Scalable Identity and Key Management for Publish-Subscribe Protocols in the Internet-of-Things

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

Kartik Palani (UIllinois), Sean Smith

Overview

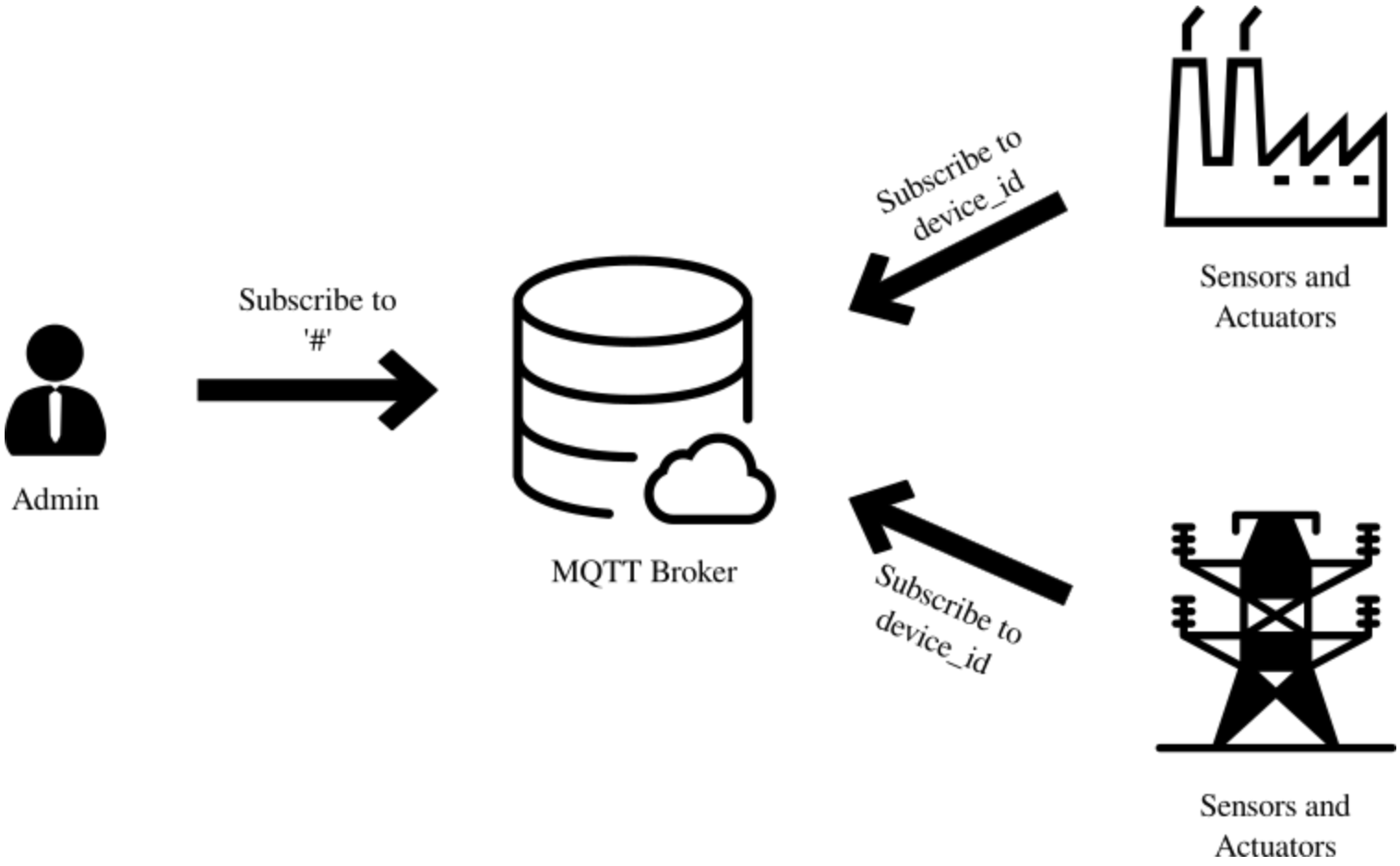
- **Introduction**
- Proposed Architecture
- Implementation
- Results
- Conclusions

Publish Subscribe Protocols

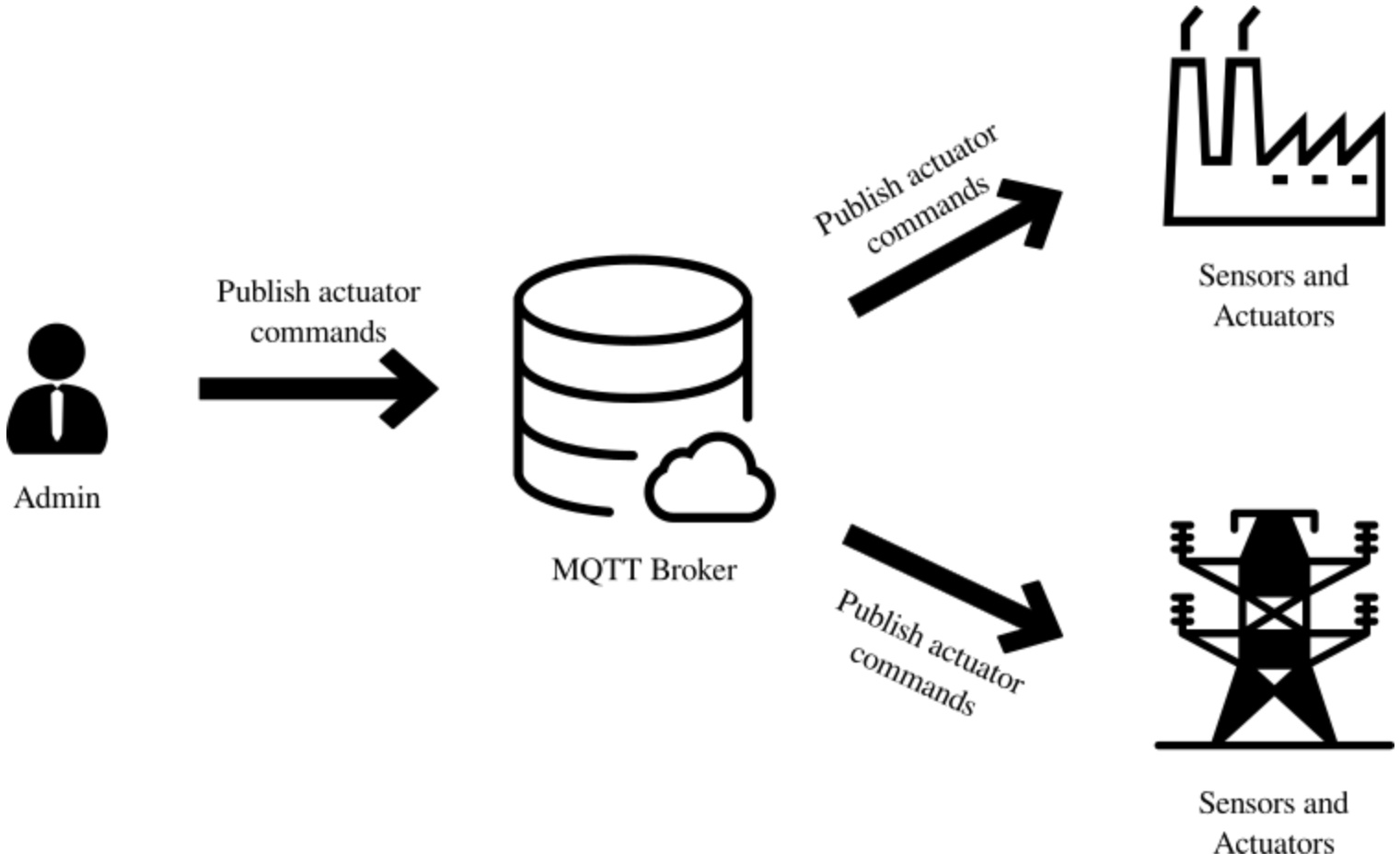
- Producers publish messages
- Consumers wait for certain messages by making use of subscriptions
- The routing of these messages to consumers is handled by a broker
- Most widely used protocols include
  - MQTT - 53,070 implementations on the internet
  - AMQP - 194,989 implementations
  - STOMP - 60 implementations
  - GOOSE - ethernet layer protocol within a substation, most SEL and ABB devices come with an option to enable GOOSE

Data source: shodan.io

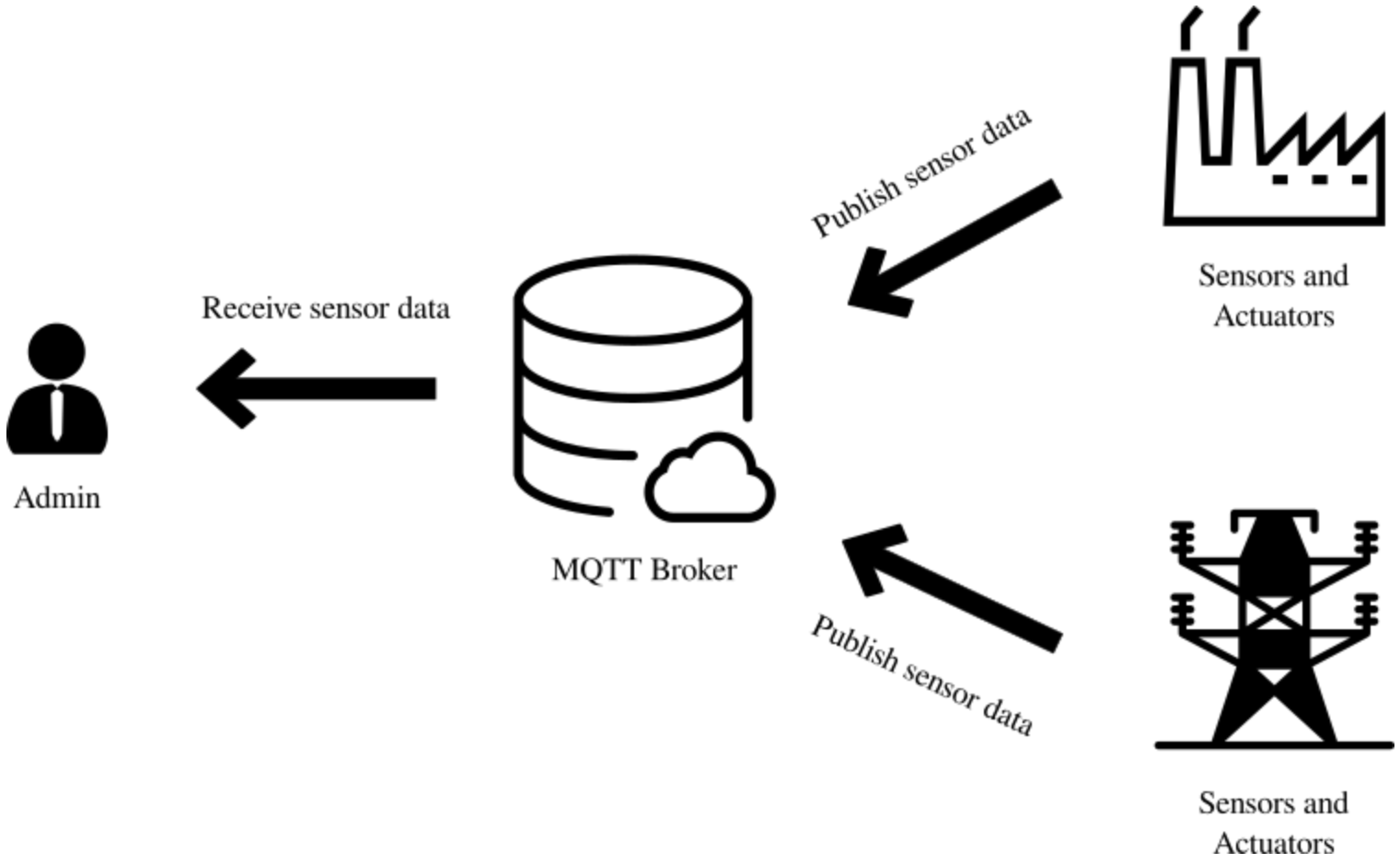
Subscription



Controller publishing commands



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

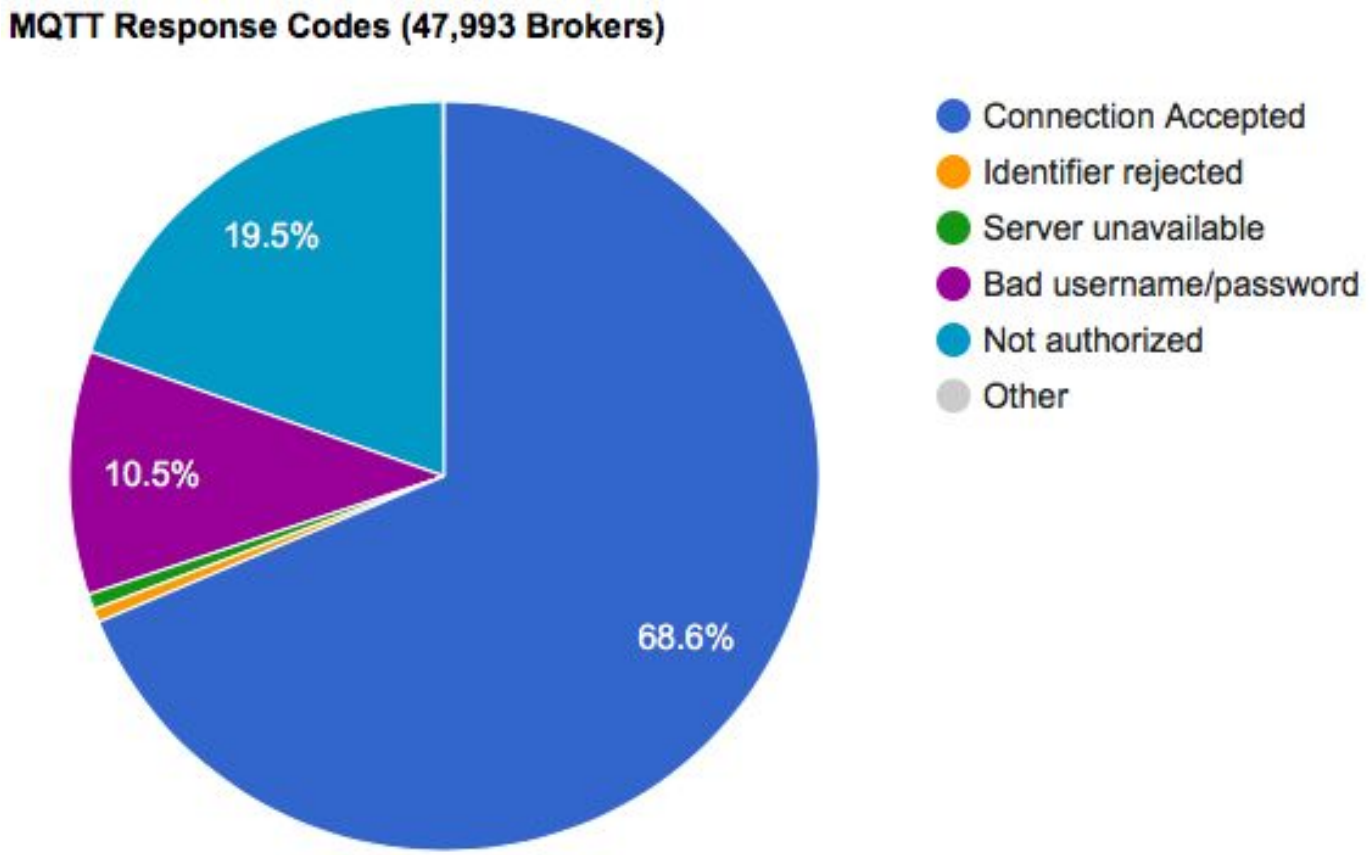
- The image is from a scan we ran using shodan.io

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

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
- **Proposed Architecture**
- Implementation
- Results
- Conclusions

Goals

- Build a system resilient to active server compromises
- Assign separate keys for long-lasting assertions such as the identity, and short-term keys for 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

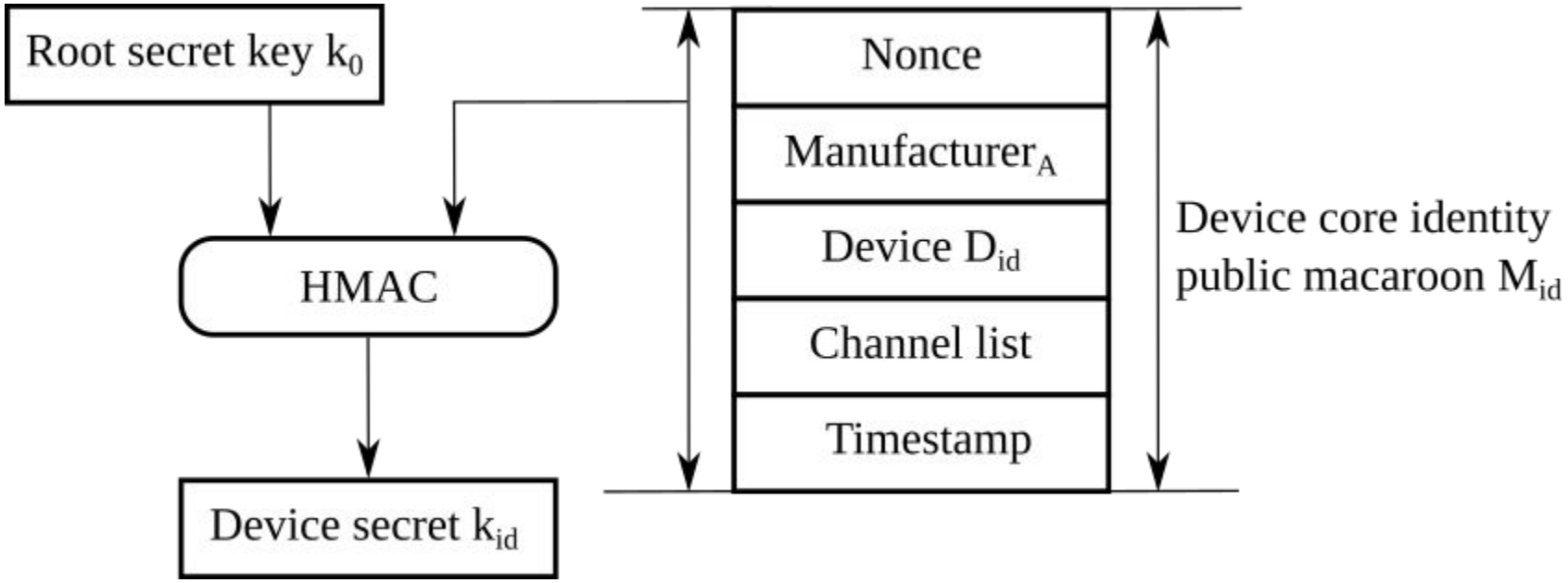
Our Approach

- 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



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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

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 $i = 4,5,6 \ldots n$
5. M: k_{final} := HMAC256 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

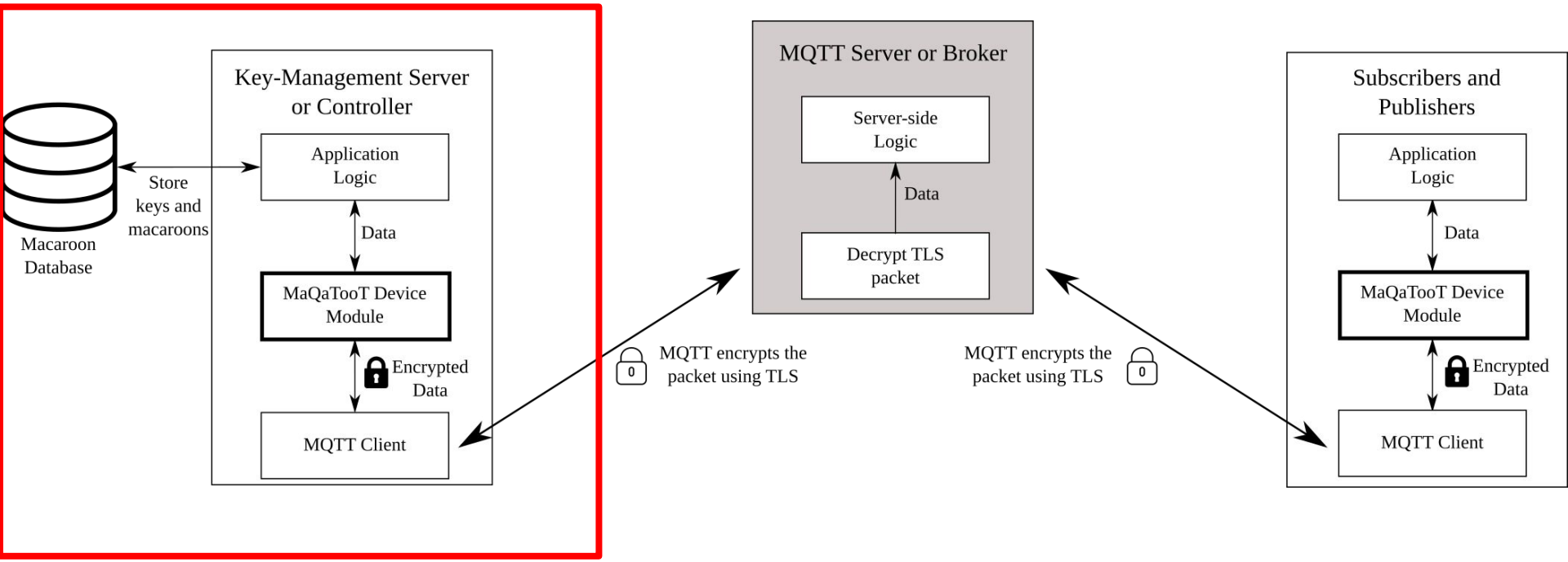
Macaroons (contd.)

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 $(\text{timestamp})_{k_n}$
6. $M \rightarrow D$: $N_M \parallel D_{id} \parallel M \parallel C_i \parallel \text{timestamp} \parallel k_{final}$

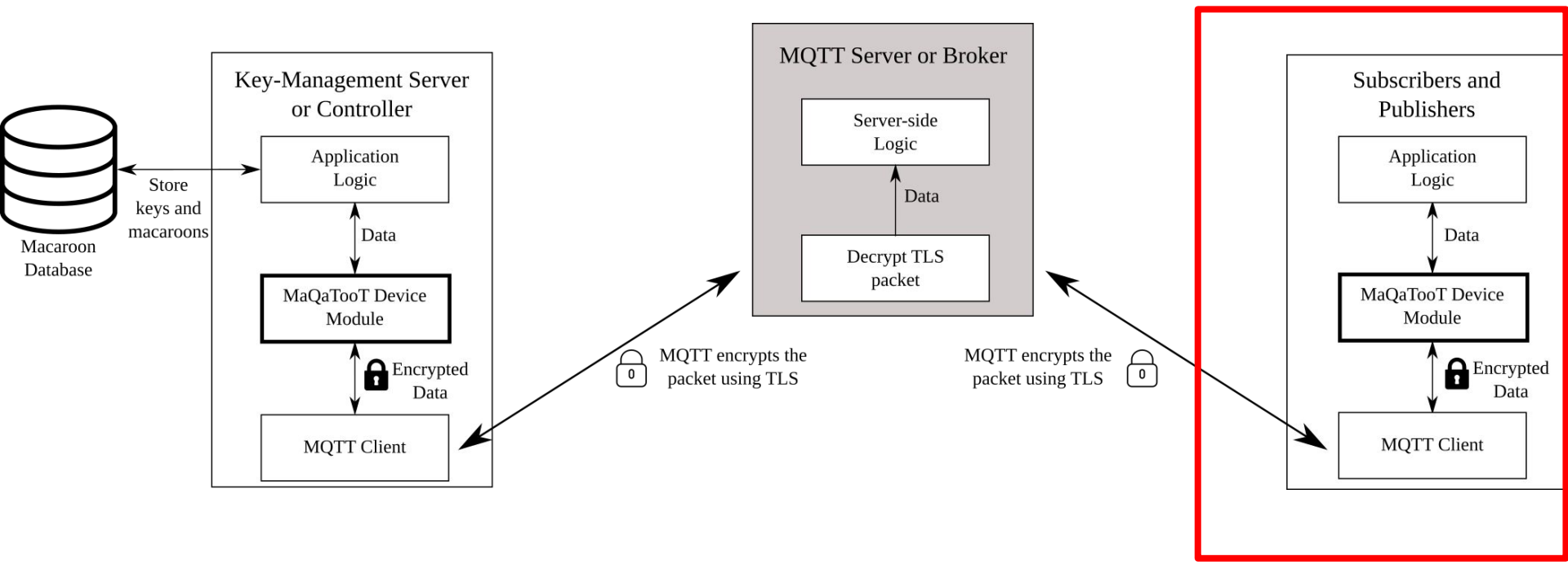
Architecture

- The thicker boxes are modules introduced by us.
- The shaded box has access to encrypted data only.



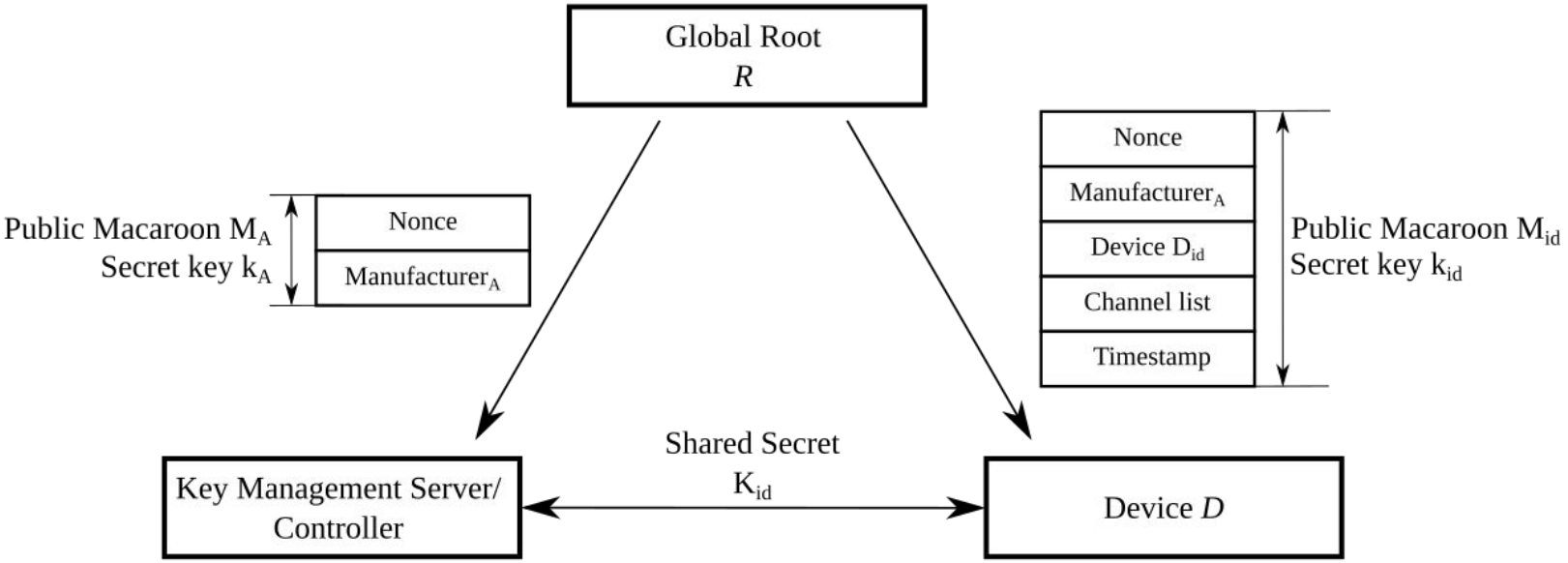
Architecture

- The thicker boxes are modules introduced by us.
- The shaded box has access to encrypted data only.



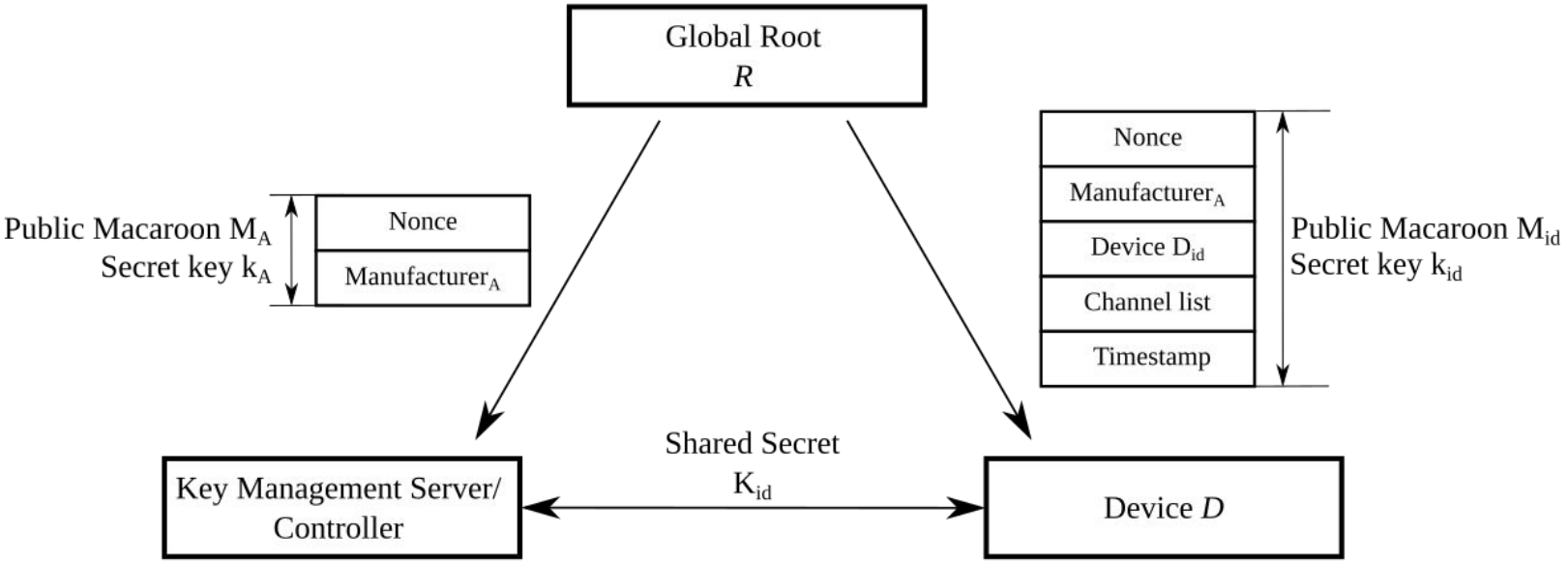
Session Keys

- 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.

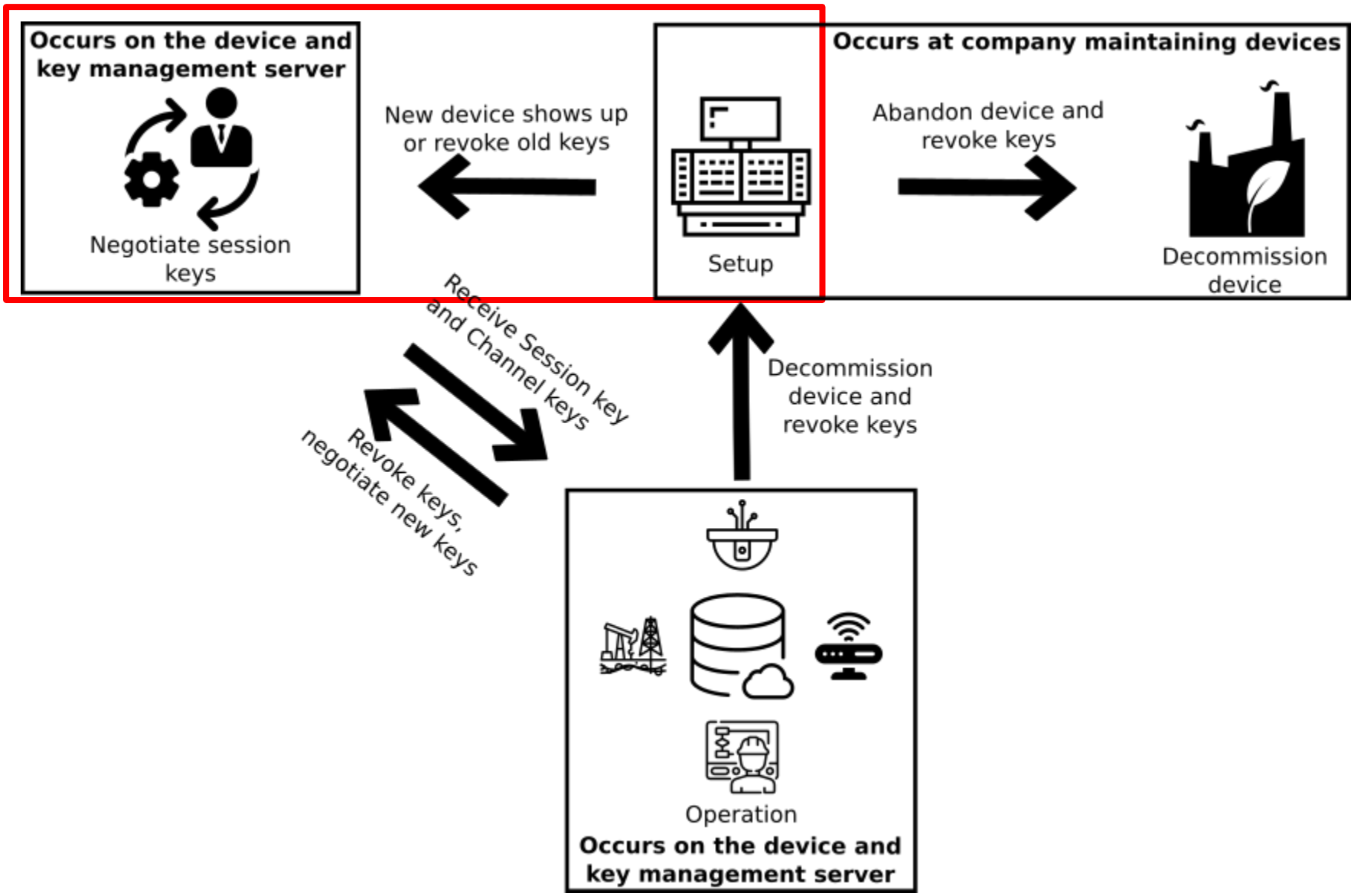


Session Keys (contd.)

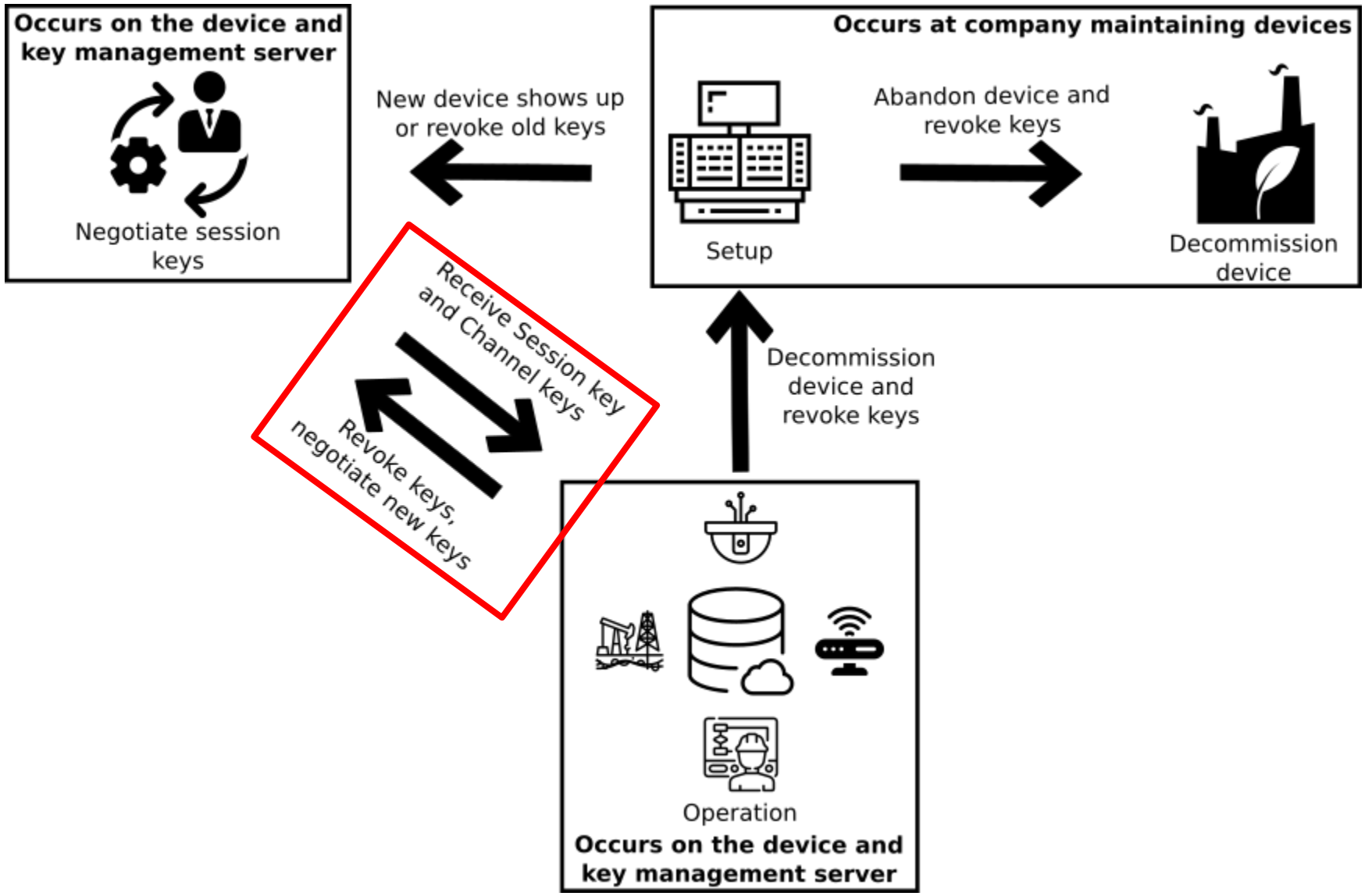
- J-PAKE generates a high entropy cryptographic key from a low entropy secret
- This session key is then used to set up the channel specific macaroons



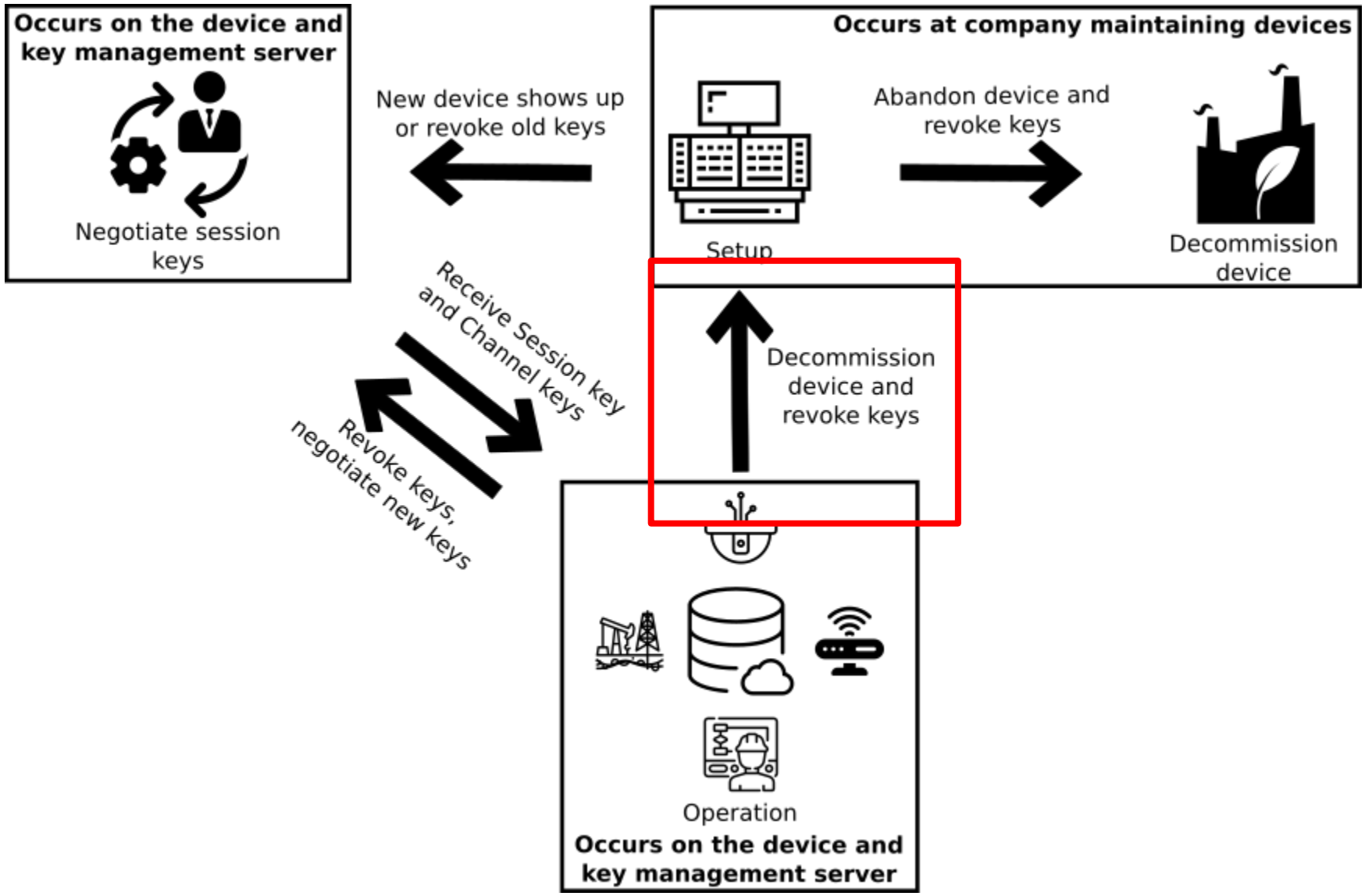
Our scheme in the lifecycle of a device in EDS



Our scheme in the lifecycle of a device in EDS



Our scheme in the lifecycle of a device in EDS



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}

Operation

OPERATION(device D, broker B, receiverGroup G)

1. D: $k_c := getChannelKey(G)$
2. D: $m_{data} := \text{data} \mid \text{timestamp}$
3. D: $m_{mac} := \text{HMAC256}(m_{data})$
4. $D \rightarrow B$: $m_{final} := enc\ (m_{data} \mid m_{mac})_{k_c}$
5. $B \rightarrow G$: m_{final}

Operation

OPERATION(device D, broker B, receiverGroup G)

1. D: $k_c := getChannelKey(G)$
2. D: $m_{data} := \text{data} \mid \text{timestamp}$
3. D: $m_{mac} := \text{HMAC256}(m_{data})$
4. $D \rightarrow B$: $m_{final} := enc\ (m_{data} \mid m_{mac})_{k_c}$
5. $B \rightarrow G$: m_{final}

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}

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}

Short Lived Macaroons

SHORT_LIVED(device D_{id}, key K_{id}, caveats C, channels S, Controller A)

1. $D_{id} \rightarrow A$: $M_{id} \parallel \text{timestamp} \parallel \text{HMAC256}(M_{id} \parallel \text{timestamp})_{k_{id}}$
2. A: $k_{id} := \text{CALC\_KEY}(M_{id})$
3. A: $k_2 := \text{HMAC256}\ (N_{new})_{k_i d}$
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 $i = 4,5,6...n$
7. A: $k_{s_j} := \text{HMAC256}\ (\text{timestamp})_{k_n}$
8. $A \rightarrow D_{id}$: $M_{id} \parallel N_{new} \parallel D_{id} \parallel C_i \parallel \text{timestamp} \parallel k_{s_j}$

Operation

OPERATION(device D, broker B, receiverGroup G)

1. D: $k_c := getChannelKey(G)$
2. D: $m_{data} :=$ data $\mid$ 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}

Revocation

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

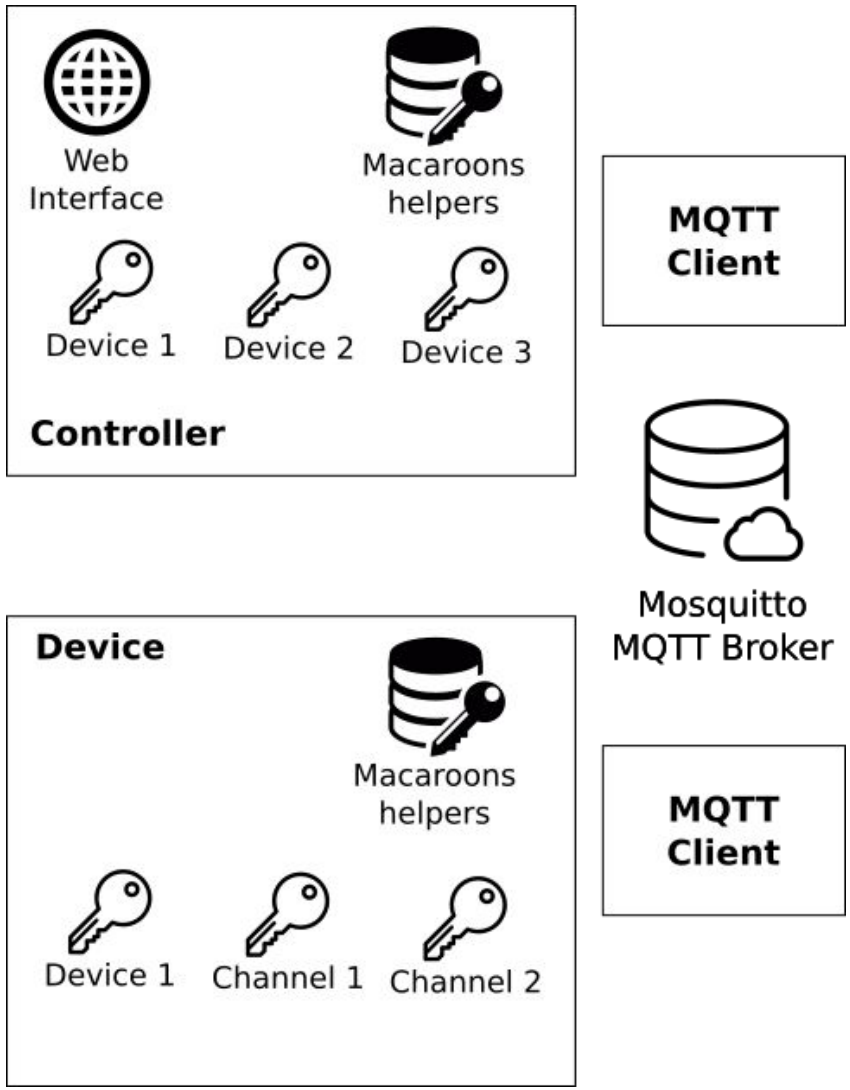
- When the channel keys to a particular device need to be revoked, the channel keys given to all the other devices with access to the channel are also revoked. Hence, there is a list maintained of what devices are connected to what channels.

- 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

- Introduction
- Proposed Architecture
- **Implementation**
- Results
- Conclusions

Implementation



- The device and the controller have been implemented in python.
- We made use of Mosquitto MQTT Broker and client off-the-shelf, and built our layer of security on top of it.

API Implementation

Key-management server

generate_macaroons(*device, manufacturer*)

revoke_macaroon(*macaroon*)

login_device(*core-identity macaroon, channel id*)

check_access(*message, channel, secret_key*)

Device

login(*core-identity macaroon*)

logout(*core-identity macaroon*)

encrypt_communication(*association macaroon, channel_id*)

Overview

- Introduction
- Proposed Architecture
- Implementation
- **Results**
- Conclusions

Results

We want to verify the following:

- Can we meet latency requirements of power grid protocols such as GOOSE?

- How much programmer effort is needed to lift an existing implementation to use Macaroons?

- Verify our choice of session-key management protocol — J-PAKE.

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

- We selected two python-based IoT applications and added our API calls to them.

Application	**Lines of code before**	**Lines of code after adding API calls**
Passive infrared sensor	75	102
Ultrasonic sensor	160	203

Verification

- We made use of Proverif 1.98pl1 to verify our cryptographic protocol.
- We implemented Macaroons using Proverif which accepts input in Pi-Calculus.
- 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

- Introduction
- Proposed Architecture
- Implementation
- Results
- **Conclusions**

Conclusions and Future work

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

Future work:

- We are exploring the option of adding our shared-secret protocol to the SSP21 protocol adopted by several utilities.
- We are in talks with various industry partners to incorporate our techniques
- Explore the use of homomorphic encryption as the encryption protocol to enable machine learning on the encrypted data

Thank you

- Toolkit will be available soon.

Prashant pa@cs.dartmouth.edu
Kartik palani2@illinois.edu
Sean sws@cs.dartmouth.edu