Modelling and Analysing a WSN Secure Aggregation Protocol: A Comparison of Languages and Tool Support

Volkan Cambazoglu
Johannes Åman Pohjola
Ramūnas Gutkovas
Björn Victor

Abstract

A security protocol promises protection of a significant piece of information while using it for a specific purpose. Here, the protection of the information is vital and a formal verification of the protocol is an essential step towards guaranteeing this protection. In this work, we study a secure aggregation protocol (SHIA) for Wireless Sensor Networks and verify the protocol in three formal modelling tools (Pwb, mCRL2 and ProVerif). The results of formal verification heavily depend on the model specification and the ability of the tools to deal with the model. Among the three tools, there is difference in data representation, communication types and the level of abstraction in order to represent SHIA. ProVerif and mCRL2 are mature and well-established tools, geared respectively towards security and distributed systems; however, their expressiveness constrains modelling SHIA and its security properties. Pwb is an experimental tool developed by the authors; its relative immaturity is offset by its increased expressive power and customisability. This leads to different models of the same protocol, each contributing in different ways to our understanding of SHIA's security properties.

Contents

1 Introduction **2**
1.1 Data Collection in WSN 3

2 The Secure Hierarchical In-network Aggregation Protocol **3**
2.1 Protocol Model 6
2.1.1 Sensor Nodes 7
2.2 Attacker Model 8

3 Languages and Tools **9**
3.1 Psi-Calculi and Pwb 9
3.2 Algebra of Communicating Processes and mCRL2 13
3.3 The Applied pi calculus and ProVerif 14

4 Modelling and Verifying SHIA **15**
4.1 Psi-Calculi Workbench 15
4.1.1 Model 15
4.1.2 Verification 17
4.1.3 Results 18
4.2 mCRL2 18
4.2.1 Model 18
4.2.2 Divergence from the Protocol Model 22
4.2.3 Verification 22
4.2.4 Usability and Experience 23
4.2.5 Results 23
4.3 ProVerif 23
4.3.1 Model 1 24
4.3.2 Model 2 26
4.3.3 Result 27

5 Pwb Development **28**

6 Related Work **30**

7 Conclusions and Future Work **31**

1 Introduction

A communication protocol consists of a set of rules that defines the type of data that will be exchanged, how this data will be represented in a syntax, the semantics behind it and the type of communication. Data is at the core of a communication protocol and is exchanged in form of messages. The syntax of the communication protocol defines its structure of representing the data and all other necessary control information in a message that will be sent from one side to the other. The semantics of the communication protocol defines the operation to handle a message; thus a receiver can understand what the message includes and what to do with it. For example, in a file transfer protocol, the receiver should be able to figure out which part of the file is received and whether it should wait for more parts or build the file from the parts by observing the control information that comes with the message. The type of the communication defines the reliability of the communication and the communication primitives that are required for the protocol. For instance, a file transfer protocol runs on a reliable point-to-point connection.

Formal verification is a way of certifying the description of such a protocol by using formal methods that are based on mathematical specification. The rules of the protocol (data, syntax, semantics, communication primitives) are expressed with a certain formal specification and, then, the specification is analysed for the properties of the protocol. When the formal specification of a protocol is analysed, intrinsic properties of the protocol can be identified clearly. These properties belong to the internal logic and operation of the protocol such as logical requirements that need to be fulfilled to reach a certain state of the protocol. These properties play a crucial role in verifying the correctness of the protocol.

In this work, we formally model and verify a secure aggregation protocol for Wireless Sensor Networks (WSN), the *Secure Hierarchical In-Network Aggregation* protocol (henceforth SHIA) by Chan et al [14]. In addition to modelling SHIA to verify its security, we also consider suitability of three different languages and tools for this goal. The tools that are used are Psi-Calculi Workbench (Pwb), mCRL2 and ProVerif. We choose Pwb, because it is expressive and customisable, and we aim to develop it further. mCRL2 is well used for modelling and verifying distributed applications, while ProVerif for security applications; however both tools have not been tried for WSN and using broadcast communication. As each of the three tools targets a different application, they do not have a large intersection on representing the model of SHIA. Fundamentally, these tools differ in data representation, communication types and level of abstraction in modelling SHIA. These differences lead to slightly different models of the same protocol, where Pwb comes on top in representing the model of SHIA completely, and mCRL2 and ProVerif come short in representing broadcast communication and recursive data structures in the model. Even after coming up with different ways of covering these shortcomings in mCRL2 and ProVerif, we are unable to obtain interesting results to the queries we made to these tools. These different models and their justifications will be explained in this paper. Considered from the different perspectives of three formal modelling tools, we could not observe any situation that implies SHIA's insecurity.

The modelling of the SHIA protocol was a significant driver for improving the Pwb tool. The result of this is a solid first-order algebraic specification

for the term language of the processes with battle-tested improvements on the interface with SMT solvers, pragmatic trade-offs on improving the constraint generation and solving run-times, and bringing the implementation of communication primitives closer to what is required by the protocol.

Before going into the details of the secure aggregation protocol, we first present a short introduction to the context of the protocol.

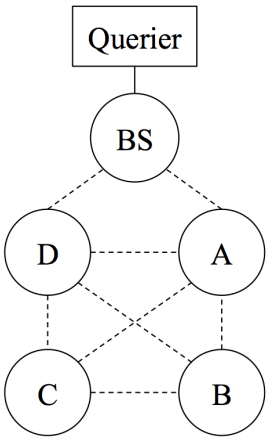
1.1 Data Collection in WSN

A Wireless Sensor Network (WSN) serves for a querier, who wants to obtain values from a specific region or area where the WSN is deployed. The WSN consists of resource constrained sensor nodes and at least one base station, which is generally more resource rich than the other nodes and acts as a gateway between the WSN and the querier. In this setting, efficiency is of high concern and the most expensive operation for a sensor node is transmitting a message. Arranging the WSN in a spanning tree structure and using in-network data aggregation for obtaining values from the WSN are ways of reducing the number of messages exchanged in the network [41]. These features allow to limit the number of neighbours a node has and also the number of messages a node has to transmit. In some cases, the values that are aggregated in the WSN need to be protected such that they are kept confidential or intact or both.

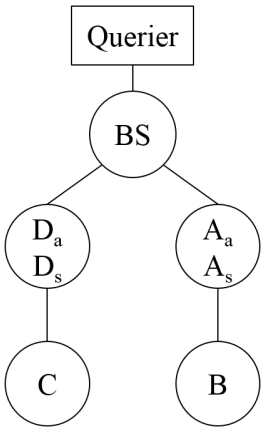
In the next section, we will describe the SHIA protocol and how we modelled it. Then we will introduce the three modelling tools and their languages in Section 3. How the SHIA model is developed in these tools will be explained in Section 4. Our efforts in developing Pwb to improve its applicability and reflect the SHIA model as close as possible will be described in Section 5. Related work will be presented in Section 6. Our conclusions and what can be done for future work will be shared in Section 7.

2 The Secure Hierarchical In-network Aggregation Protocol

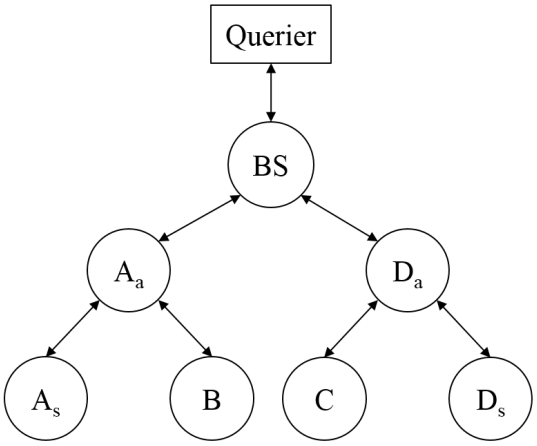
SHIA is a secure protocol that aggregates data in a wireless network by creating a hierarchy among sensor nodes. In-network data aggregation is a technique that creates a spanning tree, Figure 1b, based on a network topology, Figure 1a, in order to associate neighbouring nodes that will send data towards the root of the tree and aggregate data on the go. With this technique, each node sends only one message, which contains aggregated data, to its parent in the tree. Aggregation of data can be done in different ways depending on the type, however, the idea is combining multiple data into one. SHIA [14] provides security on top of this technique by introducing authentication and integrity. SHIA achieves these security goals by using commitments which are aggregated over a virtual binary tree structure, Figure 1c, on top of the aggregation tree, thus it has two levels of abstraction from the physical WSN. Overall SHIA has three phases: (1) query dissemination, (2) aggregation commit, and (3) result checking.



(a) Network topology



(b) Aggregation tree



(c) Commitment tree

Figure 1: SHIA organizes the nodes of the WSN, Figure 1a, into a hierarchical binary commitment tree, Figure 1c

1. Query dissemination: One round of the protocol starts with the querier sending a query to all of the sensor nodes to start sensing and aggregation in the WSN. The query is sent via authenticated broadcast [34], which adds to the security of SHIA by allowing identity confirmation.

2. Aggregation commit: Once the query reaches all of the sensor nodes, a binary commitment tree is constructed iteratively so that the sensed values will be aggregated towards the querier via the tree. In the terminology of [14], a *node* is a real wireless sensor node and a *vertex* is an entity with sensing or aggregating role in the binary commitment tree. Each sensing vertex places its value in a label and sends it to the associated aggregating vertex in the commitment tree. An aggregating vertex can be one physical node or can co-exist with a sensing vertex within the same physical node. For example, in Figure 1b, node A is both sensing and aggregating within one physical sensor node, while node B is only sensing. Each aggregating vertex receives labels from its children, aggregates them into one label and sends it to its parent. Here, an aggregated label contains not just the aggregation of two labels but also a commitment to the protocol, which supports the security of SHIA. The aggregation continues until the base station, which creates the label of the root of the tree and passes it to the querier.

3. Result checking: After receiving the label of the root of the tree, the querier starts the distributed result checking phase by sending the root label to the whole network. As the label of the root of the tree is disseminated downwards in the tree, each aggregating vertex should also disseminate off-path labels downwards. Off-path labels of a vertex consist of all the labels of siblings from every other branch on the path from the vertex to the root of the tree. For example, in Figure 1c, the off-path labels of vertex A_s are labels of vertices B and D_a. When a sensing vertex receives the label of the root of the tree and all of its off-path labels, it checks whether its contribution in the final aggregate is in place or not; therefore, the off-path labels also contribute to the security of SHIA. If the verification succeeds, each sensing vertex sends an okay message to its parent, which also aggregates these messages and send towards the querier. When the querier receives the final okay message that represents the whole network, the querier accepts the final aggregate; otherwise the aggregation fails.

SHIA fulfils its security goal during these phases of the protocol by achieving *optimal security*, which was defined by Chan et al. [14] as follows:

Definition 1. *(Definition 2 in [14]) An aggregation algorithm is* optimally secure *if, by tampering with the aggregation process, an adversary is unable to induce the querier to accept any aggregation result which is not already achievable by direct data injection.*

Definition 2. *(Definition 1 in [14]) A* direct data injection *attack occurs when an attacker modifies the data readings reported by the nodes under its direct control, under the constraint that only legal readings in* $[0, r]$ *are reported.*

There are three properties that support the optimal security of SHIA: (1) authenticated broadcast, (2) off-path labels, and (3) boundary checking.

Authenticated broadcast originates from the querier and is sent to every node in the network. Each node that receives an authenticated broadcast can check

whether the source is the querier or someone else; thus, the nodes take action for query and result-checking only after they make sure that the querier is the one that requests them. If the authenticated broadcast does not exist, an adversary can trick the sensor nodes into sensing and aggregating as if there is a query from the querier or disturb the actual result checking phase.

Off-path labels are components that are used in the result checking phase while building the label of the root of the tree in a bottom-up fashion. Each sensing vertex can compute the final aggregate by starting with its label and aggregating each off-path label on the way towards the root of the tree. If and only if all of the labels are the actual ones, then a node should be able to successfully verify that its contribution to the final aggregate is in place. For example, if vertex B, in Figure 1c, needs to check that its contribution is included in the final aggregate, it needs to compute the label of vertex A_a by aggregating its own label with the label of vertex A_s, and then compute the final aggregate by bringing the labels of vertices A_a and D_a together. If the computed final aggregate is the same as the received final aggregate, then vertex B confirms that its contribution is included correctly in the aggregation.

Boundary checking is about setting the minimum and the maximum values that can be sensed and aggregated in the commitment tree, and also making sure that no value is out of those limits. After being queried, each sensing vertex creates a label of $\langle count, value, complement, ID\rangle$, where *count* is one, *value* is the sensed value, *complement* is $(r - value)$, r is the maximum allowed value and ID is the identifier of the node. The minimum allowed value is zero; thus, the sensed values and their complements have to be non-negative at all times. When an aggregating vertex receives labels from its children, it creates a new label in which the label of the children vertices are aggregated such as

$$\begin{aligned} \text{Sensing vertices:} \quad & A_s \rightarrow A_a : \langle 1, v_{as}, \bar{v}_{as}, ID_a\rangle \\ & B \rightarrow A_a : \langle 1, v_b, \bar{v}_b, ID_b\rangle \\ \text{Aggregating vertex:} \quad & A_a \rightarrow BS : \langle 2, v_{aa}, \bar{v}_{aa}, H[N, 2, v_{aa}, \bar{v}_{aa}, A_s, B]\rangle \end{aligned}$$

where the aggregation operation is addition, $v_{aa} = v_{as} + v_b$, $\bar{v}_{aa} = \bar{v}_{as} + \bar{v}_b$, H is a hash function and N is nonce of the round. The hash value in the aggregating vertex's label is a way of commitment to the protocol since it allows verification of the values afterwards. When the querier receives the label of the root of the tree, it checks that the final value and complement are non-negative and their sum equals to $n * r$ where n is the total number of sensor nodes in the network. These boundaries allow to restrict the magnitude of the impact that an adversary can have on the final aggregate.

2.1 Protocol Model

In SHIA, there are two major types of entities: the querier and the sensor nodes. The querier is straightforward and has only one role, which is to communicate with the sensor nodes. However, the sensor nodes are more complicated and their roles need to be outlined clearly here.

2.1.1 Sensor Nodes

First of all, the base station is interpreted as a sensor node, though it may be more resource rich than the rest of the nodes. It conceptually does data aggregation as any other sensor node and forwards the final aggregation to the querier.

The model for sensor nodes is divided into two: (1) LeafVertex and (2) InternalVertex.

1. LeafVertex is the process model of leaf vertices in the binary commitment tree of SHIA. These vertices have *sensing* role and need to communicate with an InternalVertex process.

2. InternalVertex is the process model of vertices that are located between leaf vertices and the querier in the binary commitment tree of SHIA. These vertices have *aggregation* role and need to carry communication from LeafVertex processes to the Querier process and vice versa.

The protocol model involves the Querier, the LeafVertex and the InternalVertex processes. These processes communicate with each other via channels in which they input (receive) or output (send) their messages. The channels between processes are treated in an abstract way; hence, a channel could represent both an internal channel within a wireless sensor node and the wireless medium that the sensor nodes uses to communicate with each other. We assume that a binary commitment tree for a round of SHIA is built; therefore, we model the interaction between the querier and the sensor node processes over this tree. Each vertex of the tree represents a process and each edge represents a channel that two processes can communicate over it.

In addition to the communication between processes, it is necessary to model specific features of SHIA; such as types of messages, authenticated messages, aggregation of data and secure result-checking.

We tag the channels to differentiate their uses. In particular, we define *Query*, *Verify* and *Offpath* tags to relate to the three phases of the SHIA protocol. The *Query* tag is only used at the beginning of a round of SHIA, when the querier disseminates a nonce for starting the aggregation session to the WSN. When the querier wants to start the result checking phase, it disseminates the final aggregate to the WSN using the *Verify* tag. The verify tag is used from the beginning of the result checking phase until the end of it, so it carries the final aggregate to every vertex in the tree and also the MACs from every vertex to the querier. The *Offpath* tag is used for carrying the exchanged off-path labels between vertices and also not to confuse these labels with the label of the root of the tree as all of these labels take place in the communication during the result checking phase.

Authenticated broadcast is modelled as one hop communication from the querier to all of the vertices via a separate channel. This is an abstraction from the real authenticated broadcast that is suggested in SHIA to be achieved with μTESLA. The main reason for this abstraction is μTESLA is a symmetric key based protocol and every recipient can verify the source of the broadcast, i.e. the querier. Moreover, the number of hops that are necessary to reach all of the vertices should not affect the content of the authenticated broadcast. Therefore, a separate channel represents the symmetric key based authentication between the querier and all the vertices, and one hop communication represents the

nature of the broadcast.

The data is aggregated using a recursive label structure. A basic label consists of a sensed value and when two labels are aggregated, there is one resulting label. As a simple example, a label could be either $Label(SensedValue)$ or $AggrLabel(LabelLeft, LabelRight)$, where AggrLabel takes two labels from children vertices in the tree and aggregates them into one label. Therefore, the label of the root of the tree reflects the structure of the tree instead of representing just the final aggregated value.

The result checking phase of SHIA depends on distributed security by aggregating MACs from all leaf vertices. This is modelled similar to the label structure such that a leaf vertex provides $MAC(nonce, key)$ to its parent and the parent provides $XOR(macLeft, macRight)$ to its parent. Here, *nonce* is a one time bit sequence for a round of SHIA, *key* is the symmetric key that is specific to each leaf vertex and shared with the querier, and *XOR* represents the exclusive or operation that takes two MACs and outputs one MAC.

2.2 Attacker Model

The assumed attacker aims to trick the querier into accepting a different value than the actual aggregate. The attacker might also try to prevent the aggregation from happening as in a denial of service attack; however, we focus on the case in which the attacker tries stealthy attacks on SHIA. In this case, the importance is on how SHIA manages to run with dishonest participants. The stealthy attacker may have physical control over an arbitrary number of sensor nodes in the network. After taking control of them, the attacker has knowledge of the secret keys of the sensor nodes and can try to tamper with the aggregation. The attacker could also attack by capturing messages, modifying them or even injecting messages, and once again try to disturb the final aggregate.

We model these two types of attackers in the following ways:

- An attacker who has physical access to a sensor node, is modelled as a LeafVertex process with different parameters. The captured sensor node will try to join in the aggregation as every other sensor node and not reveal its captured status. For example, if all the sensor nodes would sense around 20 °C under normal conditions and the attacker would like to lower the aggregate, then the captured node contributes in the aggregation with 0 °C.

- An attacker who captures, modifies and injects messages via the wireless medium, is modelled as a gap in the SHIA model such that one or more sensor nodes are not defined in the network and the attacker decides which message to send to where as whom. For example, an InternalVertex process is missing in the executed model, so the aggregation and the communication, which the missing InternalVertex should have done, happens according to the attacker's will. In this case, the querier is not aware of absence of the sensor nodes; thus, the attacker targets certain nodes in the network, blocks their communication with other nodes and acts as those nodes. For instance, the attacker targets a node that aggregates in the network, so it will capture the aggregated value and send a false aggregate instead to the expecting parent.

3 Languages and Tools

Modelling and verification of communication protocols require formal languages and tools that can capture the interaction between processes that execute the protocol. The goal of this study is to explore and compare the capabilities and complexity of three formal languages involved in the verification of SHIA. The three tools are Psi-Calculi Workbench (Pwb), mCRL2 and ProVerif. We briefly introduce their capabilities and features that are related to modelling the SHIA protocol in this section. By this way we cover possible ways of representing the data, the operations and the process interaction that is necessary to reflect the protocol model we envision for SHIA.

The three tools that are used in this study are all based on process algebras, which are prototype specification languages for reactive systems [3]. In these systems, there are processes that communicate and interact with each other. These processes have states and the interaction happens through transitions between different states. Furthermore, the transitions have well defined structural operational semantics and take place around a logic. While based on this (very high level) process theory, the three tools differ in their process algebras, thus they have different operational semantics, logic, data representation and evaluation framework. All of these tools have their strengths, hence they are chosen for this study: Pwb is an expressive and customizable tool to model a large set of protocols, mCRL2 has a well documented and a rich specification language for analysing distributed systems and protocols, and ProVerif is a well known tool for modelling security protocols.

3.1 Psi-Calculi and Pwb

Psi-calculi [10] is a parametric semantic framework based on the pi-calculus [30], adding the possibility to tailor the data language and logic for each application. The framework provides a variety of features, such as lexically scoped local names for resources, communication channels as data, both unicast and broadcast communication [13], and both first- and higher-order communication [33].

The Psi-Calculi Workbench (Pwb) [12] is a generic tool for implementing psi-calculus instances, and for analysing processes in the resulting instances. It has a wider scope than previous works, and also allows experimentation with new process calculi with a relatively low effort. Pwb is parametric so that it provides functionality for bisimulation equivalence checking and symbolic simulation (or execution) of processes in any psi instance, and a base library for implementing new psi-calculi instances. Pwb thus has two types of users: the user analysing systems in an existing instance of the framework, and the instance implementer.

The following is a quick recapitulation of a simplified version of the psi-calculi framework. The full framework features dynamic environments, meaning that the validity of logical formulas may change during the execution of a process; this feature will not be used in the present paper and is hence not presented. For an in-depth introduction with motivations and examples we refer the reader to [10].

A psi-calculus is defined by instantiating two nominal data types and two equivariant operators. Before explaining the parameters of a psi-calculus, it might be helpful to remind about nominal data types; as names are at the core

of a psi-calculus.

Names $\mathcal{N}$ are atomic elements of a countably infinite set that is ranged over by $a, b, \ldots, z$. Intuitively, names are the symbols that can be scoped and be subject to substitution. A *nominal set* [35, 22] is a set equipped with a formal notion of what it means to swap names in an element; this leads to a notion of when a name a occurs in an element X, written $a \in \mathrm{n}(X)$ (pronounced "a is in the support of X"). We write $a\#X$, pronounced "a is fresh for X", for $a \notin \mathrm{n}(X)$, and if A is a set of names we write $A\#X$ to mean $\forall a \in A \,.\, a\#X$. In the following $\tilde{a}$ is a finite sequence of names. The empty sequence is written ϵ and the concatenation of $\tilde{a}$ and $\tilde{b}$ is written $\tilde{a}\tilde{b}$. We say that a function symbol is *equivariant* if all name swappings distribute over it.

A *nominal data type* is a nominal set together with a set of functions on it. In particular we shall consider substitution functions that substitute elements for names. If X is an element of a datatype, $\tilde{a}$ is a sequence of names without duplicates and $\tilde{Y}$ is an equally long sequence of elements of possibly another datatype, the *substitution* $X[\tilde{a} := \tilde{Y}]$ is an element of the same datatype as X. The substitution function can be chosen freely, but must satisfy certain natural laws regarding the treatment of names; it must be equivariant, and the names $\widetilde{a}$ in $X[\widetilde{a} := \widetilde{T}]$ must be alpha-convertible as if they were binding in X. See [10] for details.

Definition 3 (Psi-calculus parameters)**.** *A psi-calculus requires the two (not necessarily disjoint) nominal data types: the (data) terms* $\mathbf{T}$*, ranged over by* M, N*, the conditions* $\mathbf{C}$*, ranged over by* φ*, and the two operators:*

$$\begin{array}{ll} \leftrightarrows \in \mathbf{T} \times \mathbf{T} \to \mathbf{C} & \textit{Channel Equivalence} \\ \vdash \in \mathbf{C} \to \{\mathbf{true}, \mathbf{false}\} & \textit{Entailment} \end{array}$$

Channel equivalence will be written in infix. Thus, $M \leftrightarrows N$ is a condition, pronounced "M and N are channel equivalent". We write $\vdash\varphi$ for $\vdash(\varphi) = \mathbf{true}$.

We impose certain requisites on the sets and operators: channel equivalence must be symmetric and transitive, and substitution $M[\widetilde{a} := \widetilde{T}]$ on terms must be such that if the names $\widetilde{a}$ are in the support of M, the support of $\widetilde{T}$ must be in the support of $M[\widetilde{a} := \widetilde{T}]$.

We assume a given set of process identifiers, ranged over by A, and assume that each process identifier is associated with a *clause* $A(\widetilde{x}) \Leftarrow P$, where $\widetilde{x}$ binds into P and $\mathrm{n}(P) = \widetilde{x}$.

Definition 4 (Psi-calculus agents)**.** *Given a psi-calculus with parameters as in Definition 3, the* agents*, ranged over by* $P, Q, \ldots$*, are of the following forms.*

$\mathbf{0}$	Nil
$\overline{M}N \,.\, P$	Output
$\underline{M}(\lambda\widetilde{x})N \,.\, P$	Input
$M \mathrel{\dot{\prec}} N$	Broadcast Out
$M \mathrel{\dot{\succ}} N$	Broadcast In
$\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$\mathbf{run}\ A\langle\widetilde{M}\rangle$	Invocation

Restriction $(\nu a)P$ *binds* a *in* P *and input* $\underline{M}(\lambda\widetilde{x})N\,.\,P$ *binds* $\widetilde{x}$ *in both* N *and* P*. An occurrence of a subterm in an agent is* guarded *if it is a proper subterm of an input or output term. An agent is* well-formed *if in all* $\underline{M}(\lambda\widetilde{x})N.P$ *it holds that* $\widetilde{x} \subseteq \mathrm{n}(N)$ *is a sequence without duplicates, and in all invocations* **run** $A\langle\widetilde{M}\rangle$ $|\widetilde{M}|$ *matches the arity of the clause associated with* A*.*

The agent **case** $\varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$ is sometimes abbreviated as **case** $\widetilde{\varphi} : \widetilde{P}$. We sometimes write $\underline{M}(x).P$ for $\underline{M}(\lambda x)x.P$. From this point on, we only consider well-formed agents.

The *actions* ranged over by α, β are of the following three kinds: *Output* $\overline{M}(\nu\tilde{a})N$, *input* $\underline{M}\,N$, and *silent* τ. Here we refer to M as the *subject* and N as the *object*. We define $\mathrm{bn}(\overline{M}(\nu\tilde{a})N) = \tilde{a}$, and $\mathrm{bn}(\alpha) = \emptyset$ if α is an input or τ. As in the pi-calculus, the output $\overline{M}(\nu\tilde{a})N$ represents an action sending N along M and opening the scopes of the names $\tilde{a}$.

Definition 5 (Transitions)**.** *A* transition *is written* $P \xrightarrow{\alpha} P'$*, meaning that* P *can do* α *to become* P'*. The transitions are defined inductively in Table 1.*

We identify alpha-equivalent agents and transitions. In a transition the names in $\mathrm{bn}(\alpha)$ bind into both the action object and the derivative, therefore $\mathrm{bn}(\alpha)$ is in the support of α but not in the support of the transition.

$$\textsc{In}\ \frac{\vdash K \leftrightarrow M}{\underline{M}(\lambda\widetilde{y})N\,.\,P \xrightarrow{\underline{K}\,N[\widetilde{y}:=\widetilde{L}]} P[\widetilde{y}:=\widetilde{L}]} \qquad \textsc{Out}\ \frac{\vdash M \leftrightarrow K}{\overline{M}\,N\,.\,P \xrightarrow{\overline{K}N} P}$$

$$\textsc{Case}\ \frac{P_i \xrightarrow{\alpha} P' \qquad \vdash\varphi_i}{\mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P'} \qquad \textsc{Par}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha)\#Q$$

$$\textsc{Com}\ \frac{\vdash M \leftrightarrow K \qquad P \xrightarrow{\overline{M}(\nu\widetilde{a})N} P' \qquad Q \xrightarrow{\underline{K}\,N} Q'}{P \mid Q \xrightarrow{\tau} (\nu\widetilde{a})(P' \mid Q')}\ \widetilde{a}\#Q$$

$$\textsc{Inv}\ \frac{P[\widetilde{x}:=\widetilde{M}] \xrightarrow{\alpha} P' \qquad A(\widetilde{x}) \Leftarrow P}{\mathbf{run}\ A\langle\widetilde{M}\rangle \xrightarrow{\alpha} P'} \qquad \textsc{Scope}\ \frac{P \xrightarrow{\alpha} P'}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b\#\alpha$$

$$\textsc{Open}\ \frac{P \xrightarrow{\overline{M}(\nu\widetilde{a})N} P'}{(\nu b)P \xrightarrow{\overline{M}(\nu\widetilde{a}\cup\{b\})N} P'}\ \begin{array}{l} b\#\widetilde{a}, M \\ b \in \mathrm{n}(N) \end{array} \qquad \textsc{BrOut}\ \frac{\Psi \vdash M \mathrel{\dot\prec} K}{\Psi \rhd \overline{M}\,N\,.\,P \xrightarrow{\overline{!K}\,N} P}$$

$$\textsc{BrIn}\ \frac{\Psi \vdash K \mathrel{\dot\succ} M}{\Psi \rhd \underline{M}(\lambda\widetilde{y})N\,.\,P \xrightarrow{\underline{?K}\,N[\widetilde{y}:=\widetilde{L}]} P[\widetilde{y}:=\widetilde{L}]}$$

$$\textsc{BrMerge}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\underline{?K}\,N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{?K}\,N} Q'}{\Psi \rhd P \mid Q \xrightarrow{\underline{?K}\,N} P' \mid Q'}$$

$$\textsc{BrCom}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{!K}\,(\nu\widetilde{a})N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{?K}\,N} Q'}{\Psi \rhd P \mid Q \xrightarrow{\overline{!K}\,(\nu\widetilde{a})N} P' \mid Q'}\ \widetilde{a}\#Q$$

$$\textsc{BrOpen}\ \frac{\Psi \rhd P \xrightarrow{\overline{!K}\,(\nu\widetilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{!K}\,(\nu\widetilde{a}\cup\{b\})N} P'}\ \begin{array}{l} b\#\widetilde{a}, \Psi, K \\ b \in \mathrm{n}(N) \end{array}$$

$$\textsc{BrClose}\ \frac{\Psi \rhd P \xrightarrow{\overline{!K}\,(\nu\widetilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\tau} (\nu b)(\nu\widetilde{a})P'}\ \begin{array}{l} b \in \mathsf{n}(K) \\ b\#\Psi \end{array}$$

Table 1: Structural operational semantics where K, L, M and N are terms; P, P', Q and Q' are agents. Symmetric versions of Com and Par are elided. In Open the expression $\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere. A symmetric version of BrCom is elided. In rules BrCom and BrMerge we assume that $\mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$ where $\widetilde{b}_P$ is fresh for $P, \widetilde{b}_Q, Q, K$ and Ψ, and that $\widetilde{b}_Q$ is fresh for $Q, \widetilde{b}_P, P, K$ and Ψ.

3.2 Algebra of Communicating Processes and mCRL2

The mCRL2 language [17] is a process description language and a toolset for formal modelling. The language is based on the Algebra of Communicating Processes (ACP) [6] extended with higher-order datatypes and equation systems on data. The following presentation is based on [23].

The observable action in mCRL2 is that of finite set of multi-actions of the form $\{a_1|\ldots|a_n\}$ where a_i is are arbitrary action defined by the user. We write α for multi-actions. The intuition here that multiple actions are observable simultaneously. Then, we can define the mCRL2 language.

Definition 6. *The mCRL2 processes are defined as follows.*

$$
\begin{array}{rcll}
P, Q & ::= & P \,||\, Q & \textit{parallel composition/merge} \\
& | & P \,|\, Q & \textit{synchronisation} \\
& | & P + Q & \textit{choice} \\
& | & P \cdot Q & \textit{sequence} \\
& | & \Sigma_{x \in D} P & \textit{summation} \\
& | & \alpha & \textit{multi action} \\
& | & \delta & \textit{deadlock/inaction} \\
& | & c \rightarrow P \diamond Q & \textit{if/then/else} \\
& | & X(x_1, \ldots, x_n) & \textit{process reference} \\
& | & \mathsf{comm}_C P & \textit{communication operator} \\
& | & \mathsf{hide}_H P & \textit{hiding operator} \\
& | & \mathsf{allow}_A P & \textit{restriction operator/allow} \\
& | & \mathsf{block}_B P & \textit{block operator}
\end{array}
$$

where H, A and B are finite sets of multi-actions and C is multi-action to multi-action transformation rule given as a finite set of the form $C = \{a_{11}|\ldots|a_{1m_1} \rightarrow a'_1, \ldots, a_{n1}|\ldots|a_{nm_n} \rightarrow a'_n\}$.

The mCRL2 language is slightly richer: we here give syntax for the language fragment without time as we had not used this fragment in this report.

The semantics for mCRL2 process are defined using equations on the processes that are reminiscent that of algebraic theories, e.g., that of λ-calculus. The semantics would not surprise the reader familiar with basic process algebra such as CCS.

Most of the semantics of the operators are quite standard and self-explanatory. The parallel composition is defined in terms of synchronisation and choice, e.g., the process $a||b$ is defined as $a \cdot b + b \cdot a + a|b$ where a and b are actions; so that the process reveals either a, b, or $a|b$ multi-actions. The synchronisation operator is defined to merge multi-actions into multi-actions recursively in a process. The choice operator $+$ is the standard non-deterministic choice. The infinite choice $\Sigma_{x \in D} P$ parameterises the process P over the domain D; this operator is useful for modelling message reception where D represents all the possible input values.

The specific language features of mCRL2 are the comm, hide, allow, and block. The $\mathsf{comm}_C P$ operators fuses multi-actions found in P according to the rules in C. The $\mathsf{hide}_H P$ operator as the name suggests hides the action found in H by emitting the silent action τ instead. The $\mathsf{block}_B P$ operator simply sends the process into the deadlock process if one of the multi-actions are emitted by P are in B. Finally, the allow operator only allows the specified multi-actions to be observed.

3.3 The Applied pi calculus and ProVerif

The applied pi calculus extends the standard pi-calculus so that data structures rather than names can be transmitted on channels. The data structures may be equipped with equations, and pattern matched against. ProVerif is a tool for verification of cryptographic protocols that supports a subset of applied π as input language. Details on ProVerif can be found in [11]; applied π was first introduced in [2].

The supported subset of applied pi is the following:

Definition 7. *The* terms *are as follows:*

M, N	$::=$	a, b, c, k, m, n, s	*names*
	$\mid$	x, y, z	*variables*
	$\mid$	$(M_1, \ldots, M_n)$	*tuple*
	$\mid$	$h(M_1, \ldots, M_k)$	*constructor/destructor application*
	$\mid$	$M = N$	*equality test*
	$\mid$	$M \rangle\langle N$	*inequality test*
	$\mid$	$M \,\&\&\, N$	*conjunction*
	$\mid$	$M \,\|\|\, N$	*disjunction*
	$\mid$	$\mathbf{not}(M)$	*negation*

The processes *are as follows:*

P, Q	$::=$	$P \mid Q$	*parallel*
	$\mid$	$!P$	*replication*
	$\mid$	$\mathbf{new}\, n : t; P$	*restriction*
	$\mid$	$\mathbf{in}(M, x : t); P$	*input*
	$\mid$	$\mathbf{out}(M, N); P$	*output*
	$\mid$	$\mathbf{if}\ M\ \mathbf{then}\ P\ \mathbf{else}\ Q$	*conditional*
	$\mid$	$\mathbf{let}\ x = M\ \mathbf{in}\ P\ \mathbf{else}\ Q$	*evalutation*
	$\mid$	$R(M_1, \ldots, M_n)$	*macro*
	$\mid$	0	*nil*

where t ranges over types.

Note that unlike psi-calculus, communication and pattern matching are separate. Since communication is monadic, we must use the **let** construct to obtain the arguments of composite terms using destructors. A typical example is a process that expects a message encrypted with key k, and upon receipt tries to decrypt it using the **dec** destructor:

$$\mathbf{in}(M, x : t); \mathbf{let}\ y = \mathbf{dec}(x, k)\ \mathbf{in}\ P\ \mathbf{else}\ Q$$

When some term is received on M, the process will proceed as $P[y := N]$ in the event of successful decryption, where N is the cleartext of the message. Otherwise it proceeds as Q; this happens if x is not an encrypted message or if k is the wrong decryption key. Note that the setting is parameterised on arbitrary constructors and destructors, so as users we are free to define our own.

The semantics of the other operators would be unlikely to surprise the reader.

4 Modelling and Verifying SHIA

In this section, we present how the SHIA model from Section 2.1 is actually written in Pwb, mCRL2 and ProVerif. We go through data and process models, explain divergences from the protocol model, and share results that are obtained from each tool.

4.1 Psi-Calculi Workbench

4.1.1 Model

We model the SHIA protocol (Section 2.1) in Pwb in the following ways.

Data Model. We provide the full data model for SHIA in Listing 1. The data model consists of basic data types (Sorts and CSorts), more complex data types (Symbols) that are based on the basic ones and the relations between these data types (Axioms).

Listing 1: Data model of SHIA in Pwb

```
@Sorts

    lbl
    mac
    tch
    nonce

@CSorts

    ch
    key

@Symbols

    Label       : (i) => lbl
    AggrLabel   : (lbl,lbl) => lbl
    EmptyLbl    : () => lbl
    MAC         : (nonce,key) => mac

    XOR         : (mac,mac) => mac

    Query       : (ch) => tch
    Verify      : (ch) => tch
    Qsuccess    : () => tch
    Offpath     : (ch) => tch

@Axioms

#Commutativity
    XOR(macX,macY) = XOR(macY,macX)
    AggrLabel(lblX,lblY) = AggrLabel(lblY,lblX)

#Associativity
    XOR(macX,XOR(macY,macZ)) = XOR(XOR(macX,macY),macZ)
    AggrLabel(lblX,AggrLabel(lblY,lblZ)) =
        AggrLabel(AggrLabel(lblX,lblY),lblZ)

#Distinctness of constructors
    not(AggrLabel(lblX,lblY) = Label(iZ))
    not(AggrLabel(lblX,lblY) = EmptyLbl)
    not(Label(iX) = EmptyLbl)
```

Process Model. We provide the process model for SHIA in Listing 2. There are three kinds of processes: Querier, InternalVertex and LeafVertex.

Querier starts the secure aggregation by sending a nonce to all other processes via authenticated broadcast. It receives the label of the root of the tree and starts the result checking by sending the label back via authenticated broadcast. It receives one MAC for all of the leaf vertices, checks that the MAC indeed represents all of the leaf vertices and, depending on the outcome of the check, it either accepts the aggregation or rejects it (lines 1-7).

InternalVertex receives a nonce from Querier via authenticated broadcast that indicates the start of a round of secure aggregation. It receives labels from its children and sends the aggregation of those labels to its parent. It receives the label of the root of the tree via authenticated broadcast that indicates the start of result checking. It swaps labels of children and sends as off-path labels. It receives MACs from its children and sends the XOR of them to its parent (lines 9-19).

LeafVertex receives a nonce from Querier via authenticated broadcast that indicates the start of a round of secure aggregation. It creates a label based on the sensing and sends it to its parent. It receives the label of the root of the tree via authenticated broadcast. It also receives its off-path label from its parent. If it can re-build the label of the root of the tree using its own label and off-path label, then it sends a MAC, which is based on the nonce of the round and the key that is shared with the querier, to its parent (lines 21-27).

System represents an example scenario in which there is a querier and two sensor nodes, and all of them are honest participants in SHIA protocol. Two sensor nodes are modelled as two leaf vertices and one internal vertex as in the commitment tree; therefore, the leaf vertices do sensing and send labels whereas the internal vertex does aggregation and carries messages between leaf vertices and the querier. All of the processes in the scenario are initialised with appropriate parameters. For example; the Querier process must know the channels, where it should send and receive its messages, the nonce of the round and the final MAC value, which should be produced by collaboration of all the vertices in the tree and also based on the nonce (lines 29-36).

Listing 2: Process model of SHIA in Pwb language for 2 node network.

```
Querier(chQuerier,chBS, nonceQ, macXorAllAuthCode) <=
     '"Query(chQuerier)"!<nonceQ>.
     "Query(chBS)"(lblRoot).
     '"Verify(chQuerier)"!<lblRoot>.
     "Verify(chBS)"(macAuthCode).
     case "macAuthCode = macXorAllAuthCode"     : '"Qsuccess"<0>
     [] "not(macAuthCode = macXorAllAuthCode)" : 0;

InternalVertex(chQuerier,chParent, chLeft, chRight) <=
         "Query(chQuerier)"?(nonceQ).
         "Query(chLeft)"(lblLeft).
         "Query(chRight)"(lblRight).
         '"Query(chParent)"<"AggrLabel(lblLeft,lblRight)">.
         "Verify(chQuerier)"?(lblRoot).
         '"Offpath(chLeft)"<lblRight>.
         '"Offpath(chRight)"<lblLeft>.
         "Verify(chLeft)"(macLeft).
         "Verify(chRight)"(macRight).
         '"Verify(chParent)"<"XOR(macLeft, macRight)">;

LeafVertex(chQuerier, chParent, keyK, iSensedValue) <=
         "Query(chQuerier)"?(nonceQ).
         '"Query(chParent)"<"Label(iSensedValue)">.
         "Verify(chQuerier)"?(lblRoot).
         "Offpath(chParent)"(lblOffpath).
         case "AggrLabel(Label(iSensedValue),lblOffpath) = lblRoot" :
                  '"Verify(chParent)"<"MAC(nonceQ, keyK)">;

System(chQuerier) <= (new chBS, chLeft, chRight, keyLeft, keyRight)
         (
         (new nonceQ) (Querier<chQuerier, chBS, nonceQ,
                  "XOR(MAC(nonceQ, keyLeft),MAC(nonceQ, keyRight))">) |
         InternalVertex<chQuerier, chBS, chLeft, chRight> |
         LeafVertex<chQuerier, chLeft, keyLeft, 20> |
         LeafVertex<chQuerier, chRight, keyRight, 22>
         );
```

4.1.2 Verification

SHIA protocol is verified both manually and automatically by using Pwb. Manual verification is step-wise execution of the protocol model, where each step is an observable transition that takes place in the state space of the protocol. This verification can also be done automatically, hence we are able to draw state diagrams of the protocol for different scenarios in an automated way. An example state diagram is shown in Figure 2. The diagram belongs to a system where there are two nodes and an attacker plays man-in-the-middle between the nodes and the querier. After running the process model in the Pwb, the diagram is generated for taking transitions only for a depth of ten. The text in the figure is not visible as the diagram does not fit in this page format. The ellipses represent states and the arrows represent the transitions. The text in an ellipse or next to a transition defines the constraints that need to be fulfilled to reach the state or to take the transition.

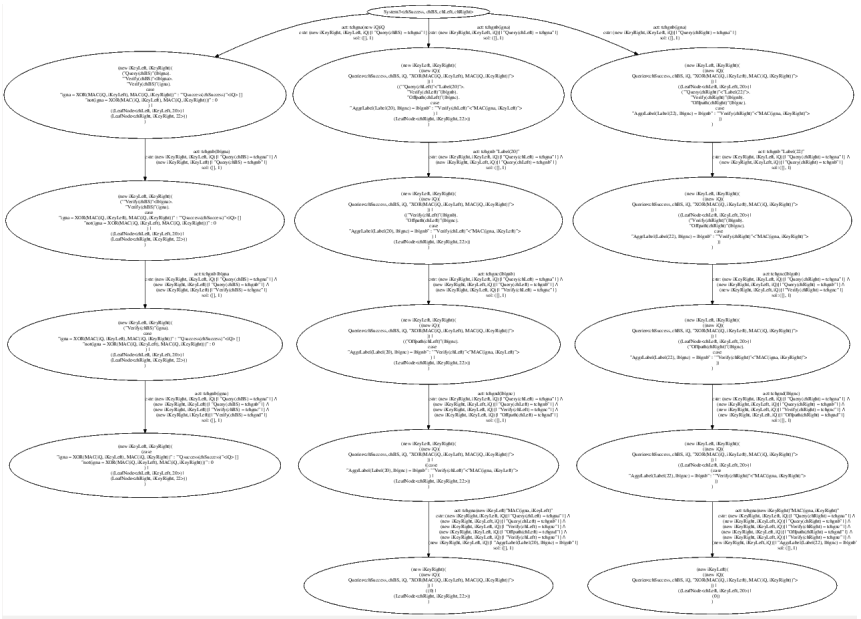


Figure 2: A state diagram from a network of two nodes with an attacker

4.1.3 Results

SHIA protocol runs correctly as modelled. The results for a small scenario with a few sensor nodes can be obtained in terms of seconds. When the network size increases, the time necessary to evaluate the model increases exponentially, i.e. from seconds to hours. Here are some example timings in Table 2.

Network Size	Evaluation Time (seconds)
2	3.67
4	10.56
8	9266.33

Table 2: Pwb timing results for evaluating the SHIA protocol model

4.2 mCRL2

4.2.1 Model

We model the SHIA protocol (Section 2.1) in mCRL2 in the following ways.

Data Model. We provide the full data model for SHIA in Listing 3.

The messages originating from the base station in SHIA need to be authenticated. We model authenticity of a message by using verification of digital signatures with asymmetric key cryptography (lines 3-25). This is needed for both nonces and the final label that the base station receives. Since the mCRL2 maps are monomorphic, we need to provide two separate maps and equations for

Listing 3: Data model of SHIA in mCRL2 language

```
sort Nonce = struct nonce(n: Int) ;

sort PubPrivKeyPair, PubKey, PrivKey;
map genKeyPair : Int -> PubPrivKeyPair;
map pubkey  : PubPrivKeyPair -> PubKey;
map privkey : PubPrivKeyPair -> PrivKey;

sort Signature = struct sg1 | sg2 | sg3 | sg4 | sg5;

map signNonce : Nonce # PrivKey -> Signature;
map signLabel : Label # PrivKey -> Signature;

map verifNonceSig : Nonce # Signature # PubKey -> Bool;
var n1, n2 : Nonce, pk1, pk2 : PubPrivKeyPair;
eqn (n1 == n2 && pk1 == pk2)
        -> verifNonceSig (n1, signNonce(n2, privkey(pk1)),
                          pubkey(pk2)) = true;
    (n1 != n2 || pk1 != pk2)
        -> verifNonceSig (n1, signNonce(n2, privkey(pk1)),
                          pubkey(pk2)) = false;

map verifLabelSig : Label # Signature # PubKey -> Bool;
var n1, n2 : Label, pk1, pk2 : PubPrivKeyPair;
eqn (n1 == n2 && pk1 == pk2)
        -> verifLabelSig (n1, signLabel(n2, privkey(pk1)),
                          pubkey(pk2)) = true;
    (n1 != n2 || pk1 != pk2)
        -> verifLabelSig (n1, signLabel(n2, privkey(pk1)),
                          pubkey(pk2)) = false;

sort Key;
map key: Int -> Key;

sort MACode;
map maCode : Nonce # Key -> MACode;
sort Digest = Bag(MACode);

map XOR : Digest # Digest -> Digest;
var a, b: Digest;
eqn XOR(a,b) = a + b;

map MAC : Nonce # Key -> Digest;
var n: Nonce, k : Key;
eqn MAC(n,k) = {maCode(n,k) : 1};

sort SensedValues = Int;
sort Label = Bag(SensedValues);
map AggrLabel : Label # Label -> Label;

var left, right: Label;
eqn AggrLabel(left, right) = left + right;

map ValueLabel : Int -> Label;
var value: Int;
eqn ValueLabel(value) = { value : 1 };

sort Chan = struct c(channelId: Int) ;

act sendQuery, recvQuery, tauQuery
: Chan # Nonce # Signature;
act sendQueryLabel, recvQueryLabel, tauQueryLabel : Chan # Label;
act sendVerify, recvVerify, tauVerify
: Chan # Label # Signature;
act sendVerifyCode, recvVerifyCode, tauVerifyCode : Chan # Digest;
act sendOffpath, recvOffpath, tauOffpath          : Chan # Label;

act Q_SUCCESS, Q_FAIL, NODE_FAIL;
```

nonce and label signature verification. In the process model, the nodes would have the public key of the base station. The digital signature verification is easily expressed equationally in mCRL2 (lines 14-18, and 20-25). The reason why we use digital signatures and not message authentication codes to obtain authenticity of nonces and labels is that these messages are part of a command that is broadcasted to a network (lines 55 and 57). That is, if we used MACs with symmetric keys, it would be sufficient for an attacker to compromise a single node and obtain the shared key to be able to forge authenticated broadcasts for sending request to a network.

Message Authentication Codes are modelled abstractly as an indeterminate map from nonces and keys to types for MACs (lines 27-31). mCRL2 allows for equational specification on data; however, the tools treat those specification as term rewriting systems (where the left hand side of equations may be rewritten to the right hand side). We cannot directly model the commutativity of XOR operation, i.e. $\mathsf{XOR}(a, b) = \mathsf{XOR}(b, a)$; however, we are able to express the commutativity of XOR by modelling it as an operation on finite multisets[1] (called bags in mCRL2) of message authentication codes. This is the reason why we introduce the type Digest and the construction of a MAC is an injection to Digest (lines 32-40). We use the same technique to model label aggregation (lines 42-51).

We model channels as a separate sort with a single injective operation of sort integer (line 53). We define distinct actions for each phase of the protocol: broadcast of query, aggregation of labels from nodes, broadcast of verification of a label, and distribution of off-path labels. For each of theses phases, we define three actions for sending and receiving ends of a channel, and an action to indicate communication (lines 55-59).

Process Model. We provide a mCRL2 process model for a SHIA network in Listing 4. For the sake of simplicity, it is for a two node network. It is straightforward to extend it to bigger networks (the complete model can be obtained at [1].

Each vertex in the commitment tree is modelled as a separate process — even the internal ones.

The querier works as expected: it first issues a query command with signed nonce, and then waits for the full label to arrive from the network. It requests verification of the label from the network by signing the label. Ultimately, if the received verification code from the network does not match the expected it signals a failure, otherwise success (lines 1-9).

SHIA relies on the messages broadcasted from the base station to be authenticated. Since mCRL2 language does not have primitives for broadcast communication, we instead model message broadcast as point-to-point forwarding of messages by the intermediate vertices to their left and right child vertex in a commitment tree (lines 24, 29, and process definitions are on lines 11 and 17).

The leaf processes check for the authenticity of messages that originate from the base station and signal a failure (lines 39 and 42) whenever the check fails. The vertices also signal failure with the same action whenever they detect that their values were not accounted in the final label.

[1] XOR over MACs and multiset union over MACs are the same kind of algebraic structures that are associative, commutative, and closed.

Listing 4: Process model of SHIA in mCRL2 language for 2 node network.

```
proc Querier(bs: Chan, non: Nonce,
             pk: PubPrivKeyPair, authCode: Digest) =
    sendQuery(bs, non, signNonce(non, privkey(pk))) .
    sum rootLabel : Label . recvQueryLabel(bs, rootLabel) .
    sendVerify(bs, rootLabel, signLabel(rootLabel, privkey(pk))) .
    sum code: Digest. recvVerifyCode(bs, code) .
    ((authCode == code) -> Q_SUCCESS
                        <> Q_FAIL
    );

proc ForwardNonce(n: Nonce, s: Signature,
                  p: Chan, l: Chan, r: Chan) =
    recvQuery(p, n, s).
    sendQuery(l, n, s).
    sendQuery(r, n, s);

proc ForwardRootLabel(lbl: Label, s: Signature,
                      p: Chan, l: Chan, r: Chan) =
    recvVerify(p, lbl, s).
    sendVerify(l, lbl, s).
    sendVerify(r, lbl, s);

proc InternalVertex(parent: Chan, left: Chan, right: Chan) =
    sum n: Nonce, s: Signature.
        ForwardNonce(n, s, parent, left, right).
    sum leftLabel: Label . recvQueryLabel(left, leftLabel) .
    sum rightLabel: Label . recvQueryLabel(right, rightLabel) .
    sendQueryLabel(parent, AggrLabel(leftLabel, rightLabel)) .
    sum rootLabel: Label, ss: Signature.
        ForwardRootLabel(rootLabel, ss, parent, left, right).
    sendOffpath(left,  rightLabel).
    sendOffpath(right, leftLabel).
    sum leftCode: Digest. recvVerifyCode(left, leftCode).
    sum rightCode : Digest. recvVerifyCode(right, rightCode).
    sendVerifyCode(parent, XOR(leftCode, rightCode));

proc LeafVertex(parent: Chan, k: Key, pk: PubKey, sensedValue: Int) =
    sum n : Nonce, s: Signature . recvQuery(parent, n, s) .
    !verifNonceSig(n, s, pk) -> NODE_FAIL <>
    sendQueryLabel(parent, ValueLabel(sensedValue)) .
    sum rootLabel: Label, ls:
                Signature . recvVerify(parent, rootLabel, ls).
    !verifLabelSig(rootLabel, ls, pk) -> NODE_FAIL <>
    sum offpathLabel: Label. recvOffpath(parent, offpathLabel).
    ((AggrLabel(ValueLabel(sensedValue), offpathLabel) == rootLabel ) ->
        sendVerifyCode(parent, MAC(n, k))
        <> NODE_FAIL
      );
```

The keys are pre-distributed and this is done by simply initialising the parameters of the processes. The keys and nonces need to be concretely constructed as terms. We also provide the expected verification result. mCRL2 does not have primitives for generation of arbitrary values as creation of new names in psi-calculi. The following is an example of two node network initialisation.

```
proc System =
    Querier(c(0), nonce(0), genKeyPair(0),
        XOR(MAC(nonce(0), key(0)), MAC(nonce(0), key(1)))) ||
    InternalVertex(c(0), c(1), c(2))                        ||
    LeafVertex(c(1), key(0), pubkey(genKeyPair(0)), 20)     ||
    LeafVertex(c(2), key(1), pubkey(genKeyPair(0)), 22) ;
```

The following completes the process model of SHIA for the two node network. We only allow communication actions, queries success and failures, and node failure actions. The comm renames the multi-actions with different polarities to

our communication actions.

```
init allow({tauQueryLabel, tauQuery, tauVerify, tauVerifyCode,
            tauOffpath, Q_SUCCESS, Q_FAIL, NODE_FAIL },
      comm( { sendQueryLabel | recvQueryLabel -> tauQueryLabel,
              sendVerifyCode | recvVerifyCode -> tauVerifyCode,
              sendOffpath    | recvOffpath    -> tauOffpath,
              sendQuery      | recvQuery      -> tauQuery,
              sendVerify     | recvVerify     -> tauVerify },
            System
      )) ;
```

4.2.2 Divergence from the Protocol Model

In some ways the model in mCRL2 is more concrete than the abstract model (Section 2.1). The abstract model does not specify how the messages are authenticated. Chan et al. [14] do not commit to a particular protocol and state that SHIA is independent of this choice, while we needed to commit to a particular method to ensure authenticity. We used digital signatures based on asymmetric key cryptography. While the asymmetric key cryptography is straightforward to express equationally, it may be unrealistic as asymmetric cryptography is computationally expensive and protocols like μTesla might be used instead. Furthermore, we have used point-to-point communication to model broadcast. This choice makes the internal vertices exhibit more behaviour than in the abstract model, while in abstract model only the leaf vertices (viz., the physical nodes) would exhibit broadcast communication.

We could not state XOR and label aggregation operations equationally (abstractly), although mCRL2 language has sufficient built-in data structures that allows us to model them in this case.

4.2.3 Verification

Unfortunately, the mCRL2 toolset uses only explicit state enumeration to generate state transitions and, what is more, it is not possible to associate equations on the traces of actions. Thus, we are forced to model the attacker explicitly, that is, as a process. This is a very weak attacker model that relies on the ingenuity of the model writer to cover all of the cases.

We then use the mCRL2 model checker on specified properties in a very powerful modal logic, modal μ-calculus (the mCRL2 parametric boolean equation system). We check that in the attacker's presence, the querier eventually rejects the aggregate value. Similarly, without the attacker the system eventually accepts the aggregate value.

For example, in Listing 5 an internal vertex that has been captured ignores the labels received from the children and injects its own sensed value label (line 6). If we use the captured internal vertex instead of internal one in the network, then the following property is attested by the mCRL2 model checker (`pbes2bool`), that is, a node eventually fails since it detects that its value was not contributed to the final label.

```
<true*><NODE_FAIL>true
```

In the network absent of malicious nodes, the following is true, that the querier always eventually accepts the final aggregate, that is, from any state there is a path ending with the Q_SUCCESS action.

```
[true*]<true*><Q_SUCCESS>true
```

Listing 5: Captured node in two node network

```
proc CapturedInternalVertex1(parent: Chan, left: Chan, right: Chan) =
    sum n: Nonce, s: Signature.
        ForwardNonce(n, s, parent, left, right).
    sum leftLabel: Label . recvQueryLabel(left, leftLabel) .
    sum rightLabel: Label . recvQueryLabel(right, rightLabel) .
    sendQueryLabel(parent, ValueLabel(999999)) .
    sum rootLabel: Label, ss: Signature.
        ForwardRootLabel(rootLabel, ss, parent, left, right).
    sendOffpath(left, rightLabel).
    sendOffpath(right, leftLabel).
    sum leftCode: Digest. recvVerifyCode(left, leftCode).
    sum rightCode : Digest. recvVerifyCode(right, rightCode).
    sendVerifyCode(parent, XOR(leftCode, rightCode));
```

4.2.4 Usability and Experience

The mCRL2 toolset is mature and well documented. It is fairly straightforward to model with efficient tools for model checking and simulation. The major shortcoming is that the transition model is overly concrete, thus it is hard to capture more abstract features as discussed in the previous sections. This also has a practical implication: the simulator of mCRL2 processes works on the intermediate language, the linearised process, that is syntactically far removed from the original and is essentially unreadable. This makes debugging even fairly small models such as this one a very daunting task.

4.2.5 Results

The largest network that we considered in mCRL2 is 8 node network. The linearised process generation is very fast, and the model checking takes around 30 min per formula on a system with dual core at 1.7 GHz clock speed and 4 GB of RAM.

4.3 ProVerif

As always, the question of how to model authenticated broadcast from the base station presents itself. The problem is twofold: we must model broadcast, and we must model authentication.

Unlike Pwb there is the lack of a broadcast communication primitive; hence we must model broadcast communication using only unicast communication. It is impossible to obtain a "good" encoding in the general case when the number of nodes is unknown [21]; fortunately we are modelling a scenario in which the topology of the network is known, so we can come up with a protocol-specific encoding.

As for authentication, the original presentation of SHIA abstracts away from the particular mechanism used to do this; several techniques are out there [42, 25, 34] and it is always nice to offer the implementer some freedom and not bog down the presentation.

We offer two different models of SHIA in ProVerif, corresponding to different ways of modelling authentication. The first model attempts to stay true to the philosophy of abstracting away from SHIA, while the second model deviates from it by committing to a public/private-key signature scheme.

4.3.1 Model 1

The first model uses a sequence of unicast messages on a private channel brAuth to model authenticated broadcasts (it is declared private in line 9, meaning that the attacker cannot partake). Other than that, the deviations from the Pwb model are mostly syntactic; the sequentialisation in particular follows the same style.

Security guarantees are formulated as relations between *events*. Events may be explicitly signalled when certain states of a process have been reached, and are useful for using ProVerif to reason about reachability and safety. Lines 16-18 define the three events we use: Success, which is signalled when the querier *believes* the aggregation result is valid; ActualSuccess, which is signalled when regardless of the querier's beliefs, the aggregation result is *actually* valid; and NodeOK(mkey), meaning that the node with private key NodeOK(mkey) accepts the aggregation result.

NodeOK events are triggered by the respective nodes when they send out MACs (lines 88 and 99). The purpose of also having the event is to be able to distinguish situations where an attacker was somehow able to spoof the MAC without any interaction with the node in question. Success and ActualSuccess are triggered in the querier process in lines 60-70. First, if the received label is equal to a label constructed from the actual sensed values of the nodes, we signal ActualSuccess. Then, if the querier is able to verify the MAC, we signal Success. Note that in reality, the querier of course cannot construct a label from the actual sensed values of the nodes since it has no knowledge of them. This unrealism is not a problem since our process has the form:

$$\textbf{if}\ \text{unrealistic test}\ \textbf{then event}\ E; P\ \textbf{else}\ P$$

In other words, the continuation is the same regardless of the outcome of the unrealistic tests, modulo raising the event E. However, the event does not influence the subsequent behaviour of P and is not observable by any other process; it is only used in the verification stage.

We use events to construct seven queries to ask ProVerif (lines 24-47). The first four queries are sanity checks, to verify that states where the various sets are satisfied are indeed reachable. The remaining three queries correspond to the correct operation of SHIA. They test that if the querier is able to verify the MAC, then all nodes are happy with the label; and most importantly, that whenever the querier is able to verify the MAC, it must be the case that the label has not been tampered with.

Listing 6: ProVerif model with private channel

```
1  type nonce.
2  type mkey.
3
4  free mynonce : nonce. (* note: nonce is not [private] *)
5  free chBS, chLeft, chRight : channel.
6  free leftvalue : bitstring.
7  free rightvalue : bitstring.
8
9  free brAuth : channel [private].
10
11 (* Cryptographic primitives for symmetric key cryptography *)
12 fun mac(nonce, mkey) : bitstring.
13 fun AggrLabel(bitstring, bitstring) : bitstring.
14 fun XOR(bitstring, bitstring) : bitstring.
```

```
15
16 event Success. (* the querier receives a correct MAC *)
17 event ActualSuccess. (* the querier receives a correct label *)
18 event NodeOK(mkey).
19 (* the node with private key accepts the aggregation *)
20
21 (* Security assumptions: *)
22 (* the attacker cannot learn the private keys of X and Y. *)
23 not attacker(new leftkey).
24 not attacker(new rightkey).
25
26 (* Ask ProVerif if... *)
27
28 (* 1. Reachability of successful state. *)
29 query event(ActualSuccess).
30 query event(Success).
31
32 (* 2. Reachability of node acceptance. *)
33 query event(NodeOK(new leftkey)).
34 query event(NodeOK(new rightkey)).
35
36 (* 3. Whenever the querier accepts the MAC, the nodes
37       have previously accepted the label
38  *)
39 query event(Success) ==> event(NodeOK(new leftkey)).
40
41 query event(Success) ==> event(NodeOK(new rightkey)).
42
43 (* 4. If the querier accepts the MAC, the querier
44       received a correct label.
45  *)
46 query event(Success) ==> event(ActualSuccess).
47
48 (* Definitions of protocol participants.
49
50  *)
51
52 let Querier(leftkey : mkey, rightkey : mkey) =
53     out(brAuth,mynonce);
54     out(brAuth,mynonce);
55     in(chBS,lblRoot : bitstring);
56     out(brAuth,lblRoot);
57     out(brAuth,lblRoot);
58     in(chBS,macAuthCode : bitstring);
59     if lblRoot = AggrLabel(leftvalue,rightvalue) then
60        event ActualSuccess;
61        if macAuthCode = XOR(mac(mynonce,leftkey),mac(mynonce,rightkey))
62        then
63           event Success
64        else
65           0
66     else
67        if macAuthCode = XOR(mac(mynonce,leftkey),mac(mynonce,rightkey))
68        then
69           event Success
70        else
71           0.
72
73 let NodeInternal() =
74     in(chLeft,lblLeft : bitstring);
75     in(chRight,lblRight : bitstring);
76     out(chBS,AggrLabel(lblLeft,lblRight));
77     out(chLeft,lblRight);
78     out(chRight,lblLeft);
79     in(chLeft,macLeft : bitstring);
80     in(chRight,macRight : bitstring);
81     out(chBS,XOR(macLeft,macRight)).
82
83 let NodeLeafLeft(leftkey : mkey,leftvalue : bitstring) =
84     in(brAuth,nnonce : nonce);
85     out(chLeft,leftvalue);
86     in(brAuth,lblRoot : bitstring);
87     in(chLeft,lblOffpath : bitstring);
88     if AggrLabel(leftvalue,lblOffpath) = lblRoot then
```

```
89          event NodeOK(leftkey);
90          out(chLeft,mac(nnonce,leftkey))
91      else
92          0.
93
94  let NodeLeafRight(rightkey : mkey,rightvalue : bitstring) =
95      in(brAuth,nnonce : nonce);
96      out(chRight,rightvalue);
97      in(brAuth,lblRoot : bitstring);
98      in(chRight,lblOffpath : bitstring);
99      if AggrLabel(lblOffpath,rightvalue) = lblRoot then
100         event NodeOK(rightkey);
101         out(chRight,mac(nnonce,rightkey))
102     else
103         0.
104
105 process new leftkey : mkey;
106         new rightkey : mkey;
107         Querier(leftkey,rightkey) |
108         NodeInternal() |
109         NodeLeafLeft(leftkey,leftvalue) |
110         NodeLeafRight(rightkey,rightvalue)
```

4.3.2 Model 2

The second model eschews the private channel approach and instead uses a public/private-key signature scheme to obtain authentication. Removing private channels from consideration yields a possibly stronger attacker model (who can now do Dolev-Yao skulduggery also on nonce propagation transitions), at the price of cluttering the model with encryption and decryption operations. Note that because of ProVerif's type system it is necessary to have two distinct signing and checking functions for messages and nonces, respectively.

Listing 7: ProVerif model with explicit public/private keys

```
1   ...
2
3   free chBS, chLeft, chRight, chAttacker : channel.
4
5   ...
6
7   (* Cryptographic primitives for public-key signatures *)
8   type sskey .
9   type spkey .
10  fun spk ( sskey ) :  spkey .
11  fun sign ( bitstring , sskey ) :  bitstring .
12
13  reduc forall m:  bitstring , k :  sskey ;  getmess ( sign (m, k )) = m.
14  reduc forall m: bitstring, k: sskey; checksign(sign(m,k),spk(k)) = m.
15
16  fun signn ( nonce , sskey ) :  nonce .
17
18  reduc forall m:  nonce , k :  sskey ;  getmessn ( signn (m, k )) = m.
19  reduc forall m: nonce, k: sskey; checksignn(signn(m,k),spk(k)) = m.
20
21  free qskey : sskey [private].
22
23  ...
24
25  let Querier(leftkey : mkey, rightkey : mkey) =
26      out(chBS,signn(mynonce,qskey));
27      in(chBS,lblRoot : bitstring);
28      out(chBS,sign(lblRoot,qskey));
29      in(chBS,macAuthCode : bitstring);
30      if lblRoot = AggrLabel(leftvalue,rightvalue) then
31         event ActualSuccess;
32         if macAuthCode = XOR(mac(mynonce,leftkey),mac(mynonce,rightkey))
33         then
34            event Success
```

```
35          else
36              0
37      else
38          if macAuthCode = XOR(mac(mynonce,leftkey),mac(mynonce,rightkey))
39          then
40              event Success
41          else
42              0.
43
44  let NodeInternal() =
45      in(chBS,nnonce : bitstring);
46      out(chLeft,nnonce);
47      out(chRight,nnonce);
48      in(chLeft,lblLeft : bitstring);
49      in(chRight,lblRight : bitstring);
50      out(chBS,AggrLabel(lblLeft,lblRight));
51      in(chBS,lblRoot : bitstring);
52      out(chLeft,lblRoot);
53      out(chRight,lblRoot);
54      out(chLeft,lblRight);
55      out(chRight,lblLeft);
56      in(chLeft,macLeft : bitstring);
57      in(chRight,macRight : bitstring);
58      out(chBS,XOR(macLeft,macRight)).
59
60  let NodeLeafLeft(leftkey : mkey,leftvalue : bitstring) =
61      in(chLeft,nnonce : nonce);
62      out(chLeft,leftvalue);
63      in(chLeft,lblRoot : bitstring);
64      in(chLeft,lblOffpath : bitstring);
65      if AggrLabel(leftvalue,lblOffpath) = checksign(lblRoot,spk(qskey))
66      then
67          event NodeOK(leftkey);
68          out(chLeft,mac(checksignn(nnonce,spk(qskey)),leftkey))
69      else
70          0.
71
72  let NodeLeafRight(rightkey : mkey,rightvalue : bitstring) =
73      in(chRight,nnonce : nonce);
74      out(chRight,rightvalue);
75      in(chRight,lblRoot : bitstring);
76      in(chRight,lblOffpath : bitstring);
77      if AggrLabel(lblOffpath,rightvalue) = checksign(lblRoot,spk(qskey))
78      then
79          event NodeOK(rightkey);
80          out(chRight,mac(checksignn(nnonce,spk(qskey)),rightkey))
81      else
82          0.
83
84  process new leftkey : mkey;
85          new rightkey : mkey;
86          out(chAttacker,spk(qskey));
87          Querier(leftkey,rightkey) |
88          NodeInternal() |
89          NodeLeafLeft(leftkey,leftvalue) |
90          NodeLeafRight(rightkey,rightvalue)
```

4.3.3 Result

Unfortunately, ProVerif doesn't produce any conclusive answers with either model. The output produced is identical in both cases:

```
RESULT event(Success) ==> event(ActualSuccess) cannot be proved.
RESULT event(Success) ==> event(NodeOK(rightkey[])) is true.
RESULT event(Success) ==> event(NodeOK(leftkey[])) is true.
RESULT not event(NodeOK(rightkey[])) is false.
RESULT not event(NodeOK(leftkey[])) is false.
RESULT not event(Success) cannot be proved.
```

```
RESULT not event(ActualSuccess) is false.
```

"Cannot be proved" in ProVerif lingo means "I don't know". In other words, in both models ProVerif fails to verify whether Success is reachable or not; it also fails to verify whether ActualSuccess always precedes Success. All other queries produce the expected result.

It is no secret that ProVerif is an incomplete (but sound) solver. It is unfortunate that we seem to have struck incompleteness in this particular case, and it would be interesting to discover why. The ProVerif manual discusses various sources of incompleteness [11, pp. 92-93], and it should be investigated which (of any) applies.

On a more positive note, the inconclusive answer arrives really fast (a matter of milliseconds at most). Since no satisfactory answers were procured on a two-node network, we have not investigated how this scales to larger networks.

5 Pwb Development

During the course of this project, we have extended Pwb in significant ways and developed a new advanced module (instance) for Pwb. We strived throughout this enterprise to make the work as generic as possible, so that it may be applicable to other similar formalisation attempts.

Our work on Pwb consists of extending the main codebase of the tool and developing a module which allows Pwb to use first order algebraic datatypes with equational logic on the data. We have implemented the needed interface with Pwb to parse and represent such algebraic specifications and also we implemented a translation to SMT-LIB language as an interface to external SMT solvers to do the heavy lifting of solving these equations. Thus, it was a conscious design decision to keep the algebraic specification simply sorted first-order language close to that of SMT solvers. An example of the specification is found earlier in this report in Listing 1.

We have implemented support for algebraic specification (signature) consisting of finite number of sorts with distinguished sort for booleans, a list of first order and sorted (function symbols), and a list of equations over the terms formed over the signature. We introduced a distinction between *sorts* (akin to variables) and *constant sorts* (akin to fixed names) which is not found in SMT-LIB. Their treatment when solving transition constraints differ as follows: we accept solutions where two values with the same sort are equated, but do not accept such solutions for constant sorts. This helps rule out many nonsensical scenarios, such as transitions that are available under the constraint that the private keys of two different nodes are equated. In our implementation, labels, MACs, node names and nonces are sorts; channels and keys are constant sorts.

Here we list a detailed list of enhancements to Pwb on top of the basis described above.

Concurrent SMT solvers. We implemented support for checking constraints on multiple SMT solvers concurrently. Implemented with the Poly/ML Threads library [29]. Queries are run simultaneously on CVC4 [8] and Z3 [20]; this turns out to be practically useful since many queries that time out on Z3 are solved instantly by CVC4, and vice versa. The implementation is such that it would

be trivial to add more solvers. However, even though there are many SMT solvers on the market, our current encoding of transition constraints into SMT queries requires an SMT-LIB logic with some rather advanced features (at least UFNIA). Very few solvers support these features. In fact, the only solvers even entered for this category for the 2015 SMT-COMP were CVC4, Z3, and CVC3. Looking beyond SMT solvers, a promising candidate for another back-end solver to include would be VAMPIRE [36].

Caching of SMT solver queries. We implemented caching of SMT solver queries. The result of all queries are stored in a hash table, that is indexed by a string representation of the query, but with all identifier names replaced by de Bruijn-indices. Using the de Bruijn representation allows us to increase the number of cache hits, since hits are up-to alpha rather than only for syntactically equal queries. An interesting direction for future work might be hash tables indexed by canonical representations of constraints up-to more sophisticated equivalences, such as associativity and commutativity (AC) congruence of formulas, and reordering of constraints.

Query caching is practically indispensable in the context of symbolic execution [31], since transition constraints are often very similar to each other. In our work we have often experienced drastic improvements in efficiency and reliability of our tool by simply introducing caching, or increasing the maximum cache size when we hit brick walls.

Reliable broadcast semantics. We implemented reliable broadcast semantics [4]. By using broadcast communication on a private channel, we can model authenticated broadcast without having to introduce an explicit protocol to disseminate messages through the network, and without having to introduce a model of the associated cryptography. By using reliable as opposed to unreliable broadcast communication, we obtain smaller state spaces by removing many uninteresting cases from consideration, namely those where the protocol cannot run to completion because not every participant knows what's going on. Since we disregard denial of service attacks such scenarios are not interesting.

The implementation of reliable broadcast is based on a simple observation. We consider the same transition candidates as in the unreliable case (namely for each unguarded output prefix, there is one candidate transition to be evaluated for every subset of the available input prefixes). For each transition candidate, we keep track of which rules are applied in its derivation. First, the constraints corresponding to all candidate transitions are solved. Of the ones that are solvable, we inspect their derivations and count how many times the rules Com and Merge are applied in the derivations; call this the *reliability value* of the transition. We then discard all solvable candidate transitions whose reliability value is less than some other solvable candidate transition.

Intuitively, the reason this algorithm works is that a lower reliability value means that at some point in the derivation tree, Par was used when Merge was applicable, which is prohibited by the semantics of reliable broadcast. It should be noted that this algorithm is incorrect in the presence of free choice. For an example, consider the process

$$P \triangleq \overline{M} \mid (\underline{M} \mid \underline{M}) + \underline{M}.Q$$

With reliable broadcast semantics this process has two transitions (up-to struct): (1) $P \xrightarrow{\overline{M}} \mathbf{0}$ and (2) $P \xrightarrow{\overline{M}} Q$. However, transition (1) has reliability value 2 and transition (2) has reliability 1, so the algorithm will discard transition (2) and thus conclude that only (1) is available. Fortunately, this does not come into play in the SHIA model since it does not use choice at all (all its case statements are unary). An interesting direction for future work is to find a symbolic semantics for reliable broadcast in the presence of free choice, and an efficient implementation thereof.

For every unguarded broadcast output we must generate one transition constraint for every subset of the unguarded broadcast inputs. For an example, the process

$$P \triangleq \overline{M} \mid \Pi_{1 \leq i \leq 16} \underline{M}.P_i$$

generates $2^{16} = 65536$ transition constraints (c.f. point-to-point semantics, where P has only 16 transition constraints)! This does not scale at all well to larger systems; merely generating (not even solving) transition constraints for larger networks (say, 8 nodes or more) can take an annoyingly long time, and represents a performance brick wall waiting to happen at even larger network sizes.

However, for the process P, the human observer instantly realises that 65535 of these constraints are in fact logically equivalent! This suggests that transition constraint generation for broadcast can probably be done in a more clever way that avoids this explosion.

The modelling of the SHIA protocol was a significant driver for improving the Pwb tool. The result of this is a solid first-order algebraic specification for the term language of the processes with battle-tested improvements on the interface with SMT solvers, pragmatic trade-offs on improving the constraint generation and solving run-times, and bringing the implementation of communication primitives closer to what is required by the protocol.

6 Related Work

SHIA [14] is a secure hierarchical in-network aggregation protocol for WSNs. SHIA is well known for introducing a hierarchical structure and the concept of optimal security to secure in-network aggregation in WSNs. It has been cited over 300 times in Google Scholar and considered in terms of efficiency in network congestion so far due to the high energy cost of data transmission in WSNs.

Chan et al.[14] claims that SHIA is a provably secure protocol, yet, as Chatterjee et al.[15] states the security proofs are provided informally. Our aim is to verify SHIA using a formal methodology and the most related works in this direction are mentioned here. Manulis and Schwenk [28] formalise optimal security using the popular sequence of games approach [37] and proves that this property is satisfied for secure aggregation protocols; however, they do not address a hierarchical scenario as in SHIA. In addition to optimal security, there is μTESLA [34] which has an essential role in SHIA by providing authenticated broadcast communication with symmetric cryptographic primitives. Ballardin and Merro [7] have formalised μTESLA using a timed broadcasting calculus for

wireless systems. Ballardin and Merro have proven that μTESLA's time dependent authentication property that takes place in the broadcast holds. μTESLA has also been modelled by Tobarra et al. [40] with the UPPAAL tool [9, 26] that uses timed automata to model real-time systems.

There are also other works with formal methods in order to verify WSN security protocols. Macedonio and Merro [27] later extended their work of modelling μTESLA with formalisation of LEAP+ [44] and LiSP [32], which are among well known key management protocols for WSNs. Tobarra et al. [38, 39] has also used AVISPA tool [5], which allows to specify security protocols with their properties via a high-level formal language, to model SNEP [34] and, then, TinySec [25], LEAP [43] and TinyPK [42]. SNEP (Secure Network Encryption Protocol) provides data confidentiality, authentication, integrity and freshness to a WSN. TinySec brings access control, message integrity and message confidentiality to link layer communication in a WSN. TinyPK allows the use of public-key based authentication and key agreement between WSNs.

There are efforts in comparing formal verification tools for modelling security protocols. One example is [18], which compares use of state spaces in AVISPA, ProVerif, Scyther and Casper/FDR. Another work [24] compares Hermes and AVISPA with respect to their complexity, front-end languages, verifiable security services, intrusion models and back-end analytical tools. Another work [16] compares Casper/FDR, STA, S^3A , and OFMC with respect to their features and ability to detect bugs under the same experimental conditions. Perhaps, the most related work among these works is [19], which implements six popular cryptographic protocols in ProVerif and Scyther in order to outline different characteristics of these tools.

7 Conclusions and Future Work

In this study, we have modelled a secure aggregation protocol for WSN (SHIA) in three modelling languages and tools (Pwb, mCRL2 and ProVerif). We needed to make some concessions on using both ProVerif and mCRL2: reliable authenticated broadcast was modelled as forwarding of point-to-point messages; authenticity was modelled with public key cryptography.

We have made significant improvements to Pwb that allowed us to model reliable authenticated broadcast, query multiple SMT solvers concurrently and cache queries for performance gain in symbolic execution. With the latest version of Pwb, we are able to represent our abstract protocol model of SHIA, which is extracted from the protocol description in [14]. At the abstraction level and size of the network that we modelled, we did not find any problems with the SHIA protocol.

As pointed out in Section 5, efficient implementation of a symbolic semantics for reliable broadcast in the presence of free choice can be considered for future work in order to achieve significant gain in the performance of constraint solving.

References

[1] SHIA. Retrieved October 19, 2015, from `http://www.it.uu.se/`

research/group/mobility/applied/psiworkbench/SHIA.

[2] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Proceedings of POPL '01*, pages 104–115. ACM, 2001.

[3] L. Aceto, A. Inglfsdttir, K. G. Larsen, and J. Srba. *Reactive Systems*. Cambridge University Press, 2007. Cambridge Books Online.

[4] J. Åman Pohjola, J. Borgström, J. Parrow, P. Raabjerg, and I. Rodhe. Negative premises in applied process calculi. Technical Report 2013-014, Department of Information Technology, Uppsala University, June 2013.

[5] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer, 2005.

[6] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, New York, NY, USA, 1990.

[7] F. Ballardin and M. Merro. A calculus for the analysis of wireless network security protocols. In *Formal Aspects of Security and Trust*, pages 206–222. Springer, 2011.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[10] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: A framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.

[11] B. Blanchet, B. Smyth, and V. Cheval. *ProVerif 1.90: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*, 2015. Originally appeared as Bruno Blanchet and Ben Smyth (2011) ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.

[12] J. Borgstrom, R. Gutkovas, I. Rodhe, and B. Victor. The psi-calculi workbench: A generic tool for applied process calculi. *ACM Trans. Embed. Comput. Syst.*, 14(1):9:1–9:25, Jan. 2015.

[13] J. Borgström, S. Huang, M. Johansson, P. Raabjerg, B. Victor, J. Å. Pohjola, and J. Parrow. Broadcast psi-calculi with an application to wireless protocols. In G. Barthe, A. Pardo, and G. Schneider, editors, *Software Engineering and Formal Methods: SEFM 2011*, volume 7041 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag, Nov. 2011.

[14] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 278–287. ACM, 2006.

[15] S. Chatterjee, A. Menezes, and P. Sarkar. Another look at tightness. In *Selected Areas in Cryptography*, pages 293–319. Springer, 2012.

[16] M. Cheminod, L. Durante, R. Sisto, A. Valenzano, et al. Experimental comparison of automatic tools for the formal analysis of cryptographic protocols. In *Dependability of Computer Systems, 2007. DepCoS-RELCOMEX'07. 2nd International Conference on*, pages 153–160. IEEE, 2007.

[17] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. de Vink, W. Wesselink, and T. Willemse. An overview of the mcrl2 toolset and its recent advances. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer Berlin Heidelberg, 2013.

[18] C. J. Cremers, P. Lafourcade, and P. Nadeau. Comparing state spaces in automatic security protocol analysis. *Formal to Practical Security*, 5458:70–94, 2009.

[19] N. Dalal, J. Shah, K. Hisaria, D. Jinwala, et al. A comparative analysis of tools for verification of security protocols. *Int'l J. of Communications, Network and System Sciences*, 3(10):779, 2010.

[20] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] C. Ene and T. Muntean. Expressiveness of point-to-point versus broadcast communications. In G. Ciobanu and G. Paun, editors, *Proceedings of FCT'99*, volume 1684 of *Lecture Notes in Computer Science*, pages 258–268. Springer-Verlag, 1999.

[22] M. Gabbay and A. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[23] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The formal specification language mcrl2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS), 27.08. - 01.09.2006*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[24] M. Hussain and D. Seret. A comparative study of security protocols validation tools: Hermes vs. avispa. In *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, volume 1, pages 6–pp. IEEE, 2006.

[25] C. Karlof, N. Sastry, and D. Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175. ACM, 2004.

[26] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[27] D. Macedonio and M. Merro. A semantic analysis of key management protocols for wireless sensor networks. *Science of Computer Programming*, 81:53–78, 2014.

[28] M. Manulis and J. Schwenk. Security model and framework for information aggregation in sensor networks. *ACM Trans. Sensor Netw.*, 5(2):1–28, 2009.

[29] D. C. J. Matthews and M. Wenzel. Efficient parallel programming in poly/ml and isabelle/ml. In L. Petersen and E. Pontelli, editors, *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*, pages 53–62. ACM, 2010.

[30] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.

[31] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 53–68, Berlin, Heidelberg, 2013. Springer-Verlag.

[32] T. Park and K. G. Shin. Lisp: A lightweight security protocol for wireless sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):634–660, 2004.

[33] J. Parrow, J. Borgström, P. Raabjerg, and J. Åman Pohjola. Higher-order psi-calculi. *Mathematical Structures in Computer Science*, 24, 4 2014.

[34] A. Perrig, R. Szewczyk, J. Tygar, V. Wen, and D. E. Culler. Spins: Security protocols for sensor networks. *Wireless networks*, 8(5):521–534, 2002.

[35] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[36] A. Riazanov and A. Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, Aug. 2002.

[37] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *IACR Cryptology ePrint Archive*, 2004:332, 2004.

[38] L. Tobarra, D. Cazorla, and F. Cuartero. Formal analysis of sensor network encryption protocol (snep). In *Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE Internatonal Conference on*, pages 1–6. IEEE, 2007.

[39] L. Tobarra, D. Cazorla, F. Cuartero, G. Díaz, and E. Cambronero. Model checking wireless sensor network security protocols: Tinysec+ leap+ tinypk. *Telecommunication Systems*, 40(3-4):91–99, 2009.

[40] L. Tobarra, D. Cazorla, F. Cuartero, and J. J. Pardo. Modelling secure wireless sensor networks routing protocols with timed automata. In *Proceedings of the 3nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, pages 51–58. ACM, 2008.

[41] S. Upadhyayula and S. Gupta. Spanning tree based algorithms for low latency and energy efficient data aggregation enhanced convergecast (dac) in wireless sensor networks. *Ad Hoc Networks*, 5(5):626 – 648, 2007.

[42] R. Watro, D. Kong, S.-f. Cuti, C. Gardiner, C. Lynn, and P. Kruus. Tinypk: securing sensor networks with public key technology. In *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 59–64. ACM, 2004.

[43] S. Zhu, S. Setia, and S. Jajodia. Leap: Efficient security mechanisms for large-scale distributed sensor networks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 62–72, New York, NY, USA, 2003. ACM.

[44] S. Zhu, S. Setia, and S. Jajodia. Leap+: Efficient security mechanisms for large-scale distributed sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 2(4):500–528, 2006.