SPPS: Secure Policy-based Publish/Subscribe System for V2C Communication

Mohammad Hamad*, Emanuel Regnath*, Jan Lauinger*, Vassilis Prevelakis† and Sebastian Steinhorst*
* Technical University of Munich, Germany, Email: firstname.lastname@tum.de
†Technical University of Braunschweig, Germany, Email: prevelakis@ida.ing.tu-bs.de

Abstract*—The Publish/Subscribe (Pub/Sub) pattern is an attractive paradigm for supporting Vehicle to Cloud (V2C) communication. However, the security threats on confidentiality, integrity, and access control of the published data challenge the adoption of the Pub/Sub model. To address that, our paper proposes a secure policy-based Pub/Sub model for V2C communication, which allows to encrypt and control the access to messages published by vehicles. A vehicle encrypts messages with a symmetric key while saving the key in distributed shares on semi-honest services, called KeyStores, using the concept of secret sharing. The security policy, generated by the same vehicle, authorizes certain cloud services to obtain the shares from the KeyStores. Here, granting access rights takes place without violating the decoupling requirement of the Pub/Sub model. Experimental results show that, besides the end-to-end security protection, our proposed system introduces significantly less overhead (almost 70% less) than the state-of-the-art approach SSL when reestablishing connections, which is a common scenario in the V2C context due to unreliable network connection.

***Index Terms*—Secure Pub/Sub Model, V2C Communication**

I. INTRODUCTION

Today, vehicles are equipped with many smart sensors that collect a vast amount of data. V2C communication will allow vehicles to benefit from cloud computation power to handle this processing via different services in order to enhance the intelligent transportation system. V2C communication faces many challenges, such as the unstable connectivity with cloud services and the need to communicate securely with many services owned by different authorities. The Pub/Sub model is a communication paradigm that enables a sender, known as a publisher, to disseminate messages to multiple receivers, known as subscribers, at once via a mediator, known as a broker. The Pub/Sub pattern provides full decoupling in time, space, and flow between publishers and subscribers, which are important properties of distributed systems. These characteristics make the Pub/Sub model an excellent candidate to implement V2C communication. However, in terms of security, the Pub/Sub model is susceptible to a wide variety of security threats that affect the confidentiality, integrity, and access control of published data [1].

The privacy-related data published by cars requires strict restrictions on who is permitted to access this information and how long it should be stored. Therefore, a vehicle may need to encrypt this data to ensure that only authorized parties can access this data for a specified time. Achieving such a level of access control on published data requires that each vehicle needs to negotiate with every cloud service and to agree on one crypto key to be used to secure the data. However, this solution violates the decoupling requirement of the Pub/Sub system. Another adopted way to provide security for the Pub/Sub model is by assuming the broker as a trust component. This allows vehicle and cloud service to set up a secure link (e.g., using SSL/TLS) individually with the broker. However, this solution does not provide end-to-end security and suffers from significant overhead due to re-establishing the secure link each time the connection is lost or closed [2]. Other solutions, such as Attribute-Based Encryption [3] were adopted recently by several researchers (e.g., in [4]). ABE is public-key encryption (asymmetric) that ensures a fine-grained access control mechanism to encrypted data based on flexible access policies. However, adopting such a solution comes with a significant overhead with regard to execution time [5].

This work is partially supported by the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763 and by the European Commission through the following H2020 projects: nIoVe under Grant Agreement No. 833742, THREAT-ARREST under Grant Agreement No. 786890, and CONCORDIA under Grant Agreement No. 830927.

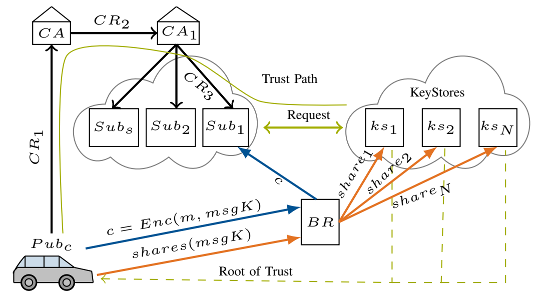

Figure 1: High-level system architecture: The vehicle (publisher) encrypts messages with a message key, which authorized subscribers can obtain from KeyStores.

Contributions

This paper proposes a secure Pub/Sub system to ensure end-to-end secure communication between cars and cloud services without trusting brokers (see Figure 1). The proposed solution uses semi-honest services, so-called KeyStores, to save secret shares provided by a car. These secrets are used by each authorized cloud service to reconstruct a symmetric key generated by the vehicle and used to encrypt the published data. Each vehicle has the capability to define a security policy that determines the conditions which allow certain cloud services to retrieve the shared secret from each KeyStore (see Section III). Only subscribers who have adequate security policies (credentials) can retrieve the saved shares from the KeyStores, reconstruct the key, and decrypt the messages. Our solution does not require any prior interaction or agreement between the vehicle and cloud services. In particular, we

- propose a policy-based secure Pub/Sub model for V2C communication that enables a car to share its data securely and control who can access published data without trusting brokers (Sections III and IV).
- implement our proposed system and empirically evaluate it using embedded devices to compare its performance with other approaches. The performance analysis indicates that our solution introduces very little overhead while outperforming the state-of-the-art approaches by 70% when reestablishing connections (Section V).

II. System and Threat Model

A. System Components

As shown in Figure 1, our system contains (1) vehicles which want to share information. Each vehicle Pub_c has a public $pubK_c$ and private $privK_c$ key and can publish messages on different topics t_1, t_2, $\dots$, t_z. (2) Cloud services that subscribe and receive messages published by Pub_c. Each cloud service Sub_j is interested in certain information (i.e., topic) and belongs to a certain authority (e.g., city, commercial organization, etc.). Each Sub_j is (or can be) authorized to receive messages published on one or more topics based on the credential(s) it has. (3) A Certificate Authority (CA) which issues credentials to intermediate CA(s) or to a subscriber Sub_j to authorize it to receive certain information based on the properties that Sub_j has. (4) A KeyStore component ks which is used to store the secret keys. We refer to the set of all KeyStores as $\mathcal{KS}$ ($\mathcal{KS} = \{ks_1, ks_2, \dots, ks_N\}; |\mathcal{KS}| = N$). Each ks has a public $pubK_{ks}$ and private $privK_{ks}$ key. (5) A broker (BR) which forwards messages published on topic t to the subscribers who are interested in this topic. Also, it can forward these messages to neighboring brokers to ensure the availability of the data. For simplicity, we will consider using one Pub_c and one BR.

We refer to the shared key between Pub_c and ks_i as *master key* $msrK_{Pub_c - ks_i} \in \{0,1\}^\lambda$ and to the key which is used to encrypt the published messages as *message key* $msgK \in \{0,1\}^\lambda$ where λ refers to the key's size. A nonce (nc) is a unique value which has not been used before. We define the encryption function Enc to translate a plain text p into a cipher-text c using a key k as $c = Enc(p, k)$. Accordingly, we define the decryption function Dec to translate c into p using k as $p = Dec(c, k)$. Based on the applied k, we can determine whether the encryption/decryption function is a symmetric or asymmetric one. We further define a hashing function H to produce the hash value $h \in \{0,1\}^l$ (l is a fixed length) of p as $h = H(p)$. $HMAC$ is a function used to produce a keyed hash value mac of p using a symmetric key k as $mac = HMAC(p, k)$. The Sig function is used to sign p using a private key $privK$ while the Ver function uses the associated public key $pubK$ to verify the produced signature of p as $Ver(.) \in \{0,1\}$. We use $||$ for concatenating messages and $\oplus$ for XOR operation. A function $Cmp(x, y, z) \in \{0,1\}$ is defined to compare whether $a \oplus b == z$ (return 0 if is true). We use Alice&Bob–notation to describe our security protocol. E.g, $A \to B : m$ is read as A sends a message m to B and $A : X$ is read as A performs X.

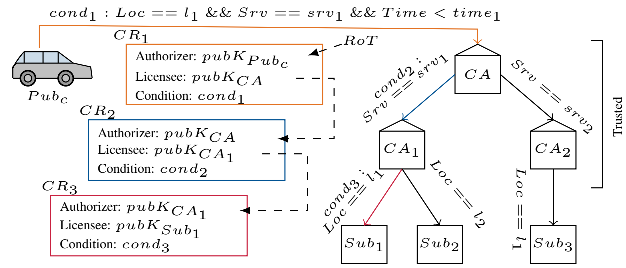

Figure 2: Security credentials.

B. Threat Model

This work assumes that the CA is fully *trusted* and will not issue credentials to untrustworthy subscribers or intermediate CAs. We also consider that a vehicle is *honest* and will not publish malicious messages to disrupt the broker or subscriber functionalities. Both KeyStores and broker are considered *semi-honest*. They will perform the protocol correctly, but they will attempt to get the content of published messages and crypto keys. Subscribers are assumed to be *malicious* in the sense that they are interested in all published information. Each Subscriber can collude with other system components (i.e., broker and KeyStores). We also consider that an external attacker can *only access, delay, and store* all transmitted messages in the system without dropping them (handling Denial of Service (DoS) attacks is out of the scope of this paper). Finally, we assume that all crypto keys are stored in a secure way such that an external attack cannot extract them in a reasonable time.

III. Policy-based Trust Management

This section details how each vehicle can create a security credential to define which cloud services are authorized to retrieve the symmetric key from KeyStores and decrypt the data. We consider an example where a vehicle shares traffic flow information that a cloud service can use to support dynamic routing. These published messages could include information that could be used to trace the vehicle if malicious services accessed it. Therefore, a car is interested in keeping its data secret and ensures that only services that analyze the data for dynamic routing ($Srv == srv_1$) and are responsible for the area where the car is traveling ($Loc == l_1$) can access this data for a specific period ($Time < time_1$). Any other services should not be able to retrieve the published messages.

We adopt the KeyNote policy definition language [6] to create security credentials that express trust relations between different components. As shown in Figure 2, each credential contains information about the entity granting the authorization (Authorizer), information about the recipient of the authorization (Licensee), and the condition under which the Authorizer trusts the Licensee to perform an action. We refer to each credential as $CR_{Licensee}^{Authorizer}$. Both the Authorizer and Licensee fields contain public keys.

One of the main characteristics of the KeyNote policy definition language is trust delegation. Each Licensee can play the Authorizer's role and delegate the trust that he/she gained by a credential to other actors (Licensee) with new conditions (without violating the initial conditions) as shown in Figure 2. Delegation allows the creation of a trust relationship between

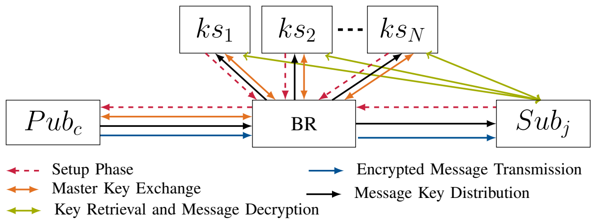

Figure 3: Communication of the five protocol phases.

one Authorizer, such as Pub_c, and a Licensee, such as Sub_1, indirectly. This property aligns with the decoupling nature of Pub/Sub paradigms since Pub_c does not need Sub_2's identity. We restrict the delegation capability to trusted parties only (i.e., CAs, see Figure 2). In our proposed system, the CA creates credential CR_2 which authorizes CA_1 to perform srv_1. Benefiting from the delegation capability, CA_1 itself is able authorize different Subscribers to provide srv_1 but in a different geographical location (i.e., l_1 and $l2$). Any new subscriber needs to communicate with the appropriate CA to receive a security credential based on its capabilities. The method how Sub_j can prove its capabilities to get such a credential is beyond the paper's scope. Whenever Pub_c issues CR_1, it indirectly authorizes Sub_1 to retrieve the key as long as it requests it during the valid period (i.e., $request\_time < time_1$).

It is important to note that Sub_j will not request the secret shares from Pub_c, but from every ks_i. We need to ensure that each ks_i will not deliver the secret shares to any subscribers unless Pub_c authorizes it. Therefore, each ks_i considers Pub_c as the Root of Trust (RoT) for every secret share generated by that Pub_c. Sub_j needs to sign its request and provide all the credentials that it has. To authorize that request, ks_i needs to validate the signature of the request and find the so-called "Trust Path" (see Figure 1), which links the requester's key (i.e., Sub_j's key) with the key of the RoT. If such a path is found and all the conditions in all credentials that form that path are satisfied, ks_i authorizes Sub_j's request and shares with it the saved share. Otherwise, the request will be denied.

IV. Proposed Protocol

Figure 3 illustrates the five phases of our proposed protocol. These phases are the: setup phase (Section IV-A), master key exchange (Section IV-B), message key distribution (Section IV-C), encrypted message transmission (Section IV-D), and key retrieval and message decryption (Section IV-D). In the remainder of this section, we will explain each phase in more detail.

A. Setup Phase

This phase occurs only once during the setup of the system. Throughout this phase, all KeyStores need to register with the broker and *subscribe* to a general topic called $msrkt$ enabling them to receive messages during the next two phases:

$$\forall ks_i \in \mathcal{KS},\ ks_i \rightarrow BR:\ subscribe(msrkt)$$

Each Sub_j needs to have a particular credential from CA to prove their capabilities and authorization to receive messages on a specific topic. Also, each Sub_j needs to register its interest to receive messages on the particular topic t (Pub_c will publish data on this topic) by sending a *subscribe* message to the broker. We will refer to the set of cloud services which are interested in a topic t as $\mathcal{SUB}_t$:

$$\mathcal{SUB}_t = \{Sub_j | Sub_j \rightarrow BR:\ subscribe(t)\}$$

Pub_c only needs the public key of the domain's CA (i.e., $pubK_{CA}$) and the public keys of all KeyStores. All this information will be delivered to Pub_c in the form of a certificate whenever it connects to any BR. Pub_c can verify this certificate based on a pre-programmed list with the intermediary Certificate Authorities and the root CA. Note that Pub_c does not need to be authorized by any BR. Each Sub_j can verify Pub_c's authenticity and validate its authorization, if needed, by asking Pub_c to transmit its certificate.

B. Master Key Exchange

This phase contains two sub-phases: Phase I, which is used to send the master key, and Phase II to receive an acknowledgment of receiving the master key.

1) Phase I: Whenever Pub_c receives the information about the existing $\mathcal{KS}$, it creates a set of master keys ($\mathcal{K}_{msr}$) and stores them securely into a list of five tuples $\langle ID_{ks_i}, pubK_{ks_i}, msrK_{Pub_c-ks_i}, nc_i, expT\rangle$. Each tuple includes ks_i's ID, ks_i's public key, the master key $msrK_{Pub_c-ks_i}$, a nonce nc_i, and the expiration time $expT$ of this key. Pub_c must renew $msrK_{Pub_c-ks_i}$ whenever $expT$ expires. Pub_c uses the public key of each ks to encrypt the generated master key with a fresh nonce nc_i. Pub_c also concatenates its public key with the message and sends it to the broker to be used as RoT:

$$Pub_c : \mathcal{K}_{msr} = \{msrK_{Pub_c-ks_1}, \ldots, msrK_{Pub_c-ks_N}\}$$
$$\forall ks_i \in \mathcal{KS}, Pub_c : c_i = Enc(msrK_{Pub_c-ks_i} || nc_i, pubK_{ks_i})$$
$$Pub_c : C = \mathop{||}_{i=1}^{N} ID_{ks_i} || c_i$$
$$Pub_c \rightarrow BR : C\ ||\ pubK_{Pub_c}$$

Upon receiving the message, the broker forwards it to all registered KeyStores. Each ks_i decrypts its part (c_i) (based on the ID_{ks_i}) of the received message and extracts the master key and nonce. Each KeyStore uses a list of tuples $\mathcal{L}$ to store the master keys of each publisher. Each tuple $\langle H(pubK_{Pub_c}), RoT, msrK_{pub_c-ks}, t_{id}, msgK\rangle$ contains a hashed value of the publisher's public key to serve as an identity for that publisher, the root of trust which will be filled by the publisher's public key, the shared master key, a topic identifier, and a message key to secure messages published on this topic. Note that each publisher can have multiple message keys, one for each topic. However, it needs only one master key. In this stage of the protocol, t_{id} and $msgK$ are empty:

$$BR \rightarrow \mathcal{KS} : C\ ||\ pubK_{Pub_c}$$
$$\forall ks_i \in \mathcal{KS} :\ \langle msrK_{Pub_c-ks_i}, nc_i\rangle = Dec(c_i, privK_{ks_i})$$

2) Phase II: After receiving the master key, each ks_i needs to acknowledge the Pub_c about Phase I's success and confirm that the master key was linked to the Pub_c's public key. Each ks_i forms a message by XORing the received nonce nc_i and the hash value of the $pubK_{pub_c}$ to avoid the known-plaintext attack. This message is encrypted using the received $msrK_{Pub_c-ks_i}$

and sent to the BR, which forwards it to the relevant Pub_c who is waiting for this acknowledgment:

$$\begin{aligned}&\forall ks_i \in \mathcal{KS}:\\ &\quad ack_i = Enc(nc_i \oplus H(pubK_{pub_c}), msrK_{Pub_c-ks_i})\\ &\quad ks_i \rightarrow BR:\ ack_i\\ &\quad BR \rightarrow Pub_c:\ ack_i\end{aligned}$$

Pub_c uses the master key linked to each KeyStore to decrypt the received acknowledgments. It then calculates the hash value of its public key $h_c = H(pubK_c)$ to verify whether the received nonce is the same nonce that was shared with that KeyStore during Phase I using the Cmp function:

$$Pub_c : \sum_{i=1}^{N} Cmp(Dec(ack_i, msrK_{Pub_c-ks_i}), h_c, nc_i) \stackrel{?}{=} 0$$

This phase ends whenever Pub_c receives the acknowledgments from every KeyStore and verifies the received nonces successfully. Otherwise, the protocol will not be able to proceed.

C. Message Key Distribution

After setting up a master key between Pub_c and every ks_i, Pub_c creates a $msgK$ to encrypt the published messages. Saving this key or any part of it on every KeyStore will put the entire system under real danger if *at least* one of these KeyStores gets compromised. Instead of that, we adopt so-called secret splitting [7, p.70] by splitting the $msgK$ into different shares and saving these shares securely within the different KeyStores without disclosing the $msgK$ itself. To achieve that, Pub_c generates $N-1$ ($N = |\mathcal{KS}|$) random keys using a secure random number generator $rndK_1, rndK_2, \ldots, rndK_{N-1}$ where the size of each of these keys is equal to $msgK$'s size (i.e., λ). Then, Pub_c computes the value of $rndK_N$ by XORing the generated random keys with $msgK$. Finally, Pub_c builds a message (msg_i) by concatenating each of these keys, the identifier of topic t ($h_t = H(t)$), and the hash value (h_c) of its public key. Pub_c shares the produced message with every KeyStore after encrypting it and its mac_i using the relevant shared master key $msrK_{Pub_c-ks_i}$:

$$\begin{aligned}&Pub_c : msgK = \ SymKgen(\lambda)\\ &Pub_c : rndK_1, \ldots, rndK_{N-1} = Split(N-1, \lambda)\\ &Pub_c : rndK_N = msgK \oplus rndK_1 \oplus \cdots \oplus rndK_{N-1}\\ &\forall rndK_i \in \{rndK_1, \ldots, rndK_N\}:\\ &\quad Pub_c : msg_i = rndK_i || h_t || h_c\\ &\quad Pub_c : mac_i = HMAC(msg_i, msrK_{Pub_c-ks_i})\\ &Pub_c \rightarrow BR : h_c \mathop{||}_{i=1}^{N} c_i = Enc(msg_i || mac_i, msrK_{Pub_c-ks_i})\end{aligned}$$

To control who is able to get this key, Pub_c creates a credential $CR^{pubK_{Pub_c}}_{pubK_{CA}}$ to authorize a CA (directly) and all subscribers that fulfill certain conditions $Conds$ and trusted by CA (indirectly) to retrieve the N shares from the KeyStores. Pub_c uses its private key $privK_{Pub_c}$ to sign this credential:

$$\begin{aligned}&Pub_c : CR^{pubK_{Pub_c}}_{pubK_{CA}} = Sig(CR^{pubK_{Pub_c}}_{pubK_{CA}}, privK_{Pub_c})\\ &Pub_c \rightarrow BR : CR^{pubK_{Pub_c}}_{pubK_{CA}}\end{aligned}$$

The broker forwards the received message from Pub_c to every KeyStore. Using the received h_c, each ks_i can determine which master key it should use to decrypt the message. Each ks_i applies the $HMAC$ function on the decrypted message ($dsmg_i$) and compare the output with the decrypted mac ($dmac_i$). Besides that, each ks_i extracts dh_c from decrypted messages and checks whether it is identical to the one delivered in clear text (h_c). If verification is passed, ks_i updates the message key if it is existing; otherwise, it creates a new tuple and adds it to $\mathcal{L}$. Also, $CR^{pubK_{Pub_c}}_{pubK_{CA}}$ is forwarded for every member of $\mathcal{SUB}_t$:

$$\begin{aligned}&BR \rightarrow BR : h_c \mathop{||}_{i=1}^{N} c_i\\ &\forall ks_i \in \mathcal{KS}:\\ &\quad dmsg_i || dmac_i = Dec(c_i, msrK_{Pub_c-ks_i})\\ &\quad HMAC(dmsg_i, msrK_{Pub_c-ks_i}) \stackrel{?}{=} dmac_i \wedge dh_c \stackrel{?}{=} h_c\\ &\quad Update(h_c, dh_t, rndK_i)\\ &BR \rightarrow \mathcal{SUB}_t : CR^{pubK_{Pub_c}}_{pubK_{CA}}\end{aligned}$$

Deleting any KeyStore after this phase requires retransmitting a new $msgK$ while adding a new one will affect the subsequent transmitted $msgK$s only.

D. Encrypted Message Transmission

Pub_c needs to guarantee the confidentiality of published messages to prevent unauthorized subscribers from disclosing them. At the same time, it wants to ensure that the message was not manipulated. To achieve that, Pub_c uses the $HMAC$ function to create mac_m for each published message (m). Then, Pub_c encrypts the message and mac_m using $msgK$ that was generated in the previous phase. Pub_c transmits the encrypted message (C_m) to the broker:

$$\begin{aligned}&Pub_c : mac_m = HMAC(m, msgK)\\ &Pub_c \rightarrow BR : C_m = Enc(m || mac_m, msgK)\end{aligned}$$

E. Key Retrieval and Message Decryption

BR forwards the encrypted message and credential to every Sub_j in $\mathcal{SUB}_t$. To decrypt that message, each Sub_j needs to communicate with the KeyStores to retrieve the N secrets to reconstruct the actual $msgK$. It is important to note that this communication does not need to go through BR. Each Sub_j in $\mathcal{SUB}_t$ forms a request R which contains the topic identifier t_{id}, the publisher identity h_c which can be calculated based on the information of a credential, and a nonce nc_r. Sub_j uses its private key $privK_{Sub_j}$ to sign the nc_r, t_{id} and h_c and includes the signature in the request. Then, Sub_j sends the R with its public key $pubK_{Sub_j}$ and a list of all credentials $\mathcal{CR}_{Sub_j}$ ($m = |\mathcal{CR}_{Sub_j}|$) to prove that CA trusts the Sub_j and it fulfills all conditions stated in $CR^{pubK_{Pub_c}}_{pubK_{CA}}$ ($CR^{pubK_{Pub_c}}_{pubK_{CA}} \in \mathcal{CR}_{Sub_j}$):

$$\begin{aligned}&BR \rightarrow \mathcal{SUB}_t : C_m\\ &Sub_j : R = t_{id} || h_c || nc_r || Sig(t_{id} || h_c || nc_r, privK_{Sub_j})\\ &Sub_j \rightarrow \mathcal{KS} : R || pubK_{Sub_j} || CR^{CA}_{Sub_j}\end{aligned}$$

Each ks_i receives the request, verifies the signature of R as well as all provided credentials in $\mathcal{CR}_{Sub_j}$. If all signatures are valid, ks_i searches in $\mathcal{L}$ to find the tuple that contains the secret key using received t_{id} and h_c. Then, ks_i extracts The RoT (i.e., $pubK_{Pub_c}$) and checks weather it can find a path of trust links $pubK_{Sub_j}$ with $pubK_{Pub_c}$ based on credentials provided by Sub_j. If so, ks_i encrypts the $randK_i$ using the $pubK_{Sub_j}$ and sends it to back to Sub_j:

$$\forall ks_i \in \mathcal{KS}:$$
$$\sum_{i=1}^{m} Ver(CR_i, CR_i.Authorizer) \stackrel{?}{=} 0; CR_i \in \mathcal{CR}_{Sub_j}$$
$$Ver(R, pubK_{Sub_j}) \stackrel{?}{=} 0$$
$$ks_i \rightarrow Sub_j : c_i = Enc(rndK_i, pubK_{Sub_j})$$

Sub_j decrypts the received message and extracts $rndK_i$. By XORing all N received shares, Sub_j reconstructs the $msgK$ and uses this key to decrypt the message and to check its integrity:

$$\forall ks_i \in \mathcal{KS}, Sub_j : rndK_i = Dec(c_i, privK_{Sub_j})$$
$$Sub_j : msgK = rndK_1 \oplus rndK_2 \oplus \cdots \oplus rndK_N$$
$$Sub_j : dmsg||dmac_m = Dec(C_m, msgK)$$
$$Sub_j : HMAC(dmsg, msgK) \stackrel{?}{=} dmac_m$$

V. Protocol Analysis

A. Informal Security Analysis

This section provides an informal security analysis of our protocol based on the assumed threat model (see Section II-B). During the phase of exchanging the master key, external attacks and a malicious broker need to access or compute the private key of every KeyStore to decrypt C and extract $msrK_{Pub_c - ks_i}$. However, using RSA as an encryption algorithm and choosing a sufficiently large modulus prevent the attackers from extracting the private keys. Similarly, attackers will not benefit from ack_i (during Phase II) or c_i messages (during the message key distribution) without having $msrK_{Pub_c - ks_i}$. It is not feasible for attackers to break the protocol by targeting the AES symmetric algorithms itself since it is considered impervious to all attacks. Also, by choosing $expT$ and changing the master key frequently, the process becomes even more difficult. One way to get $msgK$ and decrypt C_m maliciously is by compromising the N KeyStores (or if all of them decide to collude with an attacker). Even though such a case is possible, it is unlikely, especially when we use many KeyStores. It is important to mention that if one KeyStore gets compromised and/or refuses to follow the protocol honestly (i.e., DoS), Sub_j will not be able to reconstruct $msgK$. Although we are not solving this issue in this paper, we already have some solutions, such as using a different schema for secret sharing or simply using multiple KeyStores that hold the same share. Adopting these solutions will be considered as future work.

B. Implementation and Performance Evaluation

1) Implementation and Test-bed: We evaluate our proposed protocol's performance characteristics only from the publisher's side. End-to-end performance evaluation will be considered as a future work. We used two baseline systems to compare our approach to. The first one is a system without any security (we refer to it as *clear*). The second one uses SSL/TLS to secure the communication between the publisher and broker as recommended by the MQTT standard (we refer to this system as *SSL*). We have implemented our proposed protocol using the C programming language. The Publisher, KeyStore, and Subscriber were implemented using the Eclipse Paho MQTT client. We also used the open-source Mosquitto broker without any changes. All cryptography operations were implemented using the OpenSSL library. We used a 128-bit AES-CBC key for the master key and session keys and 1024-bit RSA keys for public/private keys. The publisher was running on a Raspberry Pi 3 Model B+, which includes a Broadcom BCM2837B0 SoC based on a 1.4 GHz 64-bit quad-core ARM Cortex-A53 CPU, 1 GB RAM, and a BCM43143 WiFi chip. KeyStores were running on another Raspberry Pi 3 Model B+. The broker was installed on a laptop running 64-bit Ubuntu 20.04 with a 1.9 GHz Intel Quad-Core i7 CPU and 16GB RAM.

Phase	Operation	Time (ms)
Phase I	$Connect()$	131.20
	$Enc(msrK_{Pub_c - ks_1} \|\| nc_1, pubK_{ks_1})$	13.38
	$Send(C \|\| PubK_c)$	0.38
	Total of above + other operations	156.27
Phase II	$Receive(ack_1)$	18.93
	$Dec(ack_1, msrK_{Pub_c - ks_1})$	0.15
	Total of above + other operations	19.09
Phase I +II	Total	175.36

Table I: Time performance of Phase I and II using 1 ks ($N = 1$).

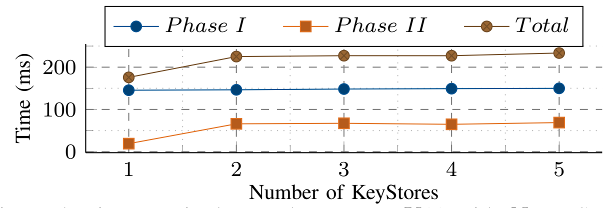

Figure 4: Time required to exchange $msrK$(s) with N KeyStores.

2) Performance Analysis:

a) Master Key Exchange: Table I details the performance evaluation of the main operations during master key exchange phase. The table shows that the time to connect with the broker consumes almost 80% of the entire time of this phase, while the asymmetric encryption of the master key depletes around 8% only. It is critical to mention that the time to receive the master key(s) by the KeyStore(s) and to receive the acknowledgment(s) by Pub_c is variable and depends on the network's latency and bandwidth. Figure 4 presents the performance evaluation of the same phase when multiple KeyStores are used. The figure shows that the change of time in Phase I is very minimal. The most introduced overhead occurs in Phase II since Pub_c needs to receive one acknowledgment from each ks. The more KeyStores are used, the more time is required to receive all acknowledgments by Pub_c. Handling these acknowledgments introduces negligible overhead since we use symmetric decryption to decrypt each one of them.

b) Message Key Distribution: Figure 5 represents the required time to establish a connection between the Pub_c and BR using one KeyStore ($N = 1$) and to setup a $msgK$. The figure also shows a comparison with the other baseline systems. For each system, we repeat the measurement 600 times. As expected, T^{conn}_{clear} was the fastest. The measurement also shows

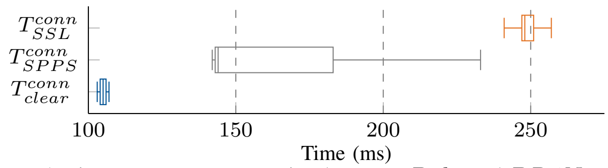

Figure 5: Time to setup a connection between Pub_c and BR $(N = 1)$.

Operation	Time (ms)
$Connect()$	131.20
$Sig(CR_{pubK_{CA}}^{pubK_{Pub_c}}) + send(CR_{pubK_{CA}}^{pubK_{Pub_c}})$	13.30
$Enc(msgK, msrK_{Pub_c - ks_1}) + send(.)$	0.26
Total	144.76

Table II: Detailed time performance of T_{SPPS}^{conn} using 1 ks $(N = 1)$.

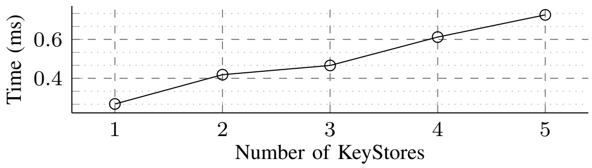

Figure 6: Time required for encrypting and sending shares.

that T_{SPPS}^{conn} is executed 70% faster than T_{SSL}^{conn} and 38% slower than T_{clear}^{conn}. This overhead comes from the different operations during the connection setup, which includes signing a credential, encrypting share(s), connecting with a broker, and sending the encrypted share(s) and credential. Table II details the time consumed by each of these operations.

To show the effect of using multiple KeyStores, we repeat our test using a different number of KeyStores each time, and we got the result presented in Figure 6. In this figure, we do not include the times required to connect with the broker and sign the credential since these functions occur once, as they will perform almost the same regardless of the number of KeyStores. We only consider the time required to encrypt N shares and send them to the broker. Results show that the increase in the time is around $0.1\,\mathrm{ms}$ for each new ks.

c) Encrypted Message Transmission: Figure 7 shows the introduced overhead of encrypting and sending different sizes of messages, as stated in the horizontal axis. For each message size, we repeated the measurement 100 times. The figure shows a comparison between $SPPS$ and $clear$ systems only since using SSL will introduce almost the same overhead as our system if the same encryption algorithm and key length were chosen. The Figure shows that using bigger message sizes will introduce more overhead. Based on our results, the introduced overhead is around $0.2\,\mathrm{ms}$ for each $5\,\mathrm{kB}$.

Summary: It is important to note that the overhead of setting the master key(s) occurs once. Our proposed system also introduces 70% less overhead compared to SSL when Pub_c needs to (re)-establish the connection with the broker. Moreover, even though we tested our proposed system using a platform with limited resources, it outperforms other solutions that provide end-to-end security over the Pub/Sub model such as ABE by orders of magnitude. A publisher needs around $1\,\mathrm{s}$ to encrypt a 128-bit key using ABE despite that it was running on a laptop with 1.60GHz Quad-Core i7 CPU [5].

VI. Related Work

Securing the Pub/Sub system is a common goal in the IoT domain [9]. Pal et al. [10] proposed a system for a content-based

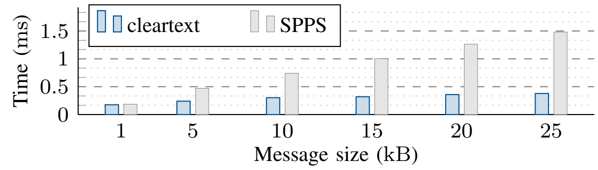

Figure 7: The required time for encrypting and sending data.

Pub/Sub model where Ciphertext Policy (CP)-ABE was used to encrypt published messages. Only subscribers who fulfill the access policy can decrypt that messages. Ion [11] proposed the use of the ABE to encrypt a symmetric key that is used to encrypt the data instead of encrypting the data itself. Only subscribers with sufficient properties can get the symmetric key and consequently decrypt the message. Although these solutions ensure both data authorization and confidentiality, they come with a massive overhead resulting from pairing operations needed in ABE.

VII. Conclusion

Using the Pub/Sub model to support V2C communication seems promising if security concerns are solved. This paper proposes a secure policy-based Pub/Sub model that allows vehicles to encrypt and control access to the published messages. Our solution leverages semi-honest KeyStores to guarantee the end-to-end confidentiality of V2C communication without trusting the brokers. Experimental results show that our solution outperforms alternative state-of-the-art methods such as SSL/TLS and ABE. Based on that, our solution is considered as a very efficient method to ensure end-to-end secure communication using Pub/Sub model.

References

[1] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf, "Security issues and requirements for internet-scale publish-subscribe systems," in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. IEEE, 2002, pp. 3940–3947.

[2] R. A. Nofal, N. Tran, C. Garcia, Y. Liu, and B. Dezfouli, "A Comprehensive Empirical Analysis of TLS Handshake and Record Layer on IoT Platforms," in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019.

[3] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE symposium on security and privacy (SP'07)*. IEEE, 2007.

[4] M. Ion, G. Russello, and B. Crispo, "Design and implementation of a confidentiality and access control solution for publish/subscribe systems," *Computer networks*, vol. 56, no. 7, pp. 2014–2037, 2012.

[5] X. Wang, J. Zhang, E. M. Schooler, and M. Ion, "Performance Evaluation of Attribute-based Encryption: Toward Data Privacy in the IoT," in *2014 IEEE International Conference on Communications (ICC)*. IEEE, 2014.

[6] M. Blaze and A. D. Keromytis, "The KeyNote trust-management system version 2," 1999.

[7] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 1996.

[8] ISO, "Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1," International Organization for Standardization, Geneva, Switzerland, ISO ISO/IEC 20922:2016, 2016.

[9] E. Onica, P. Felber, H. Mercier, and E. Rivière, "Confidentiality-preserving publish/subscribe: A survey," *ACM computing surveys (CSUR)*, vol. 49, no. 2, pp. 1–43, 2016.

[10] P. Pal, G. Lauer, J. Khoury, N. Hoff, and J. Loyall, "P3S: A privacy preserving publish-subscribe middleware," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 476–495.

[11] M. Ion, "Security of publish/subscribe systems," Ph.D. dissertation, University of Trento, 2013.