Resilient Authentication and Authorization for the Internet of Things (IoT) Using Edge Computing

HOKEUN KIM, University of California, Berkeley, USA
EUNSUK KANG, Carnegie Mellon University, USA
DAVID BROMAN, KTH Royal Institute of Technology, Sweden
EDWARD A. LEE, University of California, Berkeley, USA

An emerging type of network architecture called *edge computing* has the potential to improve the availability and resilience of IoT services under anomalous situations such as network failures or denial-of-service (DoS) attacks. However, relatively little has been explored on the problem of ensuring availability even when edge computers that provide key security services (e.g., authentication and authorization) become unavailable themselves. This article proposes a resilient authentication and authorization framework to enhance the availability of IoT services under DoS attacks or failures. The proposed approach leverages a technique called *secure migration*, which allows an IoT device to migrate to another trusted edge computer when its own local authorization service becomes unavailable. Specifically, we describe the design of a secure migration framework and its supporting mechanisms, including (1) automated migration policy construction and (2) protocols for preparing and executing the secure migration. We formalize secure migration policy construction as an integer linear programming (ILP) problem and show its effectiveness using a case study on smart buildings, where the proposed solution achieves significantly higher availability under simulated attacks on authorization services.

CCS Concepts: • **Networks** → **Network security**; **Denial-of-service attacks**; • **Computer systems organization** → **Availability**; • **Security and privacy** → *Authentication*; *Authorization*;

Additional Key Words and Phrases: Internet of things, edge computing, secure migration, resiliency

4

This article is an extended version of our previous work [Kim et al. 2017a], which appeared in at the ACM SafeThings 2017 Workshop in November 2017. Main contributions of this extended paper include the newly added section of formal problem formulation as an Integer Linear Programming (ILP) problem with a motivating example (Section 2), advanced secure migration protocols with detailed explanation (Section 3), experiments of advanced protocols at a greater scale including more migration considerations and a variety of configurations (Section 4), and a more comprehensive literature review (Section 5).

The work in this article was supported in part by the National Science Foundation (NSF), award #1446619 (Mathematical Theory of CPS), and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Avast, Denso, Ford, and Siemens. This work is also financially supported by the Swedish Foundation for Strategic Research (project FFL15-0032).

Authors' addresses: H. Kim and E. A. Lee, University of California, Berkeley, 545Q Cory Hall, Berkeley, CA 94720, USA; emails: hokeunkim@berkeley.edu, eal@eecs.berkeley.edu; E. Kang, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA; email: eskang@cmu.edu; D. Broman, KTH Royal Institute of Technology, Kistagången 16 Kista, Sweden; email: dbro@kth.se.

2577-6207/2020/02-ART4 $15.00
https://doi.org/10.1145/3375837

ACM Reference format:
Hokeun Kim, Eunsuk Kang, David Broman, and Edward A. Lee. 2020. Resilient Authentication and Authorization for the Internet of Things (IoT) Using Edge Computing. *ACM Trans. Internet Things* 1, 1, Article 4 (February 2020), 27 pages.
https://doi.org/10.1145/3375837

1 INTRODUCTION

Authentication and authorization play critical roles in ensuring safe and secure operations of the Internet-of-Things (IoT) devices in an open or hostile environment. Existing solutions for providing these security services to IoT devices rely on remote and centralized cloud servers, including OpenMTC,[1] FIWARE,[2] and AWS IoT Core.[3] In this type of approach, however, network problems or a failure of one or more cloud servers may adversely affect devices relying on these critical services. An example illustrating this risk can be observed from the recent Google OnHub incident [Morris 2017], where a failure in the Google authentication servers caused IoT gateway devices (called *OnHub*) to become inaccessible, in turn leading to failures in all devices connected to these gateway devices.

An alternative system architecture, in part designed to reduce this dependency on remote, centralized servers, is called *edge computing* [Shi and Dustdar 2016] or *fog computing* [Luan et al. 2015]. In this approach, computing devices at the edge of the network called *edge computers* serve as an Internet gateway to local, neighboring devices; these edge computers may be deployed as a wide range of devices, including smart home routers, laptops, or smartphones. Figure 1 illustrates the different characteristics of the cloud servers, edge computers, and IoT devices. As pointed out by Lopez et al. [2015], benefits of adopting edge computing include better control of 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.

Previously, we proposed an authentication and authorization infrastructure of the IoT called the *Secure Swarm Toolkit (SST)* [Kim et al. 2017b]. The key distinguishing feature of SST is a *locally centralized, globally distributed* architecture [Kim and Lee 2017]: It consists of a set of local authorization entities called *Auths* [Kim et al. 2016], each of which is deployed on edge computers. An Auth in SST enforces access policies and provides security services to local IoT devices (called *entities*). In this prior work, we demonstrated how SST could provide strong confidentiality and integrity guarantees for communication among IoT devices at scale. We also proved security and scalability of SST using mathematical analysis and a set of experiments.

In this article, building on our prior work, we propose a novel approach for providing resilience against *denial-of-service (DoS) attacks* on an IoT network—e.g., a class of attacks designed to render critical services such as authentication and authorization unavailable to IoT devices. The ultimate goal of a DoS attack is to compromise the availability of a system by disrupting or overloading one or more of its service endpoints. A common and widely known type of DoS attack is called the *distributed* DoS (DDoS) attack [Zargar et al. 2013], where an attacker controls a large number of compromised computers to repeatedly generate service requests and exhaust computational and communication resources of a server. This type of attack is becoming more prevalent as many IoT devices are poorly secured (e.g., weak passwords) and vulnerable to both physical and wireless attacks; thus, they can be relatively easily compromised to be used for launching DoS attacks.

[1] http://www.open-mtc.org/.
[2] https://www.fiware.org/.
[3] https://aws.amazon.com/iot-core/.

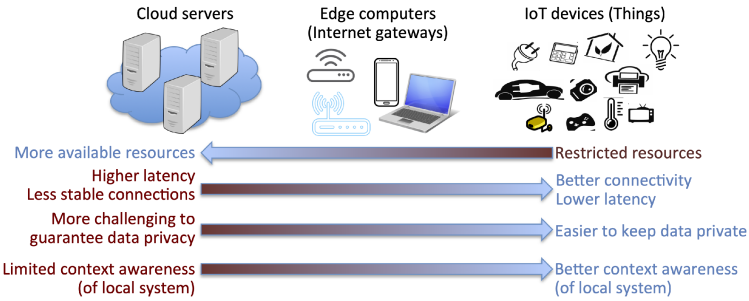


Fig. 1. Illustration of various trade-offs between cloud servers, edge computers, and IoT devices. For instance, cloud servers can provide more available resources but have higher latencies and challenges regarding data privacy, compared to edge computers.

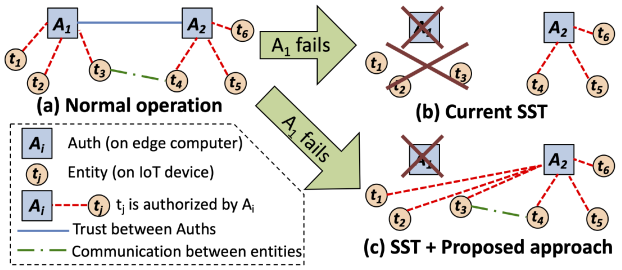


Fig. 2. (a) SST in normal operation. (b) SST without the architectural mechanism proposed in this article, in case of Auth failure. (c) Proposed secure migration technique for enhancing the availability of SST, in case of Auth failure.

In particular, our approach leverages an architectural mechanism called *secure migration*, which allows IoT devices to dynamically migrate from one Auth to another and continue to receive critical services even when some of the Auth devices become unavailable due to an attack or other failures. To illustrate, Figure 2(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. If A_1 fails due to a DoS attack, then authorization services for registered entities will become unavailable as shown in Figure 2(b). This will also affect the availability of other entities. For example, communication between t_4 and t_3 will be disrupted. However, SST with the proposed secure migration approach will gracefully migrate t_1, t_2, and t_3 from A_1 to A_2 and resume authorization services for the IoT entities, as shown in Figure 2(c).

The main research challenge is to perform such secure migration in an optimal manner, taking into account constraints such as communication costs and security requirements for specific IoT services and devices. The contributions of this article are as follows:

- A novel architectural mechanism for constructing IoT services that are resilient against DoS attacks, including a *secure migration* technique that allows IoT devices to continue to operate even when local authorization services become unavailable.
- A formulation of the secure migration problem, including formal constraints and security requirements that define what constitutes a valid migration policy (Section 2).
- A novel approach for finding an optimal migration policy by leveraging an Integer Linear Programming (ILP) solver (Section 3).

- A case study in smart buildings, demonstrating the effectiveness of our approach against DoS attacks (Section 4).

This article is organized as follows: Section 2 provides a formalization of migration as the problem of finding valid assignments of IoT entities and Auths. A technique for automatically solving the migration problem is presented in Section 3, followed by experiments and results showing the effectiveness of the proposed approach in Section 4. The article concludes with a discussion of the related work (Section 5) and future work (Section 6).

2 PROBLEM FORMULATION

We propose an approach called *secure migration* for improving the availability of IoT devices by exploiting the architectural characteristics of SST. In particular, we accomplish this goal by having Auths take over other Auths' authorization tasks when the latter group becomes unavailable due to an attack. At high level, each Auth backs up information about IoT entities that it is serving (e.g., security credentials) and shares it with other trusted Auths. When a failure occurs in an Auth, its registered entities are migrated to other trusted Auths so they can continue to rely on authorization services that are necessary for their operations.

2.1 Threat Model

We assume the existence of an active network attacker with an ability to discover and send packets to Auths running on edge devices. In particular, the attacker seeks to disrupt the availability of IoT devices connected to a gateway (e.g., an Auth) by carrying out various DoS attacks on it, such as flooding or distributed denial-of-service (DDoS) attacks. In our model, we primarily focus on DoS attacks and failures, instead of confidentiality or integrity attacks (addressed by our prior work on SST [Kim et al. 2017b]).

In addition, we exclude insider attacks; i.e., malicious acts by an individual who already has control over IoT and gateway devices on the local area network. Also, we assume that every local Auth is protected from physical tampering and that *authorized*[4] local network users are trustworthy. Furthermore, we do not consider an attack that involves impersonation of Auths; this attack 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). These assumptions allow us to rely on local Auths to perform critical steps in mitigating against a DoS attack.

2.2 Design Considerations

Although the high-level idea behind migration may appear straightforward, determining a *migration policy*—an assignment of entities to Auths to which they will be migrated to—is a non-trivial problem due to the following constraints that must be taken into account:

Auth trust relationships: SST introduces an explicit notion of *trust* between a pair of Auths and enables an entity from Auth A to communicate to an entity in another Auth A' if and only if A and A' trust each other. This communication constraint imposed by the trust must be maintained during migration while satisfying individual entities' requirements. Consider Figure 3(a), where entities t_4 and t_5 are required to communicate with each other, and there is a trust relationship between Auths A_1 and A_2, as well as between A_1 and A_3. Suppose that A_1 becomes unavailable. If we arbitrarily decide to migrate t_4 to A_2, then t_4 will no longer be allowed to communicate with t_5, since there exists no trust relationship between A_2 and A_3. A proper migration policy may

[4]We do not exclude the possibility that non-authorized users can still be malicious.

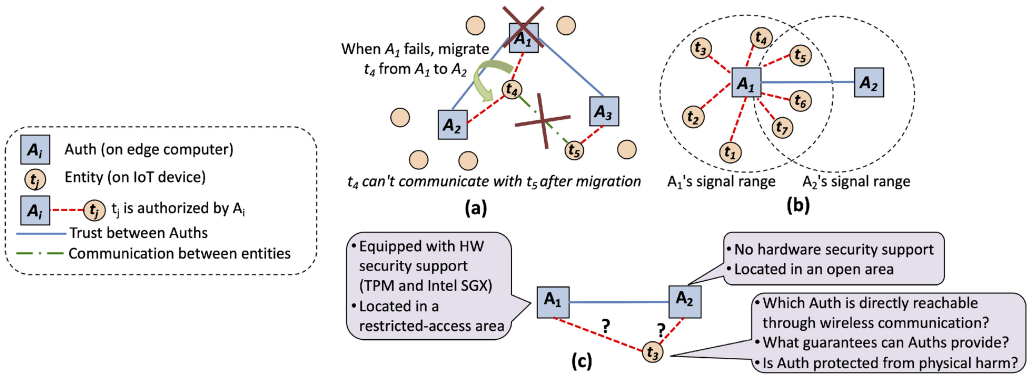


Fig. 3. Considerations for migration policies: (a) Trust among Auths and communication requirements between IoT entities (Things). (b) Balancing the workload of Auths for Authorization. (c) Characteristics and security guarantees of Auths.

instead involve migrating t_4 to A_3. Note that the trust relationship among Auths is not necessarily transitive.

Managing Auth's workload for authorization: Let us consider a different scenario where entities that have lost connections with their original Auths are placed within the signal ranges of Auths A_1 and A_2 as depicted in Figure 3(b). In this scenario, t_5, t_6, and t_7 are within signal ranges of both Auths; thus, they can migrate to either of the two Auths. An arbitrary migration decision may lead to a situation described in Figure 3(b) where all seven entities will have migrated to only A_1. However, this type of decision may adversely affect the performance of the overall network, depending on the operational conditions of Auths. Each Auth may have an inherent limit in its resource capacity to perform authorization tasks for entities; for example, A_1 may be required to run other computationally intensive tasks, or it may be running on a battery-powered edge computer with a limited energy budget. Therefore, assigning entities within A_2's signal range (t_5, t_6, and t_7) to A_2 may lead to greater performance for the overall network.

Security guarantees of Auth: Another important factor to be considered in a migration policy is the level of security provided by Auth from the viewpoint of each entity. These include whether the Auth is equipped with hardware security support such as a TPM (Trusted Platform Module) or Intel's SGX (Software Guard Extensions) and whether the Auth is located in a restricted-access area for extra physical security. For instance, consider Auths A_1 and A_2 in Figure 3(c). Suppose A_1 is equipped with hardware security support and deployed in a restricted-access area, whereas A_2 is located in an open area without any hardware security support. If a particular entity (for instance, t_3) requires a high level of security guarantee by an Auth, then t_3 may wish to be registered with A_1 rather than with A_2 even if the cost of communication with A_1 is higher.

2.3 Policy Construction as an Optimization Problem

When an Auth becomes unavailable due to an attack, we say that Things belonging to that Auth are *dangling*. A migration policy is an assignment of each dangling Thing to one of the remaining Auths in the network. As discussed in Section 2.2, a *valid* policy must ensure that the new network resulting from migration satisfies a number of hard constraints (e.g., trust relationships).

Another aspect of policy construction is that of *optimization*; that is, generating a migration policy that results in a network with a minimal amount of overall communication costs between

Auths and Things. For instance, communication costs between a pair of nodes may depend on a number of factors, such as the physical distance between them and the energy required to perform cryptographic operations.

Structure: We formulate the problem of finding an *optimal* policy as an instance of the Integer Linear Programming (ILP) problem. Formally, let *network graph* G be a tuple $G = (V, E)$, where $V = \mathcal{A} \cup T$ is the set of nodes representing Auths and Things (entities), respectively, and E is the set of edges between these nodes. We assume that G is an *undirected* graph (i.e., all edges are bidirectional), and that $\mathcal{A} \cap T = \emptyset$, i.e., a Thing cannot be an Auth and vice versa.

Each auth $A \in \mathcal{A}$ itself is associated with a set of attributes, (S_A, cap_A), where S_A is the set of security guarantees that the Auth can provide, and cap_A is the threshold of Auth's capacity for authorization of Things without causing degradation.

Similarly, each Thing $t \in T$ is associated with the tuple (S_t, r_t), where S_t is the set of security guarantees that a Thing requires, and r_t is a positive real value representing the resource requirement for authorization tasks. In particular, the sum of r_t's for Things assigned to a particular Auth should not exceed the threshold capacity of that Auth so as not to cause service degradation.

A network graph contains three distinct sets of edges, as follows (i.e., $E = E_{AA} \cup E_{TT} \cup E_{AT}$):

- E_{AA}: A set of edges representing relationships between a pair of Auths, A_1 and A_2. Each $e_{AA} \in E_{AA}$ is associated with (1) the *trust relationship* between a pair of Auths, $trust : A \times A \rightarrow \{0, 1\}$, which is set to 1 if and only if A_1 and A_2 trust each other and (2) the communication cost $cost_{comm} \in \mathbb{R}$ between the two Auths.
- E_{TT}: A set of edges representing relationships between a pair of Things, t_1 and t_2. Each $e_{TT} \in E_{TT}$ is associated with the *criticality* of communication, $critical : T \times T \rightarrow \{0, 1\}$, which is set to 1 if and only if the two Things require a secure communication channel for the availability of the IoT system.
- E_{AT}: A set of edges representing assignments of Things to Auths. In case of an Auth failure, some of the Things may be *dangling*, in that there exists no edge of type E_{AT} from these Things to an Auth. The goal of our migration technique is to assign an Auth to every one of these dangling Things. In addition, each e_{AT} is associated with some communication cost $cost_{comm} \in \mathbb{R}$.

Constraints: The set of constraints that must be satisfied by every graph G is defined as:

$$constraints(G) \equiv securityCons(G) \wedge commReqCons(G) \wedge capacityCons(G),$$

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

- **Thing-specific security requirements:** Each Thing may 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). For each Thing and its security requirement, the Thing must be connected to at least one Auth that provides the same security guarantee:

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

- **Thing-to-thing communication requirements:** After being migrated, some Things need to re-perform an authorization process to obtain the necessary credentials to communicate to other Things. 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:

$$commReqCons(G) \equiv \forall t_1, t_2 \in T, A_1, A_2 \in \mathcal{A}.$$
$$(t_1, t_2) \in E_{TT} \wedge (A_1, t_1) \in E_{AT} \wedge (A_2, t_2) \in E_{AT} \wedge \neg(A_1 = A_2) \wedge critical(t_1, t_2) = 1$$
$$\Rightarrow trust(A_1, A_2) = 1.$$

- **Auth capacity:** Each Auth has a threshold in terms of upper bounds for authorization requests served per minute or upper bounds for session keys cached in Auth. When there are more registered entities than this threshold can handle, the Auth will suffer from potential degradation in the services that it provides. For each Auth, the sum of authorization task requirements for all Things connected to this Auth must not exceed its capacity:

$$capacityCons(G) \equiv \forall A \in \mathcal{A}. \left(\sum_{t \in ts(A)} r_t \right) \leq cap_A,$$

where cap_A is the capacity of auth A, r_t is the resource requirement of Thing t, and $ts(A) = \{t \in T \mid (A, t) \in E_{AT}\}$ is the set of all Things connected to A.

Costs to be optimized: Another aspect of policy construction is that of optimization; that is, generating a migration policy that results in a network with a minimal amount of overall communication costs between Auths and Things. A cost may represent (some combination of) energy consumption, latency, or distance between nodes. For instance, as the distance between a pair of nodes increases, the frequency of packet loss may also increase, in turn triggering more frequent retransmission and, thus, overall communication cost. To accommodate different types of costs that may be relevant to different domains, we intentionally treat the notion of cost as an abstract entity. In particular, the goal here is to optimize over a cost variable, TC, which represents the total communication costs for the Auths and Things:

$$TC = w_T C_T + w_{\mathcal{A}} C_{\mathcal{A}},$$

where $C_{\mathcal{A}}$ and C_T represent the overall communication costs for Auths and Things, respectively. In addition, $w_{\mathcal{A}}$ and w_T are positive real numbers that represent the weighting factors to distribute the migration costs over Auths and Things, where $w_{\mathcal{A}} + w_T = 1$. The types of costs are further defined as follows:

- **Total communication costs of Auths:** The sum of communication costs imposed on Auths defined as:

$$C_{\mathcal{A}} = \sum_{e \in E_{TT}} (AA(e).cost_{comm}),$$

where $AA(e)$ returns the Auth-to-Auth edge for authorizing 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, because there is no need for Auth-to-Auth communication.
- **Total communication costs of Things:** The sum of communication costs imposed on 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, and similarly, $AT_2(e)$ returns the edge between the second Thing and its Auth.

ILP formulation: An ILP problem has the following canonical form:

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Ax \leq b, x \in \mathbb{Z}^n, \end{aligned}$$

where c and b are vectors, and A is a matrix. In particular, for our problems, x is a vector of binary variables, each of which encodes an assignment of a particular Thing to an Auth. In other words, $x = [x_{00}, x_{01}, \ldots, x_{ij}, \ldots]$ where x_{ij} is assigned 1 if and only if there exists an edge of type E_{AT} from some Auth $A_i \in \mathcal{A}$ to Thing $t_j \in T$. In addition, c is a vector of integers (of the same length as x) where each entry c_{ij} corresponds to the cost incurred by assigning Thing t_j to Auth A_i.

Two of the logical constraints in our model (i.e., *securityCons* and *commReqCons*) can be reduced to a propositional formula as follows:

- *securityCons*: Since each network has a fixed number of Things and Auths, quantifiers can be eliminated by unrolling them over the finite domains. For instance, suppose that a sample network consists of two Things (t_0, t_1) and two Auths (A_0, A_1) such that $S_{t_0} = \{r_A\}$ and $S_{t_1} = \{r_B\}$. Then, the security constraint can be expressed as the following quantifier-free formula:

$$\begin{aligned} securityCons(G) \equiv &((A_0, t_0) \in E_{AT} \wedge r_A \in S_{A_0} \vee (A_1, t_0) \in E_{AT} \wedge r_A \in S_{A_1}) \\ &\wedge\ ((A_0, t_1) \in E_{AT} \wedge r_B \in S_{A_0} \vee (A_1, t_1) \in E_{AT} \wedge r_B \in S_{A_1}). \end{aligned}$$

  In addition, for each security requirement $r \in S_t$, we introduce a Boolean variable that is true if and only if $r \in S_A$.
- *commReqCons*: Again, quantifiers can be eliminated through unrolling. Boolean variables are introduced to represent whether (1) a pair of Things requires a critical communication channel ($critical(t_1, t_2)$) and (2) their corresponding Auths consider each other trustworthy.

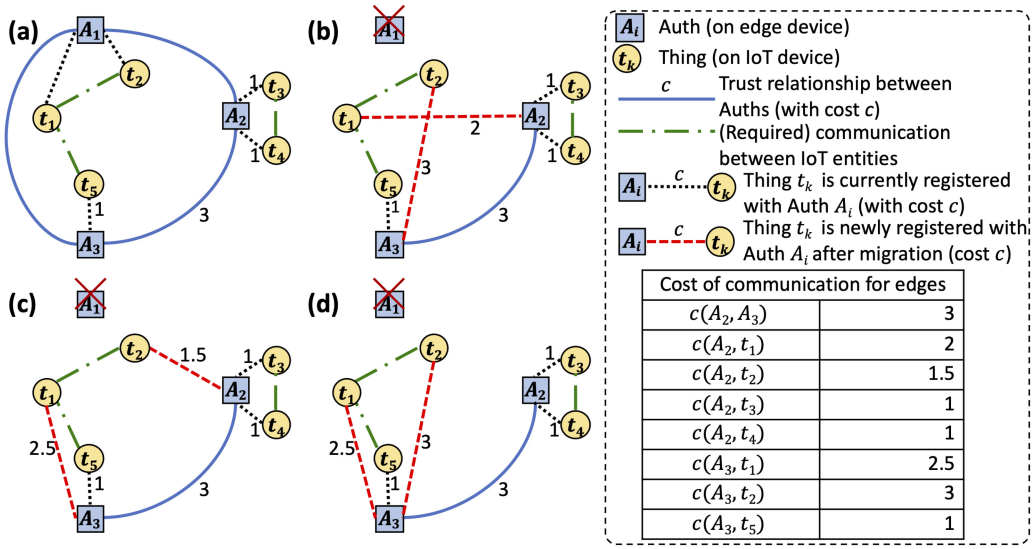
It is a well-known result that any propositional logic formula can be encoded as a set of linear inequality constraints [Hooker and Lama 1999]. In addition, the constraint *capacityCons* is already in linear inequality form and thus can be directly encoded as a constraint in the ILP problem. Given these constraints, an ILP solver attempts to find a satisfying assignment of truth values to the vector x that minimizes the associated costs. The output from the solver (if any) represents a migration policy that assigns dangling Things to the remaining Auths in the network while minimizing the overall communication cost of the resulting network.

2.4 Example

Let us consider the example shown in Figure 4, where there are three Auths, A_1, A_2, A_3, and five Things, t_1 through t_5. For brevity, let us denote the cost of communication between A_i and t_j as $c(A_i, t_j)$. Similarly, we denote the communication cost between A_i and A_k as $c(A_i, A_k)$. Here, the communication costs include energy consumption due to wireless communication, cryptographic operations, and local data storage access.

The relationships and costs of communication between Auths and Things are described in Figure 4. In this system, all Auths are assumed to 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:

$$\begin{aligned} \{(A_1, A_2), (A_1, A_3), (A_2, A_3)\} &\subset E_{AA}, \\ \{(A_1, t_1), (A_1, t_2), (A_2, t_3), (A_2, t_4), (A_3, t_5)\} &\subset E_{AT}, \\ \{(t_1, t_2), (t_1, t_5), (t_3, t_4)\} &\subset E_{TT}. \end{aligned}$$



Cost of communication for edges	
$c(A_2, A_3)$	3
$c(A_2, t_1)$	2
$c(A_2, t_2)$	1.5
$c(A_2, t_3)$	1
$c(A_2, t_4)$	1
$c(A_3, t_1)$	2.5
$c(A_3, t_2)$	3
$c(A_3, t_5)$	1

Fig. 4. 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.

In addition, we assume that all things have a uniform resource requirement (i.e., $r_t = 1$ for every $t \in T$), and each Auth is capable of servicing at most three things at any given moment ($cap_A = 3$ for every auth A).

Let us consider a scenario in which A_1 becomes unavailable due to an attack, and migration is needed. To represent this attack, all edges involving A_1 are deleted, resulting in the following set of edges:

$$\{(A_2, A_3)\} \subset E_{AA},$$
$$\{(A_2, t_3), (A_2, t_4), (A_3, t_5)\} \subset E_{AT},$$
$$\{(t_1, t_2), (t_1, t_5), (t_3, t_4)\} \subset E_{TT}.$$

To satisfy the communication requirements between t_1 and t_2, these Things must be assigned to the remaining Auths, while satisfying the capacity constraint. For example, for A_2, this constraint is equal to:

$$|\{(x, y) \mid (x, y) \subset E_{AT} \wedge x = A_2\}| \leq cap_{A_2} = 3,$$

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. This results in the following three possible migration plans:

- $Plan_1$: $\{(A_2, t_1), (A_3, t_2)\} \subset E_{AT}$,
- $Plan_2$: $\{(A_2, t_2), (A_3, t_1)\} \subset E_{AT}$,
- $Plan_3$: $\{(A_3, t_1), (A_3, t_2)\} \subset E_{AT}$.

Each plan is evaluated in terms of the overall communication costs (C_T and $C_{\mathcal{A}}$) that result after the migration takes place. Note that the communication costs incurred on t_3, t_4, and t_5 remain the same after migration; that is, $C_{t_3} = 1$, $C_{t_4} = 1$, and $C_{t_5} = 1$. Thus, we need to consider only the

Table 1. The Total Cost to Be Optimized Depending on Different Weights for the Things and Auths, w_T and w_A, respectively

(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**

Note that the minimum costs are marked as bold for each set of weights.

changes in the costs associated with $t_1$and t_2; i.e., C_{t_1} and C_{t_2}. The total communication cost of Things is $C_T = C_{t_1} + C_{t_2} + C_{t_3} + C_{t_4} + C_{t_5} = C_{t_1} + C_{t_2} + 3$.

$$\begin{aligned}
&Plan_1 : C_{t_1} = 2 \times c(A_2, t_1) = 4 \quad C_{t_2} = c(A_3, t_2) = 3 && C_T = C_{t_1} + C_{t_2} + 3 = 10,\\
&Plan_2 : C_{t_1} = 2 \times c(A_3, t_1) = 5 \quad C_{t_2} = c(A_2, t_2) = 1.5 && C_T = C_{t_1} + C_{t_2} + 3 = 9.5,\\
&Plan_3 : C_{t_1} = 2 \times c(A_3, t_1) = 5 \quad C_{t_2} = c(A_3, t_2) = 3 && C_T = C_{t_1} + C_{t_2} + 3 = 1.
\end{aligned}$$

Let us now consider Auth-related costs that result from the migration. The communication cost for A_2 to handle authorization of t_3 and t_4 remains the same after migration; 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 unaffected by migration, which is $c(A_3, t_5) = 1$.

$$\begin{aligned}
&Plan_1 : C_{A_2} = C'_{A_2} + 2 \times c(A_2, t_1) + 2 \times c(A_2, A_3) = 12\\
&\qquad C_{A_3} = c(A_3, t_2) + c(A_3, t_5) + 2 \times c(A_2, A_3) = 10 && C_{\mathcal{A}} = 22,\\
&Plan_2 : C_{A_2} = C'_{A_2} + c(A_2, t_2) + c(A_2, A_3) = 6.5\\
&\qquad C_{A_3} = 2 \times c(A_3, t_1) + c(A_3, t_5) + c(A_2, A_3) = 9 && C_{\mathcal{A}} = 15.5,\\
&Plan_3 : C_{A_2} = C'_{A_2} = 2 \quad C_{A_3} = 2 \times c(A_3, t_1) + c(A_3, t_2) + c(A_3, t_5) = 9 && C_{\mathcal{A}} = 11.
\end{aligned}$$

Let TC be the total communication cost to be optimized. TC can be expresses as a weighted sum of C_T and $C_{\mathcal{A}}$:

$$TC = w_T C_T + w_A C_{\mathcal{A}},$$

where w_T and w_A are the weights for C_T and $C_{\mathcal{A}}$, respectively, such that $w_T + w_A = 1$. The values for w_T and w_A should be determined with considerations of the IoT network's characteristics. For instance, if the IoT network consists of battery-powered Things, then we need to use greater w_T to optimize the cost for Things. However, when the edge computers hosting Auths have resource constraints such as limited communication bandwidth, w_A should be greater to take this factor into account. Table 1 shows TC for different sets of weights, demonstrating that the choice of an optimal policy depends on the allocation of weights as well. For example, when $(w_T, w_A) = (0.8, 0.2)$, $Plan_2$ is considered the optimal plan, whereas if $(w_T, w_A) = (0.7, 0.3)$, then $Plan_3$ is the one that results in a minimal cost.

2.5 Remark about Performance

For a large, densely connected network with a high number of nodes, it may take a considerable amount of time and resources to compute the total communication costs (for all possible migration plans) and solve the optimization problem. In practice, we believe that this will not be a significant obstacle to our approach, since the generation of potential migration plans and an optimal policy can be performed offline, thus incurring little run-time overhead when actual migration takes place. We further discuss the performance of our approach through an experiment on a realistic

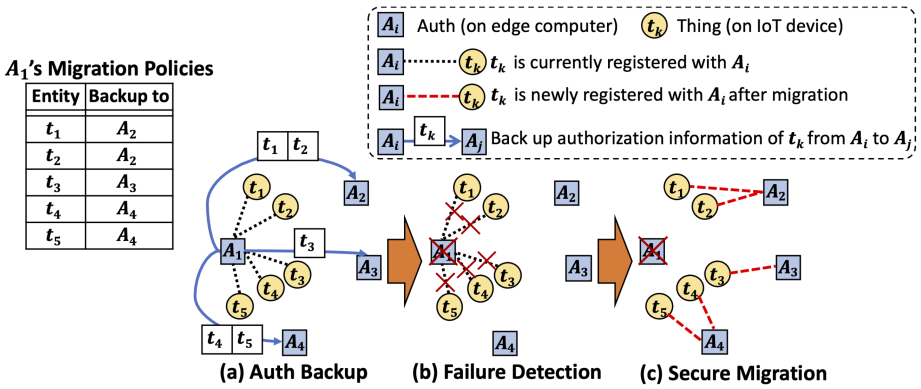


Fig. 5. Overview of operations to defend against DoS attacks: (a) Backup, (b) Detection, and (c) Migration.

IoT system in Section 4. Based on our performance measurements of solving the migration policy construction problem, the execution time for a realistic setting with 7 Auths and 70 Things takes less than a second on a typical laptop computer. Given that the execution time is almost negligible, there will not be an issue even on a rare occasion where we need to recalculate the optimization problem, for example, when there are changes in the environment or configuration such as the number of Auths or Auths' capacity. Also, it should not matter where we run the migration policy construction; for instance, it can run on any of the Auths in the network.

3 APPROACH: SECURE MIGRATION

In this section, we describe a robust architecture and implementation for achieving secure migration. Figure 5 illustrates the overall operations to enhance availability when some Auths are down; (a) *Auth backup*, (b) *Failure detection*, and (c) *Secure migration*. As shown in Figure 5(a), during normal operation, each Auth backs up credentials and information of its registered IoT entities to other trusted Auths as planned using the migration methods proposed in Section 2. The registered entities detect when their Auth is unavailable, as depicted in Figure 5(b). Example ways of detecting Auth failures are explained in Section 3.3.1. Upon recognizing its Auth's failure, each entity tries to reach alternative Auths by sending a *migration request* (explained in Section 3.2.3). For this, each entity needs to be assigned a list of hosts and ports for alternative Auths that it can potentially migrate to. The alternative Auth will accept the migration request if an entity tries to migrate to the right alternative Auth as planned in the migration policy. As a result of migration, each registered entity will be assigned to new Auths as shown in Figure 5(c).

The following subsections discuss a brief description of SST's authentication and authorization (Section 3.1), the preparation process and Auth backup protocol (Section 3.2), and details of the secure migration protocol to find right Auths to which the IoT entities migrate (Section 3.3). All of our work, including implementation and experimental setups, is also available on an open-source GitHub repository.[5]

3.1 Background: SST's Authentication and Authorization Process

3.1.1 Session Key. In SST, each IoT entity is authenticated and authorized by an Auth. Upon authorization, the entity receives a *session key* from the Auth. A session key is a temporary

[5] https://github.com/iotauth.

symmetric cryptographic key[6] for accessing a certain service or communicating with another entity (or entities). A session key can be just a key for a Message Authentication Code (MAC)[7] when the access activity does not require confidentiality of the data. For confidential data, we can use a session key as a combination of a MAC key and a cipher key.[8] A session key could be a single key if the session uses authenticated encryption[9] that does both the encryption and message authentication with a single key value.

3.1.2 Distribution Key. 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 cryptographic key between Auth and each entity called a *distribution key*. A distribution key in SST consists of a pair of keys, a cipher key for encryption, and a MAC key for message authentication.

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[10] for perfect forward secrecy[11] 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 of using a permanent distribution key so it can support resource-constrained IoT devices that cannot afford public-key cryptography.

3.1.3 Auth Database. In SST, each Auth has database tables to store authorization information of entities and trust relationships among Auths. As introduced in the SST paper [Kim et al. 2017b], the four main database tables maintained by each Auth are *Registered Entity Table*, *Communication Policy Table*, *Trusted Auth Table*, and *Cached Session Key Table*.

- **Registered Entity Table** stores information and credentials of the IoT entities (Things) registered with the Auth.
- **Communication Policy Table** specifies cryptographies and duration of communication sessions between entities.
- **Trusted Auth Table** is used to look up information and credentials of other trusted Auths.
- **Cached Session Key Table** stores session keys issued by the Auth and cached by IoT entities.

The trust relationships among Auths and credentials of the trusted Auths are stored in *Trusted Auth Table*. This information is used to securely back up Auth's IoT entities' information and credentials (stored in *Registered Entity Table*) to other trusted Auths. If the communication policy of IoT 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.

3.2 Preparing for Failures: Auth Backup

For the preparation of DoS attacks or failures on Auths, migration policies are constructed using the approach presented in Section 2. The policy construction mechanism is implemented as

[6]Symmetric cryptography uses the same key for both a sender and a receiver.
[7]A tag (short data) used to authenticate the sender of the message and guarantee data integrity that the original message has not been altered. A MAC is generated from the message and a cryptographic key called a *MAC Key*.
[8]A cryptographic key used for encryption and decryption of data.
[9]A type of encryption that provides data integrity and message authenticity as well as confidentiality.
[10]A key exchange mechanism to generate an ephemeral key for protecting a temporary communication session.
[11]A cryptographic feature to ensure that even compromised future keys do not lead to compromises of previous keys and confidentiality of past sessions.

part of Auth of SST using the Gurobi ILP solver.[12] We currently do not limit where and when the migration policy should be constructed. As stated in Section 2.5, any Auth has the capability of running the policy construction and propagating the policies. The minimum requirement for migration policy construction is that it needs to be computed before an Auth failure occurs to enable secure migration. Also, in this article, we do not limit how Auths are informed about system configurations and requirements for computing migration policies, including Auth trust relationships and communication requirements between Things. For simplicity, we assume that all necessary information is given to Auths when they are initialized. After migration policy construction, Auths back up authentication and authorization information for their entities so the entities can migrate to other trusted Auths when their Auths become unavailable.

How can we make sure that a *migrating entity* (an entity that migrates to another Auth) and a *new Auth* (an Auth to which the migrating entity is supposed to migrate) continue authentication and authorization after migration? Also, how can we make the migrating entity and the new Auth trust each other? To make these happen, the *original Auth* (the Auth that the migrating entity was originally registered with before the failure) will need to set up credentials for the new Auth and entities before failures. In our proposed scheme, the backup tasks are performed by Auths, not individual IoT entities, for the following reasons:

1. Migration policies are constructed, updated, and managed by Auths.
2. The burden of maintaining backup information can be offloaded from IoT entities, especially from resource-constrained IoT devices.

In this section, we describe the considerations for preparing credentials for different types of IoT entities (Sections 3.2.1 and 3.2.2), the Auth backup protocol to share backup information among Auths (Section 3.2.3), and how Auths store and manage backup information in database tables (Section 3.2.4).

3.2.1 Entity with Public-key Support. First, let us consider the case where a migrating entity uses public-key cryptography for updating its distribution key. In this case, the migrating entity and the new Auth should be able to verify each other's public key. For the new Auth to be able to trust the migrating entity, the original Auth must provide the public key of its registered entity (*entity public key*) beforehand. To let the migrating entity verify the new Auth's public key, the original Auth prepares a *migration certificate (cert)*, the new Auth's public key signed by the original Auth's private key. Since the migrating entity already has the original Auth's public key, it can verify the migration certificate later. A migration certificate has its validity period and expires after the validity period.

3.2.2 Entity without Public-key Support. In SST, we allow a resource-constrained IoT device without public-key cryptography support to be authenticated and authorized by using a *permanent distribution key*, a distribution key valid for the entire lifetime of the entity. For such devices, we need a different approach. The new Auth and the migrating entity need a new permanent distribution key after migration. We do not allow reuse of an *original distribution key*—a distribution key between the migrating entity and the original Auth—to prevent the new Auth from being able to access past session keys given by the original Auth. We define a *new distribution key* as a newly generated distribution key between the migrating entity and the new Auth.

The original Auth generates a new distribution key and sends it to the new Auth via a secure channel protected by HTTPS over SSL/TLS so the new Auth can communicate with the migrating entity. The original Auth also encrypts and authenticates the new distribution key

[12] http://www.gurobi.com.

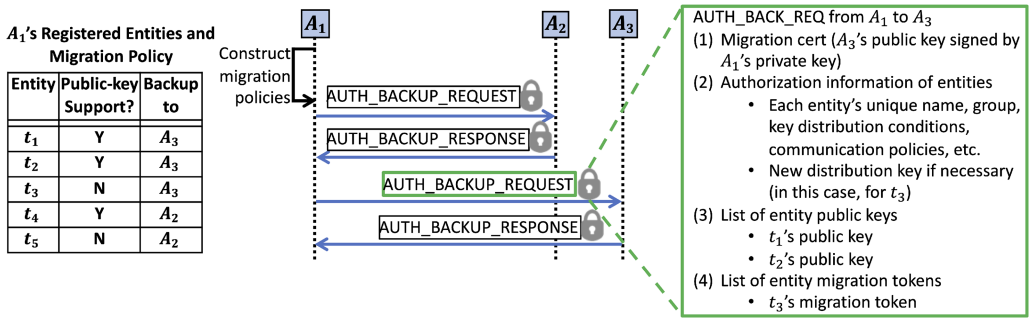


Fig. 6. An example Auth backup procedure. A_1 is the original Auth trying to back up its entities t_1–t_5. Note that t_1–t_3 should be backed up to A_3, and t_4 and t_5 are backed up to A_2 according to the migration policy. Also, note that t_3 and t_5 do not support public-key cryptography; thus, A_1 needs to back up migration tokens for them. Each lock icon means that the message is protected by HTTPS protocol over SSL/TLS.

using the original distribution key. We call this a *protected new distribution key*. The protected new distribution key should be backed up on the new Auth so the new Auth can provide it to the migrating entity. The migrating entity can decrypt and authenticate the protected new distribution key using the original distribution key, and can check whether the new Auth has the new distribution key using a challenge-response.

With this, the migrating entity can trust the new Auth using the protected new distribution key. But how about the other way around? How can the new Auth trust the migrating entity? For this, the original Auth also sends the MAC key of the original distribution key to the new Auth, because a distribution key can be decomposed into two parts: a cipher key for encryption and a MAC key for message authentication, as defined in Section 3.1.2. We explain how the migrating entity can trust the new Auth with the MAC key of the original distribution key in Section 3.3.2. By giving only the MAC key—not the cipher key of the original distribution key to the new Auth—we can make sure that the new Auth cannot see the previous communications between the original Auth and the migrating entity.

The combination of (1) the protected new distribution key and (2) the MAC key of the original distribution key is called a *migration token*. The original Auth sends these migration tokens for IoT entities without public-key cryptography to the new Auths during the Auth backup. Like a migration certificate, a migration token also has its validity period and expires after the validity period.

3.2.3 Auth Backup Protocol. In this section, we explain a network protocol called *Auth Backup Protocol* to back up the credentials prepared for secure migration. For the following communications between trusted Auths, we assume that all messages exchanged are protected by HTTPS over SSL/TLS. An example of the backup operation is described in Figure 6. First, A_1 constructs a migration policy for its registered entities, either by itself or by receiving migration policies from other trusted Auths, using the method shown in Section 2. With the constructed migration policy, (1) A_1 prepares a migration cert, (2) basic and authorization information of each entity (this also includes a new distribution key between the new Auth and an entity without public-key cryptography support—in this case, t_3), (3) entity public keys for entities supporting public-key cryptography, and (4) migration tokens for entities without a public-key cryptography support as explained in Sections 3.2.1 and 3.2.2. These four components are sent as an *AUTH_BACKUP_REQUEST* message. In the example, A_1 sends *AUTH_BACKUP_REQUEST* to A_3 with a migration cert, entity public

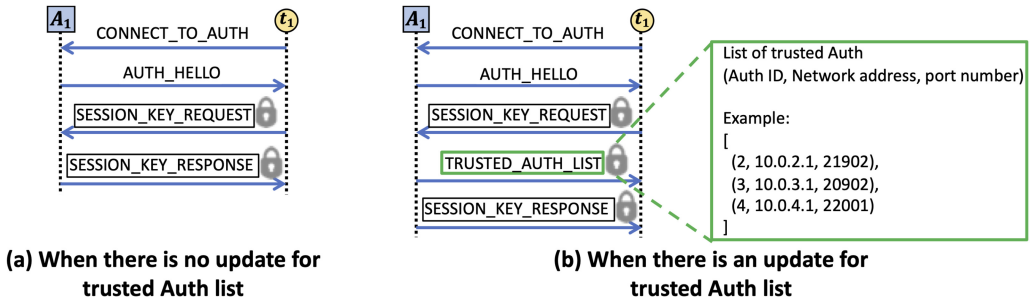


Fig. 7. Updating trusted Auth during Auth–entity communication defined in Kim et al. [2017b]. Each lock icon means that the message is encrypted and authenticated by a distribution key.

keys of t_1 and t_2 and a migration token of t_3. Upon receiving the request, A_3 updates its database tables using the information (explained in Section 3.2.4) in the request and responds to A_1 with *AUTH_BACKUP_RESPONSE* to acknowledge that the request has been processed. If there is a newly registered entity for A_1, then A_1 should send out *AUTH_BACKUP_REQUEST* to back up credentials for the newly registered entity.

We also need to make sure that the migrating entities can connect to new Auths in case of original Auth's failure. Network information of trusted Auths (potential new Auths) can be embedded in each IoT entity when each IoT entity is initialized. Also, the trusted Auth information can be updated using the Auth–entity communication. In SST, Auth–entity communication occurs when an entity requests session keys to be authorized to access other entities or services, as shown in Figure 7(a). *CONNECT_TO_AUTH* can be a TCP connection setup or a UDP message from the entity, t_1, to the original Auth, A_1. *AUTH_HELLO* includes the Auth's unique ID in a string and a nonce[13] to prevent a replay attack.[14] After receiving *AUTH_HELLO*, t_1 sends *SESSION_KEY_REQUEST* encrypted and authenticated with the distribution key between t_1 and A_1. Then, A_1 responds with *SESSION_KEY_RESPONSE* including a list of session keys, also encrypted and authenticated.

For migration, this Auth–entity communication is extended as shown in Figure 7(b). When A_1 needs to update network information of trusted Auths, it also sends *TRUSTED_AUTH_LIST*, which includes a list of trusted Auths and their network addresses and port numbers, before sending *SESSION_KEY_RESPONSE*. The *TRUSTED_AUTH_LIST* message is also protected by the distribution key between t_1 and A_1.

3.2.4 Storing Backup Information. To store backup information for secure migration, the Auth database tables of SST should be extended. *Trusted Auth table* (explained in Section 3.1.3) is extended to store *migration certs* (explained in Section 3.2.1). To store backup-related information for each IoT entity, a *registered entity table* (explained in Section 3.1.3) is also extended. Figure 8 shows an example of the extended version of a *registered entity table* after the Auth backup is finished. SST's original design of a registered entity table stores each entity's unique name, group, key distribution conditions, and distribution key name. In addition, we add four new columns for secure migration. The *Active* field indicates whether the IoT entity is currently authorized by the Auth. Thus, for the backed up IoT entities, this field is set to *False* initially, and it is changed to *True* when the IoT entities migrate to the Auth and become active. *Backup To Auth ID* is used to back

[13]Randomly generated number.
[14]The nonce is randomly generated every time an entity connects to Auth, and an authenticated nonce should be included in the following message from the entity. This makes sure that the message from the entity is fresh and not replayed by an attacker.

Original columns in Auth's Registered Entity Table in SST | New columns added for **secure migration**

Name	Group	Key Distribution Conditions <Protocol, Permanent Distribution Key?, Crypto, Max Session Keys Per Request, Distribution Key Validity>	Distribution Key Information <Value, Expiration Time (YYYY-MM-DD:HH-MM-SS)>	Active?	Backup To AuthID	Backup From AuthID	Migration Token
PG012	Power Generator	<TCP, False, RSA-2048:DH-secp384r1 , 3, 1 Day>	<90 92 67 3f ..., 2018-08-01:17-03-25>	True	104	Null	Null
CSW107	Circuit Switch	<TCP, False, RSA-2048 , 3, 7 Days>	<ed 73 41 1c ..., 2018-08-05:14-48-03>	True	103	Null	Null
VS037	Voltage Sensor	<UDP, True, AES-128-CBC:Hmac-SHA256, 3, 2 Years>	<83 d6 f6 f6 ..., 2020-04-21:08-00-00>	True	106	Null	Null
CSW065	Circuit Switch	<TCP, False, RSA-2048 , 3, 7 Days>	<3e af 04 0f..., 2018-08-04:03-26-15>	False	NULL	102	Null
CS087	Current Sensor	<UDP, True, AES-256-CBC:Hmac-SHA256, 3, 1 Year>	<a1 f0 8f db..., 2019-06-30:09-00-00>	False	NULL	103	b6 58 79 1c ...
●●●	●●●	●●●	●●●	●●●	●●●	●●●	●●●

Fig. 8. Registered entity table with migration information.

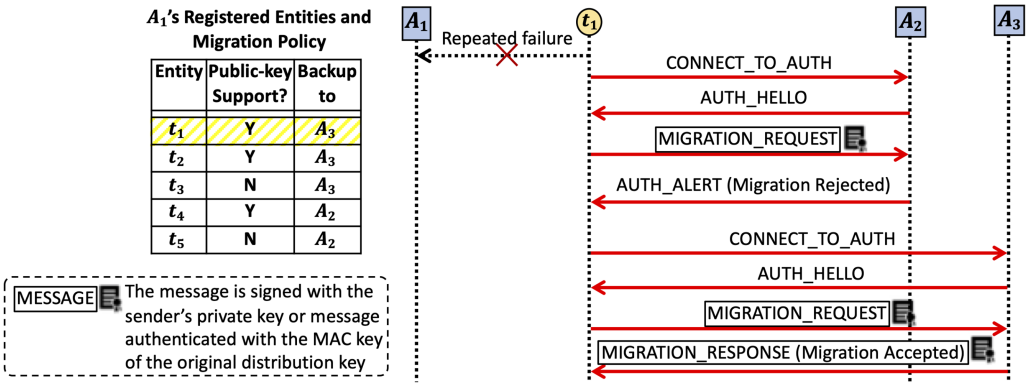


Fig. 9. Secure migration protocol example: A_1 fails and its entity t_1 tries to migrate to A_2 or A_3. Each certificate icon means that the message is signed or MAC authenticated.

up the entity to designated Auths in migration policies. The *Backup From Auth ID* field is used to keep track of from which Auth an IoT entity is backed up and to choose the right migration cert for each entity. For the entity CS087, a backed-up entity without a public-key support, a migration token (explained in Section 3.2.2) is also stored.

3.3 Secure Migration for Recovery from Auth Failure

3.3.1 Detecting Auth Failure. When an attack or a failure occurs, first, it needs to be detected by IoT entities. Each entity should have criteria for determining whether its Auth is down. This can be done by setting up a threshold for the number of connection failures or connection downtime. For example, an entity recognizes the failure of its Auths when it fails to connect to its Auth 10 times in a row or when it cannot connect to its Auth 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) and port numbers. When an entity detects its Auth's failure, it tries to connect to the Auths in its list and tries to migrate to one of them.

3.3.2 Secure Migration Protocol. Figure 9 describes an example with a set of migration operations. In this example, an IoT entity, t_1, detects the failure of its original Auth, A_1. Assume that A_2 and A_3 are in t_1's list of potential new Auths. Then t_1 connects to A_2 using the network information of A_2 and sends *MIGRATION_REQUEST* to A_2. Although A_2 is trusted by A_1, it is not the one that t_1 is supposed to migrate to, according to the migration policy. Since A_2 does not have the required credentials for t_1, A_2 sends an *AUTH_ALERT* message, indicating that the migration request is rejected and t_1 should try another trusted Auth in its list. After receiving *AUTH_ALERT*

from A_2, t_1 tries the next Auth in its list, A_3. When A_3 receives *MIGRATION_REQUEST* from t_1, it notices that t_1 can migrate to it and responds with a *MIGRATION_RESPONSE* message indicating that the request has been accepted.

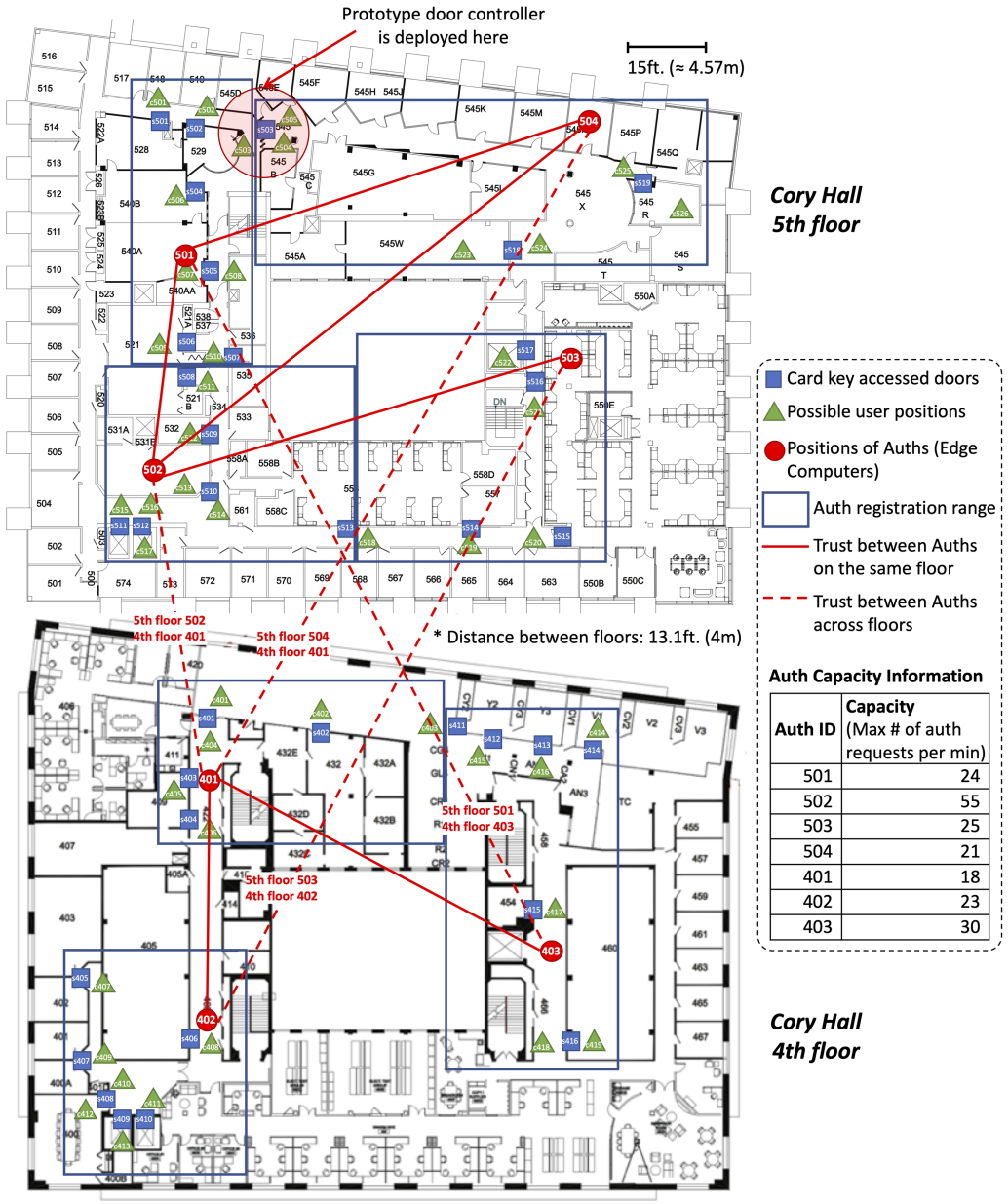
So, how can t_1 and A_3 establish a new trust relationship? This is achieved by a challenge-response where they check if the counterpart has the necessary credential using random numbers. As shown in Figure 7 in Section 3.2.3, Auth-entity communication starts with *AUTH_HELLO* after the IoT entity connects to an Auth. *AUTH_HELLO* includes an *Auth nonce*, a fresh random number generated by the new Auth. The *MIGRATION_REQUEST* message includes the migrating entity's name, Auth nonce, and *entity nonce*, another random number generated by the entity. If the migrating entity supports public-key cryptography, then *MIGRATION_REQUEST* is signed by the migrating entity's private key. If the migrating entity does not support public-key cryptography, then *MIGRATION_REQUEST* is authenticated by the MAC key of the original distribution key. Upon receiving *MIGRATION_REQUEST*, the new Auth first checks its database with the migrating entity's name to pull out the credential to verify the migrating entity's signature or MAC. If the migration entity cannot be found in the new Auth's database, then *MIGRATION_REQUEST* gets rejected. When the new Auth finds the necessary credential—either the migrating entity's public key or the MAC key of the original distribution key—it verifies if *MIGRATION_REQUEST* came from the migrating entity using the Auth nonce and the signature or the MAC. If the verification succeeds, then the new Auth trusts the migrating entity.

Now, the new Auth prepares *MIGRATION_RESPONSE* so the migrating entity can trust the new Auth. A *MIGRATION_RESPONSE* message includes the new Auth's unique ID, entity nonce, and an Auth cert or a protected new distribution key, depending on whether the migrating entity supports public-key cryptography. *MIGRATION_RESPONSE* is signed by the Auth's private key or message authenticated by the MAC key of the original distribution key. Also, the migrating entity verifies the Auth cert using the original Auth's public key or tries to decrypt the protected new distribution key using the original distribution key's MAC key. After verifying *MIGRATION_RESPONSE,* the new Auth's credential, the migrating entity can trust the new Auth.

Based on the newly established trust relationship, the migrating entity continues to be authenticated and authorized by the new Auth. If the migrating entity uses public-key cryptography, then it can get and update new distribution keys from the new Auth using public-key cryptography. If the migrating entity does not support public keys, then the new distribution key decrypted from *MIGRATION_RESPONSE* becomes a new permanent distribution key with the new Auth. The migrating entity can request session keys and use them for accessing other IoT entities and services in SST.

4 EXPERIMENTS AND RESULTS

In this section, we carry out experiments to demonstrate the effectiveness of the proposed migration approach for maintaining availability. For the following experiments, we only consider Things (IoT entities) that are capable of public-key cryptography. As an experimental scenario, we take door controllers and door opening applications in a smart building. This is inspired by a prototype door controller deployed on the fifth floor of Cory Hall at UC Berkeley. We assume a virtual environment where the door controllers are deployed on currently card-key accessed doors, door opening mobile phone apps run on user smartphones, and Auths are deployed on some of the existing Wi-Fi access points. Figure 10 illustrates this virtual environment. We also assume Auths have trust relationships depending on research centers on the fifth floor and the fourth floor, and the Things (door controllers and user smartphones) are registered with Auths, as shown in Figure 10. We measure availability as the ratio of responses from door controllers (the number of correct door operations) to the door opening requests for a given time window. In the



Auth ID	Capacity (Max # of auth requests per min)
501	24
502	55
503	25
504	21
401	18
402	23
403	30

Fig. 10. An experimental virtual environment with Auths, door controllers, and door opening mobile applications on the floor map of the fourth and fifth floors, Cory Hall at UC Berkeley. (Floor plans used with permission.)

following experimental scenarios, different numbers of Auths can be unavailable due to failures or DoS attacks. In addition, we 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.

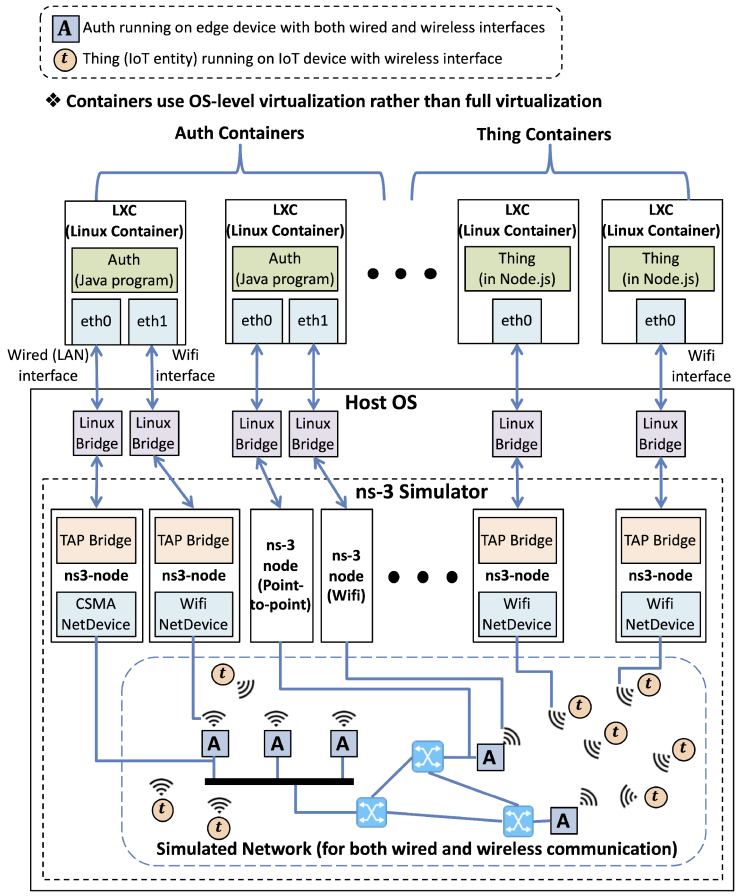


Fig. 11. Experimental setup with the ns-3 simulator network simulator.

4.1 Experimental Setup

Figure 11 describes the experimental setup with the *ns-3 network simulator* [Riley and Henderson 2010]. For realistic experiments, we 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)*[15] that provides OS-level virtualization (paravirtualization) with a separate virtual network space. For simulating the network infrastructure, we 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 Wi-Fi connections, and LXCs for IoT entities have Wi-Fi connections. For connections between Auths' wired network interfaces, we use a CSMA channel with a data rate of 100 Mbps. For connections between the wireless network interfaces of Auths and Things, we use an ad hoc IEEE 802.11a channel with a data rate of 54 Mbps. For the channel signal strength model, we use a Log

[15] https://linuxcontainers.org/lxc/.

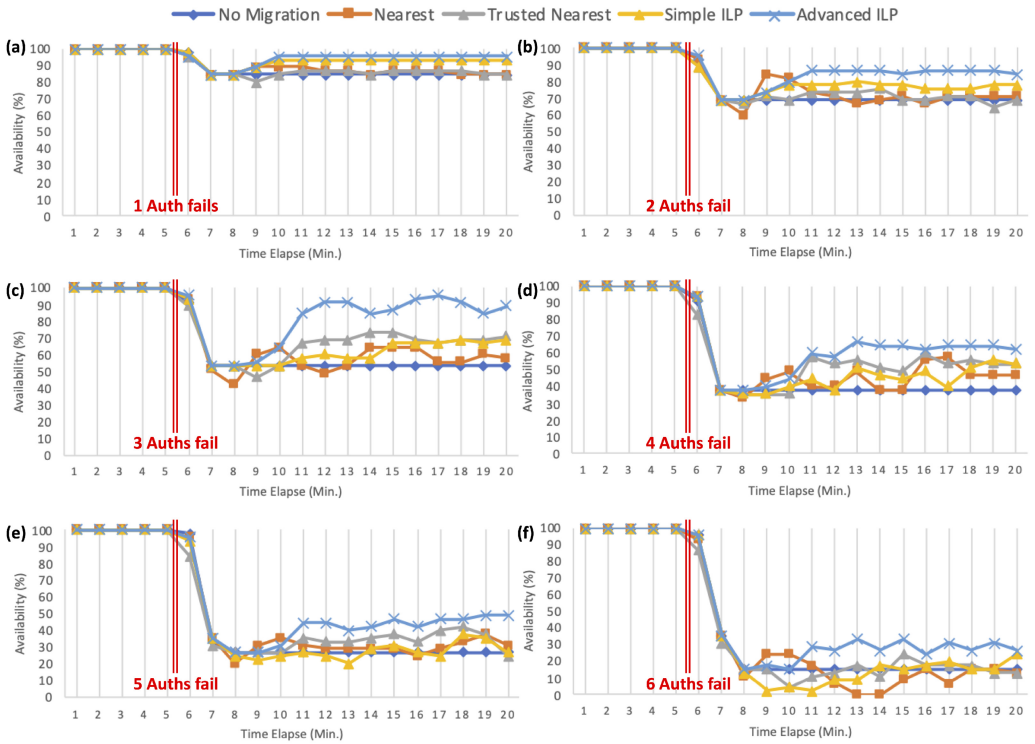


Fig. 12. Availability results with different numbers of failing Auths for four different migration policies using SST simulated on the ns-3 simulator and Linux containers: when 1–6 Auths fail.

Distance Propagation Loss Model in ns-3 to represent the channels between Auths and Things. As a simulation platform, we use Ubuntu Linux 16.04.2 LTS on Amazon's AWS EC2 "m4.4xlarge" with 16 CPUs, 64 GB RAM, and 256 GB SSD.

4.2 Experimental Results

We ran simulations with seven Auths, 35 door controllers (19 on the fifth floor and 16 on the fourth floor), and 45 user devices (26 on the fifth floor and 19 on the fourth floor) for 20 minutes for each experiment in real-time, 5 minutes before Auth failures and 15 minutes after failures. We let the whole system run for 5 minutes before the failure to allow Auths to finish backup processes for secure migration during the normal operations. Each user device sends a door opening request to the closest door controller every minute, simulating the user behaviors in front of doors.

Figure 12 illustrates the experimental results. We performed experiments with up to six Auths failing during the experiments. The failure occurs in order of Auth 504, Auth 402, Auth 501, Auth 403, Auth 503, and Auth 401, where the Auth IDs are as denoted in Figure 10. The Auth capacity, defined as the maximum authorization requests that each Auth can handle per minute, is also shown in Figure 10. When all seven Auths failed, the availability became 0% in our experiments as we expected, although we did not include this in Figure 10. It will be the same when the authorization services based on the cloud lose connections with the cloud. We also compared four different migration policies: (1) *No Migration*, (2) *Nearest*, (3) *Trusted Nearest*, (4) *Simple ILP*, and (5) *Advanced ILP*.

1. **No Migration**—No secure migration is used, same as the original SST.
2. **Nearest**—IoT entities (both door controllers and door opening applications) try to migrate to the nearest available Auth first, then try the next nearest Auth, and so on for all Auths. They repeat this until they can migrate.
3. **Trusted Nearest**—Similar to *Nearest*, but this one considers trust relationship among Auths as well as proximity. IoT entities try to migrate to the nearest available Auth that is trusted by their original Auth, then try the next nearest Auth among those that are trusted by their original Auth.
4. **Simple ILP**—Uses migration policies constructed using the proposed ILP-based approach but only with limited consideration of the communication criticality between IoT entities.
5. **Advanced ILP**—Uses migration policies constructed using the proposed ILP with more considerations, including trusts among Auths and capacity of each Auth.

For Simple ILP and Advanced ILP, we precomputed a series of migration policies *offline* before running the experiments. We measured the computation time for migration policies construction using ILP. With our open-source implementation using Gurobi ILP solver library (explained in Section 3.2), the program execution took 491 ms and 505 ms for Simple ILP and Advanced ILP, respectively, including all I/O operations, on a MacBook Pro machine with 2.2 GHz Intel Core i7 and 16 GB DDR3 RAM.

Each migration policy was constructed as an assignment of Things when a specific Auth becomes not available. The assignment of Things for multiple Auth failures is represented as a list of Auths that the Thing can migrate to. For example, assume that there is a Thing associated with Auth 504 and the assignment for the Thing when Auth 504 is not available is Auth 402, and the assignment for the Thing when Auth 402 is also not available is Auth 501. Then, the list of Auths to which this Thing can migrate will start with Auth 402, Auth 501, and so on.

Also, for Nearest, Trusted Nearest, Simple ILP, and Advanced ILP, each Auth backs up authentication and authorization information of its entities to all of its trusted Auths before the failures, using the Auth backup protocol described in Section 3.2.3. In our experiments, we assumed the same LAN interface cards were used by Auths and the same Wi-Fi interface cards were used by Auths and IoT entities, leading to the same communication costs for the same communication channels. We also assumed the security requirements of IoT entities were the same.

As shown in Figure 12(a), when one Auth failed, the availability dropped down to 84% with No Migration, Trusted Nearest, or Nearest; while Simple ILP migration and Advanced ILP policies recovered the availability up to 93% and 96%, respectively. Similarly, when two Auths failed (in Figure 12(b)), Simple ILP and Advanced ILP recovered up to 78% and 87% availability, respectively, while the availability of No Migration, Trusted Nearest, and Nearest converged to approximately 70%. The differences were more significant when more Auths became unavailable. For example, in Figure 12(c), the availability became around 53% for No Migration and 58% for Nearest migration when three Auths failed. However, Trusted Nearest and Simple ILP recovered around 69% availability and Advanced ILP recovered above 85% of availability. The Nearest migration policy could not recover as much availability as other migration policies due to the fact that it did not consider the trust relationships between Auths properly, thus leading to a migration policy that was infeasible for some IoT entities.

When there were only three or fewer Auths left available, the Simple ILP migration technique became no better than Nearest and Trusted Nearest, or even worse than Trusted Nearest sometimes, as we can see in Figures 12(d), (e), and (f). This was partly because some IoT devices ended up being stuck at an Auth that the IoT devices could not trust while they are trying to reconnect to other Auths, due to the lack of consideration for trust between Auths in constructing migration

Table 2. Considerations Applied in Migration Policy Construction for Each Approach Used in Experiments

Considerations	Nearest	Trusted Nearest	Simple ILP	Advanced ILP
Proximity (Distance between Auths and Things)	O	O	X	X
Trust relationship among Auths	X	O	X	O
Communication requirements among Things	X	X	O	O
Capacity of Auths	X	X	X	O

"O" for the ones applied and "X" for those that are not.

policies. Another important reason for this was that the Simple ILP migration assigned too many IoT devices to one Auth, failing to balance the authentication and authorization workload among Auths. Table 2 summarizes the conditions considered in migration policy construction for each approach used in experiments. Note that Simple ILP and Advanced ILP did not consider proximity, since modeling the distance between Auths and Things as costs in ILP is out of the scope of this article. Modeling distance as part of the ILP is a separate non-trivial problem including research on modeling radio signal strength, noise, and environments.

The fluctuation with the Nearest, Trusted Nearest, and Simple ILP migration was caused mainly by the infeasible migration requests and their interference with normal requests, especially when the Auths are overloaded with requests over their capacity. Even the Advanced ILP migration was not able to achieve 100% availability, because some of the entities were out of the signal ranges of Auths and some Auths could not afford the authorization workload due to their capacity limits. Overall, the proposed Advanced ILP solution maintained significantly higher availability compared to other migration policies.

4.3 Extended Experimental Results

We further extended the evaluation of our approach by performing a series of additional experiments to measure the availability under different circumstances. Specifically, we varied the (1) trust relationships between Auths and (2) Auth failure orders. Changing trust relationships between Auths allowed us to see the impact of different initial configurations and connectivity. Furthermore, varying Auth failure orders resulted in different patterns of skewed load (e.g., how many Things have to migrate to a certain Auth) and connectivity among Things (e.g., connectivity after migration).

The results for this series of extended experiments are shown in Figure 13. We used three different trust relationship configurations and three different Auth failure orders. Then, we measured the average availability for 15 minutes after the failure of Auths, which corresponds to the minutes from 6 to 20 in Figure 12 of Section 4.2.

From the results, we observed that the migration policy based on the Advanced ILP approach achieves higher average availability after the failure of Auths in most cases. However, there were configurations where the Trusted Nearest or Nearest outperformed Advanced ILP, for example, when 3 and 4 Auths failed in Figure 13(d), when 4 Auths failed in Figure 13(f), and when 2 Auths failed in Figure 13(g).

We found a couple of main reasons for these consequences. First, the distance between Auths and Things becomes critical under certain circumstances. To be specific, the network packet loss rate increase as the distance between wireless interfaces of an Auth and a Thing. This makes the distance an essential factor to be considered, especially when the two devices are far enough to cause packet losses. As illustrated in Figure 11 of Section 4.1, Auths and Things are connected via Wi-Fi channels simulated by ns-3. Our ns-3 simulator setup takes the actual physical coordinates of the devices; thus, it also simulates packet losses due to radio signal attenuation. However, our

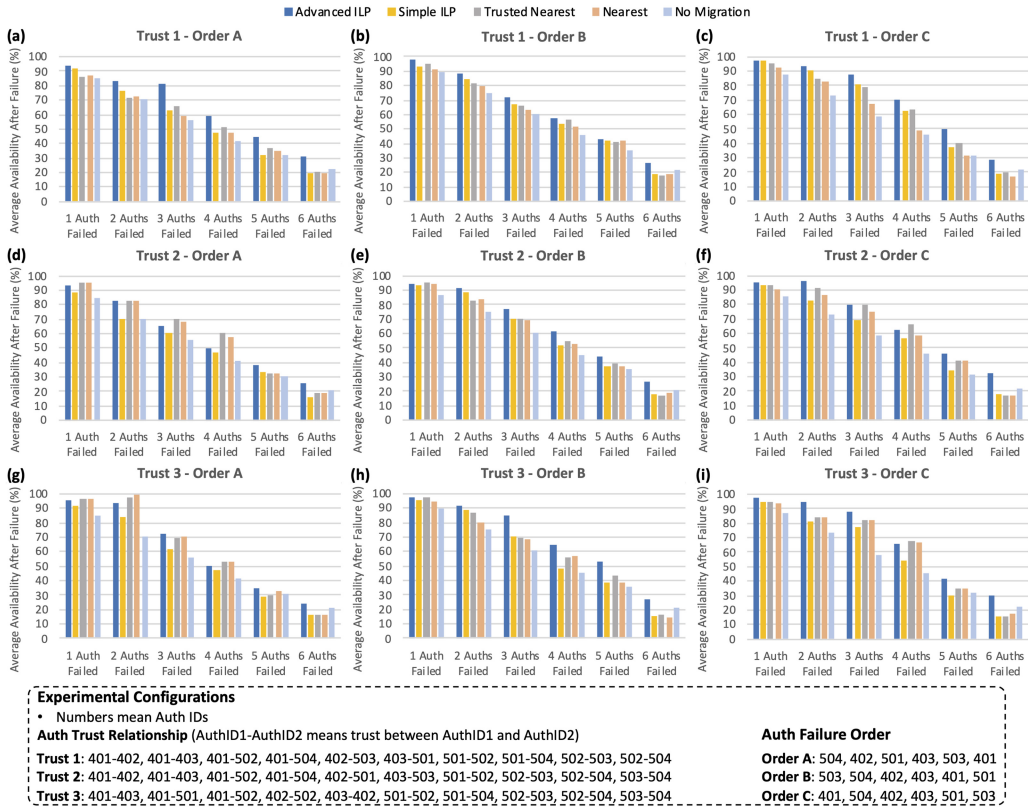


Fig. 13. Average availability for 15 minutes after Auth failures with three different Auth trust relationship configurations and three different Auth failure orders.

proposed ILP model does not take the distance between Auths and Things into account, as shown in Table 2 of Section 4.2. Therefore, in some cases where the ILP ends up migrating a Thing to an Auth that is too far from the Thing, then it underperformed compared to Trusted Nearest or Nearest migration policies. Modeling the distance between Wi-Fi devices as a cost in the ILP formulation is not a trivial problem and thus out of the scope for this article.

Another reason is the noise in the network simulation and the imperfect emulation of the Auth capacity. Auth's capacity control is implemented in terms of the number of requests per minute that an Auth can handle. In theory, the capacity considerations in the ILP modeling should match the implementation; however, because some network packets get delayed and arrive like in a real wireless network environment, causing some packets to arrive at Auths at once. This leads to unexpectedly rejected or accepted requests on Auths, depending on when the packet burst happens, making the availability measurement subject to change.

We also observed that in all 9 cases, when 6 Auths failed, the availability for Migration approaches except for Advanced ILP was no better or even worse than No Migration. This is because Simple ILP, Trusted Nearest, and Nearest approaches tried to migrate Things beyond Auths' capacity, disrupting other requests with migration requests, as happened in the case where there was only one Auth left in Figure 12(f) of Section 4.2. However, Advanced ILP did not try to migrate over Auths' capacity and, thus, did not create excessive migration requests, still achieving higher availability than No Migration.

4.4 ILP Execution Time

We also measured the execution time of solving the migration plan construction problem. We used the Java implementation of ILP solver, which is a part of the open-source implementation of SST. The migration solver included the Gurobi ILP solver library version 8.01 and ran on a MacBook Pro machine with 2.2 GHz Intel Core i7 and 16 GB DDR3 RAM. For each configuration used for Section 4.3, the ILP computation time took 500 ms on average. Also, we assumed the ILP computation is done offline for the experiments.

The ILP computation time will surely increase if there are more Auths, Things, or connections among Auths and Things. However, the execution time measurements show that the ILP computation time would be within a reasonable range for solving the migration problem even under a realistic environment with 7 Auths, 70 Things, 10 Auth-to-Auth connections, 45 Thing-to-Thing connections, 70 Auth-to-Thing initial connections (490 Auth-to-Thing connections in total, including exploration of migration policies). The scale of this setup is far beyond a simple toy example, and thus, to the best of our knowledge, we claim that the execution time for offline ILP computation will be feasible for most real-world applications.

4.5 Threats to Validity

Our experimental settings were based on realistic environments using floor plans of an actual building on the UC Berkeley campus. The realistic environments include actual Auth and Things implementation (in Java and Node.js, respectively) and real-time (synchronized to wall-clock time) network simulation using actual physical locations with packet-level accuracy. However, we did not extend the experiments to the extent where we conduct experiments with random configurations to test exhaustive possibilities. One of the main reasons for this was the temporal and computational cost of experiments. Each experiment takes more than 40 minutes per configuration, including credential creation and setup for Auths and Things, and initialization and tear-down of Linux containers and ns-3 simulation. In addition, it is already costly to run 77 (7 for Auths and 70 for Things) virtual machines (Linux containers) with 84 network interfaces, 7×2 (one wired and one wireless for each Auth) + 70 (one wireless for each Thing), and ns-3 simulator real-time mode for 84 network interfaces and the network packets among them. This hindered us from expanding the scale of experiments to hundreds or thousands of Auths or Things. We acknowledge that we have not verified that our approach works well when the environments are randomly configured or under corner cases.

5 RELATED WORK

To the best of our knowledge, we are aware of only a couple of other approaches that rely on an edge computing architecture for IoT security. To provide robust authorization of medical IoT devices without dependency on remote servers, SEA (Secure and Efficient Authentication and Authorization Architecture) [Moosavi et al. 2015] uses distributed smart e-health gateways as local authorization centers based on DTLS (Datagram TLS). TACIoT (Trust-aware Access Control for the IoT) [Bernabe et al. 2016] employs an architecture based on entities called IoT bubbles, which act as local units of authorization for the IoT, similar to Auths. As far as we know, while these approaches ensure the confidentiality and integrity of communication among IoT devices, they do not provide resilience against availability attacks.

Besides the two mentioned in the previous paragraph, a number of other authorization and authentication services for IoT devices have been proposed [Amazon Web Services 2018; Cirani et al. 2015; Neto et al. 2016; Soldatos et al. 2015; Vučinić et al. 2015]. However, all of these approaches rely on remote, cloud servers to provide critical security services. We believe that the local, distributed

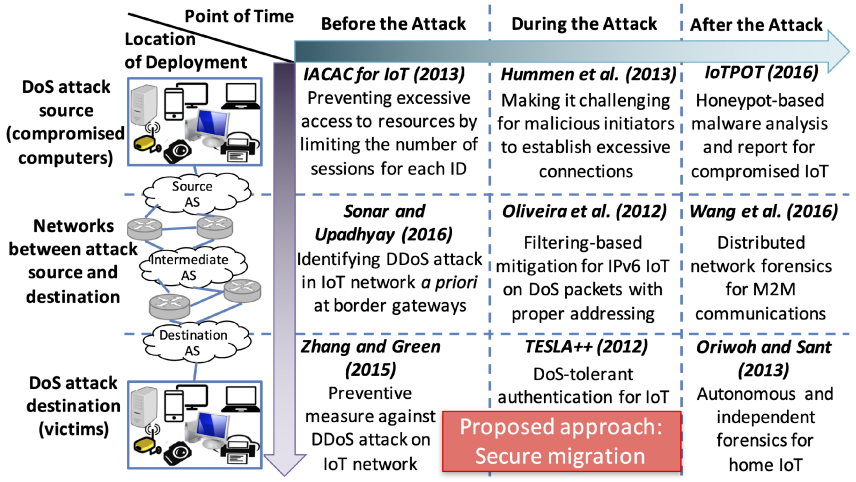


Fig. 14. Countermeasures against DoS attacks to the IoT classified by two criteria used in Zargar et al. [2013]: deployment location and point of time.

nature of Auths enables a more lightweight, flexible defense against DoS attacks: Our approach allows an Auth to continue providing critical services to its local network without requiring a connection to the Internet.

There exists a large body of work that addresses DoS attacks in the network security literature (a comprehensive survey can be found in Zargar et al. [2013]). These works can be categorized as (1) preventing a DoS attack before it takes place [Mahalle et al. 2013; Sonar and Upadhyay 2016; Zhang and Green 2015], (2) limiting the effect of an on-going attack by filtering malicious connections [Oliveira et al. 2013; Ruan and Hori 2012], or (3) performing forensics after an attack to identify compromised IoT devices [Oriwoh and Sant 2013; Pa et al. 2016; Wang et al. 2016]. As far as we know, none of these approaches involve providing resilience against a DoS attack through migration of critical services from an attacked node to another.

Although we do not limit the scope of DoS attacks to DDoS attacks in this article, we use the DDoS attack classification criteria presented by Zargar et al. [2013] to position the proposed secure migration technique in this article. Figure 14 illustrates a variety of countermeasures and mitigations against the DoS attacks to the IoT or the similar networked systems. The horizontal axis in Figure 14 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 [Mahalle et al. 2013], which limits excessive access activity by imposing capability associated to IoT devices. An approach proposed by Sonar and Upadhyay [2016] can prevent the DDoS attack traffic from reaching the IoT devices by identifying the attack traffic *a priori* at border gateways. Zhang and Green [2015] 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. [2013] have presented an approach that makes it difficult for malicious

entities to create excessive connections from the attack source. The solution proposed by Oliveira et al. [2013] mitigates the DoS traffic with filtering for the IPv6-based IoT. TESLA++ [Ruan and Hori 2012] supports DoS-tolerant authentication at the destination by extending the original TESLA (Timed Efficient Stream Loss-tolerant Authentication) protocol [Perrig et al. 2005].

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 [Pa et al. 2016], 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. [2016] 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 and Sant [2013] have proposed a user-manageable forensics solution for the home IoT.

The secure migration mechanism presented in this article 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 14. 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.

6 CONCLUSIONS

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 formulation of migration policy construction allows the proposed approach to be not only valid but also optimal with respect to the overall network costs and capacity of edge computers. Also, the secure migration protocol supports heterogeneous IoT devices, for example, whether they are capable of public-key cryptography or reliable TCP connections. The proposed secure migration approach can be appropriate for the Internet of Things under safety-critical environments, including medical centers, manufacturing systems, and electric power grids so the proposed infrastructure can maintain as much availability as possible.

As future work, we plan to research how to distribute migration policy construction across multiple Auths and extend it to consider different Auth failure cases and scenarios. In addition, we plan to investigate secure migration on various types of network architectures, including network topologies and communication protocols. To address the threat of DDoS attacks launched by the IoT botnets, we will also study the ways to mitigate the DDoS traffic from IoT devices by leveraging edge computing.

REFERENCES

Amazon Web Services. 2018. How the AWS IoT Platform Works—Amazon Web Services. Retrieved from http://aws.amazon.com/iot-platform/how-it-works/.

Jorge Bernal Bernabe, Jose Luis Hernandez Ramos, and Antonio F. Skarmeta Gomez. 2016. TACIoT: Multidimensional trust-aware access control system for the Internet of Things. *Soft Comput.* 20, 5 (May 2016), 1763–1779.

Simone Cirani, Marco Picone, Pietro Gonizzi, Luca Veltri, and Gianluigi Ferrari. 2015. IoT-OAS: An OAuth-based authorization service architecture for secure services in IoT scenarios. *IEEE Sens. J.* 15, 2 (Feb. 2015), 1224–1234.

John N. Hooker and María Auxilio Osorio Lama. 1999. Mixed logical-linear programming. *Disc. Appl. Math.* 96–97 (1999), 395–442.

René Hummen, Hanno Wirtz, Jan Henrik Ziegeldorf, Jens Hiller, and Klaus Wehrle. 2013. Tailoring end-to-end IP security protocols to the Internet of Things. In *Proceedings of the 21st IEEE International Conference on Network Protocols (ICNP'13)*. 1–10.

Hokeun Kim, Eunsuk Kang, David Broman, and Edward A. Lee. 2017a. An architectural mechanism for resilient IoT services. In *Proceedings of the 1st ACM Workshop on the Internet of Safe Things (SafeThings'17)*. ACM, New York, NY, 8–13.

Hokeun Kim, Eunsuk Kang, Edward A. Lee, and David Broman. 2017b. A toolkit for construction of authorization service infrastructure for the internet of things. In *Proceedings of the 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*. ACM/IEEE, 147–158.

Hokeun Kim and Edward A. Lee. 2017. Authentication and authorization for the internet of things. *IT Prof.* 19, 5 (Sept. 2017), 27–33.

Hokeun Kim, Armin Wasicek, Benjamin Mehne, and Edward A. Lee. 2016. A secure network architecture for the internet of things based on local authorization entities. In *Proceedings of the 4th IEEE International Conference on Future Internet of Things and Cloud*. IEEE, 114–122.

Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015), 37–42.

Tom H. Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. 2015. Fog computing: Focusing on mobile users at the edge. *Arxiv:1502.01815 [cs]* (Feb. 2015). Retrieved from http://arxiv.org/abs/1502.01815.

Parikshit N. Mahalle, Bayu Anggorojati, Neeli R. Prasad, and Ramjee Prasad. 2013. Identity authentication and Capability Based Access Control (IACAC) for the internet of things. *J. Cyber Sec. Mob.* 1, 4 (2013), 309–348.

Sanaz R. Moosavi et al. 2015. SEA: A secure and efficient authentication and authorization architecture for IoT-based healthcare using smart gateways. *Procedia Comput. Sci.* 52 (Jan. 2015), 452–459.

Ian Morris. 2017. Google's latest failure shows how immature its hardware is. *Forbes* (Feb. 2017). Retrieved from http://www.forbes.com/sites/ianmorris/2017/02/24/googles-latest-failure-shows-how-immature-its-hardware-is/.

Antonio L. Maia Neto et al. 2016. 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 (SenSys'16)*. ACM, New York, NY, 1–15.

Luís M. L. Oliveira, Joel J. P. C. Rodrigues, Amaro F. de Sousa, and Jaime Lloret. 2013. Denial of service mitigation approach for IPv6-enabled smart object networks. *Concur. Comput.ation: Pract. Exper.* 25, 1 (Jan. 2013), 129–142.

Edewede Oriwoh and Paul Sant. 2013. The forensics edge management system: A concept and design. In *Proceedings of the IEEE 10th International Conference on Ubiquitous Intelligence and Computing and the IEEE 10th International Conference on Autonomic and Trusted Computing*. 544–550.

Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2016. IoTPOT: A novel honeypot for revealing current IoT threats. *J. Inf. Proc.* 24, 3 (May 2016), 522–533.

Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. 2005. The TESLA broadcast authentication protocol. *RSA CryptoB.* (July 2005).

George F. Riley and Thomas R. Henderson. 2010. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*. Springer Berlin, 15–34.

Na Ruan and Yoshiaki Hori. 2012. DoS attack-tolerant TESLA-based broadcast authentication protocol in internet of things. In *Proceedings of the International Conference on Selected Topics in Mobile and Wireless Networking*. 60–65.

Weisong Shi and Schahram Dustdar. 2016. The promise of edge computing. *Computer* 49, 5 (May 2016), 78–81.

John Soldatos, Nikos Kefalakis, Manfred Hauswirth, Martin Serrano, Jean-Paul Calbimonte, Mehdi Riahi, Karl Aberer, Prem Prakash Jayaraman, Arkady Zaslavsky, Ivana Podnar Zarko, Lea Skorin-Kapov, and Reinhard Herzog. 2015. OpenIoT: Open source internet-of-things in the cloud. In *Interoperability and Open-Source Solutions for the Internet of Things*. Springer, Cham, 13–25.

Krushang Sonar and Hardik Upadhyay. 2016. An approach to secure internet of things against DDoS. In *Proceedings of the International Conference on ICT for Sustainable Development*. Springer, 367–376.

Mališa Vučinić, Bernard Tourancheau, Franck Rousseau, Andrzej Duda, Laurent Damon, and Roberto Guizzetti. 2015. OSCAR: Object security architecture for the internet of things. *Ad Hoc Netw.* 32 (Sept. 2015), 3–16.

Kun Wang, Miao Du, Yanfei Sun, Alexey Vinel, and Yan Zhang. 2016. Attack detection and distributed forensics in machine-to-machine networks. *IEEE Netw.* 30, 6 (Nov. 2016), 49–55.

Saman Taghavi Zargar, James Joshi, and David Tipper. 2013. A survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) flooding attacks. *IEEE Commun. Surv. Tutor.* 15, 4 (2013), 2046–2069.

Congyingzi Zhang and Robert Green. 2015. Communication security in internet of things: Preventive mMeasure and avoid DDoS attack over IoT network. In *Proceedings of the 18th Symposium on Communications & Networking (CNS'15)*. Society for Computer Simulation International, 8–15.

Received October 2018; revised September 2019; accepted November 2019