A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things

Hokeun Kim
University of California, Berkeley
hokeunkim@eecs.berkeley.edu

Eunsuk Kang
University of California, Berkeley
eunsuk@berkeley.edu

Edward A. Lee
University of California, Berkeley
eal@eecs.berkeley.edu

David Broman
KTH Royal Institute of Technology
dbro@kth.se

ABSTRACT

The challenges posed by the Internet of Things (IoT) render existing security measures ineffective against emerging networks and devices. These challenges include heterogeneity, operation in open environments, and scalability. In this paper, we propose *SST* (*Secure Swarm Toolkit*), an open-source toolkit for construction and deployment of an authorization service infrastructure for the IoT. The infrastructure uses distributed *local authorization entities*, which provide authorization services that can address heterogeneous security requirements and resource constraints in the IoT. The authorization services can be accessed by network entities through software interfaces provided by SST, called *accessors*. The accessors enable IoT developers to readily integrate their devices with authorization services without needing to manage cryptographic keys and operations. To rigorously show that SST provides necessary security guarantees, we have performed a formal security analysis using an automated verification tool. In addition, we demonstrate the scalability of our approach with a mathematical analysis, as well as experiments to evaluate security overhead of network entities under different security profiles supported by SST.

CCS CONCEPTS

•**Security and privacy →Authentication; Authorization;** *Formal security models; Security protocols;* Mobile and wireless security;

KEYWORDS

Internet of Things, Network security, Authorization, Authentication, Formal security analysis, Software infrastructure

ACM Reference format:
Hokeun Kim, Eunsuk Kang, Edward A. Lee, and David Broman. 2017. A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things. In *Proceedings of The 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation, Pittsburgh, PA USA, April 2017 (IoTDI 2017),* 12 pages.
DOI: http://dx.doi.org/10.1145/3054977.3054980

1 INTRODUCTION

As recognized by many researchers including [36], the main challenges in security of the Internet of Things (IoT) include heterogeneity, operation in an open environment, and scalability. These challenges render existing security measures ineffective against emerging IoT networks and devices.

The networked entities in the IoT are heterogeneous in terms of both security requirements and resource availability. For example, safety-critical systems such as electric power grid, autonomous vehicles, or drones will require the strongest possible guarantees for authorization and authentication. For mobile payment applications such as Apple Pay, high performance may be desirable in addition to confidentiality and authentication of transactions. However, for battery-powered devices such as temperature sensors, the lifetime and availability are considered just as important as data security. For some sensors, guaranteeing data integrity can be enough; it may not be necessary to keep sensor data confidential.

Insisting on maximum security for all devices in the IoT is not practical. To the best of our knowledge, there has not been a single integrated security solution for the IoT that supports heterogeneous requirements from safety-critical systems to sensor nodes. Existing, widely used security measures including SSL/TLS (Secure Socket Layer/Transport Layer Security), Kerberos, and various solutions for WSN (Wireless Sensor Network) and MANET (Mobile Ad hoc Network) are designed for homogeneous networks, and may not be directly applicable in a heterogeneous IoT setting. For instance, an approach purely based on SSL/TLS would be too prohibitive in an IoT network, due to the high computational requirements of public-key cryptography operations.

Another challenge arises due to risks involved with operating safety-critical components in open, untrusted, and even hostile environments. The threat model for existing, web-connected networks is reasonably well-understood, with a variety of mitigations developed to guard against potential attacks. Due to its open nature, however, an IoT network is susceptible to entirely new classes of attacks, which may include illegitimate access through mediums other than traditional networks (e.g., physical access, Bluetooth, radios). For instance, Ghena *et al.* [14] demonstrate an attack on a traffic controller on the streets of Ann Arbor, Michigan, including manipulation of actual traffic lights; this attack was made possible via direct radio communication with the traffic controller. Thus, the security solution for the IoT should provide ways to mitigate the potential effect of compromised entities in an open environment.

The final challenge is scalability of connected devices in the IoT. Many reports on scalability of the IoT, including one by Cisco [9], expect there will be tens of billions of connected devices by 2020, far exceeding the world population. Hence, the security solution must scale accordingly; in particular, the overhead of adding and removing devices to/from the security solution should be minimal.

To address these challenges, we propose *SST (Secure Swarm Toolkit)*, a toolkit for building an authorization service infrastructure for the IoT. The key features of our approach include:

- We propose an open-source implementation of a local authorization entity, *Auth*, that can be downloaded and deployed by anyone with moderate knowledge of computer security. Auth can be deployed on smart gateways including Intel's IoT gateways[1] and SwarmBox[2] from the TerraSwarm project,[3] to authenticate and authorize IoT devices and establish secure connections.
- Auth provides a variety of security alternatives depending on security requirements and resource availability. These alternatives range from strong and frequent authorization for safety-critical components to lightweight message integrity guarantees for resource-constrained sensor nodes.
- The proposed Auth has control over existing connections among the IoT devices, providing mechanisms to mitigate damage caused by compromised or subverted entities, by revoking credentials of compromised entities.
- Our infrastructure includes actor-based software components that are designed to help non-security expert developers design secure software for the IoT. This is achieved by enforcing a secure implementation and composition of software through actor-based programming semantics.
- To rigorously show that SST provides necessary security guarantees, we have performed a formal analysis on a model of our authorization protocol using an automated verification tool. As far as we know, this is the first security framework for the IoT that has been subjected to a rigorous, formal security analysis.

2 MOTIVATION

In this section, we discuss why current state-of-the-art network security solutions cannot address some of the IoT security challenges, and why an integrated security framework is needed. We summarize the main challenges as follows.

1. *Heterogeneity*: Diversity in security requirements and resource availability (including devices with *resource constraints* and/or *intermittent connectivity*).
2. *Open environment*: Increased risks of operation in an environment where adversaries have *physical* and/or *wireless* access to IoT devices.
3. *Scalability*: A large number of devices and a high volume of communication traffic including one-to-many traffic patterns such as broadcasting or publish-subscribe.

For our discussion, consider data collection in WSN (wireless sensor network) using a UAV (unmanned aerial vehicle), as shown in Figure 1. This example is motivated by Shih *et al.* [35]

[1]http://www.intel.com/content/www/us/en/internet-of-things/gateway-solutions.html
[2]https://swarmlab.eecs.berkeley.edu/projects/5378/swarmbox
[3]https://www.terraswarm.org/

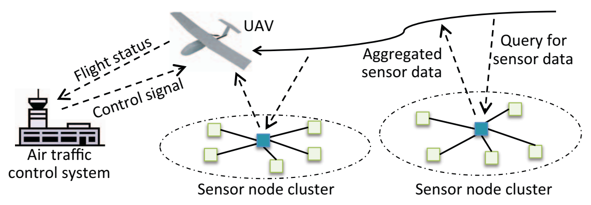


Figure 1: Motivational example for a security measure for the IoT inspired by [35]; WSN data collection using a UAV

To secure communication among the network nodes, one option is to apply a security solution purely based on SSL/TLS. This approach, however, will run into the following issues. First, resource-constrained sensor nodes (*1. Heterogeneity*) will suffer heavy energy consumption, due to the high computational requirements of public-key cryptographic operations as well as the transmission of large certificates. The air traffic control system and UAV are especially critical: If either of these two is compromised (*2. Open environment*), it will be difficult to prevent catastrophic consequences on the overall system. To mitigate the effect of compromised entities, SSL/TLS supports CRLs (Certificate Revocation List); however, CRLs must be updated frequently for all devices to revoke compromised certificates in a timely fashion. This will create scalability problems for resource-constrained devices. Moreover, SSL/TLS uses a server/client model based on one-to-one connections, which does not scale to broadcasting communication within sensor node clusters (*3. Scalability*).

The Kerberos authentication system [26], another popular security mechanism, employs the notion of a *ticket*, which includes an encrypted session key and a timestamp-based authenticator. The ticket is issued by the centralized Kerberos AS/TGS (Authentication Server / Ticket Granting Service), and the authenticator generated by a client proves the freshness of the authentication request. Kerberos provides centralized and timely control of authentication; thus, in the context of the example in Figure 1, it will provide means to limit damage even when the critical components have been compromised. However, this approach is not suitable for the UAV and sensor nodes with intermittent connectivity (*1. Heterogeneity*) because the authentication process requires direct communication with the AS/TGS. In addition, if the network contains a large number of sensor node clusters, the centralized AS/TGS will be a bottleneck for authentication (*3. Scalability*).

Lightweight security solutions for WSN or MANET [27] [10] will be suited to the requirements of the resource-constrained sensor nodes in this example. Keys with long lifetimes will mitigate the intermittent connectivity of the UAV. However, most of these solutions are not designed to work on an Internet scale, relying on local wireless communications and using local base stations for key distribution (*3. Scalability*). Furthermore, these lightweight solutions accept weaker security guarantees as a trade off for better energy efficiency (e.g., no confidentiality), and may not be suited to meeting the requirements of safety-critical components (*1. Heterogeneity*).

While the existing approaches provide a partial solution for some of these challenges, none of them offers a complete, integrated solution. In the rest of the paper, we describe our proposed approach, SST, which provides an *integrated*, *Internet-scale* authorization framework that can satisfy a diverse set of security and resource requirements found in an IoT network.

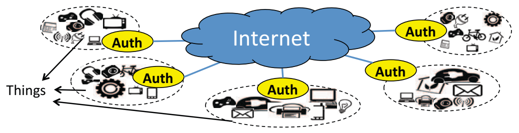


Figure 2: Network architecture of the SST infrastructure for the IoT based on local authorization entities, *Auths*

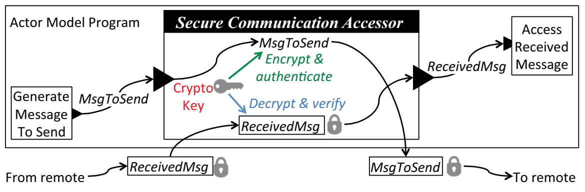


Figure 3: Software component for accessing authorization service, *secure communication accessor*

3 PROPOSED APPROACH OVERVIEW

In this section, we show an overview of SST, and discuss how it addresses the main challenges stated in Section 2.

3.1 Open-source Local Authorization Entity, Auth

Figure 2 illustrates the network architecture for the SST infrastructure based on local authorization entities, *Auths* [20]. An *Auth* is a program to be deployed on edge devices [13] including smart gateways, responsible for authentication/authorization of locally registered entities. An open-source implementation of Auth is available on our GitHub repository[4]. Compared to the conceptual prototype in [20], our new implementation is written in a memory-safe language (Java), supports connectionless protocols such as UDP, provides more security configurations, and uses a full-fledged database, SQLite, with all credentials encrypted.

3.2 Software Components for Accessing Authorization Service

We also propose actor-based software components for accessing the authorization service called *Secure Communication Accessors* shown in Figure 3. *Accessors*[5] [21] are actors [22] [16] specialized for accessing remote services to enable composition of heterogeneous devices and services in the IoT. The interaction of accessors is orchestrated by the actor model, allowing concurrent execution, segregation of private data and message passing. A secure communication accessor internally manages its credentials (cryptographic keys), and uses the keys for encryption, decryption, and message authentication. Thus, an accessor liberates application developers from the need to manage cryptographic keys and operations, while providing security guarantees for accessing remote services.

3.3 How the SST Infrastructure Works

Communications between the IoT entities in our infrastructure are protected by symmetric cryptographic keys, called *session keys*. These keys are generated by Auth and distributed only to entities

[4]https://github.com/iotauth/iotauth
[5]https://accessors.org/

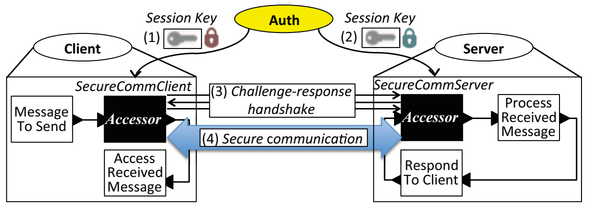


Figure 4: Process of building a secure connection between Client and Server

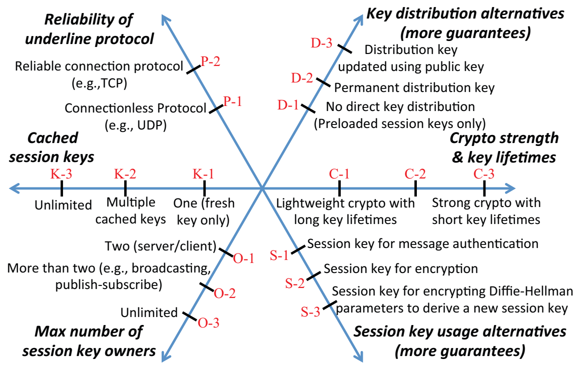

Reliability of underline protocol
Reliable connection protocol (e.g.,TCP)
P-2
Connectionless Protocol (e.g., UDP)
P-1
Key distribution alternatives (more guarantees)
D-3
Distribution key updated using public key
D-2
Permanent distribution key
D-1
No direct key distribution (Preloaded session keys only)
Cached session keys
Crypto strength & key lifetimes
K-3
K-2
K-1
C-1
C-2
C-3
Unlimited
Multiple cached keys
One (fresh key only)
Lightweight crypto with long key lifetimes
Strong crypto with short key lifetimes
Two (server/client)
O-1
More than two (e.g., broadcasting, publish-subscribe)
O-2
Unlimited
O-3
Max number of session key owners
S-1
Session key for message authentication
S-2
Session key for encryption
S-3
Session key for encrypting Diffie-Hellman parameters to derive a new session key
Session key usage alternatives (more guarantees)

Figure 5: Security configuration space provided by Auth

that are *authorized for access/communication*. For secure delivery of session keys, Auth and an entity share another symmetric key called *distribution key*. Figure 4 illustrates the process of establishing a secure connection between *Client* and *Server*. Both Client and Server are registered with Auth, and employ *SecureCommClient* and *SecureCommServer* accessors, respectively, for secure communication with Auth and with each other. Details on accessors are found in Section 4.6.

To build a secure connection, Client and Server must obtain a *session key* from Auth. In step (1) of Figure 4, Client receives a session key from Auth, encrypted with the *distribution key between Client and Auth*. Through step (2), Server receives the same session key encrypted with the *distribution key between Server and Auth*. Details of (1) and (2) are described in Section 4.3. To prove the ownership of the session key to each other, Client and Server perform a challenge-response handshake using nonces (random values) in step (3). After step (3) succeeds, they can start a secure communication as in step (4). Details of (3) and (4) are explained in Section 4.4.

3.4 How the Proposed Approach Addresses Challenges

3.4.1 Heterogeneity. SST supports various security configurations, which can be used to achieve tradeoffs between security guarantees and resource usage. Figure 5 depicts the space of configuration options. This space includes multiple alternatives for key distribution (D-1, D-2, D-3), cryptography strength and key lifetimes (C-1, C-2, C-3), session key usage (S-1, S-2, S-3), the number of session key owners (O-1, O-2, O-3), cached session keys (K-1, K-2, K-3), and reliability of the underlying protocol (P-1, P-2). An example of cryptography strengths is AES ciphers with different key sizes.

Table 1: Example security configuration profiles

Config. \ Profile	High-risk safety-critical	Resource-constrained	Sensitive information	Broad-casting
Key distribution	D-3	D-1	D-2	D-2
Crypto strength	C-3	C-1	C-2	C-2
Session key use	S-2	S-1	S-3	S-1
Max key owners	O-1	O-2	O-1	O-3
Cached keys	K-1	K-3	K-2	K-2
Protocol	P-2	P-1	P-2	P-1

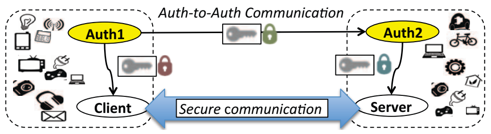


Figure 6: Operation example of communication between Client and Server registered with two different Auths, Auth1 and Auth2, respectively

Table 1 shows sample profiles using different configuration alternatives. For a *safety-critical* entity in a *high-risk* environment, we enforce short-term keys to limit the damage when it is compromised. *Resource-constrained* devices are allowed to use cached keys and connectionless protocols to cope with intermittent connectivity. Entities dealing with *sensitive information* can derive a new key for communication by exchanging Diffie-Hellman parameters using the session key (details in Section 4.4). For *broadcasting* devices, we allow unlimited number of devices sharing the same session key.

3.4.2 Open environment. Auth is a central point of authorization, keeping track of credentials and authorization requests. Thus, Auth can revoke credentials of compromised entities to limit their potential damage. This is important for devices operating under a high risk of being compromised due to the physical or wireless access by potential adversaries. For attack detection, various IDSs (Intrusion Detection Systems) [7] can be deployed in combination with Auth, leveraging the fact that all traffic relevant to authorization is directed through Auth.

3.4.3 Scalability. The scalability problem is twofold: (1) how to handle a large number of entities and (2) how to handle increased data traffic. Our approach addresses the first problem by allowing multiple Auths to be deployed in a network. To show how this works, we use a simple operation example described in Figure 6, where Client and Server are registered with two different Auths, Auth1 and Auth2, respectively. For Client and Server, the overhead for establishing a secure connection is no more than it would be with a single Auth, since they only need to communicate with their own Auth at all times. Auth1 and Auth2 still need to communicate with each other to deliver the same session key to Client and Server, but this exchange needs to occur only once before Client and Server can communicate without additional overhead. An analysis in Section 6 shows our approach's scalability.

To handle increased data traffic, our infrastructure supports shared session keys for one-to-many communication patterns. Figure 7 describes an example of secure publish-subscribe communication in SST. For Publisher and Subscriber programs, we use corresponding accessors, *SecurePublisher* and *SecureSubscriber*. Publisher and two Subscribers, Subscriber1 and Subscriber2, are first

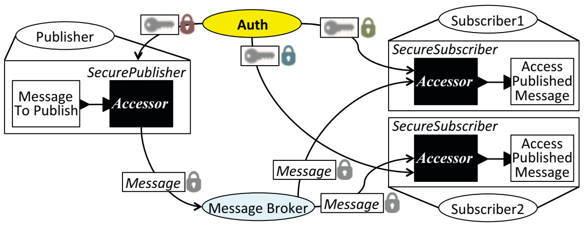
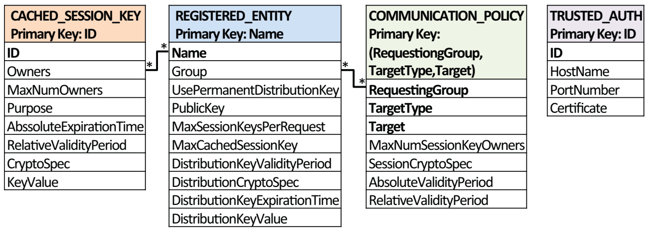


Figure 7: Process of scalable key sharing for publish-subscribe communication



CACHED_SESSION_KEY Primary Key: ID
ID
Owners
MaxNumOwners
Purpose
AbssoluteExpirationTime
RelativeValidityPeriod
CryptoSpec
KeyValue

REGISTERED_ENTITY Primary Key: Name
Name
Group
UsePermanentDistributionKey
PublicKey
MaxSessionKeysPerRequest
MaxCachedSessionKey
DistributionKeyValidityPeriod
DistributionCryptoSpec
DistributionKeyExpirationTime
DistributionKeyValue

COMMUNICATION_POLICY Primary Key: (RequestingGroup, TargetType,Target)
RequestingGroup
TargetType
Target
MaxNumSessionKeyOwners
SessionCryptoSpec
AbsoluteValidityPeriod
RelativeValidityPeriod

TRUSTED_AUTH Primary Key: ID
ID
HostName
PortNumber
Certificate

Figure 8: Auth database table schema (* for many-to-many relationship)

registered with Auth. They are also connected with a possibly untrusted *message broker* which forwards published messages to subscribers. Publisher and two Subscribers are authorized by Auth, and each receives the same session key to be used for published messages. Publisher only needs to encrypt the message and send it once even when the number of Subscribers increases. This process is further explained in Section 4.4 and evaluated in Section 7.2.

4 DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of SST, including the local authorization entity, communication protocols, and accessors.

4.1 Local Authorization Entity, Auth

Auth's role is to authenticate and authorize locally registered entities. It also interacts with other Auths to allow communication between entities on different networks. Auth makes its authorization decisions based on a database of access policies and configurations, as shown in Figure 8. The database includes:

- *Registered entity table*: Stores information about entities registered with the Auth, including its credentials (cryptographic keys) and the configuration related to key distribution (details in Section 4.3).
- *Communication policy table*: Stores access policies between entities (for example, which entity can talk to which entity, what kind of cryptography should be used, and how long the cryptographic keys should be valid).
- *Cached session key table*: Stores cached session keys, which Auth allows for entities with limited connectivity. Each session key is associated with its owners and the max possible number of owners (two for server/client, and three or more for one-to-many communication).
- *Trusted Auth table*: Stores information and credentials for other trusted Auths, including each Auth's unique *ID*, network address, port, and certificate.

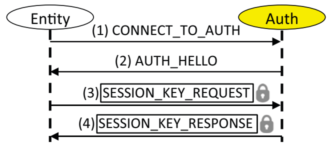


Figure 9: Steps for $Auth - Entity$ communication for session key distribution; a padlock next to a message indicates that the message is encrypted and/or authenticated

4.2 Entity Registration

Each entity must be *registered* with Auth in order to access the authorization infrastructure. The main purpose of this registration process is to set up credentials between Auth and an entity. An entity's credentials may be generated[6] during the registration or shipped by the manufacturer[7] with the entity.

If an entity is capable of performing public-key cryptography operations to update[8] its *distribution key* (explained in Section 3.3), the entity and Auth must exchange their public keys. If an entity cannot perform public-key cryptography, then the entity and Auth can set up a *permanent distribution key*. In addition to setting up the credentials, the entity's unique name, security configurations, and communication policies are also set up during entity registration.

If a severely resource-constrained entity cannot directly connect to Auth or perform symmetric-key decryption, the entity will not be able to obtain any session key from Auth. However, even such entity can be part of the infrastructure if it has preloaded session keys. In this case, the entity's preloaded keys are stored in Auth during entity registration.

4.3 Auth – Entity Communication

Auth authorizes entities to communicate with each other by distributing a session key shared by entities. Figure 9 shows the authorization process, which starts with step (1) CONNECT_TO_AUTH. If an entity uses TCP, step (1) is TCP connection establishment with Auth. If the entity uses a connectionless protocol such as UDP, step (1) is entity's ENTITY_HELLO message, which simply triggers step (2). After step (1), Auth sends (2) AUTH_HELLO message, which includes Auth's ID and its fresh random nonce, N_{Auth}.

Step (3) SESSION_KEY_REQUEST must include N_{Auth} and N_{Entity} (entity's random nonce), the name of the requesting entity, the purpose of the request (e.g., for communication with an entity in a certain group or a certain publish-subscribe topic), and the number of keys requested. N_{Entity} is to ensure step (4) is not replayed. (3) must be at least authenticated, and can be optionally encrypted as well for confidentiality. There are two cases of (3) depending on whether the distribution key is to be updated or not:

- If the entity already has a valid distribution key, the message authentication (and optionally, encryption) must be done using the distribution key.
- If the entity does not have a valid distribution key or wants to update it using public-key cryptography, (3) must be authenticated with the entity's private key, and optionally encrypted with Auth's public key.

Given that the message in (3) is valid, Auth consults its communication policy table and determines whether the requesting entity should be authorized. If so, it generates new session keys or fetches existing, cached keys to be returned to the requesting entity; in addition, if necessary, it also generates a new distribution key.

In (4), Auth sends back SESSION_KEY_RESPONSE, which includes N_{Entity}, new session keys, a security specification for the session keys, as well as a new distribution key (if requested in (3)). This message must be authenticated and encrypted with the distribution key; when a new distribution key is sent, it must be encrypted with the entity's public key and signed with Auth's private key. After receiving (4), the entity decrypts it to check the validity of N_{Entity} and Auth's MAC (Message Authentication Code) and/or signature. If the message is valid, the entity stores the received session keys (and if applicable, the updated distribution key).

To support UDP, a connectionless protocol, Auth maintains its responses until a specified timeout so that it can respond again in case any message is lost. If Auth detects anything wrong or suspicious such as use of an expired distribution key, it sends an AUTH_ALERT message to the entity.

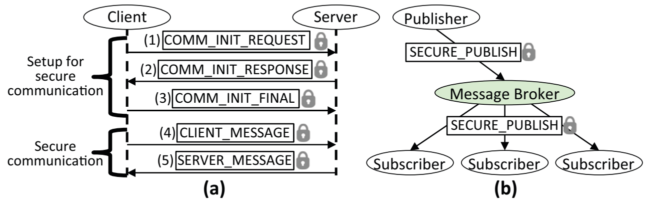


Figure 10: Process of secure communication for (a) Server-client (b) Publish-subscribe

4.4 Entity – Entity Communication

After an entity receives a valid session key from Auth, it can start secure communication with other entities. The secure communication means messages are encrypted and/or authenticated. Figure 10 describes two ways of secure communication provided by the SST infrastructure.

Figure 10 (a) shows a server-client type of secure communication. To first confirm the validity of each other's session key, Server and Client carry out a simple challenge-response by performing cryptographic operations on random nonces in steps (1) to (3). This process is similar to PSK Key Exchange Algorithm of PSK cipher suites for TLS [11]. For identification of the session key, we use its unique identifier, *session key ID*. As part of its value, the session key ID also includes the ID of Auth who generated it, and thus it can be used to identify the generator. The session key ID is analogous to the PSK identity of PSK Key Exchange Algorithm of TLS.

Optionally, we can configure the session key to be used to authenticate an ephemeral Diffie-Hellman key exchange to derive a new session key in steps (2) to (3). This process is similar to DHE_PSK Key Exchange Algorithm of PSK TLS [11], which provides additional protection such as Perfect Forward Secrecy (PFS).

[6] Generation of credentials (cryptographic keys) can be done using tools such as OpenSSL command line tools.
[7] This is becoming more common for IoT devices that need credentials for cryptography operations.
[8] The distribution key is important because it is used for encrypting session keys. If a distribution key is compromised, session keys encrypted with the distribution key can also be compromised. Updating distribution keys can mitigate the effect of a compromised distribution key.

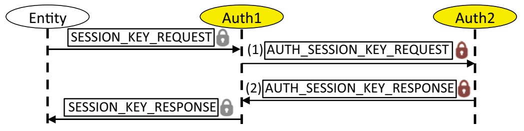


Figure 11: Steps for *Auth* – *Auth* communication

Having successfully performed the handshake, Server and Client can start a secure communication protected by the session key. Each CLIENT_MESSAGE or SERVER_MESSAGE includes a read/write sequence number that increases per message. These sequence numbers are used to detect whether a certain message is missing or replayed by attackers. The sequence numbers are similar to those in the application data record protocol of SSL/TLS. Our approach supports both TCP and UDP for this server-client communication.

Figure 10 (b) shows a publish-subscribe style of communication supported by SST. Publisher and Subscribers have the same session key to be used for messages. Publisher encrypts and/or authenticates a SECURE_PUBLISH message, attaches the session key ID in plaintext (so that Subscribers can identify which session key is used for the message), and sends it to the *Message Broker*, which in turn forwards the message to Subscribers. Only those Subscribers with a valid session key can decrypt and/or check authenticity of published messages. To mitigate risks where a compromised subscriber illegally publishes messages, SST supports delayed disclosure of keys using a technique similar to that of the TESLA protocol [28].

4.5 Auth – Auth Communication

Auth communicates with other Auths to request a session key that was generated by the other Auths. Trusted Auths are connected to each other over HTTPS on top of SSL/TLS, using POST request/response for communication. Figure 11 illustrates the steps of Auth – Auth communication. Entity, which is registered with Auth1, requests a session key that was generated by Auth2. This case can happen when Entity wants to set up a secure communication with another entity registered with Auth2. Auth1 receives SESSION_KEY_REQUEST that specifies the session key's ID. As explained in Section 4.4, the session key ID includes the generator's ID, in this case, Auth2's ID. From this ID of Auth2, Auth1 discovers that the requested session key was generated by Auth2 and sends (1) AUTH_SESSION_KEY_REQUEST which includes Entity's information, the purpose of the request, and the session key's ID. Auth2 responds to Auth1 if the Auth1's request is eligible with (2) AUTH_SESSION_KEY_RESPONSE which includes the requested session key and cryptography specification of the session key. Upon receiving (2), Auth1 responds to Entity.

4.6 Secure Communication Accessors

Accessors use a JavaScript file to specify interactions (inputs, outputs, and parameters) and functionality implementations (reaction to inputs and/or production of outputs). Many accessors use asynchronous atomic callbacks (AAC), for requesting remote services and handling following responses asynchronously and atomically.

For constructing a secure swarm applications, we provide four secure communication accessors for accessing authorization services, *SecureCommClient*, *SecureCommServer*, *SecurePublisher*, and

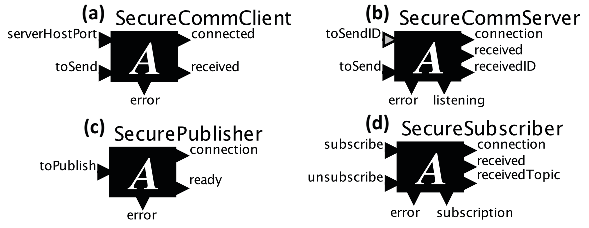


Figure 12: Secure communication accessors of SST

SecureSubscriber as shown in Figure 12. In common, all these accessors manage a distribution key and cached session keys internally with parameters for security configurations and credentials. The proposed accessors provide standardized interfaces over different underlying implementations. Incoming and outgoing triangles on each accessor indicate input and output ports of the accessor, respectively. If any security condition is violated, these accessors generate an output on their *error* output port. Detailed documents are available on our accessor library[9] under *net* group.

SecureCommClient (Figure 12 (a)) establishes a secure connection with *SecureCommServer* (Figure 12 (b)) when there is an input on *serverHostPort* which specifies the destination server information. Both *SecureCommClient* and *SecureCommServer* generate an output *connection* when a new secure connection is established. Both *SecureCommClient* and *SecureCommServer* send a secure message to the counterpart when there is an input on *toSend* and generate an output on *received* when a secure message arrives. *toSendID* and *receivedID* of *SecureCommServer* are used to specify a specific client since there can be multiple clients connected to the same server.

SecurePublisher and *SecureSubscriber* use a MQTT [5] message broker for publishing and subscribing secure messages. When they are connected to the broker, they generate an output on *connected*. If *SecurePublisher* obtained the session key and is ready to publish, it generates an output on *ready*. When there is an input on *toPublish*, *SecurePublisher* sends a secure publish message for the topic specified as a parameter. *SecureSubscriber* can *subscribe* and *unsubscribe* on a specific topic and an output on *subscription* indicates the subscription status. When a secure publish message arrives on the topic, outputs are generated on *received* and *receivedTopic* ports.

5 SECURITY ANALYSIS

In this section, we present a security analysis of our proposed authorization infrastructure. To make the analysis rigorous, we constructed a formal model of the Auth protocol, and applied an automated verification tool to exhaustively explore all possible behaviors of the model for vulnerabilities.

5.1 Security Properties and Threat Model

The purpose of Auth is to provide a secure channel for trusted entities on an Auth network to communicate to each other, even in the presence of possibly malicious entities. To be more specific, in our analysis, we wish to establish the following two security properties: (1) Each message sent from an entity should be accessible to its intended recipient(s) (*confidentiality* of a message), and (2) A message delivered to an entity has the same content as it is sent

[9]https://accessors.org/library/

by its source entity (*data integrity* and *authenticity* of a message). In our current threat model, we do not consider security guarantees against availability attacks, such as denial of service (DoS) or depletion of energy resources; this remains part of our future work.

We assume the presence of an active network attacker, who is able to eavesdrop on communication among network nodes, and potentially modify or replay any messages. We further allow the attacker to take on the role of an entity itself, interacting with Auths or other entities on the network. The attacker may have access to public keys of Auths and entities, their IDs and names, and impersonate another entity while interacting with an Auth. However, we assume that the attacker is not capable of impersonating Auth.

5.2 Formal Analysis

5.2.1 Modeling Auth in Alloy. Alloy is a modeling language based on a first-order relational logic [18]. It has been used to analyze a wide range of systems, including web applications [1], network configurations [24], and security policies [25]. Alloy is a particularly suitable choice for specifying and analyzing IoT networks, thanks to (1) its expressiveness, which allows modeling of a dynamic network where its topology evolves as nodes enter and exit, (2) its type system (with a flexible subtyping mechanism), which allows modeling of heterogeneous components that share common characteristics, and (3) its analysis engine, which can perform simulation and verification of a model against various properties, such as safety, security, and functional correctness.

Figure 13 shows a snippet of a model of our authorization infrastructure in Alloy; due to limited space, a simplified version is shown here[10]. The model[11] begins with declarations of datatypes that will be used for communication in an Auth network (lines 2-8). In particular, two types of `Key` are declared: `SymKey`, which represents symmetric keys that will be used as distribution and session keys, and `AsymKey`, each of which is associated with a corresponding asymmetric key (`pair`) that can be used for public-key cryptographic operations. A constraint is introduced to ensure that each asymmetric key is assigned a unique pair (line 8)[12].

The set of Auth and entities in the world are collectively referred as `Node` in our model. Each node is assigned a pair of public and private keys that can be used for secure communication with other nodes in the network (line 10). Every `Auth` object is associated with an ID, and has access to a set of public keys that it uses to encrypt messages sent to entities (line 15). Similarly, each `Entity` is assigned a name, and knows the public keys of Auths that it communicates to (line 24). For our analysis, we will assume some subset of the entities to be malicious (line 29)[13].

Modeling behavior. To reason about the dynamic behavior of a network, we use a style of modeling where an execution is modeled as a sequence of time steps, and each mutable object is associated

```
1  sig Time {} // Totally ordered time steps
2  /* Datatypes (keys, payloads, names, IDs) */
3  sig Payload // data to be sent between entities
4  sig Name, ID {} // entity names and Auth IDs
5  abstract sig Key {}
6  sig SymKey extends Key {} // symmetric keys
7  sig AsymKey extends Key { pair : AsymKey }
8  fact { no disj k1, k2: AsymKey | k1.pair = k2.pair }
9  /* Auth and entities */
10 abstract sig Node {  publicKey, privateKey: AsymKey }{
11   publicKey.pair = privateKey
12 }
13 sig Auth extends Node {
14   id: ID,
15   entityPublicKeys: Name -> AsymKey,
16   // session keys allocated so far
17   sessionKeys: SymKey -> Time,
18   // owners associated with session keys
19   owners: sessionKeys -> Name -> Time
20 }
21 sig Entity extends Node {
22   name: Name,
23   payloads: set Payload,
24   authPublicKeys: ID -> AsymKey,
25   // session keys obtained from Auth
26   sessionKeys: Name -> SymKey -> Time
27 }
28 // Some of the entities may be malicious
29 sig Attacker in Entity {}
30 /* Messages */
31 abstract sig Message { sender,receiver: Node, t: Time }
32 sig SESSION_KEY_REQUEST extends Message {
33   entity, target: Name, id: ID
34 }{
35   encryptWith[entity+target,sender.authPublicKeys[id]]
36   signWith[entity+target,sender.privateKey]
37   some newKey: SymKey |
38     insert[receiver.sessionKeys,newKey,t] and
39     insert[receiver.owners,newKey->entity,t]
40 }
41 sig SESSION_KEY_RESP extends Message {
42   distrKey, sessionKey: SymKey,
43   req: SESSION_KEY_REQUEST
44 }{
45   encryptWith[sessionKey,distrKey]
46   encryptWith[distrKey,sender.entityPublicKeys[req.entity]]
47   insert[receiver.sessionKeys,req.target->sessionKey,t]
48 }
49 sig SECURE_MESSAGE extends Message {
50   payload: Payload, target: Name
51 }{
52   encryptWith[payload,(sender.sessionKeys.t)[target]]
53 }
54 /* Security property */
55 check Confidentiality {
56    no t: Time, e: Entity - Attacker, a: Attacker |
57      some s : e.payloads | accesses[a,s,t]
58 } for 5 but 10 Time, 10 Message
```

Figure 13: A snippet of an Alloy model of the Auth protocol.

with a state at each step [18]. In this model, we declare the signature `Time` to represent the set of time steps, and attach it as the last column of every relation that stores mutable records. For example, consider the field `sessionKeys` (line 17), which is a ternary relation of type `Auth` $\times$ `SymKey` $\times$ `Time`; tuple (a, k, t) belonging to `sessionKeys` means that k is one of the session keys that Auth a has allocated for its entities at time t.

Communication between two nodes is modeled using a set of objects called `Message`. Each message is associated with a sender and a receiver, and a time step (`t`) at which the message is sent

[10] The full model is available at https://github.com/iotauth/security_analysis.

[11] The Alloy keyword `sig` introduces a signature, which defines a set of elements in the universe. A signature may contain one or more *fields*, each introducing a relation that maps the elements of the signature to the field expression; for example, field `name` in `Entity` is a binary relation that maps each `Entity` object to its name (line 22). The keyword `extends` creates a subtyping relationship between two signatures; an `abstract` signature has no elements except those belonging to its extensions.

[12] A `fact` is a constraint that must hold over every instance of the model.

[13] The keyword `in` imposes a subset relationship between two sets.

and delivered[14]. The sender and receiver behavior associated with each type of message is defined using *signature constraints*[15]. For instance, consider SESSION_KEY_REQUEST, which corresponds to a set of messages that an entity sends to an Auth (with id) in order to request a session key for communicating to another entity (identified by target); here, we only depict the case in which the sender does not possess a distribution key. The definition of SESSION_KEY_REQUEST requires that both the names of the sender and target entities are encrypted using the receiving Auth's public key, and then signed with the sender's private key (lines 35-36). When Auth receives the message, it allocates a new symmetric key (newKey) and inserts it into its current list of session keys and their owners (lines 37-39)[16].

In response, Auth sends back a SESSION_KEY_RESP message with a distribution key and the newly generated session key. It encrypts the session key with the distribution key (line 45), which is, in turn, encrypted with the public key of the receiving entity to ensure its secrecy (lines 46). Having obtained the session key, the entity is now able to send a secure message (SECURE_MESSAGE) to its target entity by encrypting the payload using the key (line 52).

5.2.2 Verification Procedure. The Alloy Analyzer is a tool that can be used to execute a model or automatically verify it against a desired property. The tool is capable of *exhaustive, bounded* verification; that is, it will explore all possible behaviors of the modeled system, up to certain upper bounds on the length of an execution trace and the number of system and data components. Verifying an infinite system is an undecidable problem in general [4], and so to render the analysis fully automatic, the tool makes a trade-off by asking the user to specify the bounds for the input model.

One of the security properties of Auth that we verified using the tool is shown in Figure 13 (lines 55-58). This confidential property says that there should never be a time (t) at which an attacker (a) is able to access one of the payloads (s) that belongs to a non-attacker entity (e)[17]. When executed with a check command, the analyzer will attempt to generate a *counterexample* (if it exists within the bounds) that demonstrates how the model violates the property. In this case, such a counterexample would show an execution where there is at least one time step t at which the attacker receives a message containing a payload (s) of the victim entity (e).

5.2.3 Results. We analyzed our model of Auth against the properties stated in Section 5.1: confidentiality, data integrity and authenticity of each message. We specified an upper bound of 5 for the size of each signature (at most 5 unique Node objects, etc.), except 10 for the number of time steps and messages, which allowed the analyzer to explore all possible traces up to length 10.

Figure 14 shows the average times taken by the analyzer to generate a counterexample for different trace lengths[18]. Overall, the analysis time shows an exponential growth over the maximum length of a trace explored by the analyzer. This trend is not surprising, since as the maximum length of a trace is incremented, the number of all possible traces also increases exponentially. For example, consider the three types of messages in Figure 13; since every message contains multiple parameters, each of which takes on one of five possible values (given the general upper bound of 5 on each signature), the number of messages that may be sent at a particular time step is $(5^3 + 5^2 + 5^2) = 175$. Thus, given a maximum trace length of 10, the number of possible combinations of messages (i.e., number of traces potentially explored by the analyzer) is approximately $175^{10} \approx 2.69 * 10^{22}$.

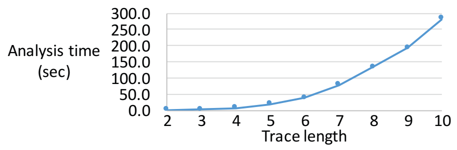


Figure 14: Verification times on the Auth model.

The analyzer generated 17 counterexample traces during our analysis. We examined each of these traces and incrementally fixed the model to ensure that the attack scenario captured by the trace would no longer be allowed by the model. These traces did not point to a fundamental security flaw in the design of Auth itself, but were due to missing *security assumptions* in our model. An assumption is a condition that must hold in order for a protocol to satisfy its security properties. For example, an assumption may describe the initial knowledge of an attacker (e.g., it does not have access to an entity's private key), configuration requirements (each trusted entity and Auth pair are configured with each other's correct public key), or what each protocol agent is not allowed to do (Auth never reuses a distribution key when it responds to a new entity).

Our initial model of Auth omitted many of these assumptions, since they were implicit in our original, informal protocol description. These counterexamples nevertheless demonstrate possible attacks on implementations that do not satisfy one of these assumptions. The analysis process improved our understanding of Auth, and helped us come up with a precise specification that explicitly lists security assumptions that every Auth implementation must satisfy. We believe this is especially important, since many implementations of cryptographic protocols have suffered from attacks due to missing or violated assumptions [2].

5.3 Limitations

Our current threat model assumes that all Auths are trusted and cannot be controlled by an attacker. Possible consequences of a compromised Auth are significant: The attacker may be able to manipulate messages from and to entities, undermining the security of the local Auth network and possibly its neighbors. We plan to extend SST with a detection and recovery mechanism in the presence of a compromised Auth.

Due to the bounded nature of the verification technique used, it is possible that our analysis may have missed one or more attacks on Auth. In our experience with Alloy, however, often a small number of messages are sufficient to demonstrate a flaw in a system [3]. For example, the smallest counterexample that we found required

[14]For our analysis, we assume that messages are delivered without delays.
[15]A *signature constraint*, specified in the appendix to field declarations, is a statement that is imposed on every member of the signature.
[16]encryptWith[d,k] and signedWith[d,k] are custom-defined predicates that mean data d is encrypted/signed with key k, respectively. insert[x,r,t] means tuple x is added to mutable relation r at time t.
[17]The keyword no is a quantifier meaning $\neg\forall$; the custom-defined predicate accesses[e,d,t] means that entity e can access data d at time t.
[18]The analysis was performed on a Mac OS X 10.11 machine with 2.7 GHz Intel Core i5 and 8 GB of RAM.

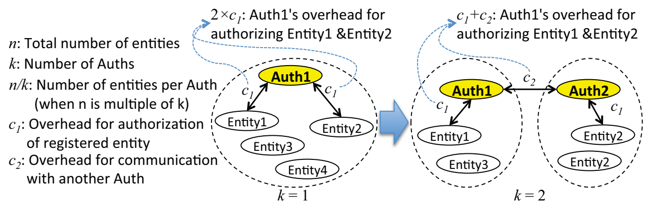

n: Total number of entities
k: Number of Auths
n/k: Number of entities per Auth (when n is multiple of k)
c_1: Overhead for authorization of registered entity
c_2: Overhead for communication with another Auth
$2\times c_1$: Auth1's overhead for authorizing Entity1 &Entity2
c_1+c_2: Auth1's overhead for authorizing Entity1 &Entity2
Auth1
Auth2
Entity1
Entity2
Entity3
Entity4
$k = 1$
$k = 2$

Figure 15: Division of entities into two groups registered with separate Auths

only 4 messages to demonstrate a possible attack on Auth, and the longest one involved 8 messages. To further increase the confidence in its results, one may repeat the analysis with increasingly larger bounds. We believe that this is an acceptable trade-off to achieve automation and an ability to generate counterexamples, which greatly aided our understanding of Auth.

6 SCALABILITY ANALYSIS

In this section, we provide a mathematical analysis of the scalability of the SST infrastructure. Figure 15 shows an example where entities registered with one Auth are divided into entity groups with two Auths. Let n be the total number of entities, k be the number of Auths, and n/k be the number of entities registered with each Auth (given n is divisible by k). Let c_1 be Auth's overhead for session key request and response for its entity, and let c_2 be each Auth's overhead for session key request and response between two Auths. Although actual c_1 and c_2 vary depending on underline cryptography and communication configurations, we assume that they are the worst-case upper bounds for all possible configurations.

We assume that each entity sets up a server/client-style secure communication with a constant number of entities (m). For each connection, Auth needs to authorize a pair of entities involved. When $k = 1$, there are $m \times n$ secure connections; thus, the total overhead for Auth is

$$t_1 = mn \times 2c_1 \tag{1}$$

Now consider the case where $k \geq 2$. Among m entities that some entity e wishes to communicate to, let p ($0 \leq p \leq 1$) be the proportion of the entities registered with the same Auth as e is. Then, pm entities are registered with the same Auth as e is, while $(1 - p)m$ entities are registered with other Auths. Since each Auth has $\frac{n}{k}$ registered entities, the overhead for authorizing connections between entities in a single Auth is

$$pm \times \frac{n}{k} \times 2c_1 \tag{2}$$

In addition, there is overhead for authorization of the entities that communicate with entities outside the Auth. Since this overhead for each Auth is $(c_1 + c_2)$ as in Figure 15, the resulting overhead is

$$(1 - p)m \times \frac{n}{k} \times (c_1 + c_2) \tag{3}$$

Summing (2) and (3), for $k \geq 2$, the total overhead for an individual Auth is

$$t_k = pm \times \frac{n}{k} \times 2c_1 + (1 - p)m \times \frac{n}{k} \times (c_1 + c_2) \tag{4}$$

Now, let $r = \frac{n}{k}$ be the ratio of n and k. The ratio r can also be considered as the number of entities per Auth. Even when n increases, we can keep r constant by having linearly more Auths.

Table 2: Energy cost model used in [20] (energy numbers from [32] and [12])

Operation	Energy cost
RSA-2048	91.02 mJ per encrypt/sign operation
	4.41 mJ per decrypt/verify operation
AES-128-CBC	0.19 μJ per byte encrypted/decrypted
SHA-256	0.14 μJ per byte digested
Send packet	454 μJ + 1.9 μJ $\times$ packet size (bytes)
Receive packet	356 μJ + 0.5 μJ $\times$ packet size (bytes)

Then, equation (4) becomes

$$t_k = pm \times r \times 2c_1 + (1 - p)m \times r \times (c_1 + c_2) \tag{5}$$

Then, we can make t_k (the overhead of each Auth) independent of n, assuming that we add more Auths linearly to the number of entities. Hence, in theory, our infrastructure should be scalable for an increasing number of entities.

7 EXPERIMENTS AND RESULTS

To evaluate our approach, we demonstrate a range of tradeoffs between security guarantees and energy consumption for different configurations provided by SST. We performed experiments for both client-server and one-to-many styles of communication. During our experiments, we measured the overhead of entities in establishing secure connections and sending secure messages. For each experiment, we tested different security configurations and varying numbers of communicating entities. In addition, we compared our approach against SSL/TLS as a reference.

The entities for our experiments were built using secure communication accessors. To run these entities as a composition of accessors, a special type of application called an *accessor host* is needed; we used a *Node.js host*[19], which is based on Node.js [38], a JavaScript runtime platform. Java 1.8 was used to run Auth. In addition, Auth and entities were deployed on a single host with different port numbers.

We measured (1) computational security overhead by logging cryptographic operations[20] and (2) communicational overhead by capturing network packets using a packet sniffing tool[21]. For cryptography operations, we used RSA-2048 for public-key cryptography, AES-128-CBC for bulk encryption, and SHA-256 for message authentication. This specification is the same as one of the TLS 1.2 cipher suites, TLS_RSA_WITH_AES_128_CBC_SHA256, which is the cipher suite we used for our experiments of TLS. We converted the measurements into energy consumption to estimate overall security overhead. For this conversion, we used the energy cost model used in [20] (shown in Table 2).

7.1 Server-Client Communication

We describe our findings on the security overhead in establishing secure connections for the client-server communication architecture. For this experiment, we varied three configuration parameters: (1) the maximum number of allowed cached session keys, (2) the underlying network protocol, and (3) distribution key management

[19] https://accessors.org/hosts/node/
[20] This is done by modifying OpenSSL library (version 1.0.2k) included in Node.js (version 7.6.0).
[21] WireShark, https://www.wireshark.org/

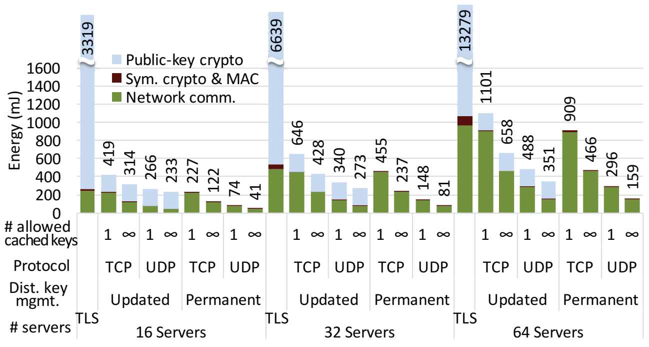


Figure 16: Estimated energy consumption of a client for setting up and closing secure connections with 16, 32, and 64 servers (Note that the energy consumption results for TLS are cut off due to the space limitation.)

alternatives. For each entity, either only one cached key aw allowed or there was no limit on the number of cached keys. For the network protocol, an entity was allowed to use either TCP or UDP. For distribution key management, an entity was given either a distribution key to be updated using public-key cryptography or a permanent distribution key. If an entity was a distribution key to be updated, we assumed that the entity did not have a distribution key in its initial deployment.

Figure 16 shows the estimated energy consumption of a client for establishing/closing secure connections with 16, 32, and 64 servers under different configurations. Figure 17 shows the results for a server with 16, 32, and 64 clients. We can see that SST uses far less energy for secure connections than TLS for both the client and server. This is mainly because of the overhead associated with public-key cryptography: It rapidly increases with the number of communicating entities in TLS, but remains constant in SST, which employs public-key cryptography only for communication with Auth. However, note that this does not necessarily mean our approach is always more desirable than TLS, since the latter provides different types of security guarantees.

From the results, we can also observe that the estimated energy consumption of SST ranges approximately an order of magnitude under different configurations. An entity can save energy on public-key cryptography by trading off updatability of the distribution key. An entity can save energy on network communication by using cached keys and/or UDP. TLS consumes more energy on SHA-256 MAC than SST does, since it needs to verify client and server certificates, although there is only negligible difference in energy used for AES (symmetric cryptography) on data encryption/decryption.

7.2 A Sender and Multiple Receivers

In this section, we describe the security overheads in one-to-many communication architecture, where one node sends encrypted messages to multiple other entities. We conducted experiments with four different settings for a sender and receivers described in Figure 18. The first setting in Figure 18 (a) employed a separate, individual TLS connection between the sender and each receiver. Figure 18 (b) shows another setting using individual secure connections but with a shared session key distributed by Auth. The setting in Figure 18 (c) used a publish-subscribe protocol, MQTT [5],

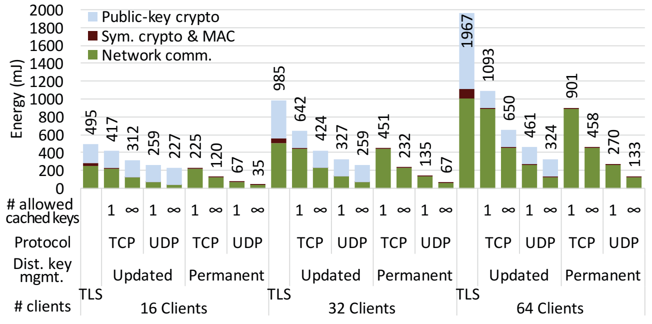


Figure 17: Estimated energy consumption of a server for setting up and closing secure connections with 16, 32, and 64 clients

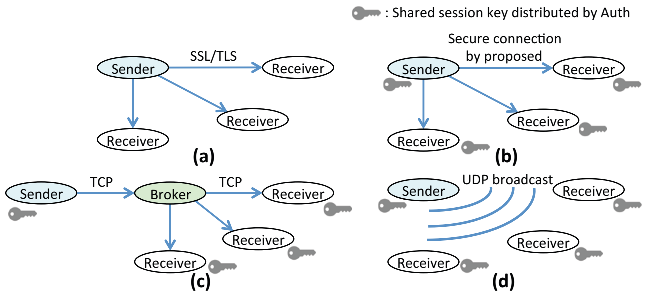

: Shared session key distributed by Auth
Sender
SSL/TLS
Receiver
(a)
Secure connection by proposed
(b)
TCP
Broker
(c)
UDP broadcast
(d)

Figure 18: Four different settings of a sender and receivers; (a) individual SSL/TLS connections (b) individual secure connections by the proposed approach using a shared session key (c) publisher and subscribers connected via a message broker (d) via UDP broadcast in a local network

to connect the sender and receivers sharing a single session key. We used an open-source MQTT message broker[22] for forwarding published messages from the sender to receivers. We assumed that the broker should not be able to decrypt the published messages.

In the final setting shown in Figure 18 (d), we assumed that the sender and receivers were on the same local network; here, the sender employed a UDP broadcast for sending messages encrypted with a shared session key. An example of this last setting is one where messages are made broadly available to the local network, such as alerts or notifications. In addition, we varied the distribution key management for each experiment (i.e., updated or permanent distribution keys).

Figure 19 shows the estimated energy consumed for setting up keys and connections with different numbers of receivers. The energy consumption for *TLS* (Figure 18 (a)) and *ISC* (Individual Secure Connections, Figure 18 (b)) increases as the number of receivers increased. However, the energy consumption for *MB* (Message Broker, Figure 18 (c)) and *UB* (via UDP Broadcast, Figure 18 (d)) remains constant. This is because the sender in *MB* only needs to communicate with Auth and the broker, and the only overhead for the sender in *UB* occurs when obtaining a shared session key from Auth. The overhead of public-key cryptography occurs at most once in SST, resulting in less energy consumption than TLS as explained in Section 7.1.

[22]Mosquitto (http://mosquitto.org)

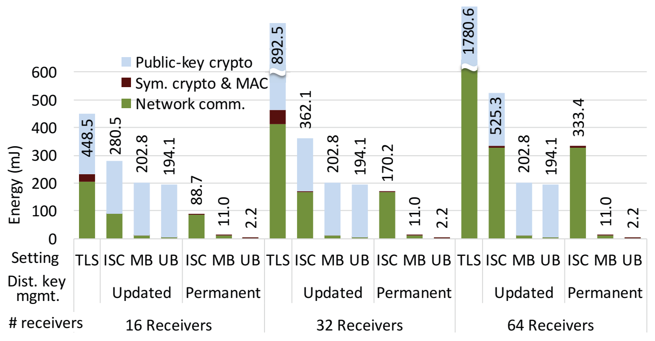


Figure 19: Estimated energy consumption of a sender for setting up secure connections with 16, 32, and 64 receivers (*ISC*: Individual Secure Connections, *MB*: with a Message Broker, *UB*: via UDP Broadcast)

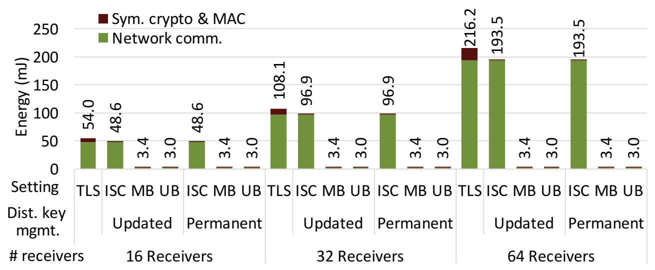


Figure 20: Estimated energy consumption of a sender for sending a 1 KB message to 16, 32, and 64 receivers

Figure 20 depicts the estimated energy consumption for a scenario where the sender attempts to deliver a 1 KB message to different numbers of receivers. The results show that the sender in MB and UB uses a constant amount of energy even when the number of receivers increases; this is because the sender only needs to encrypt and send the message once to the broker in MB and to the local network in UB. The sender uses less energy in ISC than TLS because the sender in ISC only needs to encrypt the message once, thanks to the shared session key. However, the impact of this is not significant because energy consumption in communication is dominant, and both TLS and ISC require sending messages to individual receivers separately. There is no difference between two distribution key management alternatives in this experiment because no public-key cryptography was used.

To illustrate how different security configurations affect the lifetime of IoT devices, let us consider two battery-powered sensor nodes, each sending a 1 KB message to 64 receivers every minute. Assume that one uses ISC (193.5 mJ/message) while the other uses UB (3.0 mJ/message), and sending 1 KB messages is the only activity for these two sensor nodes. If we use a 500 mAh battery operating on 1.5 V, the total energy budget will be 0.75 Wh, which is 2.7 kJ. Under these conditions, the sensor node using ISC will die within 10 days while the one using UB will last for 625 days.

8 RELATED WORK

The most significant improvements of SST compared to our previous work [20] are the newly proposed accessors, a rigorous, formal security analysis and scalability analysis, and more in-depth experiments comparing a variety of security configurations. Other improvements include supports for more alternatives such as UDP and Diffie-Hellman key exchange, and more concrete open-source implementation of Auth using Java and relational database, SQLite, with encrypted credentials.

OpenIoT [37] is a platform designed to enable integration among a collection of heterogeneous IoT applications. The platform leverages a publish-subscribe architecture to allow different types of devices to communicate to each other. For privacy and security, OpenIoT relies on a central authentication mechanism based on SSL, which, as we have discussed, is likely to face scalability challenges in dynamic IoT networks.

Hummen *et al.* [17] propose a security framework for IoT devices based on the datagram TLS (DTLS) protocol [31]. Similar to our approach, their framework employs specialized authorization entities, *delegation servers*, to reduce the amount of public-key cryptography computations. In comparison, SST provides a wider range of configurations, as shown in Figure 5, allowing each entity to create its own profile based on its security and resource requirements.

Seitz *et al.* [33] outline a set of desirable security and performance requirements for an IoT network, and propose a conceptual framework for controlling access to device resources using the XACML policy language [15]. However, their approach is not based on a particular authentication scheme, and does not directly address scalability issues.

The Constrained Application Protocol (CoAP) [34] is designed to support the types of low-power devices that are common on an IoT network. Communication between CoAP devices is secured using DTLS, which still relies on each individual device to perform public-key operations, or share symmetric keys with other trusted devices prior to the deployment. Furthermore, CoAP is mainly designed for one-to-one communication (e.g., a client-server model), and does not directly support one-to-many settings (e.g., publish-subscribe).

A variety of security frameworks for sensor networks have been proposed and studied [6, 19, 23, 29, 30]. Sensors and IoT devices have similar resource constraints, but we expect the latter group to be more diverse in terms of the types of applications that they implement. SST can be deployed as an underlying infrastructure for a mixture of traditional sensor nodes as well as entities with application-specific requirements.

Wei *et al.* [39] propose a conceptual design of security infrastructure for deploying smart grid networks. Although their focus is on power grids, their approach is similar to ours in that it provides integration between different types of devices with varying performance and security requirements.

SHAWK [8] provides a secure mechanism of integrating heterogeneous wireless networks including cellular and WLANs. Like SST, SHAWK addresses heterogeneity by integrating existing solutions but at a different layer of abstraction.

To the best of our knowledge, SST is the first working implementation of an Internet-scale authorization infrastructure that covers heterogeneous security requirements from sensor nodes to safety-critical components, with an automated, formal security analysis. The proposed infrastructure is not just a protocol or key management system but it also provides standardized software components, accessors, for secure composition of IoT applications.

9 CONCLUSIONS

In this paper, we present SST, a novel toolkit for constructing an authorization service infrastructure for the IoT. We expect heterogeneous IoT devices, ranging from sensor nodes to electric power grid control systems, can be integrated into the authorization infrastructure by virtue of SST's diverse security alternatives. Auth's scalability will enable Internet-scale deployment of the proposed infrastructure together with SST's support for one-to-many communication to cope with increasing data traffic. We also envision SST can facilitate further integration in IoT network protocols, for example, by providing key distribution mechanisms for existing network protocols for the IoT such as CoAP over DTLS.

As future work, we plan to solve challenges that still need to be addressed for further security of the IoT. One of the most important challenges is defense and mitigation against denial-of-service attacks breaching availability. We speculate the distributed nature of Auths in SST can help enhancing the IoT's availability. Ease of deployment of authorization services for the IoT is another important challenge. Accessors included in the open-source SST are expected to reduce the burden of IoT developers and increase accessibility to security solutions. Other remaining challenges include timely detecting malicious behavior in the IoT and providing guarantees for swarm applications running remotely on untrusted IoT platforms.

ACKNOWLEDGMENTS

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. The authors would like to thank Salomon Lee for his contribution to the open-source implementation of Auth.

REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *23rd IEEE Computer Security Foundations Symposium, Edinburgh, UK*. 290–304.

[2] Ross J. Anderson. 2008. *Security engineering - a guide to building dependable distributed systems (2. ed.)*. Wiley.

[3] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. *Evaluating the Small Scope Hypothesis*. Technical Report. MIT CSAIL.

[4] Krzysztof R. Apt and Dexter Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 22, 6 (1986), 307–309.

[5] Andrew Banks and Rahul Gupta. 2014. MQTT Version 3.1.1. OASIS Standard, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html. (Oct 2014).

[6] Paolo Baronti, Prashant Pillai, Vince W. C. Chook, Stefano Chessa, Alberto Gotta, and Yim-Fun Hu. 2007. Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Comput. Commun.* 30, 7 (2007), 1655–1695.

[7] Ismail Butun, Salvatore D. Morgera, and Ravi Sankar. 2014. A Survey of Intrusion Detection Systems in Wireless Sensor Networks. *IEEE Communications Surveys Tutorials* 16, 1 (2014), 266–282.

[8] Jiannong Cao and others. 2012. SHAWK: Platform for Secure Integration of Heterogeneous Advanced Wireless Networks. In *26th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 13–18.

[9] Cisco Press Release. 2016. *Cisco Visual Networking Index Predicts Near-Tripling of IP Traffic by 2020*. Technical Report. https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=1771211

[10] Seyed Hossein Erfani, Hamid H.S. Javadi, and Amir Masoud Rahmani. 2015. A dynamic key management scheme for dynamic wireless sensor networks. *Security and Communication Networks* 8, 6 (April 2015), 1040–1049.

[11] P. Eronen and H. Tschofenig. 2005. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). RFC 4279. (Dec. 2005).

[12] Laura M. Feeney and Martin Nilsson. 2001. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proc. of IEEE INFOCOM 2001. 20th Annual Joint Conf. of the IEEE Comput. and Commun. Societies.*, Vol. 3. 1548–1557.

[13] Pedro Garcia Lopez and others. 2015. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015), 37–42.

[14] Branden Ghena, William Beyer, Allen Hillaker, Jonathan Pevarnek, and J. Alex Halderman. 2014. Green Lights Forever: Analyzing the Security of Traffic Infrastructure. In *The 8th USENIX Workshop on Offensive Technologies (WOOT '14)*. San Diego, CA.

[15] Simon Godik and Tim Moses. 2005. eXtensible Access Control Markup Language (XACML). OASIS Standard Version 2.0, http://www.oasis-open.org/committees/xacml. (Feb 2005).

[16] Carl Hewitt. 1977. Viewing control structures as patterns of passing messages. *Artificial Intelligence* 8, 3 (June 1977), 323–364.

[17] René Hummen, Hossein Shafagh, Shahid Raza, Thiemo Voig, and Klaus Wehrle. 2014. Delegation-based authentication and authorization for the IP-based Internet of Things. In *11th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. 284–292.

[18] Daniel Jackson. 2006. *Software Abstractions: Logic, language, and analysis*. MIT Press.

[19] Chris Karlof, Naveen Sastry, and David Wagner. 2004. TinySec: a link layer security architecture for wireless sensor networks. In *SenSys 2004*. Baltimore, MD, USA, 162–175.

[20] Hokeun Kim, Armin Wasicek, Benjamin Mehne, and Edward A. Lee. 2016. A Secure Network Architecture for the Internet of Things Based on Local Authorization Entities. In *The 4th IEEE International Conference on Future Internet of Things and Cloud*. Vienna, Austria, 114–122.

[21] Elizabeth Latronico, Edward A. Lee, Marten Lohstroh, Chris Shaver, Armin Wasicek, and Matthew Weber. 2015. A Vision of Swarmlets. *IEEE Internet Computing* 19, 2 (March 2015), 20–28.

[22] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. 2003. Actor-Oriented Design of Embedded Hardware and Software Systems. *Journal of Circuits, Systems and Computers* 12, 03 (June 2003), 231–260.

[23] Mark Luk, Ghita Mezzour, Adrian Perrig, and Virgil D. Gligor. 2007. MiniSec: a secure sensor network communication architecture. In *Proc. of the 6th Int. Conf. on Inform. Process. in Sensor Networks (IPSN) '07*. Cambridge, MA, USA, 479–488.

[24] Sanjai Narain and others. 2005. Network Configuration Management via Model Finding. In *Proc. of LISA '05*, Vol. 5. 15–15.

[25] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *LISA, San Jose, CA, USA*.

[26] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. 2005. The Kerberos Network Authentication Service (V5). RFC 4120. (July 2005).

[27] Kim Thuat Nguyen, Maryline Laurent, and Nouha Oualha. 2015. Survey on secure communication protocols for the Internet of Things. *Ad Hoc Networks* 32 (Sept. 2015), 17–31.

[28] Adrian Perrig, Ran Canetti, J.D. Tygar, and Dawn Song. 2005. The TESLA Broadcast Authentication Protocol. *RSA CryptoBytes* (July 2005).

[29] Adrian Perrig, John A. Stankovic, and David Wagner. 2004. Security in wireless sensor networks. *Commun. ACM* 47, 6 (2004), 53–57.

[30] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. 2002. SPINS: Security Protocols for Sensor Networks. *Wireless Networks* 8, 5 (2002), 521–534.

[31] E. Rescoria and N. Modadugu. 2012. Datagram Transport Layer Security Version 1.2. RFC 6347. (Jan. 2012).

[32] Helena Rifà-Pous and Jordi Herrera-Joancomartí. 2011. Computational and Energy Costs of Cryptographic Algorithms on Handheld Devices. *Future Internet* 3, 1 (Feb. 2011), 31–48.

[33] L. Seitz, G. Selander, and C. Gehrmann. 2013. Authorization framework for the Internet-of-Things. In *IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 1–6.

[34] Z. Shelby, K. Hartke, and C. Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 6347. (June 2014).

[35] Chia-Yen Shih and others. 2014. On the Cooperation between Mobile Robots and Wireless Sensor Networks. In *Cooperative Robots and Sensor Networks 2014*. Number 554. Springer Berlin Heidelberg, 67–86.

[36] Jatinder Singh, Thomas Pasquier, Jean Bacon, Hajoon Ko, and David Eyers. 2016. Twenty Security Considerations for Cloud-Supported Internet of Things. *IEEE Internet of Things Journal* 3 (June 2016), 269–284.

[37] John Soldatos and others. 2015. OpenIoT: Open Source Internet-of-Things in the Cloud. In *Interoperability and Open-Source Solutions for the Internet of Things*. Number 9001. Springer International Publishing, 13–25.

[38] Stefan Tilkov and Steve Vinoski. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[39] Dong Wei, Yan Lu, M. Jafari, P. Skare, and K. Rohde. 2010. An integrated security system of protecting Smart Grid against cyber attacks. In *Innovative Smart Grid Technologies (ISGT), 2010*. 1–7.