Securing the Internet of Things via Locally Centralized, Globally Distributed Authentication and Authorization

Hokeun Kim

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-139
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-139.html

August 9, 2017

Securing the Internet of Things via Locally Centralized, Globally Distributed Authentication and Authorization

by

Hokeun Kim

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Professor Alberto L. Sangiovanni-Vincentelli
Professor Raja Sengupta

Summer 2017

Securing the Internet of Things via Locally Centralized, Globally Distributed Authentication and Authorization

Abstract

Securing the Internet of Things via Locally Centralized, Globally Distributed Authentication and Authorization

by

Hokeun Kim

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

The Internet of Things (IoT) brings about benefits through interaction with humans and the physical world using a variety of technologies including sensors, actuators, controls, mobile devices and cloud computing. However, these benefits can be hampered by malicious interventions of attackers when the IoT is not protected properly. Hence, authentication and authorization comprise critical parts of basic security processes and are sorely needed in the IoT. Characteristics of the IoT render existing security measures such as SSL/TLS (Secure Socket Layer/Transport Layer Security) and network architectures ineffective against emerging networks and devices. Heterogeneity, scalability, and operation in open environments are serious challenges that need to be addressed to make the IoT secure. Moreover, many existing cloud-based solutions for the security of the IoT rely too much on remote servers over possibly vulnerable Internet connections.

This dissertation presents *locally centralized, globally distributed* authentication and authorization to address the IoT security challenges. Centralized security solutions make system management simpler and enable agile responses to failures or threats, while having a single point of failure and making it challenging to scale. Solutions based on distributed trust are more resilient and scalable, but they increase each entity's overhead and are more difficult to manage. The proposed approach leverages an emerging network architecture based on edge computers by using them as locally centralized points for authentication and authorization of the IoT. This allows heterogeneity and an agile access control to be handled locally, without having to depend on remote servers. Meanwhile, the proposed approach has a globally distributed architecture throughout the Internet for robustness and scalability.

The proposed approach is realized as *SST* (*Secure Swarm Toolkit*), an open-source toolkit for construction and deployment of an authentication and authorization service infrastructure for the IoT, for validation of locally centralized, globally distributed trust management. SST includes a local authorization entity called *Auth* to be deployed on edge computers which are used as a gateway for authorization as well as for the Internet. Software building blocks provided by SST, called *accessors*, enable IoT developers to readily integrate their

IoT applications with the SST infrastructure, by encapsulating cryptographic operations and key management. In addition to protection against network-based intruders, SST supports a secure migration mechanism for enhancing availability in the case of failures or threats of denial-of-service attacks, based on globally distributed and trusted Auths.

For evaluation, I provide a formal security analysis using an automated verification tool to rigorously show that SST provides necessary security guarantees. I also demonstrate the scalability of the proposed approach with a mathematical analysis, as well as experiments to evaluate security overhead of network entities under different security profiles supported by SST. The effectiveness of the secure migration technique is shown through a case study and simulation based on a concrete IoT application.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor, Prof. Edward A. Lee, for always guiding and supporting me. He has been a great mentor, teacher and role model for me throughout my studies. He taught me not only how to do research, but also the philosophy of research. Without his encouragement and advice, I could not have accomplished all that I have at Berkeley. I would also like to wholeheartedly thank Prof. Alberto L. Sangiovanni-Vincentelli, Prof. Raja Sengupta and Prof. Sanjit A. Seshia for providing valuable feedback and guidance while on my qualifying exam and dissertation committees.

I am sincerely grateful to my mentors and collaborators, and I was truly blessed to meet and work with them. From David Broman, I have learned how to shape and develop research problems, which has made me a better researcher. Michael Zimmer helped me join the PRET project and explore research during my first two years in the graduate school. Discussions with Vern Paxson and Ben Mehne sparked initial interests in the security aspects of the Internet of Things and had let me figure out challenges to be addressed. By working with Armin Wasicek, I could refine the first design of the proposed approach in my dissertation. Eunsuk Kang helped security analysis and was always available for research discussions which made my work more advanced. My old friend Salomon Lee enabled the first release of the Secure Swarm Toolkit.

I would like to acknowledge the current and past members of the Ptolemy group and DOP Center for their friendliness and valuable discussions, including: Ilge Akkaya, Ravi Akella, Patricia Derler, John Eidson, Schahram Dustdar, Antonio Iannopollo, Inigo Incer, Chadlia Jerad, Yooseong Kim, Marten Lohstroh, Victor Nouvellet, Aviral Shrivastava, Chris Shaver, Marjan Sirjani, Stavros Tripakis, Matt Weber, and Ben Zhang. I would like to especially thank Christopher Brooks, Mary Stewart, and Shirley Salanio for always being supportive and available.

I also sincerely appreciate the generous support of the Korea Foundation for Advanced Studies (KFAS), which had funded my graduate studies for five years.

Last but not least, I would like to thank my parents, for their endless support and love.

Chapter 1

Introduction

The Internet of Things (IoT) [7] faces challenges [68] to enable scalable, safe and secure systems. Since the IoT interacts with humans, machines and environments, failures in the IoT can lead to very serious consequences. This fact makes the safety of the IoT particularly important. Safety extends to security in the sense that security guarantees (e.g., protection from intrusion or unauthorized access) can help prevent an adversary from damaging safety. Safety measures such as Airbus flight envelope protection [100], which prohibits pilots from performing risky maneuvers, can help prevent a successful intruder from doing damage.

The security of the traditional Internet has been enhanced by well-developed security measures such as the SSL/TLS (Secure Socket Layer/Transport Layer Security) protocol suites[1]. However, the IoT has unique characteristics that distinguish it from the traditional Internet, and these characteristics lead to special requirements for security solutions of the IoT. Widely-used network security measures do not adapt one-to-one to the IoT because of these special requirements.

1.1 The Internet of Things (IoT) and Security

On October 21, 2016, domain name service provider Dyn suffered a distributed denial-of-service (DDoS) attack [44] leading to a significant collapse of fundamental infrastructures comprising the Internet. In a DDoS attack, a number of compromised or zombie computers, forming a botnet, send a flood of traffic to the target server, causing a denial of service by exhausting computation or communication resources. What made this incident remarkable was the fact that many of compromised computers launching the attack were relatively small devices including printers, webcams, residential gateways, and baby monitors, i.e., the Internet of Things (IoT).

The IoT benefits from connectivity that facilitates collaboration among a variety of computing systems, ranging from sensor nodes and mobile devices to large control systems and cloud computers. The IoT closely interacts with human beings and physical systems such

[1] Widely used by web servers, clients, and remote logins.

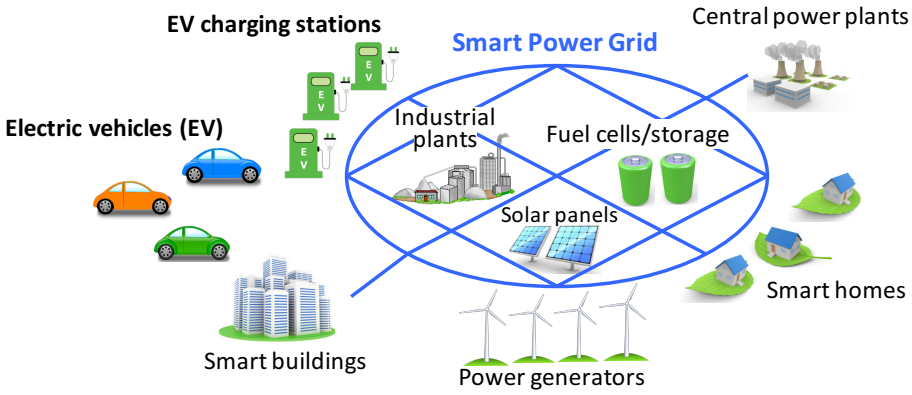


Figure 1.1: Electric vehicles (EVs) and charging infrastructure.

as medical devices or smart power grids. However, as seen in the Dyn incident, the IoT also brings about security challenges, especially when the "Things" lack security.

Another incident that showed the threat of the connectivity was the cyberattack on Ukrainian power grid [61] on December 23, 2015. The attackers gained control over the SCADA (Supervisory Control and Data Acquisition) system of the Ukrainian power grid and caused a blackout for several hours in the large area of Ukraine's Ivano-Frankivsk region populated by 1.4 million residents. This case showed that, unlike cyberattacks in the past, the consequences of attacks on the IoT can be more devastating than information theft or financial loss. The consequences can be life-threatening.

To avoid such incidents, IoT can be secured by widely-used cloud computing technologies where the IoT can benefit from plentiful resources of cloud servers for security services. However, this means the security services such as authentication and authorization can be affected by the availability of the cloud servers and the Internet connections to remote servers. A recent example showing this is Google OnHub incident [70], where a failure in the companys authentication servers caused IoT gateway devices (called OnHub) to become unavailable, in turn leading to failures in all IoT devices connected to these gateway devices. The connection to the cloud can be disrupted by impairing Internet infrastructure services such as DNS (Domain Name Service). The indication of the Google OnHub case is that depending on remote cloud servers over possibly brittle Internet connections can be risky, especially for authentication and authorization, considering that access control is essential for the system's availability.

As recognized in the literature including [95], the main challenges in the security of the Internet of Things (IoT) include heterogeneity, operation in open environments, and scalability. For example, IoT components including electric vehicles (EVs) and EV charging infrastructure in Figure 1.1 are considered safety-critical. Unlike servers in data centers, EVs and EV charging stations are physically accessible not only by valid users but also by potential adversaries. This leads to an increased number of physical points of access; thus,

there can be a higher risk of being subverted. Therefore, an authorization infrastructure for the IoT must have ways to revoke access of the compromised devices within a short amount of time to limit the damage when they are under control of adversaries.

In addition, mobile phones or EVs can migrate from one network to another, possibly making their network connection unstable. There are also IoT devices with constrained resources and the number of IoT devices is expected to grow rapidly. Therefore, the security infrastructure for the IoT should work well with unstable connection and resource-constrained devices at a great scale. However, with security measures such as TLS based on certificates provided by certificate authorities, it will be very difficult to have control over authorization of a huge number of devices, some of which may have seriously limited resources.

There is a variety of examples showing that the networked entities in the IoT are heterogeneous in terms of both security requirements and resource availability. For example, safety-critical systems such as above-mentioned 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.

Therefore, insisting on maximum security for all devices in the IoT is not appropriate. To the best of my 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, 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.* [38] 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. Jamming attacks on wireless communication channels [107] can also be a threat to the availability of the IoT operating over wireless networks.

The last but not least challenge is scalability of connected devices in the IoT. Many reports on scalability of the IoT, including ones by Ericsson [22] and Cisco [25], expect there will be tens of billions of connected devices by 2020, far exceeding the world population, and billions

of Terabytes (10^{12} Bytes) of IP traffic, a significant portion of which will be generated by the IoT. Hence, the security solution must scale accordingly; in particular, the overhead of adding and removing devices to/from the security solution should be minimal. And securing a large amount of data should be done in an efficient way. This includes supporting object or data security for communications such as information-centric networks [1], publish-subscribe [35] or broadcasting.

1.2 Motivation

1.2.1 Challenges with Current Security Measures

This section discusses 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. I 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 discussion in this section, consider data collection in a WSN (wireless sensor network) using a UAV (unmanned aerial vehicle), as shown in Figure 1.2. This example is motivated by Shih *et al.* [94]

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 particularly critical for safety: 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 [75], 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

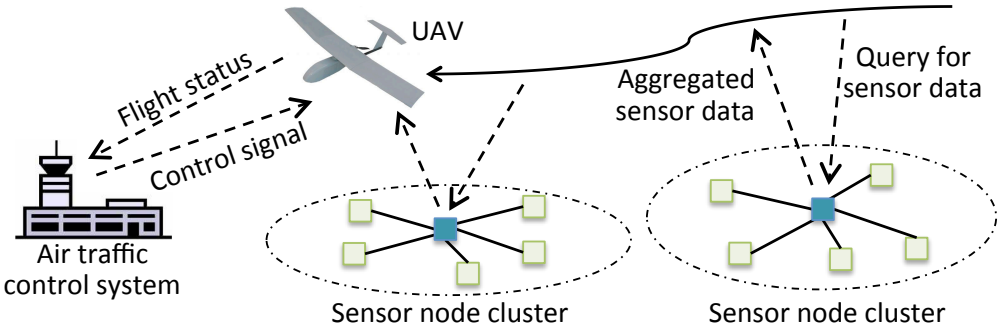


Figure 1.2: Motivational example for a security measure for the IoT inspired by [94]; WSN data collection using a UAV.

freshness of the authentication request. Kerberos provides centralized and timely control of authentication; thus, in the context of the example in Figure 1.2, 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 [76] [33] 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. The proposed approach and the toolkit as its realization provide an *integrated*, *Internet-scale* authentication and authorization framework that can satisfy a diverse set of security and resource requirements found in an IoT network.

1.2.2 Security Requirements for the IoT

This section summarizes security requirements of the security solution for the IoT.

- **Frequent authorization and authentication** – Due to the criticality of some devices in the IoT, tight authorization and authentication will be necessary. Machines operate at higher speeds than human beings, and physical accessibility of devices creates more vulnerability. Dynamically varying situations resulting from mobility may

change authorization of devices. Moreover, frequent authorization can provide ways to limit the damage in case critical devices are subverted.

- **Automated mutual authentication** – It will be prohibitive for users to remember passwords for a large number of devices. Thus, the IoT devices must be able to authenticate themselves without user intervention.
- **Intermittent connectivity** – Mobility of devices changes the network environment under which the devices operate, which can lead to unstable connectivity. However, we cannot compromise security when the network connection is unstable, thus we should be able to handle intermittent connectivity of devices.
- **Dynamic registration of IoT entities** – Unlike the traditional Internet, the IoT includes devices with shorter life cycles than general computers. Mobile devices may be added to and removed from authentication/authorization systems dynamically. Therefore, adding and removing entities should be manageable in the secure network architecture for the IoT.
- **Support for scalability features** – There is a variety of approaches for network scalability, which benefit the IoT devices, including the publish-subscribe protocols [35] such as MQTT [8]. The security architecture should be able to work together with these scalability features.
- **Consideration for resource constraints** – Some devices in the IoT have scarce resources; for example, battery-powered devices have limited energy budget for computation or communication. Stronger security measures generally cost more energy; however, excessive energy consumption can harm the availability. Hence, the security measure should be able to balance security and energy consumption.
- **Privacy** – Personal devices such as mobile phones can easily collect and carry private information of their users. Therefore, the security measure also needs to be able to protect user's privacy.
- **Resiliency and Robustness** – For some IoT applications, availability is as critical as information protection. Therefore, the security solution for the IoT needs to be able to maintain some degree of availability even in the case of failures or availability threats such as Denial-of-Service (DoS) attacks.
- **Locality** – The IoT security measure must not depend too much on remote systems. Specifically, authentication and authorization services should not be interrupted by the Internet connections or operational states of remote servers.
- **Ease of deployment** – To deal with scalability and management of heterogeneous IoT systems, the security solution should be easy enough to be deployed by local system managers. This includes enabling IoT developers with moderate knowledge of computer security to integrate their applications with the security solution.

1.3 Contributions

In this dissertation, I propose locally centralized, globally distributed authentication and authorization to secure the IoT while addressing the IoT-related requirements. This dissertation also presents a toolkit called *SST (Secure Swarm Toolkit)* [53] and its infrastructure designed as a realization of the proposed approach. To the best of my 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. SST is not just a protocol or key management system but it also provides standardized software components, accessors, for secure composition of IoT applications. The key contributions of the dissertation are summarized as follows:

- This dissertation proposes locally centralized, globally distributed authentication and authorization as a solution to address security challenges in the IoT. This approach takes the hybrid of centralized and distributed solutions and leverages the emerging network architecture called edge computing.
- As a concrete implementation of the proposed approach, this dissertation presents 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 edge-computing devices including Intel's IoT gateways and SwarmBox which will be introduced in Section 2.2 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. For example, the proposed Auth can use the cryptography algorithms that are used in TLS to provide a comparable level of security guarantees for safety-critical systems while providing lightweight solutions for battery-powered 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.
- The proposed 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, I have performed a formal analysis on a model of the proposed authorization protocol using an automated

verification tool. As far as I know, this is the first security framework for the IoT that has been subjected to a rigorous, formal security analysis.

- A robust authentication and authorization service infrastructure for the IoT is built upon the open-source implementation of SST. SST is particularly interesting because, by its design nature, it provides distributed Auths and leverages trust among Auths. In the previous design of SST, each Auth takes responsibility of local IoT entities for authorization. In the proposed approach, Auths are allowed to share the authorization information (credentials and communication policies) of the registered entities with other trusted Auths. When an Auth becomes unavailable due to a DoS attack, another trusted Auth can take over the authorization tasks for the entities of the unavailable Auth.

The rest of this dissertation is organized as follows. Chapter 2 presents the background of research. Chapter 3 describes the proposed approach, locally centralized, globally distributed authentication and authorization along with an open-source toolkit, SST, a realization of the approach. The design and implementation of the proposed approach are demonstrated in Chapter 4, followed by evaluation through analysis and experiments in Chapter 5. Chapter 6 introduces a secure migration mechanism for enhancing availability with the proposed approach and evaluates its robustness through experiments. Related work is summarized in Chapter 7. Conclusions and future directions of the research are presented in Chapter 8.

Chapter 2

Background

This chapter discusses the background of research, starting with important security-related concepts including authentication, authorization, and trust in the context of the IoT. I also provide a survey of an emerging network architecture that opens up opportunities for security research of the IoT, called "edge computing".

2.1 Authorization, Authentication and Trust

In Section 1.1, we saw two cases where the security of the IoT was undermined, the Dyn cyberattack and the Ukrainian power grid attack. So why did the IoT fall under the attacker's control in these two aforementioned incidents? First, things that did not used to be connected are now connected to the Internet, a wilderness full of potential adversaries. Second, the connected things were not robust enough to be exposed to the wilderness in part because they lacked proper mechanisms for access control.

Access control, or *authorization*, is the process of determining whether an entity (a device or a user) can access resources, e.g., read or write data, execute programs, and control actuators. Authorization also includes denying or revoking access, especially for someone or something malicious. *Authentication*, a process of identifying an entity, is a prerequisite for authorization. In most cases, authorization is not even possible without proper authentication. How can we grant or deny access to someone or something that we do not know about?

Authentication is intrinsically based on *trust*. For example, when we check someone's ID, we first have to trust the issuer of the ID, such as a national government, as shown in Figure 2.1 (a). The same analogy applies to networked computers. Figure 2.1 (b) illustrates the process of establishing trust between a browser (client) and a bank's website (server). A number of modern websites use HTTPS, a secure version of HTTP running over SSL/TLS (Secure Socket Layer/Transport Layer Security), a widely-used protocol providing channel security guarantees. SSL/TLS uses public-key cryptography for channel establishment; thus, the authenticity of a server's public key is critical. A (digital) certificate, a token for au-

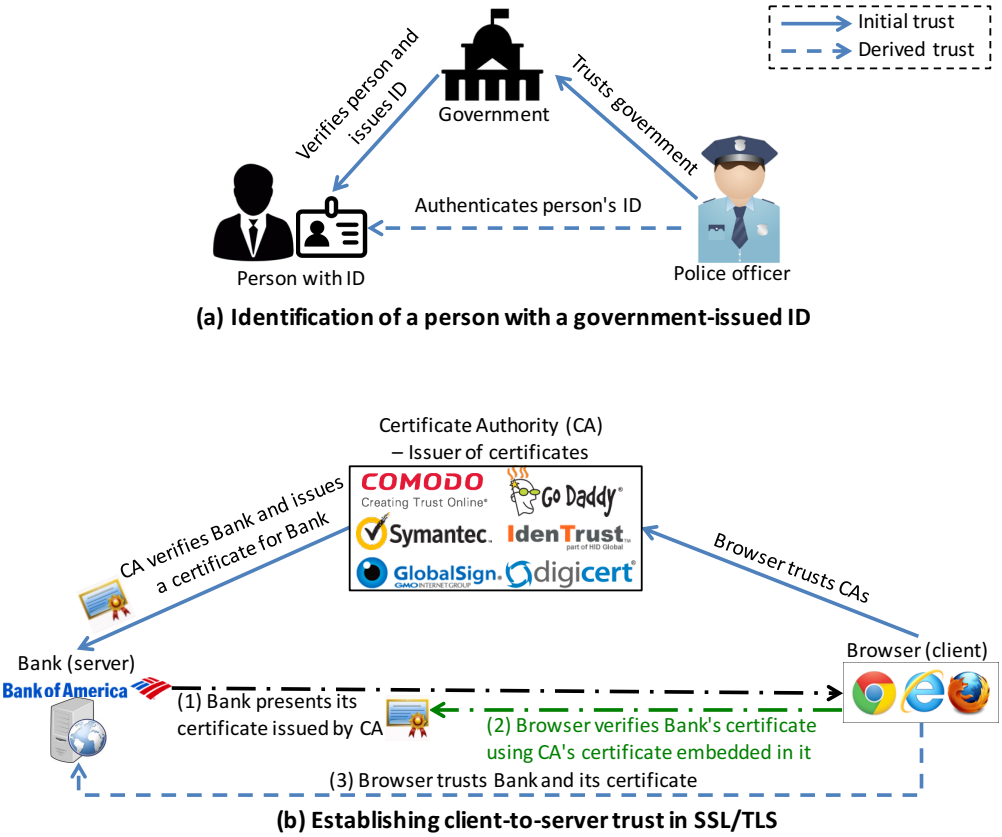


Figure 2.1: Trust and Authentication. (a) In identifying a person with a government-issued ID, we must trust the issuer. (b) There are several steps to establishing client-to-server trust in SSL/TLS.

thentication, includes the server's public key and its identity called a common name which is usually the server's domain name. A certificate is issued and digitally signed by a certificate authority (CA). Initially, the browser trusts CA and the CA has issued a certificate for the bank's website. When the browser connects, the web server presents its certificate (Figure 2.1 (b)-(1)). The browser verifies the bank's certificate using CA's certificate (Figure 2.1 (b)-(2)). If the verification succeeds, the browser trusts the bank's web server (Figure 2.1 (b)-(3)). A Public Key Infrastructure (PKI) is a set of roles and policies for issuing and managing these certificates.

Indeed, there is a variety of ways to implement tokens for authentication in computer systems. Passwords are the most common ways to authenticate human users. For additional security, we often use two-factor authentication using what we have (such as phones) and what we are (for instance, fingerprints) in addition to what we know (passwords).

In machine-to-machine communications, cryptography is a powerful tool for providing security guarantees. In such crypto systems, cryptographic keys are commonly used as tokens

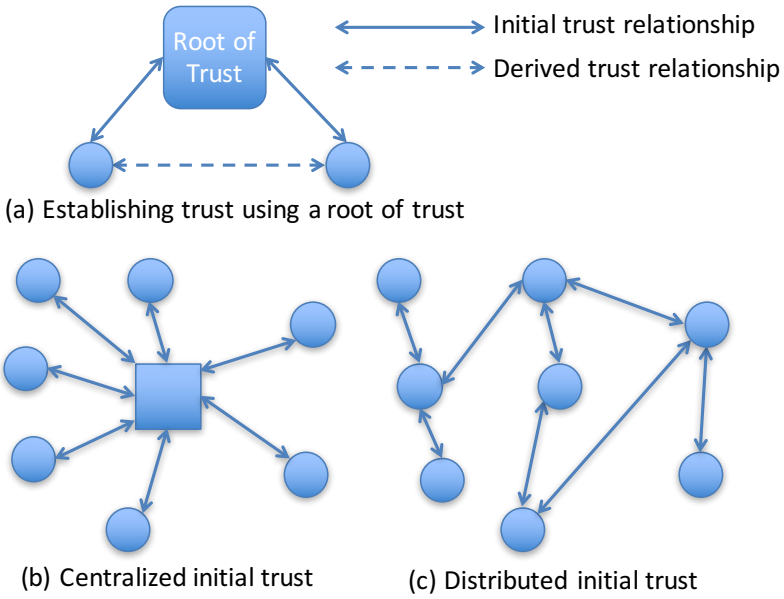


Figure 2.2: Building trust in networked systems. Ways to build trust include (a) establishing trust using a root of trust and having either (b) a centralized initial trust or (c) a distributed initial trust.

for authentication and also for authorization. Therefore, in many cases, authentication, authorization, and trust management come down to the problems of generating and managing cryptographic keys.

Establishing trust in computing usually starts with a *root of trust*, which constructs an initial trust relationship. Using the root of trust, further trust relationships are derived as shown in Figure 2.2 (a). A root of trust can be a special hardware component such as a TPM (Trusted Platform Module) [99]. Another example is the *root certificate* in PKI.

For networked computers, there are two broad classes of ways to set up the initial trust relationship. One is to ask a centralized authority, as in Figure 2.2 (b); and the other is to use distributed trusted fellows, as in Figure 2.2 (c).

2.1.1 Asking a Centralized Trusted Authority

In centralized trust schemes, the centralized authority is often called a trusted third party because it does not participate in the communication. SSL/TLS based on PKI is one of the most widely-used approaches using a centralized CA for authentication. Although there are multiple trusted CAs in PKI, the trust management is still centralized in the same sense that multiple national governments are centralized authorities.

Another widely-used approach using a centralized trusted third party is the Kerberos authentication system [75]. Kerberos uses temporary access tokens called *tickets* to au-

thenticate clients and servers and to grant access to the services. Thus, Kerberos provides authorization as well as authentication.

Wireless sensor networks (WSNs), one of the predecessors to the IoT, have their own security solutions. Many WSNs use a base station with plentiful resources to coordinate the battery-powered sensor nodes. Sometimes, a base station also works as a root of trust for sensor nodes, especially as a key distributor. One representative example is Sensor Network Encryption Protocol (SNEP) as part of Security Protocols for Sensor Networks (SPINS) [85]. In SNEP, each sensor node shares a secret key called *master key* with its base station. Further keys between sensor nodes are derived using master keys – in other words, based on trust with the base station, which is a centralized authority.

A pitfall of centralized trust management is that failure of the centralized authority can result in failure of the whole system. One example of this is the WoSign incident that occurred in 2016 [101]. A Chinese certificate authority WoSign mistakenly issued certificates to false subjects; for instance, if you control foobar.github.com, WoSign issued a certificate for *.github.com. This incident showed how an entire security system can be broken when the root of trust gets broken.

2.1.2 Using Distributed and Trusted Participants

Distributed trust schemes can avoid the problem of the centralized authority being a single point of failure. In distributed schemes, there is no centralized trusted third party and the participants coordinate autonomously to build further trust.

The concept of a *web of trust* used by OpenPGP [20] – an encryption standard widely used for email encryption – leverages trust between participants. OpenPGP uses public keys for encrypting messages. The association of a public key with a recipient is as critical as in PKI. Therefore, OpenPGP also uses certificates, but in this case, the certificates are signed by other trusted users rather than a centralized authority. Unlike in PKI, even if a single user gets compromised and has the private key stolen, the effect of the attack would not be catastrophic as in the WoSign incident.

Bitcoin [50], a cryptography-based digital currency, also uses distributed trust. There is no single authority validating Bitcoin transactions. Whenever a Bitcoin transaction occurs, the Bitcoin client broadcasts the transaction to the entire Bitcoin network. Other clients verify the transaction and attach it to a public ledger called a *blockchain*. The blockchain is shared by the Bitcoin clients, therefore it is infeasible for a single malicious Bitcoin client to forge transactions.

Localized Encryption and Authentication Protocol (LEAP+) [113] is an example security protocol for WSNs, based on distributed trust. LEAP+ also uses base stations, but their involvement is limited to certain tasks such as node initialization. For pairwise key creation and management, the sensor nodes collaborate with neighboring nodes. In this way, LEAP+ can reduce the management overhead of a base station and communication overhead of direct communication between a sensor node and a base station which is usually more costly than communication between neighboring nodes.

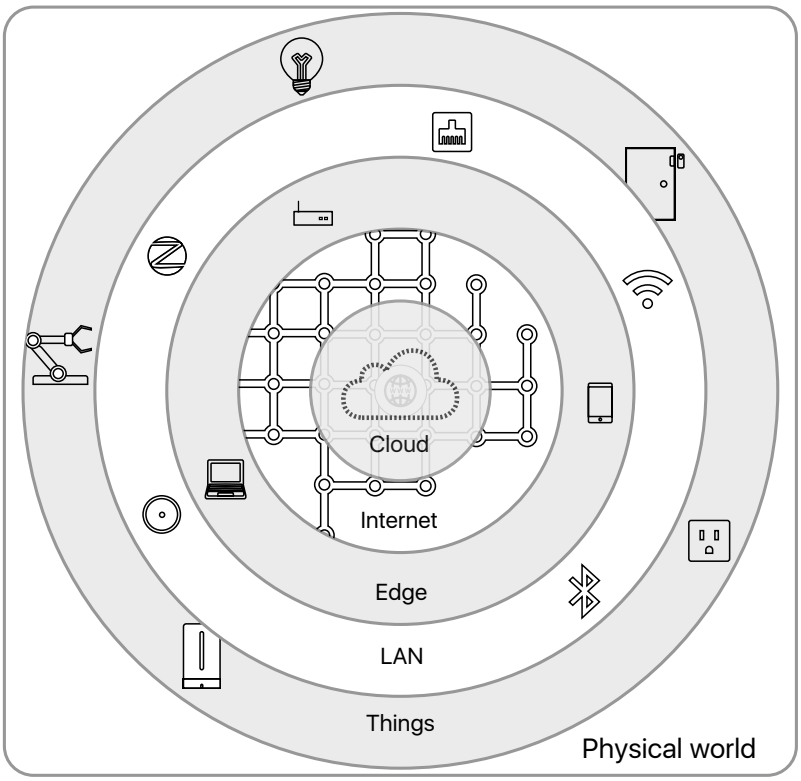


Figure 2.3: Edge computers mediate interactions between Things on a LAN and the Internet.

In general, security schemes based on distributed trust are more resilient than centralized schemes. And the overhead of authentication and authorization can be distributed to participants, leading to better scalability. However, distributed schemes are more vulnerable to collusion attacks, it is harder to manage and keep track of the whole system, and overhead of individual entities tends to be higher than that of centralized schemes.

2.2 Trust and Edge Computing

Building IoT systems using cloud computing has been widely adopted [16], [40]. Services such as Amazon's AWS IoT [45] are primarily cloud services, in this case using Amazon Web Services (AWS). An alternative that I view more promising is to mix cloud services, edge computing, and Things, rather than just cloud services and Things. Edge computing is an emerging architecture, depicted in Figure 2.3, that puts services on devices that are physically much closer to the Things, residing in smart gateways or in networking equipment [63]. Cisco Systems coined the term "fog computing" [14] for such architectures to suggest that it is kind of like the cloud, but closer to the ground. I will refer to this architecture more generically

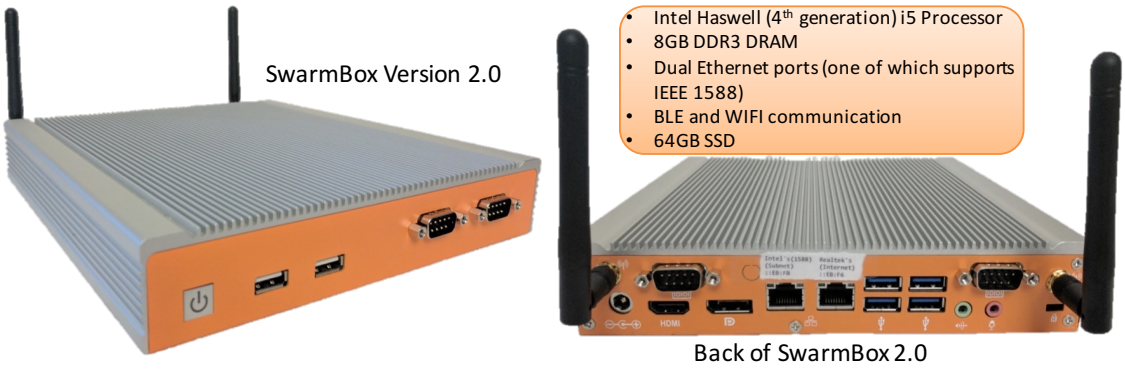


Figure 2.4: The TerraSwarm SwarmBox 2.0 edge computing server is hosted in IMB-186-4300U manufactured by ASRock Inc.

as "edge computing," as in [32].

There are various aspects that need to be considered for the IoT, including scalability, context-awareness, and ease of deployment, in addition to security and privacy, as Pal [81] points out. The amount of data generated by the IoT is increasing rapidly [24], and demand for real-time processing is surging for cyber-physical systems [47] such as autonomous vehicles or factory floors. To satisfy these needs, Cisco Systems coined the term *fog computing* for the IoT. [15] Fog computing utilizes edge-computing devices which include mobile phones, smart gateways (wireless routers with plentiful computational power) and laptop computers, which can act as a gateway to the Internet.

Edge computing has some advantages compared to cloud computing, which is prevalent in recent days. Lopez *et al.* [63] make the following points relevant to security and privacy:

1. Private and sensitive data are kept on the edge rather than sent to the centralized cloud, enhancing privacy.

2. The edge has more context information related to the security and privacy, reducing the overhead for the cloud and better serving the heterogeneous entities.

3. Proximity and intelligence at the edge enable real-time interaction with the IoT, predictable latency, and clock synchronization.

Many computing devices with enough resources can play a role as an edge device. An example is the *SwarmBox* that is proposed as a hardware platform in the TerraSwarm project (`https://terraswarm.org/`). The version 2.0 SwarmBox is an edge computing server hosted in an IMB-186-4300U manufactured by ASRock Inc. SwarmBox 2.0 has the following useful features as an edge device. It supports both WiFi and BLE (Bluetooth Low Energy) for wireless communication, functioning as a smart gateway for devices that connect to the Internet through the SwarmBox. It has dual Ethernet ports, one for the Internet and the

other for local networks. The one for the local network is also equipped with hardware support for the IEEE 1588 Precision Time Protocol (PTP), enabling nanoseconds-scale clock synchronization, a desirable property to support real-time systems.

An edge computer is a computing device that can act as an Internet gateway or a router, such as Intel's IoT gateway[1] or the aforementioned SwarmBox. But rather than being just a provider of networking, edge computers also provide services, including, for example, monitoring sensors for anomalies, brokering authentication and authorization [53], filtering data feeds to ensure privacy, and preprocessing data feeds to forward to the cloud only what the cloud needs.

Any device with computing and networking capability can function as an edge computer, so it is the role of the device not its physical characteristics that I focus on throughout the dissertation. For instance, a smart phone is a Thing when used for its sensors or actuators. At the same time, it is an edge computer when it acts as an Internet gateway for other Things such as wearable devices. A device can perform both roles simultaneously.

Varghese *et al.* [102] classify edge computers as either an "edge node," edge computers running on traditional Internet routers, and an "edge device" for edge computers based on user mobile devices including smart phones and laptops. Since "node" and "device" are almost synonymous, I distinguish "immobile" edge computers from "mobile" edge computers. To be an edge computer, it must interact with Things, and mobility is determined by whether it moves with respect to the Things it interacts with. For example, a mobile edge computer can be inside of a car; the car itself is mobile, but the edge computer is stationary relative to Things on the car connected to it. Similarly, mobile phones are typically bound to humans, and they follow their owners around, remaining stationary from the perspective of their owner's wearables. But they are mobile from the perspective smart building devices, and hence less effective for edge computing.

Immobile edge computers have a big advantage over mobile ones: they become part of the same physical environment occupied by the Thing. For immobile edge computers, it is much easier to ensure proximity and real-time interaction under the assumption that the edge computer has fixed physical location and thus stable network connections, compared to mobile edge computers. They can leverage their locality to control latency; to keep data local (for privacy and security); to offload computation from battery-powered devices; to provide temporary storage for memory-constrained devices; to firewall a local network; and to authorize devices based on physical proximity. The cloud, on the other hand, is a better choice for services that require aggregating data from multiple sources or that exceed the computing and/or memory capabilities of edge computers.

I believe that many IoT applications can benefit from both edge and cloud capabilities, and that services should be carefully partitioned between the edge and the cloud based on their requirements. This presents a significant challenge to the community: to develop APIs and infrastructure for edge/cloud combinations that bring to developers the convenience

[1] `http://www.intel.com/content/www/us/en/internet-of-things/gateway-solutions.html`

that frameworks such as Apache Hadoop, Apache Spark, and Microsoft Orleans [13] have brought to cloud developers.

In the IoT, some services are critical to safety and cannot be beholden to remote services with highly variable latencies and potential network and service outages. Imagine a scenario where a power plant operator is locked out of the local network because a remote authentication service goes offline. This risk was also shown in the recent Google OnHub incident [70]. On February 23, 2017, many of Google's smart gateway (router) devices called *OnHub* failed for about 45 minutes because a failure in authentication servers caused users' Google accounts for OnHub devices to be disconnected, causing problems for all other smart devices connected to OnHub routers. Users could not even access local resources like printers on their LAN.

The Internet itself is also vulnerable. Recall the DNS Dyn cyberattack on October 21, 2016, which led to Internet connection problems to major websites including Twitter, Netflix, Spotify and the Financial Times [44]. These incidents show that depending on remote cloud servers over possibly brittle Internet connections can be risky and cannot be depended on for safety. Edge computers with adequate local authentication and authorization services can mitigate these risks, as shown in Chapter 6.

Chapter 3

Approach

This chapter presents the proposed approach, locally centralized, globally distributed authentication and authorization for the IoT. And I show how the approach addresses the challenges of the IoT security by introducing an open-source toolkit, *Secure Swarm Toolkit (SST)*, which is a concrete realization of the proposed approach as an authentication and authorization service infrastructure for the IoT. SST includes a local authorization software entity, *Auth*, its protocols, and software building blocks called *secure communication accessors* for developing IoT applications integrated with SST. This chapter also describes authentication and authorization processes of the proposed approach and how these processes achieve the goals.

3.1 Locally Centralized, Globally Distributed Infrastructure

I introduce a network architecture shown in Figure 3.1 using local authentication and authorization entities that I call *Auth* [55] to be deployed on edge-computing devices of any kind including mobile edge computers. Auth's open-source software implementation in Java is available on GitHub (`https://github.com/iotauth`). Auth provides authorization services for locally registered entities (IoT devices), while managing trust relationships with other Auths globally. I call this a *locally centralized* and *globally distributed* infrastructure [54]. Auth is now part of a toolkit for constructing an authorization infrastructure called the *Secure Swarm Toolkit (SST)* [53]. In Section 5.1, I show through formal analysis that Auth and SST satisfy fundamental security requirements.

3.1.1 Locally Centralized

As noted in Section 2.1, the locality of edge computing has advantages over globally centralized cloud computing in terms of security and privacy. Auth keeps credentials of registered devices locally because I expect local domain experts can better manage Auth and registered

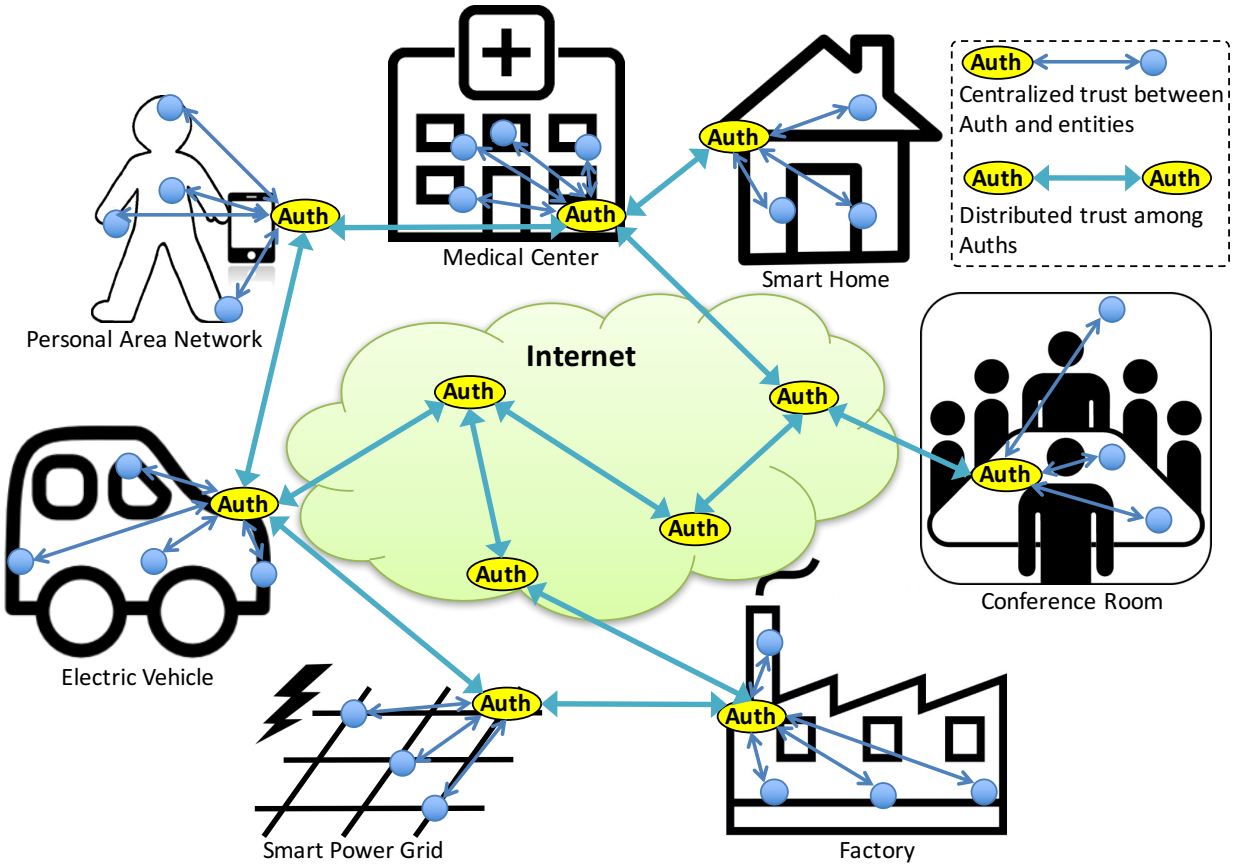


Figure 3.1: Locally centralized, globally distributed authorization service infrastructure using *Auth*.

devices. I assume that the network granularity of local IoT devices may vary depending on the network's nature; it can be a personal area network, a vehicle, or a building, for example. SST also includes software building blocks to program IoT applications accessing Auth and IoT services. These software building blocks encapsulate cryptographic keys and operations to help local system designers with only moderate knowledge in security.

Auth stores credentials and access policies of its locally registered entities in its database. The authorization process is achieved by distributing *session keys* which are cryptographic keys valid only for specific access activities. To serve different contexts composed of heterogeneous IoT devices, Auth provides a variety of security alternatives. For example, it can support multiple underlying communication protocols, including TCP and UDP over WiFi, and BLE. Auth allows resource-constrained devices to use longer-term, cached session keys with less power-hungry cryptography. Auth also supports secure one-to-many communication such as broadcasting or publish-subscribe by distributing the same session keys to more than two entities. The effect of different security configurations on energy consumption is

demonstrated in Section 5.3.

3.1.2 Globally Distributed

The trust relationship between Auths is globally distributed. The current implementation of Auth uses HTTPS for communication between Auths, based on certificates. These certificates are managed in a way that is similar to the web of trust in OpenPGP, trusting other Auths for signing certificates. When an entity needs to access other entities registered with another Auth, then the two Auths collaborate for authorization of their entities, leveraging the distributed trust. With this, I expect to achieve better scalability. A more detailed analysis of Auth's scalability is discussed in Section 5.2. Unlike certificates in PKI, Auths do not need to have a domain name or fixed network address, allowing edge devices without fixed network addresses to run Auth.

Distributed trust can make the entire system more resilient, limiting the impact of attacks even when an Auth gets compromised. The globally distributed architecture also provides ways to protect the internal network from the Internet, particularly by firewalling or physically disconnecting the external connection. This is possible because the local authorization service does not depend on an external authority. Even when an Auth becomes unavailable due to an attack or a failure, the resulting effect should be limited to the local authorization services. For further resilience, Auth can back up the authorization information of its registered entities to other trusted Auths and let the entities migrate to the trusted Auths for authorization in case of the Auth's failure.

3.2 SST: Secure Swarm Toolkit

This section presents an overview of SST, a concrete realization of the proposed approach, and discuss how SST addresses the main challenges stated in Section 1.2.

3.2.1 Open-source Local Authorization Entity, Auth

Figure 3.2 illustrates the network architecture for the SST infrastructure based on local authorization entities, *Auths* [55]. An *Auth* is a program to be deployed on edge devices [63] including smart gateways, responsible for authentication and authorization of locally registered entities. An open-source implementation of Auth is available on SST's GitHub repository (`https://github.com/iotauth`). Auth's implementation is written in a memory-safe language (Java), supports connectionless protocols such as UDP in addition to TCP, provides a diversity of security configurations, and uses a full-fledged but lightweight database, SQLite, with all credentials encrypted.

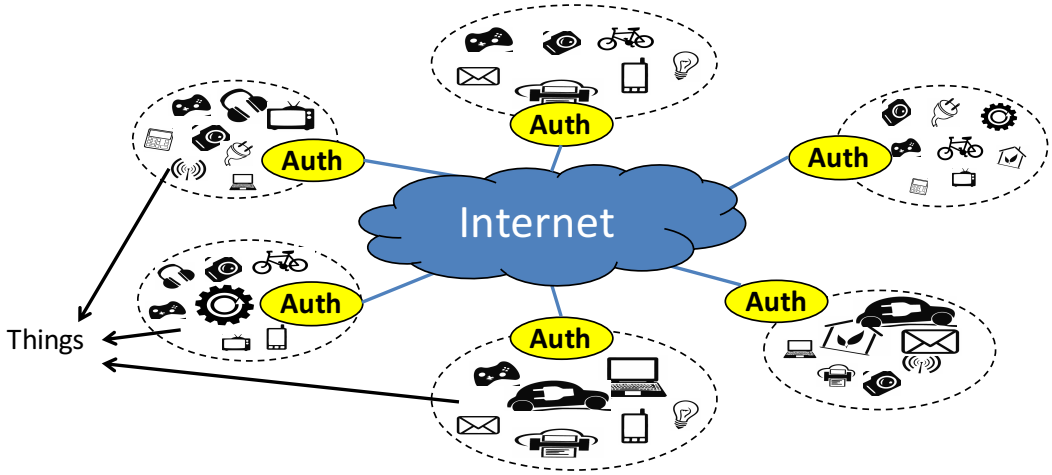


Figure 3.2: Network architecture of the SST infrastructure for the IoT based on local authorization entities, *Auths*.

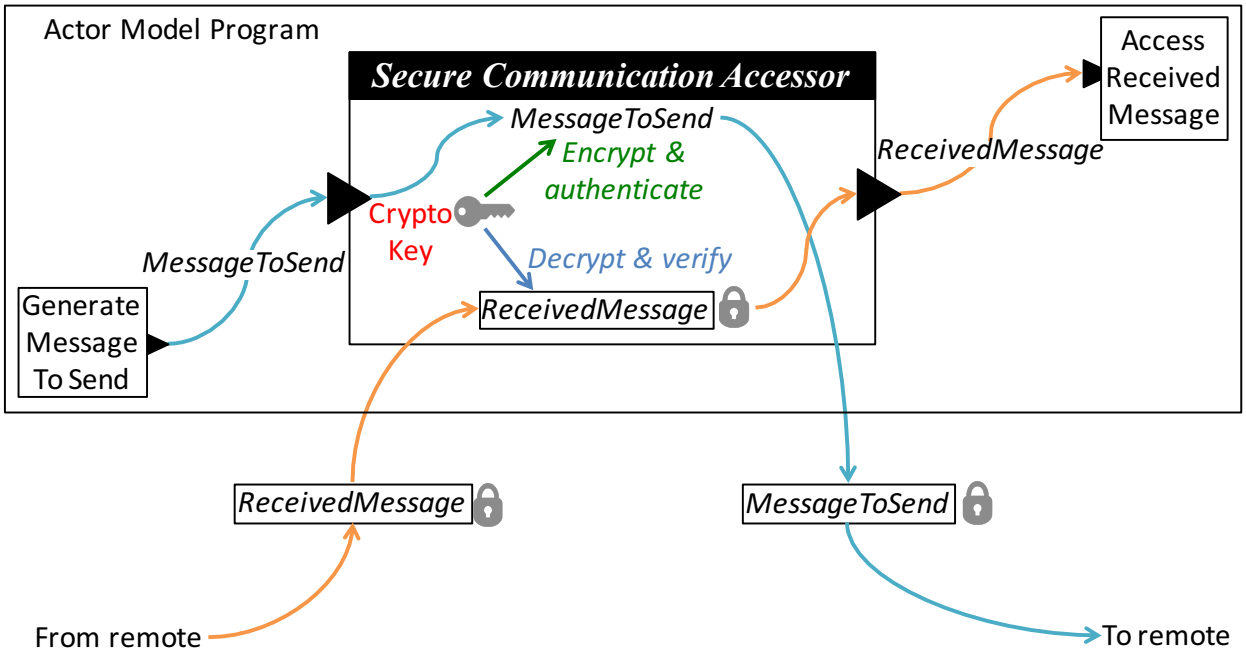


Figure 3.3: Software component for accessing authorization services, *secure communication accessor*.

3.2.2 Software Components for Accessing Authorization Services

I also propose actor-based software components for accessing the authorization services called *Secure Communication Accessors* shown in Figure 3.3. *Accessors*[1] [58] are actors [60] [43] 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.

3.2.3 How SST Works

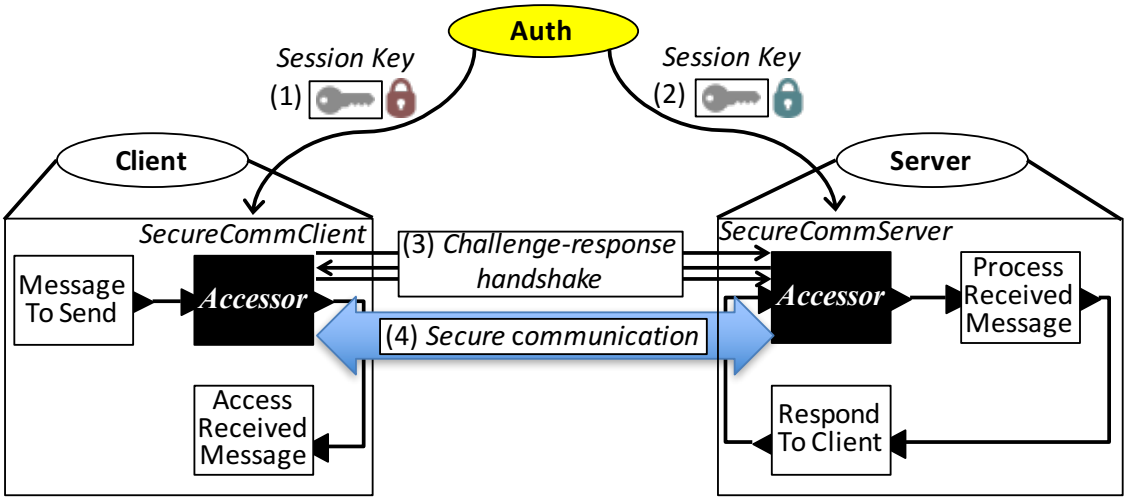


Figure 3.4: Process of building a secure connection between Client and Server.

Communications between the IoT entities in the proposed infrastructure are protected by symmetric cryptographic keys, called *session keys*. These keys are generated by Auth and distributed only to entities that are *authorized for access and communication*. For secure delivery of session keys, Auth and an entity share another symmetric key called a *distribution key*. Figure 3.4 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 4.3.

To build a secure connection, Client and Server must obtain a *session key* from Auth. In step (1) of Figure 3.4, Client receives a session key from Auth, encrypted and authenticated

[1] https://accessors.org/

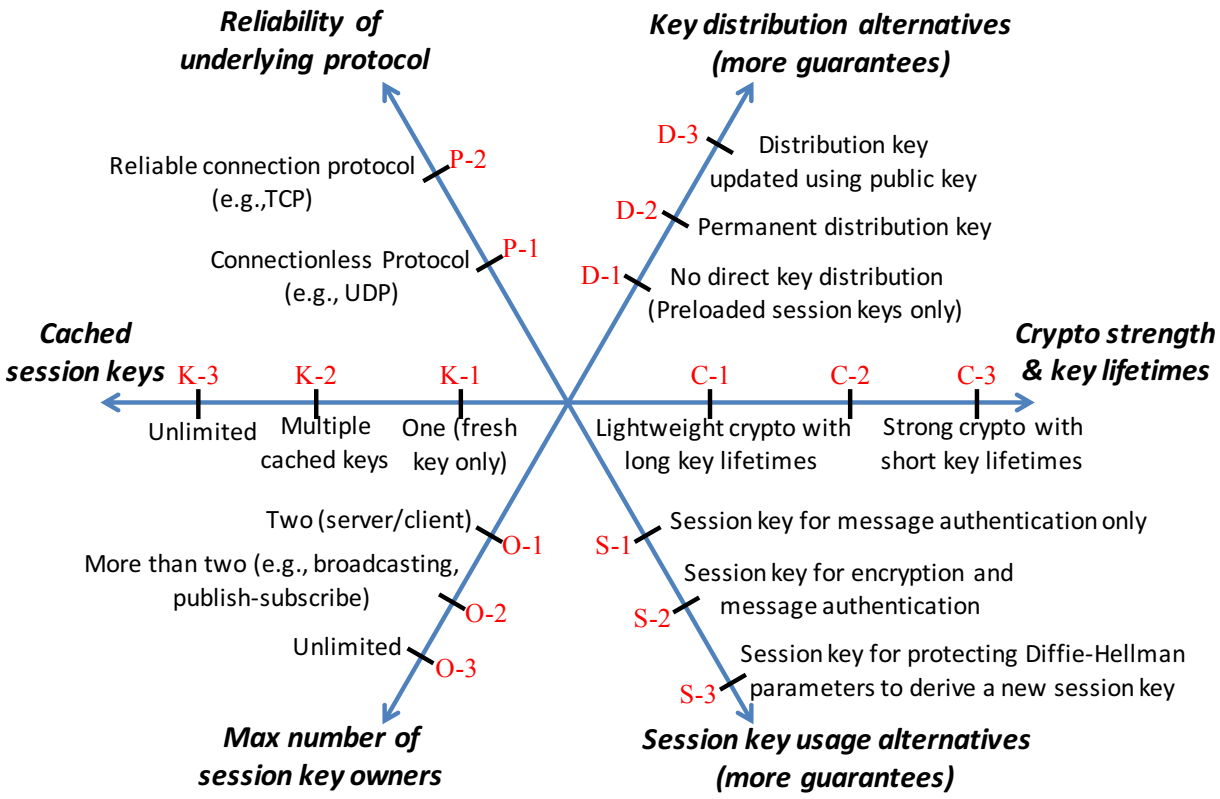


Figure 3.5: Security configuration space provided by SST.

with a *distribution key between Client and Auth*. Through step (2), Server receives the same session key encrypted and authenticated with another *distribution key between Server and Auth*. Details of (1) and (2) are described in Section 3.3.2. 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 3.3.3.

3.2.4 How SST Addresses Challenges

Heterogeneity

SST supports various security configurations, which can be used to achieve trade-offs between security guarantees and resource usage. Figure 3.5 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), the number of cached session keys (K-1, K-2, K-3), and reliability of the underlying protocol (P-1, P-2). An example of different cryptography

Table 3.1: Example security configuration profiles.

Configuration \ Profile	High-risk safety-critical	Resource-constrained	Sensitive information	Broadcasting
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

strengths is AES ciphers with different key sizes.

Table 3.1 shows sample profiles using different configuration alternatives. For a *safety-critical* entity in a *high-risk* environment, we can enforce short-term keys to limit the damage when the entity 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 3.3.3). For *broadcasting* devices, we can allow an unlimited number of devices sharing the same session key.

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) [19] can be deployed in combination with Auth, leveraging the fact that all traffic relevant to authorization is directed through Auth.

Scalability

The scalability problem is twofold: (1) how to handle a large number of entities and (2) how to handle increased data traffic. The proposed approach addresses the first problem by allowing multiple Auths to be deployed on a network. To show how this works, I use a simple operation example described in Figure 3.6, 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 5.2 shows the proposed approach's scalability.

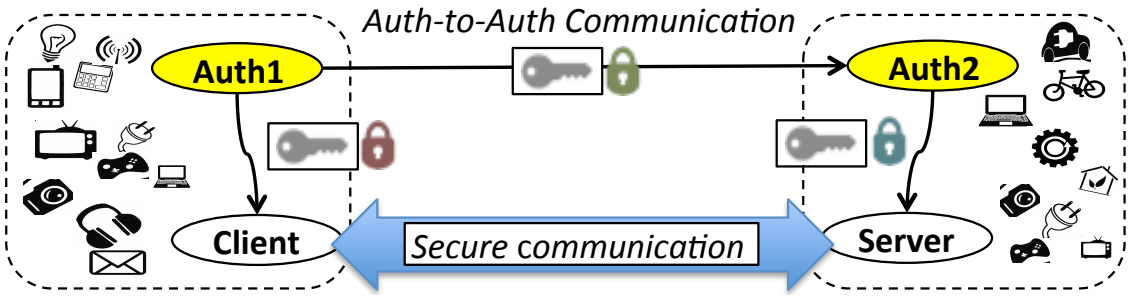


Figure 3.6: Operation example of communication between Client and Server registered with two different Auths, Auth1 and Auth2, respectively.

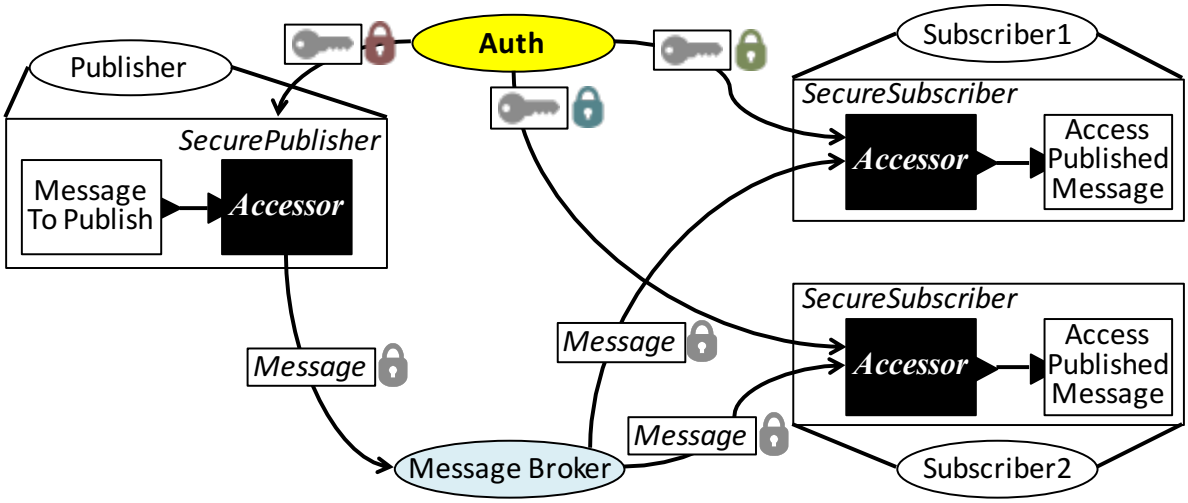


Figure 3.7: Process of scalable key sharing for publish-subscribe communication.

To handle increased data traffic, the proposed infrastructure supports shared session keys for one-to-many communication patterns. Figure 3.7 describes an example of secure publish-subscribe communication in SST. For creating Publisher and Subscriber programs, an IoT application developer can use corresponding accessors, *SecurePublisher* and *SecureSubscriber*. Publisher and two Subscribers, Subscriber1 and Subscriber2, are first 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 3.3.3 and evaluated in Section 5.3.2.

3.3 Authentication and Authorization

3.3.1 Entity Registration Process

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[2] during the registration or shipped by the manufacturer[3] with the entity.

If an entity is capable of performing public-key cryptography operations to update[4] its *distribution key* (explained in Section 3.2.3), 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 directly obtain any new 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.

Auth stores the basic information and initial credentials in Auth's database, specifically, in the *registered entity table* which is explained in detail in Section 4.1.2 with Table 4.1. If the access policy of the newly registered entity is not already in Auth's database, the access policy should be inserted to Auth's *communication policy table* which is described in detail in Section 4.1.2 with Table 4.2.

3.3.2 Auth – Entity Communication

Auth authorizes entities to communicate with each other by distributing a session key shared by entities. Figure 3.8 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 that an old message of step (4) cannot be reused in

[2] Generation of credentials (cryptographic keys) can be done using tools such as OpenSSL command line tools.

[3] This is becoming more common for IoT devices that need credentials for cryptography operations.

[4] 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.

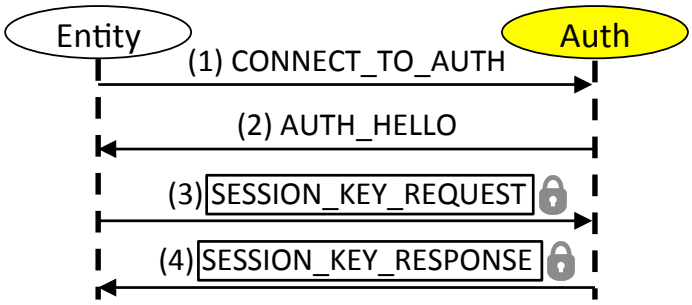


Figure 3.8: Steps for *Auth – Entity* communication for session key distribution; a padlock next to a message indicates that the message is encrypted and/or authenticated.

replay attacks. (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 an existing distribution key using public-key cryptography, (3) must be authenticated (digitally signed) 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* (Auth's database table storing access policies, details are in Section 4.1.2 with Table 4.2) 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, the new distribution key 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, also 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 the entity does not get a response from Auth, it repeats the message after timeout, TO_{Entity}. Auth keeps the N_{Auth} and (4) SESSION_KEY_RESPONSE each UDP port until Auth's timeout TO_{Auth} so that it can respond to the entity when a message is requested again. $TO_{Auth} \gg TO_{Entity}$ should

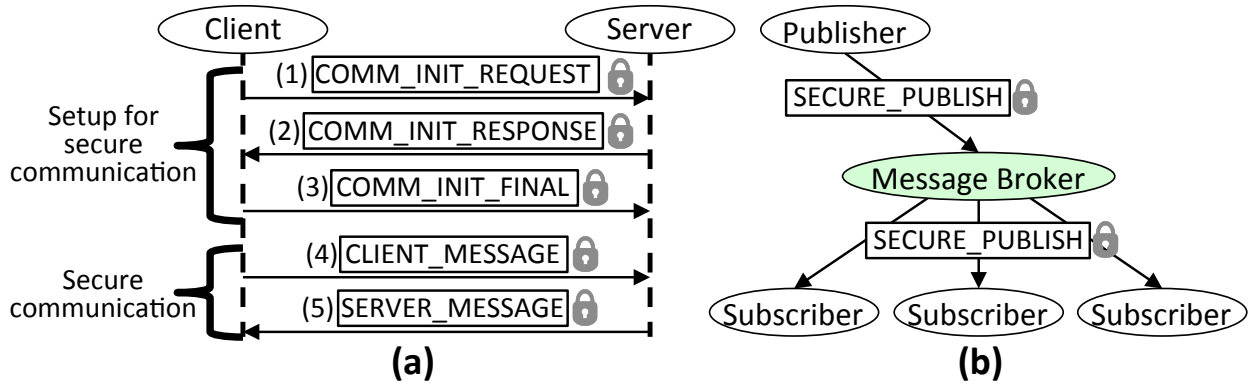


Figure 3.9: Process of secure communication for (a) Server-client (b) Publish-subscribe.

hold to make sure the entity can retry multiple times. After TO_{Auth}, the nonce and message are destroyed.

If Auth detects anything wrong or suspicious during the authorization process, it sends $AUTH\_ALERT$ message to the entity. For example, use of an expired distribution key or a session key request violating communication policies will result in an $AUTH\_ALERT$ message to the requesting entity.

3.3.3 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 3.9 describes two ways of secure communication provided by the SST infrastructure.

Figure 3.9 (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 challenge-response by performing cryptographic operations on random nonces in steps (1) to (3). This process is similar to the PSK Key Exchange Algorithm of Pre-shared key (PSK) cipher suites for TLS [34]. For identification of the session key, Auths 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 [34], which provides additional protection such as Perfect Forward Secrecy (PFS).

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 num-

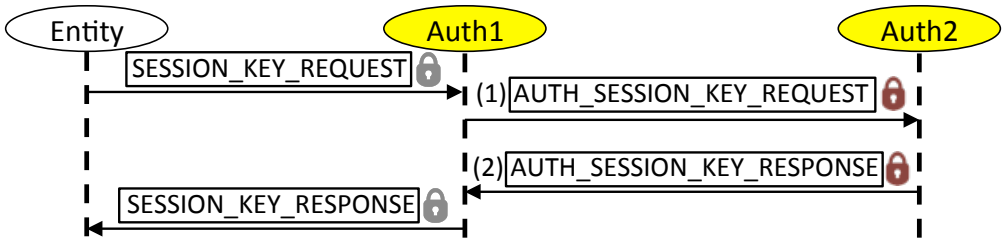


Figure 3.10: Steps for *Auth* – *Auth* communication.

bers 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. The proposed approach supports both TCP and UDP for this server-client communication.

Figure 3.9 (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 the 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 [83].

3.3.4 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 3.10 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 3.3.3, 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. Further details of this procedure are illustrated in Section 4.2.5.

Chapter 4

Design and Implementation

This chapter focuses on the design and implementation of the proposed approach. These include design considerations of necessary components for the proposed approach and how the design is realized through a concrete implementation. Specifically, this chapter describes the database design of the local authorization entity, Auth, and design of messages used in communication throughout operation processes of the proposed toolkit, SST. I also illustrate the design and implementation of APIs (or software building blocks) for integration of IoT application with the proposed toolkit.

4.1 Auth

In this section, I describe the design and implementation of the local authorization entity of SST, *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, registered with different Auths. Auth makes its authorization decisions based on a database of access policies and configurations.

4.1.1 Key Words and Overall Operation Phases

The proposed infrastructure uses a local authorization entity, called *Auth*, to automatically handle frequent authentication and authorization of IoT devices. For scalability and resource constraints, the proposed approach uses symmetric keys for authentication and authorization rather than asymmetric keys. Before diving into the details of the design and implementation of the proposed toolkit, I clarify some important terms used throughout this chapter.

- **Entity** – Any device connected to the network in the IoT to be authenticated and authorized. Each entity has a unique identifier and cryptographic keys for secure communication.

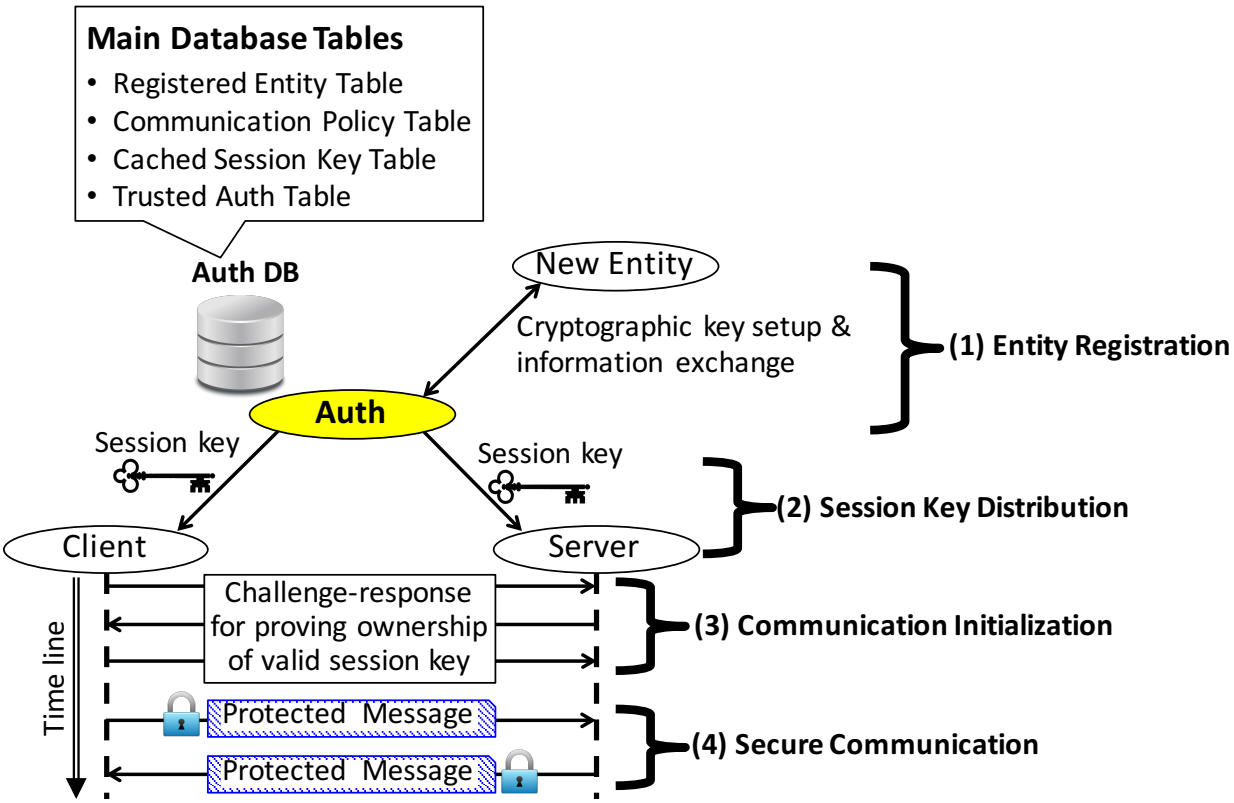


Figure 4.1: Overview of four operation phases and Auth database of SST (Secure Swarm Toolkit).

- **Auth** – An entity responsible for authenticating and authorizing registered entities. Auth maintains and manages database tables to store information for authorization of entities.
- **Client** – An entity that initiates communication.
- **Server** – An entity accepting communication requests.
- **Public key** and **Private key** – The public and private components of entity's asymmetric key pair.
- **Distribution key** – A symmetric key-wrapping key used to encrypt session keys for distribution.
- **Session key** – A symmetric key used to protect a single session of communication. Since only authorized entities receive a valid session key, an entity can prove that it is authorized, by proving ownership of the session key. Each session key is assigned a

unique session key ID and its validity periods. A session key plays a similar role as a ticket in the Kerberos authentication system [75].

The operations of the proposed approach can be divided into four phases, (1) *entity registration*, (2) *session key distribution*, (3) *communication initialization*, and (4) *secure communication*, as shown in Figure 4.1. A newly added entity must be registered with at least one Auth during the entity registration phase. Registered entities can obtain session keys through session key distribution. The dotted lines below the client and server describe the time line of communication initialization and secure communication. The following sections explain the roles of Auth and details of each phase.

4.1.2 Auth Database Design

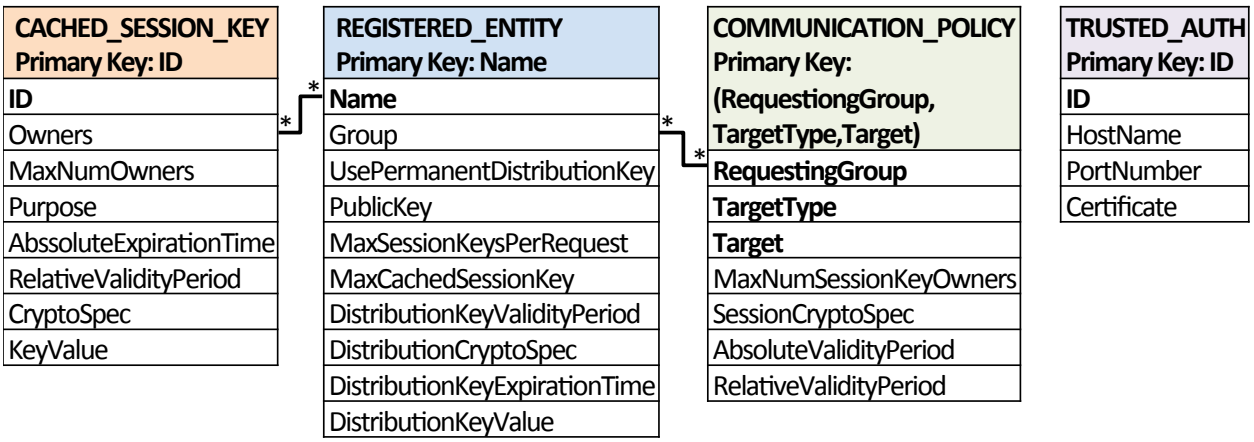


Figure 4.2: Auth database table schema. (* for many-to-many relationship)

The Auth entity allows authentication and authorization through distributing session keys to entities that are valid for communication. For session key distribution, Auth stores various information in its database, as shown in Figure 4.2. The database tables include:

- **Registered entity table** (Table 4.1): Stores information about entities registered with the Auth, including its credentials (cryptographic keys) and the configuration related to key distribution.
- **Communication policy table** (Table 4.2): 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** (Table 4.3): 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** (Table 4.4): Stores information and credentials for other trusted Auths, including each Auth's unique *ID*, network address, port, and certificate.

Figure 4.3, Figure 4.4 and Figure 4.5 illustrate examples of database tables maintained and managed by Auth.

Registered entity table

Name	Group	Public key	Distribution Key <Value, Expiration Time>	Distribution Key Validity Period	Distribution Crypto Spec		Operational Conditions
					Cipher Algorithm	MAC Algorithm	
EV001	Electric Vehicle	8c 98 7c ...	<28 3d ef ..., 9/1/16,16:00>	10 days	AES-128 -CBC	SHA256	Max session keys:30, Session keys/req:5, ...
CS003	EV Charging Station	30 8b 3b ...	<da 9f f1 ..., 8/23/16,4:00>	12 hours	AES-256 -CBC	SHA384	Max session keys:5, Time precision: 100us,...
FC005	Fuel Cell Storage	2c 4a b1 ...	<d3 28 3d ..., 9/21/16,16:00>	30 days	AES-128 -CBC	SHA384	Max session keys:10, Session keys/req:10,...
CS024	Current Sensor	N/A	<ef c0 20 ..., 8/22/18,16:00>	2 years	AES-128 -CBC	SHA256	Max session keys: 10, Session keys/req: 5,...
•••	•••	•••	•••		•••		•••

Figure 4.3: Example of Auth's registered entity table.

The registered entity table in Figure 4.3 stores entity-specific information such as an entity's unique name, group, public/distribution key, and conditions for key distribution and operations. The key distribution conditions determine the cipher/MAC algorithm and validity period for the distribution key. The distribution key field has the key's value and when it expires. The validity period or cryptoperiod of cryptographic keys including the distribution key can be determined depending on risk factors as recommended in [9]. Risk factors include the cryptography strength, operating environment, number of transactions, and potential threats.

In the example table Figure 4.3, an EV charging station named CS003, has the shortest validity period for distribution keys. This is because the charging station operates in an open space, requires a relatively large number of session keys to interact with many vehicles, and causes critical damage to the power grid if compromised. An electric vehicle named EV001 has a longer validity period because it interacts with others less frequently and it is less accessible (e.g., protected by locks/garage doors). A fuel cell storage, FC005 has an even longer cryptoperiod, since it operates under restricted access.

The distribution key can be updated using public key cryptography, thus, Auth stores public keys of the registered entities as in Figure 4.3. The proposed approach also supports devices incapable of public key cryptography such as the (electric) current sensor named CS024. In this case, Auth stores distribution key during the entity registration phase and

Table 4.1: Registered entity table fields.

Field	**Explanation**
Name	Entity's unique name in string
Group	Entity's group in string for communication policy
DistributionProtocol	Network protocol used for communication with Auth (e.g, TCP or UDP)
UsePermanentDistributionKey	Boolean value that is true if the entity uses permanent distribution key without updates, false if the entity's distribution key is updated using public key cryptography
MaxSessionKeysPerRequest	Maximum number of session keys to be delivered to the entity for each request
MaxCachedSessionKey	Maximum number of valid session keys that the entity can cache in advance
PublicKey	Public key of the entity null if the entity uses permanent distribution key
DistributionKeyValidityPeriod	Validity period of one distribution key
DistributionCryptoSpec	Cryptography specification of the key distribution communication (e.g., cipher/MAC algorithm)
DistributionKeyExpirationTime	Expiration time of the current distribution key
DistributionKeyValue	Binary value of current distribution key in use for the entity, encrypted with Auth's database key

uses it for the entire life cycle of the device. Sections 4.2.1 and 4.2.2 explain the details of this. The operational conditions include the maximum number of total cached session keys, a maximum number of session keys per request, and the entity's time precision required for time synchronization.

Communication policy table

The communication policy table in Figure 4.4 is used to determine the conditions of communication between certain entities such as cipher/MAC algorithms and validity periods. A client (or communication initiator) and a server (or connection listener) can also be specified as a group to set a communication policy between groups of entities. With this table, Auth has a full control over authentication/authorization of entities, and the communication policies can be dynamically managed.

The cipher/MAC algorithms used for communication can be determined by various criteria, including security requirements (e.g., confidentiality, integrity), available resources of entities, and performance/costs of algorithms on entities. A session key validity period is a tuple of two values, *absolute validity period* and *relative validity period*. The absolute validity

Requesting Group (Client or Publisher)	Target Type	Target (Server Group or Pub-Sub Topic)	Session Crypto Spec		Session Key Validity Period <Absolute, Relative>
			Cipher Algorithm	MAC Algorithm	
Electric Vehicle	Server-Client	EV Charging Station	AES-128-CBC	SHA256	<2 days, 30 min>
EV Charging Station	Server-Client	Fuel Cell Storage	AES-256-GCM	SHA384	<1 hour, 5 min>
Current Sensor	Publish-Subscribe	Current Sensor Value	RC4-128	SHA1	<7 days, 1 day>
•••		•••	•••	•••	•••

Figure 4.4: Example of Auth's communication policy table.

Table 4.2: Communication policy table fields.

Field	Explanation
RequestingGroup	The group of entities requesting the communication
TargetType	The communication target's type, it can be another group of entities or publish-subscribe topic
Target	The communication target, a group or a topic
MaxNumSessionKeyOwners	Maximum number of entities that can share the same session key, two for a server/client model, more than two for publish-subscribe
SessionCryptoSpec	Cryptography to be used for communication
AbsoluteValidityPeriod	How long the session key is valid after it is generated by the Auth
RelativeValidityPeriod	How long the session key is valid after it is first used

period specifies how long a session key is valid after creation and the relative validity period denotes the validity of the session key after its first use for communication.

Cached session key table

Once a session key is generated, Auth stores it in the cached session key table in Figure 4.5. Auth assigns each session key a unique identifier, called session key ID. When a session key is distributed to entities, Auth updates its owners, the entities who share the session key. The *Expires on* and *Valid for* fields of each session key are set according to the communication policy table. Details of each field in the cached session key table are explained in Table 4.3.

Session Key ID	Session Key Value	Crypto Spec		Owner(s)	Expiration Time (Absolute Validity)	Relative Validity Period
		Cipher Algorithm	MAC Algorithm			
212371	42 d5 49 1c ...	AES-128-CBC	SHA256	EV001	8/24/2016, 16:00	30 min
212372	0a de d6 90 ...	AES-128-CBC	SHA256	EV001	8/24/2016, 16:00	30 min
212373	56 e2 f8 1a ...	AES-256-GCM	SHA384	FC005	8/22/2016, 17:00	5 min
212374	37 6b 1a b9 ...	RC4-128	SHA1	CS024	8/29/2016, 16:00	1 day
•••	•••	•••	•••	•••	•••	•••

Figure 4.5: Example of Auth's cached session key table.

Table 4.3: Cached session key table fields.

Field	**Explanation**
ID	Unique ID of the session key, encoded with the ID of the Auth who generated it
Owners	Names of the entities who have the session key
MaxNumOwners	Maximum number of entities that can have the key
Purpose	Purpose of the key, set up during session key generation, used to determine whether it can be given to a certain entity
ExpirationTime	Expiration time of the session key no matter it is actually used or not, set up when the key is generated
RelativeValidityPeriod	Validity period after first use of the session key, the entity who starts using this key will discard the key after this time period
CryptoSpec	Cryptography to be used with the key
KeyValue	Binary value of the key

Table 4.4: Trusted Auth table fields.

Field	**Explanation**
ID	Unique ID of the trusted Auth, integer value
Host	Network address of the trusted Auth
Port	Port number of the trusted Auth
Certificate	Current certificate value of the trusted Auth

Trusted Auth table

The trusted Auth table contains information about other trusted Auths. This includes the trusted Auths' unique identifiers, credentials (a public key or certificate), and connection information (a host name, port, and protocols). Important fields of the trusted Auth table are shown in Table 4.4.

4.1.3 Scalability and robustness of Auth

SST supports multiple Auths, making the proposed approach more scalable. When entities registered with different Auths want to communicate, each entity just needs to contact with its own Auth for authorization. Auths communicate with one another to deliver the same session key to their entities for setting up a secure connection. This process is explained in detail with an example in Section 4.2.5.

Note that Auth is a logical entity that can be implemented in a variety of ways, like the controller of a software-defined network (SDN). It is a logical entity and can be implemented in a distributed way [29]. I expect that Auth can also be implemented in a distributed manner, possibly with replicas. Such Auth implementations can avoid being a single point of failure, providing robustness against denial of service attacks. Also note that some Auths must be in a safe place where only valid users can access, such as general servers in data centers, for an additional layer of security. The architectural advantage for resilience and robustness is shown through the proposed secure migration technique in Chapter 6.

4.2 Protocol Design and Operation Phases

This section illustrates the design and implementation of the communication protocol used in SST. Specifically, message-level details are explained for the communication introduced in Section 3.3, including description and requirements of components in messages and exchanged credentials, following the operation phases shown in Figure 4.1.

4.2.1 Entity Registration Phase

In the entity registration phase, Auth and a newly added entity exchange information for session key distribution. There are two possible options for the distribution key, *updatable distribution key* and *permanent distribution key*, depending on whether the distribution key can be updated using public key cryptography. Figure 4.6 shows types of exchanged data during the entity registration and their security requirements.

When the distribution key of the new entity is *updatable*, Auth and the entity should exchange their public keys for the secure delivery of the distribution key from Auth to the entity. The public keys must not be tampered with during the exchange to prevent an adversary from spoofing the identity of Auth or the new entity, although the public keys need not be confidential. If the distribution key of the new entity is *permanent*, meaning that

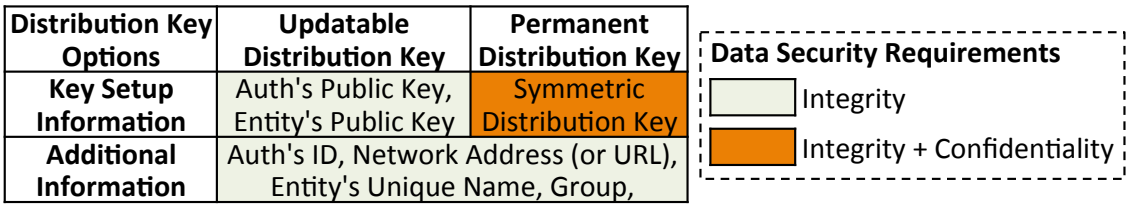


Figure 4.6: Exchanged information during entity registration phase, and data security requirements for different types of data.

a single distribution key is to be used for the entire life cycle of the entity, the distribution key must be kept confidential, only known to Auth and the entity.

For both options, Auth and the new entity should exchange additional information, including the new entity's unique name and group, and the Auth's ID and network address or URL (can be more than one). Data integrity must be guaranteed for the additional information exchanged to prevent masquerading. After the entity registration, Auth stores the entity's information in its registered entity table. The new entity stores Auth's information in its local storage. If the Auth is replicated, then the entity stores information of Auth's replicas as well.

As long as the specified data security requirements are satisfied, many methods can be used for the entity registration. An authorized person can plug a new device or use Near Field Communication (NFC) to securely connect to Auth or Auth's delegate that is connected to Auth via a secure connection, for information exchange. For personal devices, for example, two-factor authentication [3] can be used to guarantee the authenticity of the information from the device.

4.2.2 Session Key Distribution Phase

Auth delivers one or more session keys to an entity during the session key distribution phase. An entity can request multiple session keys a priori for future use. Hence, an entity with intermittent connectivity can authenticate and authorize itself even without direct connection to Auth. A resource-constrained entity can save communication and computation to obtain session keys. Session key distribution works on top of TCP/IP to ensure reliable message delivery. Figure 4.7 illustrates the session key distribution process for two cases, depending on whether there is an available distribution key.

When an entity already has a valid distribution key as in Figure 4.7 (a), the distribution key is used for the session key distribution. When a TCP/IP connection is established, Auth sends AUTH_HELLO, including Auth's information such as its ID, and a nonce (random number) generated by Auth. Upon receiving AUTH_HELLO, the entity sends SESSION_KEY_REQ, which specifies the requesting entity's name, communication purpose including the target of communication, the number of requested keys, and the session key's identifier, if necessary. The nonce from Auth and the entity's own nonce are appended to

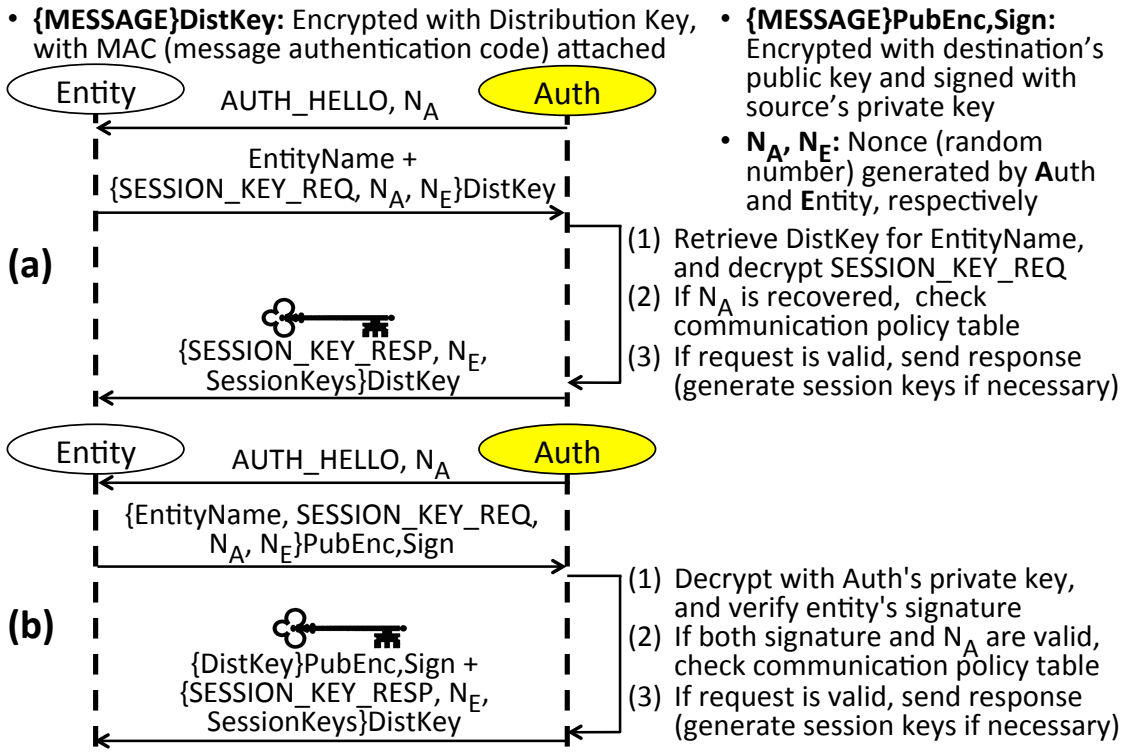


Figure 4.7: Two cases of session key distribution phase (a) when the distribution key is available, and (b) when the distribution key needs to be updated.

SESSION_KEY_REQ, to prevent replay attacks. The whole message is encrypted with the distribution key after attaching MAC (message authentication code). The entity's unique name is also sent in plain text.

Upon receiving the entity's request, Auth retrieves the distribution key using the entity's name, checks the Auth's nonce and the communication policy table for the entity's eligibility, and sends the response if the request is valid. Auth generates session keys before sending the response if it is necessary. The Auth's response, SESSION_KEY_RESP includes the entity's nonce, communication policy and session keys, encrypted by the distribution key. After receiving SESSION_KEY_RESP, the entity checks the nonce and stores the session keys.

Figure 4.7 (b) describes the case when the distribution key should be updated, after registering a new entity or expiry of a distribution key. In this case, public key cryptography is used to deliver the distribution key securely. The entity's request, SESSION_KEY_REQ along with the entity's name and the nonces should be encrypted with Auth's public key and signed with the entity's private key. When Auth receives the request, it decrypts the request with its private key and verifies the signature using the entity's public key. If the signature and the nonce are valid, Auth checks the request's eligibility and responds. The response includes the distribution key, *DistKey*, encrypted with the entity's public key and signed with Auth's private key. The response also contains SESSION_KEY_RESP, a list of

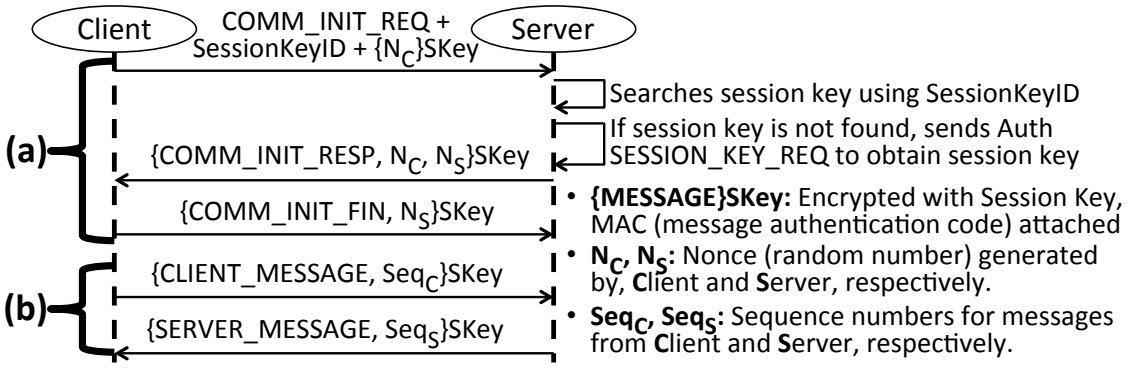


Figure 4.8: (a) Communication initialization phase, followed by (b) secure communication phase.

session keys and nonces, encrypted with DistKey.

4.2.3 Communication Initialization Phase

After obtaining a session key, a client can initiate communication with a server. The main objective of this phase is proving ownership of the session key, since the ownership indicates the entity is authenticated and authorized to communicate. Although there are many different ways to prove the ownership, I choose a simple challenge-response handshake, where each entity shows its ability to perform cryptographic operations on randomly generated numbers (nonces) by its counterpart. The random nonces are used to prevent replay attacks. Figure 4.8 (a) shows the operations of the communication initialization phase between the client and server.

The communication initialization phase begins with the client's COMM_INIT_REQ with SessionKeyID, a unique identifier for the session key. Note that the client must send the communication initialization request's header and SessionKeyID in clear text, with the client's nonce and its message authentication code (MAC) encrypted by the session key.

Upon receiving COMM_INIT_REQ, the server searches for the session key using SessionKeyID in its cache. The server finds the session key if it is already cached. Otherwise, the server sends SESSION_KEY_REQ to Auth with SessionKeyID specified to obtain the session key. After decrypting COMM_INIT_REQ, the server sends COMM_INIT_RESP with the client's nonce and the server's nonce encrypted with the session key. The client receives COMM_INIT_RESP, decrypts it, and compares the nonce in it against the nonce generated by the client. If the two nonces match, the server is verified to have the session key. In the same way, the client sends COMM_INIT_FIN with server's nonce encrypted, and the server verifies the client's ownership of the session key. If either the client or the server is unable to match nonces, communication initialization fails.

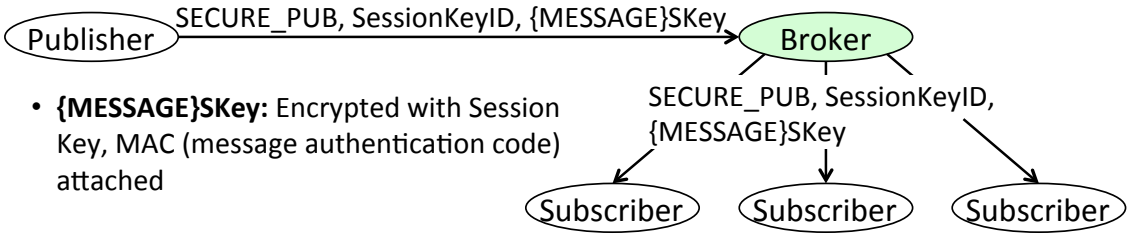


Figure 4.9: Alternative secure communication phase for publish-subscribe protocols.

4.2.4 Secure Communication Phase

After the client and server initialize communication, they can exchange encrypted messages as shown in Figure 4.8 (b). This works almost the same as the TLS record layer with application data after TLS handshake. Each message is assigned a sequence number, starting from 0 for the first message, to prevent replay attacks. Every message has MAC attached, encrypted with the symmetric session key.

The proposed approach also provides an alternative way of secure communication phase for publish-subscribe protocols such as MQTT, as depicted in Figure 4.9. The packet for this method of secure communication, SECURE_PUB, includes its header and SessionKeyID in clear text. Any entity with the session key specified by SessionKeyID is authorized to decrypt the encrypted messages. The sender only needs to encrypt and send the message once to all receivers, thus, this scales very well together with one-to-many communication such as broadcasting and publish-subscribe patterns.

Lagutin *et al.* [57] introduce other various ways to secure a publish-subscribe network architecture using certificates, including packet level authentication (PLA). Such methods require an entity to either carry large certificates or perform expensive asymmetric key operations for published messages. Compared to these approaches, the proposed approach can significantly reduce the overhead to secure publish-subscribe by using a small and lightweight symmetric session key.

When the published or broadcasted data does not require confidentiality, the TESLA [83] protocol can be used to guarantee data integrity and authenticity for message receivers. The proposed approach can be integrated with TESLA, in a way that Auth establishes an authentication key for a sender and discloses the key to receivers after key disclosure delay.

4.2.5 Scaling to Multiple Auths

The proposed approach can also support authentication/authorization between entities that are registered with different Auths. I illustrate this with an example shown in Figure 4.10. In this example, a client registered with Auth1 communicates with a server registered with Auth2. Each entity is authorized by its own Auth, that is, the client and the server receive a session key from Auth1 and Auth2, respectively. Auth1 and Auth2 are connected via a

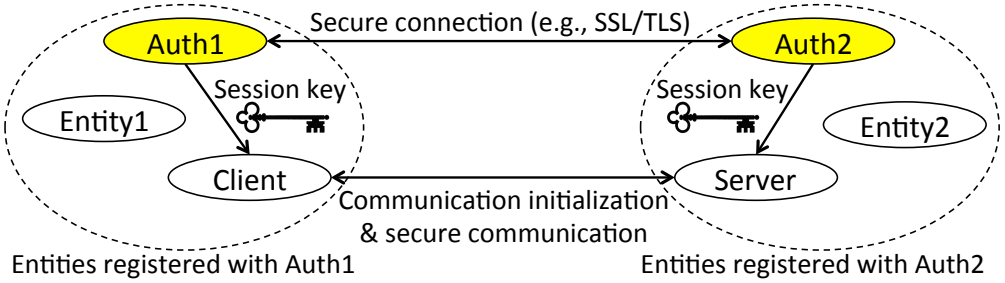


Figure 4.10: An example where a client and a server that are registered with two different Auths initialize a secure communication.

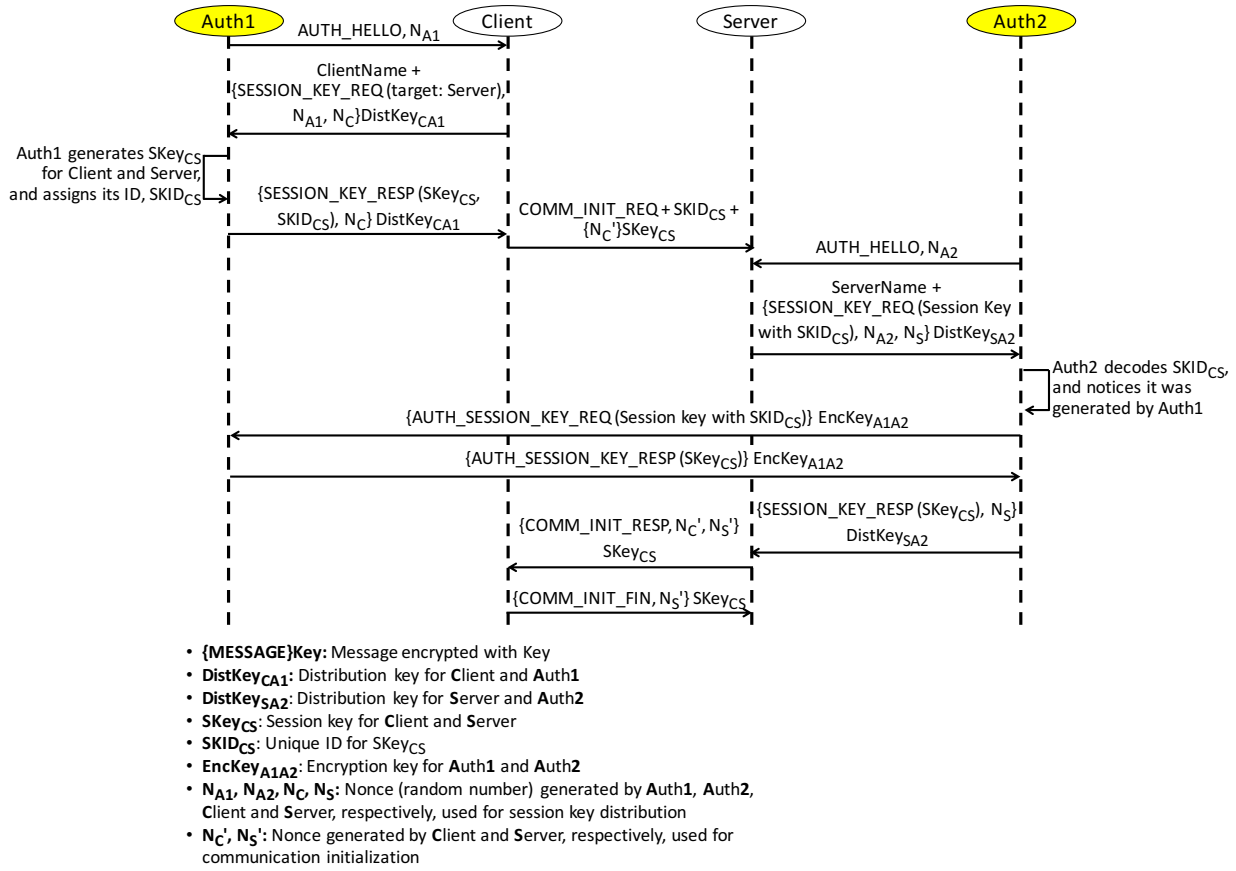


Figure 4.11: Details of the example authorization process of a client and a server that are registered with two different Auths. (Continued from Figure 4.10.)

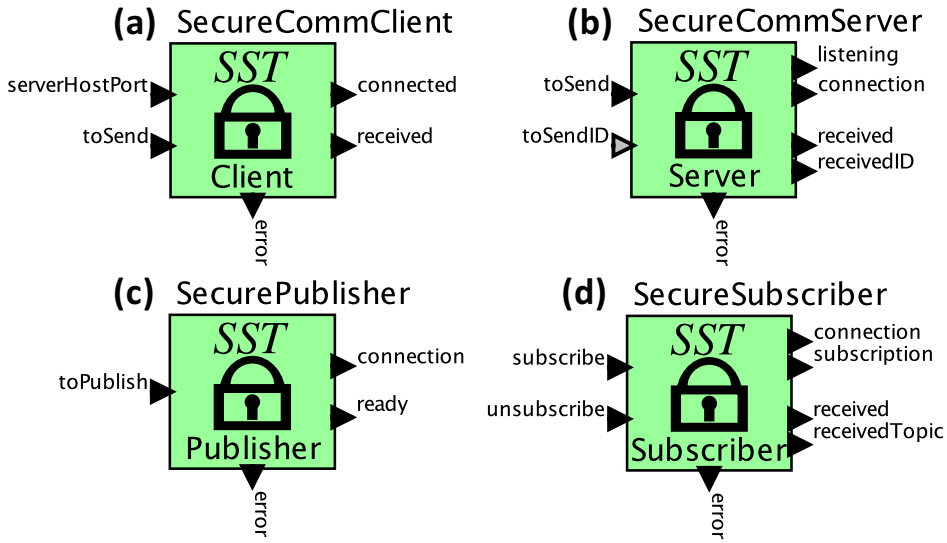


Figure 4.12: Secure communication accessors of SST.

secure connection such as SSL/TLS. They work together to distribute the same session key for the client and server.

Figure 4.11 describes the detailed authorization process of this example. The client sends SESSION_KEY_REQ to Auth1, specifying the server as a target of communication, as explained in Section 4.2.2. Auth1 responds to the client's request with a session key, Skey_{CS}, and its unique ID, SKID_{CS}. SKID_{CS} is encoded with information of the session key's generator, in this case, Auth1. Using Skey_{CS} and SKID_{CS}, the client sends COMM_INIT_REQ to the server for initialization of a secure communication, as explained in Section 4.2.3.

To continue communication initialization, the server requests its own Auth, Auth2, for a session key specified by SKID_{CS}. Auth2 decodes SKID_{CS} and finds that the requested session key was generated Auth1. Auth2 sends AUTH_SESSION_KEY_REQ (a session key request between Auths) to Auth1 specifying the session key ID, SKID_{CS}, via a pre-established secure connection such as SSL/TLS. Auth1 replies to Auth2 with AUTH_SESSION_KEY_RESP, which includes Skey_{CS}. Auth2 delivers Skey_{CS} to the server, and the server continues to set up a secure connection with the client.

4.3 Secure Communication Accessors

For constructing a secure swarm applications, I provide four secure communication accessors for accessing authorization services, *SecureCommClient*, *SecureCommServer*, *SecurePublisher*, and *SecureSubscriber* as shown in Figure 4.12. 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 accessor's library (`https://accessors.org/library/`) under *net* group.

SecureCommClient (Figure 4.12 (a)) establishes a secure connection with *SecureCommServer* (Figure 4.12 (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 [8] 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.

4.3.1 APIs for Accessor Modules

```
1   // Functions for communication with Auth
2   function sendSessionKeyRequest(options,
        sessionKeyResponseCallback, callbackParameters)
3   function sessionKeyResponseCallback(status, distributionKey,
        sessionKeyList, callbackParameters)
4   // Functions for server/client communication
5   function initSecureCommunication(options, eventHandlers)
6   function initSecureServer(options, eventHandlers)
7   // Functions for publish-subscribe
8   function encryptSecurePub(message, cryptoSpec, sessionKey)
9   function getKeyIDOfSecurePub(rawData)
10  function decryptSecurePub(encrypted, cryptoSpec, sessionKey)
```

Figure 4.13: JavaScript APIs used to implement secure communication accessors.

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

To construct secure communication accessors, I define APIs (Application Program Interfaces) shown in Figure 4.13. These APIs are in the form of JavaScript functions based on a callback structure.

Common Interfaces For Communicating with Auth

```
1     options = {
2       authHost,
3       authPort,
4       entityName,
5       numKeysPerRequest,
6       purpose,
7       distributionKey = {value, absoluteValidity},
8       distributionCryptoSpec,
9       publicKeyCryptoSpec,
10      authPublicKey,
11      entityPrivateKey
12    }
```

Figure 4.14: Options for `sendSessionKeyRequest` function in Figure 4.13.

For any entity registered with Auth, I provide an interface including a function for sending a session key request to Auth and a callback that is triggered when a session key response is received from Auth. The function `sendSessionKeyRequest` (line 2 of Figure 4.13) triggers a session key request to Auth. The parameter `options` is consisted of information specifying the session key request as shown in Figure 4.14. These options include Auth connection information (Auth's hostname, port, protocol), entity's unique name, credentials, security configurations, the purpose of the request, the number of session keys to request, etc. Other parameters are a callback function, `sessionKeyResponseCallback`, and additional parameters transferred to the callback, `callbackParameters`.

When a response is received from Auth, a callback function `sessionKeyResponseCallback` (line 3 of Figure 4.13) is called. The parameter `status` indicates whether the session key request succeeded and also an error message if it failed. `distributionKey` is set if a new distribution key was included in the respond or null otherwise. `sessionKeyList` includes new session keys from Auth. Further behaviors can be specified using additional `callbackParameters` such as connecting to a server right after receiving new session keys.

Interfaces For Seucre Comm Client

A client entity can start a secure connection with a server using `initSecureCommunication` (line 5 of Figure 4.13). The parameter `options` (shown in Figure 4.15) includes the target server's address and port, a session key to be used, and cryptography configurations.

```
1     options = {
2       serverHost,
3       serverPort,
4       sessionCryptoSpec,
5       sessionKey = {id, value, absoluteValidity, relativeValidity}
6     }
```

Figure 4.15: Options for `initSecureCommunication` function in Figure 4.13.

```
1     eventHandlers = {
2       onClose,
3       onError,
4       onConnection,
5       onData
6     }
```

Figure 4.16: Event handlers for `initSecureCommunication` function in Figure 4.13.

The event handlers for `initSecureCommunication` function are presented in Figure 4.16. These are callback functions that are called while and after calling `initSecureCommunication`. `onClose` and `onError` are called when the underlying connection is closed and an error occurs while initializing secure communication with a SST server, respectively. `onConnection` is called when the secure communication channel is successfully established and `onData` is called when a message arrives from the server after the secure channel is established. A secure socket is given as a parameter of the callback function, `onConnection`, upon secure connection establishment. This secure socket is used for an accessor to send secure messages to the target server.

Interfaces For Secure Comm Server

```
1     options = {
2       serverPort,
3       sessionCryptoSpec
4     }
```

Figure 4.17: Options for `initSecureServer` function in Figure 4.13.

`initSecureServer` (line 6 of Figure 4.13) initializes a secure server. `options` specify the server's listening port and cryptographic configurations. `eventHandlers` include callback functions for when a client requests a secure communication (`onClientRequest`),

```
1     eventHandlers = {
2       onServerError,       // for server
3       onServerListening,
4       onClientRequest,     // for client's communication
          initialization request
5
6       onClose,             // for individual sockets
7       onError,
8       onData,
9       onConnection
10    }
```

Figure 4.18: Options for `initSecureServer` function in Figure 4.13.

when the server starts listening (`onServerListening`), and when a server error occurs (`onServerError`). Server's `eventHandlers` also include four callback functions(`onConnection`, `onData`, `onClose`, `onError`) that are similar to those of a client for each secure connection.

Interfaces For Secure Publisher

```
1     message = {
2       sequenceNum,
3       data
4     }
```

Figure 4.19: Details of the `message` parameter for `encryptSecurePub` function in Figure 4.13.

`encryptSecurePub` (line 8) takes a securely published `message` (Figure 4.19) as a parameter which will be encrypted and/or authenticated with a `sessionKey` using a specified `cryptoSpec`. The parameter, `message`, includes a sequence number of the published message, `sequenceNum`, which increases per each `message`, and the actual `data` to be published.

Interfaces For Secure Subscriber

When a subscriber receives a secure publish message, it uses `getKeyIDOfSecurePub` (line 9) to get the session key ID used for the message. Then, the subscriber decrypts and/or authenticates the `encryptedMessage` using `decrypteSecurePub` (line 10). Although the functions have "encrypt" and "decrypt" in their names, they can also be used to just authenticate messages without encryption.

Figure 4.20 shows the return value of `getKeyIDOfSecurePub`. This includes `success`, a boolean variable indicating whether getting the key ID was successful and `error` for the rea-

```
1     ret = {
2       success,
3       error,
4       keyId,
5       encryptedMessage
6     }
```

Figure 4.20: Return value for `getKeyIDOfSecurePub` function in Figure 4.13.

son of an error when the function call was not successful. `keyId` is for the session key's ID extracted from the secure publish message and `encryptedMessage` is for an encrypted message from the secure publish message. `encryptedMessage` is used for calling `decryptSecurePub`.

```
1     ret = {
2       success,
3       error,
4       sequenceNum,
5       message
6     }
```

Figure 4.21: Options for `decryptSecurePub` function in Figure 4.13.

The return value of `decryptSecurePub` is shown in Figure 4.21. The meaning of `success` and `error` is the same as that of the return value of `getKeyIDOfSecurePub`. `sequenceNum` and `message` are the sequence number and actual data of the published message after decryption.

4.3.2 Benefits of Secure Communication Accessors

Accessors are untrusted code that serves as proxies for sensors, actuators, and services. Inspired by the web, accessors are therefore executed in a virtualized environment that controls access to resources and data. Such encapsulation provides a starting point for ensuring security and privacy, but it is not sufficient by itself. In particular, the execution environment will have to grant access to physical resources such as sensors and actuators in order to realize IoT applications. How should it authenticate the IoT applications (e.g., whether an application running remotely is modified from the original program components)? Moreover, how can we make sure we only grant permission to legitimate IoT applications to access certain resources (authorization or access control)?

SST provides a set of accessors for bringing authentication and authorization to the IoT while addressing challenges mentioned in Section 1.2.

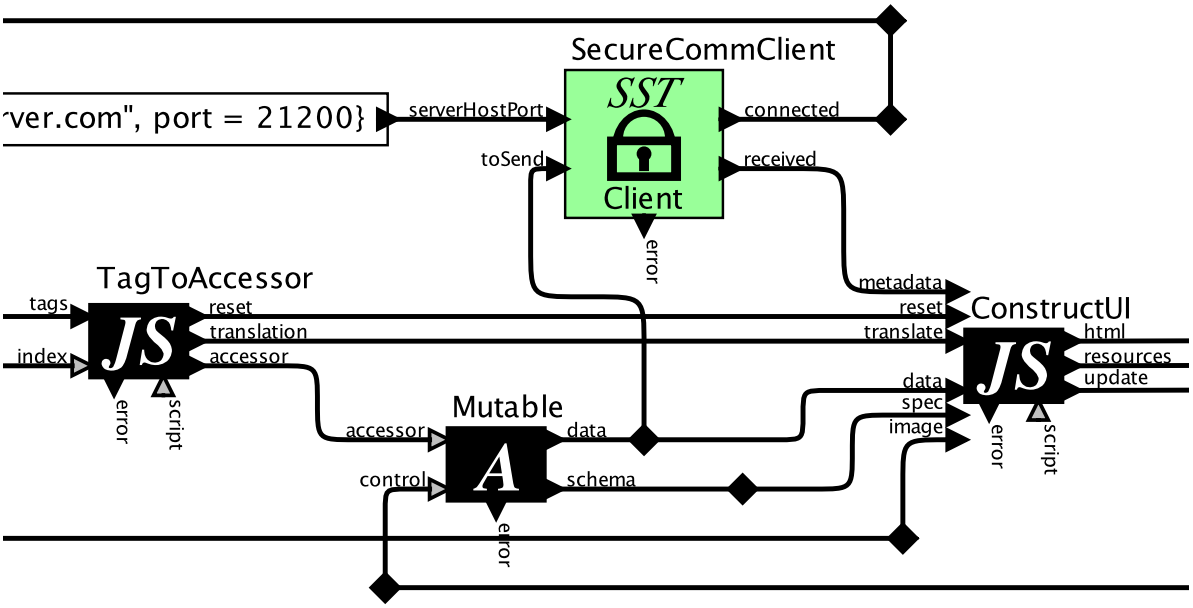


Figure 4.22: Modified part of the example of augmented reality model in [17] with a SecureCommClient accessor for additional security.

Figure 4.22 illustrates a part of an extended version of the augmented reality example in [17], secured by one of the accessors in SST, SecureCommClient. A stream of output data from another program component (marked as Mutable) is encrypted and sent to a cloud server via the SecureCommClient accessor. Let us assume there is another IoT application, namely SensorAnomalyDetector, running on a remote cloud server and programmed using another accessor in SST, SecureCommServer, shown in Figure 4.12. SensorAnomalyDetector takes streams of data from the distributed augmented reality applications reporting their sensor data, executes a machine learning algorithm on collected data, and sends feedback to the applications when any sensor data anomaly is detected. When a client application receives feedback on a detected anomaly from the server, the feedback is sent to the graphic overlays additional input port, *metadata*, to indicate the anomaly as part of the overlay.

In this extended example, the main function of these security accessors is to access authentication and authorization services provided by Auths. After authenticated and authorized by their Auths, SecureCommClient and SecureCommServer establish a secure communication channel between the client and server similar to that of SSL/TLS. But with SST, we can have more choices such as different underlying network protocols (e.g., TCP or UDP) or different cryptography. We do not have to maintain a centralized certificate authority (CA). Adding SST accessors provides additional security guarantees including confidentiality and message authenticity, preventing network-based attackers from eavesdropping or spoofing a device ID. In addition to the two aforementioned accessors, SST provides accessors for con-

structing IoT applications based on a publish-subscribe communication style, using accessors SecurePublisher and SecureSubscriber also shown in Figure 4.12.

Another benefit of using SST accessors comes from encapsulation of cryptography operations and cryptographic key management. As Myers and Stylos [71] point out, the design of APIs is critical for software security, especially in the sense that misuse of APIs can lead to serious security problems. With SST accessors, all software developers need to do is specify configuration parameters and set up initial credentials (e.g., generate a public-private key pair and register the public key with an Auth). Even developers with moderate knowledge in security need not worry about internal cryptographic operations and encryption key management for accessors once the accessor design is correct. Specifying security configurations can be further simplified by using the given profiles as suggested in Table 3.1 of Section 3.2.4.

Combined with actor-oriented modeling semantics where actors communicate only through input and output ports, isolation of cryptographic keys and operations in SST accessors can enhance security when supported by OS-level or architecture-level security mechanisms. By sandboxing the execution of SST accessors, a swarmlet host can restrict the privilege of accessors to read from or write to arbitrary files or network ports, preventing credentials from being leaked or being used maliciously (e.g., by an attacker to spoof the device). If the host is equipped with an architectural security mechanism such as Intel's Software Guard Extensions (SGX) [67], the credentials can be protected even when other processes on the host or the host's middleware or hardware components are compromised. Like other accessors, SST accessors also require some modules to be available on the host. These modules include the crypto module and the TCP or UDP modules.

Chapter 5

Evaluation of Approach

This chapter evaluates security, scalability, and support for heterogeneous security requirements of the proposed approach through analysis and experiments. A security analysis proves that the proposed approach and protocol design satisfy security requirements including confidentiality, integrity, and authenticity of messages. A mathematical scalability analysis shows that the proposed approach can linearly scale with the number of IoT entities by deploying more Auths in theory. The effectiveness of various security configurations provided by SST is shown through experiments and results.

5.1 Security Analysis

In this section, I present a security analysis of the proposed authorization infrastructure. This security analysis was carried out in collaboration with Eunsuk Kang. To make the analysis rigorous, Eunsuk and I 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.

5.1.1 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 the analysis of this section, I 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 by its source entity (*data integrity* and *authenticity* of a message). In the threat model presented in this section, I do not consider security guarantees against availability attacks, such as a denial of service (DoS) or depletion of energy resources; this is in part addressed by an architectural mechanism presented in Chapter 6.

Let us 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. Let us 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, it is assumed that the attacker is not capable of impersonating Auth.

5.1.2 Formal Analysis

Modeling Auth in Alloy

Alloy is a modeling language based on a first-order relational logic [51]. It has been used to analyze a wide range of systems, including web applications [2], network configurations [72], and security policies [73]. 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 5.1 shows a snippet of a model of the proposed authorization infrastructure and its protocol in Alloy. A simplified version is shown here and the full model is available at `https://github.com/iotauth/security_analysis`. The model[1] begins with declarations of data types 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)[2].

The set of Auth and entities in the world are collectively referred as `Node` in our model introduced in [53]. 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 analysis in this section, I will assume some subset of the entities to be malicious (line 29)[3].

[1] 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.

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

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

```
sig Time {} // Totally ordered time steps
/* Data types (keys, payloads, names, IDs) */
sig Payload // data to be sent between entities
sig Name, ID {} // entity names and Auth IDs
abstract sig Key {}
sig SymKey extends Key {} // symmetric keys
sig AsymKey extends Key { pair : AsymKey }
fact { no disj k1, k2: AsymKey | k1.pair = k2.pair }
/* Auth and entities */
abstract sig Node {  publicKey, privateKey: AsymKey }{
        publicKey.pair = privateKey
}
sig Auth extends Node {
        id: ID,
        entityPublicKeys: Name -> AsymKey,
        // session keys allocated so far
        sessionKeys: SymKey -> Time,
        // owners associated with session keys
        owners: sessionKeys -> Name -> Time
}
sig Entity extends Node {
        name: Name,
        payloads: set Payload,
        authPublicKeys: ID -> AsymKey,
        // session keys obtained from Auth
        sessionKeys: Name -> SymKey -> Time
}
// Some of the entities may be malicious
sig Attacker in Entity {}
/* Messages */
abstract sig Message { sender,receiver: Node, t: Time }
sig SESSION_KEY_REQUEST extends Message {
        entity, target: Name, id: ID
}{
        encryptWith[entity+target,sender.authPublicKeys[id]]
        signWith[entity+target,sender.privateKey]
        some newKey: SymKey |
        insert[receiver.sessionKeys,newKey,t] and
        insert[receiver.owners,newKey->entity,t]
}
sig SESSION_KEY_RESP extends Message {
        distrKey, sessionKey: SymKey,
        req: SESSION_KEY_REQUEST
}{
        encryptWith[sessionKey,distrKey]
        encryptWith[distrKey,sender.entityPublicKeys[req.entity
            ]]
        insert[receiver.sessionKeys,req.target->sessionKey,t]
}
sig SECURE_MESSAGE extends Message {
        payload: Payload, target: Name
}{
        encryptWith[payload,(sender.sessionKeys.t)[target]]
}
/* Security property */
check Confidentiality {
        no t: Time, e: Entity - Attacker, a: Attacker |
        some s : e.payloads | accesses[a,s,t]
} for 5 but 10 Time, 10 Message
```

Figure 5.1: A snippet of an Alloy model of the Auth protocol.

Modeling behavior. To reason about the dynamic behavior of a network, I use a style of modeling where an execution is modeled as a sequence of time steps, and each mutable object is associated with a state at each step [51]. In this model, I 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 and delivered[4]. The sender and receiver behavior associated with each type of message is defined using *signature constraints*[5]. 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, I 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)[6].

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).

Verification Procedure

The Alloy Analyzer is a tool that can be used to execute a model or automatically verify it against desired properties. 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 [6], 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 has been verified using the tool is shown in Figure 5.1 (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

[4] For the analysis in this section, I assume that messages are delivered without delays.

[5] A *signature constraint*, specified in the appendix to field declarations, is a statement that is imposed on every member of the signature.

[6] `encryptWith[d,k]` and `signedWith[d,k]` are custom-defined predicates that mean data `d` is encrypted and signed with key `k`, respectively. `insert[x,r,t]` means tuple `x` is added to mutable relation `r` at time `t`.

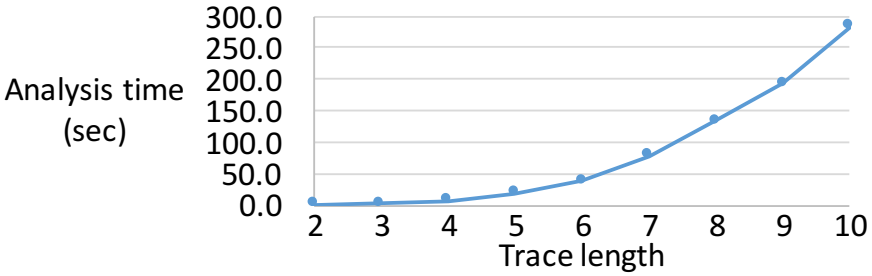


Figure 5.2: Verification times on the Auth model.

non-attacker entity (`e`)[7]. 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`).

Results

In the SST paper [53], we analyzed a model of Auth against the properties stated in Section 5.1.1: 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 5.2 shows the average times taken by the analyzer to generate a counterexample for different trace lengths[8]. 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 5.1; 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., the number of traces potentially explored by the analyzer) is approximately $175^{10} \approx 2.69 * 10^{22}$.

The analyzer generated 17 counterexample traces during the 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*

[7] 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`.

[8] The analysis was performed on a Mac OS X 10.11 machine with 2.7 GHz Intel Core i5 and 8 GB of RAM.

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).

The 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 [4].

5.1.3 Limitations

The threat model used in this section 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. A possible future work is 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 [5]. For example, the smallest counterexample that we found required 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.

5.2 Scalability Analysis

In this section, I provide a mathematical analysis of the scalability of the SST infrastructure. Figure 5.3 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 underlying cryptography and communication configurations, I assume that they are the worst-case upper bounds for all possible configurations.

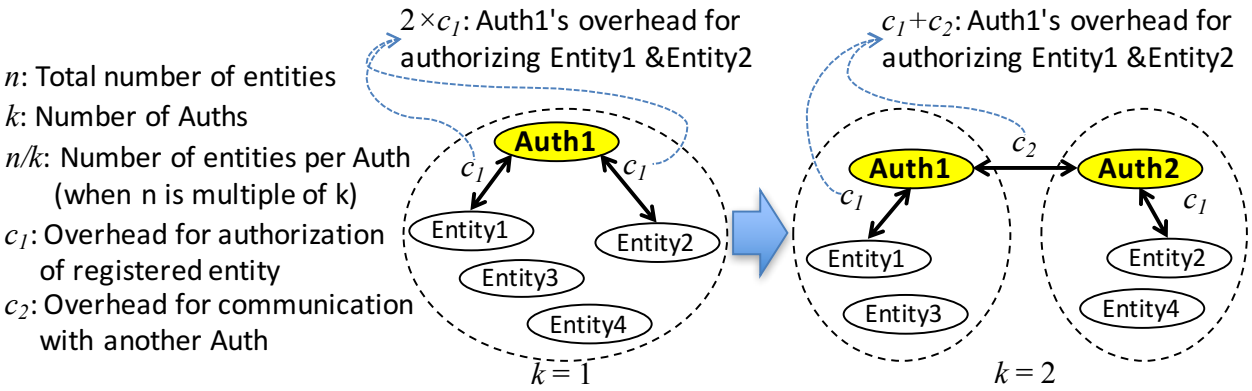


Figure 5.3: Division of entities into two groups registered with separate Auths

I 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

$$t_1 = mn \times 2c_1 \tag{5.1}$$

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

$$pm \times \frac{n}{k} \times 2c_1 \tag{5.2}$$

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 5.3, the resulting overhead is

$$(1-p)m \times \frac{n}{k} \times (c_1 + c_2) \tag{5.3}$$

Summing (5.2) and (5.3), for $k \geq 2$, the total overhead for an individual Auth is

$$t_k = pm \times \frac{n}{k} \times 2c_1 + (1-p)m \times \frac{n}{k} \times (c_1 + c_2) \tag{5.4}$$

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. Then, equation (5.4) becomes

$$t_k = pm \times r \times 2c_1 + (1-p)m \times r \times (c_1 + c_2) \tag{5.5}$$

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, the proposed infrastructure should be scalable for an increasing number of entities.

Table 5.1: Energy cost model used in [55] (energy numbers from [87] and [36])

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 × packet size (bytes)
Receive packet	356 μJ + 0.5 μJ × packet size (bytes)

5.3 Experiments and Results

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

The entities for the experiments in this section 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; I used a *Node.js host* (`https://accessors.org/hosts/node/`), which is based on Node.js [98], 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.

I measured (1) computational security overhead by logging cryptographic operations[9] and (2) communicational overhead by capturing network packets using a packet sniffing tool, WireShark (`https://www.wireshark.org/`). For cryptography operations, I 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 I used for the experiments of TLS. I converted the measurements into energy consumption to estimate overall security overhead. For this conversion, I used the energy cost model used in [55] (shown in Table 5.1).

5.3.1 Server-Client Communication

I describe the findings on the security overhead in establishing secure connections for the client-server communication architecture. For this experiment, I varied three configuration parameters: (1) the maximum number of allowed cached session keys, (2) the underlying

[9] This was done by modifying OpenSSL library (version 1.0.2k) included in Node.js (version 7.6.0).

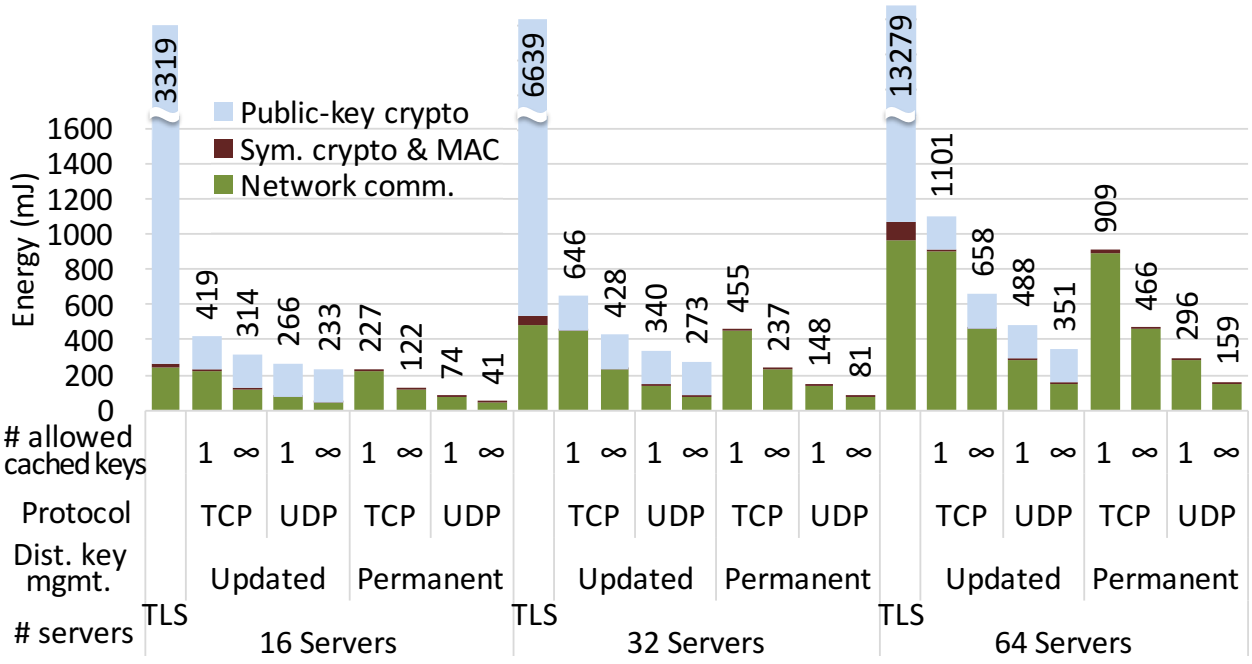


Figure 5.4: Estimated energy consumption of a client for setting up and closing secure connections with 16, 32, and 64 servers. (Note that the energy consumption results for TLS are cut off due to the space limitation.)

network protocol, and (3) distribution key management alternatives. For each entity, either only one cached key was 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, I assumed that the entity did not have a distribution key in its initial deployment.

Figure 5.4 shows the estimated energy consumption of a client for establishing/closing secure connections with 16, 32, and 64 servers under different configurations. Figure 5.5 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 the proposed 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.

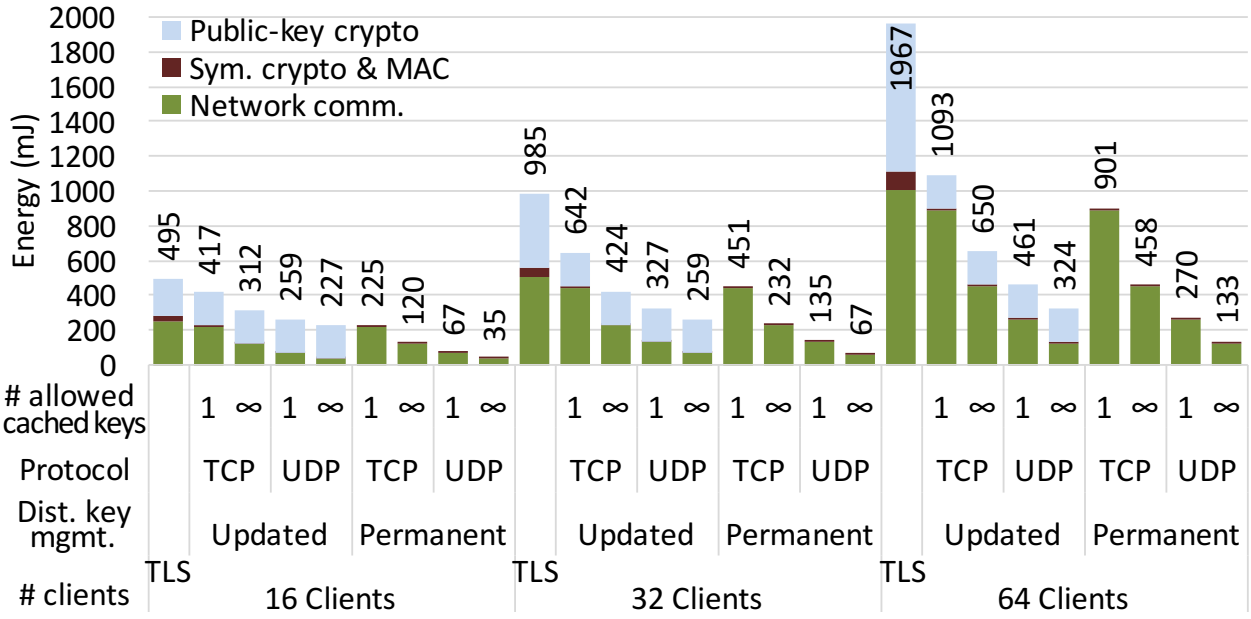


Figure 5.5: Estimated energy consumption of a server for setting up and closing secure connections with 16, 32, and 64 clients.

TLS consumes more energy on SHA-256 MAC than SST does, since it needs to verify client and server certificates, although there is an only negligible difference in energy used for AES (symmetric cryptography) on data encryption/decryption.

5.3.2 A Sender and Multiple Receivers

In this section, I describe the security overheads in one-to-many communication architecture, where one node sends encrypted messages to multiple other entities. I conducted experiments with four different settings for a sender and receivers described in Figure 5.6. The first setting in Figure 5.6 (a) employed a separate, individual TLS connection between the sender and each receiver. Figure 5.6 (b) shows another setting using individual secure connections but with a shared session key distributed by Auth. The setting in Figure 5.6 (c) used a publish-subscribe protocol, MQTT [8], to connect the sender and receivers sharing a single session key. I used an open-source MQTT message broker Mosquitto (`http://mosquitto.org`) for forwarding published messages from the sender to receivers. I assumed that the broker should not be able to decrypt the published messages.

In the final setting shown in Figure 5.6 (d), I 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, I varied the distribution key management for each experiment (i.e., updated or permanent

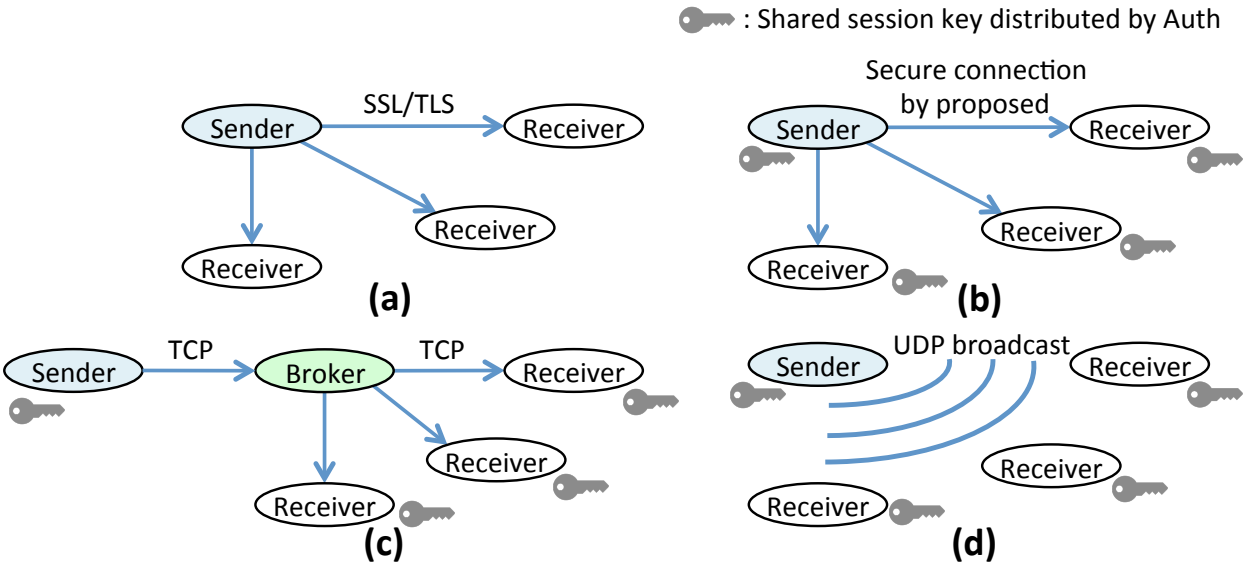


Figure 5.6: Four different settings of a sender and receivers; (a) Individual SSL/TLS connections. (b) Individual secure connections by the proposed approach using a shared session key. (c) Publisher and subscribers connected via a message broker. (d) Sender and Receivers over UDP broadcast in a local network.

distribution keys).

Figure 5.7 shows the estimated energy consumed for setting up keys and connections with different numbers of receivers. The energy consumption for *TLS* (Figure 5.6 (a)) and *ISC* (Individual Secure Connections, Figure 5.6 (b)) increases as the number of receivers increased. However, the energy consumption for *MB* (Message Broker, Figure 5.6 (c)) and *UB* (via UDP Broadcast, Figure 5.6 (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 5.3.1.

Figure 5.8 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

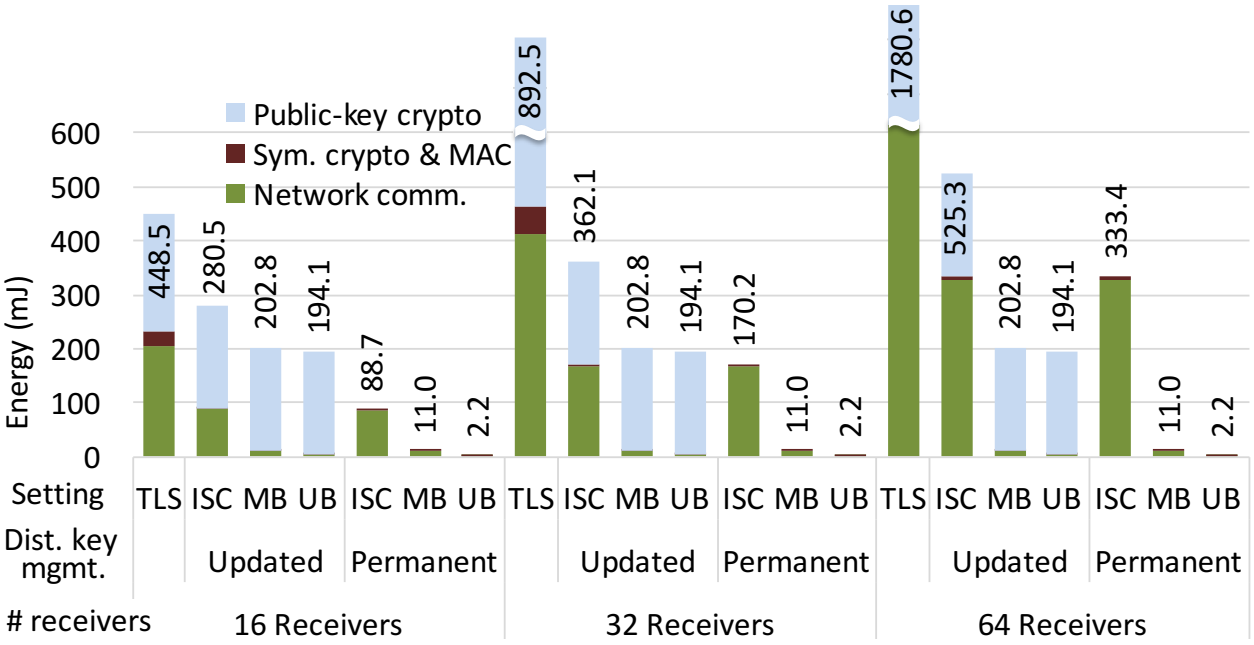


Figure 5.7: Estimated energy consumption of a sender for setting up secure connections with 16, 32, and 64 receivers. (*ISC*: Individual Secure Connections, *MB*: with a Message Broker, *UB*: via UDP Broadcast)

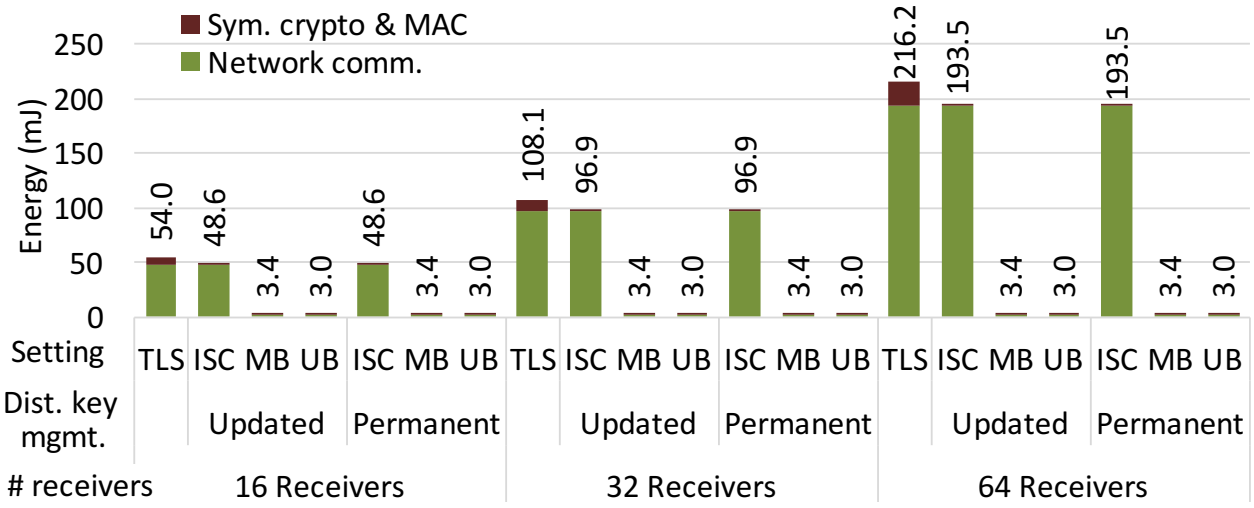


Figure 5.8: Estimated energy consumption of a sender for sending a 1 KB message to 16, 32, and 64 receivers.

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.

Chapter 6

Enhancing Availability

This chapter introduces an architectural mechanism for enhancing availability, based on locally centralized globally distributed authentication and authorization. Specifically, I propose a secure migration technique where the IoT entities can migrate to other trusted authorization entities, Auths, when their Auth becomes unavailable due to a failure or an availability attack such as a Denial-of-Service (DoS) attack. This secure migration technique is designed and integrated into SST as part of its implementation. This chapter also presents a problem formulation for constructing migration policies that satisfy access requirements between IoT entities and optimize the communication costs. The effectiveness of the proposed secure migration is illustrated through network simulation and a concrete IoT application.

6.1 Background and Goals

6.1.1 Denial-of-Service (DoS) Attacks

The ultimate purpose of Denial-of-Service (DoS) attacks is to breach a system's availability. There can be many different motivations for launching a DoS attack, revenge, blackmail, etc. The most common and widely known type of DoS attacks will be a DDoS (Distributed Denial-of-Service) [109] attack, where an attacker control a number of compromised computers (a botnet) to exhaust computational and communication resources. For the IoT, there can be more than remote DDoS attacks. An adversary can have more points of access to the system either physically or via direct wireless communication. One example of this is a DoS attack using jammers [82] which can produce high-energy radio signals into the air to disrupt the wireless communication around.

6.1.2 Proposed Architectural Mechanism

The main goal of this chapter is to provide defense and mitigation mechanisms against DoS attacks and other failures on the IoT by providing a distributed and robust authorization infrastructure based on an edge-computing architecture. Specifically, I propose a mechanism

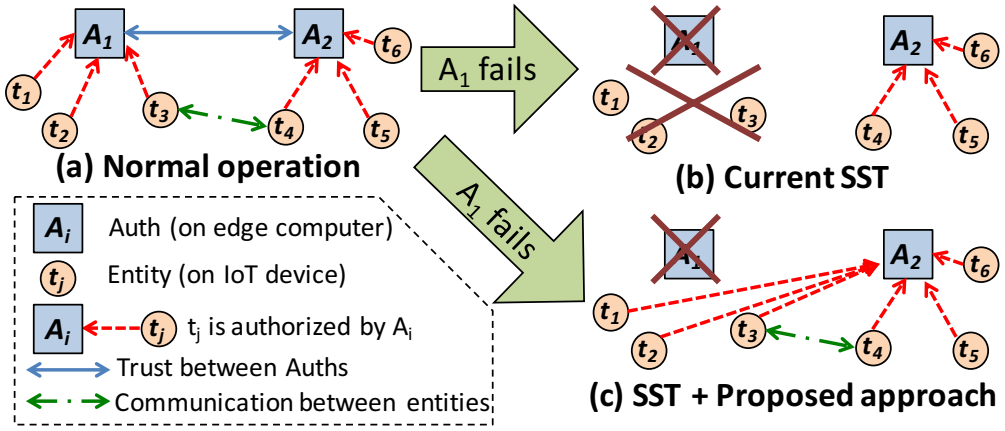


Figure 6.1: (a) SST in normal operation. (b) SST without the architectural mechanism proposed in this chapter, in case of Auth failure. (c) Proposed secure migration technique for enhancing availability of SST, in case of Auth failure.

that can make the proposed authorization infrastructure resilient even when some of the edge devices become unavailable due to denial-of-service attacks and other failures. I accomplish this goal by having Auths take over other Auths' authorization tasks when they are not available. Architectures based on distribution and replication have been used in many systems such as Google's Spanner [26] to achieve better availability.

Figure 6.1 (a) shows an example of a small IoT network in normal operation using SST: there are two Auths, A_1 and A_2, and each Auth is responsible for authorization of three entities, t_1, t_2 and t_3 for A_1 and t_4, t_5 and t_6 for A_2. Let us assume the current SST design without the architectural mechanism that is proposed in this chapter. If A_1 fails due to a DoS attack, authorization services for registered entities will become unavailable as shown in Figure 6.1 (b). This will also affect the availability of other entities. For example, communication between t_4 and t_3 will be disrupted.

However, this does not fully utilize the *globally distributed* architecture of SST. The goal of this chapter is to provide enhanced availability of IoT authentication and authorization services through a mechanism that leverages SST's globally distributed architecture. The architectural mechanism proposed in this chapter includes backing up information for authorization services to other trusted Auths and securely migrating IoT entities to continue the IoT services even in case of Auth failures as shown in Figure 6.1 (c).

6.1.3 Considerations for Migration Policies

Although the high-level idea behind migration may appear straightforward, determining migration policies is a non-trivial problem due to a number of factors that need to be considered. Figure 6.2 demonstrates examples of Auths and entities with the consideration of Auth failures. First, we need to consider trust relationships among Auths to guarantee the

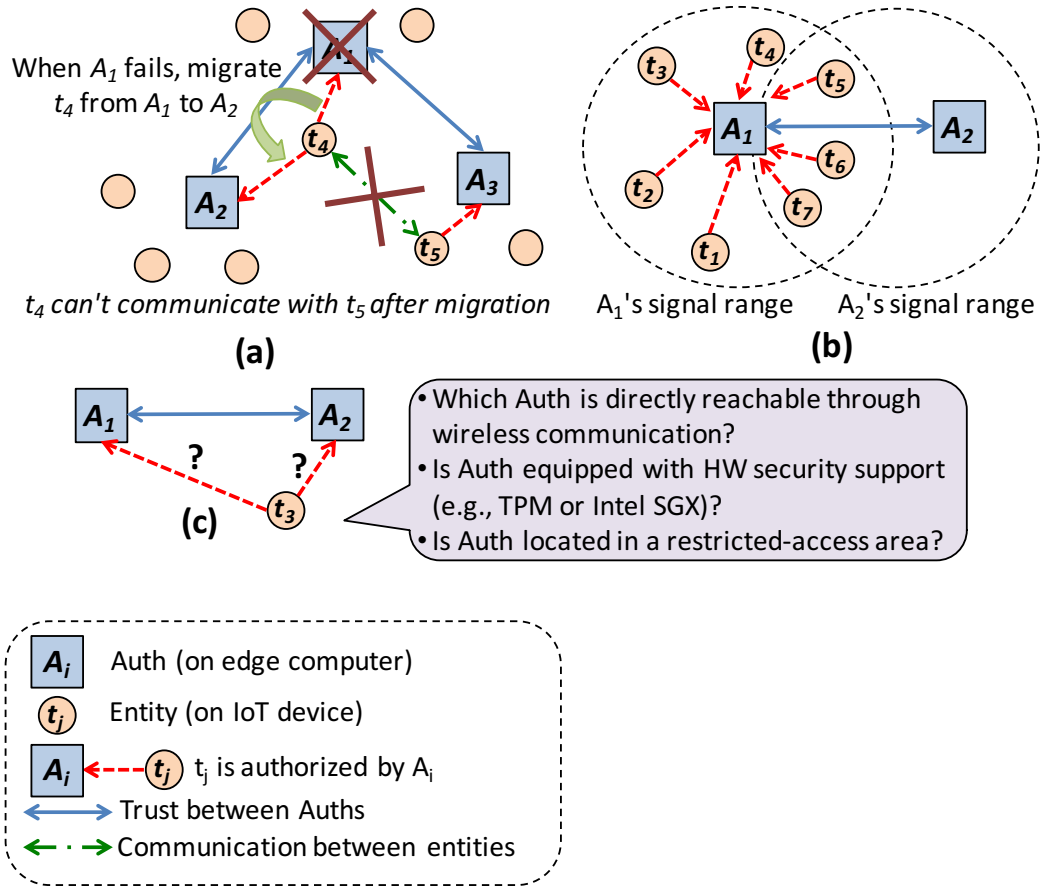


Figure 6.2: Considerations for migration policies: (a) Trust among Auths and communication requirements between IoT entities (Things). (b) Balancing workload of Auths for Authorization. (c) Characteristics and security guarantees of Auths.

required communication channels between certain entities. In the example of Figure 6.2 (a), t_4 and t_5 are required to communicate with each other, meaning that there must be a way for them to share the same session key distributed by Auths. Trust relationships between Auths are A_1–A_2 and A_1–A_3, and t_4 and t_5 are registered with A_1 and A_3, respectively. If we arbitrarily decide to migrate t_4 to A_2 in the event of A_1's failure due to a DoS attack, then t_4 will not be authorized to communicate with t_5 anymore after migration because there is no trust relationship between A_1 and A_3. Therefore, a proper migration policy should migrate t_4 to A_3 instead in case of A_1's failure.

Let us consider a different case described in Figure 6.2 (b), where the nodes are connected via wireless communication, with signal ranges of A_1 and A_2 as depicted. I assume that communication costs increase as the distance between nodes increases. Then, it will be reasonable for each entity to connect to A_1 to minimize the communication costs. However, this decision may affect the overall system, depending on the operational conditions of A_1.

What if A_1 has a limit in its capacity for authorization tasks for some reason? For example, A_1 has to run other computation-intensive tasks other than a job as an Auth. Thus, authorization of entities above a certain threshold can lead to degradation of its tasks. Or A_1 may be running on a battery-powered edge device with a limited energy budget. In such cases, assigning entities within A_2's signal range (t_5, t_6 and t_7) to A_2 can be more desirable for the overall system despite the higher communication costs for those entities. This example shows that we must consider Auths' operating conditions for entity assignment as well as the communication costs of entities.

Another important factor to be considered for Auth-entity assignments is security guarantees provided by Auths. In the example shown in Figure 6.2 (c), there are two Auths, A_1 which is equipped with hardware security support, TPM (Trusted Platform Module) and Intel's SGX (Software Guard Extensions) and located in a restricted-access area, and A_2 which is located in an open area without any hardware security support. If security guarantees provided an Auth are a more important factor for the entity, t_3, then it may want to be registered with A_1 rather than with A_2 even if the cost of communication with A_1 is higher than A_2.

For the implementation of migration policy construction, Gurobi [41] library was used as the underlying ILP solver and integrated to SST.

6.1.4 Threat Model and Goals

Threat mode: As a threat model, I consider any types of denial-of-service (DoS) attacks or failures on the Auths running on edge devices. I assume the attackers target the authorization entities, Auths, to cause a denial of service on access control (authorization) for the IoT devices. The types of DoS attacks include flooding attacks such as distributed denial-of-service (DDoS) attacks and attacks leveraging physical/wireless access to the edge devices to cause a denial of service. This is because, compared to cloud servers, it is relatively easier to access some edge devices, for example, public Wi-Fi routers. I do not consider impersonation of Auths. This problem can be addressed by preventing access to private keys of Auths, using various approaches including hardware security support such as Intel's Software Guard Extensions (SGX).

Requirement constraints: I consider following constraints for determining assignments (or mapping) between entities (IoT devices) and Auths.

- Each entity can have security requirements that must be provided by Auth. These requirements include hardware support (e.g., TPM/SGX), restricted physical location, cryptography specification (e.g., ephemeral keying for perfect forward secrecy).
- Each Auth has a threshold for authorization tasks in terms of upper bounds for authorization requests per minute or upper bounds for session keys cached in Auth. In other words, there is a threshold of registered entities for each Auth. When there are more registered entities than this threshold the Auth will experience degradation in the authorization tasks.

- Entity-to-entity communication requirements. Some entities need to be authorized to communicate with certain entities even after migration. This requires trust relationships between Auths with which those entities are registered.
- There exists multiple criticality levels for the communication requirements among entities, which often called mixed-criticality systems [18]. This means we should prioritize communication requirements of high-criticality entities.

Optimization criteria: For determining assignments between entities and Auths, I consider following criteria for optimization. Among the assignments that can meet constraints above, I chose the assignments that can optimize following requirements.

- For some entities, especially those which run on battery-powered IoT devices, reducing communication costs with Auths can help increase their availability. I assign entities with Auths so that we can minimize recurring costs for communication with Auths.
- Migration costs, which are the costs for migrating entities from unavailable Auth to another, can be minimized. Given that the assignments meet required constraints, it would be desirable to minimize migration costs for Auths and entities.
- For some Auths running on battery-powered edge devices such as mobile phones or laptops, reducing their costs for authorizing their registered entities can increase overall systems availability. I consider this factor in the exploration of possible assignments between entities and Auths.

6.2 Problem Formulation

In this section, I formulate the problem of finding the best assignment of entities and Auths in the event of DoS attacks or other failures on Auths. I assume that I have a set of available Auths running on edge devices. For each DoS attack case for each Auth, I use the set with a reduced number of Auths. I also assume that I have a set of entities on IoT devices with security and communication requirements. I describe the input of the problem-solving algorithm in the form of a graph with two type of nodes (*Auths* and *Things*) and edges connecting these nodes.
Input: A graph G which is as follows.

- G: A graph which includes two sets of nodes (A and T) and edges (E). These two sets are A for Auths and T for thing entities. $G =< V, E >$
- V: A set of all vertices (nodes) such that $V = A \cup T$, $A \cap T = \emptyset$.
- A: A set of nodes representing Auths.
- T: A set of nodes representing things (or IoT entities).

- $a \in A$: An Auth node. Each a has a set of attributes. $a =< S_A, C_A >$
  - S_A: A set of security guarantees that an Auth can provide.
  - C_A: A threshold of Auth's capacity for authorization of entities without causing degradation.
- $t \in T$: A thing entity node. Each t has a set of attributes. $t =< S_T, R_T >$
  - S_T: A set of security guarantees that a thing requires.
  - R_T: Requirements for authorization tasks. A sum of these values for entities assigned to a certain Auth should not exceed the threshold capacity of an Auth, in order not to cause degradation.
- E: A set of edges representing relationships for nodes from A and T. E consists of different types of edges. $E =< E_{AA}, E_{TT}, E_{AT} >$
  - E_{AA}: A set of edges representing relationships between Auths. Each $e_{AA} \in E_{AA}$ includes trust relationship as a binary value (0 or 1) and a communication cost value between Auths as a positive real number. e_{AA} also includes the migration cost for an entity that migrates from one Auth to another.
  - E_{TT}: A set of edges representing relationships between things (IoT entities). Each $e_{TT} \in E_{TT}$ includes the communication requirement represented as a positive integer meaning the criticality of communication. e_{TT} also includes communication cost as a positive real number.
  - E_{AT}: A set of edges representing assignments between entities and Auths. For computing assignments in the event of Auth's failure, it has a partial assignment. For computing initial assignments, this set will be an empty set. Each edge is represented as exists or not exists.

Output: The output of the algorithm is E_{AT} which contains full assignments for Auths and entities.
Constraints: The set of constraints that must be satisfied by every valid graph G is defined as:

$$constraints(G) \equiv securityCons(G) \wedge trustCons(G) \wedge capacityCons(G)$$

which are, in turn, defined in terms of the following constraints:

- **Security constraint**: For each thing and its security requirement, the thing must be connected to at least one Auth that provides the same security guarantee:

$$securityCons(G) \equiv \forall t \in T \; \forall r \in t.S_T \; \exists a \in A \cdot (a, t) \in E_{AT} \wedge r \in a.S_A$$

- **Trust constraint**: For every pair of things that communicate with each other but belong to different Auths, the latter pair must satisfy the trust constraint for the things:

$$\begin{aligned} &trustCons(G) \equiv \\ &\quad \forall t_1, t_2 \in T, a_1, a_2 \in A \cdot \\ &\qquad (t_1, t_2) \in E_{TT} \wedge (a_1, t_1) \in E_{AT} \wedge (a_2, t_2) \in E_{AT} \wedge \neg(a_1 = a_2) \implies \\ &\qquad\quad critical(t_1, t_2) = 1 \implies trust(a_1, a_2) = 1 \end{aligned}$$

  where $critical(t_1, t_2)$ returns the criticality of the communication between things t_1 and t_2, and $trust(a_1, a_2)$ returns the trust relationship between the two Auths.

- **Auth capacity constraint**: For each Auth, the sum of authorization task requirements for all entities connected to this Auth must not exceed its capacity:

$$capacityCons(G) \equiv \forall a \in A \cdot (\sum_{t \in ts(a)} t.R_T) \leq a.C_A$$

  where $ts(a) = \{t \in T \mid (a, t) \in E_{AT}\}$

Costs to be optimized:

I formulate the migration task as the problem of multi-objective optimization (MOO); in particular, the goal here is to optimize over two cost variables, TC_A and TC_T, which represent the total costs for the Auths and things:

$$TC_A = C_A + w_A \times C_M \qquad TC_T = C_T + w_T \times C_M$$

Here, w_A and w_T are positive real numbers that represent the weighting factors to distribute the migration costs over Auths and things. The types of costs are defined as follows:

- **Auth-to-Auth costs**: The sum of communication costs between Auths is defined as:

$$C_A = \sum_{e \in E_{TT}} (AA(e).cost_{comm})$$

  where $AA(e)$ returns the Auth – Auth edge for communication between the pair of things in the given edge, e. Note that if the things belong to the same Auth, then the cost associated with $AA(e)$ is 0.

- **Thing-to-thing costs**: The sum of communication costs between things is defined as:

$$C_T = \sum_{e \in E_{TT}} (AT_1(e).cost_{comm} + AT_2(e).cost_{comm})$$

  where $AT_1(e)$ returns the edge between the first thing and the Auth that it belongs to (similarly, $AT_2(e)$ returns the edge between the second thing and its Auth).

- **Migration costs**: The overall migration costs for all entities that were migrated during the attack are defined as:

$$C_M = \sum_{(t,e) \in migrated(G,G')} (e.cost_{migr})$$

where I define (t, e) to be a tuple in $migrated(G, G')$ if and only if thing t was migrated from $e.a_1$ to $e.a_2$ between graphs G and G'; that is,

$$migrated(G, G') = \{(t, (a_1, a_2)) \in T \times E_{AA} \mid (a_1, t) \in E_{AT} \wedge (a_2, t) \in E'_{AT}\}$$

I solve this problem as Mixed-Integer Programming (MIP).

6.2.1 Migration example with cost optimization

Let us consider the example shown in Figure 6.3, where there are three Auths, a_1, a_2 and a_3, five things (IoT entities) t_1 through t_5. Let the cost of communication between a_i and t_j, $(a_i, t_j).cost_{comm}$ be $c(a_i, t_j)$ for readability. Similarly, I set $(a_i, a_k).cost_{comm} = c(a_i, a_k)$. Here the communication costs include energy consumption on wireless communication, cryptographic operations and accessing local data storage. In reality, most of the communication costs should be asymmetric. That is, the cost of communication between an Auth, a_i, and a thing t_j, $c(a_i, t_j)$, should not be the same for a_i and t_j because the amount of tasks for authorization and the operational environment (including the platform and underlying hardware) are different. The cost of communication between two Auths may also be different for each Auth, depending on their underlying platforms. However, here I assume the cost is the same for each communicating part in this example for simplicity.

The relationships and costs of communication between Auths and things are described in Figure 6.3. All Auths trust one another. t_1 and t_2 are required to communicate with each other and so are t_1 and t_5. Originally, t_1 and t_2 are registered with a_1, t_3 and t_4 are registered with a_2, and t_5 is registered with a_3. These relationships can also be stated as follows:

$$\{(a_1, a_2), (a_1, a_3), (a_2, a_3)\} \subset E_{AA}$$
$$\{(a_1, t_1), (a_1, t_2), (a_2, t_3), (a_2, t_4), (a_3, t_5)\} \subset E_{AT}$$
$$\{(t_1, t_2), (t_1, t_5), (t_3, t_4)\} \subset E_{TT}$$

Let us assume that the maximum number of things that a_2 can handle is three. Note that the capacity limit for a_2 is exaggerated to demonstrate the effect of capacity with a simple example.

Let us consider the situation where we want to compute a migration plan when a_1 becomes unavailable. First, we delete all edges including a_1 and resulting edges are as follows.

$$\{(a_2, a_3)\} \subset E_{AA}$$
$$\{(a_2, t_3), (a_2, t_4), (a_3, t_5)\} \subset E_{AT}$$
$$\{(t_1, t_2), (t_1, t_5), (t_3, t_4)\} \subset E_{TT}$$

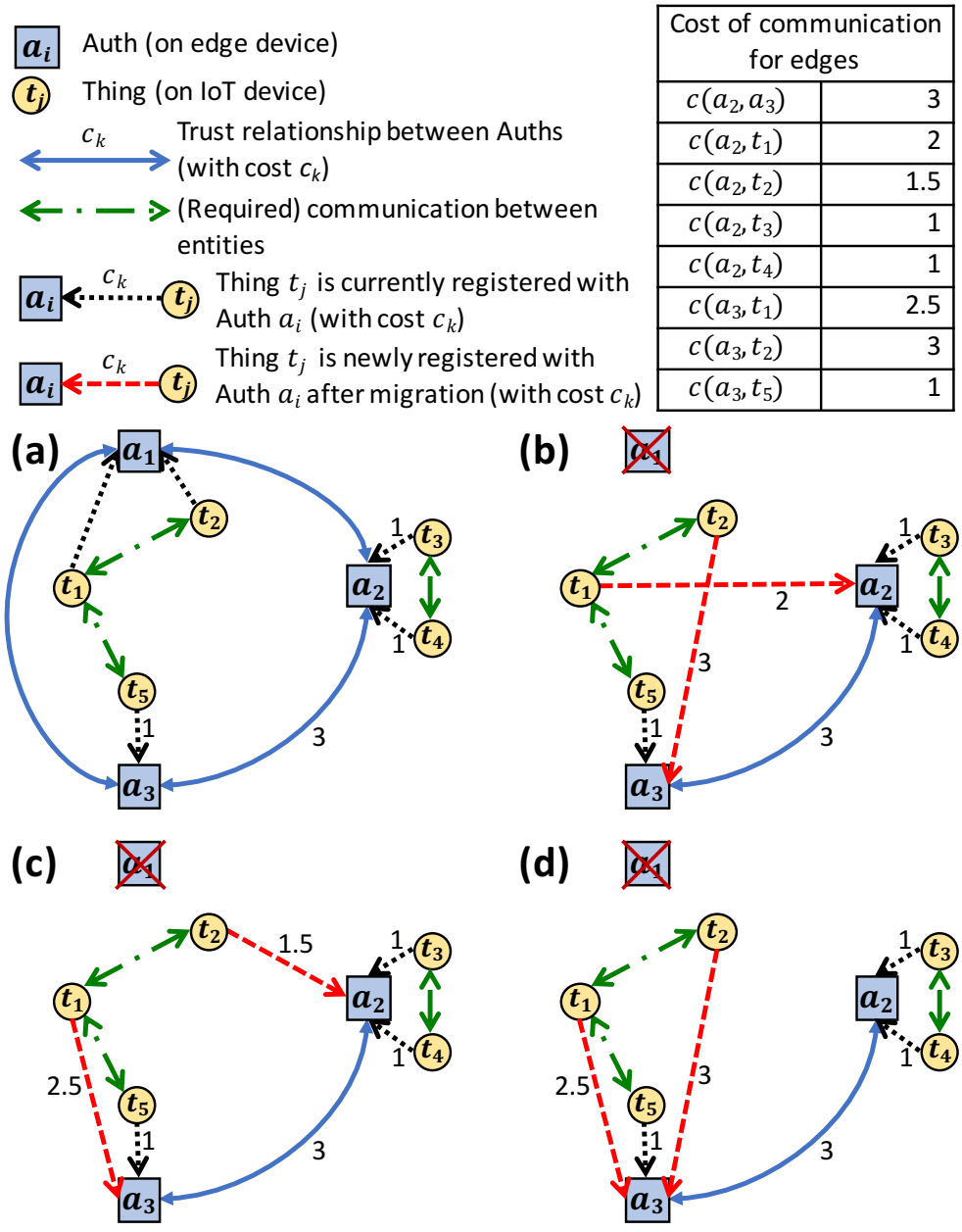


Figure 6.3: Migration plan examples with considerations for communications costs for authorization of IoT entities after migration; (a) Relationships and communication costs between Auths and entities during normal operation. (b) Migration $plan_1$ after a_1's failure. (c) Migration $plan_2$ after a_1's failure. (d) Migration $plan_3$ after a_1's failure.

Now, we have to assign t_1 and t_2 Auths so that we can meet the communication requirements specified in E_{TT}. We also need to meet the Auth capacity constraint which is

$$|\{(x,y)|(x,y) \subset E_{AT} \wedge x = a_2\}| \leq 3$$

meaning that the number of things registered with a_2 should not exceed 3. Due to this constraint, either only one of t_1 or t_2 can migrate to a_2 but not both. Therefore, there can be three possible migration plans and let us call these plans $plan_1$, $plan_2$ and $plan_3$.

- $plan_1$: $\{(a_2, t_1), (a_3, t_2)\} \subset E_{AT}$
- $plan_2$: $\{(a_2, t_2), (a_3, t_1)\} \subset E_{AT}$
- $plan_3$: $\{(a_3, t_1), (a_3, t_2)\} \subset E_{AT}$

Let us consider the resulting authorization communication costs of $t_1$and t_2, which are C_{t_1} and C_{t_2}. I just consider authorization costs to meet the communication requirements, so C_{t_1} will be twice (with t_2 and t_5) of the authorization cost with its registered entity and C_{t_2} will be the authorization cost with its registered entity. Note that the communication costs of t_3, t_4 and t_5 remain the same after migration, that is $C_{t_3} = 1$, $C_{t_4} = 1$ and $C_{t_5} = 1$. The total communication cost of things is $C_T = C_{t_1} + C_{t_2} + C_{t_3} + C_{t_4} + C_{t_5} = C_{t_1} + C_{t_2} + 3$.

For $plan_1$:

$$\begin{aligned} C_{t_1} &= 2 \times c(a_2, t_1) = 4 \\ C_{t_2} &= c(a_3, t_2) = 3 \\ C_T &= C_{t_1} + C_{t_2} + 3 = 10 \end{aligned}$$

For $plan_2$:

$$\begin{aligned} C_{t_1} &= 2 \times c(a_3, t_1) = 5 \\ C_{t_2} &= c(a_2, t_2) = 1.5 \\ C_T &= C_{t_1} + C_{t_2} + 3 = 9.5 \end{aligned}$$

For $plan_3$:

$$\begin{aligned} C_{t_1} &= 2 \times c(a_3, t_1) = 5 \\ C_{t_2} &= c(a_3, t_2) = 3 \\ C_T &= C_{t_1} + C_{t_2} + 3 = 11 \end{aligned}$$

Let us consider the resulting authorization communication costs of a_2 and a_3. The communication cost for a_2 to handle authorization of t_3 and t_4 remains the same after migration and let us set this cost $C'_{a_2} = c(a_2, t_3) + c(a_2, t_4) = 2$. Similarly, the communication cost for a_3 to handle authorization of t_5 remains the same after migration which is $c(a_3, t_5) = 1$.

Table 6.1: The total cost to be optimized depending on different weights for the things and Auths, w_T and w_A, respectively. (Note that the minimum costs are marked as bold for each set of weights.)

(w_T,w_A)	(0.9,0.1)	(0.8,0.2)	(0.7,0.3)	(0.5,0.5)
$plan_1$	11.2	12.4	13.6	16
$plan_2$	**10.1**	**10.7**	11.3	12.5
$plan_3$	11	11	**11**	**11**

For $plan_1$:

$$C_{a_2} = C'_{a_2} + 2 \times c(a_2, t_1) + 2 \times c(a_2, a_3) = 12$$
$$C_{a_3} = c(a_3, t_2) + c(a_3, t_5) + 2 \times c(a_2, a_3) = 10$$
$$C_A = 22$$

For $plan_2$:

$$C_{a_2} = C'_{a_2} + c(a_2, t_2) + c(a_2, a_3) = 6.5$$
$$C_{a_3} = 2 \times c(a_3, t_1) + c(a_3, t_5) + c(a_2, a_3) = 9$$
$$C_A = 15.5$$

For $plan_3$:

$$C_{a_2} = C'_{a_2} = 2$$
$$C_{a_3} = 2 \times c(a_3, t_1) + c(a_3, t_2) + c(a_3, t_5) = 9$$
$$C_A = 11$$

Let TC be the total communication cost to be optimized. TC can be expresses as a weighted sum of C_T and C_A:

$$TC = w_T C_T + w_A C_A$$

where w_T and w_A are the weights for C_T and C_A, respectively. Table 6.1 shows TC for different sets of weights, illustrating the plan minimizes TC can differ depending on the weights. For example, when $(w_T, w_A) = (0.8, 0.2)$, $plan_2$ leads to a minimum communication cost, while $(w_T, w_A) = (0.7, 0.3)$ make $plan_3$ a plan with a minimum communication cost.

6.3 Secure Migration

In this section, I describe a robust architecture and implementation for achieving secure migration. In SST, each Auth has database tables to store authorization information of entities and trust relationships among Auths. As introduced in Chapter 4, the four main database

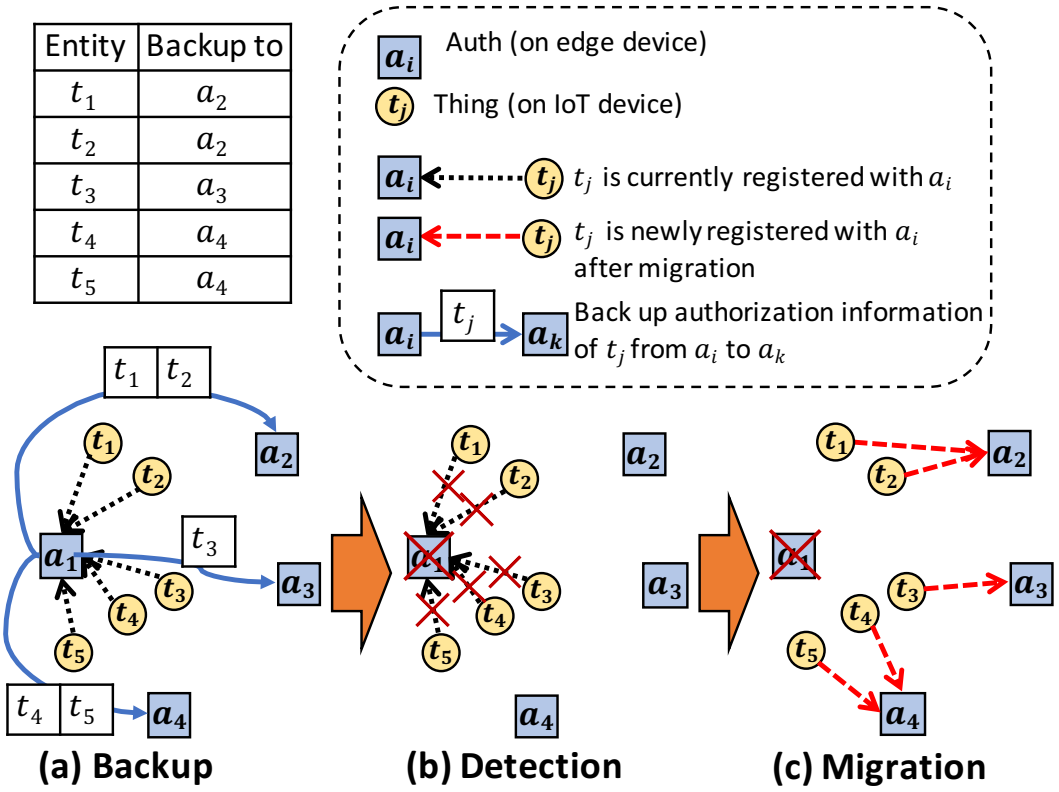


Figure 6.4: Overview of operations to defend against DoS attacks: (a) Backup, (b) Detection, and (c) Migration.

tables maintained by each Auth are *Registered Entity Table*, *Communication Policy Table*, *Trusted Auth Table*, and *Cached Session Key Table*. The trust relationships among Auths are stored in *Trusted Auth Table* and this information is used to securely deliver information of registered entities in *Registered Entity Table* to trusted Auths. If the communication policy of entities to be backed up is not available in the trusted Auth's *Communication Policy Table*, then the necessary communication policies should be backed up as well.

Figure 6.4 illustrates the operations for backup, detection, and migration. As shown in Figure 6.4 (a), during normal operation, each Auth backs up credentials and information about its registered IoT entities to other trusted Auths as planned using the migration methods proposed in Section 6.2. The registered entities detect that their Auth is unavailable as depicted in Figure 6.4 (b). This can be done by setting up a threshold for connection failures in trying to reach their Auth for each entity; for example, an entity decides that its Auths is down after 10 connection failures for 10 minutes. Upon detection of the Auth's failure, each entity tries to reach alternative Auths as shown in Figure 6.4 (c). For this, each entity needs to be assigned a list of hosts and ports for alternative Auths that it can potentially migrate to. Each of alternative Auth will reply either with a new credential that

can be used to be authorized by the alternative Auth or with a message saying that the entity should try another alternative Auth.

6.3.1 Authorization Procedure: In Context of Secure Migration

In SST, each entity is authorized by an Auth by receiving a *session key*, which is a temporary symmetric cryptographic key for accessing a certain service or communicating with another entity (or entities). A session key can be just a MAC key when the access activity does not require confidentiality of the data, or it can include both a cipher key for encryption and also a MAC key for message authentication. A session key could be a single key value if the session cryptography uses authenticated encryption that does both the encryption and message authentication with a single key value.

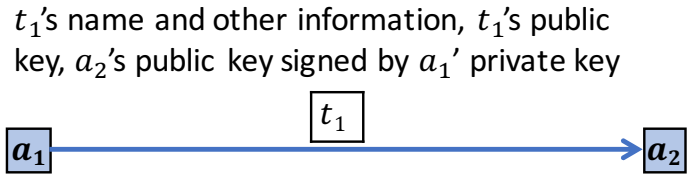
Session keys need to be confidential; they must be only known to the entities participating in a certain access activity. Thus, these session keys must always be encrypted with another symmetric key between Auth and each entity called a *distribution key*. A distribution key includes a cipher key and a MAC key for encryption and for message authentication, respectively. Or, a distribution key can be a single value if the distribution key uses authenticated encryption. In this chapter, I only consider a distribution key with a pair of a cipher key and a MAC key for simplicity.

A distribution key can be optionally updated using public-key cryptography. In this case, Auth and the registered entity should have already exchanged public keys during the registration (initialization) process of SST. This public-key cryptography can also include an ephemeral Diffie-Hellman key exchange for perfect forward secrecy of the distribution key, meaning that the previously used distribution keys cannot be recovered even when the private key of either Auth and the entity is stolen. SST provides an option using permanent distribution key so that it can support resource-constrained IoT devices that cannot afford public-key cryptography.

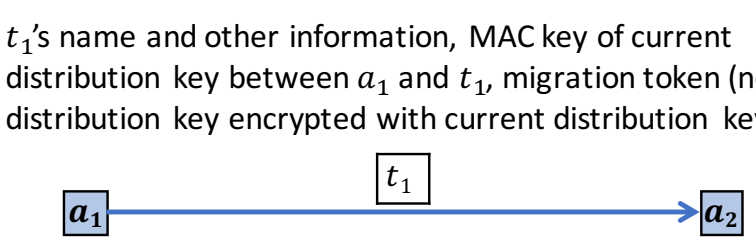
6.3.2 Prepare for Migration

For preparation before a DoS attack is launched or a failure occurs on Auths, migration plans are computed using the approach presented in Section 6.2. After this computation, the migration plan is entered into each Auth's registered entity table, specifying an ID of the trusted Auth to which each entity should migrate. For the following communications between trusted Auths, I assume that all messages exchanged are protected by HTTPS over TLS (Transport Layer Security).

As backup information, Auth needs to prepare the credential that will be transferred from the trusted Auth to the registered entity that is migrating. For example, if the entity uses public-key cryptography for authorization as shown in Figure 6.5 (a), then Auth needs to issue a certificate for the trusted Auth signed by its private key, so that entity can verify that the Auth to which it is migrating is trusted by its original Auth.



(a) If t_1 uses public-key cryptography for authorization



(b) If t_1 uses symmetric key cryptography for authorization

Figure 6.5: Backup operation details depending on the type of cryptography used for authorization. Note that the channel between two Auths a_1 and a_2 is protected by HTTPS over TLS (Transport Layer Security). Assume that the permanent distribution key comprises of a cipher key for encryption and a MAC key for message authentication.

If the entity does not have a capability of performing public-key cryptography and uses symmetric cryptography for authorization (i.e., using a permanent distribution key), then Auth prepares a *migration token* which is a new (permanent) distribution key encrypted with the current distribution key as shown in Figure 6.5 (b). Let us assume that a distribution key comprises a cipher key (for encryption) and a MAC key (for message authentication). The process is as follows. The Auth generates a new distribution key value (both the cipher key and message authentication key) for the registered entity and the new Auth (that the registered entity will migrate to). Then the Auth encrypts and authenticates this new distribution key using current distribution key.

The new Auth's certificate and the migration token have validity periods that are configurable. Having proper expiration times (which are part of the validity periods) can prevent an obsolete certificate and migration token from being used by adversaries and also make sure that the entity migrates to the Auth as supposed in the up-to-date migration plan. If there is a newly registered entity, then the migration plan for the new entity will be updated and its credentials and information should be backed up to one or more trusted Auth. Upon receiving the backup request from its trusted Auth, an Auth stores the information and credentials into its database tables in order to allow the other Auth's entities to migrate to it in case of the trusted Auth's failure.

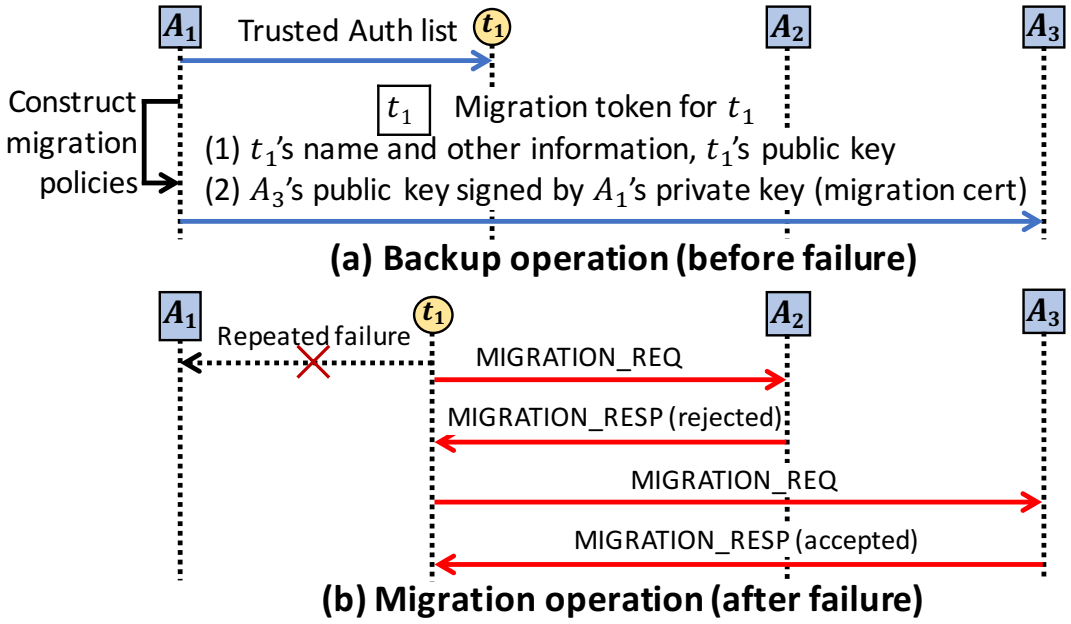


Figure 6.6: Secure migration procedure (a) Backup operation: A_1 updates its Thing with a trusted Auth list and sends a migration token after migration policy construction (b) Migration operation: A_1 fails and its entity t_1 tries to migrate to A_2 or A_3. (Note that the AUTH_HELLO messages, the very first messages containing a nonce to make sure each request is fresh, are omitted for simplicity.)

6.3.3 Detect and Migrate

During an attack or a failure, first, we will need to detect the unavailability of Auths. Each entity has criteria for determining whether its Auth is down. One example is having a threshold for the number of connection failures in a row and down time of Auth; for instance, an entity can determine that its Auth is under a DoS attack or a failure when it fails to reach its Auth more than 10 times in a row for more than 10 minutes. Each entity should have a list of Auths that it can connect to a priori in the case of its Auth's failure. This list includes other Auths' network addresses (host names), ports and underlying protocols (e.g., TCP or UDP).

When an entity detects its Auth's failure, it tries to connect to the Auths in its list and sends a *MIGRATION_REQ (migration request) message*. Whenever any entity connects to an Auth, the Auth sends an *AUTH_HELLO message* that includes a fresh nonce, as explained in Section 3.3.2.

An example of the backup operation is described in Figure 6.6 (a). For its registered entity t_1, Auth A_1 gives out a list of its trusted Auths (A_2 and A_3) and their host names (IP addresses) and port numbers. This is first given during the entity registration and then updated when the information changes. Then, A_1 constructs the migration policies for

its registered entities, either by itself or by receiving migration policies from other trusted Auths, using the method shown in Section 6.2. With the constructed migration policies, A_1 prepares *migration token* for each of its registered entities, and sends it to A_3 to which A_1's entity t_1 will migrate, according to the constructed migration policy.

Figure 6.6 (b) describes an example with a set of migration operations. In this example, an IoT entity, t_1 detects the failure of its Auth, a_1. Then it sends MIGRATION_REQ to a_2, an Auth trusted by a_1, but not the one that t_1 is supposed to migrate to. Since a_2 does not have the required credentials for t_1, a_2 sends a response, MIGRATION_RESP indicating that the request is rejected and t_1 should try another trusted Auth in its list. After receiving MIGRATION_RESP (rejected) from a_2, t_1 tries the next Auth in its list, a_3. When a_3 receives MIGRATION_REQ from t_1, it notices that t_1 can migrate to it and responds with a MIGRATION_RESP message indicating that the request has been accepted.

A MIGRATION_REQ message includes the entity's name and a verification token. This verification token can be a digital signature of the entity if it uses public-key cryptography for authorization. The verification token can be a MAC (message authentication code) if it cannot afford public-key cryptography for authorization. In either case, the Auth that the entity is supposed to migrate to will be able to verify the MIGRATION_REQ either with the entity's public key or the MAC key of the entity's current distribution key. This is because the Auth should have been backed up with the entity's public key or MAC distribution key from the Auth that the entity was previously registered with.

As explained with the example, MIGRATION_RESP can indicate either rejected or accepted. The rejected MIGRATION_RESP is just to say that this is not the Auth which the entity is supposed to be authorized by. The accepted MIGRATION_RESP contains the certificate of the a_3 signed by a_1 if t_1 uses public-key cryptography for authorization, or the migration token for t_1, if t_1 does not use public-key cryptography for authorization. The whole MIGRATION_RESP should be authenticated by the Auth's private key or the MAC key of the previous distribution key, so that the entity can verify the MIGRATION_RESP message upon receiving it. When the migration request and response are successful, both the Auth and entity update the counterpart's credentials for further authorization.

6.4 Experiments and Results

In this section, I carry out experiments to demonstrate the effectiveness of the proposed migration approach for maintaining availability. For the following experiments, I only consider things (IoT entities) that are capable of public-key cryptography. As an experimental scenario, I take a door controller and door opening application in a smart building. This is inspired by a prototype door controller deployed on the 5th floor of Cory Hall at UC Berkeley. I assume a virtual environment where the door controllers are deployed on currently card-key accessed doors, door opening mobile phone apps run on user smart phones, and Auths are deployed on some of the existing WiFi access points. Figure 6.7 illustrates this virtual environment. I also assume Auths have trust relationships depending on research

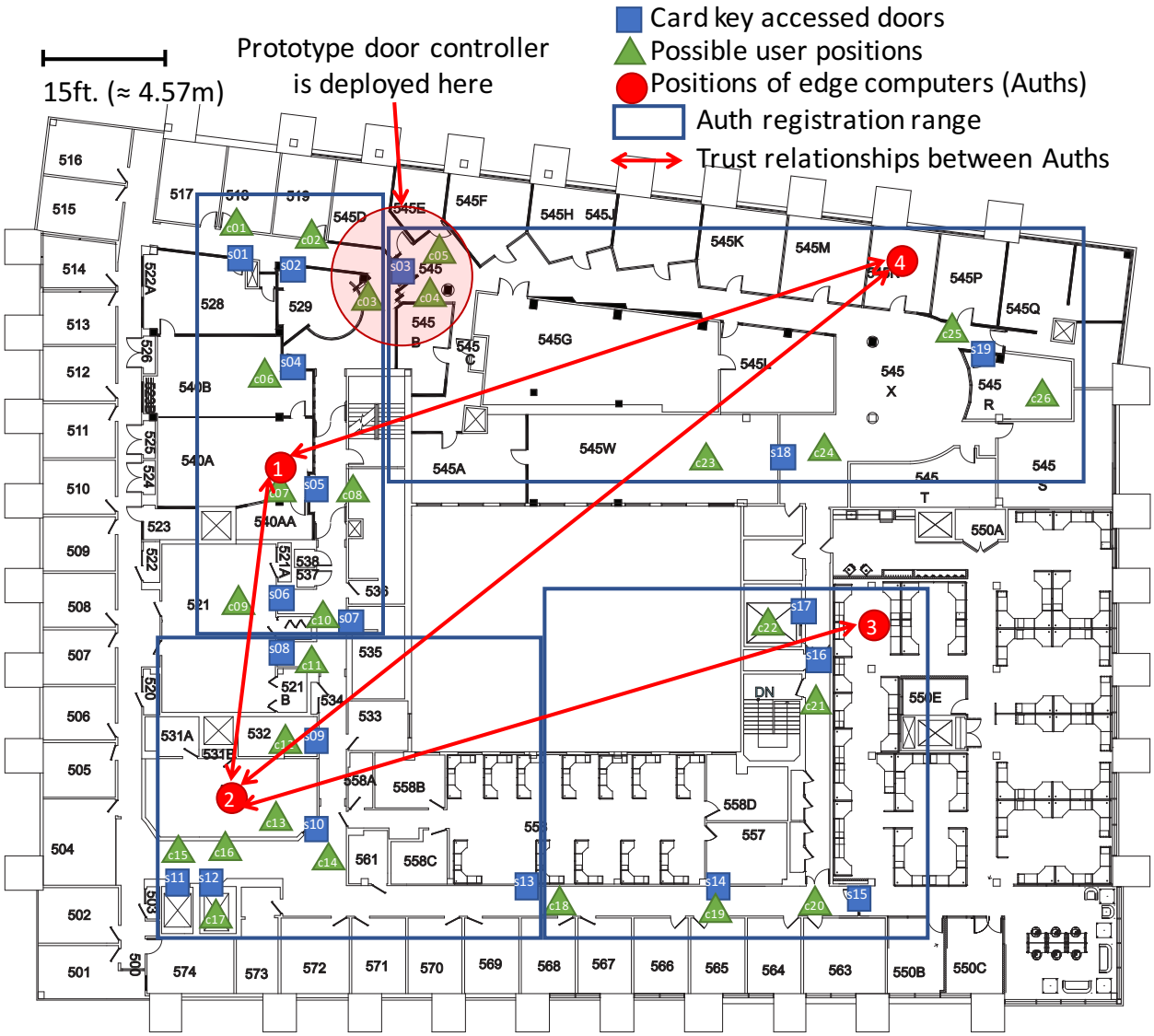


Figure 6.7: Experimental virtual environment with Auths, door controllers, and door opening mobile applications on the floor map of the 5th floor, Cory Hall at UC Berkeley.

centers on the 5th floor, and the Things (door controllers and user smart phones) are registered with Auth as shown in Figure 6.7. I measure availability as the ratio of responses from door controllers (the number of correct door operation) to the door opening requests for a given time window. In the following experimental scenarios, different numbers of Auths can be unavailable due to failures or DoS attacks. In addition, I also compare against scenarios without secure migration and also a scenario where all Auths are unavailable, which is equivalent to the case where an authorization entity is deployed on a remote cloud and the cloud is not reachable.

6.4.1 Experimental Setup

Figure 6.8 describes the experimental setup with the *ns-3 network simulator* [88]. For realistic experiments, I use the actual implementation of Auth available on the GitHub repository and IoT entities (door controllers and opening apps) written using SST's APIs, *secure communication accessors*. Each of Auths and IoT entities runs within an individual *Linux Container (LXC)* [62] which provides OS-level virtualization (paravirtualization) with a separate virtual network space. For simulating the network infrastructure, I use ns-3. The LXCs' virtual Ethernet interfaces are connected to the host OS's Linux bridges, then to the TAP bridges of ns-3 nodes in the ns-3 simulator. The other side of ns-3 nodes are either CSMA or WiFi NetDevice and are connected to the network simulated in ns-3. LXCs on which Auths are running have both the wired and WiFi connections and LXCs for IoT entities have WiFi connections. For connections between Auths' wired network interfaces, I use a CSMA channel with data rate 100Mbps. For connections between the wireless network interfaces of Auths and Things, I use an ad-hoc IEEE 802.11a channel with data rate of 54Mbps. For the channel signal strength model, I use a Log Distance Propagation Loss Model in ns-3 to represent the channels between Auths and things. As a simulation platform, I use Ubuntu Linux 16.04.2 LTS on Amazon's AWS EC2 with 4 CPUs, 16GB RAM, and 256GB SSD.

6.4.2 Simulation Results

I ran simulations with 4 Auths, 19 door controllers, and 26 user devices for 15 minutes for each experiment in real time, 2.5 minutes before Auth failures and 12.5 minutes after failures. Each user device sends a door opening request to the closest door controller every minute, simulating the user behaviors in front of doors.

Figure 6.9 illustrates the experimental results. I performed experiments with up to three Auths failing during the experiments. The failure occurs in order of Auth 1, Auth 3, and Auth 4 where the Auth numbers are as denoted in Figure 6.7. When all four Auths failed, the availability became 0% in our experiments as I expected, although I did not include this in Figure 6.7. It will be the same when the authorization services based on the cloud lose connections with the cloud. I also compared three different migration policies, (1) without any secure migration (original SST, denoted as "*No Migration*"), (2) with a naïve migration policy where the clients (door opening applications) try the nearest available Auth first

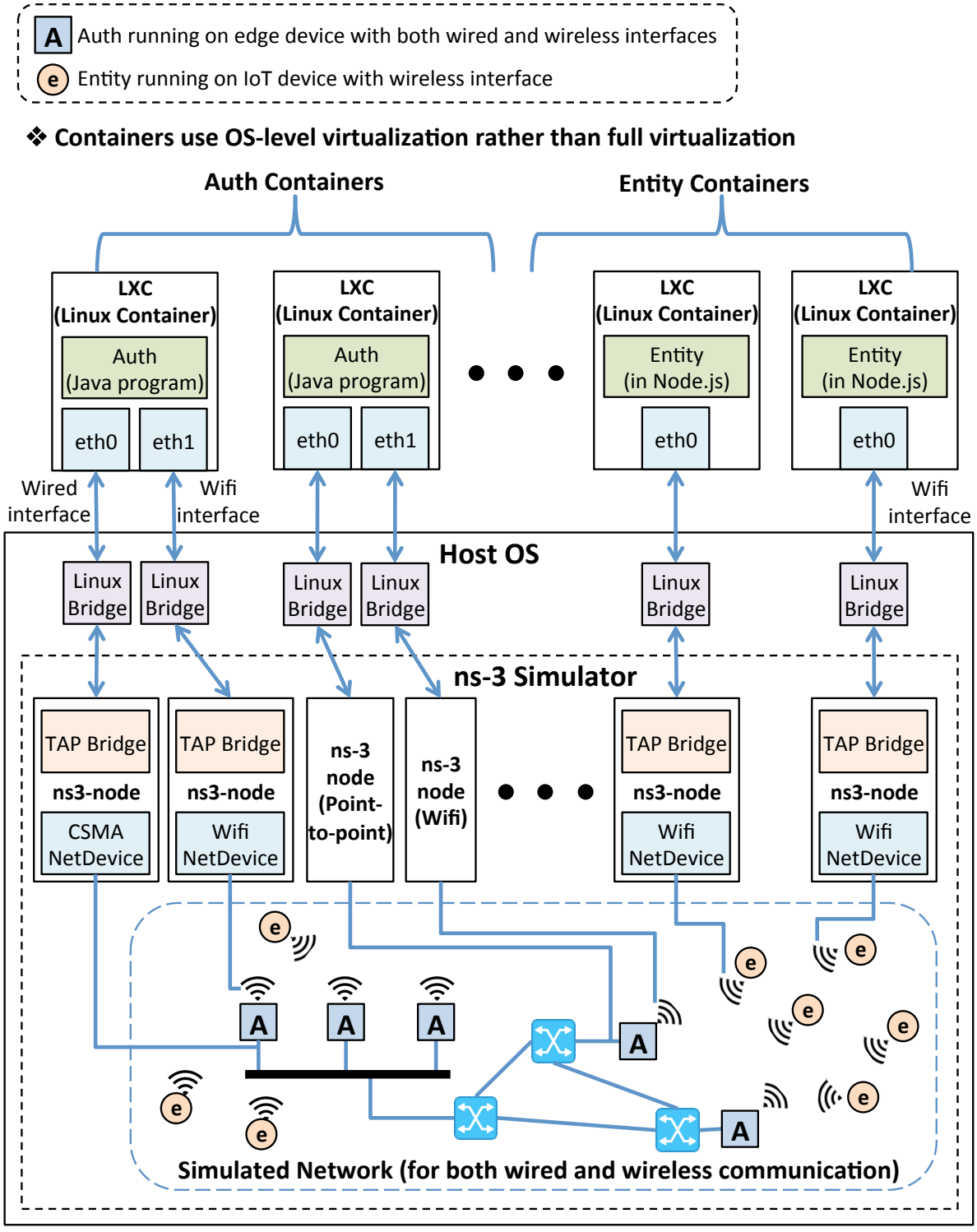


Figure 6.8: Experimental setup with the ns-3 simulator network simulator.

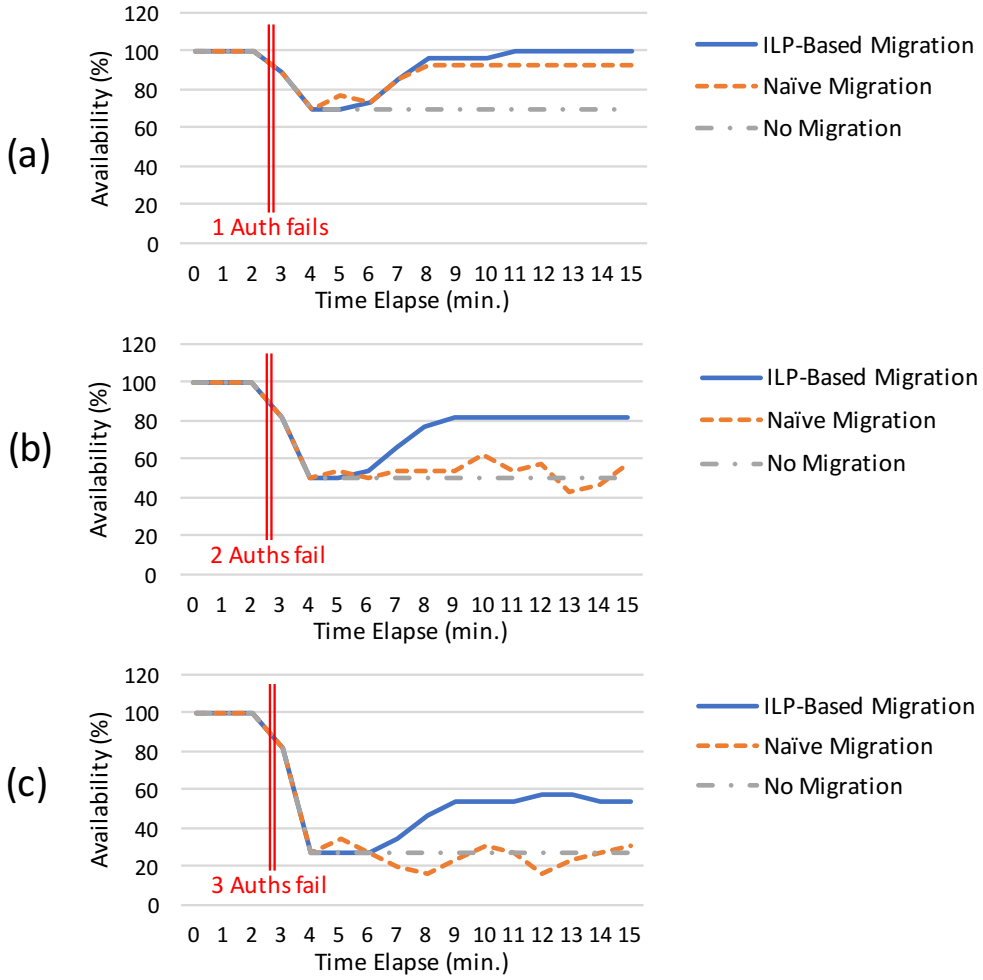


Figure 6.9: Availability results with different numbers of failing Auths for three different migration policies using SST simulated on the ns-3 simulator and Linux containers: (a) When one Auth fails, (b) When two Auths fail, and (c) When three Auths fail.

then try the next nearest Auth, and so on (denoted as "*Naïve Migration*"), and (3) with a migration policy constructed using the proposed approach in Section 6.2 (denoted as "*ILP-Based Migration*" in the results).

As shown in Figure 6.9 (a), when one Auth fails the availability drops down to 69% without any migration, while the naïve migration and ILP-based migration policies recover the availability up to 92% and 100%, respectively. The naïve migration policy could not recover 100% availability due to the fact that it did not consider the trust relationships between Auths properly, thus leading to a migration policy which is infeasible for some IoT entities. In Figure 6.9 (b), we can see that the availability drops down to 50% without any migration and fluctuates around 58% with the naïve migration policy, while the proposed ILP-based solution recovers 81% of availability. Figure 6.9 (c) shows decreased availability around 27% with no or the naïve migration policy and recovered availability of 54% with the proposed technique. The fluctuation with the naïve migration policy was caused mainly by the infeasible migration requests and their interference with normal requests, and unbalanced workload among functioning Auths. The proposed ILP-based solution was not able to achieve 100% availability because of some of the entities were out of the signal ranges of Auths, however, the proposed solution maintained significantly higher availability compared to other two cases.

Chapter 7

Related Work

This chapter summarizes related work with a thorough literature review on the security of the IoT, especially on authentication and authorization. The related work includes security measures for the IoT based on traditional security solutions and network architectures, security solutions for the predecessors of the IoT, and measures to enhance the availability of the IoT against failures or Denial-of-Service (DoS) attacks.

7.1 Using Traditional Security Solutions for the IoT

Secure Socket Layer/Transport Layer Security (SSL/TLS), or simply TLS [28], has been providing security for the Internet. For authentication, TLS uses certificates, usually provided by a certificate authority (CA). However, this cannot be the best option for the IoT due to the overhead for CAs to manage the huge number of certificates. To make TLS with CAs more scalable, the Let's Encrypt project[1] launches free and automated CAs based on a protocol called Automatic Certificate Management Environment (ACME) [10]. Nevertheless, it will be too demanding for resource-constrained devices to carry large certificates and perform computationally expensive asymmetric key (public-key cryptography) operations for every TLS connection.

OpenIoT [96] 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 TLS, which, as we have discussed, is likely to face scalability challenges in dynamic IoT networks.

Kothmayr *et al.* [56] propose an authentication system for the IoT using DTLS [86], a datagram variant of TLS. Hummen *et al.* [48] propose a security framework for IoT devices also based on DTLS. Similar to the proposed 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

[1] `https://letsencrypt.org`

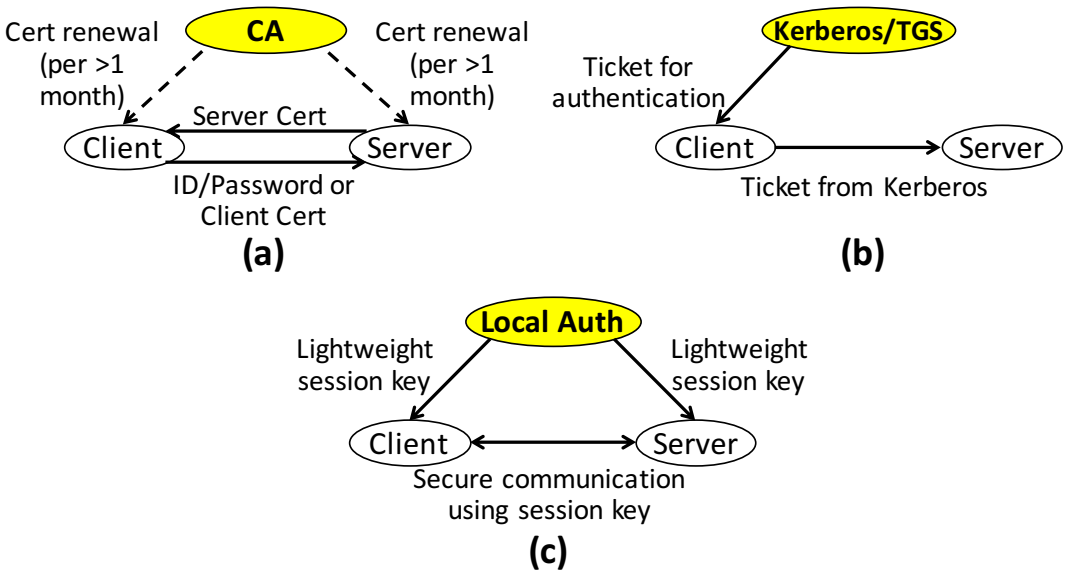


Figure 7.1: Authentication/authorization flows of different approaches; (a) CA (certificate authority) and Certs (certificates). (b) The Kerberos authentication system and TGS (ticket granting service). (c) Proposed approach with locally centralized, globally distributed Auth (Authentication/Authorzation entity).

Figure 3.5, allowing each entity to create its own profile based on its security and resource requirements.

The Constrained Application Protocol (CoAP) [92] 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 and this kind of approach can work better with resource-constrained devices than TLS. However, DTLS still relies on each device to perform public-key operations, or share symmetric keys with other trusted devices before the deployment. Furthermore, CoAP is mainly designed for one-to-one communication (e.g., a client-server model) like TLS, and does not directly support one-to-many settings. This fact makes it challenging to secure such one-to-many communications common in the IoT such as broadcasting or publish-subscribe patterns [35]. In addition, a certificate used in TLS and DTLS contains a unique value for an entity. This risks exposing the entity's identity, leading to a potential threat to privacy.

Another issue with certificate-based approaches is revocation of authentication. As illustrated in Figure 7.1 (a), the CA is only involved in the issuance of certificates, and the client and server authenticate each other using the certificates as long as the certificates are valid. The validity period of certificates is typically longer than several months due to the management overhead of certificates, especially for renewal of certificates. Therefore, it is challenging to revoke authentication of entities using certificates. This can be a potential threat to the safety of critical components in the IoT in case they are compromised.

For authentication of entities, the Kerberos authentication system [75] issues temporary tickets through its ticket granting service (TGS), as illustrated in Figure 7.1 (b). Thus, Kerberos provides a centralized control over the validity period of authentication, addressing the challenges of access revocation. However, such systems are designed for human users, requiring user intervention such as entering passwords. This makes it hard to support automated mutual authentication of IoT devices.

There are extensions for Kerberos [112] that use public key cryptography for authentication, to replace the human intervention. Even with these extensions, however, clients with intermittent connectivity may face authentication problems when they are not connected to Kerberos/TGS. Although caching tickets can address the intermittent connectivity problem, it creates another problem of allowing a compromised client with cached tickets to authenticate with the server. This is because a ticket is delivered to a server by a client, not by Kerberos/TGS, as shown in Figure 7.1 (b), and the server trusts the client as long as the ticket is valid.

The authentication flows for certificate-based approaches and the Kerberos authentication system are designed for the networks with general-purpose computers. Although these approaches have been successful for the traditional Internet, we note that the authentication flows shown in Figure 7.1 (a) and (b) cannot address some of the IoT-related security and scalability issues. Therefore, we propose a security framework that has the authentication/authorization flow as shown in Figure 7.1 (c).

In the proposed approach, the local authorization entity, *Auth*, assigns lightweight session keys to entities involved in communication. The Auth entity controls the validity period of session keys and covers authorization as well as authentication. This is possible because Auth is aware of communication context of registered entities, determining whether an entity is authorized to communicate with others. While Auth serves as a local point of authorization, it also interacts with other Auths to control communication between entities registered with different Auths, distributing the authorization overhead locally. Potential deployment targets for Auth include edge computers covered in Section 2.2.

7.2 Contemporary Security Solutions for the IoT

There are many current approaches proposed for authorization and authentication of the IoT. Most of these approaches at least partly rely on remote and centralized cloud servers for providing authentication and authorization for the IoT. IoT-OAS [23] uses OAuth servers for providing access control for third-party applications in the IoT. AOT (Authentication of Things) [74] offers various authentication schemes and uses a cloud server to control trust relationships between the manufacturer and devices. OpenIoT [96] is primarily based on cloud servers and adopts flexible methods for authentication and authorization of the IoT by leveraging CAS (Central Access Control) services. OSCAR [103] provides integrated security architecture of the cloud and IoT using authorization servers. Amazon's AWS IoT [45] takes advantage of its well-known cloud infrastructure, AWS, for authentication and authorization

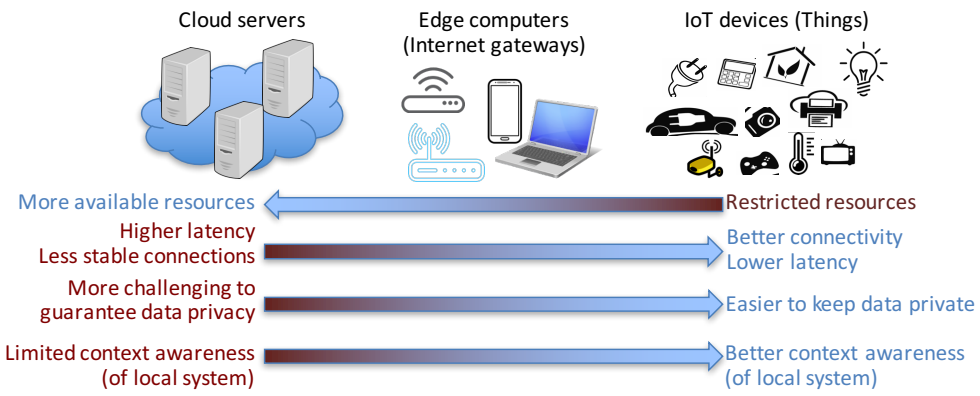


Figure 7.2: Characteristics of cloud, edge and things.

of IoT devices. However, as pointed out in the Google's OnHub incident [70] in Section 1.1, depending too much on remote cloud servers can render the system vulnerable to connection failures or availability attacks.

An alternative system architecture that reduces the dependency on remote servers, called *edge computing* [93] or *fog computing* [64], is based on computers serving as Internet gateways (e.g., smart home routers, laptops, smart phones). Cloud computing and edge computing are often reckoned to be complementary rather than exclusive. Figure 7.2 illustrates different characteristics of the cloud servers, edge computers, and IoT devices. As pointed out in Lopez *et al.* [63], benefits of adopting edge computing include better privacy, lower latency for real-time applications, less dependency on cloud servers and its connections, and better context awareness and manageability of the local systems. Thus, edge computing creates an opportunity for a more robust and efficient authorization service infrastructure for the IoT.

A few approaches leverage edge computing for constructing authorization service infrastructure for the IoT. To provide robust authentication and authorization of medical IoT devices without depending on remote servers, SEA (Secure and Efficient Authentication and Authorization Architecture) [69] uses distributed smart e-health gateways as local authorization centers based on DTLS (Datagram TLS). TACIoT (Trust-aware Access Control for the IoT) [12] employs an architecture called IoT bubbles as local units of authorization of the IoT. The toolkit proposed in this dissertation, SST (Secure Swarm Toolkit) [53], constructs an authorization infrastructure for the IoT based on an open-source local authorization entity called *Auth*. Auth in SST runs on edge devices and works as a local center of access control for the locally registered things.

7.3 Security Measures for Predecessors of the IoT

The concept of connected "Things" has been existed for several decades under different names including The Swarm [59], Smart Dust [105], Mobile Ad-hoc Networks (MANET) [27], and

Wireless Sensor Networks (WSN). A variety of security frameworks for WSN have been proposed and studied [11], [52], [65], [84], [85] to support automated authentication for resource-constrained devices. 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.

Approaches using random or pairwise pre-distributed encryption keys [30], [31] provide energy efficient solutions. There has been a static key management approach that allows key establishment for newly added nodes [37]. However, static key management systems may not be proper for the safety-critical devices running under hostile environments, due to the increased probability of being attacked for the cryptographic key with a long lifetime.

He *et al.* [42] survey dynamic key management systems for WSN, which can address the problem with pre-distributed keys by supporting key revocation mechanisms. These approaches for WSN can be categorized into *distributed* and *centralized approaches*. In *distributed approaches* including EDDK [111] (Energy-efficient Distributed Deterministic Key management), neighboring nodes collaborate to dynamically establish keys. TinyPBC Pairing-Based Cryptography (PBC) provides key agreement without any network interaction [77] to save energy for communication especially for carrying certificates. To consider resource constraints in certificate-based approaches, there have been various methods such as MOCA [108], a distributed, mobile certificate authority for mobile ad-hoc networks. Such approaches can avoid a single point of failure in authentication systems; however, they tend to be more vulnerable to collusion attacks and prone to design errors.

In *centralized approaches*, a central trusted third party (e.g., a base station) is responsible for key generation and distribution for nodes. Many of these approaches have similar authentication flows as the proposed approach's authentication flow in Figure 7.1 (c). Huang *et al.* [46] propose a centralized forward authentication approach for hierarchical and heterogeneous sensor networks composed of high-end and low-end sensor nodes. Sahingoz [90] provides a key distribution system using an unmanned aerial vehicle (UAV) as a center of key distribution and coordination, for large scale WSNs. To support addition and deletion of mobile sensor nodes, Erfani *et al.* [33] propose a key management system that uses key pre-distribution and post-deployment key establishment.

The dynamic key management systems for WSN can address some part of the IoT-related security requirements in Section 1.2.2. However, they still have limited results to cover the vast heterogeneity of devices in the IoT, from the resource-constrained sensor nodes to the critical components that require frequent authentication and authorization for safety. The support for the dynamic environment such as intermittent connectivity and dynamic entity registration is still not sufficient.

There have been other research efforts for the security of the IoT from a diversity of angles. Seitz *et al.* [91] 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 [39]. However, their approach is not based on a particular authentication scheme and does not directly address scalability issues. Wei

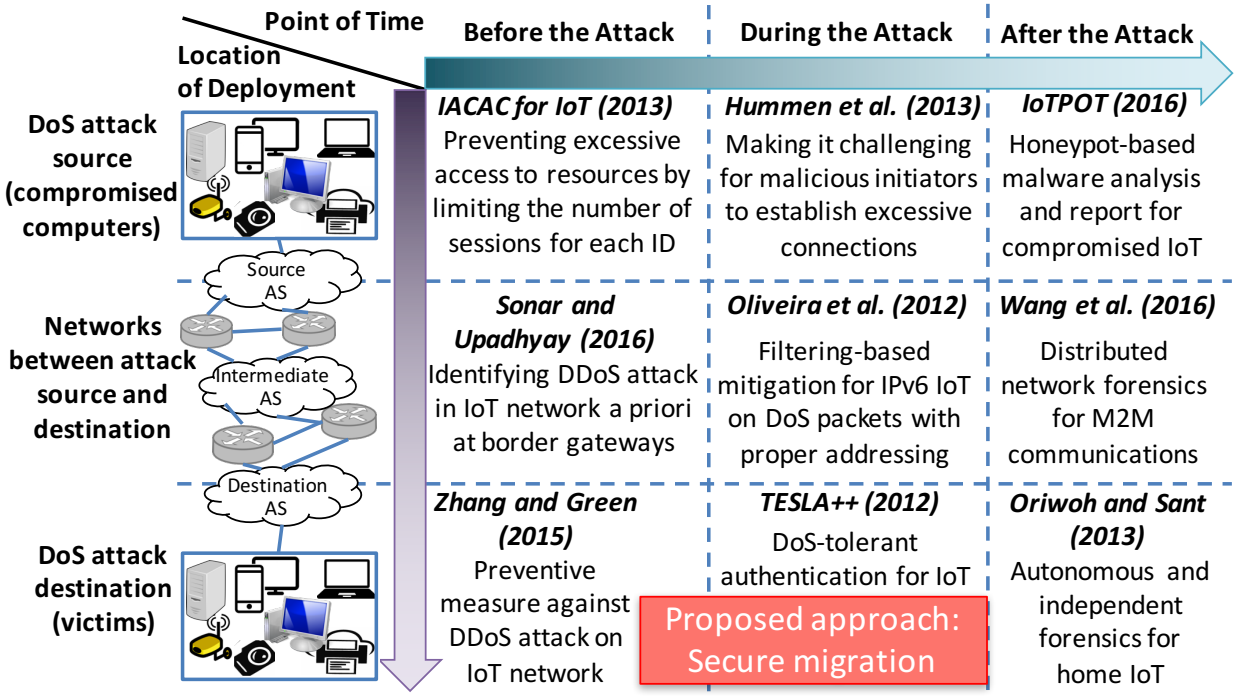


Figure 7.3: Countermeasures against DoS attacks to the IoT classified by two criteria used in [109], deployment location and point of time.

et al. [106] 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 [21] provides a secure mechanism for integrating heterogeneous wireless networks including cellular and WLANs. Like SST, SHAWK addresses heterogeneity by integrating existing solutions but at a different layer of abstraction.

7.4 Defense Against Availability Attacks on the IoT

Although I do not limit the scope of DoS attacks to DDoS attacks in this dissertation, I use the DDoS attack classification criteria presented by Zargar *et al.* [109], to position the proposed secure migration technique in Chapter 6. Figure 7.3 illustrates a variety of countermeasures and mitigations against the availability attacks to the IoT or the similar networked systems. The horizontal axis in Figure 7.3 shows the stages of a DoS attack broken into three, according to the point of time when the measure takes effect relative to the attack: before the attack, during the attack, and after the attack. The vertical axis shows three categories of DoS attack countermeasures based on their deployment locations: at potential attack sources, on networks between the attacker and the target, and at attack

destinations – the victims.

Before the attack, a preventive measure can be taken at the attack source such as Identity Authentication and Capability Based Access Control (IACAC) for the IoT [66], which limits excessive access activity by imposing capability associated to IoT devices. An approach proposed by Sonar and Upadhyay [97] can prevent the DDoS attack traffic from reaching the IoT devices by identifying the attack traffic a priori at border gateways. Zhang and Green [110] have proposed a lightweight defensive algorithm for an IoT end network by leveraging distributed intelligence at IoT end nodes to detect and prevent malicious packet streams.

There have been DoS mitigations proposed for the IoT, which take effect while a DoS attack is happening. Hummen *et al.* [49] have presented an approach which makes it difficult for malicious entities to create excessive connections from the attack source. The solution proposed by Oliveira *et al.* [78] mitigates the DoS traffic with filtering for the IPv6-based IoT. TESLA++ [89] supports DoS-tolerant authentication at the destination by extending the original TESLA (Timed Efficient Stream Loss-tolerant Authentication) protocol [83].

Forensics and log analysis-based approaches for the IoT have been proposed at a variety of deployment locations for investigation after DoS attacks. Such approaches include a honeypot-based malware analysis, IoTPOT [80], which is used to track and report malware that can potentially compromise the IoT devices and use them as zombie computers for launching DDoS attacks. Forensics using distributed anti-honeypots has been proposed by Wang *et al.* [104] for M2M communication (machine-to-machine, another name for the IoT) to be deployed on networks between the attack source and destination. As a solution to be deployed on the victim side, Oriwoh *et al.* [79] have proposed a user-manageable forensics solution for the home IoT.

The secure migration mechanism presented in Chapter 6 maintains the availability of the IoT services by migrating Things while Auths are under attack, thus it falls into the category of the victim side, during the attack, as marked in Figure 7.3. Since the secure migration approach involves migration policy construction and backup operations before the attack, it has some overlaps with the "before the attack" category. Although TESLA++ falls into the same category, the proposed secure migration technique is a more comprehensive approach that covers more diverse IoT devices and cryptography, while TESLA++ focuses on broadcasting network and message authentication.

Chapter 8

Conclusions

8.1 Conclusions

Security schemes solely based on centralized trust do not take advantage of emerging edge-computing devices and could face the problems of a single point of failure. Fully distributed solutions may not be practical for the IoT due to the overhead on individual IoT devices, especially resource-constrained devices. I envision the authentication and authorization infrastructure for the IoT to be locally centralized and globally distributed, which is achievable by local authorization entities based on globally distributed trust among these authorization entities.

In this dissertation, I propose a locally centralized and globally distributed authentication and authorization infrastructure to address IoT-related security requirements, as summarized in Table 8.1. The proposed approach has been realized as a novel toolkit for constructing an authorization service infrastructure for the IoT called the Secure Swarm Toolkit (SST). The proposed approach supports frequent, automated authentication and authorization by using a local authorization entity called Auth. Auth authorizes registered entities through session key distribution. By caching the session keys and allowing a variety of cryptographic algorithms, even the entities with intermittent connectivity or resource constraints can be authorized effectively. For authentication and authorization, an entity only needs to use temporary session keys provided by Auth. Thus, it does not have to risk exposing its identity by using its unique value such as a certificate, maintaining its privacy. This dissertation also provides a rigorous, formal security analysis to ensure that SST meets necessary security guarantees. This analysis shows SST's scalability, and the experimental results illustrate security overheads under a range of different security configurations.

Availability of IoT services can be critical for the system's safety. By leveraging emerging network architecture based on edge computing and SST's distributed authorization infrastructure, the proposed approach achieves much higher availability even under failures of local authorization entities running on edge computers. The proposed secure migration approach will be appropriate especially for the Internet of Things under safety-critical environments

Table 8.1: How the proposed approach addresses IoT-related security requirements introduced in Section 1.2.2.

IoT Security Requirements	**Proposed Approach**
Frequent authentication and authorization	Auth controls every secure communication, and it can enforce short key validity periods of session keys
Automated mutual authentication	Auth provides fully automated authentication; no human intervention is required except for entity registration
Intermittent connectivity	Auth allows use of cached session keys
Dynamic registration of IoT entities	An entity can be seamlessly registered/unregistered with Auth, without interrupting other entities
Support for scalability features	Session keys can be shared by more than two entities for one-to-many communication (e.g., publish-subscribe)
Consideration for resource constraints	Small and lightweight symmetric session keys are used for authentication; Auth allows various cryptographic algorithms for resource-constrained devices, including the ones that cannot afford public key cryptography
Privacy	No unique identifier is needed for authentication, thanks to the use of temporary session keys
Resiliency and Robustness	SST's secure migration technique can mitigate the effect of failures of Auths or DoS attacks on edge computers hosting Auths, by migrating Things to other trusted available Auths.
Locality	Auths are designed to be deployed local edge-computing devices without dependence on remote servers for authentication and authorization
Ease of deployment	SST is accessible to public as an open-source toolkit, and secure communication accessors help developers build IoT applications and integrate with SST.

including medical centers, manufacturing systems, and electric power grids, so that the proposed infrastructure can maintain as much availability as possible.

I 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. I 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.

8.2 Remaining Challenges and Future Work

There are still challenges that need to be addressed for further secure authorization infrastructure. Auth is fully automated once entity registration is completed. During registration, Auth and the entity to be registered set up credentials, security configurations, and access control policies. In many existing security solutions, this initialization process is quite costly. However, to cope with scalability and dynamically added and removed IoT devices, there should be an automated or semi-automated registration process. One possibility is to exploit physical proximity, where the ability to place a device within a few centimeters of an edge computing server establishes a trust relationship. Another remaining challenge is dealing with authorization for mobile devices. I envision this can be done in a similar way as the current cellular network, which deals with cellular handoff (changing cell towers as a mobile phone moves).

To enhance the availability of Auth under a denial-of-service attack and avoid being a single point of failure, a distributed implementation of Auth will be required. To provide further security guarantees, we can use Auth as a point of intrusion detection, since it can see access-related activities of local devices. Auth also can serve as a software attestation center to guarantee that the IoT device is running a legitimate program not tampered with by adversaries. Further studies need to be carried out on the usability of the SST's software building blocks for accessing Auth and IoT services.

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 problems include timely detecting malicious behavior in the IoT and providing guarantees for swarm applications running remotely on untrusted IoT platforms.

Another remaining problem is to improve the proposed defense and mitigation against denial-of-service attacks breaching availability. It will be necessary to investigate the use of an ILP solver to construct a migration policy that is not only valid but also *optimal* with respect to the overall network costs. The effectiveness and limitations of the proposed migration mechanism will be further evaluated by carrying out larger scale experiments.

Another future direction is to evaluate the proposed secure migration approach on examples with a more diverse range of security requirements and resource availability.

Bibliography

[1] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, Jul. 2012.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song, "Towards a formal foundation of web security," in *23rd IEEE Computer Security Foundations Symposium, Edinburgh, UK*, 2010, pp. 290–304.

[3] F. Aloul, S. Zahidi, and W. El-Hajj, "Two factor authentication using mobile phones," in *IEEE/ACS International Conference on Computer Systems and Applications*, May 2009, pp. 641–644.

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

[5] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, "Evaluating the small scope hypothesis," MIT CSAIL, Tech. Rep., 2003.

[6] K. R. Apt and D. Kozen, "Limits for automatic verification of finite-state concurrent systems," *Inf. Process. Lett.*, vol. 22, no. 6, pp. 307–309, 1986.

[7] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[8] A. Banks and R. Gupta, *MQTT version 3.1.1*, OASIS Standard, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 2014.

[9] E. Barker, "Recommendation for key management – Part 1: General," *NIST Special Publication 800-57: Part 1 (Revision 4)*, Jan. 2016.

[10] R. Barnes, J. Hoffman-Andrews, and J. Kasten, *Automatic certificate management environment (ACME)*, IETF Internet-Draft Version 7, ACME Working Group, IETF, Jun. 2017.

[11] P. Baronti, P. Pillai, V. W. C. Chook, S. Chessa, A. Gotta, and Y. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and zigbee standards," *Comput. Commun.*, vol. 30, no. 7, pp. 1655–1695, 2007.

[12] J. B. Bernabe, J. L. H. Ramos, and A. F. S. Gomez, "TACIoT: Multidimensional trust-aware access control system for the Internet of Things," *Soft Computing*, vol. 20, no. 5, pp. 1763–1779, May 2016.

[13] P. A. Bernstein and S. Bykov, "Developing cloud services using the Orleans virtual actor model," *IEEE Internet Computing*, vol. 20, no. 5, pp. 71–75, 2016.

[14] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, ser. Studies in Computational Intelligence 546, Springer International Publishing, 2014, pp. 169–186.

[15] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop*, New York, NY, USA: ACM, 2012, pp. 13–16.

[16] A. Botta, W. d. Donato, V. Persico, and A. Pescapé, "On the integration of cloud computing and Internet of Things," in *2014 International Conference on Future Internet of Things and Cloud*, 2014, pp. 23–30.

[17] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of IEEE*, 2017, to appear.

[18] A. Burns and R. Davis, "Mixed criticality systems: A review," *Dept. of Computer Science, University of York, Tech. Rep, Sixth Edition*, Jan. 2015.

[19] I. Butun, S. D. Morgera, and R. Sankar, "A Survey of Intrusion Detection Systems in Wireless Sensor Networks," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 266–282, 2014.

[20] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, *OpenPGP message format*, RFC 4880, IETF, Nov. 2007.

[21] J. Cao *et al.*, "SHAWK: Platform for Secure Integration of Heterogeneous Advanced Wireless Networks," in *26th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Mar. 2012, pp. 13–18.

[22] P. Cerwall *et al.*, "Ericsson mobility report," Tech. Rep., Jun. 2017. [Online]. Available: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf`.

[23] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, "IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios," *IEEE Sensors Journal*, vol. 15, no. 2, pp. 1224–1234, Feb. 2015.

[24] "Cisco global cloud index: Forecast and methodology, 2015–2020," Cisco Public, Tech. Rep., 2016.

[25] Cisco Public White paper, "Cisco visual networking index: Forecast and methodology, 2016-2021," Tech. Rep., Jun. 2017. [Online]. Available: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf`.

[26] J. C. Corbett *et al.*, "Spanner: Google's Globally Distributed Database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 8:1–8:22, Aug. 2013.

[27] S. Corson and J. Macker, *Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations*, RFC 2501, IETF, Jan. 1999. [Online]. Available: `https://tools.ietf.org/html/rfc2501.html`.

[28] T. Dierks and E. Rescorla, *The transport layer security (TLS) protocol version 1.2*, RFC 5246, IETF, Aug. 2008.

[29] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed SDN controller," in *Proc. of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13, New York, NY, USA: ACM, 2013, pp. 7–12.

[30] W. Du, J. Deng, Y. Han, S. Chen, and P. Varshney, "A key management scheme for wireless sensor networks using deployment knowledge," in *INFOCOM 2004. Twenty-third Annu. Joint Conf. of the IEEE Comput. and Commun. Societies*, vol. 1, Mar. 2004, p. 597.

[31] W. Du, J. Deng, Y. S. Han, P. K. Varshney, J. Katz, and A. Khalili, "A pairwise key predistribution scheme for wireless sensor networks," *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 2, pp. 228–258, May 2005.

[32] J. Eidson, C. Jerad, H. Kim, M. Lohstroh, E. A. Lee, V. Nouvellet, and B. Osyk, "Reconciling safety with the internet for cyber- physical systems," *Special Issue on Formal Aspects of Computing*, 2017, under review.

[33] S. H. Erfani, H. H. Javadi, and A. M. Rahmani, "A dynamic key management scheme for dynamic wireless sensor networks," *Security and Communication Networks*, vol. 8, no. 6, pp. 1040–1049, Apr. 2015.

[34] P. Eronen and H. Tschofenig, *Pre-shared key ciphersuites for transport layer security (TLS)*, RFC 4279, IETF, Dec. 2005.

[35] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[36] L. M. Feeney and M. Nilsson, "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, 2001, pp. 1548–1557.

[37] F. Gandino, B. Montrucchio, and M. Rebaudengo, "Key management for static wireless sensor networks with node adding," *IEEE Trans. on Industrial Informatics*, vol. 10, no. 2, pp. 1133–1143, May 2014.

[38] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman, "Green Lights Forever: Analyzing the Security of Traffic Infrastructure," in *The 8th USENIX Workshop on Offensive Technologies (WOOT '14)*, San Diego, CA, Aug. 2014.

[39] S. Godik and T. Moses, *eXtensible Access Control Markup Language (XACML)*, OASIS Standard Version 2.0, http://www.oasis-open.org/committees/xacml, 2005.

[40] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[41] Gurobi Optimization, Inc., *Gurobi optimizer reference manual*, 2016. [Online]. Available: `http://www.gurobi.com`.

[42] X. He, M. Niedermeier, and H. de Meer, "Dynamic key management in wireless sensor networks: A survey," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 611–622, Mar. 2013.

[43] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, Jun. 1977.

[44] S. Hilton, *Dyn Analysis Summary Of Friday October 21 Attack | Dyn Blog*, Oct. 2016. [Online]. Available: `http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/`.

[45] *How the AWS IoT Platform Works - Amazon Web Services*. [Online]. Available: `http://aws.amazon.com/iot-platform/how-it-works/`.

[46] J.-Y. Huang, I.-E. Liao, and H.-W. Tang, "A forward authentication key management scheme for heterogeneous sensor networks," *EURASIP Journal on Wirel. Commun. Netw.*, vol. 2011, 6:1–6:10, Jan. 2011.

[47] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-Physical Systems Security – A Survey," *arXiv:1701.04525[cs]*, Jan. 2017.

[48] R. Hummen, H. Shafagh, S. Raza, T. Voig, and K. Wehrle, "Delegation-based authentication and authorization for the IP-based Internet of Things," in *11th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, Jun. 2014, pp. 284–292.

[49] R. Hummen, H. Wirtz, J. H. Ziegeldorf, J. Hiller, and K. Wehrle, "Tailoring end-to-end IP security protocols to the Internet of Things," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct. 2013, pp. 1–10.

[50] G. Hurlburt, "Might the Blockchain Outlive Bitcoin?" *IT Professional*, vol. 18, no. 2, pp. 12–16, Mar. 2016.

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

[52] C. Karlof, N. Sastry, and D. Wagner, "Tinysec: A link layer security architecture for wireless sensor networks," in *SenSys 2004*, Baltimore, MD, USA, Nov. 2004, pp. 162–175.

[53] H. Kim, E. Kang, E. A. Lee, and D. Broman, "A toolkit for construction of authorization service infrastructure for the internet of things," in *The 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, Pittsburgh, PA: ACM/IEEE, Apr. 2017, pp. 147–158.

[54] H. Kim and E. A. Lee, "Authentication and Authorization for the Internet of Things," *IT Professional*, vol. 19, no. 5, 2017, to appear.

[55] H. Kim, A. Wasicek, B. Mehne, and E. A. Lee, "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, Aug. 2016, pp. 114–122.

[56] T. Kothmayr, C. Schmitt, W. Hu, M. Brünig, and G. Carle, "DTLS based security and two-way authentication for the Internet of Things," *Ad Hoc Networks*, vol. 11, no. 8, pp. 2710–2723, Nov. 2013.

[57] D. Lagutin, K. Visala, A. Zahemszky, T. Burbridge, and G. Marias, "Roles and security in a publish/subscribe network architecture," in *2010 IEEE Symp. on Comput. and Commun. (ISCC)*, Jun. 2010, pp. 68–74.

[58] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber, "A Vision of Swarmlets," *IEEE Internet Computing*, vol. 19, no. 2, pp. 20–28, Mar. 2015.

[59] E. A. Lee *et al.*, "The swarm at the edge of the cloud," *IEEE Design Test*, vol. 31, no. 3, pp. 8–20, Jun. 2014.

[60] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-Oriented Design of Embedded Hardware and Software Systems," *Journal of Circuits, Systems and Computers*, vol. 12, no. 03, pp. 231–260, Jun. 2003.

[61] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the Ukrainian power grid," *SANS Industrial Control Systems*, 2016.

[62] *Linux Containers - LXC - Introduction*. [Online]. Available: `https://linuxcontainers.org/lxc/`.

[63] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric Computing: Vision and Challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[64] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun, "Fog Computing: Focusing on Mobile Users at the Edge," *arXiv:1502.01815 [cs]*, Feb. 2015, arXiv: 1502.01815. [Online]. Available: `http://arxiv.org/abs/1502.01815`.

[65] M. Luk, G. Mezzour, A. Perrig, and V. D. Gligor, "Minisec: A secure sensor network communication architecture," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN) '07*, Cambridge, MA, USA, Apr. 2007, pp. 479–488.

[66] P. N. Mahalle, B. Anggorojati, N. R. Prasad, and R. Prasad, "Identity Authentication and Capability Based Access Control (IACAC) for the Internet of Things," *Journal of Cyber Security and Mobility*, vol. 1, no. 4, pp. 309–348, 2013.

[67] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel(R) Software Guard Extensions (Intel(R) SGX) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ser. HASP 2016, New York, NY, USA: ACM, 2016, 10:1–10:9.

[68] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, Sep. 2012.

[69] S. R. Moosavi, T. N. Gia, A.-M. Rahmani, E. Nigussie, S. Virtanen, J. Isoaho, and H. Tenhunen, "SEA: A Secure and Efficient Authentication and Authorization Architecture for IoT-Based Healthcare Using Smart Gateways," *Procedia Computer Science*, The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), vol. 52, pp. 452–459, Jan. 2015.

[70] I. Morris, "Google's Latest Failure Shows How Immature Its Hardware Is," *Forbes*, Feb. 2017. [Online]. Available: `http://www.forbes.com/sites/ianmorris/2017/02/24/googles-latest-failure-shows-how-immature-its-hardware-is/`.

[71] B. A. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, pp. 62–69, May 2016.

[72] S. Narain, "Network configuration management via model finding," in *Proceedings of LISA '05*, vol. 5, 2005, pp. 15–15.

[73] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "The margrave tool for firewall analysis," in *LISA, San Jose, CA, USA*, 2010.

[74] A. L. M. Neto *et al.*, "AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, ser. SenSys '16, New York, NY, USA: ACM, 2016, pp. 1–15.

[75] C. Neuman, T. Yu, S. Hartman, and K. Raeburn, *The Kerberos network authentication service (V5)*, RFC 4120, IETF, Jul. 2005.

[76] K. T. Nguyen, M. Laurent, and N. Oualha, "Survey on secure communication protocols for the Internet of Things," *Ad Hoc Networks*, Internet of Things security and privacy: design methods and optimization, vol. 32, pp. 17–31, Sep. 2015.

[77] L. B. Oliveira, D. F. Aranha, C. P. L. Gouvêa, M. Scott, D. F. Címara, J. López, and R. Dahab, "TinyPBC: Pairings for authenticated identity-based non-interactive key distribution in sensor networks," *Comput. Commun.*, vol. 34, no. 3, pp. 485–493, Mar. 2011.

[78] L. M. L. Oliveira, J. J.P. C. Rodrigues, A. F. de Sousa, and J. Lloret, "Denial of service mitigation approach for IPv6-enabled smart object networks," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 1, pp. 129–142, Jan. 2013.

[79] E. Oriwoh and P. Sant, "The Forensics Edge Management System: A Concept and Design," in *2013 IEEE 10th International Conference on Ubiquitous Intelligence and Computing and 2013 IEEE 10th International Conference on Autonomic and Trusted Computing*, Dec. 2013, pp. 544–550.

[80] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoT-POT: A Novel Honeypot for Revealing Current IoT Threats," *Journal of Information Processing*, vol. 24, no. 3, pp. 522–533, May 2016.

[81] A. Pal, "Internet of Things: Making the Hype a Reality," *IT Professional*, vol. 17, no. 3, pp. 2–4, May 2015.

[82] K. Pelechrinis, M. Iliofotou, and S. V. Krishnamurthy, "Denial of Service Attacks in Wireless Networks: The Case of Jammers," *IEEE Communications Surveys Tutorials*, vol. 13, no. 2, pp. 245–257, 2011.

[83] A. Perrig, R. Canetti, J. Tygar, and D. Song, "The TESLA broadcast authentication protocol," *RSA CryptoBytes*, Jul. 2005.

[84] A. Perrig, J. A. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[85] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "SPINS: Security Protocols for Sensor Networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 521–534, Sep. 2002.

[86] E. Rescoria and N. Modadugu, *Datagram transport layer security version 1.2*, RFC 6347, IETF, Jan. 2012.

[87] H. Rifà-Pous and J. Herrera-Joancomartí, "Computational and energy costs of cryptographic algorithms on handheld devices," *Future Internet*, vol. 3, no. 1, pp. 31–48, Feb. 2011.

[88] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator," in *Modeling and Tools for Network Simulation*, Springer Berlin Heidelberg, 2010, pp. 15–34.

[89] N. Ruan and Y. Hori, "DoS attack-tolerant TESLA-based broadcast authentication protocol in Internet of Things," in *2012 International Conference on Selected Topics in Mobile and Wireless Networking*, Jul. 2012, pp. 60–65.

[90] O. K. Sahingoz, "Large scale wireless sensor networks with multi-level dynamic key management scheme," *Journal of Systems Architecture*, vol. 59, no. 9, pp. 801–807, Oct. 2013.

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

[92] Z. Shelby, K. Hartke, and C. Bormann, *The constrained application protocol (CoAP)*, RFC 7252, IETF, Jun. 2014.

[93] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[94] C.-Y. Shih *et al.*, "On the Cooperation between Mobile Robots and Wireless Sensor Networks," in *Cooperative Robots and Sensor Networks 2014*, 554, Springer Berlin Heidelberg, 2014, pp. 67–86.

[95] J. Singh, T. Pasquier, J. Bacon, H. Ko, and D. Eyers, "Twenty Security Considerations for Cloud-Supported Internet of Things," *IEEE Internet of Things Journal*, vol. 3, pp. 269–284, Jun. 2016.

[96] J. Soldatos *et al.*, "OpenIoT: Open Source Internet-of-Things in the Cloud," in *Interoperability and Open-Source Solutions for the Internet of Things*, 9001, Springer International Publishing, 2015, pp. 13–25.

[97] K. Sonar and H. Upadhyay, "An Approach to Secure Internet of Things Against DDoS," in *Proceedings of International Conference on ICT for Sustainable Development*, Springer, Singapore, 2016, pp. 367–376.

[98] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[99] *TPM (Trusted Platform Module) main specification*, Revision 116, Trusted Computing Group, Mar. 2011.

[100] P. Traverse, I. Lacaze, and J. Souyris, "Airbus Fly-By-Wire: A total approach to dependability," in *Building the Inform. Soc.* Ser. IFIP Intl. Federation for Inform. Process. 156, Springer, 2004, pp. 191–212.

[101] L. Tung, "Mozilla to China's WoSign: We'll kill Firefox trust in you after mis-issued GitHub certs," *ZDNet*, Sep. 27, 2016. [Online]. Available: `http://www.zdnet.com/article/mozilla-to-chinas-wosign-well-kill-firefox-trust-in-you-after-mis-issued-github-certs/`.

[102] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos, "Challenges and opportunities in edge computing," in *2016 IEEE Int. Conf. on Smart Cloud (SmartCloud)*, 2016, pp. 20–26.

[103] M. Vučinić, B. Tourancheau, F. Rousseau, A. Duda, L. Damon, and R. Guizzetti, "OSCAR: Object security architecture for the Internet of Things," *Ad Hoc Networks*, Internet of Things security and privacy: design methods and optimization, vol. 32, pp. 3–16, Sep. 2015.

[104] K. Wang, M. Du, Y. Sun, A. Vinel, and Y. Zhang, "Attack Detection and Distributed Forensics in Machine-to-Machine Networks," *IEEE Network*, vol. 30, no. 6, pp. 49–55, Nov. 2016.

[105] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister, "Smart dust: Communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, Jan. 2001.

[106] D. Wei, Y. Lu, M. Jafari, P. Skare, and K. Rohde, "An integrated security system of protecting Smart Grid against cyber attacks," in *Innovative Smart Grid Technologies (ISGT), 2010*, Jan. 2010, pp. 1–7.

[107] W. Xu, W. Trappe, Y. Zhang, and T. Wood, "The feasibility of launching and detecting jamming attacks in wireless networks," in *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '05, New York, NY, USA: ACM, 2005, pp. 46–57.

[108] S. Yi and R. Kravets, "MOCA: Mobile certificate authority for wireless ad hoc networks," in *In Proceedings of the 2nd Annual PKI Research Workshop (PKI 03*, 2003.

[109] S. T. Zargar, J. Joshi, and D. Tipper, "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks," *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[110] C. Zhang and R. Green, "Communication Security in Internet of Thing: Preventive Measure and Avoid DDoS Attack over IoT Network," in *Proceedings of the 18th Symposium on Communications & Networking*, ser. CNS '15, San Diego, CA, USA: Society for Computer Simulation International, 2015, pp. 8–15.

[111] X. Zhang, J. He, and Q. Wei, "EDDK: Energy-efficient distributed deterministic key management for wireless sensor networks," *EURASIP Journal on Wirel. Commun. Netw.*, vol. 2011, 12:1–12:11, Jan. 2011.

[112] L. Zhu and B. Tung, *Public key cryptography for initial authentication in Kerberos (PKINIT)*, RFC 4556, IETF, Jun. 2006.

[113] S. Zhu, S. Setia, and S. Jajodia, "LEAP+: Efficient Security Mechanisms for Large-scale Distributed Sensor Networks," *ACM Trans. Sen. Netw.*, vol. 2, no. 4, pp. 500–528, Nov. 2006.