Efficient Enforcement of Fine-grained Access Control in Sparkplug-based Industrial Internet of Things

Pietro Colombo[1] and Elena Ferrari[2]

***Abstract*— Sparkplug [1] is an emergent open-source software specification for Industrial Internet of Things (IIoT) systems, designed to favor data integration and device interoperability in an MQTT infrastructure. Although the security issues of IIoT systems can have relevant safety implications, Sparkplug only provides basic security features and essential, coarse-grained access control (AC) mechanisms. Effective AC solutions for Sparkplug-based IIoT systems still need to be designed, and, due to the Sparkplug's increasing popularity and its recent definition as an ISO Standard [1], this has become a crucial need. To fill this void, this paper proposes an approach to efficiently enforcing fine-grained AC in Sparkplug-based IIoT systems. In particular, we define a fine-grained discretionary AC model and a related reference monitor implementing an efficient enforcement mechanism. Early performance evaluations show a reasonably low time overhead.**

I. Introduction

The industry has become a key application domain for the Internet of Things [3]. By interfacing industrial control systems (ICSs) to the Internet, companies can improve the effectiveness and efficiency of industrial operations, producing better products at a lower cost [3]. The Industrial Internet of Things (IIoT) is an emerging paradigm that connects ICSs to enterprise systems, business processes, and analytics [4]. IIoT systems integrate Operational Technology (OT), used for the backend production, with Information Technology (IT), for the management of core functions.

The safety-critical nature of manifold industrial operations makes the security protection of IIoT systems a key requirement. Security problems can cause equipment damage, regulatory issues, and even personal safety hazards. To create broad industry consensus on securing IIoT systems, the Industry IoT Consortium (IIC) has proposed the Industrial Internet Security Framework (IISF) [4]. However, the general guidelines proposed in [4], probably finalized to maximize the framework applicability, make IISF rather generic [5]. Specialized directives for IIoT system classes are therefore needed to make it applicable in real-life deployments.

In this work, we target a relevant class of IIoT systems, that is, those systems designed to operate with Sparkplug [1]. Sparkplug is an emerging open software specification recently defined as an ISO standard [1], which has been designed to favor data integration and device interoperability in an MQTT [6] infrastructure. The relevance of Sparkplug within IIoT has been corroborated by several studies (e.g. [7], [8]). A Sparkplug system comprises MQTT servers and specialized MQTT clients that communicate by means of the publish/subscribe paradigm, exchanging MQTT messages with a standardized topic namespace and payload structure. Sparkplug messages are defined as collections of metrics, namely properties that refer to diagnostics and state values.

Access control (AC) is a core security feature whose efficient enforcement can significantly contribute to secure IIoT systems. IIoT systems often integrate AC solutions designed for the more general IoT domain [13]. The leading AC model families for IoT comprise CapBAC, UCON, RBAC, and ABAC (e.g., see [14]). Frameworks referring to CapBAC, like [15], materialize access authorizations into capability tokens assigned to subjects, who must prove their authorizations by presenting the received token. In contrast, UCON frameworks, such as [12], grant authorizations when access control conditions that refer to mutable subjects, objects, and environment attributes are satisfied. Authorizations are adapted to attribute changes. On the other hand, RBAC frameworks, like [16], grant privileges to users via role assignments. Finally, ABAC frameworks [17] authorize access when conditions on subjects, objects, and environment attributes are satisfied.

Although basic, access control lists (ACLs) are still popular and a subject of research. For instance, Dong et al. [11] proposed an approach to reduce the enforcement overhead of ACLs. The efficiency is attained by mapping user permissions into low-dimensional vectors in the Euclidean space, and using vector operations instead of rule traversing.

Crosscutting to the employed AC model, several recent works propose AC frameworks that leverage blockchain technology (e.g., [13], [19]). For instance, [13] proposes a solution where the role-based, rule-based, and organization-based AC strategies are combined to develop a hybrid approach for smart contracts. In contrast, a Hyperledger Fabric-based implementation of an AC mechanism is proposed in [19], where the blockchain is used to deploy access policies and ensure the integrity of zero-knowledge proofs from tampering and cyberattacks. However, blockchain-based approaches typically suffer from a significant enforcement overhead.

Sparkplug supports essential, coarse-grained AC mechanisms based on ACLs. Policies specify a client's authorized read/write topics, and access is granted for any message with a compliant topic. This simple paradigm is unable to support relevant security requirements, such as the possibility

*This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU-NGEU. Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them.

[1,2] P. Colombo and E. Ferrari are with Department of Theoretical and Applied Science, University of Insubria, Via O. Rossi, 9 - 21100 Varese, Italy {pietro.colombo,elena.ferrari}@uninsubria.it

of granting access only to selected metrics. Therefore, more effective AC solutions are required, but we are unaware of other research initiatives on AC for Sparkplug.

Within a Sparkplug systems the communication is enabled by MQTT, and although the literature proposes a variety of solutions to secure MQTT ecosystems (e.g., see [18] for a recent survey), only a few works have focused on AC (e.g., [9], [10], [12]). In addition, these AC solutions appear unsuited to Sparkplug. Even the finest-grained enforcement mechanism proposed in [10], operates at the message level, and therefore is too coarse-grained for Sparkplug systems. Indeed, by enforcing AC at the message level, a Sparkplug message addressed to a subscriber would be either blocked or received with all the metrics it comprises. In contrast, a subscriber should receive Sparkplug messages with just the metrics it can access, and any message should only comprise the metrics authorized for its publisher. In addition, the access should be granted based on conditions over the metrics' value and metadata.

Example 1 Let us consider a manufacturing company that is planning to integrate third-party analytics into its Sparkplug-based IIoT system, to analyze the production process and optimize the machine workload. This service should be authorized to read only a subset of the metrics generated within the IIoT system. In addition, only the metrics that have not been marked as sensitive should be accessed. Moreover, this service should not issue commands to the machines, except for commands requesting message retransmissions.

To address the access control requirements of the example above, in this paper, in line with the reference architecture we envisaged in [2], we propose a fine-grained discretionary AC model that operates up to the metrics level, granting subject access based on conditions referring to metric properties. We complement the AC model with the design of a reference monitor that implements an efficient enforcement mechanism seamlessly integrated into the Sparkplug architecture. Our monitor intercepts any Sparkplug message sent by/to MQTT clients and, based on the applicable AC policies, either blocks the message or consents the transit of an authorized view from which the unauthorized metrics are removed.

We have implemented a reference monitor prototype to experimentally assess the efficiency and scalability of the enforcement mechanism. The early experiments we have run show a reasonably low time overhead on varying the size of the AC policy sets and the number of clients.

The remainder of the paper is organized as follows. Section II provides a short introduction to basic background concepts on Sparkplug. Section III introduces the proposed AC model for Sparkplug systems, whereas Section IV the enforcement mechanism and the design of the reference monitor. Section V presents our experimental performance evaluation, and finally, Section VI concludes the paper.

II. Sparkplug

A Sparkplug system comprises specialized MQTT clients operating as: i) *edge nodes*, ii) *primary applications*, and iii) *secondary applications*, which communicate by means of an MQTT server (see Figure 1).

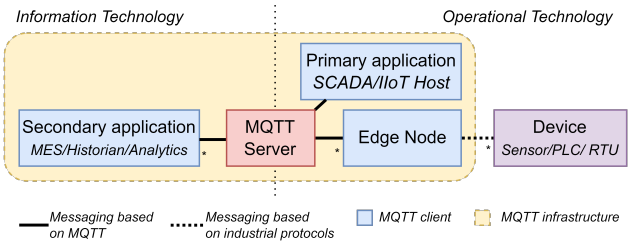

Fig. 1. Sparkplug system architecture

Edge nodes are gateways that control devices like sensors and PLCs, and allow them to communicate with the remainder of the MQTT infrastructure. Edge nodes and related devices are organized in groups. The grouping is commonly based on shared properties (e.g. the location). Primary applications are SCADA/IIoT hosts that control edge nodes and devices. Finally, secondary applications are components, like analytics services, which process edge node and device data and possibly send them commands.

Communication among applications and edge nodes is achieved by exchanging Sparkplug messages, namely MQTT application messages with specific *payload* and *topic* structures. The payload of a Sparkplug message aggregates properties, referred to as *metrics*, which typically denote diagnostics, current state values, and metadata. The topic of a message specifies the components to which that message should be addressed and the message type. Table I lists the supported message types.

TABLE I
SPARKPLUG MESSAGE TYPES

NBIRTH/DBIRTH	An edge node/device is online. Provides information related to the handled metrics.
NDEATH/DDEATH	An edge node/device went offline.
NDATA/DDATA	One/multiple metrics handled by an edge node/device changed value.
STATE	The primary application changed state from online to offline or vice versa.
NCMD/DCMD	Request by an application to modify the value of metrics controlled by an edge node/device.

Sparkplug supports the *Report-by-Exception (RbE)* communication strategy, based on which states and data are only published when change.

III. The Access Control Model

The proposed AC model allows security administrators to specify policies regulating the exchange of Sparkplug messages by authorized subjects. Subjects are MQTT clients operating as edge nodes, or primary and secondary applications (cfr. Section II), that aim to publish or receive Sparkplug messages. A subject s operating as an edge node, a primary or secondary application, is identifiable through its MQTT client identifier, hereafter denoted *cid*. However, a subject operating as an edge node e can also be referred

to by concatenating the name of the group *gid* to which *e* belongs (cfr. Section II), and the unique identifier *eid* of *e* within this group.

Protection objects *o* of the proposed AC model are the messages that subjects seek to exchange. They are modeled as pairs $\langle$ *tp, pl* $\rangle$, where *tp* specifies the topic of the Sparkplug message, and *pl* is the payload of the message. The payload format depends on the message type (cfr. Table I). NBIRTH, DBIRTH, NDATA, DDATA, NDEATH, DDEATH, NCMD, and DCMD messages are defined by aggregation of metrics. The payload of these messages is characterized by a field *timestamp*, specifying the time at which the message is issued, and a field *metrics*, which keeps track of a list of metrics, each defined as a record characterized by the fields *name, value*, *dataType*, and *timestamp*, which denote the metric name, value, the data type of that value, and the time at which the metric has been set. Additional fields could also be used to specify optional metadata. For example, a *sensitivity* property could be used to refer to the sensitivity level of the metric. In contrast, the payload of STATE messages is modeled as a record with the fields *online* and *timestamp* and does not include any metric.

AC can be enforced at the message granularity level for any Sparkplug message type, and at the metric level for messages of type NBIRTH, DBIRTH, NDATA, DDATA, NDEATH, DDEATH, NCMD, and DCMD. Therefore, AC policies can authorize access to entire messages, or to *message views*, defined by referring to the same topic as the original messages, and payloads with metric lists exclusively composed of authorized metrics. Access privileges are granted based on the satisfaction of conditions, that is, boolean expressions built by the composition of object attributes, mathematical, logical, and set operators, and predefined functions that allow the processing of attribute values. To ease and shorten the notation, we refer to the metrics and related metadata properties directly by name, rather than through the path to these object attributes within the message payload.

Definition 1 (Access control policy): An access control policy *p* is a tuple $\langle$*sid, tf, exc, pr, cd*$\rangle$, where: i) *sid* refers to the identifier of the subject constrained by *p*; ii) *tf* is a topic filter that denotes the topic of the messages covered by *p*; iii) *exc* is the list, possibly empty, of the metrics to be excluded from the message payload, hereafter denoted as exception list; iv) *pr* specifies the *read* /*write* privileges granted by *p*; whereas, v) *cd* specifies the condition on object attributes for granting the access.

Example 2 Let us reconsider the scenario introduced in Example 1. Suppose the Sparkplug system comprises the analytics service a_1, operating as a secondary application, and the edge node e_1 of the group g_1, which handles the metrics mt_a, mt_b, mt_c, and mt_d. Let us focus on the definition of a policy *p* that grants a_1 read access to NBIRTH messages generated by e_1. Suppose *p* grants access to the whole payload of NBIRTH messages except for the metric mt_c if the current value of this metric is greater than 5 or if it has been marked as *sensitive*. Policy *p* can be specified as $\langle a_1,$ *spBv1.0/G1/NBIRTH/E1*, $[mt_c]$, *r*, mt_c.value$> 5 \vee mt_c$.sensitive$= true\rangle$

IV. Enforcement

The access control model described in Section III, allow the specification of fine-grained access control policies to protect message exchange in a Sparkplug system. To enforce these policies, we define an enforcement mechanism based on authorized message views, where any Sparkplug message *m* that a subject *s* seeks to access is substituted by the *authorized view* of *m* for *s*, namely a copy of *m* exclusively composed of metrics authorized for *s*, according to the specified access control policies.

In what follows, we start introducing the logic to compose authorized message views (Section IV-A), which is instrumental in defining the enforcement mechanism presented in Section IV-B.

A. Authorized message views

We present the definition of authorized message views by referring to the access requests issued by a subject *s* in a scenario where a set *Ps* of AC policies has been specified. Both *write* and *read* access requests are supported, for which we respectively assume that: i) *s* has requested the publishing of a message *m* on topic *tp*, or ii) *s* subscribed to the receiving of messages on topic filter *tf*, and *s* is going to receive a just-published message *m* on a topic *tp* that is matched by *tf*.

Definition 2 (Applicable policy set): Let us denote with Aps_m^{s+} the set of AC policies that apply to the access request issued by *s*, whose conditions, evaluated with respect to the object attributes of *m*, are satisfied. Aps_m^{s+} includes any *Ps* policy with: i) a *sid* component that refers to *s* through the MQTT client identifier *cid* ii) a topic filter *tf* that is matched by *tp*; iii) a privilege *pr* that is equal to the requested access type (i.e., *read* / *write*); and iv) a condition *cd* satisfied by the object attributes of *m*.

Example 3 Suppose the analytic service a_1 in Example 2 has subscribed to the receiving of e_1's messages of any type, and e_1 has just published an NBIRTH message *m*, which comprises the metrics mt_a, mt_b, mt_c, and mt_d, all set to 10, and mt_c is marked as sensitive. The access to *m* by a_1 is regulated by a set of applicable policies that comprises policy *p* in Example 2.

Authorized message views depend on the requested access and message types.

1) Write access: Let us first consider the case where subject *s* requests to publish *m*. If Aps_m^{s+} is empty, no message view needs to be defined, since *s* request must be denied. In contrast, if Aps_m^{s+} includes at least one policy, the view of *m* that *s* can access is defined as follows.

Definition 3 (Authorized view for write requests): The authorized view of *m* to be released to *s*, referred to as m^s, is a copy of *m* lacking those metrics in the exception lists of the policies in Aps_m^{s+}. More precisely, $m^s.metrics = \{mt | mt \in m.metrics \wedge mt.name \notin \cup_{p \in Aps_m^{s+}} p.exc\}$.

Example 4 Let us consider a secondary application a that seeks to issue a DCMD message m to update to 10 the value of the metrics mt_1, mt_2 and mt_3 handled by device d_1 managed by edge node e_1. Suppose that the set of AC policies that apply to this request comprises: 1) a policy p_1, with an exception list that only contains metric mt_1, and with a condition requesting that mt_1.value ≥ 5, and 2) a policy $p2$, with an empty exception list, and specifying as condition *true*. Since the desired new value of mt_1 is 10, both p_1 and p_2 conditions are satisfied. Therefore, $Aps_m^{a+} = \{p_1, p_2\}$, and the view of m authorized to a contains all the metrics in m except mt_1. Thus, only mt_2 and mt_3 are updated.

If the policies in Aps_m^{s+} do not specify exceptions, protection is enforced at the message granularity level. In this case, the view is equal to the original message.

2) Read access: Read requests are managed with the same logic as for publishing requests, except for NDATA and DDATA messages, as the exchange of these messages follows the RbE communication strategy (see Section II). The rationale is the same for both of these message types. Therefore, in what follows, we present it by referring to NDATA messages.

Let us assume that a subject s subscribed to a topic filter tf that is matched by the topics of the NBIRTH and NDATA messages published by an edge node e. Let us denote with $m_{i,0}$ the *i-th* NBIRTH message published by e, which shows the initial state (i.e., state 0) of e reporting any metric it handles, and let us denote with $m_{i,j}$ the *j-th* NDATA message published by e related to $m_{i,0}$, which notifies the *j-th* update of some of e's metrics since the publication of $m_{i,0}$.

Once $m_{i,j}$ is published, s should receive an authorized view of this message. This view can comprise metrics in $m_{i,j}$ but even metrics excluded from the views authorized to s of the NDATA messages that preceded $m_{i,j}$.

Example 5 Let us assume that a secondary application a subscribed to receiving NBIRTH and NDATA messages published by the edge node e. Suppose that e first publishes an NBIRTH message $m_{1,0}$ that includes, among others, the metrics mt_1, mt_2, mt_3, and mt_4, all set to 0. Later, it publishes an NDATA message $m_{1,1}$, which notifies the update of mt_2 to 6 and mt_3 to 8, followed by another NDATA message $m_{1,2}$ specifying the update of mt_1 to 7. Let us assume that the applicable AC policies grant a read access to $m_{1,0}$, $m_{1,1}$ and to $m_{1,2}$ except for the metric mt_3 within $m_{1,1}$. Consequently, the view of $m_{1,0}$ authorized to a can be defined as the exact copy of $m_{1,0}$, whereas the view of $m_{1,1}$ as a copy of $m_{1,1}$ from which mt_3 is removed. When $m_{1,2}$ is published, no metric must be pruned out, and a is newly authorized to access the whole state of e, comprising mt_3. Therefore, the authorized view of $m_{1,2}$ is not just a simple copy of $m_{1,2}$, but it is a copy augmented with mt_3.

To illustrate the view generation logic, we first need to introduce some additional notations. We denote with $E_{i,j}$ the set of metrics characterizing the state of e at the publishing time of $m_{i,j}$. Moreover, we denote with $M_{i,j}$ the set of metrics in the payload of $m_{i,j}$.

The view of $m_{i,j}$ authorized to s could also comprise *complementary* metrics not included in $m_{i,j}$. A metric mt included in $E_{i,j}$ but not in $M_{i,j}$, is said *complementary*, if it has been excluded from the authorized view of a previous NDATA message $m_{i,k}$ delivered to s (where, $1 \leq k < j$), and subsequently, has not been included nor excluded from the view of another NDATA message $m_{i,k'}$ (where, $k < k' < j$). The set of these additional metrics is formalized by the concept of complementary metric set.

Definition 4 (Complementary metric set): Let us denote with $m_{i,k}^s$ the view of $m_{i,k}$ authorized to s, and with $M_{i,k}^{s-}$ and $M_{i,k}^s$ the set of metrics in $m_{i,k}$ that respectively have been excluded from / compose $m_{i,k}^s$. The set of complementary metrics to be considered for deriving $m_{i,j}^s$, referred to as $C_{i,j}^s$, is defined as: $C_{i,j}^s = \{mt : (mt \in E_{i,j} \setminus M_{i,j}) \wedge (\exists k | 1 \leq k < j \wedge mt \in M_{i,k}^{s-}) \wedge (\nexists k' | k < k' < j \wedge (mt \in M_{i,k'}^s \vee mt \in M_{i,k'}^{s-}))$.

The inclusion of the $C_{i,j}^s$ metrics in $m_{i,j}^s$ is constrained by the applicable AC policies in $Aps_{m_{i,j}}^{s+}$, which rule the receiving of $m_{i,j}$ by s, as the following defintion states.

Definition 5 (Authorized view for read requests): Let $m_{i,j}^{s*}$ be a copy of $m_{i,j}$ with a metric list that has been augmented by the metrics in $C_{i,j}^s$, (i.e., $M_{i,j}^{s*} = M_{i,j} \cup C_{i,j}^s$). The view of $m_{i,j}$ authorized to s, referred to as $m_{i,j}^s$, is a copy of $m_{i,j}^{s*}$ lacking the metrics in the exception lists of the policies in $Aps_{m_{i,j}}^{s+}$. More precisely, $m_{i,j}^s.metrics = \{mt | mt \in m_{i,j}^{s*}.metrics \wedge mt.name \notin \cup_{p \in Aps_{m_{i,j}}^{s+}} p.exc\}$.

Example 6 Let us consider again Example 5. When $m_{1,1}$ is published, e is characterized by the metric set $E_{1,1} = \{mt_1 : 0, mt_2 : 6, mt_3 : 8, mt_4 : 0\}$, whereas $M_{1,1} = \{mt_2 : 6, mt_3 : 8\}$. As such, $E_{1,1} \setminus M_{1,1} = \{mt_1 : 0, mt_4 : 0\}$, but neither mt_1, nor mt_4 are selected as complementary metrics, since $j = 1$. Therefore, $C_{1,1} = \emptyset$, and $m_{1,1}^{a*} = m_{1,1}$. As mentioned in Example 5, the applicable policies $Aps_{m_{1,1}}^{a+}$ require the exclusion of mt_3, and indeed, the view $m_{1,1}^a$ is defined as a copy of $m_{1,1}$ that only includes mt_2.

In contrast, when $m_{1,2}$ is published, $E_{1,2} = \{mt_1 : 7, mt_2 : 6, mt_3 : 8, mt_4 : 0\}$ and $M_{1,2} = \{mt_1 : 7\}$. Thus, $E_{1,2} \setminus M_{1,2} = \{mt_2 : 6, mt_3 : 8, mt_4 : 0\}$. The complementary metric set $C_{1,2} = \{mt_3 : 8\}$, as the update of mt_3 to 8 is the only one that has been excluded from the previous view, and it has not been notified yet to a. Therefore, $m_{1,2}^{a*} = \{mt_1 : 7, mt_3 : 8\}$. The policies in $Aps_{m_{1,2}}^{a+}$ do not require metrics exclusion, thus, the view $m_{1,2}^a$ is equal to $m_{1,2}^{a*}$.

B. The reference monitor

The proposed enforcement mechanism leverages the fact that Sparkplug messages are MQTT *publish* packets [6], and edge nodes, as well as primary and secondary applications, are specialized MQTT clients [1]. The mechanism is designed to be implemented within an MQTT server. This choice relies on the availability of extension functionalities by the major MQTT servers, which allow for MQTT control packet interception and manipulation,[1] as well as the identi-

[1]For instance, HiveMQ APIs allow the definition of interceptors specialized per MQTT control packet type and transiting verses (e.g., see https://docs.hivemq.com/hivemq/latest/extensions/interceptors.html)

fication of the MQTT client that published an intercepted message, or to which that message is addressed. Therefore, we assume a system architecture where the MQTT server integrates an enforcement monitor that operates as a server proxy, mediating the communication with primary and secondary applications, edge nodes, and further MQTT clients. The enforcement monitor interacts with a data management system that manages the access control policies and temporary data generated by the enforcement mechanism. Unlike other proxy-based enforcement techniques proposed in the literature (e.g., [10]), the enforcement monitor does not require an independent dedicated host, and this results in a gain in performance, cost saving, and ease of configuration.

In essence, the enforcement mechanism is defined in such a way as to intercept any Sparkplug message issued by edge nodes or primary and secondary applications to the MQTT server, and vice-versa, and, based on the applicable access control policies, either blocking the transit or consenting the transit of the authorized message view.

At this purpose, a monitoring task is executed whenever an MQTT *publish* packet m is intercepted along a communication channel cc at the I/O interface of the MQTT server with an MQTT client. The pseudocode of function *handleMsg* in Listing 1 presents the activities of this monitoring task.

Listing 1: Function *handleMsg*

```
1  function handleMsg is
       input : m, cc;
2      var cid =getClient (cc);
3      var at =accessType (cc, m);
4      var tp =topicOf (m);
5      var m^s =⊥;
6      if isValid (tp) then
7          var ps =prepareStatement ("select * from
             ACP where p.sid=? and p.pr=? and
             isMatchedBy(p.tf,?)");
8          ps.setParameters (1, cid, 2, at, 3, tp);
9          var Aps^s_m =executeQuery (ps);
10         if at =WRITE then
11             m^s =genView (Aps^s_m, m, s);
12         end
13         if at =READ then
14             if typeOf (m) ∈ {NDATA, DDATA} then
15                 Γ^s_i = getCMetricS(m, s)
16                 m^s =genViewRbE (Γ^s_i, Aps^s_m, m, s);
17             else
18                 m^s =genView (Aps^s_m, m, s);
19             end
20         end
21     end
22     if m^s ≠⊥ then
23         delField (m^s,"pruned");
24         substituteAndForward (m, m^s, cc);
25     else
26         preventDelivery (cc, m);
27     end
28 end
```

The communication channel cc where m is intercepted allows one to determine whether m is addressed to an MQTT client or it has been issued to the MQTT server by an MQTT client, as well as the identifier *cid* of the involved MQTT client. In the former case, *cid* refers to a subject s that requests *read* access to *pp*, while in the latter case to a subject requesting *write* access. The topic associated with m, referred to as *tp*, allows for determining whether m is a valid Sparkplug message. If so, *tp* specifies the type of message and the encoding scheme of m's payload.

Listing 2: Function *genView*

```
1  function genView is
       input : Aps^s_m, m, s
       output: the view of m authorized to s
2      var ect_m =getCtx (m)
3      var Aps^{s+}_m = {p | p ∈ Aps^s_m ∧ eval(p.cd, ect_m)}
4      if Aps^{s+}_m = ∅ then
5          return ⊥
6      end
7      var m^s = m
8      if typeOf (m)∈ {STATE} then
9          return m^s
10     end
11     if typeOf (m)∈ {DBIRTH, NBIRTH} then
12         genCMetricS (m, s)
13     end
14     m^s.metrics={t | t ∈ m.metrics ∧ t.name ∉ ∪_{p∈Aps^{s+}_m} p.exc}
15     m^s.pruned={t | t ∈ m.metrics ∧ t.name ∈ ∪_{p∈Aps^{s+}_m} p.exc}
16     return m^s
17 end
```

The AC policies that could grant s access to m are derived from the policy database, based on Def. 2. We select as *candidate* any AC policy specified for s, with a topic filter matched by *tp*, and specifying the requested access type. The candidate policy set is instrumental in defining the message view of m authorized to s.

Different view generation techniques are employed for *write* and *read* requests, which will be presented in Section IV-B.1, and IV-B.2, respectively. The derived view allows making a decision on the access requested by s. If the derived view m^s is not empty, m is substituted by m^s and continues the transit along cc. Otherwise, the transit of m is blocked as the access request is denied.

In what follows, we illustrate the view generation approach for write and read access requests, respectively.

1) Write requests: Let us consider the case where subject s requests to publish m. The generation of the view is handled by function *genView*, presented in Listing 2, which takes as input the message m that s seeks to access, and the set of *candidate* AC policies Aps^s_m that apply to the request of s, and returns the view of m authorized to s. Function *genView* first derives the applicable AC policies Aps^{s+}_m from Aps^s_m. Aps^{s+}_m contains any Aps^s_m policy p, whose condition, evaluated for the attributes of m, is satisfied. Policy evaluation relies on function *eval(exp,ctx)*, which takes as input a parametric boolean expression *exp* specified as a string, and an evaluation context *ctx* defined as a map of properties, and checks if *exp* is satisfied for *ctx*. The evaluation context for a message *msg* is derived by function *getCtx(msg)*, which remaps the payload of *msg* to a map of

attributes. Referred to as ect_m the evaluation context of m, such that ect_m=*getCtx(m)*, Aps_m^{s+} is defined as composed of any $p \in Aps_m^s$ such that *eval(p.cd, ect_m)=true*. If $Aps_m^{s+} = \emptyset$, the empty view is returned, which notifies that the access request by s is denied; otherwise, the function generates the view of m authorized for s, denoted as m^s.

The view m^s is initially defined as a copy of m. If m is of type STATE, m^s is immediately returned, as the view corresponds to m. If m is of type NBIRTH or DBIRTH, the function *genCMetricS* creates an initially empty set of complementary metrics, used to calculate the authorized views of the NDATA and DDATA messages that will follow the birth notification communicated by m (see Section IV-A). Finally, before returning the view m^s, the metric list in the payload of m^s (i.e., the value of field *metrics*) is redefined as composed of any metric in m that is not referred to in the exception lists of policies in Aps_m^{s+}. Similarly, the ad-hoc defined field *pruned* is introduced to keep track of the metrics that have been excluded from the view m^s.

2) Read requests: Let us now consider the case where, based on s subscriptions, a message m is forwarded to s and should be read by s. If m is of type NBIRTH, DBIRTH, NDEATH, DDEATH, NCMD, and DCMD, the enforcement mechanism is the same as the one for write access requests (see Section IV-B.1). In such a case, view generation is managed by function *genView* (see Listing 2). In contrast, due to the RbE communication, a different technique is used with NDATA or DDATA messages. In what follows, we present the enforcement mechanism by referring to NDATA messages, but the same applies to DDATA messages. Let us assume that m is the *j-th* NDATA message published by an edge node e since the publishing of the *i-th* NBIRTH message by the same node, and let us refer to m with the notation $m_{i,j}$. Moreover, we denote with $Aps^s_{m_{i,j}}$ the set of candidate policies that apply to the receiving of $m_{i,j}$ by s. The generation of the view is managed by function *genViewRbE*, presented in Listing 3, which is invoked any time an NDATA message $m_{i,j}$ is published. As mentioned in Section IV-A, the view of $m_{i,j}$ authorized to s can include metrics in the complementary metric set $C^s_{i,j}$ that are not enclosed in $m_{i,j}$. To efficiently calculate the complementary metrics we employ a cumulative approach.

Let Γ^s_i be the cumulative set of complementary metrics.

Listing 3: Function *genViewRbE*

```
1  function genViewRbE is
       input : Γ^s_i, Aps^s_{m_{i,j}}, m_{i,j}, s
       output: the view of m_{i,j} authorized to s
2      var m^{s*}_{i,j} =m_{i,j}
3      m^{s*}_{i,j}.metrics = m_{i,j}.metrics ∪ Γ^s_i
4      var m^s_{i,j} =genView (Aps^s_{m_{i,j}}, m^{s*}_{i,j}, s)
5      if m^s_{i,j} ≠⊥ then
6          Γ^s_i = Γ^s_i ∪ m^s_{i,j}.pruned \ m^s_{i,j}.metrics
7      end
8      return m^s_{i,j}
9  end
```

The set Γ^s_i is created at the forwarding time of the *i-th* NBIRTH message $m_{i,0}$ to s (see line 12 of Listing 2), and it is initially defined as an empty set. Function *genViewRbE* updates Γ^s_i any time an NDATA message $m_{i,j}$ is forwarded to s. The complementary metrics referred to by Γ^s_i are used to derive $m^{s*}_{i,j}$, which, as mentioned in Section IV-A, is a copy of $m_{i,j}$ with a metric list that has been augmented by the complementary metrics. The view $m^s_{i,j}$ is generated by function *genView*, presented in Listing 2, based on $m^{s*}_{i,j}$ and the candidate policy set $Aps^s_{m_{i,j}}$. Afterwards, Γ^s_i is updated by: i) adding the metrics excluded from $m^s_{i,j}$, and ii) removing those that compose $m^s_{i,j}$.

V. Experiments

In this section, we describe the experiments we ran to empirically assess how efficiently our AC framework regulates the exchange of messages in a Sparkplug system.

The experiments rely on a reference monitor prototype implementing the enforcement mechanism presented in Section IV-B, and designed to be hosted in HiveMQ community edition,[2] a popular open-source MQTT server. Our Java-based implementation exploits core services of the HiveMQ Extension SDK, such as the Publish Inbound and Outbound Interceptors,[3] to intercept PUBLISH messages, modify message payloads, and prevent message delivery.

Although traditional DBMSs fully support AC policy management, to minimize the selection time of the AC policies that apply to an access request we have used Redis,[4] a popular key-value datastore allowing the execution of lookup-by-key operations with sub-millisecond latency.

Edge nodes, devices, primary and secondary applications employed for our experiments have all been defined by customizing the open-source Sparkplug-compatible Java applications *Tahu-Host-Compat* and *Tahu-Edge-Compat* provided by the Eclipse Tahu Project.[5] The implemented customizations allow the setup of behavioral and structural properties through configuration parameters. For instance, one can specify the number of devices handled by an edge node, the number of metrics managed by edge nodes and devices, and the frequency with which changes in metric values are notified to applications.

To manage scenarios where numerous devices, edge nodes, and primary and secondary applications need to interact, we have containerized the applications with Docker.[6]

For our experiments, we used two virtual machines (VMs) hosted in a server with Intel(R) Xeon(R) Gold 6238R CPU at 2.20GHz. The first VM employs 10 dedicated cores (20 threads) and 24 GB of RAM and hosts the execution of the customized version of HiveMQ CE (the MQTT server) that embeds the enforcement monitor, and Redis. In contrast, the second VM, with 30 dedicated cores (60 threads) and 24

[2]https://github.com/hivemq/hivemq-community-edition
[3]https://docs.hivemq.com/hivemq/latest/extensions/interceptors.html
[4]https://redis.io/
[5]https://github.com/eclipse/tahu
[6]https://www.docker.com/

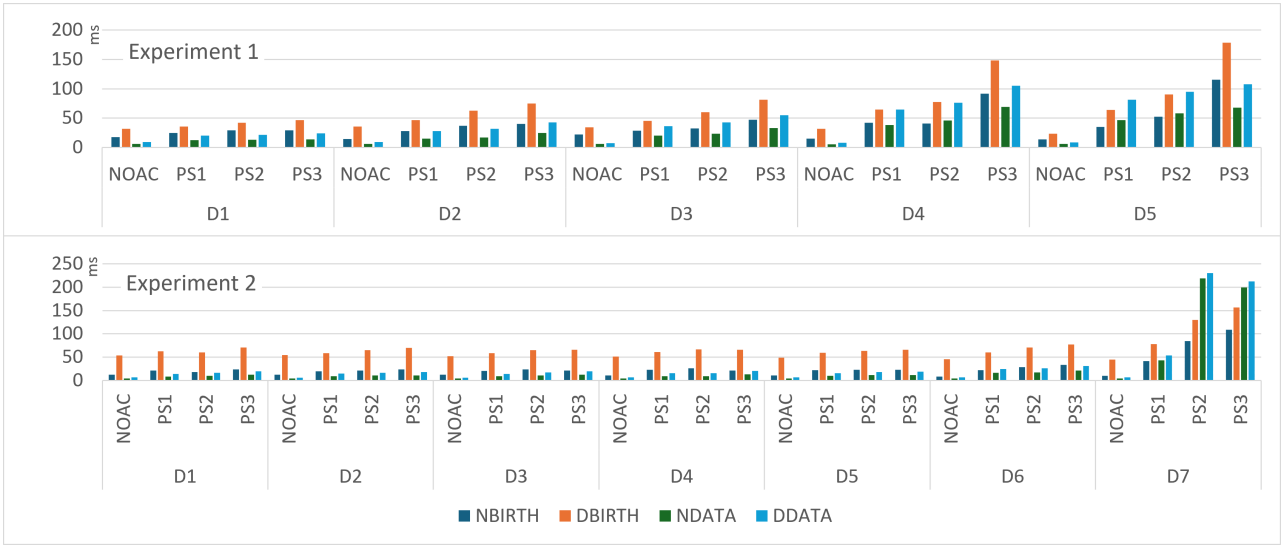

Fig. 2. The average transmission time in the two experiments

TABLE II
THE CONFIGURATIONS CONSIDERED FOR THE EXPERIMENTS

		Experiment 1					Experiment 2						
		D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D6	D7
Deployment properties	#Primary apps	1	1	1	1	1	1	1	1	1	1	1	1
	#Secondary app	1	1	1	1	1	1	1	1	1	1	1	1
	#Edge nodes	50	50	50	50	50	60	70	80	90	100	150	200
	#Dev per edge node	9	13	17	21	25	5	5	5	5	5	5	5
	#Metrics per edge node	20	25	30	35	40	10	10	10	10	10	10	10
	#Metrics per dev	8	8	8	8	8	5	5	5	5	5	5	5
Policy set	#Policies in PS1	4505	6305	8105	9905	11705	3245	3785	4325	4865	5405	8104	10804
	#Policies in PS2	5160	7392	9492	11760	13412	3816	4418	5001	5653	6259	9348	12468
	#Policies in PS3	5903	8279	10842	13297	15737	4344	5067	5691	6606	7058	10677	14269

GB of RAM, hosts the execution of the containerized edge nodes, primary and secondary applications.

Our experiments aim to stress the performance of the proposed AC framework, assessing the time overhead introduced by the enforcement of AC policies. We focus on the time elapsed from the edge node publishing of NBIRTH, DBIRTH, NDATA, and DDATA messages, referred to by a timestamp in the message payload, and the receiving of these messages. We refer to this as *transmission time*. We have selected these message types as they cover most of the message exchange in an industrial setting.

Our experiments refer to a typical industrial setting where all the devices, edge nodes, and primary applications belong to a local deployment, whereas the secondary applications (e.g., MES, Historians, and Analytics) are external and managed by third parties, thus representing a potential threat.

We assess the transmission time by varying the characteristics of the deployments and with policy sets of increasing size and complexity. The considered configurations have been proposed to study the enforcement overhead under different test conditions, and do not map realistic scenarios.

We conducted two experiments. In our first experiment, we scale the number of devices managed by edge nodes, as well as the edge node metrics. We consider five deployments (D1-D5), all including a primary and a secondary application and 50 edge nodes, but differing in the number of devices managed by each edge node and the metrics handled by edge nodes. The characteristics of the deployments are shown in the top part of Table II.

For each deployment, we synthetically generate three AC policy sets, referred to as PS1, PS2, and PS3, of increasing size, granting access privileges to the primary and secondary applications and the edge nodes in the 5 deployments. PS1, PS2, and PS3 include AC policies specifying exception lists with at most 1, 2, and 3 exceptions, respectively (see Def. 1 in Section III). The size of the policy sets specified for the target deployments is shown in the bottom part of Table II.

The bar diagram in Figure 2 shows the measured average transmission time (ms) per deployment and message type when varying the policy set. The diagram shows that in each deployment and for each message type the transmission time grows with the policy set size and the number of exceptions specified per AC policy. Therefore, for any deployment, the height of the bars referring to the same message type (i.e., bars of the same color) grows with the index of the policy set. In addition, the height of the bars related to the same message type and policy set grows with the deployment index. Thus, the average transmission time per policy set grows with the

deployment size across the deployments.

To assess the enforcement overhead, the bar diagram in Figure 2 also shows, for any deployment, the transmission time measured when no AC enforcement mechanism is enabled. The enforcement overhead in a deployment *d* for a policy set *ps* and messages of type *mt* corresponds to the height difference of the bar within *d* that refers to *ps* and *mt*, and the same color bar in *d*, labeled *NOAC*, which denotes the transmission time when the enforcement monitor is disabled.

Time overhead ranges from tens of milliseconds in D1, D2, and D3, up to 140 ms in D5 with the policy set PS3 and DBIRTH messages, thus appearing reasonably contained.

In our second experiment, we scale from 60 to 200 the number of edge nodes, significantly increasing the volume of exchanged messages. However, we configure devices and edge nodes to generate fewer metrics, thus reducing the size of message payloads. We consider 7 deployments, denoted D1-D7, which share the presence of a primary and a secondary application, the number of devices managed by the edge node, and metrics handled by any edge node and device (see Table II).

The rationale for generating the AC policy sets PS1, PS2, and PS3 used in this second experiment is the same as the first experiment. Table II shows the number of AC policies composing these sets for any deployment.

The results of the second experiment are shown in Figure 2. As in the first experiment, in any deployment, the height of the bars of the same color grows with the index of the policy set. Therefore, for each deployment and message type, the transmission time appears proportional to the size of the policy set. However, this time, the growth is slower than in Experiment 1. We believe the observed lower growth gradient is attributable to the message payload size, which is smaller than in the previous experiment.

The height of the bars related to the same message type and policy set appears approximately the same across the deployments D1-D6, and higher in D7. This trend suggests that the enforcement overhead mainly depends on the number of specified AC policies and exceptions until the number of edge nodes hosted on the same server starts causing a scheduling delay.

Time overhead ranges from a few milliseconds to tens of milliseconds in D1-D6, up to 230 ms in deployment D7. However, even in this critical case, the overhead is contained.

Overall, the results obtained in these early experiments show a good efficiency of the implemented AC framework with a reasonably low time overhead.

VI. Conclusions

In this paper, we have proposed a discretionary AC framework to regulate data sharing in a Sparkplug system at a very fine-grained level. Access is granted based on conditions specified over metric properties. AC is enforced by intercepting any Sparkplug message sent by/to a subject and substituting it with a message view authorized for the subject. Our early experimental evaluations revealed a reasonably low time enforcement overhead. We plan to extend the analysis by studying the use of computational resources like CPU time and memory usage. In future work, we plan to use our framework in a real industrial setting and study the performance of the approach in a real scenario. Moreover, to provide a more flexible data protection form tailored to the real-time requirements of IIoT systems, we aim to provide support for temporal AC policies. Finally, we are considering the development of tools for AC policy analysis and management.

References

[1] ISO, "Sparkplug® version 3. ISO/IEC 20237:2023 - Information technology." 2023.

[2] P. Colombo and E. Ferrari, "Access Control Integration in Sparkplug-Based Industrial Internet of Things Systems: Requirements and Open Challenges," in Proc. of *WEBIST - 2024*, Porto (PT), Scitepress 2024.

[3] D. Serpanos, "Industrial Internet of Things: Trends and Challenges," *Computer*, vol. 57, no. 1, pp. 124–128, 1 2024.

[4] K. Caindec, M. Buchheit, B. Zarkout, S. Schrecker, F. Hirsch, I. Dungana, R. Martin, and M. Tseng, "Industry Internet of Things Security Framework (IISF)," ver. 2, 2023

[5] V. R. Kebande and A. I. Awad, "Industrial Internet of Things Ecosystems Security and Digital Forensics: Achievements, Open Challenges, and Future Directions," *ACM Computing Surveys*, vol. 56, no. 5, 2024.

[6] OASIS, "MQTT Version 5.0 OASIS Standard," 2019.

[7] P. Koprov, X. Fang, and B. Starly, "Machine identity authentication via unobservable fingerprinting signature: A functional data analysis approach for MQTT 5.0 protocol," *Journal of Manufacturing Systems*, vol. 76, pp. 59–74, 10 2024.

[8] P. Koprov, A. Ramachandran, Y. S. Lee, P. Cohen, and B. Starly, "Streaming Machine Generated Data via the MQTT Sparkplug B Protocol for Smart Factory Operations," *Manufacturing Letters*, vol. 33, 2022.

[9] P. Colombo and E. Ferrari, "Access Control Enforcement within MQTT-based Internet of Things Ecosystems," in *Proceedings of ACM SACMAT*, 2018.

[10] P. Colombo, E. Ferrari, and E. D. Tümer, "Regulating data sharing across MQTT environments," *Journal of Network and Computer Applications*, vol. 174, 2021.

[11] L. Dong, T. Wu, W. Jia, B. Jiang and X. Li, "Computable Access Control: Embedding Access Control Rules Into Euclidean Space," in IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 53, no. 10, 2023.

[12] A. La Marra, F. Martinelli, P. Mori, A. Rizos, and A. Saracino, "Introducing usage control in MQTT," in LNCS vol. 10683, 2018.

[13] R. Saha, G. Kumar, M. Conti, T. Devgun, T. H. Kim, M. Alazab, and R. Thomas, "DHACS: Smart Contract-Based Decentralized Hybrid Access Control for Industrial Internet-of-Things," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 5, pp. 3452–3461, 5 2022.

[14] S. Dramé-Maigné, M. Laurent, L. Castillo, and H. Ganem, "Centralized, Distributed, and Everything in between: Reviewing Access Control Solutions for the IoT," *ACM Computing Surveys*, vol. 54, no. 7, 2022.

[15] R. Xu, Y. Chen, E. Blasch, and G. Chen, "BlendCAC: A smart contract enabled decentralized capability-based access control mechanism for the IoT," *Computers*, vol. 7, no. 3, 2018.

[16] A. Gouglidis and I. Mavridis, "DomRBAC: An access control model for modern collaborative systems," *Computers and Security*, vol. 31, no. 4, 2012.

[17] D. Han, Y. Zhu, D. Li, W. Liang, A. Souri, and K. C. Li, "A Blockchain-Based Auditable Access Control System for Private Data in Service-Centric IoT Environments," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 5, 2022.

[18] S. U. A. Laghari, W. Li, S. Manickam, P. Nanda, A. K. Al-Ani and S. Karuppayah, "Securing MQTT Ecosystem: Exploring Vulnerabilities, Mitigations, and Future Trajectories," in IEEE Access, vol. 12, 2024

[19] D. Li, N. Crespi, R. Minerva, W. Liang, K. C. Li, and J. Kołodziej, "DPS-IIoT: Non-interactive zero-knowledge proof-inspired access control towards information-centric Industrial Internet of Things," *Computer Communications*, vol. 233, 3, 2025.