our definition of security in order to handle node corruptions in a reasonable way.

Definition 2.2 (*Aggregation Input/Output Predicates*). By $B_v(v; \mathtt{aux}_v)$ and $B_a(a; \mathtt{aux}_a)$ we denote a *Boolean predicate* for any input $v \in \mathbb{R}$ and output $a \in \mathbb{R}$ of *agg* where $\mathtt{aux}_v$ and $\mathtt{aux}_a$ 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 $B_a(a; \mathtt{aux}_a)$.

The predicates B_v and B_a, can be used to restrict inputs and outputs of *agg*, for example., 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 $nv_{\mathtt{max}}$, 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 properties of correctness and consistency are satisfied for any $\mathbf{v}$ of size n. The predicates 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 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 2.3 (*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 $\mathcal{S}^s$).

Definition 2.4 (*Correctness of* $\mathtt{InAP1}_{agg}$). 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 $B_v(v_i) = true$; in any other case R^s rejects.

2.4 Adversarial Model

Next we specify the adversarial setting for the in-network aggregation protocols. The whole communication is assumed to be controlled by the probabilistic polynomial-time (PPT) adversary $\mathcal{I}$: $\mathcal{I}$ is able to replay, modify, delay, drop, and deliver protocol messages out of order as well as inject own messages. Since $\mathcal{I}$ can always refuse to deliver protocol messages our model does not address any denial-of-service attacks (similar to Przydatek et al. [2003], and Chan et al. [2006]), which aim to prevent R from obtaining any result at all. 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.

2.4.1 *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 that 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 its 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 that the receiving instance would generate after processing m. This outgoing message might be an empty string if m is unexpected or if a failure occurred. Further, there are two special *Send* queries of the form $Send(S_i^s, \text{'start'}, \mathcal{S}^s, \mathrm{A}^s, R^s, m)$ and $Send(R^s, \text{'start'}, \mathcal{S}^s, \mathrm{A}^s, m)$. 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 $\mathcal{S}^s \setminus S_i^s$, 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. Both queries may contain auxiliary message m, which may be needed to invoke the protocol execution. In response to these queries, $\mathcal{I}$ receives the first message generated by the requested 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 $Corrupt(S_i)$ respectively $Corrupt(R)$ the adversary can obtain the secret key k_i respectively k_R.

Definition 2.5 (*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*.

2.4.2 *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 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 occurred or an attack has been recognized. Consequently, the goal of $\mathcal{I}$ is to influence R^s accepting some biased aggregation result. After its termination, the instance, cannot be invoked for any 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 $\texttt{InAP1}_{agg}$ 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 2.6 (*Freshness of* R). Let R^s be the instance that has accepted in session s of $\texttt{InAP1}_{agg}$, 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.

Basically, whenever $\mathcal{I}$ corrupts R, all its instances that have not yet terminated 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 the consideration of corrupted sensor nodes, which equally holds for our protocol and the protocols in Przydatek et al. [2003] and Chan et al. [2006]. Namely, corrupted nodes can report data that strongly deviates from the actual. Even actual 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 Chan et al. [2006] 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 from 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 Chan et al. [2006], does not aim to detect such attacks. Instead, it focuses on the modification of the aggregated result based on attacks in which corrupted nodes try to report semantically incorrect inputs to the aggregation function, that is inputs v_i with $B_v(v_i) = \mathit{false}$. In the single aggregator scenario such *stealthy attacks* [Przydatek et al. 2003] 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 an indispensable part of any secure protocol in the strong corruption model.

Definition 2.7 (*Optimal Security of* $\texttt{InAP1}_{agg}$). Let $\mathcal{I}$ be a PPT adversary operating in the strong corruption model and $\mathsf{Game}^{\texttt{opt-sec}}_{\texttt{InAP1}_{agg}}(\mathcal{I}, \kappa)$ 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 $\texttt{InAP1}_{agg}$ such that at the end of this interaction there is a **fresh** instance R^s, which has accepted with the aggregation result a^*. Let $\mathcal{S}^s_{\mathrm{h}} \subseteq \mathcal{S}^s$ 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 $\mathbf{v}_{\mathrm{h}}$ be a set/list of size $n_{\mathrm{h}} \in [1, n_s]$ containing original inputs of instances in $\mathcal{S}^s_{\mathrm{h}}$ and $a_{\mathrm{h}} := agg(\mathbf{v}_{\mathrm{h}})$.

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

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

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

We say that $\mathtt{InAP1}_{agg}$ is *optimally secure* if $\mathsf{Succ}^{\mathtt{opt-sec}}_{\mathtt{InAP1}_{agg}}(\kappa)$ 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, where the receiver instance accepts a^* such that $a_{\mathtt{c}}$ does not satisfy B_a. The only general condition for $B_a(a_{\mathtt{c}}) = \mathit{false}$ is when all inputs $\mathbf{v}_{\mathtt{c}}$ with $a_{\mathtt{c}} := agg(\mathbf{v}_{\mathtt{c}})$ do not satisfy B_v: if $B_v(\mathbf{v}_{\mathtt{c}}) = \mathit{false}$ (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 $B_v(\mathbf{v}_{\mathtt{c}}) = \mathit{true}$, 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. SPECIFICATION OF BOOLEAN PREDICATES FOR COMMON AGGREGATION FUNCTIONS

In the following we give practical examples that illustrate specification of reasonable input/output predicates B_v/B_a for several commonly used aggregation functions. In order to achieve reasonable settings one usually needs to restrict possible input intervals, otherwise any $\mathcal{I}$ could provide any input value of its choice and would still satisfy the requirement of optimal security (as also mentioned in Chan et al. [2006]).

3.1 MIN, MAX

Let *agg* be a MIN (or MAX) function: 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 the minimal (or maximal) value in $\mathbf{v}$. Restricting each v_i to a value in the interval between $[v_{\mathtt{min}}, v_{\mathtt{max}}]$ ($v_{\mathtt{min}} \leq v_{\mathtt{max}}$) we obtain $B_v(v) = \mathit{true}$ if and only if $v_{\mathtt{min}} \leq v \leq v_{\mathtt{max}}$ whereby $v_{\mathtt{min}}$ and $v_{\mathtt{max}}$ are part of $\mathtt{aux}_v$. Consequently, $B_a(a) = \mathit{true}$ if and only if $v_{\mathtt{min}} \leq a \leq v_{\mathtt{max}}$ whereby $\mathtt{aux}_v = \mathtt{aux}_v$.

3.2 SUM, COUNT, Φ-QUANTILE

Let *agg* be a SUM function: 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 $\sum_{i=1}^{n} v_i$. Assuming that each v_i is restricted to $[v_{\mathtt{min}}, v_{\mathtt{max}}]$ as in MIN/MAX $B_a(a) = \mathit{true}$ if and only if $nv_{\mathtt{min}} \leq a \leq nv_{\mathtt{max}}$ whereby n, $v_{\mathtt{min}}$, and $v_{\mathtt{max}}$ are part of $\mathtt{aux}_a$. If *agg* is COUNT then $v_i \in [0, 1]$, $v_i \in \mathbb{N}$. Chan et al. [2006] show how to implement ϕ-QUANTILE based on COUNT.

3.3 PRODUCT

Let *agg* be a PRODUCT function: 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 $[v_{\min}, v_{\max}]$ as in MIN/MAX. For the specification of the output predicate, we need to take into account that $v_{\min}$ and $v_{\max}$ 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 $v_{\max} \leq 0$ then
  if n *even* then $B_a(\mathbf{v}) = true$ if and only if $v_{\max}^n \leq a \leq v_{\min}^n$
  if n *odd* then $B_a(\mathbf{v}) = true$ if and only if $v_{\min}^n \leq a \leq v_{\max}^n$
if $v_{\min} < 0$ and $v_{\max} > 0$ then
  if $|v_{\min}| \leq |v_{\max}|$ then $B_a(\mathbf{v}) = true$ if and only if $v_{\min} v_{\max}^{n-1} \leq a \leq v_{\max}^n$
  if $|v_{\min}| > |v_{\max}|$ then
    if n *even* then $B_a(\mathbf{v}) = true$ if and only if $v_{\min}^{n-1} v_{\max} \leq a \leq v_{\min}^n$
    if n *odd* then $B_a(\mathbf{v}) = true$ if and only if $v_{\min}^n \leq a \leq v_{\min}^{n-1} v_{\max}$
if $v_{\min} \geq 0$ then $B_a(\mathbf{v}) = true$ if and only if $v_{\min}^n \leq a \leq v_{\max}^n$

3.4 Additive and Multiplicative AVERAGE

Let *agg* be an *additive* AVERAGE function: 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 $v_i \in [v_{\min}, v_{\max}]$, as in MIN/MAX $B_a(a) = true$ if and only if $v_{\min} \leq a \leq v_{\max}$.

Let *agg* be a *multiplicative* AVERAGE function: 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 $v_i \in [v_{\min}, v_{\max}]$ 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, for example, if $v_{\min} \geq 0$ then $B_a(\mathbf{v}) = true$ if and only if $\frac{v_{\min}^n}{n} \leq a \leq \frac{v_{\max}^n}{n}$.

4. BUILDING BLOCKS OF THE $\texttt{InAP1}_{AGG}$ FRAMEWORK

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

4.1 Background on used Symmetric Cryptographic Primitives

Definition 4.1 (*Collision-Resistant Hash Function*). Let $\mathtt{H} : \{0,1\}^* \to \{0,1\}^\kappa$, $\kappa \in \mathbb{N}$ be a hash function. We say that $\mathtt{H}$ is *collision-resistant* if the following success probability over all PPT adversaries $\mathcal{I}$ running within time κ, is negligible:

$$\mathsf{Succ}_{\mathtt{H}}^{\mathtt{cr}}(\kappa) := \max_{\mathcal{I}} \left| \Pr\left[x_1, x_2 \leftarrow \mathcal{I}(1^\kappa) \ : \ \begin{array}{c} x_1 \neq x_2 \wedge \\ \mathtt{H}(x_1) = \mathtt{H}(x_2) \end{array} \right] \right|.$$

Definition 4.2 (*EUF-CMA Secure Message Authentication Code*). Let $\mathtt{MAC} := (\mathtt{Gen}, \mathtt{Sign}, \mathtt{Verify})$ be a *message authentication code* consisting of the following algorithms:

`Gen` A probabilistic algorithm that on input of a security parameter 1^κ, outputs a secret key $k \in_R \{0,1\}^\kappa$.

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

`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.

We say that `MAC` is *existentially-unforgeable under chosen message attacks* (EUF-CMA) if the following success probability over all PPT forger algorithms $\mathcal{F}$ given access to the signing oracle $\texttt{Sign}(k,\cdot)$ and running within time κ, is negligible:

$$\mathsf{Succ}^{\texttt{euf-cma}}_{\texttt{MAC}}(\kappa) := \max_{\mathcal{F}} \left| \Pr \left[\begin{array}{c} k \in_R \{0,1\}^\kappa; \\ (m,\mu) \leftarrow \mathcal{F}^{\texttt{Sign}(k,\cdot)}(1^\kappa) \end{array} : \begin{array}{c} \texttt{Verify}(k,m,\mu) = 1 \wedge \\ \neg \texttt{Sign}(k,m) \end{array} \right] \right|.$$

4.2 List Structures

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

Definition 4.3 (*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 ith position*. An empty element is denoted ε. Upon initialization, 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 *prepended* 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 $\mathbf{x}[|\mathbf{x}|] = y$.

The defined lists can also be represented via binary trees and vice versa, for example, 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 4.4 (*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 *copath 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 $\mathbf{x}[2p]$ we denote the *first* and by $\mathbf{x}[2p+1]$ the *second child element* of $\mathbf{x}[p]$. Consequently, $\mathbf{x}[p]$ is the *parent element* of $\mathbf{x}[2p]$ and $\mathbf{x}[2p+1]$.

5. SPECIFICATION OF THE $\texttt{InAP1}_{AGG}$ FRAMEWORK

Our $\texttt{InAP1}_{agg}$ framework consists of a protocol that proceeds in three stages: `UPFLOW`, `DOWNFLOW`, and `VERIFICATION`. In the `UPFLOW` stage the aggregator node, A, collects individual inputs of other nodes, $S_i \in \mathcal{S}$, computes the aggregation result, and forwards it together with some additional authentication information to R. In the `DOWNFLOW` stage every S_i receives information that it then uses

in the VERIFICATION stage to check and confirm that the aggregation result contains its individual input. Finally, R checks that every S_i, is confirmed and accepts the aggregation result in the positive case. We first give a high level overview of the framework and introduce useful notations before going into the specification of details.

5.1 High Level Description

5.1.1 *Assumptions.* We assume that prior to the execution of the aggregation protocol, all sensor nodes in $\mathcal{S}$ know the identity of the previously chosen aggregator node, A $\in \mathcal{S}$. Additionally, we assume that there is an authenticated broadcast channel between R and nodes in $\mathcal{S}$, and that R knows how many nodes exist (are alive) and should participate in the protocol, and what their identities are. Finally, we assume that unique sensor node identities are used for the purpose of addressing and that every received message reveals the identity of its sender.

5.1.2 *Protocol Invocation.* The aggregation protocol is invoked by R, whose invocation message is sent over the authenticated broadcast channel and contains a *random nonce r* (chosen freshly for every new invocation) and the expected number of nodes n_s (so many nodes, including A are expected to participate and contribute their data). Only after the successful verification of this invocation message, each sensor node starts with the first stage of the protocol. The random nonce will be part of every authenticated protocol message. In this way, the framework prevents any replay attacks on these messages from other protocol executions and achieves security among different sessions.

5.1.3 *Reference Lists.* All stages of the framework make use of four reference lists described in the following, and exemplified for a better understanding in Figure 1 for a network of seven nodes $\mathcal{S} := \{S_1, \ldots, S_7\}$, where S_3 plays the role of A.

Each list can be visualized as a binary tree, as shown in the figure. The binary tree structure is useful for the explanation of the framework but is not needed for the implementation. We stress that prior to the protocol execution no party knows what the contents of these lists are, so that the reader may think of these lists as being empty in the beginning of the protocol. Also it is not necessary that the lists correspond to a balanced tree, as in the figure. Our protocol works for any number of nodes, not only for a power of 2.

The first list, denoted **id**, contains the *identities* of the nodes; the second list, denoted **v**, contains the *initial data values* of the nodes (the goal of the framework is to aggregate these values); the third list, denoted **a**, contains the *intermediate aggregation values* (whereby **a**[1] represents the final result of the aggregation which should be accepted or rejected by R); the fourth list, denoted **h**, contains the *intermediate hash commitments*. The composition of the contents of each list is self-explanatory from the figure. The specific computation of entries of **a** and **h** is used in the framework to allow individual verification by sensor nodes and ensure optimal security.

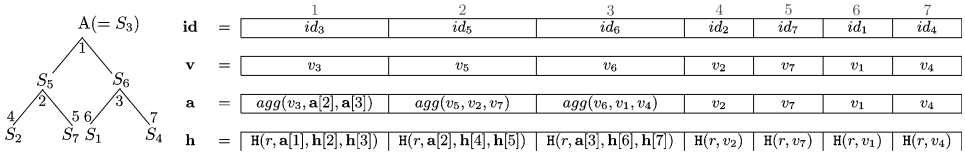

Fig. 1. List structures and notations of the framework for $\mathcal{S} := \{S_1, \ldots, S_7\}$, A $= S_3$. Left side: Visualization of node assignments in a binary tree structure. Right side: Reference lists **id**, **v**, **a**, and **h**, computed by A. Some exemplary notations: sibling of S_7 (**id**[5])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

5.1.4 *The* UPFLOW *Stage.* This is the first stage of the protocol. Immediately after the stage is invoked every sensor node S_i sends its individual data value, v_i, to the aggregator, A, which starts filling the initially empty lists, **id** and **v**, with the identities and data values of nodes from which it receives the messages. The first position in both lists belongs to A. The order in which messages arrive is reflected in the positions of the node identities in **id**, and of their individual data values in **v**. For example, in Figure 1 node S_5 is in the second position because its message was the first that the aggregator S_3 received. The aggregator aborts the protocol if it does not receive messages from all n_s expected nodes within some time period controlled by the aggregator's timer. The abortion prevents waste of further costs and is done by sending an authenticated error message to R. If all messages have been received, then A computes lists **a** and **h** in an iterative manner. The initial inputs for the iterative computations are individual data values of nodes that appear as leaves when the lists are visualized as a binary structure. The values **a**[1] and **h**[1] are then sent by A to R in an authenticated way, whereby **a**[1] is the final aggregation result whose secure computation is the goal of the protocol. The random nonce r, is used as part of this authenticated message. Except for A, no other sensor node knows the contents of the four computed lists. After R receives the aggregator's message it checks whether it is valid, even though R has not yet ensured that **a**[1] is not biased as requested by the optimal security definition. Therefore, R needs to verify **a**[1], whcih this is done through both subsequent protocol stages, DOWNFLOW and VERIFICATION. To invoke the DOWNFLOW stage, R broadcasts a tuple (r, **a**[1], **h**[1]) and every node checks the authenticity of this broadcast.

5.1.5 *The* DOWNFLOW *and* VERIFICATION *Stages.* For explanatory reasons, we first proceed with the high level description of the VERIFICATION stage. Considering reference lists as binary tree structures, the verification process will require from each sensor node that it recompute intermediate aggregation results from **a**, and hash commitments from **h** starting with its own position in the tree. The goal here is to let every node recompute and compare values for **a**[1] and **h**[1], as broadcast by R. The matching values at the end of the verification process provide each node the assurance that its own input has been correctly aggregated and that the aggregation result is optimally secure so that, at the end, every node can send its positive acknowledgement indicating its confirmation for the result.

For example, in Figure 1, node S_6 needs to compute intermediate aggregation values $\mathbf{a}[3] = agg(v_6, v_1, v_4)$ and $\mathbf{a}[1] = agg(v_3, \mathbf{a}[2], \mathbf{a}[3])$ as well as, hash commitments $\mathbf{h}[3] = \mathtt{H}(r, \mathbf{a}[3], \mathbf{h}[6], \mathbf{h}[7])$ and $\mathbf{h}[1] = \mathtt{H}(r, \mathbf{a}[1],\mathbf{h}[2], \mathbf{h}[3])$ that belong to its path. The only information S_6 knows after the UPFLOW stage is v_6, r, and n_s. V_1 and V_4 are required in order to compute $\mathbf{a}[3]$; and for $\mathbf{a}[1]$: v_3 and $\mathbf{a}[2]$. Observe, that knowledge of v_1 and v_4 will allow S_i to recompute $\mathbf{h}[6]$ and $\mathbf{h}[7]$ and so to compute $\mathbf{h}[3]$. Thus, the node still needs to learn $\mathbf{a}[2]$ and $\mathbf{h}[2]$. For security reasons, it would not be enough if S_6 just receives $\mathbf{a}[2]$ and $\mathbf{h}[2]$, since it would not be able to verify whether the received $\mathbf{a}[2]$ has been used in the computation of $\mathbf{h}[2]$. However, $\mathbf{h}[2] = \mathtt{H}(r, \mathbf{a}[2], \mathbf{h}[4], \mathbf{h}[5])$. Thus, it is sufficient if S_6 receives $\mathbf{a}[2]$, $\mathbf{h}[4]$, and $\mathbf{h}[5]$ to perform these computations. Beside computing the values in its path, S_6 additionally checks whether the input or output predicates are true for the corresponding values that the node receives: input predicates should be checked for v_1, v_4, and v_3, whereas the output predicate should be checked for $\mathbf{a}[2]$. There is no need to check predicates for values computed by S_6 itself—for $\mathbf{a}[3]$ and $\mathbf{a}[1]$—due to the correctness of the predicates. In order to perform checks of the output predicate for $\mathbf{a}[2]$, node S_6 may need to know the total number of the aggregated inputs for $\mathbf{a}[2]$. Due to the binary tree structure, it can easily figure this out using own position, p, in this tree and the total number of nodes n_s. Hence, it is important that S_6 also learns its position, p.

The idea of the DOWNFLOW stage is that the information required by every node to recompute $\mathbf{a}[1]$ and $\mathbf{h}[1]$ and to perform predicate checks comes directly from its parent node. Thus, the aggregator, which knows all list contents after the UPFLOW stage starts the dissemination process of the DOWNFLOW stage by sending two messages, one for each of its both children: nodes with identities $\mathbf{id}[2]$ and $\mathbf{id}[3]$. Then, each of these children extracts information that it in turn sends to its own children: grandchildren of the aggregator. In order to forward the extracted information, every node obviously needs the identities of its child nodes or it will not be able to correctly address them. Therefore, needed identities are also sent together with other values. The dissemination process is terminated after each sensor node has received the information needed to perform the VERIFICATION stage. In general, the dissemination process requires a logarithmic number of rounds, with each node sending at most two messages. It can be easily verified that messages sent at the end of the dissemination process, by nodes located towards the bottom of the tree, are in general shorter than messages sent by the aggregator and nodes that are close to it.

Summarizing our example-based description of the DOWNFLOW stage, we list all values that are sent from A to S_6 and then forwarded by S_6 to S_1 and S_4. A sends the following content to S_6: $id_1, id_4, v_1, v_4, v_3, \mathbf{a}[2], \mathbf{h}[4], \mathbf{h}[5]$, and $p = 3$. S_6 forwards to S_1: $v_4, v_6, v_3, \mathbf{a}[2], \mathbf{h}[4], \mathbf{h}[5]$, and $p = 6$ (to compute the position of the first child node the parent's node position should be doubled). S_6 forwards to S_4: $v_1, v_6, v_3, \mathbf{a}[3], \mathbf{h}[6], \mathbf{h}[7]$, and $p = 7$ (to compute the position of the second child node, the parent's node position should be doubled and increased by one).

5.2 Specification of the UPFLOW Stage

The main goal of the stage is that the aggregator collects individual inputs from all expected sensor nodes, then computes the final aggregation result and

On input r, n_s, and k_{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. 2. UPFLOW stage specification for the aggregator A.

the final hash commitment through an iterative evaluation of the aggregation function, *agg*, and the hash function, H, (these iterative executions are part of a separate function Commit). The formal specification of the aggregator's calculations during the UPFLOW stage is given in Figure 2 so that we focus on the brief description of its most important parts in the following.

Assuming that the verification of the invocation message is successful, the aggregator A initializes the *node counter* denoted c, the *timer* t, as well as the first two mentioned reference lists, **id** and **v**. It assigns own identity, id_{A}, to the first position of **id** and its own data value, v_{A} to the first position of **v**; whereas every other node, S_i, sends own initial data value, v_i, to A. Upon receiving a corresponding value from S_i, the aggregator extends **id** with the identities of S_i and **v** with the input of S_i. This extension is performed until A obtains messages from all other $n_s - 1$ nodes; otherwise it sends an error message, ERR, to R indicating that a failure has occurred.

Under the assumption that messages arrive in the order that 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. As already mentioned during the high level description, this improves the communication efficiency of the DOWNFLOW stage.

An important observation here is that if not all expected nodes send their inputs during this stage, then R will not be able to verify the result and will thus reject the value anyway. Therefore, we let A abort the protocol by sending the authenticated error message to save further costs.

After having received all required inputs, A computes the last two reference lists, **a** and **h**, using the auxiliary Commit function specified in Figure 3. The same function will also be used by other nodes in the DOWNFLOW stage. The computations of the Commit function start with nodes whose identities and initial data values are assigned to the later positions in the corresponding lists, **id** and **v** (note that both lists have identical sizes and that the data stored in the same position in both lists belongs to the same sensor node).

Figure 1 also specifies the output of the Commit function for the example with seven sensor nodes, $\mathcal{S} := \{S_1, \ldots, S_7\}$, where S_3 plays the role of A. Let $\mathbf{id}[i]$ be a sensor node's identity. Then, $\mathbf{a}[i]$ is the output of 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. In Figure 1, for $i = 2$ we have $\mathbf{a}[2] = agg(v_5, v_2, v_7)$, since nodes S_2 and S_7 have S_5 in their paths.

Further, $\mathbf{h}[i]$ is a hash commitment computed on r, $\mathbf{a}[i]$, $\mathbf{h}[2i]$, and $\mathbf{h}[2i + 1]$. The latter two values are included in the hash commitment only if they really

```
Commit(r, v, a, h) :
    c_v := |v|, n := |v|
    while c_v ≥ 1 do
        if 2c_v ≥ n then a := v[c_v].a, h := H(r, a[1]).h
        else if 2c_v ≤ n and 2c_v + 1 > n then
            a := agg(v[c_v], a[c_v]).a, h := H(r, a[1], h[c_v]).h
        else a := agg(v[c_v], a[c_v], a[c_v + 1]).a, h := H(r, a[1], h[c_v], h[c_v + 1]).h
        c_v := c_v − 1
    return (a, h)
```

Fig. 3. `Commit` function.

exist; otherwise they are treated as empty elements. In Figure 1, for $i = 4$ the positions $2i = 8$ and $2i + 1 = 9$ are empty, so that $\mathbf{h}[4] = \mathtt{H}(r, \mathbf{a}[4])$. Equation $2c_v \geq n$ with the running variable c_v allows checking whether positions $2i$ and $2i + 1$ exist. Per construction it is also possible that only position $2i$ exists but not $2i + 1$ (checked through $2c_v \leq n$ and $2c_v + 1 > n$), however, it is not possible that $2i + 1$ is filled without $2i$, due to the subsequent extension of lists **id** and **v** by A during the UPFLOW stage.

The construction of **a** ensures that $\mathbf{a}[1]$ gives the aggregation result $agg(\mathbf{v}[1], \ldots, \mathbf{v}[n_s])$. Similarly, the construction of **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 $B_a(\mathbf{a}[1]) = \mathit{true}$. R terminates if ERR is received or if $B_a(\mathbf{a}[1]) = \mathit{false}$. 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.

5.3 Specification of 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 that will be used by the node during the VERIFICATION stage to individually recompute the intermediate aggregation values and hash commitments along the path (in the spirit of Chan et al. [2006]). However, unlike the tree structure in Chan et al. [2006] all lists computed during the UPFLOW stage including the positions of the nodes within these lists are first known to A, but not to the other nodes. However, for a successful execution of the VERIFICATION stage, each node must receive additional information, as mentioned in the high level description of the stage. A is the first to start the dissemination process specified in Figure 4 and described in the following. First, (honest) A must check that the message received from R contains the same values that have been sent by A at the end of the UPFLOW stage; otherwise the verification process would fail. Therefore, if A notices the mismatch it sets its boolean variable acc:=*false* and immediately turns into the VERIFICATION stage, where it will send its error message to R. If no mismatch is found, A, whose identity is assigned to $\mathbf{id}[1]$ sends one message to each of its child nodes, $\mathbf{id}[2]$ (left child node) and $\mathbf{id}[3]$ (right child node). Whether any child nodes exist or not, is checked via $n_s \geq 2$.

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

Fig. 4. DOWNFLOW stage specification for the aggregator A.

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

Fig. 5. SplitIdV function.

The message addressed to $\mathbf{id}[2]$ ($\mathbf{id}[3]$) contains: (1) a list of identities $\mathbf{id}_\texttt{L}$ ($\mathbf{id}_\texttt{R}$), which consists of elements from $\mathbf{id}$ that have $\mathbf{id}[2]$ ($\mathbf{id}[3]$) in their paths, (2) a list of initial data values $\mathbf{v}_\texttt{L}$ ($\mathbf{v}_\texttt{R}$), 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 $\mathbf{v}_\texttt{P}$ consisting of $v_\texttt{A}$, (5) a list of intermediate aggregation values $\mathbf{a}_\texttt{L}^{co}$ ($\mathbf{a}_\texttt{R}^{co}$), which contains $\mathbf{a}[3]$ ($\mathbf{a}[2]$), and (6) a list of intermediate hash commitments $\mathbf{h}_\texttt{L}^{co}$ ($\mathbf{h}_\texttt{R}^{co}$), which contains $\mathbf{h}[6]$ and $\mathbf{h}[7]$ ($\mathbf{h}[4]$ and $\mathbf{h}[5]$), if such values exist.

In, our, example from Figure 1 the aggregator A $= S_3$ sends to its left child node, S_5, the following contents: $\mathbf{id}_\texttt{L} := \{id_2, id_7\}$, $\mathbf{v}_\texttt{L} := \{v_2, v_7\}$, $p = 2$, $\mathbf{v}_\texttt{P} := \{v_3\}$, $\mathbf{a}_\texttt{L}^{co} := \{agg(v_6, v_1, v_4)\}$, and $\mathbf{h}_\texttt{L}^{co} := \{\texttt{H}(r, v_1), \texttt{H}(r, v_4)\}$. Similarly, A sends $\mathbf{id}_\texttt{R} := \{id_1, id_4\}$, $\mathbf{v}_\texttt{R} := \{v_1, v_4\}$, $p = 3$, $\mathbf{v}_\texttt{P} := \{v_3\}$, $\mathbf{a}_\texttt{R}^{co} := \{agg(v_5, v_2, v_7)\}$, and $\mathbf{h}_\texttt{R}^{co} := \{\texttt{H}(r, v_2), \texttt{H}(r, v_7)\}$ to its right child node S_6.

In order to compute the required lists we specify two functions, SplitIdV and SplitAH, described in the following. The auxiliary function SplitIdV (Figure 5) is used by A to build the corresponding $\mathbf{id}_\texttt{L}$ and $\mathbf{v}_\texttt{L}$ as well as $\mathbf{id}_\texttt{R}$ and $\mathbf{v}_\texttt{R}$.

First to be mentioned is that SplitIdV is executed only if $n_s \geq 4$, that is if $\mathbf{id}[2]$ and $\mathbf{id}[3]$ have in turn, further child nodes. SplitIdV function splits the initial list of identities $\mathbf{id}$ into the sublists $\mathbf{id}_\texttt{L}$, and $\mathbf{id}_\texttt{R}$, and the initial list of original data values, $\mathbf{v}$, into the sublists, $\mathbf{v}_\texttt{L}$ and $\mathbf{v}_\texttt{R}$. Lists indexed with L: $\mathbf{id}_\texttt{L}$ and $\mathbf{v}_\texttt{L}$ contain identities and original data values of sensor nodes that have $\mathbf{id}[2]$

```
SplitAH(c, a, h, a_L^co, h_L^co, a_R^co, h_R^co) :
    if c ≥ 3 then a_R^co := a_R^co.a[2], a_L^co := a_L^co.a[3]
    else a_R^co := a_R^co.a[2], a_L^co := a_L^co.ε
    if c ≥ 7 then
        h_R^co := h_R^co.h[4].h[5], h_L^co := h_L^co.h[6].h[7]
    else if c ≥ 6 then
        h_R^co := h_R^co.h[4].h[5], h_L^co := h_L^co.h[6]
    else if c ≥ 5 then h_R^co := h_R^co.h[4].h[5]
    else if c ≥ 4 then h_R^co := h_R^co.h[4]
    return (a_L^co, h_L^co, a_R^co, h_R^co)
```

Fig. 6. SplitAH function.

and $\mathbf{v}[2]$ in their paths. Lists indexed with R contain identities and original data values of sensor nodes that have $\mathbf{id}[3]$ and $\mathbf{v}[3]$ in their paths. Recall that $\mathbf{id}[2]$ identifies the first child node, whereas $\mathbf{id}[3]$ identifies the second child node of A; thinking of a binary tree we may say *left* and *right* child nodes, hence, the indices L and R. The idea behind the SplitIdV function is to move along the reference lists, **id** and **v**, and insert their elements into either $\mathbf{id}_{\mathrm{L}}$ or $\mathbf{id}_{\mathrm{R}}$ and into $\mathbf{v}_{\mathrm{L}}$ or $\mathbf{v}_{\mathrm{R}}$, based on the condition $y < \frac{x}{2}$, which says whether $\mathbf{id}[x + y]$ has $\mathbf{id}[2]$ or $\mathbf{id}[3]$ in its path.

The SplitAH function (Figure 6) is used by A to compute lists of intermediate aggregation values, $\mathbf{a}_{\mathrm{L}}^{\mathrm{co}}$ and $\mathbf{a}_{\mathrm{R}}^{\mathrm{co}}$, as well as hash commitments, $\mathbf{h}_{\mathrm{L}}^{\mathrm{co}}$ and $\mathbf{h}_{\mathrm{R}}^{\mathrm{co}}$, in the copaths of its left and right child nodes. First, the SplitAH function is executed only if A has at least one child node (the left one). Therefore, the first check made during SplitAH is to see whether the second child node (the right one) exists. This is done by $c \geq 3$. In this case both lists $\mathbf{a}_{\mathrm{R}}^{\mathrm{co}}$ and $\mathbf{a}_{\mathrm{L}}^{\mathrm{co}}$ need to be updated; otherwise, only $\mathbf{a}_{\mathrm{R}}^{\mathrm{co}}$. The conditions of the form $c \geq \alpha$ for α from 7 to 4, point out how many grandchild nodes exist in order to ensure the correct contents of $\mathbf{h}_{\mathrm{L}}^{\mathrm{co}}$ and $\mathbf{h}_{\mathrm{R}}^{\mathrm{co}}$.

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 $(\mathbf{id}, \mathbf{v}, p, \mathbf{v}_{\mathrm{P}}, \mathbf{a}^{\mathrm{co}}, \mathbf{h}^{\mathrm{co}})$.

Before, S_i performs computations of the DOWNFLOW stage, it prepends id_i to **id** and v_i to **v**. 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. The equality $c_{\mathrm{P}} = \lfloor \log_2 p \rfloor$ ensures consistency between the node's position, p, and the number of nodes in its path. If any of these verifications fails, S_i sets its Boolean variable acc to false and turns directly into the VERIFICATION stage. 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, **a**, and hash commitments, **h**. Then S_i checks whether there are any further child nodes via the condition $c \geq 2$. If so, S_i splits **id** respectively **v** into $\mathbf{id}_{\mathrm{L}}$ and $\mathbf{id}_{\mathrm{R}}$ respectively $\mathbf{v}_{\mathrm{L}}$ and $\mathbf{v}_{\mathrm{R}}$ using the SplitIdV function, updates $\mathbf{a}_{\mathrm{L}}^{\mathrm{co}}$ and $\mathbf{a}_{\mathrm{R}}^{\mathrm{co}}$ respectively $\mathbf{h}_{\mathrm{L}}^{\mathrm{co}}$ and $\mathbf{h}_{\mathrm{R}}^{\mathrm{co}}$ based on the previously computed lists, **a** and **h**, using the SplitAH function, extends $\mathbf{v}_{\mathrm{P}}' := \mathbf{v}_{\mathrm{P}}.v_i$ (note that the received

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

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

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

5.4 Specification of the VERIFICATION Stage

As mentioned in the high level description, during 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 its own intermediate aggregation value, $\mathbf{a}[1]$, and its corresponding hash commitment, $\mathbf{h}[1]$. Furthermore, every S_i (and A) knows its own data value, v_i (and v_A), data values in its path, given by $\mathbf{v}_\mathrm{P}$, intermediate aggregation values in its copath, given by $\mathbf{a}^{\mathrm{co}}$, hash commitments in its copath given by $\mathbf{h}^{\mathrm{co}}$, as well as the aggregation result a^* and hash commitment its h^* from the broadcast message of R. Additionally, every S_i knows its 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. In addition, every S_i maintains a Boolean variable, acc, indicating whether the node will confirm the obtained final values or not. During the DOWNFLOW stage, acc could possibly be changed to *false*. Figure 8 describes calculations of S_i. According to the construction of $\mathbf{id}$ by A in the UPFLOW stage for every node $\mathbf{id}[p]$ with odd position $p > 1$, there exists a sibling node $\mathbf{id}[p - 1]$. However, if p is even, additional verification via $p + 1 \leq n_s$ becomes necessary to ensure that $\mathbf{id}[p + 1]$ exists. The 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 $\mathbf{id}[\lfloor p/2 \rfloor]$. If acc is already set to *false* no further checks are necessary and S_i replies to A with a negative acknowledgement in the form of

On input r, n_s, a^*, h^*, p, $\mathbf{a}$, $\mathbf{h}$, $\mathbf{v}_\mathtt{P}$, $\mathbf{a}^{\mathtt{co}}$, $\mathbf{h}^{\mathtt{co}}$, $\mathtt{acc}$, k_i every sensor node S_i proceeds as follows:

if $\mathtt{acc} = \mathit{true}$ then
  $a := \mathbf{a}[1]$, $h := \mathbf{h}[1]$, $c_p := |\mathbf{v}_\mathtt{P}|$, $c_h := |\mathbf{h}^{\mathtt{co}}|$
  while $c_p \geq 1$ do
    if p *even* then
      if $p+1 \leq n_s$ then
        $a := agg(\mathbf{v}_\mathtt{P}[c_p], a, \mathbf{a}^{\mathtt{co}}[c_p])$
        if $2(p+1)+1 \leq n_s$ then
          $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p], \mathbf{h}^{\mathtt{co}}[c_h - 1], \mathbf{h}^{\mathtt{co}}[c_h])$, $c_h := c_h - 2$
        else if $2(p+1) \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p], \mathbf{h}^{\mathtt{co}}[c_h - 1])$, $c_h := c_h - 1$
        else $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p])$
        $h := \mathtt{H}(r, a, h, \bar{h})$
      else $a := agg(\mathbf{v}_\mathtt{P}[c_p], a)$, $h := \mathtt{H}(r, a, h)$
    else
      $a := agg(\mathbf{v}_\mathtt{P}[c_p], \mathbf{a}^{\mathtt{co}}[c_p], a)$
      if $2(p-1)+1 \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p], \mathbf{h}^{\mathtt{co}}[c_h - 1], \mathbf{h}^{\mathtt{co}}[c_h])$, $c_h := c_h - 2$
      else if $2(p-1) \leq n_s$ then $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p], \mathbf{h}^{\mathtt{co}}[c_h - 1])$, $c_h := c_h - 1$
      else $\bar{h} := \mathtt{H}(r, \mathbf{a}^{\mathtt{co}}[c_p])$
      $h := \mathtt{H}(r, a, \bar{h}, h)$
    $c_p := c_p - 1$, $p := \lfloor \frac{p}{2} \rfloor$
  if $a \neq a^*$ or $h \neq h^*$ then $\mathtt{acc} = \mathit{false}$
if $\mathtt{acc} = \mathit{false}$ 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.

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 initially sets $a := \mathbf{a}[1]$ and $h := \mathbf{h}[1]$. Recall 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 respectively h to the aggregation value respectively hash commitment corresponding to the next position in its path using the auxiliary aggregation value $\mathbf{a}^{\mathtt{co}}[c_p]$ and hash commitment $\bar{h}$ from its copath. $\bar{h}$ is computed by S_i from the received commitments and $\mathbf{a}^{\mathtt{co}}[c_p]$, whereas $\mathbf{a}^{\mathtt{co}}[c_p]$ is taken directly from the parent node's message. It is easy to check that after the final iteration, a respectively h should (ideally) match, a^* respectively 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.The aggregator, does not need to recompute the final aggregation result and hash commitment, as it knows them since the UPFLOW stage, and has already compared them to the values received from R in the beginning of the DOWNFLOW stage. In the case, of mismatch, acc is already set to *false*. In this case, A sends an error, ERR, to R, together with its 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

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$(k_A, (r, \texttt{OK}))$, send $(\texttt{OK}, \mu_A)$ to R
  else if acc $= true$ and $n_s > 1$ then
    $\mu :=$ MAC.Sign$(k_A, (r, \texttt{OK}))$, $c := 1$, nxt $= true$, initialize timer t
    while $c < n_s$ and nxt $= true$ and t is not expired do
      receive new (m, μ_i)
      if $m = \texttt{OK}$ then $\mu := \mu \oplus \mu_i$, $c := c + 1$
      else if $m = \texttt{ERR}$ then send $(\texttt{ERR}, \mu_i)$ to R, nxt $= false$
    if nxt $= true$ and $c = n_s$ then send $(\texttt{OK}, \mu)$ to R
    else if nxt $= true$ and $c < n_s$ then
      $\mu_A :=$ MAC.Sign$(k_A, (r, \texttt{ERR}))$, send $(\texttt{ERR}, \mu_A)$ to R
  else if acc $= false$ then $\mu_A :=$ MAC.Sign$(k_A, (r, \texttt{ERR}))$, send $(\texttt{ERR}, \mu_A)$ to R

Fig. 9. VERIFICATION stage specification for aggregator A.

timer t and starts waiting for the acknowledgements of other nodes. A counts the number of received acknowledgements until every node has replied. In our protocol (unlike Chan et al. [2006]) any node S_i, can reply with the error message. In this case A simply aborts and forwards this error message and the MAC value μ_i, to R. Otherwise, A aggregates MAC values from all positive acknowledgements using the XOR function as in Chan et al. [2006] 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 error message.

Finally, we provide a description of the operations performed by R upon receiving the verification result (m, μ) from A. R accepts the aggregation result a^* only if $m = \texttt{OK}$ and the received value μ is valid: 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 error message $m = \texttt{ERR}$), R terminates without accepting. Note that at the end of the UPFLOW stage R has already verified that $B_a(a^*) = true$. It is easy to check that the proposed framework is correct according to Definition 2.4.

Remark 5.1. In Chan et al. [2006], 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 or not it still needs to wait for further acknowledgements. Furthermore, the solution in Chan et al. [2006] does not explicitly abort further protocol execution in cases where failures are identified before all nodes receive the required information and recompute the final hash value. By introducing authenticated error messages we can abort the protocol execution at any time (also during the DOWNFLOW process) saving further processing costs. Any node that identifies a failure aborts and reports an error to A. Its a failure is identified and reported by some parent node before the required information is sent to its child node(s), then sending this information becomes obsolete. Thus, error messages prevent wasting computation and communication costs.

5.5 Relationship between a Node's Position and the Total Number of Nodes

Recall that for the aggregation functions described in Section 3, in order to verify the output predicate B_a for some intermediate aggregation value $\mathbf{a}[p]$ it is necessary to know how many individual data inputs have been aggregated

to obtain $\mathbf{a}[p]$. Let n_p be the number of these inputs. In the following we show how to efficiently compute n_p based solely on the knowledge of the position p and the total number of nodes, n_s.

Definition 5.2 (*Relative Distance of Two List Elements*). Let $\mathbf{x}$ be a list of size n and $p, q \in [1, n]$ two positions from $\mathbf{x}$. The *relative distance* $\delta(p, q)$ between the list elements $\mathbf{x}[p]$ and $\mathbf{x}[q]$ is defined as:

$$\delta(p, q) := \lfloor \log_2 p \rfloor - \lfloor \log_2 q \rfloor.$$

Visualizing the list $\mathbf{x}$ as a binary tree, the relative distance $\delta(p, q)$ equals the difference between the levels of the vertices in that tree that correspond to the elements $\mathbf{x}[p]$ and $\mathbf{x}[q]$, for example, if $\delta(p, q) = 0$ then both vertices that represent elements $\mathbf{x}[p]$ and $\mathbf{x}[q]$ in the binary tree are located at the same level.

In order to compute n_p one first computes the relative distance $\delta(n_s, p) := \lfloor \log_2 n_s \rfloor - \lfloor \log_2 p \rfloor$ and two auxiliary values $p_r := (p + 1)2^{\delta}(n_s, p) - 1$ and $p_\ell := p2^{\delta}(n_s, p)$. Then, n_p can be estimated as follows:

if $p_r \leq n_s$ then $n_p := 2^{\delta(n_s,p)+1} - 1$
else if $p_\ell \leq n_s < p_r$ then $n_p := 2^{\delta(n_s,p)+1} - (p_r - n_s)$
else $n_p := 2^{\delta(n_s,p)}$.

For example, in Figure 1, given $n_s = 7$ and $p = 2$ we obtain $n_p = 3$, that is the intermediate aggregation value $\mathbf{a}[2]$ is the output of *agg* on 3 individual data inputs. Assuming that the tree is unbalanced such that $n_s = 4$ and $p = 2$ we obtain $n_p = 2$.

The computed value n_p is also useful for the following analysis of individual computation and communication costs of the nodes during the protocol execution.

5.6 Performance of $\texttt{InAP1}_{agg}$

In the following we analyze the exact and asymptotic complexity of computation and communication costs of the proposed framework. Although our evaluation of performance is theoretical, we stress that the framework seems to be practical enough to be deployed on the currently available hardware platforms for sensor nodes. For the practical evaluation of symmetric cryptographic primitives used in the framework we refer to the survey in Roman et al. [2007].

5.6.1 *Computation Costs.* Our analysis of computation costs is given from the perspective of the required cryptographic operations since their costs prevail over the simple calculations of the aggregation function *agg* and verification of the input and output predicates, B_v and B_a.

In the UPFLOW stage every node S_i, including the A, verifies the authenticated broadcast message of R. Assuming that the deployed mechanism is based on the message authentication codes (e.g., Perrig et al. [2002] and Liu and Ning [2004]) we consider costs for this verification equal to the computation of a MAC value. Additionally, A computes the hash commitment list, $\mathbf{h}$, or if a failure has occurred then the MAC value on the error message ERR. The computation of $\mathbf{h}$ requires $2n_s - 1$ executions of the hash function, H, where n_s denotes the total number of sensor nodes.

Table I. Asymptotic Computation Costs of $\texttt{InAP1}_{agg}$

	Computation Costs	
	Hash values	MAC values
$\texttt{InAP1}_{agg}$ UPFLOW		
Aggregator A	$O(n_s)$	$O(1)$
Sensor node S_i	—	$O(1)$
$\texttt{InAP1}_{agg}$ DOWNFLOW		
Aggregator A	—	$O(1)$
Sensor node S_i	$O(n_s)$	$O(1)$
$\texttt{InAP1}_{agg}$ VERIFICATION		
Aggregator A	—	$O(1)$
Sensor node S_i	$O(\log_2 n_s)$	$O(1)$
$\texttt{InAP1}_{agg}$ **Total**		
Aggregator A	$O(n_s)$	$O(1)$
Sensor node S_i	$O(n_s)$	$O(1)$

In the DOWNFLOW stage every sensor node needs to verify the authenticated broadcast message of the sink R, which corresponds to the computation of the MAC value. Additionally, every S_i, except for A, assigned to the position p, $1 < p \leq n_s$, in the identity list, **id**, computed by A must compute one hash value for each value in **id** that has **id**$[p]$ in its path, as part of the executed Commit function. The total number of such values corresponds to the value n_p computed from p and n_s using the relative distance $\delta(n_s, p)$, as described in Section 5.5.

In the VERIFICATION stage every node S_i, except for A, assigned to position p in **id** computed by A must compute one hash value for each position in its own path. The total number of these values is given by $\lfloor \log_2 p \rfloor$. Additionally, every S_i, including A, computes one MAC value.

Table I summarizes computation costs for $\texttt{InAP1}_{agg}$ in terms of its asymptotic complexity while considering the worst case for every type of the required computation costs. Obviously, per each sensor node the number of hash computations is linear in n_s and the number of the computed MAC values is constant.

5.6.2 *Communication Costs.* In a similar way we analyze the communication complexity of the proposed framework, $\texttt{InAP1}_{agg}$. In our analysis we focus on the total number of communication rounds (considering all messages that can be sent in parallel as part of the same round), as well as the total number of sent messages per sensor node and the total size of sent messages per sensor node. The total size of transmitted messages is given in the size of a single hash value, denoted $|h|$, assuming that it prevails over other information types, for example, measured data or sensor node identity. Moreover, the size of the hash value is usually similar to that of a MAC value since many message authentication codes (e.g., HMAC) are constructed based on the hash functions.

In the UPFLOW stage every sensor node S_i sends exactly one message. Only the aggregator's message to the sink contains the hash value (final hash commitment). In the DOWNFLOW stage there are at most $\lfloor \log_2 n_s \rfloor$ communication rounds. Furthermore, every sensor node S_i sends at most two messages, one to its left and one to its right child node. The total size of the messages sent by S_i assigned

Table II. Asymptotic Communication Costs of $\texttt{InAP1}_{agg}$

	Communication Costs		
	Rounds	Messages per S_i	Message size per S_i (in $\|h\|$)
$\texttt{InAP1}_{agg}$ UPFLOW	$O(1)$		
Aggregator A		$O(1)$	$O(1)$
Sensor node S_i		$O(1)$	—
$\texttt{InAP1}_{agg}$ DOWNFLOW	$O(\log_2 n_s)$		
Aggregator A		$O(1)$	$O(n_s)$
Sensor node S_i		$O(1)$	$O(n_s)$
$\texttt{InAP1}_{agg}$ VERIFICATION	$O(1)$		
Aggregator A		$O(1)$	$O(1)$
Sensor node S_i		$O(1)$	$O(1)$
$\texttt{InAP1}_{agg}$ **Total**	$O(\log_2 n_s)$		
Aggregator A		$O(1)$	$O(n_s)$
Sensor node S_i		$O(1)$	$O(n_s)$

to position p within identity list **id** computed by A, is given by up to n_p hash values, where n_p (computed using $\delta(n_s, p)$ as described in Section 5.5) is the total number of values in **id** that have **id**$[p]$ in their paths. Therefore, sensor nodes whose positions are closer to the aggregator node (i.e., **id**[1]) generally send larger messages than nodes located further away. In the VERIFICATION stage every node sends at most one MAC value.

Table II summarizes communication costs for $\texttt{InAP1}_{agg}$ in terms of its asymptotic complexity, based on the measurements for the worst case. The number of rounds is given per protocol stage, the number of messages, and their size per each sensor node. The protocol requires a logarithmic number of communication rounds and a constant number of messages with linear overall message size per each sensor node. The overall exact message size additionally depends on the position of the sensor node in identity list **id**, which in turn reflects the physical distance to the aggregator node.

5.7 Security of $\texttt{InAP1}_{agg}$

Security of our framework can be proved in the formal model from Section 2 using the classical cryptographic proving technique called *sequence of games* [Shoup 2006].

THEOREM 5.3. *Let* H *be collision-resistant and* MAC *secure in the sense of Definitions* 4.1 *and* 4.2. *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* $\texttt{InAP1}_{agg}$ *framework from Section 5 is optimally secure in the sense of Definition 2.7.*

PROOF. We define a sequence of games G_i, $i = 0, \ldots, 4$, with adversary $\mathcal{I}$ against the optimal security of $\texttt{InAP1}_{agg}$. In each game we denote Win_i, the event that $\mathcal{I}$ breaks the optimal security of $\texttt{InAP1}_{agg}$ (wins in $\mathsf{Game}^{\mathtt{opt-sec}}_{\texttt{InAP1}_{agg}}(\mathcal{I}, \kappa)$), that is there exists session s in which R^s accepts the aggregation result a^* and there exists NO list $\mathbf{v}_\mathrm{c}$ of size $n_\mathrm{c} = n_s - n_\mathrm{h}$ with $B_v(\mathbf{v}_\mathrm{c}) = \mathit{true}$ such that $a^* = agg(a_\mathrm{h}, \mathbf{v}_\mathrm{c})$. 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 2.7) 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* [Shoup 2006, Lemma 1]) one can upper-bound the total probability of a successful attack.

Game 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 2.7, where instances of all uncorrupted parties are replaced by the simulator, Δ, which has a global view, of all simulated computations. Δ also answers all queries of $\mathcal{I}$ as defined in the adversarial model. Some explanations follow. For example, upon asking $Send(R^s, \mathtt{'start'}, \mathcal{S}^s, \mathrm{A}^s)$ the simulator creates a new instance of R, which is supposed to execute the protocol with the nodes in $\mathcal{S}$ and aggregator A. The simulator, Δ, then acts according to the protocol specification: it chooses a random nonce r, sets $n_s := |\mathcal{S}|$, and returns an authenticated broadcast message (r, n_s, α) to $\mathcal{I}$, where α denotes the authentication part of the message. In order to invoke the protocol at the instances of the sensor nodes in $\mathcal{S}$, the adversary queries Δ with a separate $Send(S_i^s, \mathtt{'start'}, \mathcal{S}^s, \mathrm{A}^s, R^s, (r, n_s, \alpha))$ query for each instance S_i^s. This query contains (r, n_s, α) needed by S_i^s in order to start with the UPFLOW stage. In this way $\mathcal{I}$ can execute the protocol with the simulator. In order to recognize possible attacks in the later games of the sequence (in particular to recognize possible forgeries of authenticated messages in Games G_2 and G_3) we assume that the simulator keeps track of each message returned to $\mathcal{I}$ on behalf of any honest party. Additionally, Δ should keep track of every computed hash value in order to be able to find possible hash collisions in Game G_4.

Of course, $\mathcal{I}$ is not restricted to executing the protocol honestly, to just forwarding, messages in the form of its *Send* queries. For example, $\mathcal{I}$ could ask $Send(S_j^s, \mathtt{'start'}, \mathcal{S}^s, \mathrm{A}^s, R^s, (r', n_s, \alpha))$ for one of the instances, S_j^s, where the original random nonce r is replaced by some other value r'. In this way the adversary can try to mount an attack on the honest protocol execution. Additionally, the adversary may ask a $Corrupt(S_i)$ query at any time. In response to it, $\mathcal{I}$ receives from Δ every secret value known to S_i—in particular its keys used to produce valid MAC values; note that Δ initially knows all secrets of every sensor node and of R and is, therefore, able to answer the query. From that time, point Δ stops simulating all instances of the corrupted node S_i, since its behavior is then controlled by $\mathcal{I}$.

These were just examples to show the relationship between the queries defined in the model and this security proof. The idea behind the sequence of games technique is to consider this initial simulation and make small changes to this simulation in order to exclude all potential attacks. These changes appear in the form of further games. Finally, we mention that in order to mount a successful attack, R must *accept* the biased aggregation result. The event of the simulation failure used in subsequent games of the proof means that the simulation will not finish, and thus R will not accept, so that no attack occurs. In fact the probability given in each game is the probability that the

simulation does not fail in that game and this probability in turn is bound to some specific action of the adversary. In fact all relevant actions of the adversary can be eliminated so that the probability of the successful attack in the final game becomes zero.

Game G_1. This game is identical to Game G_0 with the only exception being 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 bounded by ${q_\mathrm{s}}^2/2^\kappa$. Hence,

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

This game implies that as long as R remains uncorrupted its choices of r are always different in this and all subsequent games of the proof.

Game G_2. This game is identical to Game G_1 with the only exception being 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 $\mathsf{Succ}_{\mathtt{BC}}^{\mathtt{uf}}(\kappa)$ denote the maximal probability of the successful forgery attack on the applied broadcast authentication mechanism, BC. By assumption $\mathsf{Succ}_{\mathtt{BC}}^{\mathtt{auth}}(\kappa)$, is negligible. Considering two broadcast messages in each session we get:

$$|\Pr[\mathsf{Win}_2] - \Pr[\mathsf{Win}_1]| \leq 2q_\mathrm{s}\mathsf{Succ}_{\mathtt{BC}}^{\mathtt{auth}}(\kappa). \tag{2}$$

Having excluded collisions of random nonces and attacks against the broadcast messages of R, this game excludes any forgeries and replay attacks on the messages of R. Thus, in this and all subsequent games of the proof every authenticated message of R received by the nodes is neither injected, nor generated, by the adversary, but is the original message sent by R.

Game G_3. This game is identical to Game G_2 with the only difference being 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 $\mathcal{I}$. Hence, the simulation fails if $\mathcal{I}$ output a successful forgery $(\mathtt{OK}, \mu_i)$. The simulator can notice whether or not a forgery has occurred, since it keeps track of all messages that it generated on behalf of the uncorrupted parties. Considering EUF-CMA security of MAC and at most n_s participating sensor nodes and q_s protocol sessions, we obtain:

$$|\Pr[\mathsf{Win}_3] - \Pr[\mathsf{Win}_2]| \leq n_s q_\mathrm{s}\mathsf{Succ}_{\mathtt{MAC}}^{\mathtt{euf-cma}}(\kappa). \tag{3}$$

Similar to Game G_2, this game excludes any forgeries and replay attacks on the acknowledgements of sensor nodes.

Game G_4. This game is identical to Game 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 collision-resistance of $\mathtt{H}$ and at most n_s computed hash commitments for each executed session, we obtain:

$$|\Pr[\mathsf{Win}_4] - \Pr[\mathsf{Win}_3]| \leq n_s q_\mathsf{s} \mathsf{Succ}^{\mathtt{cr}}_{\mathtt{H}}(\kappa). \tag{4}$$

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 $\mathbf{v}$, $\mathbf{v}_\mathtt{P}$, and $\mathbf{a}^{\mathtt{co}}$ 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 in the aggregation result, a^*, and all checked predicates are *true*. Successful verification of predicates implies that for $a_\mathtt{c}$ corresponding to the aggregation value of all adversarial inputs, $B_a(a_\mathtt{c}) = \mathit{true}$ should hold. Hence, due to the correctness property of *agg*, there exists a tuple $\mathbf{v}_\mathtt{c}$ of size $n_s - n_\mathtt{h}$ such that $B_v(\mathbf{v}_\mathtt{c}) = \mathit{true}$. 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:

$$\mathsf{Succ}^{\mathtt{opt-sec}}_{\mathtt{InAP1}_{agg}}(\kappa) \leq \frac{{q_\mathsf{s}}^2}{2^\kappa} + 2q_\mathsf{s}\mathsf{Succ}^{\mathtt{uf}}_{\mathtt{BC}}(\kappa) + n_s q_\mathsf{s}\mathsf{Succ}^{\mathtt{euf-cma}}_{\mathtt{MAC}}(\kappa) + n_s q_\mathsf{s}\mathsf{Succ}^{\mathtt{cr}}_{\mathtt{H}}(\kappa),$$

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

6. CONCLUSIONS AND FUTURE WORK

Along the lines of this article, we have presented a formal communication and security model as well as a novel framework for in-network aggregation in WSNs, focusing on the single aggregator scenario. The rigorously derived security model provides formal definition of the optimal security requirement introduced earlier in Chan et al. [2006]. Our $\mathtt{InAP1}_{agg}$ framework is provably secure in the cryptographic sense, and also seems practical. The practical relevance results from the use of symmetric cryptographic primitives whose computation is supported by the technology in today's sensor nodes [Roman et al. 2007]. Future work on the framework may address extended evaluation of performance, for example, power consumption and simulation.

Another advantage of our security model and framework is the modular construction, which provides, basis for further extensions (e.g., towards a hierarchical scenario [Chan et al. 2006] or concealed data aggregation processes [Castelluccia et al. 2005; Westhoff et al. 2006]). The abstract definition of the aggregation function *agg* and its input and output predicates (B_v and B_a) allow tailoring the specification of the integrity checks that become necessary for the optimal security of the aggregation process.

REFERENCES

CASTELLUCCIA, C., MYKLETUN, E., AND TSUDIK, G. 2005. Efficient aggregation of encrypted data in wireless sensor networks. In *Proceedings of the International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*. IEEE CS, 109–117.

CHAN, H., PERRIG, A., AND SONG, D. 2006. Secure hierarchical in-network aggregation in sensor networks. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, Internation (CCS)*. ACM, 278–287.

DESHPANDE, A., NATH, S. K., GIBBONS, P. B., AND SESHAN, S. 2003. Cache-and-query for wide area sensor databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 503–514.

ESTRIN, D., GOVINDAN, R., HEIDEMANN, J. S., AND KUMAR, S. 1999. Next century challenges: scalable coordination in sensor Networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MOBICOM)*. 263–270.

HU, L. AND EVANS, D. 2003. Secure aggregation for wireless network. In *Symposium on Applications and the Internet Workshops (SAINT)*. IEEE Computer Society, 384–394.

INTANAGONWIWAT, C., ESTRIN, D., GOVINDAN, R., AND HEIDEMANN, J. S. 2002. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the International conference on Distributed Computing Systems (ICDCS)*. 457–458.

LIU, D. AND NING, P. 2004. Multilevel μTESLA: broadcast authentication for distributed sensor networks. *ACM Trans. Embed. Comput. Sys. 3,* 4, 800–836.

MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. 2002. TAG: A Tiny AGgregation Service for ad-hoc sensor networks. In *Proceedings of the USENIX Symposium on Operating systems Design and Implementation (OSDI)*.

MERKLE, R. C. 1990. A certified digital signature. In *Advances in Cryptology (CRYPTO '89)*. Lecture Notes in Computer Science, vol. 435. Springer, 218–238.

PERRIG, A., CANETTI, R., TYGAR, J. D., AND SONG, D. 2002. The TESLA broadcast authentication protocol. *RSA CryptoBytes 5,* Summer.

PERRIG, A., SZEWCZYK, R., WEN, V., CULLER, D. E., AND TYGAR, J. D. 2001. SPINS: security protocols for sensor networks. In *Proceedings of the Internationl Conference on Mobile Computing and Networking (MOBICOM)*. 189–199.

PRZYDATEK, B., SONG, D. X., AND PERRIG, A. 2003. SIA: Secure information aggregation in sensor networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 255–265.

ROMAN, R., ALCARAZ, C., AND LOPEZ, J. 2007. A Survey of cryptographic primitives and implementation for hardware-constrained sensor network nodes. *Mobile Netw. Appl. 12,* 4 (August), 231–244.

SHOUP, V. 2006. Sequences of games: A tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332. `http://eprint.iacr.org/2004/332.pdf`.

SIRIVIANOS, M., WESTHOFF, D., ARMKNECHT, F., AND GIRAO, J. 2007. 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)*. IEEE Computer Society. `http://www.ics.uci.edu/~msirivia/publications/sane-fullpaper.pdf`.

WESTHOFF, D., GIRAO, J., AND ACHARYA, M. 2006. Concealed data aggregation for reverse multicast traffic in sensor networks: encryption, key distribution, and routing adaptation. *IEEE Trans. Mobile Comput. 05,* 10, 1417–1431.

Received July 2007; revised December 2007, March 2008; accepted May 2008