SCIoT: A Secure and sCalable end-to-end management framework for IoT Devices *

Moreno Ambrosin[1][0000−0002−2520−9092], Mauro Conti[3][0000−0002−3612−1934], Ahmad Ibrahim[2], Ahmad-Reza Sadeghi[2], and Matthias Schunter[1]

[1] Intel Corporation, Hillsboro, OR, USA
{moreno.ambrosin, matthias.schunter}@intel.com
[2] TU Darmstadt, Darmstadt, Germany
{ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de
[3] University of Padova, Padova, Italy
conti@math.unipd.it

Abstract. The Internet of Things (IoT) is connecting billions of smart devices. One of the emerging challenges in the IoT scenario is how to efficiently and securely manage large deployments of devices. This includes sending commands, monitoring status and execution results, updating devices firmware, and interactively resolving problems.
In this paper we propose SCIoT, a Secure and sCalable framework for IoT management. SCIoT guarantees low complexity in terms of communication, storage and computation on both managed devices and the management entity. SCIoT enables secure management of large deployments with a single low-power management device, by leveraging trees of common untrusted intermediate infrastructures. SCIoT brings three technical contributions: (1) a domain-independent management specification by means of extended finite state machines, which specifies states and desired transitions to describe the whole management process; (2) a protocol for securely and efficiently distributing applicable transitions of the automaton corresponding to commands; and (3) a protocol for securely aggregating status responses from the managed nodes using a tree of untrusted nodes. We show feasibility and efficiency of SCIoT by both a proof-of-concept implementation of the client agent on Riot-OS – an operating system for the IoT, and a large scale evaluation, using realistic assumptions. Our thorough evaluation highlights the efficiency of our command distribution protocol, as well as the small (logarithmic) runtime and overhead of data collection.

1 Introduction

The increasing demand of connectivity and services that rely on distributed sensing and control is populating the world with billions of interconnected devices. Cisco [2] forecasts that 50 billion of such devices will exist in 2020. This phenomenon is commonly

* This research was co-funded by the German Science Foundation, as part of project S2 within CRC 1119 CROSSING, HWSec, the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), and by the Intel grant "Scalable IoT Management and Key security aspects in 5G systems".

called the Internet of Things (IoT). IoT devices are utilized in many different domains, ranging from small-size ecosystems, such as smart homes, to very large scale deployments for automation or distributed sensing. Examples of large IoT deployments are the experimentation facility at Santander city [27], which currently counts more than 2000 interconnected devices, and, at a much larger scale, smart metering systems, which only in the US count over 65 million devices [17].

IoT devices have constrained resources and limited (usually intermittent) connectivity. They are usually connected to edge (or gateway) devices, which provide services such as protocol translation, access to intermediate connectivity infrastructures, and data caching and aggregation at the edge of the network; these features are particularly useful in large scale deployments [12,22,20,33].

In many deployments, an efficient and effective management of IoT devices is fundamental [29]. Device management comprises critical tasks, such as distribution of commands and software updates, or device monitoring. Management processes are typically planned and controlled by systems administrators. In this paper, we consider a scenario in which a system administrator, which may have limited computational resources, needs to manage a large population of IoT devices.[4] We consider a management process comprising two main tasks: (1) broadcasting a subset of commands to targeted devices (accompanied by additional corresponding data, such as command parameters or a firmware update package); and (2) collecting statistics on the outcome of commands execution. As an example, the system administrator of a large deployment may want to know the percentage of devices that are in a correct (known) state, after a collective software update has been executed. Management operations are performed over an intermediate aggregation and cache-capable network, which is untrusted for providing data integrity or authenticity.

In the above scenario, secure and efficient management turns out to be particularly challenging: On the one hand, while solutions and standards for secure and lightweight IoT device management already exist (e.g., the work in [29], or the Lightweight Machine to Machine protocol from the Open Mobile Alliance – OMA LWM2M [25]), they are designed for *individual* device management. Therefore, unless all intermediate aggregation nodes are trusted, their cost scales linearly with the number of devices to be managed. On the other hand, existing approaches for efficient aggregate statistics collection over an aggregation tree impose a linear verification overhead on the management entity [16,34].

Contribution. This paper presents SCIoT, a framework for IoT device management that targets large deployments. SCIoT considers a layered and realistic architecture, and on top of it defines a set of protocols for scalable and secure IoT device management. In particular, this paper brings the following contributions:

- A simple domain-independent management process abstraction by means of a finite state machine, that we call Management Finite State Machine (M-FSM). M-FSM allows to express potentially complex management tasks using a concise and high-level representation.

[4] Industrial trends envision using low-power devices, e.g., a smartphone, for managing a large number of devices.

- The design of a simple, fully-cacheable, and end-to-end secure protocol for commands distribution, based on the management representation provided by M-FSM. Our protocol can sit on top of any pull-based message-response protocols. It leverages in-network caching to speed-up commands distribution. SCIoT's commands distribution protocol allows clients to "manage themselves", i.e., only selectively download the specific subset of information needed to take the next management action (e.g., a specific software update).
- The design of a protocol for scalable monitoring of large deployments. We devise an aggregation protocol based on the protocol from [16] that leverages an *untrusted* tree-based aggregation infrastructure to aggregate inbound status information, while maintaining a constant verification overhead at both device and management side, and a logarithmic traffic. Our protocol ensures that even if millions of nodes report back to a central management node, traffic and required computation at the server remains manageable.
- We implemented and tested a client device agent for Riot-OS – an operating system for resource-constrained devices – and ran a thorough experimental evaluation of our protocols via simulation (similar to [8,7]); our evaluation demonstrates the scalability of SCIoT, and its low overhead at the management side.

2 Background and Primitives

2.1 Multi-Signature

A multi-signature scheme allows a set of users to compute a signature on the same message m so that individual signatures can be aggregated into a single compact multi-signature. The multi-signature can be verified in constant time by means of a unique aggregate public key. Signature verification succeeds if *all* the computed signatures are included into the multi-signature. In this paper, we consider the multi-signature scheme in [10], built using bilinear pairings [11].

Consider three multiplicative groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p, and an efficiently computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ s.t., $e(g_1, g_2)^{xy} = g_t^{xy}$, where g_1, g_2, g_T are generators for $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$, respectively, and $x, y \in \mathbb{Z}_p$. Let $H : \{0,1\}^* \rightarrow \mathbb{G}_1$ be a hash function that maps a bitstring of arbitrary size into an element of $\mathbb{G}_1$. A multi-signature scheme is defined as follows:

Key Generation. Each signer i generates a random secret key $x_i \in \mathbb{Z}_p$, and computes its public key as $pk_i \leftarrow g_2^{x_i}$. Public keys can be aggregated into an aggregate public key $Y \leftarrow \prod_{i=1}^{n} pk_i$, where n is the number of signers.

Multisignature Generation. A signer i produces a signature σ_i on a message m as $\sigma_i \leftarrow H(m)^{x_i}$; all σ_i-s can be combined into a multi-signature $\Sigma \leftarrow \prod_{i=1}^{n} \sigma_i$, where n is the number of signers.

Multisignature Verification. Given the aggregate public key Y, the multi-signature Σ can be verified by checking whether $e(\Sigma, g_2) = e(H(m), Y)$.

This multi-signature scheme is provably secure against existential forgery under chosen message attacks in any Gap Diffie-Hellman (GDH) group [10].

2.2 Secure In-Network Aggregation

In-network aggregation allows reducing the communication overhead when performing queries and collecting statistics from nodes in large networks. In this paper, we devise a hierarchical in-network aggregation scheme with constant verification overhead. Our scheme is based on the solution from [16] and satisfies the requirement of SCIoT.

Our in-network aggregation scheme is organized in two main phases: (i) a query dissemination and response collection phase, and (ii) a result verification phase.

Collection Phase. In this phase a central querying entity (i.e., the manager in SCIoT) broadcasts a query to all nodes in the network along an aggregation tree. Then, starting at leaves, nodes recursively aggregate responses coming from their child nodes and forward the result to their parent nodes. Each node also commits to its aggregation by computing and forwarding a hash over all the responses it aggregates. The computed hash also include hashes that come from child nodes. Finally, the final aggregate response and commitment are reported back to the querying entity.

Verification Phase. In this phase the querying entity broadcasts the received aggregate response and commitment, asking nodes to check whether their contribution has been integrated correctly in that response. Each individual device verifies their correct contribution to the final response and creates an acknowledgment message and sends it the querying entity. Acknowledgment messages are authenticated using the multi-signature scheme we introduced above, which allows their secure aggregation with constant communication and verification overhead.

3 SCIoT Architecture Design

3.1 System Model

We define the system model in Fig. 1, where a manager $\mathcal{M}$, is in charge of carrying out the management of (some or all the devices in) a network G. More precisely, we consider a network of interconnected physical devices $\mathcal{D}_i \in G$ (each pictured as a dotted rectangle in Fig. 1), where each can act as one or more of the following logical entities: *endpoint* (v_j), *aggregator* (a_l), or *cache* (c_u). A endpoint v_j is the endpoint entity of the management process; v_j receives and executes commands from $\mathcal{M}$ and, upon request, provides $\mathcal{M}$ with statistical information regarding its current status. Aggregators and caches are relay entities (i.e., edge or gateway devices) that have different roles: a_l is capable of aggregating statistics collected from endpoints, while c_u caches commands distributed by $\mathcal{M}$. As a consequence, they play a role in distinct parts of the management process, i.e., c_u helps speeding up one-to-many commands distribution, while a_l has the purpose of reducing both network and $\mathcal{M}$-side computation overhead when collecting statistics from v_j.

Entities in the system are organized into two logical tree structures:[5] a *distribution tree* where inner nodes are caching entities and leaf nodes are managed entities (solid lines in Fig. 1), and an analogous *aggregation tree* that has aggregating entities as inner nodes, and managed entities are leaves (dashed lines in Fig. 1). Note that, in this model a failing inner node can be simply replaced by its parent in the tree. The connection

[5] See [8] for how aggregation trees are constructed and maintained.

Fig. 1. System model as a network of devices; each device acts as at least one of the following entities: endpoint (v_j), aggregator (a_l), and cache (c_u).

interfaces between nodes are purely logical, i.e., they do not necessarily have a one-to-one mapping with a single physical communication interface. A clear example is v_1 in Fig. 1: interactions with both c_1 and a_1 are performed internally to the physical device $\mathcal{D}_1$. Similarly, v_4 communicates with a_3 through an internal interface, while it communicates with c_3 (which is located in $\mathcal{D}_5$) through a network link.

This representation is sufficiently generic to represent different scenarios and use cases, from Wireless Sensor Networks (WSNs), where all the devices in the network act as all the three entities, to infrastructured settings, where IoT devices act as endpoint entities, while gateways represent either caches, or aggregators, or both. Note that, the definition of our management scheme is independent from the caching strategy adopted by caching entities. However, the capacity of caches together with the adopted caching policy, play an important role in improving the performance of the system. Nevertheless, this usually depends on the deployment scenario, and the capabilities of devices. Thus, we consider this to be out-of-scope.

3.2 Requirements and Assumptions

Scalability and Security Requirements. We aim at providing a highly scalable solution for management systems, which enables handling a large number of devices, through a resource constrained manager. Our goal is to reduce both computation and storage complexity for $\mathcal{M}$, while at the same time maintain a low communication and computation overhead on a_l, c_u and v_j. More precisely, we identify the following set of properties that defines a scalable and secure management system:

1. *Outbound efficiency.* The management system should guarantee an efficient broadcast distribution of management commands to endpoints.
2. *Commands freshness.* The system should provide mechanisms to allow endpoints to assess whether a received command is still valid.
3. *Inbound efficiency.* $\mathcal{M}$ should efficiently collect aggregate statistics of endpoints (e.g., the number of endpoints in a certain state).
4. *Outbound security.* It should be guaranteed that only legitimate management commands coming from the manager are executed on endpoints.
5. *Inbound security.* The integrity of the statistics collected from endpoints should be ensured.

Security Model. We assume $\mathcal{M}$ is trusted, i.e., it honestly follows the management process and protocols. We also assume that $\mathcal{M}$ issues authorized management commands

for distribution. We do not trust all the intermediate entities that are responsible for aggregation and caching, i.e., a_l, and c_u. All these entities can be under full control of the adversary. As for v_j, we assume these entities are trusted in executing management commands and providing statistical information. We assume all devices that contain a v_j to have the necessary security hardware that protect v_j from compromise (e.g., TrustLite [24]). Finally, we consider a stealthy adversary that aim at manipulating the management and collection process without being detected. Thus, we consider Denial of Service (DoS) attacks that aim at undermining the availability of these services to be out-of-scope.

Attacker Goals. The goals of the adversary controlling c_u are to: (i) Tamper with commands sent by $\mathcal{M}$; and (ii) Impersonate $\mathcal{M}$ issuing commands to v_j. Analogously, an adversary controlling one or more aggregating entities a_l, has the following goals: (a) Tampering with the statistics collected from one or more devices; and (b) Impersonating a device sending fake statistics to $\mathcal{M}$.

3.3 FSM Abstract Specification of Management Objectives

An important component of SCIoT is the abstraction we use to decouple domain-specific management requirements from the actual realization of the management process. Such abstraction allows to define a management-independent communication protocol between endpoints and $\mathcal{M}$, which is both simple and highly scalable. The main intuition behind this abstraction is to allow $\mathcal{M}$ to carry out the whole management process by simply serving, upon devices' request, a set of static (and therefore cacheable) contents. These contents are efficiently delivered to the endpoints by leveraging the intermediate caching entities c_u.

We represent our management process specification by means of an extended finite state machine, that we call Management Finite State Machine (M-FSM). M-FSM represents, in its minimal form (i.e., sub-M-FSM), a single command execution. Sub-M-FSM comprises (see Fig. 2):

- At least three *states* a device can assume: (1) a *starting* state, representing a device waiting for a command to execute; (2) an *attempted execution* state, representing the device after the execution of the command; and (3) at least one *termination* state (e.g., a system failure). Each state is uniquely identified by an ID SID.
- At least two *transitions*: (1) one transition from the starting state to the attempted execution state. This transition is labeled by an `execute` event and a corresponding `COMMAND` action (i.e., a command to execute); and (2) at least one transition ending to a terminal state. Actions are executed by the function Execute, and may write into global variables. In particular, the `COMMAND` writes its outcome (i.e., the return code of the command) in the `out` variable. Outgoing transitions from the attempted execution state are labeled with a `switch` event, parametrized on the value of the `out` variable, and an `OTHER_ACTION` to execute. These transitions can "point" to either a terminal state, or the starting state of another sub-M-FSM.

Fig. 2 provides a graphical representation of a sub-M-FSM, where ovals represent states, and arrows represent state transitions. Events and corresponding actions are placed on top of each transition and separated by "|". Boolean guards, based on which transition is chosen, are indicated within squared brackets. The sub-M-FSM in

Fig. 2 represents a single command execution (or may represent a loop, in case the sub-M-FSM has a transition from the executing to the starting state). More complex execution processes can be obtained combining several sub-M-FSMs, to represent the execution of consecutive commands where the execution of a subsequent command depends on the successful execution of the previous one. This is done by adding an outgoing transition (based on the outcome of the command) from the attempted execution state to the starting state of another sub-M-FSM.

Fig. 2. Basic sub-M-FSM. A device in "Starting" state executes the only transition to the attempted execution state, performing an action Execute. Depending on the outcome (e.g., return code) `out` of Execute, the device might follow one of the outgoing transitions: to the starting state, to a termination state, or to (the starting state of) another sub-M-FSM.

M-FSM Composability and Overhead. It is worth noticing that, as the M-FSM is a composition of single sub-M-FSMs, each representing a command execution, in a management process the M-FSM can be arbitrarily incremented with additional M-FSMs *over time*. This property is particularly useful in the management scenario, as it allows to model management processes that cannot be completely defined statically, such as subsequent firmware/software update releases. As a consequence, from an endpoint perspective, at a generic point in time t_i the entire management process can be represented only as the *current* command to execute. This guarantees an almost *constant* overhead at the endpoint.

Use Case Example (Device Firmware Update M-FSM). An interesting use case M-FSM is the (simplified) device firmware update process shown in Fig. 3. A single device update process is composed of an update installation phase, and a recovery attempt phase. These two phases are represented by analogous sub-M-FSMs. The update process starts from a "Not Updated" state (S1); the `execute` transition (and the consequent execution via Execute of `UPDATE`) brings the device into an "Update Attempted" state (S2). The function Execute writes its outcome (e.g., an integer code) into the global variable `out`. Based on `out`, the device follows a specific `switch` transition, and executes the `NULL` action (i.e., no action is executed). In case of `FATAL_ERROR`, the process moves to a terminal "System Failure" state (S3). If, instead, the update process terminates successfully (i.e., `out` $==$ `SUCCESS`), the device jumps to the starting state of the next sub-M-FSM in the process specification.[6] Finally, if the update process encountered a recoverable error (`SIMPLE_ERROR`), it switches to a recovery phase, jumping to

[6] New "Not Updated" state, which will have a different SID w.r.t. the previous analogous state.

the initial state "Erroneous State" of the Recovery Phase sub-M-FSM. In such phase, the device tries to recover the previous software state by executing a `RECOVERY` action with the function `execute`, jumping to a "Recovery Attempted" state. The outcome of `execute` is written into `out2`, which is used to switch to an end state (representing a fatal unrecoverable error), or to the previous "Not Updated" state.

Fig. 3. Example: Firmware update management.

Note that, in order to avoid an infinite number of attempts, the action `RECOVERY` maintains a counter, recording the number of attempts made by the device; if this number is greater than a threshold, `execute` will return a `FATAL_ERROR` (this is not shown in Fig. 3 for simplicity). Furthermore, while shown in Fig. 3 as a transition to a different state S7, in practice, in order to avoid state explosion [32], S2 `switch` transition may simply return to S1, which represents a "Not Updated" state, but with a different SID.

4 SCIoT Protocols

4.1 A Scalable Self-Management Protocol

The first main component of SCIoT is a simple and scalable protocol to distribute management commands from $\mathcal{M}$ to endpoints v_j. Commands distribution is based on an M-FSM specification (e.g., firmware update M-FSM in Section 3.3). Based on abstraction provided by the M-FSM, we designed a secure pull-based message-response protocol which allows: (1) domain-independent device management; (2) efficient cacheable distribution of management commands, suitable for caching networks or content delivery networks; and (3) minimal storage requirement on endpoints.

In order to simplify the exposition, in what follows we detail our self-management protocol between a single endpoint v_j, and $\mathcal{M}$.

The main idea behind our protocol is the following. Each endpoint v_j "moves" inside the M-FSM maintaining information about its current state only, while pulling the next available transition from $\mathcal{M}$. More precisely, v_j pulls either: (a) An `execute` event, and corresponding `COMMAND` action, from a starting state; or (b) A `switch` event and corresponding `OTHER_ACTION` action from an attempted execution state. v_j queries $\mathcal{M}$

issuing a *request message* (req) that is forwarded through intermediate c_u entities. $\mathcal{M}$ then responds with a *response messages* ($resp$). Note that, caching entities may cache response messages, before serving them back to the querier, to better serve "bursty" requests and reduce latency. This is particularly important when devices request large payloads, such as firmware updates [6]. This communication model is supported by existing application level protocols (such as CoAP [14], which implements a message-response protocol on top of UDP), as well as by recently proposed information-centric protocols (such as Named-Data Networking [23]).

Fig. 4. Self-management protocol using μTesla. Here, we assume v_i already has a commitment (i.e., a key it trusts) corresponding to time interval $\tau - 2$.

Protocol Description. As shown in Fig. 4, from a state SID, v_j queries $\mathcal{M}$ for the next available transition (and event-action pair). More precisely, v_j sends a req message, which contains v_j's current state ID SID, and a list of key-value pairs $[< var_1 : val_1 >, \ldots]$ indicating M-FSM variables, and their current value. These parameters are used by $\mathcal{M}$, or by caching entities, to select the matching response packet to return to v_j. Note that, the way SID and the key-value pairs are included as parameters of v_j's request depends on the adopted underlying transport protocol.

The response supplied by $\mathcal{M}$ contains the next event and action to execute (using the function Execute). Once the command in action is executed, v_j jumps to the next attempted execution state, and issues a new request message req'. The endpoint then obtains a new event and action to execute and move to the next M-FSM state, which can be either terminal or starting state – MoveToState.

In case of large command payloads, e.g., a new firmware, the action specifies only a "pointer", e.g., a hash of the payload, to use for (potentially cached) payload retrieval. v_j then downloads the payload in an additional step. Note that, as caching entities may directly respond to req with a cached response, we added a timestamp parameter t and a validity interval Δt to each (signed) response returned to v_j. In this way, endpoints can determine whether a received transition (or command payload) is "fresh", i.e., not expired according to t and Δt. In order to guarantee availability, intermediate caching

entities must ensure that devices are able to detect whether a content is fresh or not, and should provide mechanisms to "force" requests to be served directly from the source.[7]

Protocol Security. SCIoT works in conjunction with several security layers suitable for large scale broadcast distribution. In particular, in SCIoT $\mathcal{M}$ may either use digital signatures, or μTesla authenticated broadcast protocol [26] to authenticate management commands. Using μTesla, SCIoT's management automation protocol guarantees public verifiability for resource-constrained devices (i.e., devices able to compute only basic cryptographic operations, such as hash functions and Message Authentication Codes – MACs), while preserving the cacheability of the distributed data.

Depending on the authentication mechanism in use, responses generated by $\mathcal{M}$ are sent along with either a digital signature, or a MAC. In the case of digital signatures, $\mathcal{M}$ signs each response with its secret key $sk_{\mathcal{M}}$ and endpoints verify it using $\mathcal{M}$'s public key $pk_{\mathcal{M}}$. On the other hand, while using μTesla $\mathcal{M}$ attaches a MAC to each response, computed using a symmetric key k_τ that is valid only within a certain time interval τ. At time $\tau + d$, k_τ is disclosed, i.e., broadcasted in a special packet. Endpoints can then verify the MAC on the buffered response packets received during time interval τ [26]. In detail, v_j downloads the next transition packet from $\mathcal{M}$ at time τ, and stores it in a local cache. v_j verifies the message at time $\tau + d$, i.e., after receiving the broadcasted key k_τ. This process is shown in Fig. 4. In order to build a cryptographically verifiable key series, $\mathcal{M}$ makes use of one way hash chains, i.e., the key used at time τ is obtained as the hash of the key that will be used at time $\tau + 1$ [26]. Note that, different applications may require different key disclosure time intervals. For this reason, $\mathcal{M}$ keeps several *key sequences*, generated from different hash chains and have different key disclosure time intervals. Upon receiving a request req, $\mathcal{M}$ computes the MAC on each response using different keys. The key sequence to be used is specified in req.

While the digital signature is permanently cacheable, MACs have an expiration period, which corresponds to the key disclosure time. Endpoints are free to choose between requesting a response with a digital signature or a MAC. In other words, endpoints can autonomously determine the best trade-off between computation overhead and the delay in the reception of the data. Devices choose between different options based on a set of factors, including their computational power, remaining energy, and the time limits specified by the application. Moreover, endpoints can choose between MACs with different "delays" (i.e., key disclosure interval $\Delta\tau$) based on their degree of synchronization. This provides a trade-off between security level and response delay. The number of MACs and the time interval for each hash chain are design parameters that may depend on the properties of the network (e.g., bandwidth or size), and on the requirements for different applications.

[7] This feature is transport specific: In content-centric protocols such as Named-Data Networking (NDN) [23], content freshness is controlled by flags contained inside headers, i.e., via data packet's `Freshness` and interest's `MustBeFresh` header fields. In CoAP [14], however, this is not possible. Response packets carry a `Max-Age` option indicating that the response is to be considered not fresh after its age is greater than the specified number of seconds.

4.2 Scalable Device Monitoring and Assessment

The protocol described in Section 4.1 alone enables managed entities to execute available commands, perform state transitions, and conduct error recovery as specified by the management finite-state automaton. However, it does not allow the management layer to learn to what extent the management strategy has been successful. A simple example is that $\mathcal{M}$ would not learn if a given firmware update always leads to failures. More generally, $\mathcal{M}$ needs to collect and maintain statistics, such as the percentage of endpoints that are in a certain state in the update process shown in Section 3.3.

Naïve approach. A naïve approach for device state assessment would be by requesting the required information from each device individually; $\mathcal{M}$ could broadcast a challenge, and collect the individual responses from endpoints. This approach, however, is hard to scale, as it would result in $\mathcal{O}(|G|)$ traffic and verification complexity.

In-Network Aggregation. A more scalable way to collect the global network state is relying on in-network aggregation. Each device reports its state to its upstream aggregating node. This, in turn, computes the aggregate sum of each value coming from its children and forwards it to its parent aggregating node in the internal tree structure, and so on. Using authenticated channels, $\mathcal{M}$ can efficiently verify the authenticity of the received aggregate counts. This simple approach has been adopted in several solutions, such as in [8]. However, a major important drawback of simple aggregation is the absence of end-to-end integrity in presence of malicious aggregating entities, i.e., in-network aggregation requires fully trusted aggregators [7].

Secure In-Network Aggregation. Our approach for collecting statistics on endpoints over untrusted aggregators is based on the hierarchical secure in-network aggregation scheme presented in Section 2. It allows: (1) using in-network aggregation to compute an aggregate value, and (2) integrity verification by $\mathcal{M}$ in *constant time*. Recall that aggregation in SCIoT is performed by logical aggregating entities, which (similarly to [8,7]) can form an overlay aggregation tree rooted at $\mathcal{M}$, where aggregating entities a_l are inner nodes, and v_j are leaves. Finally, aggregating nodes are also untrusted for authenticity of aggregation. The overall protocol runs as follows:

- The manager $\mathcal{M}$ broadcasts the state it is interested in collecting statistics for (either signed with $\mathcal{M}$'s secret key, or using an authenticated broadcasts protocol, such as the one described in Section 4.1).
- Each endpoint v_j responds with 1 if it is currently in that state, and with 0 otherwise.
- Intermediate aggregators sum the received values, and forward the computed value up to $\mathcal{M}$.
- After collecting the aggregate value computed on phase (i), $\mathcal{M}$ broadcasts the final aggregate result authenticated in the same manner as above.
- Based on the commitments (see Section 2), endpoints can verify that their contribution has been added to the aggregate value. If this is the case, each endpoint v_i produces a multi-signature σ_i on a pre-established "OK" message using its secret key sk_i. Otherwise (in case the verification fails), it sends a negative acknowledgment (NACK) to its gateway aggregator.
- Aggregators combine all the signatures (along the formed overlay aggregation tree) according to the multi-signature scheme described in Section 2, and finally deliver a single aggregate signature Σ to $\mathcal{M}$.

– $\mathcal{M}$ can verify the signature using the pre-computed aggregate public key Y.

Note that, in the case in which the verification fails, $\mathcal{M}$ can conclude that an error happened, i.e., the contribution of a node was lost, or that some aggregator maliciously modified either the aggregate value, or the signature.

Inspecting Individual Devices. The protocol discussed in the previous sections, count the devices in each given state. However, in some cases, inspection of a given small number of devices may be desirable. In order to enable device inspection, the manager can issue a call-back command to all endpoints in a given state. This command triggers the devices to "call home", report their ID, and then be available for further debugging. To enable this, an endpoint can be "probed" by $\mathcal{M}$, and respond with the identifier of its current status in a signed response message. Note that, unless debugging is constrained to few devices, this might quickly create a bottleneck on the whole system, especially in the case in which $\mathcal{M}$ needs to collect several periodical statistics from the devices.

5 Prototype Implementation

We implemented SCIoT's client agent as a module for Riot-OS [9,21] (i.e., targeting IETF Class 1 and 2 devices [13]). This module implements both SCIoT's commands distribution protocol, and responds to device assessment requests from $\mathcal{M}$. $\mathcal{M}$ implementation is fairly simple, as it consists in a simple server application that exposes basic APIs (later discussed in this section), and periodically queries devices; for this reason, it will not be discussed in this section.

Riot-OS [9,21] is an operating systems suitable for resource constrained environments. It implements a micro-kernel architecture, and allows applications to include only the minimum modules necessary for their execution. Furthermore, Riot-OS does not differentiate between processes and threads. Each application runs on its own thread of execution, but can freely create other threads (the limit in number is given by the available memory). Our client implementation module exposes a concise set of APIs, and can be easily utilized by applications to automate management tasks.

Our implementation uses CoAP [14] for both M-FSM management, and to deliver statistics collection queries from $\mathcal{M}$ to endpoints.

The device agent runs on its own thread of execution (see Fig. 5), and interacts with a simple CoAP server. An application that needs to carry out a management process should wait for transitions (i.e., commands) coming from the agent via Riot-OS IPC (Inter-Process Communication), and react accordingly, i.e., execute a command with a specific ID. The device "talks" to a server via a minimal set of CoAP REST APIs. The server runs either at the manager, or on an edge node, which may act as a proxy and translate CoAP requests into HTTP [18]. The client device requests transitions by issuing a CoAP request

$$\texttt{coap} : //[\texttt{SERVER\_IP}]/\texttt{sid}?\texttt{sid} = \texttt{SID}\& \ldots,$$

where SERVER_IP is either the IP address of $\mathcal{M}$, or of the 1st-hop aggregating node, and $\texttt{sid} = \texttt{SID}$ is the only mandatory parameter of the query. Similarly, the agent running on the device accepts CoAP assessment requests for a state ID SID, of the form:

$$\texttt{coap} : //[\texttt{BROADCAST\_IP}]/\texttt{assess}/?\texttt{nonce} = \texttt{N}\&\texttt{sid} = \texttt{SID}.$$

Fig. 5. Client agent module for Riot-OS.

6 Performance Evaluation

In this section, we present an evaluation of our solution, based on our implementation presented in Section 5, and on an emulated, yet realistic setting. Our considered setting consists of low-end devices compatible in capabilities with M3 Open Node devices from the IoT-Lab/SensLAB testbed [3]. These devices are featured with an ARM Cortex M3, 32-bits microcontroller running at 72 MHz, 64 Kbyte of RAM, and a 2.4 GHz IEEE 802.15.4 capable transceiver [4]. Moreover, we consider $\mathcal{M}$ to be a low-cost medium-power device, compatible with a Raspberry Pi Mod B, i.e., equipped with a 700 MHz CPU, 512 Mbyte of RAM, and 2 Gbyte of storage.

We implemented the multi-signature scheme we introduced in Section 4.2, based on the embedded system library in [31]; we used the mbedTLS library [1] for the remaining cryptographic operations: SHA-1 based HMAC (Hmac_1), and ECDSA. We evaluated the approaches we presented in Section 4 at large scale using network simulation.

6.1 Storage overhead

Aggregating nodes, a_l, do not need to store any information. Caching entities have a storage overhead which depends on the size of their cache, and the data currently contained in it. An endpoint v_i keeps in its persistent storage: (i) $\mathcal{M}$'s public key $pk_{\mathcal{M}}$ (32 byte in case of public key), or the commitment for the whole key chain (20 byte in case of μTesla [26]); (ii) the current state of the M-FSM, which comprises the ID SID_j (2 byte); (iii) $\mathcal{D}_i$'s public and private multi-signature keys (256 byte and 32 byte, respectively). The overall storage requirement of each device is 322 byte, if public key is used, and 310 byte if μTesla is used. Low-end devices targeted by SCIoT have at least 1024 bytes of secondary memory [7], and thus SCIoT will use 31.4% of it when the public key is used, and 30.3% otherwise.

6.2 Communication overhead

We now provide an estimate of the bytes transmitted between an endpoint v_j, and $\mathcal{M}$. In general the use of μTesla generates an overhead of one key release (approx 30 byte [26]) per time interval τ of each time series. Note that, we focus only on the overhead introduced by SCIoT protocols, and thus, we do not include the overhead generated by the underlying protocol stack.[8]

[8] Typically, the stack comprises CoAP, 6LowPAN, IPv6 and 802.15.4. Additional overhead is introduced by protocol headers, plus possible segmentation or fragmentation.

Commands Distribution. When requesting a transition, $\mathcal{D}_i$ produces a request indicating the ID SID of its current state, and, if using μTesla, the parameter $\Delta\tau$, indicating the time series $\mathcal{D}_i$ is using. This generates at most as little as 6 bytes. $\mathcal{M}$ sends out a packet comprising a transition (TID, SID_S, SID_D, and a command), a timestamp t, a validity interval Δt, and an authenticator (i.e., a digital signature or a MAC). Referring to our implementation in Section 5, and considering 4 bytes for both t and Δt, the overall communication overhead of command distribution protocol is between 80 and 334 byte, when using digital signatures, and between 37 and 291 byte, when using μTesla.
Device Assessment. In the first phase of this scheme each device sends a 26 byte *label*. The amount of bytes generated by the second part of the protocol is logarithmic in the size of the network. More precisely, the overhead of this protocol varies based on the height of the aggregation tree, and the number of leaf endpoint nodes. This overhead is mainly due to the *off-path*[9] information required by the scheme to allow each device to verify whether its contribution has been added to the aggregate value. The off-path values are locally cached by each aggregating node during the data collection, and re-distributed by the network in the second step of the scheme. Each label has a size of 26 byte. Thus, let h be the height of the tree formed by aggregating nodes (only), and l the number of leaves (i.e., endpoints) connected to the last layer of the aggregating tree; the total communication overhead on each endpoint, in terms of received data, is $26 \times (h + l)$ byte. As an example, consider a binary tree, and let $l = 2^4 = 16$, and $n = 2^{10}$; in this case, $h = 14$, and thus, the average amount of bytes received by each endpoint will be 780 byte. Finally, the acknowledgment sent by each endpoint (and aggregated by aggregators) consists of 84 byte (a 20 byte nonce, and a 64 byte multi-signature).

6.3 Runtime

We estimate the runtime of both the command distribution protocol (Section 4.1), and the statistics collection protocol (Section 4.2). Execution time is mainly dominated by cryptographic operations, and data transmission. Table 1 shows the time overhead introduced by the adopted cryptographic operations on two types of devices: M3 device (low- end) from IoT-LAB, and Raspberry Pi Mod B (higher-end).

Table 1. Cryptographic overhead

Function	Time (ms)	
	M3 IoT-LAB (Endpoint)	**Raspberry Pi Mod B (Aggregator)**
$H(m) \in \mathbb{G}_1$ [(1)]	360.319	89.168
$g_1^x,\ g_1 \in \mathbb{G}_1$	494.619	124.604
$g_1 \times g_1',\ g_1, g_1 \in \mathbb{G}_1$	23.615	8.459
$e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$	– [(2)]	1.736
Hash_1 [(3)]	0.102	0.031
Hmac_1 [(3)]	0.408	0.124
ECDSA Verify [(3)]	1181.140	– [(2)]

[(1)] Computed on a 20 bytes nonce
[(2)] Not performed by the device during the protocol
[(3)] Computed on 64 bytes

[9] For each node, off-path information are the commitments of every child nodes of each node that is on its path to the manager.

In addition to real world implementation and testing, we evaluated scalability of SCIoT based on a large scale simulation using the OMNeT++ discrete event simulator [5]. We considered two different settings: (I) An infrastructured setting where low-end devices, acting as endpoints, are directly connected to higher-end nodes, which form a layer of aggregators and caches; and (II) an ad-hoc setting comprising low-end devices acting as both endpoints, aggregators and caches. We simulated the execution of the various protocol operations by adding respective delays. Furthermore, we configured the communication rate for links among low-end devices, and between them and high-end devices, to 75 Kbps, i.e., the effective measured data rate for ZigBee, a common communication protocol for IoT devices [30]. We set links among high-end devices (comprising manager), with a bandwidth of 10 Mbps.

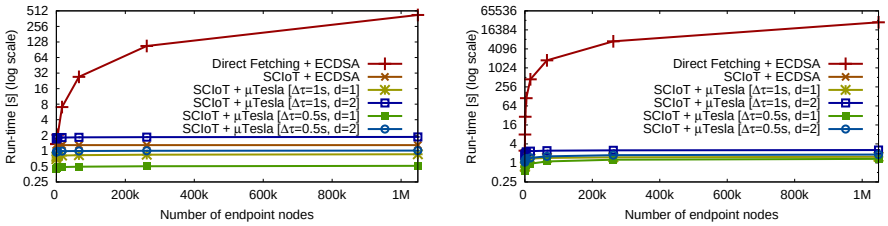
Setting (I) has a variable number of low-end nodes (i.e., endpoints), between 2^6 and $2^{20} = 1,048,5761$; the layer of aggregators and caches is internally organized as a binary tree, e.g., as an overlay. We set the size of this intermediate layer to be proportional to the number of low-end devices, i.e., the number of endpoints per aggregator/cache is constant. We indicate with r the ratio between the number of high-end nodes acting as aggregators/caches, and low-end devices. For simplicity, we assume the tree configuration is static, and pre-determined; as an example, this may be the case of an infrastructure supporting data collection in a smart city scenario.

Setting (II) comprises a variable number of low-end devices that embody all the three entities, between 2^6 and $2^{20} = 1,048,5761$. Similarly, we assume low-end devices can form a binary tree, rooted at the manager.

Commands Distribution. We configured setting (I) with $r = 32$. Caches use a First-In-First-Out (FIFO) policy. Endpoints (i.e., low-end devices) request a transition from $\mathcal{M}$, starting at a random time between 0 and 1 s, and can either verify a digital ECDSA signature on the received response, or use μTesla; in the latter case, the endpoint waits for the subsequent key disclosure interval $\tau + d$ (in our setting, we considered $\Delta\tau \in \{0.5, 1\}$ s, and $d \in \{1, 2\}$ s) to fetch the necessary information and verify the response from $\mathcal{M}$. Similar to [6], we compared direct fetching, and cache-aided fetching of transitions (the latter is enabled by SCIoT); we measured the average time it takes for an endpoint low-end device to fetch a transitions from $\mathcal{M}$. Results are shown in Fig. 6. As expected the distributed caching of responses helps speed up the response fetching for a given request: The download time grows logarithmically in the size of the device population. Moreover, with the considered parameters, μTesla with $d = 1$ shows a reduced overhead than using digital signatures; this, however, comes at the price of a more complex and expensive key management, and stricter constraints (e.g., each device must be loosely synchronized with $\mathcal{M}$) [26].

This simple experiment shows the scalability of our protocol, which indeed maximizes the cacheability of each response issued by $\mathcal{M}$. These results are in line with previous evaluation, such as the one in [6], where the experiments where conducted on top of a Named-Data Networking (NDN) network [23], but on smaller scale.

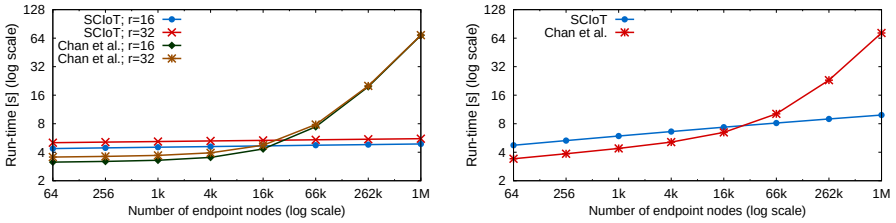
Device Assessment. We compared our in-network aggregation scheme to the work from [16]. We evaluated these protocols in the same settings, settings (I) and (II), used in the evaluation of the commands distribution protocol. In Setting (I) the ratio between the number of endpoints and aggregators is constant. Results are shown in Fig. 7. In general,



(a) Setting (I) (b) Setting (II)

Fig. 6. Commands fetching in SCIoT.

we observe that the runtime introduced by the protocol in [16] grows linearly in the number of endpoints, while the runtime of our scheme grows logarithmically with the number of endpoints. The most expensive part of the protocol in [16] is the verification of the acknowledgments received by $\mathcal{M}$, which consists of computing linear number of HMACs (i.e., n). Instead, our scheme that is adopted by SCIoT, introduces a *constant* overhead for such verification.



(a) Setting (I) (b) Setting (II)

Fig. 7. Device assessment overhead. Axes are in logarithmic scale.

The runtime of both [16] and our aggregation scheme depends also on the depth of the aggregation tree, which in our settings depends on the ratio between the number of endpoints r and aggregator nodes; in our setting, the runtime is higher when $r = 32$, compared to $r = 16$. This is due to the required off-path information that the network must provide to endpoints, and the derived computation for verifying the inclusion of each endpoint. As previously mentioned in Section 6.2, this is proportional to both the height of aggregation tree, and r.

For small-medium scale settings, the scheme from [16] is more efficient than our scheme, requiring less than 4 s to complete the assessment. Indeed, computing a multi-signature costs more than computing a Hmac for low-end devices. However, in case of very large settings the runtime of the scheme from [16] quickly grows, requiring a non-negligible overhead on $\mathcal{M}$. On the other hand, the use of multi-signatures presents a much more manageable overall overhead. As an example, considering $r = 16$ in our evaluation setting, when number of endpoints is $32,768$ the use of multi-signatures shows an improvement in system's scalability: The runtime grows slowly compared to the scheme from [16], taking 4.7 s to run an assessment (compared to 5.4 s of [16]). This suggests the possibility of using an hybrid approach tailored to the specific setting, where $\mathcal{M}$ can select the protocol to use depending on the number of endpoints.

7 Security Consideration

We now briefly discuss the security of our management system, w.r.t. our requirements. We consider a probabilistic polynomial time (PPT) adversary $\mathcal{A}$, whose target is twofold: (1) inject fake commands, i.e., transitions, inside the network of devices, with the aim to interfere with the management process (i.e., with the protocol in Section 4.1) and thus fooling *benign* endpoints into performing different actions than the ones specified by the M-FSM; (2) manipulate the aggregate state collected by $\mathcal{M}$ (i.e., interfere with the protocol in Section 4.2), and make $\mathcal{M}$ accept such manipulated value, that does not reflect the values reported by endpoints. In order to perform the attack, $\mathcal{A}$ can compromise one or more aggregators or caching entities, i.e., a_l or c_u, or act as a man-in-the-middle. Furthermore, $\mathcal{A}$ can also compromise a limited number of endpoints v_j. However, we assume that the number of compromised endpoints is too small to influence the collected statistics.

We formalize goals (1) and (2) as two security experiments: $\mathbf{Exp}_1$, between $\mathcal{A}$ and a benign endpoint v_j, and $\mathbf{Exp}_2$, between $\mathcal{A}$, and v_j and $\mathcal{M}$, respectively. In $\mathbf{Exp}_1$, after a polynomial number of steps by $\mathcal{A}$, in terms of the security parameters ℓ_{Sign}, ℓ_{Hash}, and ℓ_{MAC}, v_j outputs $o_1 = 1$ if it accepts the received transition, or $o_1 = 0$ otherwise. Similarly, in $\mathbf{Exp}_2$ after a polynomial number of steps by $\mathcal{A}$ in terms of ℓ_{Sign} or ℓ_{Hash} and ℓ_{MAC}, and ℓ_N, $\mathcal{M}$ outputs $o_2 = 1$, if it accepts the manipulated aggregate value, or $o_2 = 0$ otherwise.

Definition 1 (Secure management service). *A management service is secure if* $Pr[o_1 = 1|\mathbf{Exp}_1(1^\ell) = o_1]$ *is negligible in* $\ell = f(\ell_{\mathsf{Sign}}, \ell_{\mathsf{Hash}}, \ell_{MAC})$*, and* $Pr[o_2 = 1|\mathbf{Exp}_2(1^\ell) = o_2]$ *is negligible in* $\ell' = f'(\ell_{\mathsf{Sign}}, \ell_N, \ell_{\mathsf{Hash}}, \ell_{MAC})$*; the functions* f *and* f' *are polynomial in all the parameters specified.*

Theorem 1 (Management service security). *Our management service is secure, according to Definition 1, if both the adopted multi-signature scheme and the public key signatures are unforgeable, and* μ*Tesla is secure.*

Proof (Proof (Sketch)). We now provide an intuition of our statement regarding the security of our scheme.

(1) $Pr[o_1 = 1|\mathbf{Exp}_1(1^\ell) = o_1]$: v_j outputs $o_1 = 1$ iff $\mathsf{IsValid}(resp) = true$, that is, if the verification of the digital signature, or MAC in case of using μTesla, σ taken over $\{\text{TID}, ..., t, \Delta t\}$ is valid. In order to carry out this attack, $\mathcal{A}$ can create a new response with a signature σ' attributed to $\mathcal{M}$. If $\mathcal{M}$ uses public key signatures, e.g., using RSA, $\mathcal{A}$ should be able to forge σ. However, using an unforgeable public key signature scheme, the success probability for $\mathcal{A}$ is negligible in ℓ_{Sign}.

In case of using μTesla, authenticity and integrity of the received transition is guaranteed by a MAC. In this scenario, however, besides trying to forge the MAC σ (which has negligible success probability in ℓ_{MAC}), $\mathcal{A}$ may also try to use an older key $k_{\tau'}$ belonging to a time interval $\tau' < \tau$, where τ is the current time interval, to compute the MAC on the response, for the time interval τ. Recall that, a key sequence is created from a reverse hash chain, in a way such that: $k_{\tau-1} \leftarrow \mathsf{Hash}(k_\tau)$; thus, for the properties of hash algorithms, the probability of $k_{\tau-1} = k_\tau$ is negligible in ℓ_{Hash}.

(2) $Pr[o_2 = 1|\mathbf{Exp}_2 1^\ell) = o_2]$: $\mathcal{A}$ can perform the following attacks on the assessment protocol: a) attack part (i) of the device assessment protocol by modifying the

value sent by $\mathcal{M}$ to v_j; b) attack part (ii) of the protocol by creating a valid acknowledgment of v_j, using an old signature σ_{old} from a previous interaction; or c) attack part (ii) of the protocol by creating a fake acknowledgment with a multi-signature σ that attributes to v_j.

In order to perform the attack (a), $\mathcal{A}$ should be able to either forge a signature generated by $\mathcal{M}$, or to violate the security of μTesla; this is unfeasible for $\mathcal{A}$, similar to (1). Finally, strategies (b) and (c) are unfeasible for a PPT attacker like $\mathcal{A}$, due to the security of the multi-signature scheme against existential forgery attacks.

8 Related Work

Device Management. The Lightweight Machine to Machine protocol (LWM2M) [25], proposed by the Open Mobile Alliance (OMA), is a protocol designed for secure device management. Unfortunately, while certainly a valid solution, the protocol is intended for management of individual devices, and therefore not suitable in our scenario. In general, previous work in the literature either focus on network management for IoT devices [28], or consider scenarios where devices can be managed individually [29]. We consider all the above works to be complementary to ours; they can be used, for example, to perform one time bootstrap operation, topology maintenance, or individual device inspection. In [6] Ambrosin et al. proposed a protocol for efficient and secure delivery of confidential software updates to devices, by leveraging untrusted inner cache enabled networks. The authors provided the description of their solution over a Named-Data Networking (NDN) based inner network. However, different from our work, the authors did not provide an efficient protocol to collect device statistics. Burke et al. [15] presented a secure NDN-based security architecture for instrumented environments, such as building automation systems, and in particular for one of its sub-domains, i.e., lighting control. Their proposed solution provides privacy and authenticity for both command and acknowledgment messages, but unfortunately does not provide multicast features, i.e., for management of multiple devices, the management entity must issue multiple individual commands.

Secure Data Aggregation. There is a rich literature dealing with secure in-network data aggregation, especially in the context of Sensor Networks (SN), and Wireless Sensor Networks (WSN). These approaches are typically executed on top of an aggregation tree, and allow to combine the contribution of each node in a secure way, i.e., in a way that is *verifiable* by the collector node. In other words, the collector can verify that the aggregate result has not been tampered by inner aggregator nodes, and that all nodes contributed[10] to the computed aggregate value. Secure aggregation protocols usually focus on limiting communication and computation overhead for end nodes, and in the network, but pay less attention to the overhead at the verifier, which is assumed to be powerful enough to perform a (usually linear) number of cryptographic operations to verify the aggregate result. However, in our scenario, i.e., in case of large scale network managed by a low/medium power entity, the complexity at the management entity should be reduced as much as possible. In the following, we discuss only some related protocols. In [16], Chan et al. propose a secure data aggregation scheme for SN and

[10] This does not apply to every in-network data aggregation scheme.

WSN. Overall, the algorithm incurs in $\mathcal{O}(\Delta \log^2 n)$ node congestion, where node congestion is the worst case communication load on each sensor node. Frikken et al. [19] further reduces the node congestion of [16] to $\mathcal{O}(\Delta \log n)$, proposing a new commitment structure. Unfortunately, both schemes impose a linear verification overhead on the collector node, which needs to compute the XOR of all MACs created by end nodes. A different approach is considered by Yang et al. in SDAP [34]. SDAP is a non-exact mechanism which reduces the complexity of the verification while adding an (albeit small) overhead on the data collector.

9 Conclusions

In this paper we present the design of SCIoT, a framework for scalable and secure IoT device management. SCIoT represents the management process using an abstract finite state machine, thus decoupling it from its specific domain. Based on this representation, we design a protocol that allows devices to efficiently retrieve control messages, such as commands or firmware updates, from the management control entity. Another important feature provided by SCIoT is the ability for the control entity to monitor the status of the managed devices (e.g., number of devices that are in a given state). This is done by efficiently collecting device state information. Messages carrying device statistics are securely aggregated by an inner aggregation network, to minimize communication and computation complexity. Our evaluation shows the benefits of our approach in terms of improved scalability and manageable overhead.

References

1. ARM® mbedTLS cryptographic library. `https://tls.mbed.org/` (2016)
2. Cisco Forecast on Internet of Things. `newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342` (2016)
3. IoT-LAB: a very large scale open testbed. `https://www.iot-lab.info/` (2016)
4. IoT-LAB M3 Open Node. `https://www.iot-lab.info/hardware/m3/` (2016)
5. Omnet++ Discrete Event Simulator. `https://omnetpp.org/` (2016)
6. Ambrosin, M., Busold, C., Conti, M., Sadeghi, A.R., Schunter, M.: Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution over Untrusted Cache-enabled Networks. In: ESORICS'14. pp. 76–93 (2014)
7. Ambrosin, M., Conti, M., Ibrahim, A., Neven, G., Sadeghi, A.R., Schunter, M.: SANA: Secure and Scalable Aggregate Network Attestation. In: CCS'16. pp. 731–742 (2016)
8. Asokan, N., Brasser, F., Ibrahim, A., Sadeghi, A.R., Schunter, M., Tsudik, G., Wachsmann, C.: Seda: Scalable embedded device attestation. In: CCS '15. pp. 964–975 (2015)
9. Baccelli, E., Hahm, O., Gunes, M., Wahlisch, M., Schmidt, T.C.: Riot os: Towards an os for the internet of things. In: INFOCOM WKSHPS '13. pp. 79–80 (2013)
10. Boldyreva, A.: Threshold Signatures, Multisignatures and Blind Signatures based on the Gap-Diffie-Hellman-group Signature Scheme. In: PKC '03. pp. 31–46 (2003)
11. Boneh, D., Gentry, C., Lynn, B., Shacham, H.: Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In: EUROCRYPT '03. pp. 416–432 (2003)
12. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: MCC '12. pp. 13–16 (2012)
13. Bormann, C., Ersue, M., Keranen, A.: Terminology for constrained-node networks. Tech. rep. (May 2014), iETF RFC-7228

14. Bormann, C., Shelby, Z.: Block-Wise Transfers in the Constrained Application Protocol (CoAP). Tech. rep. (Aug 2016), iETF RFC-7959
15. Burke, J., Gasti, P., Nathan, N., Tsudik, G.: Securing instrumented environments over Content-Centric Networking: the case of lighting control and NDN. In: INFOCOM WKSHPS '13. pp. 394–398 (2013)
16. Chan, H., Perrig, A., Song, D.: Secure hierarchical in-network aggregation in sensor networks. In: CCS '06. pp. 278–287 (2006)
17. Cooper, A.: Electric Company Smart Meter Deployments: Foundation for A Smart Grid. Tech. rep. (Oct 2016)
18. Dijk, E., Rahman, A., Fossati, T., Loreto, S., Castellani, A.: Internet-Draft: Guidelines for HTTP-CoAP Mapping Implementations. Tech. rep. (Nov 2016), iETF-draft
19. Frikken, K.B., Dougherty, IV, J.A.: An Efficient Integrity-preserving Scheme for Hierarchical Sensor Aggregation. In: WiSec '08. pp. 68–76 (2008)
20. Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., Riviere, E.: Edge-centric computing: Vision and challenges. ACM SIGCOMM Computer Communication Review **45**(5), 37–42 (2015)
21. Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N.: Operating Systems for Low-End Devices in the Internet of Things: A Survey. IEEE Internet of Things Journal **3**(5), 720–734 (Oct 2016)
22. Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B., Koldehofe, B.: Mobile fog: A Programming Model for Large-scale Applications on the Internet of Things. In: MCC '13. pp. 15–20 (2013)
23. Jacobson, V., Smetters, D.K., Thornton, J.D., Plass, M.F., Briggs, N.H., Braynard, R.L.: Networking named content. In: CoNEXT '09. pp. 1–12 (2009)
24. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: TrustLite: A security architecture for tiny embedded devices. In: European Conference on Computer Systems (2014)
25. Open Mobile Alliance: Lightweight Machine to Machine Technical Specification, v 1.0. Tech. rep. (Apr 2016)
26. Perrig, A., Szewczyk, R., Tygar, J.D., Wen, V., Culler, D.E.: Spins: Security protocols for sensor networks. Wireless networks **8**(5), 521–534 (2002)
27. Sanchez, L., Muñoz, L., Galache, J.A., Sotres, P., Santana, J.R., Gutierrez, V., Ramdhany, R., Gluhak, A., Krco, S., Theodoridis, E., et al.: SmartSantander: IoT experimentation over a smart city testbed. Computer Networks **61**, 217–238 (2014)
28. Sehgal, A., Perelman, V., Kuryla, S., Schonwalder, J.: Management of resource constrained devices in the internet of things. IEEE Communications Magazine **50**(12), 144–149 (2012)
29. Sheng, Z., Mahapatra, C., Zhu, C., Leung, V.C.: Recent advances in industrial wireless sensor networks toward efficient management in IoT. IEEE Access **3**, 622–637 (2015)
30. Spanogiannopoulos, G., Vlajic, N., Stevanovic, D.: A simulation-based performance analysis of various multipath routing techniques in zigbee sensor networks. In: ADHOCNETS '09. pp. 300–315 (2009)
31. Unterluggauer, T., Wenger, E.: Efficient pairings and ecc for embedded systems. In: CHES '09, pp. 298–315 (2014)
32. Valmari, A.: The state explosion problem. In: Lectures on Petri nets I: Basic models, pp. 429–528. Springer (1998)
33. Vögler, M., Schleicher, J.M., Inzinger, C., Dustdar, S.: A scalable framework for provisioning large-scale iot deployments. ACM Transactions on Internet Technology **16**(2), 11 (2016)
34. Yang, Y., Wang, X., Zhu, S., Cao, G.: Sdap: A secure hop-by-hop data aggregation protocol for sensor networks. ACM Transactions on Information and System Security **11**(4), 18:1–18:43 (Jul 2008)