Scalable Identity and Key Management for Publish-Subscribe Protocols in EDS

Prashant Anantharaman, Dartmouth College
https://cs.dartmouth.edu/~pa
pa@cs.dartmouth.edu

CYBER RESILIENT ENERGY DELIVERY CONSORTIUM

Funded by the U.S. Department of Energy and the U.S. Department of Homeland Security | cred-c.org

Joint work with

- Kartik @ Illinois
- Elizabeth @ Illinois
- David @ Illinois
- Jason
- Sean

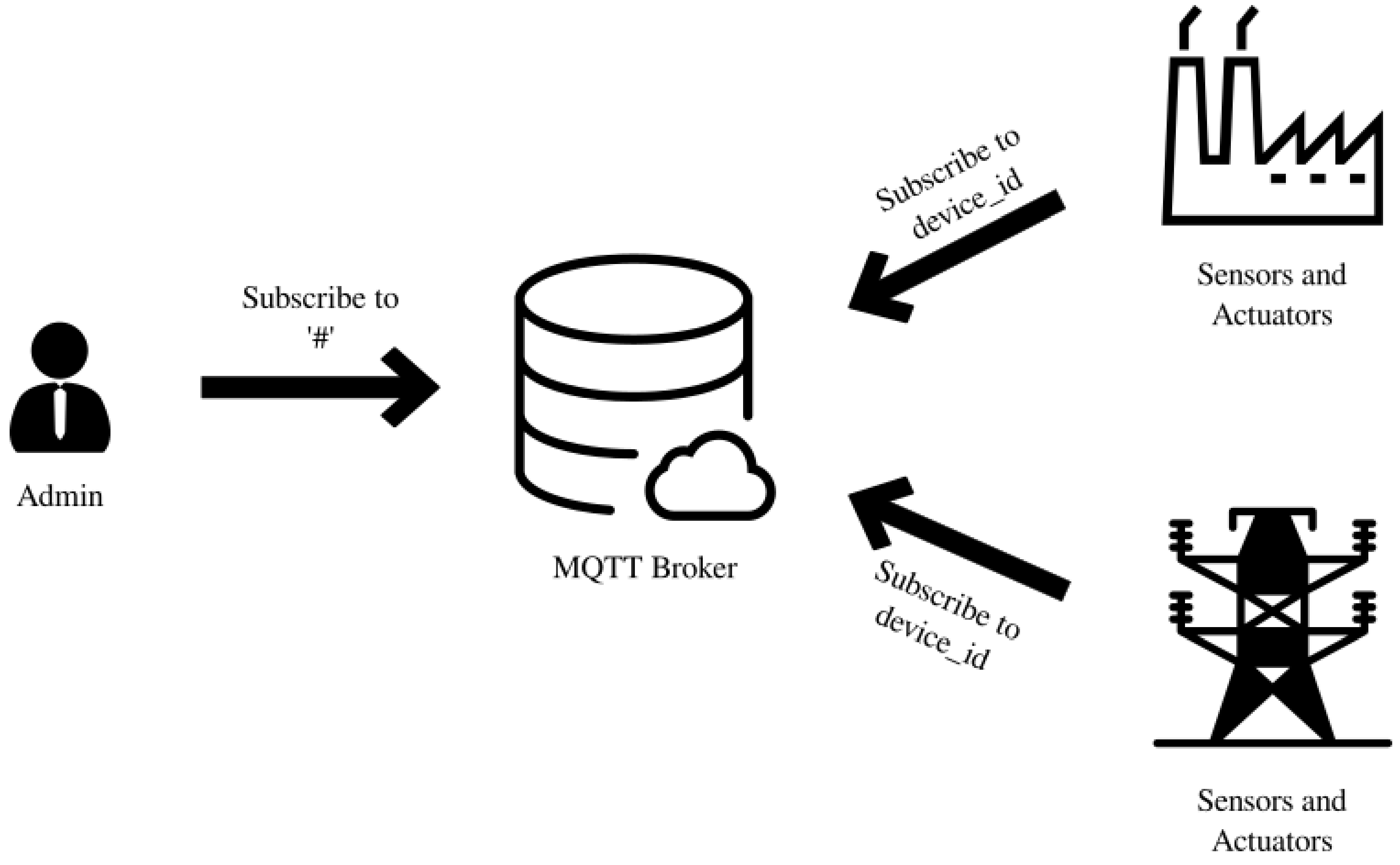
Overview

Publish Subscribe Protocols

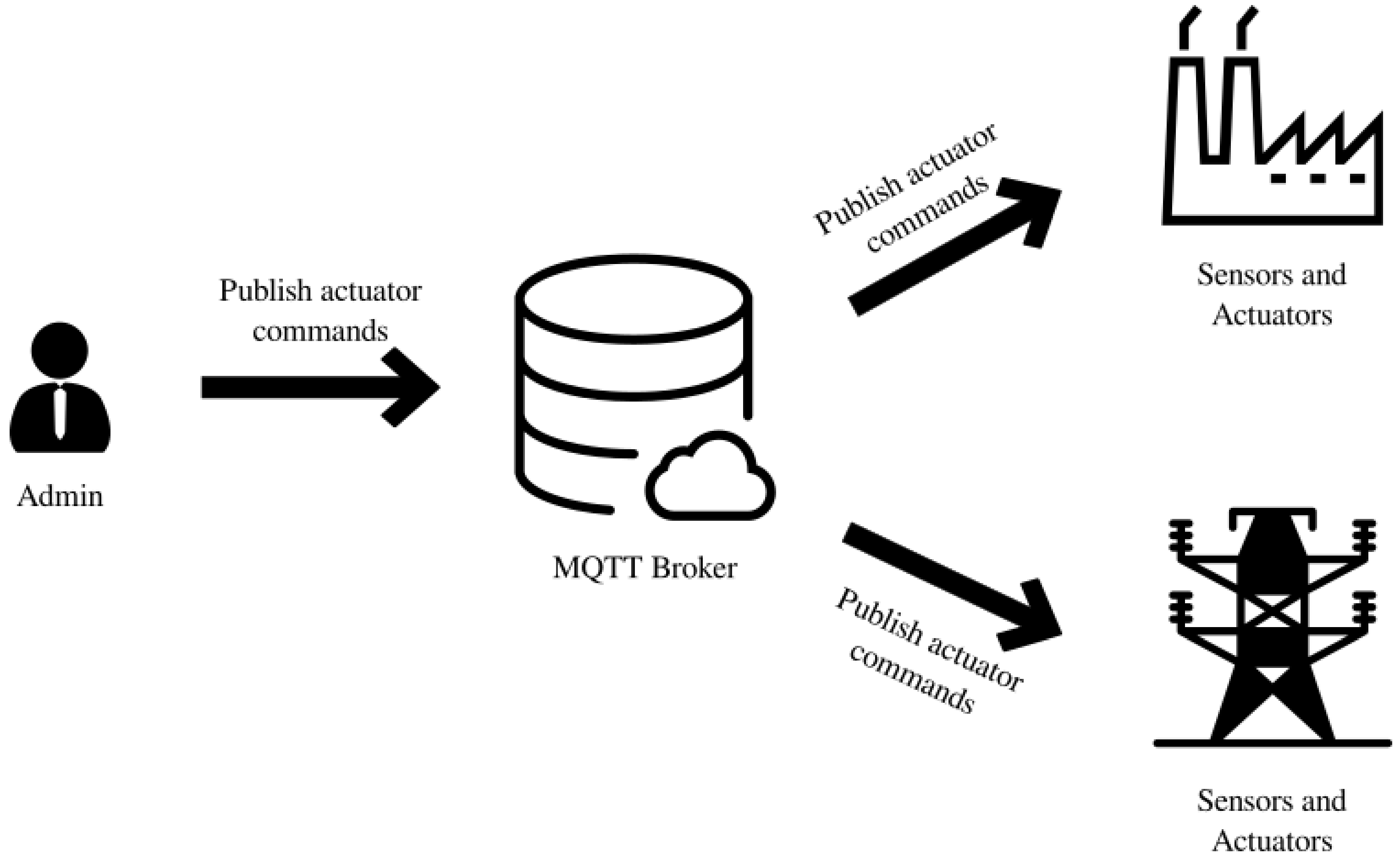
- Producers publish messages
- Consumers wait for certain messages by making use of subscriptions
- The routing of these messages to consumers is handled by a broker
- Most widely used protocols include
    - MQTT - 47,993 implementations on the internet
    - AMQP - 194,989 implementations
    - STOMP - 60 implementations
    - GOOSE - ethernet layer protocol within a substation, most SEL and ABB devices come with an option to enable GOOSE

Data source: shodan.io

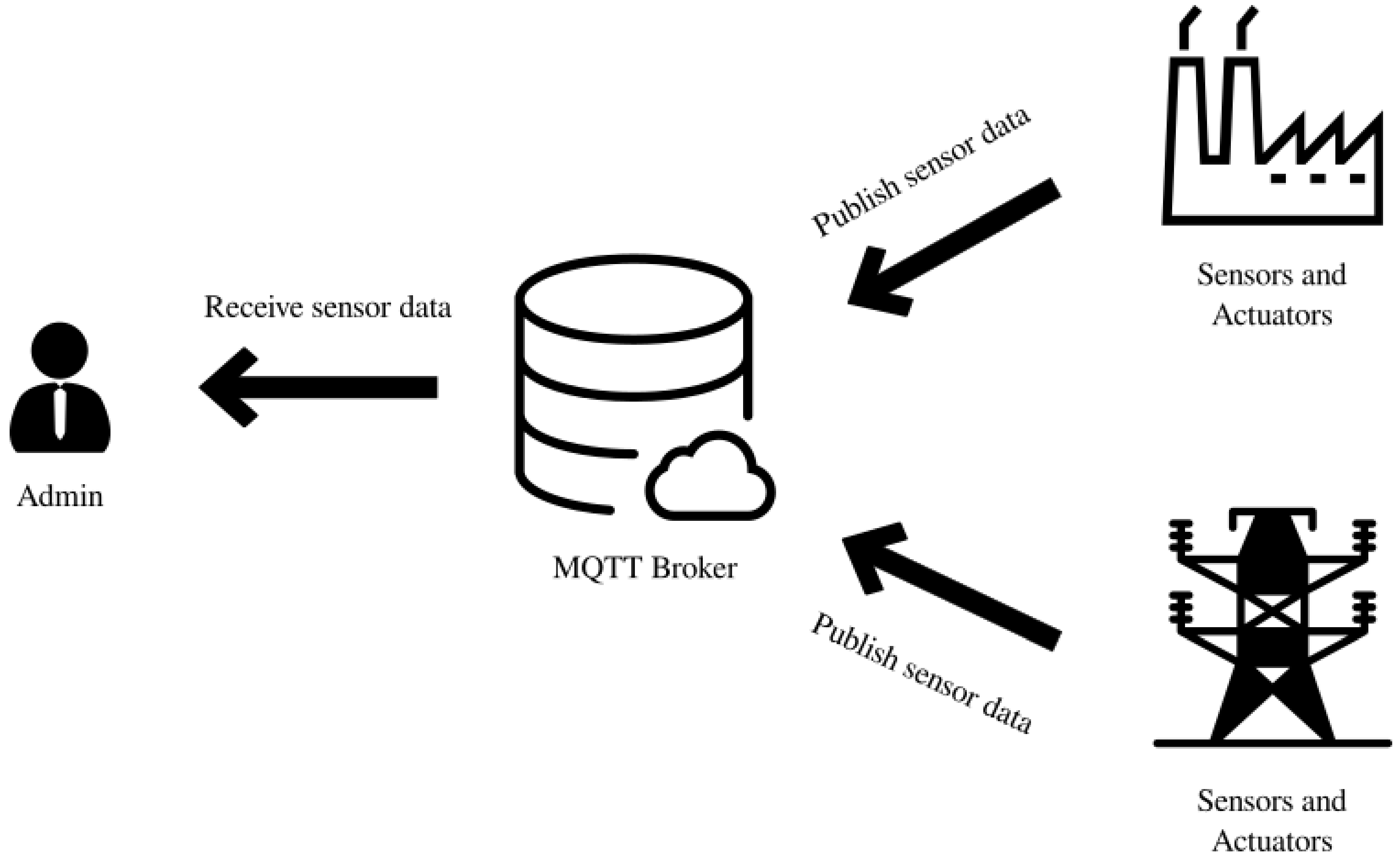
Subscription



Controller publishing commands

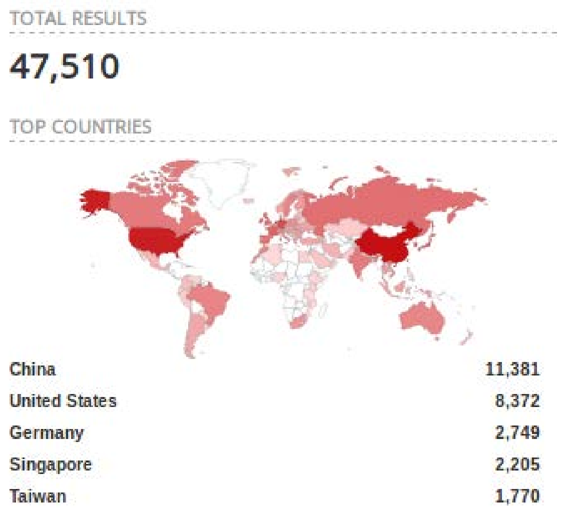

Admin
Publish actuator commands
MQTT Broker
Publish actuator commands
Publish actuator commands
Sensors and Actuators
Sensors and Actuators

Controller receiving data



The MQTT Protocol

- Our industry partners use MQTT as a SCADA protocol.
    - Smart meters - It is being used to enable communications between smart meters and controllers
    - Grid control - manages control devices ranging from generators, thermostats and other sensors
- Our partners acknowledge the issues in MQTT implementations pointed out by us, and are working with us for a key management scheme
- Limited messages - connect, subscribe, unsubscribe, push
- Everyone subscribed on a channel will get the message
    - No details of sender
    - Subscribe to '#'
    - TLS is optional
- The image is from a scan we ran using shodan.io



TOTAL RESULTS

47,510

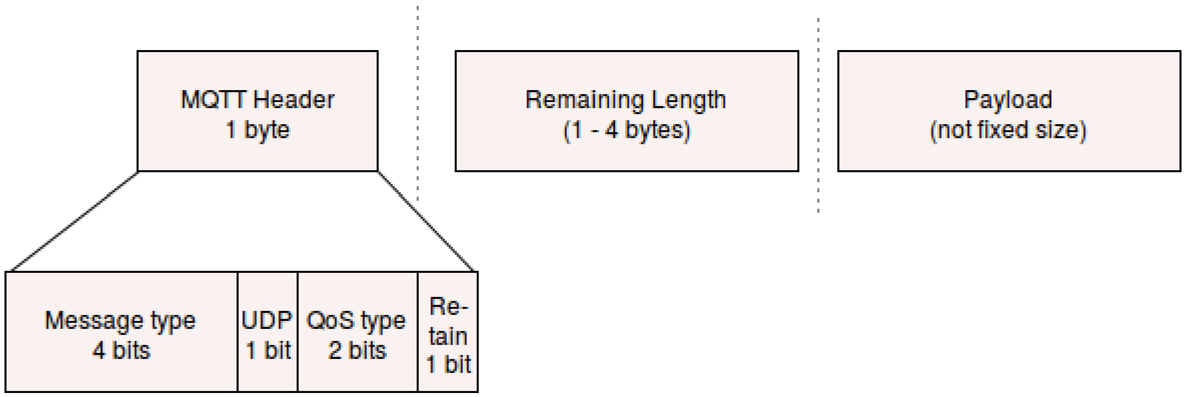
TOP COUNTRIES

China	11,381
United States	8,372
Germany	2,749
Singapore	2,205
Taiwan	1,770

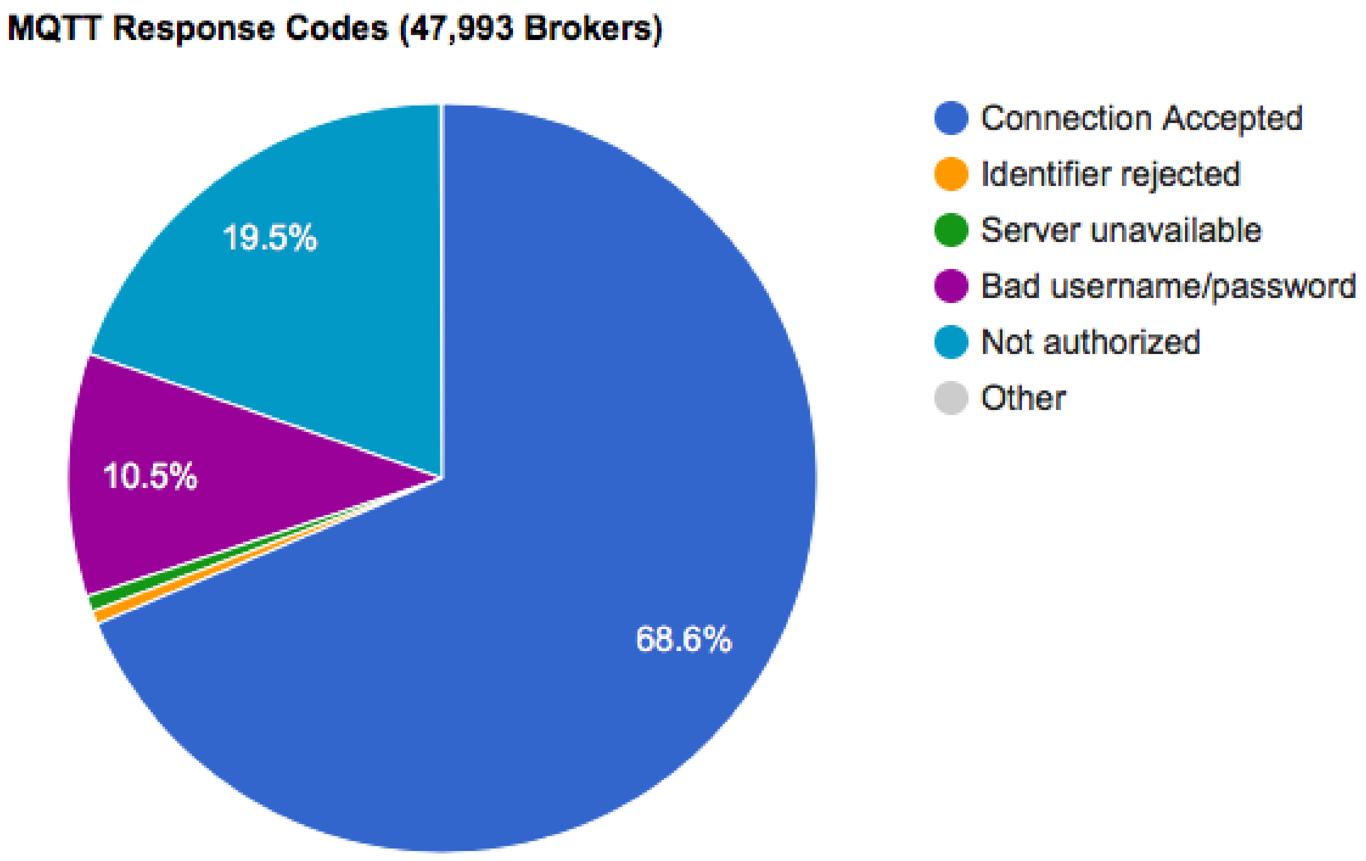
The MQTT Protocol (contd.)

Issues identified:

- No senders identification
- Active broker compromise is an issue
- No real key management solution for MQTT
- Existing solutions talk about lightweight crypto solutions, but not key management or revocation



The MQTT Protocol (contd.)



Data source: shodan.io

GOOSE Protocol

- **Replay attacks** and **Spoofing attacks** are very easy to perform due to the lack of identification and confidentiality mechanisms.
- It is an ethernet layer protocol, and all the attacks on the ethernet layer are still possible, and we can do very little to mitigate them.
- Packet Spoofing is a major issue, since GOOSE ignores packets based on SqNum field. In case the packet is less than the SqNum field in the previous spoofed packet, it is ignored, leading to several packets to be ignored.

Contributions

- We provide results from measurements of the authentication (or lack thereof) of publish-subscribe protocols appearing on the internet.

- Demonstrate the effectiveness of our macaroon-based key management scheme in the context of publish-subscribe schemes.

- We built a toolkit for our scheme, and run experiments on an implementation of MQTT. We show that in terms of CPU-time and developer effort, our systems performs well within the acceptable latency bounds.

Overview

- Introduction
  - What are Publish-subscribe protocols?
  - Issues in publish-subscribe protocols
  - Contributions
- **Proposed Architecture**
  - Goals
  - Assumptions
  - Our Approach
  - PKI vs Macaroons
  - Protocols
- Implementation
- Results and Discussion
- Conclusions

Goals

- Build a system resilient to active server compromises
- Assign separate keys for long lasting assertions like the identity, and short lasting ones like accesses to channels
- Broker doesn’t see any raw messages, and hence any clients connected without keys won’t see them either
- In case of device compromise, only the channels the device has access to are compromised
- Release the toolkit for the controller and the individual devices

Assumptions

- We can install keys securely during deployment of a device
- The cryptographic primitives are bug-free
- The devices can securely store the identity and attribute channel specific keys

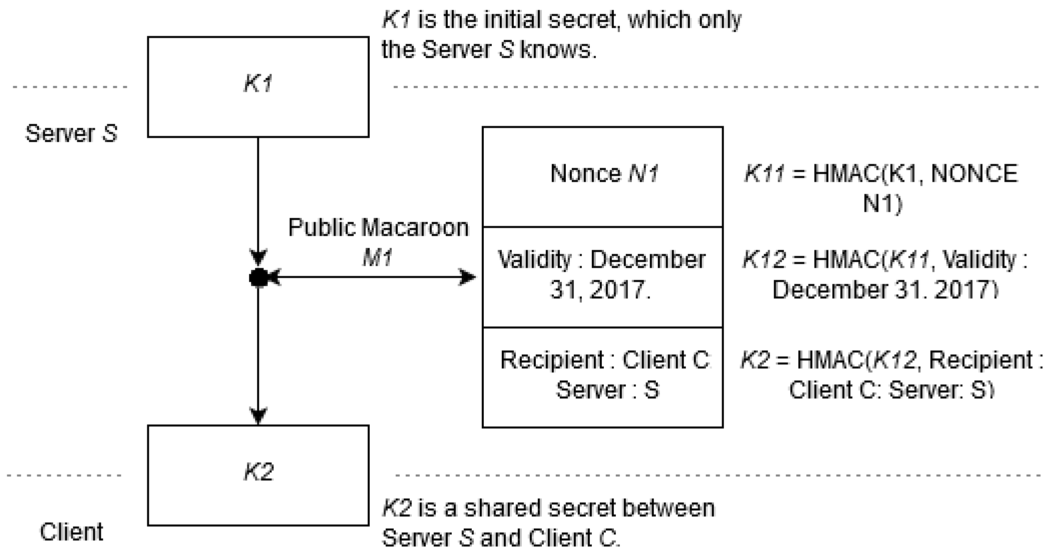
Our Approach

- Each device has two kinds of identities -
    - A core identity - installed either during manufacturing or during deployment.
    - Association attributes - one macaroon per channel. These are shorter lived, and need to be changed frequently.

- An attribute can be formalized as a tuple (P,O, Δ).
- Property P holds for Object O for Time Δ.
- The core identity assertions are made by the manufacturer or deployer, and are verified by the controller.
- The attribute assertions are made by the controller, when provided with the identity assertion.

Macaroons

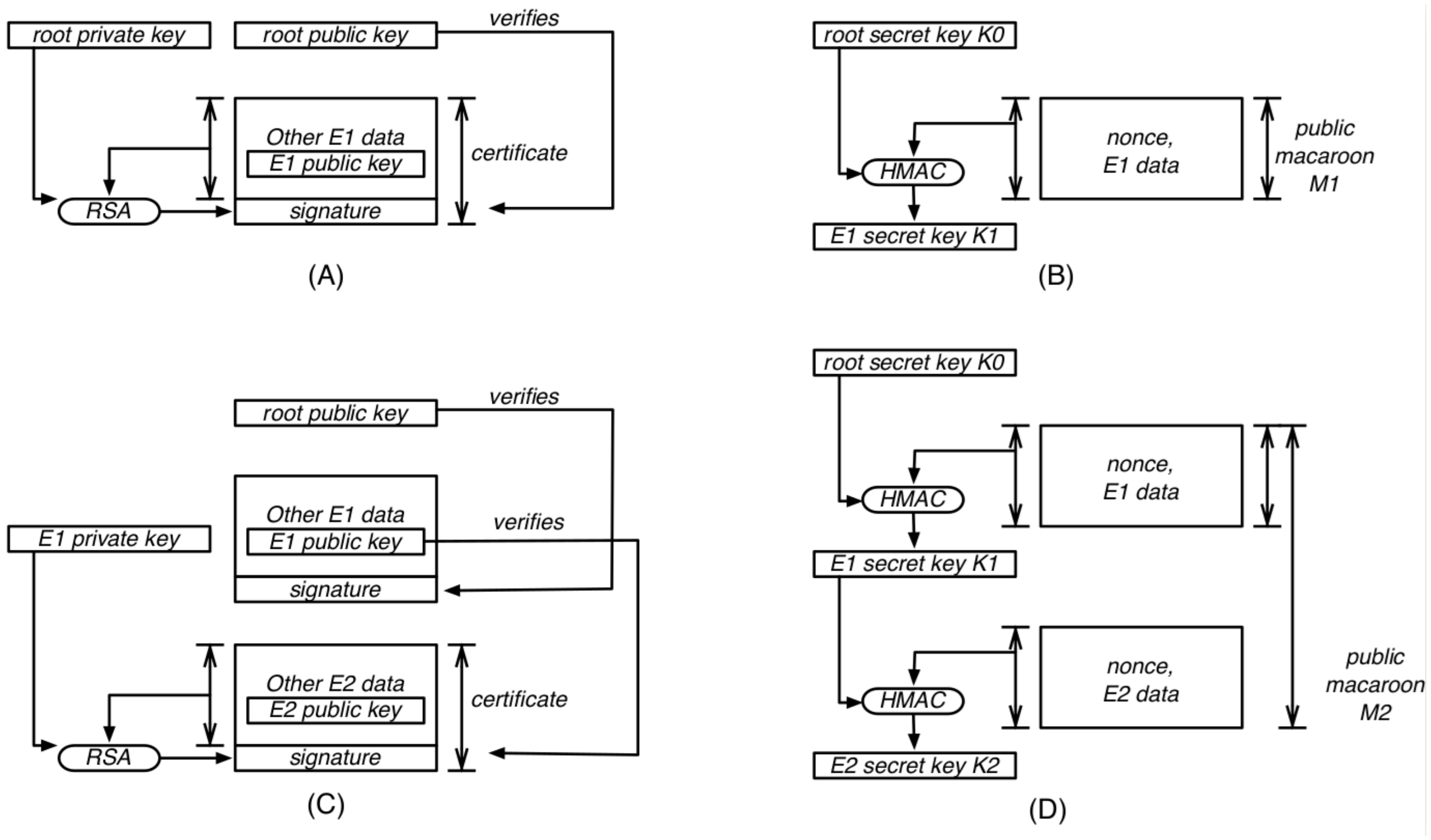
- A macaroon consists of two parts
  - A public part: consisting a random nonce and a set of caveats
  - A secret part: the final HMAC value generated by iteratively computing the HMAC on individual caveats. This secret is used as a shared secret to compute subsequent session keys


K1 is the initial secret, which only the Server S knows.
K1
Server S
Nonce N1
K11 = HMAC(K1, NONCE N1)
Public Macaroon M1
Validity : December 31, 2017.
K12 = HMAC(K11, Validity : December 31. 2017)
Recipient : Client C Server : S
K2 = HMAC(K12, Recipient : Client C: Server: S)
K2
Client
K2 is a shared secret between Server S and Client C.

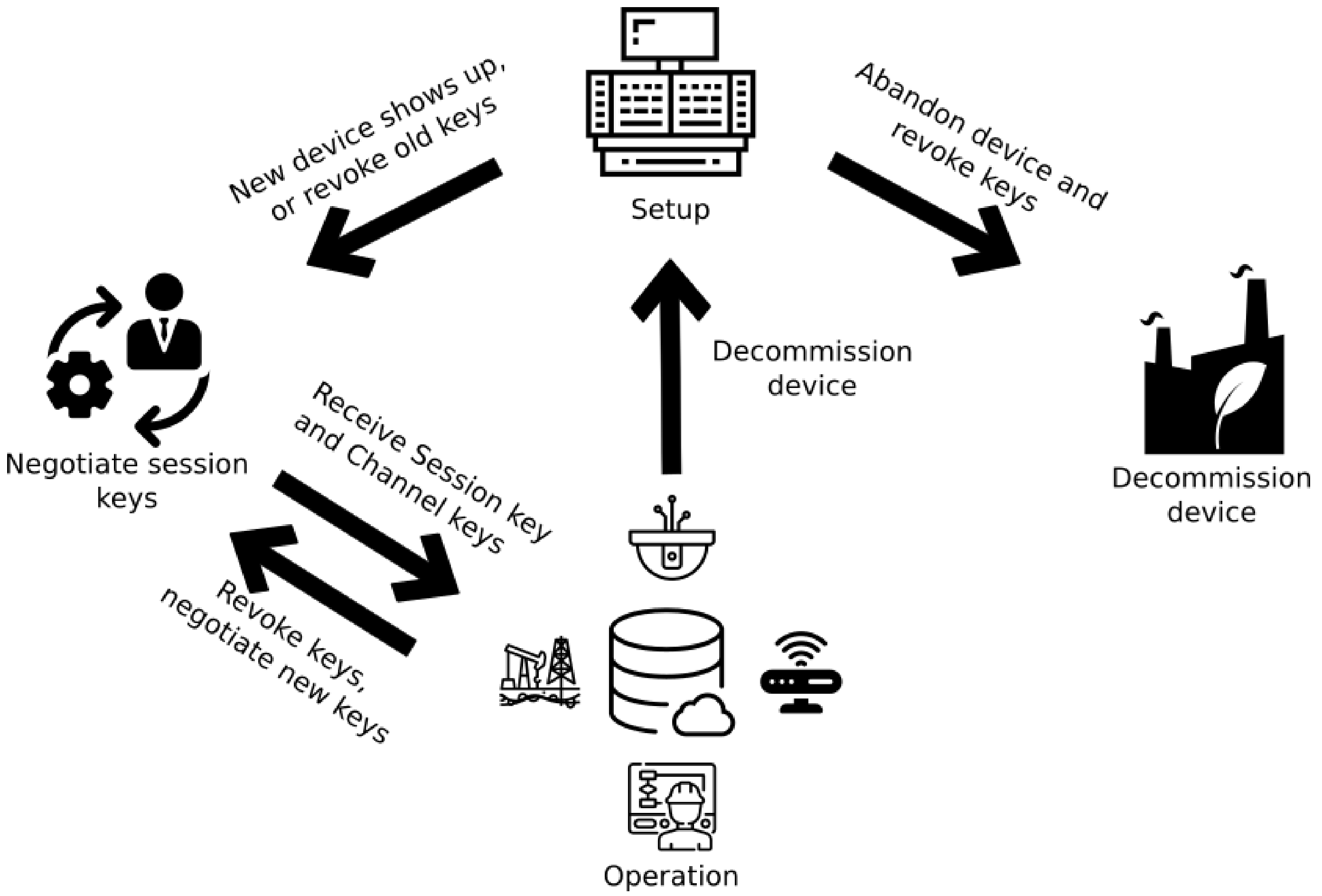
Macaroons (contd.)

```
MACAROON_GENERATE(device $D_{id}$, manufacturer $M$, key $k_1$, caveats $C$)
1.        $M$: $k_2$ := HMAC256 $(N_M)_{k_1}$
2.        $M$: $k_3$ := HMAC256 $(D_{id})_{k_i}$
3.        $M$: $k_4$ := HMAC256 $(M)_{k_i}$
4.        $M$: $k_{i+1}$ := HMAC256 $(C_i)_{k_i}$ for $i$ = 4,5,6...$n$
5.        $M$: $k_{final}$ := HMAC256 (timestamp)$_{k_n}$
6.    $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel$ timestamp $\parallel k_{final}$
```

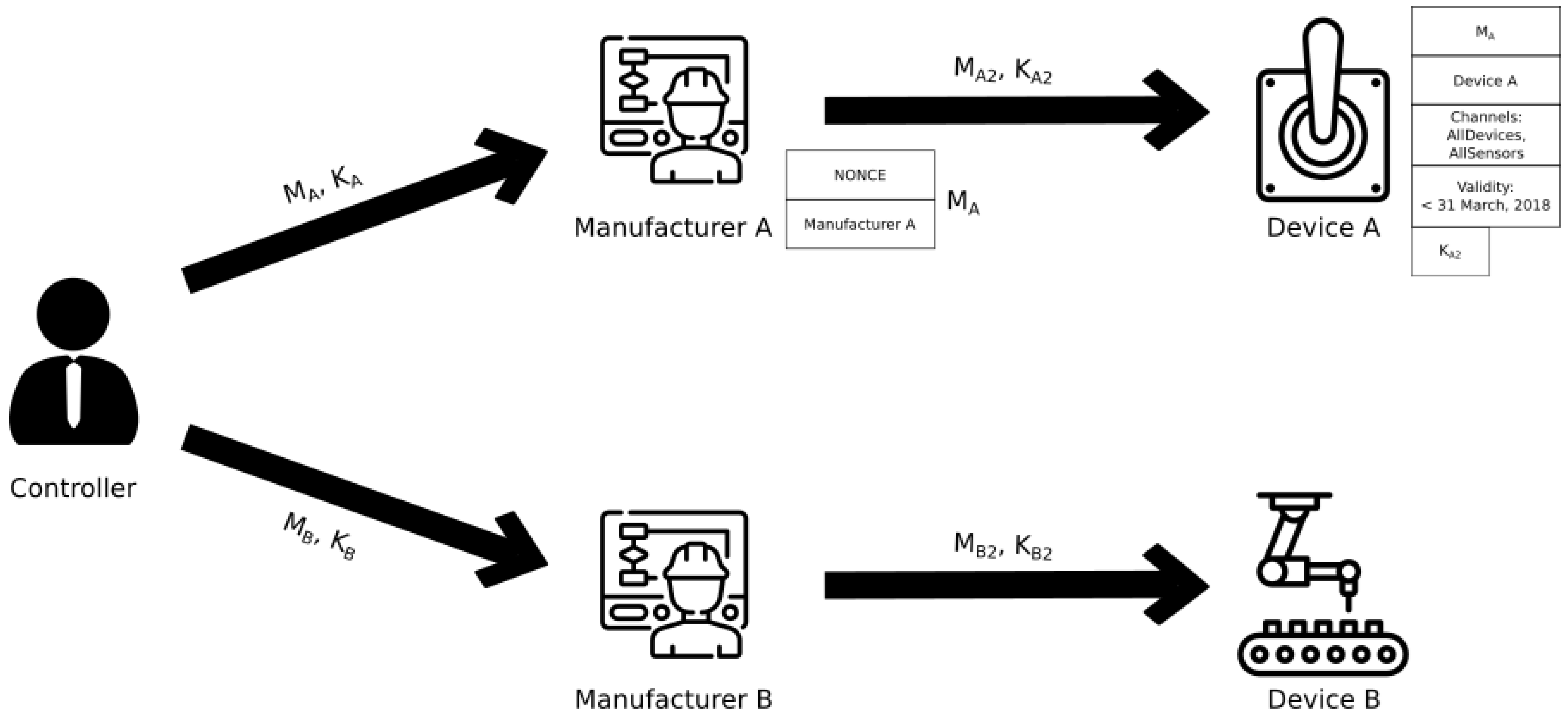
PKI vs Macaroons



Our scheme in the lifecycle of a device in EDS

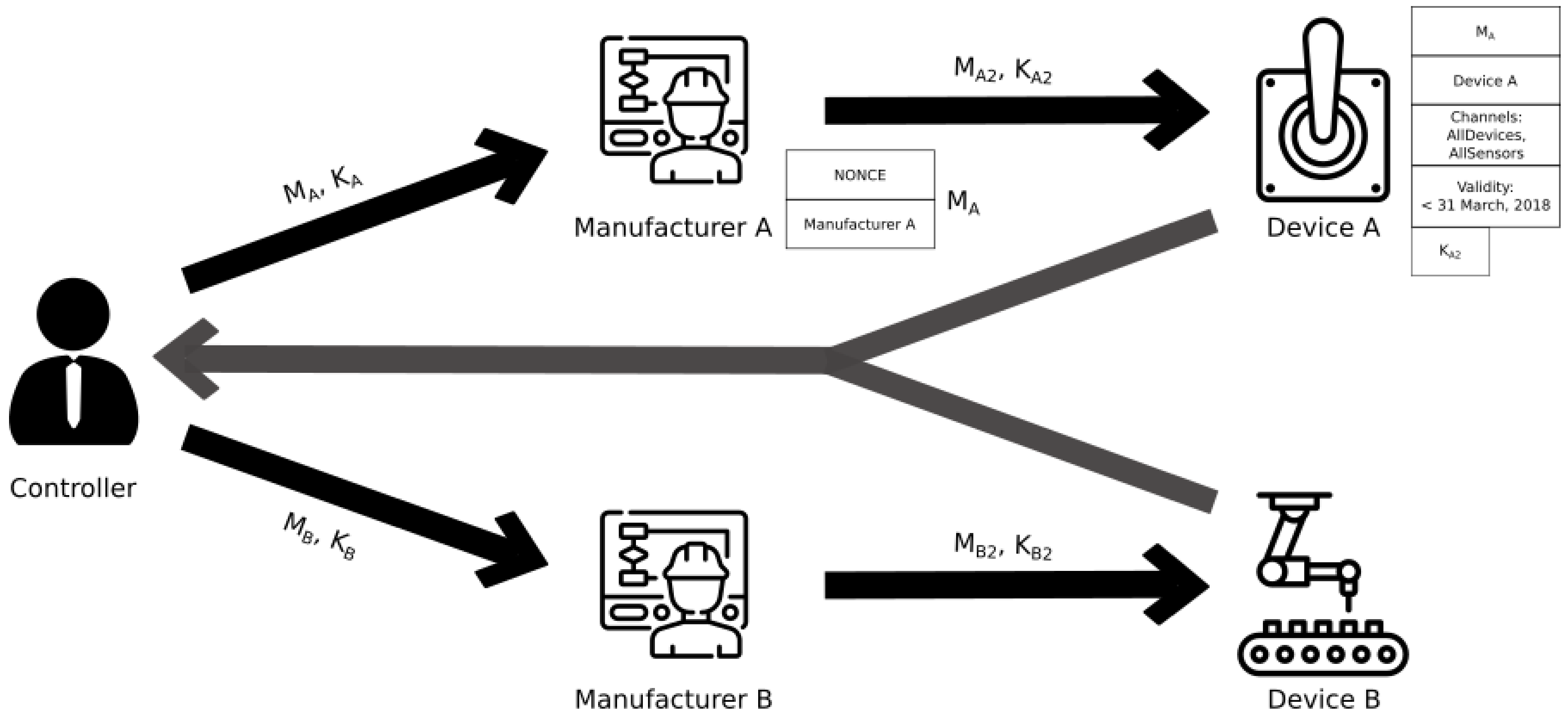

Setup
New device shows up, or revoke old keys
Abandon device and revoke keys
Negotiate session keys
Decommission device
Decommission device
Receive Session key and Channel keys
Revoke keys, negotiate new keys
Operation

Who makes the assertions?



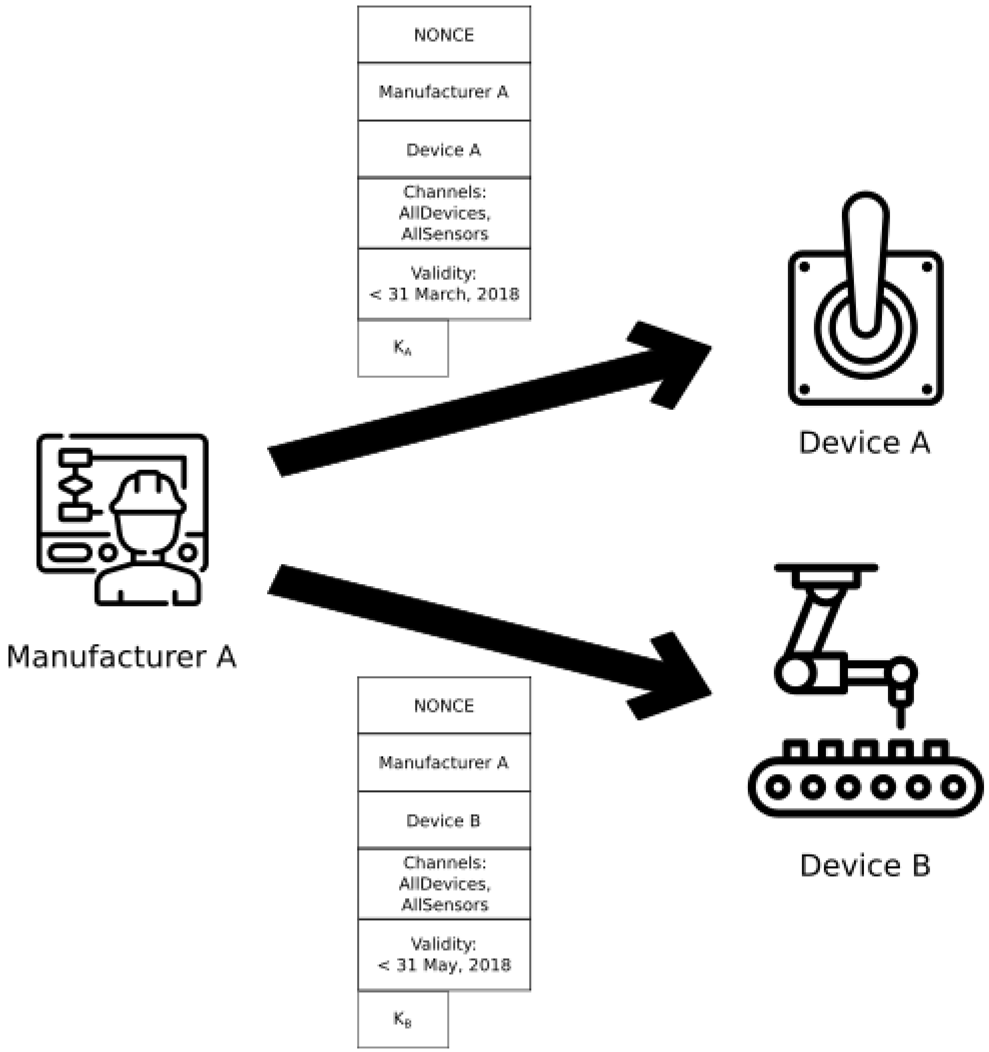
CREDC

Who makes the assertions?



CREDC

Who makes the assertions? (contd.)



Session Keys

- We make use of the J-PAKE algorithm to establish a session key between the device and the controller
- The J-PAKE algorithm is resilient to Known-key attacks, online and offline dictionary attacks, and provides forward secrecy
- It generates a high entropy cryptographic key from a low entropy secret
- This session key is then used to set up the channel specific macaroons

Short Lived Macaroons

```
SHORT_LIVED(device $D_{id}$, key $K_{id}$, caveats $C$, channels $S$, Controller $A$)
1.    $D_{id} \rightarrow A$: $M_{id} \,\|\, \text{timestamp} \,\|\, \text{HMAC256}(M_{id} \,\|\, \text{timestamp})_{k_{id}}$
2.        $A$: $k_{id} := \text{CALC\_KEY}(M_{id})$
3.        $A$: $k_2 := \text{HMAC256}\ (N_{new})_{k_id}$
4.        $A$: $k_3 := \text{HMAC256}\ (D_{id})_{k_i}$
5.        $A$: $k_4 := \text{HMAC256}\ (S_j)_{k_i}\ \forall j \in S$
6.        $A$: $k_{i+1} := \text{HMAC256}\ (C_i)_{k_i}$ for $i = 4,5,6 \ldots n$
7.        $A$: $k_{s_j} := \text{HMAC256}\ (\text{timestamp})_{k_n}$
8.    $A \rightarrow D_{id}$: $M_{id} \,\|\, N_{new} \,\|\, D_{id} \,\|\, C_i \,\|\, \text{timestamp} \,\|\, k_{s_j}$
```

Operation

```
OPERATION(device $D$, broker $B$, receiverGroup $G$)
1.            $D$: $k_c := getChannelKey(G)$
2.            $D$: $m_{data} :=$ data | timestamp
3.            $D$: $m_{mac} :=$ HMAC256$(m_{data})$
4.   $D \rightarrow B$: $m_{final} := enc\,(m_{data} \mid m_{mac})_{k_c}$
5.   $B \rightarrow G$: $m_{final}$
```

CREDC

Revocation

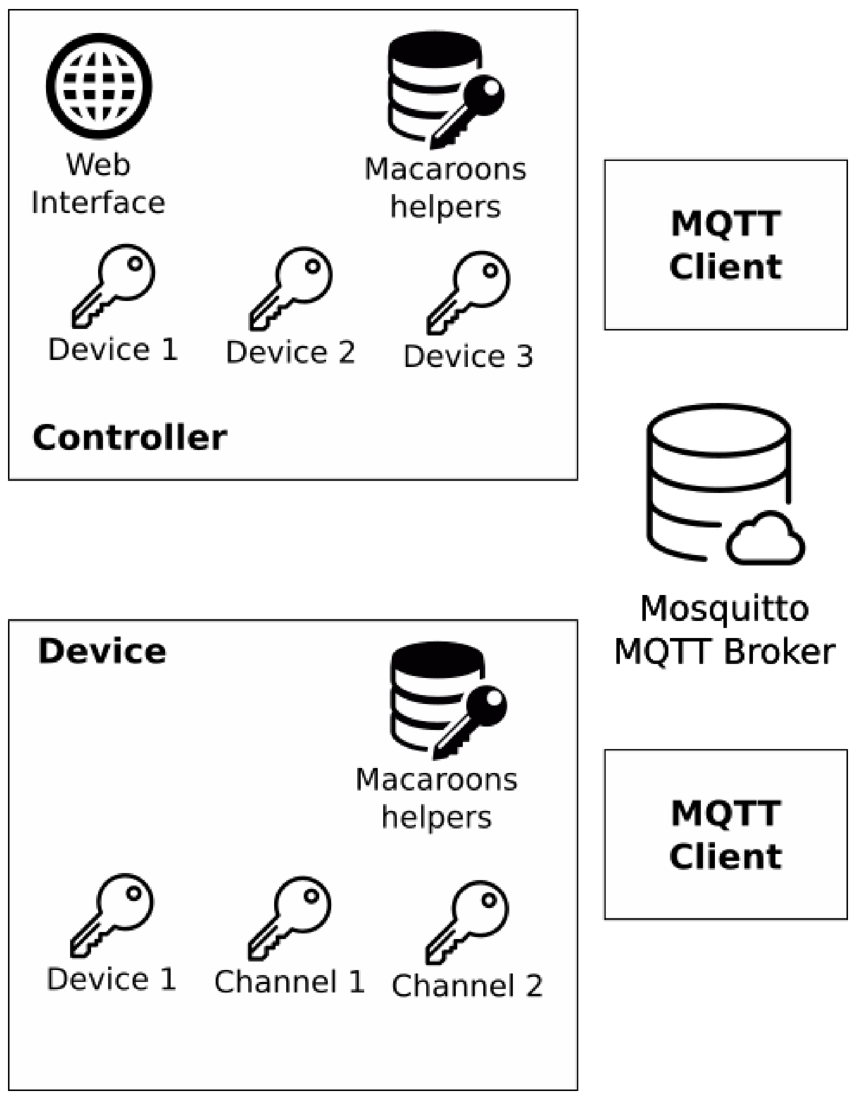
- The short-lived keys expire soon, and are subsequently blacklisted.

- When the channel keys to a particular device need to be revoked, the channel keys given to all the other devices with access to the channel are also revoked. Hence, there is a list maintained of what devices are connected to what channels.

- A whitelist of all the manufacturer keys is maintained, so that the macaroons can be verified, and session keys can be generated from the verified macaroons.

Overview

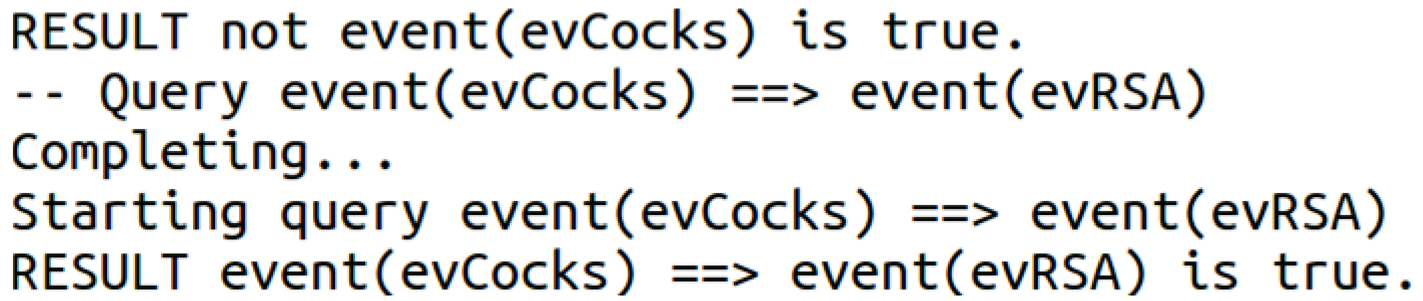
Implementation



- The device and the controller have been implemented in ruby.
- Web Server in Ruby on Rails is a work in progress.
- We made use of Mosquitto MQTT Broker and client off-the-shelf, and built our layer of security on top of it.

Verification

- We made use of Proverif 1.98pl1 to verify our cryptographic protocol.
- We were able to prove that the shared secret *k* used in the session key establishment, is never leaked by the protocol.



```
RESULT not event(evCocks) is true.
-- Query event(evCocks) ==> event(evRSA)
Completing...
Starting query event(evCocks) ==> event(evRSA)
RESULT event(evCocks) ==> event(evRSA) is true.
```

Overview

Preliminary Results

Our experiments were performed on an ARM Firefly RK3288 development board.

- The GOOSE protocol has a prescribed maximum latency of *4ms*.
- Since all the other schemes make use of elliptic curves, we show that elliptic curves are highly infeasible for such constrained devices and show that macaroons are much more usable.

Algorithm	**Creation time**	**Verification time**
Elliptic Curves		
Ed25519-256 bits	25.79 ms	29.34 ms
Macaroons		
SHA-1-HMAC	662 µs	513 µs
SHA-256-HMAC	761 µs	566 µs

Developer Effort

- The device specific programs would have to be instrumented with our macaroon generation and verification protocols.
- In our experimental setup, we only had to add 20 lines of python code to the device clients for MQTT.
- The biggest development effort in similar architectures, is re-creating the MQTT broker. Our scheme does not require any broker changes.

Overview

Conclusions and Future work

- We built a system that meets our latency and security goals

- We intend to release our tool by June 2018

Future work:

- Integrating into the shared secret option of SSP21 protocol

- Working with GE to incorporate our scheme into their MQTT ecosystem

- We are also in talks with another potential industrial partner and already had a few initial discussions

Thank you

- Toolkit will be available soon.

Prashant	pa@cs.dartmouth.edu
Kartik	palani2@illinois.edu
Elizabeth	ereed@illinois.edu
Jason	reeves@cs.dartmouth.edu
Sean	sws@cs.dartmouth.edu

CYBER RESILIENT ENERGY DELIVERY CONSORTIUM

www http://cred-c.org

@credcresearch

facebook.com/credcresearch/

Funded by the U.S. Department of Energy and the U.S. Department of Homeland Security