2023-01-04

Context-Aware History-Based Access Control for IoT Devices

Shadman, Shauvik

UNIVERSITY OF CALGARY

Context-Aware History-Based Access Control for IoT Devices

by

Shauvik Shadman

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

JANUARY, 2023

Abstract

Internet-of-things (IoT) devices has taken over every aspect of our daily lives. They control the environment around us and collect personal data. Access to these devices need to be protected, specially in a post-pandemic world where patient data security is of utmost importance. The accessibility of IoT devices is often found to be context dependent, meaning that whether a user may interact with a device often depends on contextual information such as environmental parameters (e.g., time, temperature, pressure).

This thesis is a proposal to add context dependency into a previously published distributed authorization system for IoT devices. In this authorization system, the more proof of context a requester can produce, the more access they are granted. A security property is also put forward to make sure that a malicious requester cannot gain more access by willfully withholding context information. I formally proved that the proposed authorization system satisfies this security property. This thesis also presents an implementation of this authorization system and evaluates its performance.

Acknowledgments

I would firstly like to thank my supervisor, Dr. Philip W.L. Fong for his guidance and his advice throughout my time here. I have learned, in meticulous details, the proper way to approach ideas in research and in life from him. I would also like to thank all the faculty and staff at the university who have taught me to look at things from many different perspectives.

I would like to thank my parents and my sister for their unending love, support and patience throughout my time here. I am what I am because of them. Everything I do, everything I am is for them and because of them. My work in this thesis is dedicated to the three of you.

I would like to thank all my peers and friends spread out over Canada, Dhaka and Australia for always being there when I needed help. It was a lonely and tough battle, but you made it better. A very very special thanks to *Muhammad Younus Nobin*, *Dr. Tanvir Hossain*, *Talha Siddique*, *AKM Nivrito*, *Ahmed Farazi* and *Rahian Islam*. This would not be possible without you. At all. Period. Thank you for not letting me fall into the abyss of insanity.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
IoT	Internet of Things
HCAP	History-Based Capability System
NFA	Non-Deterministic Finite Automata
DFA	Deterministic Finite Automata
SA	Security Automata
DFSA	Deterministic Finite Security Automata
CASA	Context-Aware Security Automata
PoLP	Principal of Least Privilege
TPIW	Tamper Proof against Information Withholding
REST	Representational State Transfer
CoAP	Constrained Application Protocol
PKI	Public Key Infrastructure
SIC	Sensor Integration Centre
API	Application Programming Interface

Chapter 1

Introduction

In an attempt to make our lives easier, we have made use of Internet of Things (IoT) devices. IoT devices are physically small and are able to wirelessly connect to the internet and communicate with other IoT devices and/or servers. IoT devices range from having personal use such as wearable watches (smartwatch, fitbit), smart vehicles, home automation (smart lock, smart TV, smart fridge) to having control over all the electrical components in an industry or a building. They are also used in healthcare extensively in recent years to monitor the medical condition of an individual and take necessary steps in ensuring the safety of said individual [19]. In a post-pandemic world, concerns for patient data privacy and security has risen because medical records and data of IoT devices monitoring the health of a person are very private [13, 43, 22]. These devices are computationally constrained and are very duty oriented. Usage of these devices are growing at an exponential rate and it is expected to reach 125 billion devices by 2030 [14]. According to the United Nations, the population projection of Earth by 2030 should be about 8.5 billion [2]. That is around 14.7 IoT devices for every person on Earth. The sheer volume of IoT devices and its popularity cannot be overlooked.

Since these IoT devices collect personal data, control the safety of people and the environment they are in, it is of utmost importance that access to these sensitive devices is moderated [60]. A problem that arises with these devices is that they are embedded physically over a wide area. Due to this, *distributed authorization* is mostly used to enforce access control policies. On the other hand, access control policies need to be generated and maintained centrally. Due to this, a popular choice for enforcing access control policies is a *capability-based system* for distributed authorization [51, 40, 41].

History-Based Capability System (HCAP) is such a capability-based system which takes into

consideration the physical embeddedness of these devices and imposes sequencing constraints on the order that the IoT devices can be accessed [54]. That is, access decisions are based on previous accesses. These constraints are a way of effectively realizing the Principle of Least Privilege [51]. In order to protect against malicious parties, we can make use of HCAP to monitor executions such that the sequencing constraints are followed. Any deviations from the sequencing constraints can be considered as malicious and access can be rejected. Let us look at two examples of policies that HCAP can enforce. The following two examples were put forward by Tandon *et al.* [54].

Example 1. *(Physical Embeddedness.)* Alice is a very hard working individual who stays post 5 p.m. when her organization has locked all doors to outsiders. Alice has special permissions which allows her to unlock certain doors after hours. After her work she goes through doors A, B and C corresponding to lab door, building entrance door and finally campus exit door in that physical order. Alice requires permissions to smart locks A, B and C but sequencing constraints can be imposed on these so that during off-hours Alice unlocks door A first before accessing door B, and unlocks B before unlocking door C. Unlocking door B or C before unlocking door A could mean that an unauthorized party may be trying to enter the campus.

The sequencing constraints on accessing IoT devices are also noteworthy when accesses must comply with a workflow specification. Users of a smart system can be guided to follow operational procedures of, for example, an industrial process. A system aware of such procedures and can aide a user to follow procedures is called *process awareness* [36]. The following example was also put forward by Tandon *et al.* [54].

Example 2. *(Process Awareness.)* An industrial process has many steps consisting of many machines doing specific tasks one after another in a well defined workflow. Let S_1 and S_2 be two sequential steps (S_1 to be done before S_2) in the workflow where some permissions are assigned. S_1 allows permissions p_1 and p_2 to be exercised and S_2 allows p_2 and p_3. It is evident that the workflow steps and the permission assignments together create a form of sequencing constraint where, after p_3 is executed, p_1 cannot be executed because executing p_3 implies S_1 is completed.

If there is a violation of this sequencing constraint, it might be the case that there is something wrong with the machines, or that there is a malicious party misusing the equipment.

In HCAP, these sequencing constraints are prescribed by a central entity called the *Authorization Server*. The Authorization Server is tasked with managing access control and policy administration. The Authorization Server provides an unforgeable token called a *capability*, which contains sequencing constraints in the form of a Security Automaton (SA) [54]. An SA is much like a Non-Deterministic Finite Automata (NFA) where there are states and transitions. Transitions are labeled with permissions. A transition originating from one state and resulting to the same state is called a stationary permission, and if the transition results in a different state, that is called a transitioning permission. A user (or Client) can then use that capability to request access to an IoT device from an authorizing entity called a *Resource Server*. In HCAP, the Resource Server does not maintain any knowledge of the access control policies and only operate on the information in the capability. Having a centralized policy *administration* and a decentralized policy *enforcement* created a clear separation of duties in HCAP where the Resource Server operates on very little information, i.e., only the information provided in the capability. When presented with a capability by a Client, the Resource Server only validates the capability. If the permission requested is a transitioning permission in the SA in the capability, the Resource Server provides an updated capability and invalidates the existing capability. In this way, by only validating a capability, HCAP also addresses the issue of the constrained computational nature of IoT devices. This was a brief explanation of how HCAP works on a higher level. The research presented in this thesis is an extension of HCAP named *Context-Aware History-Based Capability System*. Let us discuss what is meant by context-awareness in this work.

Dey *et al.* [8] defines *context* as any information that is relevant to understanding the situation of an entity. Entity refers to a person, place or object related to the user of a system and the machine interacting with the user. For example, the current *location* of a person, the current *temperature* of a room or the current *state* of a machine in a production line all falls under the definition of context.

It could also be the case that the set of all three examples together is a context. A *context-aware* system is such a system that not only has access to the context of an entity[1] but also uses this information, wherever relevant, to allow proper functionality of the system [8]. Such context can be provided for, by sensors. The state or situation of any entity can be validated by the readings of sensors. In today's world, we have sensors for measuring almost anything [18] and the readings they provide can be used to provide context; an idea of the environment. A fleet of sensors together can form a coherent picture of the environment, which can be used to construct a "context-aware" system.

Previously, we have talked about the importance of access control in IoT devices and HCAP. We can use readings of multiple sensors to understand the context in which access is requested. In addition to the sequencing constraint of HCAP, we can also use contextual data in the authorization decisions. This added layer introduces yet another constraint on top of the sequencing constraint of HCAP in order to better realize the Principle of Least Privilege [51]. This work introduces context-awareness as a constraint on the access of IoT devices in the form of *Context-Aware Security Automaton (CASA)* (Section 3.2), which requires proof of certain conditions to be met in addition to sequencing constraints. For example, if access is needed to a thermostat to change the temperature of a room, we can use the reading of a temperature sensor and a motion sensor together as a part of the authorization decision. We can say that if the current temperature is greater than 18 degrees and if there is someone in the room, then a user can access the thermostat. In this way, the context of the entities: the user and the room is taken into consideration before providing an access decision.

The Clients in context-aware HCAP gathers certain proofs culminating to a context, and provides it to the Resource Server along with the capability. The Resource Server validates the capability and the proof and then provides the access decision. In order for a client to construct the proof, this research introduces an entity called *Sensor Integration Centre (SIC)*. This entity has the power to provide certain digitally signed statements which can be used as proof of a context. The client communicates to relevant SICs in order to construct a proof of a context and provide it to

[1] Or has the means to access the context information of an entity.

the Resource Server along with the capability requesting access to an IoT device. The Resource Server validates the capability (similar to the original HCAP), and also checks the proof provided and then either accepts or rejects the request.

A *first motivation* for this work is making use of the data of the deployed sensors all around us. There is a vast literature on context-aware access control [35, 61, 55, 34, 15, 11] and a growing literature on the need of context in access of IoT devices [10, 25, 23, 50]. But there is a lack of research making *use of contextual data on top of sequencing constraints* for authorization decisions in the current literature. This work fills that gap and takes sensor data on top of sequencing constraints to validate an access request in real time; meaning that the sensor readings can be designed to be valid for very short to long periods of time (e.g., 5 seconds to 15 minutes). The access control policies in this work can be designed in ways that can require proof of a specific context before granting access.

The computing power of the IoT devices are very constrained and these devices are expected to do very little computation and communication in order to provide access decisions without lag. A *second motivation* of this work is to incorporate context-awareness to HCAP in such a way so that the IoT devices do not have to do a lot of work and therefore, the burden of proving the current context is kept with the user. The only extra thing the IoT devices are expected to do is validate the proof that the user provides. In this way, the IoT devices can be free from gathering contextual data and in turn provide access decisions without much lag.

A *third motivation* for this work is realizing Principle of Least Privilege (PoLP) [51]. This principle maintains that a user should only have access to objects that the user needs and no more. This work adds a layer of contextual constraints on HCAP by having the user prove certain context at the time of requesting access of an IoT device, realizing Principle of Least Privilege.

Thesis Statement. Context-Awareness can be feasibly integrated into History-Based Access Control in such a way that: (a) It is *secure*; (b) Policies can be *authored with ease*; (c) The integration can be *feasibly* and *efficiently* implemented to be used in practice.

The contributions of this work in context-aware HCAP are listed below.

1. The first contribution of this work is the design of the Context-Aware History-Based Capability System. It is such a capability system that incorporates contextual data on top of sequencing constraints of History-Based Access Control (Chapter 3).

2. A second contribution of this work is the design of *Context-Aware Security Automaton (CASA)* which is an extension of a Security Automaton (SA) designed by Fred Schneider [52]. A CASA is essentially like an SA but we can only express sequencing constraints on an SA. On the other hand, a CASA can incorporate contextual constraints on top of sequencing constraints. In an SA, transitions are labeled with permissions, but in a CASA transitions are labeled with a permissions and a set of contextual conditions. A detailed description can be found in Section 3.2.

3. This work puts forward a security property called *Tamper-Proof against Information Withholding (TPIW)*. TPIW ensures users with malicious intent will not gain access by withholding certain proof of conditions for a context (Section 3.3). Let's say a client provides p_1, p_2, p_3 as a proof of contextual conditions, but does not gain access. With a malicious intent, the client throws away the proof of certain conditions and provides a subset of proofs, say p_1, p_2 only, and the access control policy grants access. That is, by willfully withholding information, a malicious client may end up gaining access. TPIW prevents this from happening. This research proves that CASAs is Tamper-Proof against Information Withholding (Section 3.3).

4. CASAs are non-deterministic in nature. That means that from one state in the CASA, by providing proofs p_1, p_2, p_3, there could be multiple valid transitions. That means, without knowing the full CASA, choosing one valid transition might leave room for a future access request to fail. Also, the capabilities discussed earlier houses only a *partial specification of CASAs* called a fragment. Using the fragment

of a non-deterministic CASA, the Resource Server cannot choose a transition if there are multiple valid transitions. This research claims the introduction of a class of special type of CASAs called Deterministic CASAs (See Section 3.4.2) which allows Resource Servers to make that choice. Only designing and embedding fragments of these special type of CASAs in capabilities allows the Client to safely access permissions in the future without any hindrance.

5. This work also introduces an algorithm to convert CASAs to deterministic versions of themselves and proves that the deterministic version accepts the same inputs that is accepted by the original CASA (Sections 3.5.1 and 3.5.1).

6. HCAP was implemented with an Authorization Server, Resource Servers and Clients. Context-Aware HCAP inherited and modified these servers to accommodate for context-awareness with the added implementation of Sensor Integration Centres (SICs) (Chapters 4 and 5).

7. A caching mechanism is also introduced to optimize authorization procedures (Section 4.5.4).

8. An empirical evaluation of Context-Aware HCAP was done which evaluates average authorization times and the effect of the caching mechanism on performance of the Context-Aware HCAP (Chapter 6).

In this chapter, we talked about the importance of IoT devices in our lives and why access control is such an important concept in the IoT space. We looked back on History-Based Capability System which imposes a sequencing constraint in the access of IoT devices. Next, we introduced context-awareness as a constraint on top of sequencing constraints of HCAP as an extra layer of security. Then we talked about the motivating factors of this work and claimed some contributions. In the next chapter, we will look at the related works, important concepts, ideas and technologies that are necessary to fully understand this work.

Chapter 2

Background and Related Work

This chapter introduces some relevant techniques, important concepts and technologies that were used during the course of this research. This chapter is divided into three sections where section 2.1 describes the History-Based Capability System (HCAP) in brief. Section 2.1 goes over the basic concepts and introduces the different entities acting in the protocol and the building blocks of the History-Based Capability System. Sections 2.2 and 2.3 go over the protocol in details including the authorization scheme that HCAP follows. Section 2.6 briefly goes over some of the technologies used in the implementation of HCAP in previous work [54].

2.1 History-Based Capability System

History-Based Capability System was put forward by Tandon *et al.* in [54]. It is a lightweight authorization scheme similar to ICAP [26]. An understanding of the History-Based Capability System is necessary in order to explain the system design of the work in this thesis, which is described in details in Chapter 4. In this section however, I will discuss the HCAP protocol, the participants and the algorithms that dictate their behaviour in a brief manner based on the work in [54].

2.1.1 Concepts

The following concepts are briefly introduced in order to make the later concepts easier to understand. These concepts are necessary to facilitate the proper understanding of later sections.

- Finite State Machines

  A finite-state machine or a finite-state automaton is a mathematical model for representing the behaviour of a computing system [29]. This is a state and event driven

abstract machine which can only be in one state among a finite number of states. The finite-state machine can change from one state to another once a specific event takes place, which is known as a transition. As a user interacts with a computing system by giving an input, its state changes and a new state becomes the current state of the finite-state machine. Therefore, a finite-state machine is used to represent the state of a machine at the current time.

Let us consider the following figure of different states of water as ice, water and vapor to visualize how a state machine works. If ice is the current state, once we heat it, the ice becomes water and water becomes vapor. Here, heat is the event which prompts a state change from ice to water and then to vapor. Another event here is cooling, once vapor is cooled down, it turns to water and then to ice.

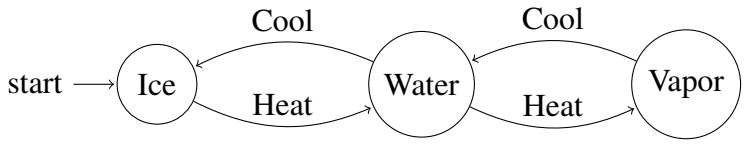

Figure 2.1: **Representation of the states of water using a state machine**

A finite-state machine M is represented by a quintuple [29]:

$$M = (\Sigma, Q, q_0, F, \delta)$$

where,

- Σ is a finite set of symbols representing all possible *inputs* to M.
- Q is a finite set of automaton *states* for M.
- $q_0 \in Q$ is the *initial state* of M.
- $\delta : Q \times \Sigma \nrightarrow Q$ is a partial *transition function* mapping a state and an input symbol to another state.
- $F \subseteq Q$ is a set of *final* or *accepting* states in M.

2.1.2 HCAP protocol participants

There are three participants in HCAP. They are: the Resource Server, the Authorization Server and the Client. They each have their own functionality and purpose. I will explain each of them in the following list. For a much more detailed description please refer to [54].

1. Resource Server

   A resource server is an IoT device which controls the access of several resources. The resource servers act as guardians to several resources. These servers are computationally constrained and are not able to do complex calculations before providing an access decision. An algorithm is in place to provide protection and prevent malicious access to resources. Resource server accepts requests from the client to gain access to resources and after running through an authorization algorithm, provides an access decision. Resources are unitary IoT devices on which a client may perform an operation. The authorization procedure on a set of these resources are carried out by the resource server. A resource server may protect a door lock mechanism, may be in control of traffic lights, emergency procedures, each process in an industry lineup, a lift in a building and also small things like smart television or a smart lightbulb.

   The client not only requests access to the resources, but also requests operations to be done on those resources [54]. For example, a resource like a machine in an industrial process may have a start and a stop function. To the resource server, this machine is a resource and “start” or “stop” are operations that may be performed on this resource.

   A client wishing to perform an operation on a resource is labeled as **“exercising a permission”** in this scope of work. Hence, a permission is a combination of operation and a resource. This operation-resource pair basically says that a client

wishes to perform this specific operation on this resource. This pair is known as a permission in this work following the work in [54].

One of the design objectives of the resource server protocol is to make sure that the algorithm does not take too long to process an access request by the client since the resource server is not computationally well gifted. The resource server has to perform authentication and run an authorization algorithm laid out in HCAP in [54].

2. Clients

   Clients are requesters of the resources protected by the resource server. They move about in the world and exercises permissions in a random order. The clients are real-life people (or their devices) who want to exercise permissions and contain certain tokens which lets them interact with the several servers in play. In HCAP, the clients communicate with the authorization server, gets tokens and goes about exercising permissions by talking to the resource servers.

   The client application is designed to run on any machine with minimal computational power. It is designed to run on a computer as an application or a smartphone with an internet connection. The clients in HCAP only interacts with the authorization server and the resource server. The clients mostly make requests to the resource server.

3. Authorization Server

   There is only one authorization server in this ecosystem. The authorization server is responsible for managing the access of the resources by the clients. It has information about the policies dictating access which are centrally stored in the authorization server. Any administrator has access to the authorization server and they hold the rights to change and add the policies that govern the access control mechanism. An Application Programming Interface (API) is in place, working for the

administrators in the authorization server so that they can control the authorization policies. The clients can perform operations on resources depending on the policies laid out in the Authorization Server.

2.1.3 Trust preliminaries

Before the discussion about the building blocks and protocols, we need to discuss which entities are trusted to do what. This is necessary to determine which components are trusted to work securely and which components have the capacity to act in a malicious fashion.

A brief of the trust mapping of the entities are provided below.

TA-1 The two servers: authorization server and the resource servers are both trusted.

TA-2 The clients are not trusted and may try to gain access that they are not supposed to.

TA-3 A Public-Key Infrastructure is used so that the authorization server and the resource server can authenticate each other and the clients can prove their identity.

TA-4 A secret key is shared between each resource server and the authorization server.

TA-5 All secret values of entities (i.e., clients, resource servers, authorization server) is protected by hardware which cannot be modified.

TA-6 A centralized clock is in place which the entities use to synchronize the time.

2.1.4 Building Blocks

There are a few building blocks in the HCAP system. These notions enable a better understanding of the protocols and the nature of the History-Based Capability System. Features, such as sequential constraints on the access requests of the client and the formation of the tokens to provide to the clients require a proper understanding of the building blocks of the HCAP system. A brief description of the building blocks are presented below.

1. Security Automaton

Schneider introduced the term Security Automata as a subclass of Büchi automata [52]. They are similar to non-deterministic finite-state automata that accept safety properties. A variation of the deterministic security automata by Fong [24] has been adopted by Tandon *et al.* in HCAP [54]. A ***Deterministic Finite Security Automaton (DFSA)*** or in simpler term ***Security Automaton(SA)*** is a quadruple (Σ, Q, q_0, δ) where,

- Σ is a finite set of permissions
- Q is a finite set of automaton states
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \nrightarrow Q$ is a partial transition function mapping a state and a permission to a state, i.e., $\delta(q, p)$ maybe undefined for some pairs of state $q \in Q$ and permission $p \in \Sigma$.

A sequence of permissions from Σ^* may be fed into a SA. If the transition function is defined for each permission in the input sequence, then the sequence is accepted.

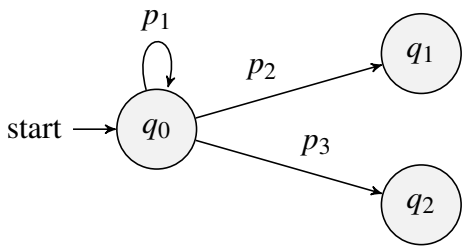

Figure 2.2: **An example of a Security Automaton**

Figure 2.2 is a graphical representation of the Security Automaton $M = (\Sigma, Q, q_0, \delta)$ where, $\Sigma = \{p_1, p_2, p_3\}$, $Q = \{q_0, q_1, q_2\}$ and $\delta(q_0, p_1) = q_0$, $\delta(q_0, p_2) = q_1$, $\delta(q_0, p_3) = q_3$.

Fixing a state, the permissions are classified into two types: **1. the stationary permissions** and **2. the transitioning permissions**. In the previous example, SA M is in state q_0. From the current state q_0, permission p_1 would yield the same state q_0. Such permissions are called stationary permissions. On the other hand, the permissions that would yield a different state than the state the SA is in, are called transitioning permissions. In the previous example, from the current state q_0, permissions p_2 and p_3 would output states q_1 and q_2 respectively. In this case, p_2 and p_3 are transitioning permissions.

2. Fragments

   An SA fragment is a *partial specification* of a full DFSA. Fragments were introduced as a result of the following considerations [54].

   (a) An SA may contain a huge number of states and have very complex transitions from state to state. Encoding the full SA in an authorization token is inefficient considering their size. Therefore, fragments are introduced to be encoded in authorization tokens as they are only partial specifications and are much smaller in size.

   (b) A synchronization goes on between the authorization server and the resource servers, so that the authorization server knows which state the client session is in. If a full SA is provided, then the resource server will need to update the authorization server more often, which would decrease the efficiency of the resource server.

   (c) The communication protocol used in the implementation is designed in a way such that a smaller authorization token would make it extremely efficient.

Due to the reasons mentioned above, a *fragment* is encoded in an authorization

token (instead of the full SA) and provided to the clients. A fragment facilitates a lighter-weight communication between entities because a smaller size of packets of data are transmitted.

A fragment of an SA $M = (\Sigma, Q, q_0, \delta)$ currently in state $q \in Q$ is a triple, $F = (SP, trans, defs)$ [54]. *SP* and *trans* describe the stationary and the transitioning permissions of the *current state q* respectively. *defs* specifies the stationary and transitioning permissions of all the other states in the fragment.

In the fragment F, $SP \subseteq \Sigma$ is a set of stationary permissions which, when executed, outputs the same state as the current state q (permission p_1 in Figure 2.2 is an example). The fragment F also describes $trans : \Sigma \nrightarrow Q \cup \{\circ\}$, which is a finite partial function mapping a permission to the name of a state or an 'unknown', symbolized by a marker '$\circ$'. A distinction is made between the stationary and the transitioning permission because executing a transitioning permission would render the current fragment invalid. This is because when a transitioning permission is exercised, the current state changes and *SP* and *trans* need to be changed. Therefore, a new fragment becomes necessary and the old fragment becomes invalid.

$defs : Q \nrightarrow (SP', trans')$ is also a finite partial function mapping a state name to a pair $(SP', trans')$ where SP' is a set of stationary permissions and *trans'* is defined the same way mentioned before. *defs* is set up in the fragment in such a way so that the client does not have to go to the authorization server every time a transitioning permission is exercised. Information is provided in *defs* so that the resource server can formulate a new fragment with minimum computation, embed it in an authorization token and send it to the client.

An example of an SA fragment represented in the form of a JSON object [33] is given below.

$$\{SP : [p_1], trans : \{p_2 : q_1, p_3 : \circ\}, defs : \{q_1 : \{[p_4], \{p_5 : \circ\}\}\}\}$$

When these SA fragments are encoded in JSON format, it is efficient to compute transitions by an authorizing entity. These computations are done by the RS.

Given a fragment $F = (SP, trans, defs)$ exercising a permission p is expressed as $\Delta(F, p)$ which can be computed resulting in the following possible options.

(a) Return F if $p \in SP$. If a stationary permission is exercised then the state of the SA does not change. So the output is the same fragment F.

(b) Return a new fragment $(SP', trans', defs)$ if $trans(p) = n$ and $defs(n) = (SP', trans')$. If p is a transitioning permission, then the SA transitions to a new state. A new fragment is therefore generated.

(c) Return $\circ$ if $trans(p) = \circ$. If the permission p can be exercised but the next state information is not found in the fragment, then an unknown is returned as a result.

(d) $\Delta(F, p)$ is undefined if there are no permission p that can be performed from the current SA state and a new fragment cannot be generated.

3. Tickets

A ticket is an authorization token used by the client to gain access to resources. The authorization server and the resource server are capable of providing the client with tickets. In order to gain access to a resource, the client provides a ticket to the resource server. These tickets are unforgeable and specific to each client, i.e., ticket provided to one client cannot be used by another client. In order to achieve this, the ticket contains an authentication tag.

The process of creating an authorization tag is inspired by ICAP which uses a lightweight tagging mechanism [26]. Suppose α is an assertion, h is a hash function, *uid* is an identification of a client, k is a shared secret between the authorization server and the resource server and "$x \mid y$" is a concatenation of strings x and y; then a ticket $\langle\langle\alpha\rangle\rangle_{k,uid}$ is $\alpha \mid h(\alpha \mid uid \mid k)$ [54]. The authorization server or the resource server concatenates the assertion α, the client id *uid* and the secret key k. Then the concatenated value is hashed to form $h(\alpha \mid uid \mid k)$. This hash value is appended to the assertion α as the authentication tag.

The purpose of *uid* is ensuring non-transferability of a ticket, therefore, it is used to compute the hash function. On the other hand, the purpose of the shared secret k is ensuring that only the resource server for which the ticket is to be used can rehash this ticket [54]. The client *uid* receives ticket $\langle\langle\alpha\rangle\rangle_{?,?}$ and presents it to an authorizing entity. An authorizing entity can check authenticity of the ticket $\langle\langle\alpha\rangle\rangle_{?,?}$ by checking if the hash value is equal to $h(\alpha \mid uid \mid k)$ [54].

There are two types of tickets in HCAP [54]. They are explained below.

- **Capabilities**

  A capability is a ticket used by a client to request access to resources. The Authorization Server and the Resource Servers can issue capabilities to the client. When the client wants to exercise a permission, the client presents this to the resource server as an access token. Each client is in a session maintained by the authorization server. The capability has the form $\langle\langle$*sessid: cap*$(t_{ser}, F)\rangle\rangle_{k,cid}$, where *sessid* is the id of the session the client is in, t_{ser} is a timestamp and F is a SA fragment. Each session is mapped to a Security Automaton and every capability contains information about which state a client is currently in. Therefore, the resource server can verify if the client

can exercise the permission that was requested by the client from that state. Let the client be in state q, therefore, the client can exercise all permissions allowed from the state q specified in the fragment F as long as they provide the required certificates which fulfill the conditions in that transition. t_{ser} is a timestamp which is a record of when the client entered in state q.

- **Update Request**

  An update request is a ticket that is provided to a client by a resource server. It is constructed by a resource server in the event of a permission leading to an unknown state as mentioned in item (c) of the list in Page 16. It is also known as a **state update request**. When the resource server observes that it cannot create a new capability from a new state, it returns an update request of the form $\langle\langle \textit{sessid: upd}(e)\rangle\rangle_{k,cid}$, where e is a list of transitioning permissions that the client has exercised with the authorizing resource server since the time the resource server synchronized with the authorization server. Here, e is called an **exception** and it describes the permissions the client exercised since the last known state of the client known by the authorization server. The next section describes this in detail.

4. Exceptions

   A record of the authorization history of a client is called "Exceptions". Clients use capabilities to request execution of permissions. If the client exercises a stationary permission, no SA state transition occurs. On the other hand, whenever a client exercises a transitioning permission, the state of the SA is changed. These transitioning permissions need to be recorded. This history of transitioning permissions

are called exceptions.

The clients are not trusted in this protocol as per the assumptions in Section 2.1.3. Therefore, the resource servers are tasked with recording the authorization history of the client. This creates the issue of the authorization server not knowing the current state the SA for that client is in. As a consequence, not only does the resource server have to track the authorization history, it also has to let the authorization server know what was accessed. That is where exceptions come in.

An exception is a list enumerating the permissions exercised since the last update, together with the timestamps of the permission exercising events. Only the transitioning permissions are stored by the resource server because the stationary permissions lead to the same state. If a client exercises a transitioning permission p at time t, then the exception e is defined in an inductive way as follows.

$$e ::= nil(t) \mid ex(p,t,e)$$

In this construct, $p \in \Sigma$ and $t \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This permission and timestamp pairs indicate the permission exercised and the time it was exercised in. These pairs are stored in a descending order of time, such that the first permission exercised is the first entry in e and the last permission exercised is the last permission-timestamp pair stored in e.

The resource server has a separate e for each client session. The authorization server calculates the current state of a client for a session by applying the sequential permissions stored in e to the last known state known to the authorization server.

2.2 Brief Description of the Protocol with a Single Resource Server

This section briefly explains the working principle of HCAP with a single resource server. I will describe how the authorization server works and the internal states it keeps track of, how the resource server authorizes the clients to exercise a permission and also some concepts specific to HCAP which help facilitate the HCAP principles. This section will go over the protocol when there is only one resource server. Tandon *et al.* [54] expanded this protocol for multiple resource servers which are discussed in the next section (Section 2.3).

2.2.1 Authorization Server States and Session Initialization

The authorization server retains some information about each client. The resource server and the authorization server share information about the activities of the client to keep the authorization server up to date about the client session. Every resource server shares a previously agreed upon secret key with the authorization server. The resource server uses this secret key to hash tickets it *receives from* and *generates for* the client.

The authorization server maintains three maps to keep track of the clients sessions and states. They are as follows.

1. $monitor[sessid]$ is a map which maps each session identifier of a client to a security automaton for that specific client.

2. $state[sessid]$ is a map which maps each session identifier of a client to the last known state that the authorization server knows the client to be in[1].

3. $serial[sessid]$ is a map which maps each session identifier of a client to the timestamp referring to the time at which *sessid* entered $state[sessid]$.

[1]The clients go about randomly exercising permissions, which may change their state. It is not immediately reported back to the authorization server, so the authorization servers information is outdated. This is why the last known state stored in by the authorization server may not reflect the state the client is currently in.

4. *fragment*$[M,q]$ is a map which maps every state in every SA to a fragment with q as the current state of the fragment.

For each *SA* M stored in *monitor*$[\cdot]$, the authorization server creates a fragment beforehand for each state in M and populates *fragment*$[M,q]$. It is designed in this way in order to prevent creating the same fragment over and over again because there is a possibility that the clients may end up in the same state for a cyclic *SA*. The authorization server only has to compute these fragments only one time and refer to the *fragment*$[M,q]$ map to get a fragment from M with the current state as q.

The clients who wishes to exercise permissions is required to contact the authorization server first. The authorization server creates a new session for the client as long as the client authenticates itself. Let the client be identified by the user identifier *uid*. The authorization server consults the access control policy set up by the administrators. If the policies are found for *uid*, the authorization server starts a new session for the client. The access control policy is specified as an *SA* $M = (\Sigma, Q, q_0, \delta)$ for *uid* in order to control what kind of permissions and which order is applicable for that client, for that session.

The maps are initialized as follows.

1. *monitor*$[sessid] \leftarrow M$, the *SA* M is assigned for *sessid*.

2. *state*$[sessid] \leftarrow q_0$, the initial state q_0 for the automaton M is assigned to be the last known state that the client is in.

3. *serial*$[sessid] \leftarrow current\_time()$ stores the exact time the client entered in q_0. It also serves as a serial number for *sessid*.

Finally, the authorization server creates a fragment F with q_0 as the current state. This fragment is then put into a capability of the form $\langle\langle sessid : cap(t_{ser}, F)\rangle\rangle_{k,cid}$, where k is the shared secret, $F = fragment[M, q_0]$ and $t_{ser} = serial[sessid]$.

2.2.2 Resource Server Authorization

The resource server is assumed to be computationally constrained and is not able to store a lot of information. This is why the resource server is designed to only maintain the following information.

1. t_{rs}, which is a clock reading or timestamp which stores the minimum serial number of capabilities that the resource server is willing to recognize. All the capabilities of greater timestamps is considered valid by the resource server. If the client presents a capability with a serial number smaller than t_{rs}, then it is considered a replay attack. Tracking t_{rs} also facilitates garbage collection, a feature which accounts for the memory limitations of a resource server. This is explained in a later section.

2. $ex[\cdot]$, which maps each session ID to an exception (explained in Section 2.1.4).

The client when requesting to exercise permission to the resource server does not only present the capability from the authorization server, but also presents some extra information. The client presents a triple $(uid, p, \langle\langle sessid : cap(t_{ser}, F)\rangle\rangle_{?,?})$ where *uid* is a client identification number which is unique, p is the permission the client is requesting and a hashed capability acquired from the authorization server.

After receiving the triple, the resource server executes Algorithm 1 (Authorization procedure of the resource server) [54]. The steps from step 2 onwards are a brief description of Algorithm 1 by Tandon *et al.* [54].

1. The resource server authenticates the client (*uid*) using the PKI in place.

2. [Line $1-2$] To authenticate the capability, the resource server firstly does a hash check using the shared secret (k) with the authorization server and the *uid* (Lines 1 and 2). Secondly, the resource server checks the serial number (t_{ser}) of the capability and compares against t_{rs}. If $t_{ser} > t_{rs}$ ***or*** if it fails the hash check the request is denied by the resource server (Lines 1 and 2).

Algorithm 1: Authorization procedure of the resource server. [54]

Input: A client access request (uid, p, cap), where *uid* is the client's authenticated identity, p is the permission to be exercised, and *cap* is a capability $\langle\langle sessid : \mathsf{cap}(t_{ser}, F)\rangle\rangle_{?,?}$ for session *sessid*, such that $F = (SP, trans, defs)$.

Output: A set of tickets, or a failure response.

Data: The resource server maintains the following persistent data: (a) a secret k it shares with the authorization server, (b) a timestamp t_{rs}, and (c) a map $ex[\cdot]$ that assigns an exception to each known session ID.

if *cap is not signed by k for uid, or* $t_{ser} < t_{rs}$ **then**
  return failure
if $ex[sessid]$ *is undefined, or* $t_{ser} > last(ex[sessid])$ **then**
  $ex[sessid] \leftarrow \mathsf{nil}(t_{ser})$
else if $t_{ser} < last(ex[sessid])$ **then**
  return failure
if $p \in SP$ **then**
  Exercise permission p;
  return $\emptyset$
else if $p \in dom(trans)$ **then**
  Exercise permission p;
  $t \leftarrow current\_time()$;
  $ex[sessid] \leftarrow \mathsf{ex}(p, t, ex[sessid])$;
  $F' \leftarrow \Delta(F, p)$;
  if $F' = \circ$ **then**
    return $\{\langle\langle sessid : \mathsf{upd}(ex[sessid])\rangle\rangle_{k,uid}\}$
  else
    return $\{\langle\langle sessid : \mathsf{cap}(t, F')\rangle\rangle_{k,uid}\}$
else return failure ;

3. [Line $3-6$] If the exception for the session *sessid* in the capability does not exist or if the timestamp on the capability (t_{ser}) is greater than the most recent entry on the exception list ($last(ex[sessid])$), then the exception list for that session is set to null (Lines 3 and 4). On the other hand, if $t_{ser} < last(ex[sessid])$, then the request is denied (Lines 5 and 6), as the request is a replay attack.

4. [Line $7-9$] The resource server then checks the type of requested permission (p) (Line 7). If the permission is a stationary permission, then the resource server exercises the permission (Line 8). No new capability is issued to the client because a state change does not occur. Nothing is added to the exception list either.

5. [Line $10-18$] If the permission requested (p) is a transitioning permission, the permission is exercised and a new capability is created to be given to the client. The resource server computes a new fragment ($\Delta(F,p)$) (Line 14) with the information in the fragment and embeds it in a new capability. Next the resource server signs the capability with the shared secret and communicates the capability to the client (Line 18). In case the resource server does not have enough information to create a new fragment (i.e., $\Delta(F,p) = \circ$), then an *update request* is sent to the client (Line 17). The details of the update request is described in the following section.

2.2.3 Update Requests

This kind of response is triggered when the client exercises a transitioning request but the fragment does not provide enough information ($\Delta(F,p) = \circ$) for the resource server to create a new capability. This means that the client has to involve the authorization server for a new capability. In order to construct a new capability, the authorization server also has to know the current state the client is in. So, the resource server passes an update request of the form $\langle\langle sessid\text{: } upd(e)\rangle\rangle_{k,uid}$ to the client and the client goes to the authorization server to get a new capability. The authorization server accepts the update request if $first(e) = serial[sessid]$.

The authorization server upon receiving the update request, updates its internal states. The authorization server goes through the transitioning permissions exercised and updates the current state of the session. Map $state[sessid]$ is updated to $\delta^*(state[sessid], e)$ where δ is a transition function of $monitor[sessid]$. After updating its internal state, the authorization server issues a new capability to the client with $state[sessid]$ as the current state and the fragment being $fragment[monitor[sessid], state[sessid]]$.

The client can then re-attempt the access request with the new capability. In this way the authorization server knows the current state of the client and the resource server does not have to store the authorization history for an extended period of time.

2.2.4 Garbage Collection

Resource servers are computationally constrained and does not have a lot of memory. HCAP requires the knowledge of authorization history that an entity performed in order to make access decisions. This responsibility of storing the authorization history falls to the resource server, although the resource server has constrained memory. Therefore, to solve this contradiction, garbage collection was introduced in [54].

Garbage collection is a process where the resource server communicates all the exception lists it is holding for all sessions to the authorization server. The authorization server receives the $ex[\cdot]$ map and begins the process of updating the current state of all the client sessions. The authorization server updates all the session states to $\delta'(state[sessid], ex[sessid])$ using the appropriate SAs from $monitor[sessid]$. The first entry is applied to the known state $state[sessid]$ and subsequently arrives at the final state using the final entry in the exception list. Once it gets the final state, it updates $state[sessid]$ and also $serial[sessid]$ is modified to the current time for all sessions.

The authorization server then confirms the resource server that this process was a success. The resource server updates the exception list $ex[\cdot]$ to an empty map. The last step of this process is that the resource server updates t_{rs} to the current time.

Performing garbage collection ensures the resource server memory is not filled up and updat-

ing the t_{rs} ensures all the capabilities that the clients are holding with time stamps before t_{rs} is invalidated. The clients presenting capabilities which have been invalidated are asked to go to the authorization server to ask for a valid capability.

This was a brief description of HCAP with a single resource server. This single resource server protocol for HCAP has been formally verified to avoid replay attacks [54, Lemma 3.1]. Next, we look at an extended version of HCAP which deals with multiple resource servers.

2.3 Brief Description of the Protocol with Multiple Resource Servers

We have discussed the protocols of the core HCAP with a single resource server. This section describes *extended* HCAP as an extension to core HCAP. It is quite impractical to have a single resource server as different resources must be protected by different resource servers. That is why *extended* HCAP is introduced and this extension can accommodate multiple resource servers.

In extended HCAP, there are multiple resource servers which host and protect multiple resources. Each resource server has a unique identifier *rsid* and each resource server maintains a shared secret with the authorization server k_{rsid}. Extended HCAP brings about three new concepts which are explained in brief in the following [54].

1. *Baton Holding.* Previously, with a single resource server model, the authorization history was recorded by that resource server. But with multiple resource servers, it becomes an issue of who records the authorization history and how this record is the same throughout the network. A baton is defined as the exception list $ex[sessid]$ for session *sessid*. If a resource server is maintaining an exception list $ex[sessid]$ for *sessid*, then it is said that that resource server is holding the baton for that session. A global invariant in this design is that **only one** resource server holds the baton for a client in session *sessid*.

   An issue that arises from that is that a resource server only knows if it itself is holding the baton for that session but holds no information of other resource servers.

So, when a client presents a capability for *sessid* to resource server *rsid* which does not hold the baton for that session, then it becomes impossible to differentiate between: 1. no resource server holds the baton, or, 2. there is a different resource server holding the baton. To tackle this, the authorization server maintains a boolean flag for each session, indicating whether some resource server currently holds the baton for that session. Therefore, when the resource server cannot differentiate between the two cases, it consults the authorization server and the authorization server consults the boolean flag.

2. *Remote Capability Validation.* Each resource server in extended HCAP has its own unique shared secret with the authorization server. Moreover, each capability is signed by this shared secret k. With multiple resource servers, let's say a client presents a capability signed by a shared secret k_{rsid_1} to a resource server other than $rsid_1$. Then the resource server will not be able to validate the capability, nor will it be able to rehash the capability should the need occur.

   This is why *remote capability validation* is introduced in extended HCAP. With remote capability validation, resource server $rsid_2$ can request $rsid_1$ to validate the capability which was signed by $rsid_1$ and also holds the baton for *sessid*. This feature requires a change in the structure of the capability which is as follows.

$$\langle\langle vid, sessid : cap(t_{ser}, F)\rangle\rangle_{k,uid}$$

   Here, *vid* is an extra element added to the capability which signifies the identity of the resource server who signed the capability and is responsible for validating that capability.

3. *Baton Passing.* If $rsid_2$ requests $rsid_1$ for remote capability validation, after validation, $rsid_1$ passes the exception list $ex[sessid]$ to $rsid_2$. This event is called *baton*

passing. Now, $rsid_2$ maintains the exception list $ex[sessid]$ for *sessid* and is therefore said to hold the baton for that session. This is done so that the stationary permissions exercised on $rsid_2$ does not have to do remote capability validation. This implies that $rsid_2$ does not have to do extra communication with $rsid_1$ and stationary permissions would be executed efficiently.

Next I will talk about information maintained by the authorization server and how sessions are initialized in extended HCAP. Then I will talk about how the resource server authorizes permissions and briefly talk about garbage collection in extended HCAP with multiple resource servers.

2.3.1 Authorization Server States and Session Initialization

Previously we have seen that the authorization server maintain maps $monitor[\cdot]$, $state[\cdot]$, $serial[\cdot]$ and $fragment[\cdot,\cdot]$. With multiple resource servers in extended HCAP, the authorization server also have to maintain another map $baton[sessid]$ for each session. This map maps a session to a boolean variable signifying if a resource server holds a baton for that session. The authorization server maintains this map to differentiate if the baton for that session is held by any resource server.

The initialization of the other four maps is unchanged from core HCAP. In addition, when a new session is initialized, the $baton[sessid]$ map has a new entry for that session set to false because this is a new session and no resource server holds the baton yet. In other words, no resource server maintains $ex[sessid]$ for that session yet.

2.3.2 Resource Server Authorization

The client presents the following request to resource server *rsid*: $(uid, p, sessid : cap(t_{ser}, F)\rangle\rangle_{k,uid})$. The resource server follows authorization procedure of Algorithm 2 (Authorize Access Request (multiple resource servers)) [54]. I will now describe Algorithm 2 in brief.

To ease understanding of Algorithm 2, let us look at the following Table 2.1 which outlines different symbols representing different items used in the algorithm and their description.

Algorithm 2: Authorize Access Request (multiple resource servers).

Input: A client access request (uid, p, cap), where *uid* is the client's authenticated identity, p is the permission to be exercised, and *cap* is a capability $\langle\langle vid,\ sessid : \mathsf{cap}(t_{ser}, F)\rangle\rangle_{?,?}$ for session *sessid* and validator *vid*, such that $F = (n, defs)$.

Output: A set of tickets, or a failure response.

Data: The resource server maintains the following persistent data: (a) its identity *rsid*, (b) a secret k_{rsid} it shares with the authorization server, (c) a timestamp t_{rs}, and (d) a map $ex[\cdot]$ that assigns exceptions to session IDs.

if $RS(p) \neq rsid$ **then return** *failure* ;
if $vid = rsid$ **then**
    Invoke Algorithm 3 locally on *rsid* to validate *cap* ;
    if *Algorithm 3 fails* **then return** *failure*;
else
    Request *vid* to run Algorithm 3 to validate *cap* ;
    if *Algorithm 3 succeeds* **then**
        vid sends *rsid* the contents of $ex[sessid]$;
        vid deletes its copy of $ex[sessid]$;
        rsid stores the received contents locally in $ex[sessid]$;
    else return *failure* ;
if $p \in SP$ **then**
    Exercise permission p ;
    return $\emptyset$
else if $p \in dom(trans)$ **then**
    Exercise permission p ;
    $t \leftarrow current\_time()$;
    $ex[sessid] \leftarrow \mathsf{ex}(p, t, ex[sessid])$;
    $F' \leftarrow \Delta(F, p)$;
    if $F' = \circ$ **then**
        return $\{\langle\langle rsid, sessid : \mathsf{upd}(ex[sessid])\rangle\rangle_{k_{rsid}, uid}\}$
    else
        return $\{\langle\langle rsid, sessid : \mathsf{cap}(t, F')\rangle\rangle_{k_{rsid}, uid}\}$
else return *failure* ;

Algorithm 3: Validate capability.

Input: An authenticated client identifier *uid* and a capability *cap* of the form $\langle\langle vid,\ sessid : \mathsf{cap}(t_{ser}, F)\rangle\rangle_{?,?}$.

Output: Success or failure.

Data: The resource server maintains the following persistent data: (a) its identity *rsid*, (b) a secret k_{rsid} it shares with the authorization server, (c) a timestamp t_{rs}, and (d) a map $ex[\cdot]$ that assigns exceptions to session IDs.

1 **if** *cap is not signed by* k_{rsid} *for uid* **then return** *failure* ;
2 **if** $ex[sessid]$ *is undefined* **then**
3 **if** $t_{ser} < Time[sessid]$ **then**
4 **return** *failure*
5 Request the authorization server to confirm that (a) $baton[sessid] = false$, and (b) $t_{ser} = serial[sessid]$;
6 on confirmation $baton[sessid]$ is set to *true*;
7 **if** *(a) and (b) are not confirmed* **then return** *failure* ;
8 $ex[sessid] \leftarrow \mathsf{nil}(t_{ser})$
9 **else if** $t_{ser} > last(ex[sessid])$ **then** $ex[sessid] \leftarrow \mathsf{nil}(t_{ser})$;
10 **else if** $t_{ser} < last(ex[sessid])$ **then return** *failure*;
11 **return** *success*

Table 2.1: Symbols used in Algorithm 2.

Symbol	Definition
p	Permission to be exercised
vid	Identity of the validator who signed the capability
sessid	Session identification of the session the client is in
t_{ser}	Timestamp representing the time when the SA entered in the current state
F	Fragment in the capability presented
cap	The capability presented to the resource server. Has the form of $\langle\langle vid, sessid : cap(t_{ser}, F)\rangle\rangle_{?,?}$
(uid, p, cap)	Access request by the client and the input to Algorithm 2
rsid	Identity of the resource server to whom the client sent a request

Algorithm 2 can be broken up into three parts which are described below.

Step 1. The algorithm first checks if the resource being requested is hosted at *rsid*. That is, it checks if *rsid* can exercise permission p (Line 1).

Step 2. In the second part the algorithm compares *rsid* against *vid* (Line $2-11$). Two outcomes are possible which are as follows.

(a) If $rsid = vid$, that means *rsid* is the validator of *cap* and it can invoke Algorithm 3 locally (Lines $2-4$). The purpose of Algorithm 3 is to *validate the capability*. Algorithm 3 serves similar functions as steps 2 and 3 of Algorithm 1 explained in Section 2.2.2. If validation fails, *rsid* responds with a *failure*.

(b) If $rsid \neq vid$, then *rsid* needs to do remote capability validation (Line $5-11$). Here *rsid* requests *vid* to run Algorithm 3 (Line 6). If validation fails, *rsid* returns failure. Otherwise, *vid* sends the contents of

$ex[sessid]$ to *rsid* and deletes its own copy of $ex[sessid]$ (Lines $8-9$). *rsid* then stores the contents of $ex[sessid]$ locally (Line 10). In other words, *vid* passes the baton to *rsid*.

Step 3. This part has the same functionality as steps 4 and 5 of Algorithm 1 explained in Section 2.2.2 (Lines $12-23$). In this part of the algorithm, the type of permission is checked and behaves exactly like Algorithm 1 with two differences.

(a) The capabilities and the update requests returned to the client is signed by *rsid* instead of *vid* (Lines 21 and 23).

(b) When p is a stationary permission, it is guaranteed that $rsid = vid$ (Lines $12-14$) and no new capabilities are issued [54].

This was a brief description of Algorithm 2 to authorize access request for multiple resource servers. Next I will briefly talk about how garbage collection is changed in extended HCAP.

2.3.3 Garbage Collection with Multiple Resource Servers

We have discussed garbage collection previously in Section 2.2.4. Let us look into garbage collection in the context of multiple resource servers.

Algorithm 3 performs capability validation either locally or remotely (remote capability validation). If the resource server performing Algorithm 3 does not find an exception defined for that session, that resource server contacts the authorization server to confirm two conditions. Either the client is presenting capability for the first time or, the exception list has been garbage collected. Now if the client is exercising a lot of permissions, that would result in more garbage collection. More garbage collection would trigger more resource servers contacting the authorization server in order to perform Algorithm 3. This would create excessive communication overhead between the authorization server and the resource server.

That is why two kinds of garbage collection are introduced.

1. Soft Garbage Collection.

   In the event of soft garbage collection, the exception list for a client in a session is not completely removed. That means, in soft garbage collection, $ex[sessid]$ is set to $nil(t)$ instead of removing the exception list altogether. Therefore, after soft garbage collection, the exception list is not undefined. Algorithm 3 would find that the resource server still holds the baton for that client. This improves the issue of communication overhead.

2. Hard Garbage Collection.

   Soft garbage collection would mean that there is a *nil* entry for a lot of sessions and a resource server can very easily get bloated with all this storage as they are constrained devices. With hard garbage collection, these *nil* entries are cleared out. Hard garbage collection is triggered when a session has a *nil* entry and has been inactive for a long time. The inactive time threshold can be set by administrators such that hard garbage collection will be triggered if threshold amount of time has passed since soft garbage collection.

These two kinds of garbage collection solves the issue of bottleneck by having resource servers holding batons for sessions for a longer period of time.

2.4 A Short Survey of Distributed Authorization System

A very brief discussion on distributed authorization schemes such as ABLP, RT and SPKI is presented in this section.

2.4.1 ABLP

Abadi *et al.* [7] presented a calculus of access control that is based on the notion of *principals* and connects authentication and authorization control. This work also takes into account the notion

of delegation of privileges among the principals. These principals can *make requests* and *make statements*. Principals can be simple or composite. Simple principals include users, machines or communication channels. Composite principals on the other hand can be achieved by using constructors that allows formation of groups and delegation. Principles can be briefly explained below [7].

- Users and machines.
- Channels, for example, input devices or cryptographic channels.
- Conjunctions of principals, which is expressed with the form $A \wedge B$. When A and B make the same statement s, it implies that $A \wedge B$ says s as well.
- Groups, this defines a group of principals. Principal P_i is a member of group G_i is expressed as $P_i \Longrightarrow G_i$. A disjunction $A \vee B$ denotes a group with only principals A and B.
- Principals in roles, expressed as A *as* R. A principal A may take the role R effectively diminishing their power and act under the name "A *as* R".
- Principals speaking on behalf of other principals, expressed as A *for* B. Here, principal A may delegate authority to B and then B can act on behalf of A using the identity B *for* A.
- Principal speaking for other principals, expressed as $A \circ B$. This denotes that B speaks on behalf of A. It does not necessarily mean that this *speaking for* is with the proof that A actually delegated any authority to B.

To determine that requests from principals are granted or denied, Abadi *et al.* [7] uses a modal logic which extends the algebra of principals. This serves as a basis for different algorithms and protocols [17]. A request on objects is granted if the request is authorized in accordance with

the authorizations present in the ACL of the object. The implication relationships and delegations among principals is also considered while processing the request.

2.4.2 SPKI/SDSI

Rivest *et al.* [49] introduces the Simple Distributed Security Infrastructure (SDSI) as a system to manage distributed name spaces. In this system, name spaces are defined and structured locally but can be referenced non-locally [49, 17]. Access rights are associated with a local name and principals bound to that name by SDSI name certificates can gain access. The Simple Public Key Infrastructure (SPKI) on the other hand, provides much more complex authorization policies without the need to manage identities. SPKI manages to extend upon SDSI to allow binding a capability to a name or a key [49]. A principal can explicitly delegate a subset of their rights to another principal or a name representing a collection of principals in SPKI/SDSI [49].

In SDSI the public keys are considered the status of principals [17]. There is no connection made to associate public keys with people, machines or other entities. It is assumed that the private keys are secure and statements signed by such private keys confirms that the statements are made by that principal. Therefore, SDSI does not handle the signature checking part of the certificate and it is done before authorization. Each principal in SDSI need to define a local name space local to that principal by issuing *name certificates* [17]. These certificates binds names to principals. An SDSI name in the form of $K\ A$ is read as "K's A", that is, A is a local name in K's name space. A name certificate is a 4-tuple (K, A, S, V) where [17],

- K is the principal issuing the certificate.
- A is the name in K's name space.
- S is the subject of the certificate.
- V is the certificate validity information.

The subject S can be other names or principals. The certificate validity field V has the expiration information time or information where to locate and obtain revocation or revalidation information. An authorizer can check revocation of a certificate using these information. The SDSI system ignores the revoked certificates [17].

In SDSI $K\ A \rightarrow S$ means that the A is a name in K's name space. A is defined to be a local name for the subject S. SDSI also allows certificate subjects to refer to nonlocal names through the form $K_1\ A_1 \rightarrow K_2\ A_2$. This denotes that all names bound to $K_2\ A_2$ are also bound to $K_1\ A_1$ [17].

SPKI extends the SDSI framework by allowing delegation from principal to principal and introduces *authorization certificates* in order to do so. This certificate is a 5-tuple (K, S, D, T, V) where,

- K is the principal issuing the certificate.
- S is the subject of the certificate.
- D is the boolean delegation flag.
- T is the authorization tag.
- V is the certificate validity information.

SDSI and SPKI share the same fields for K, S and V. T field is an authorization tag formatted as an S-expression with specific structural requirements. The meaning of the tag is application specific and left undefined. The field D is a boolean delegation flag which allows subjects to delegate authorization through the creation of new authorization certificates. SPKI however does not allow principals to specify a delegation depth. In general, principals cannot know how many levels of delegation an authorization may need. It is inherently difficult to specify an appropriate depth [17].

Any keyholder who are empowered to grant or delegate the authorization can generate authorization certificates [5]. A real life implementation requires that SPKI certificates should be

usable in a very constrained environment. The code to process the certificates should be as small as possible. The SPKI certificate should be very simple and should have the least amount of fields necessary in the optional fields. It is important that no library code is needed to parse SPKI certificates [5].

2.4.3 Role-based Trust management

RT trust management is a framework which is a collection of trust management systems with different levels of expressiveness and complexity [38]. Li *et al.* [38] presents RT_0 as the base trust management system which is similar to SDSI [17]; the difference being that the extended names in RT_0 is limited to only one level of indirection and provides intersection of roles [38, 17]. RT_1 extends RT_0 by introducing parameterized roles. RT^D adds a mechanism to introduce delegation of rights and role activations and RT^T includes support for threshold and separation of duty policies [38]. RT^T and RT^D can be combined with RT_0 and RT_1 to create systems such as RT_0^T, RT_1^{TD}, RT_1^D and so on and so forth.

The RT framework refers to principals as entities and represents them as public keys. The RT framework grants each entities the power to define their own role in a name space which is local to that entity [38]. To access a resource a requester needs to prove membership in the role which is allowed that permission. Entities can issue credentials that can specify the membership of the roles. These credentials can be a part of the private policies or can be signed by the issuer and made public [38].

Let A, B, C be entities and let r, s, t be role names. If the role r is local to the entity A, it is denoted by $A.r$. RT_0 credentials take the form $A.r \longleftarrow f$, where f can be the following [17].

- $A.r \longleftarrow E$: Entity E is a member of role $A.r$.
- $A.r \longleftarrow B.s$: All members of role $B.s$ are also members of role $A.r$. This can be used to delegate authority.

- $A.r \longleftarrow B.s.t$: For each member E belonging to $B.s$, all members of role $E.t$ are members of role $A.r$. Credentials like this can be used to delegate membership authority to all entities who have the attribute $B.s$.

- $A.r \longleftarrow f_1 \cap \cdots \cap f_n$: Each entity who is a member of all roles $f_1 \cap \cdots \cap f_n$ is a member of the role $A.r$.

RT_1 improves on RT_0 by allowing roles to be parameterized [38]. RT_1^C on the other hand improves the expressive power of RT_1 by allowing structured constrains to be applied to the role parameters [38]. RT^T interprets roles as sets of sets of entities called *principal sets*. RT^T allows threshold policies and separation of duty policies to be written. RT^D introduces concepts of role activations and delegations to RT_0 using the credential of the form $A \xleftarrow{C \text{ as } D.x} B$ where A delegates the *role activation* of C *as* $D.r$. B can then access whatever C can access from role $D.r$. Li *et al.* [39] describes an algorithm to find a path in a construct called a credential graph. That algorithm is described as distributed credential chain discovery and it assumes that the issuer or the subject of a credential can be contacted in some way on queried about the credential. The objective of the algorithm is to find the missing credentials to complete a proof of authorization. This algorithm can work backwards, forwards or bidirectional [39]. To work backwards, the algorithm starts at the governing role, following the credentials to issuer to subject. The forwards working version works from the entity who is the requester and follows credentials from subject to issuer [39].

2.5 Recent Works on Context in IoT Security

Li *et al.* [53] presented a capability-based system that supports "permission sequence" and "context" which allows for the enforcement of finite sequence of permissions. The work by Li *et al.* [53] is a continuation of the work by Tandon *et al.* [54]. Each sequence has their own specific context. Li *et al.* integrated the system into OAuth 2.0 [53]. This work also allows efficient capability revocation as well.

Studies have also been done and solutions proposed for context-aware middleware in smart homes [27, 56, 32]. Guo *et al.* [27] put forward a framework for context-management in IoT security. [27] focuses on the mobile entity problem in cross-domain context sharing. The framework also proposed a transparent query mechanism which enables entities to obtain context information from remote locations and the contexts generated are managed by a home context manager [27]. Vahdat *et al.* [56] put forward a context-aware middleware that simplifies and makes it easier to share context information between devices in a smart home. This method has mechanisms in place for smooth publishing of context information and also retrieval of context information [56]. This is important because in every and all context-aware systems for IoT, it is expected that there will be a large number of entities providing context. [56] attempts to solve the issue of gathering a set number of context information from a large complicated system. Hyun *et al.* [32] attempts similar targets of acquiring, processing, reasoning and disseminating context information by designing and proposing their own middleware architecture. Hyun *et al.* models context based on ontology using Web Ontology Language (OWL). [32] designed an improved rule-based reasoning algorithm to allow the middleware to infer high-level context from the available low-level contexts. Their experimental results show that their middleware provides more accurate and faster reasoning outcome compared to traditional rule-based reasoning methods.

Alagar *et al.* [9] put forward a comprehensive security and privacy architecture for Healthcare Internet of Things (HIoT). Alagar *et al.* [9] proposes Context-Sensitive Role-Based Access Control Scheme (CRBAC) which combines roles, attributes and context together to enforce access and flow control. CRBAC also proposes some innovations to medical devices in order to make them more resilient against attacks from adversaries. Work on context-sensitive access control was done in [58, 37]. Kumar *et al.* [37] presents Context-Sensitive Role-Based Access Control (CS-RBAC) which extends the Role-Based Access Control (RBAC) model by adding a level of context sensitivity to it. In their work the notions of role context and context filters are introduced to RBAC. In [58] Wan *et al.* proposed a framework which introduces context-awareness for IoTs such that

context information can be gathered *at the right time, for the right price* and *at the right location*. They also discuss context and context-awareness in deep details with respect to, and how they can be integrated with IoT descriptions.

Da *et al.* discusses different notions of context-awareness and their types [21]. They discuss types of context-awareness indicator that are applied to Edge selection algorithms, their roles and scope and approaches currently used. Da *et al.* comes to the conclusion that context-awareness in IoT security is indeed acknowledged in the community but its implementation and integration to support more dynamic IoT environments is limited. Context-awareness in IoT is mostly limited to location information to assist traffic locality and node resources [21]. They also present several guidelines on how future research should be conducted to improve context-awareness by considering different levels of context-aware indicators derived from behavior of the users (e.g., roaming patterns; preferred network to connect to).

2.6 Implementation Technologies Used

This section briefly explains some technologies used in the implementation process which is explained in detail in Chapter 5.

2.6.1 Representational State Transfer (REST)

Representational State Transfer or REST is a well-formed and universally accepted standard of architecture for web services [44]. These web services are built in a way that that provides support to a website. Client programs have access to application programming interfaces (APIs), which provide a means to exchange information between programs. In the implementation of this work, RESTful APIs were used where resources are accessed via some standard methods provided in the outline for the REST architectural design.

Some of the standard methods that are made available in the HTTP and CoAP are described below.

1. **GET**

   This provides access to a resource where only read is enabled.

2. **POST**

   This allows creation of new resources or to update a current resource.

3. **PUT**

   This allows creation of a new resource.

4. **DELETE**

   This allows removal of a resource.

REST allows us to represent each resource using a Uniform Resource Identifier (URI) in Context-Aware HCAP. We have talked about resources guarded by resource servers. For a client to access these resources, they have to send a request to a URI. For example, if a client wants access to *"resource-3"*, the client has to send a request to a URI for example: `coaps://127.0.0.1:8080/hcap/res-3`.

2.6.2 Constrained Application Protocol (CoAP)

We all know Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) as the two most prominent internet protocol suits in use today. Constrained Application Protocol or CoAP is such a protocol belonging to the set of internet protocol suits.

CoAP is an internet protocol which was specially designed to work with devices with very limited capabilities and to work with constrained devices and limited network capabilities [1]. This is a perfect tool to be used with small IoT devices which may be placed in places with low network coverage. CoAP has been designed as a request and response model between two nodes in a network. In HCAP, two nodes such as clients and servers interact with one another. The clients send CoAP requests and the servers process that request using algorithms in place and respond to

the clients appropriately. It is designed to resemble HTTP so that it is easily understandable for someone with HTTP knowledge, and can be integrated with the web easily.

CoAP has been standardized for computationally constrained devices running on a low-power source and limited network connections defined in RFC 7252 [1]. Communicating with one another needs to be safe, secure and needs to be done using very small packets. TCP is very well defined for safety and security but the packets are very complicated to build and also have a very complex structure, therefore, CoAP uses UDP and is compatible with devices that can support UDP.

The structure and communication protocols of CoAP is very similar to a client-server model as seen in HTTP. Although every communication requires parties to assume a role between a client and a server. As mentioned before, entities communicating uses a CoAP request on resources available by a second party based on the RESTful model explained in Section 2.6.1. The entities assuming a server role makes resources available to be requested using a Uniform Resource Identifier (URI). The entity making the request uses methods similar to HTTP, such as PUT, POST, GET and DELETE. CoAP can also be interchanged with data formats such as JSON, CBOR and XML, which makes it very compatible and flexible.

I have used the Java Implementation of CoAP named Californium in the context-aware HCAP which implements the RFC 7525 version. The implementation is described in details in chapter 5.

2.6.3 X.509 Certificates

X.509 certificates is a standard, universally accepted, well-defined format for public keys. In cryptography, in order to encrypt messages across the internet, we need two mathematically related keys: public keys and private keys. Public keys are used to encrypt messages and private keys are used to decrypt messages. As the names suggest, public keys are open to the public and shared with everyone and the private keys are not shared. A public key infrastructure was put into place to address the issue of sharing public keys in a secure manner [47]. Using this public-key infrastructure, public keys are distributed to clients in the form of digital certificates. These digital

certificates hold the public key of a subject.

As a part of the Public-Key Infrastructure, Certification Authorities(CAs) were put in place, which acts as a trusted repository of public keys for subjects. If a CA is trusted by two participants or subjects, they can ask the CA for the others public keys. The CA provides them with digitally signed certificates, signed by the CA. The subjects using a Public-Key Infrastructure should be aware of the public key of the CAs. Because when one subject sends his certificate, the other party needs to verify the authenticity of the certificate. This verification is done by using the public key of the CA.

X.509 is a universally accepted format for a digital certificate used in many well known internet protocols like TLS and DTLS [47]. An X.509 certificate contains the following information [47].

1. **Version**

   An X.509 certificate has 3 versions pointing to different formats and the information it holds. Version 1 has code 0, version 2 has code 1 and version 3 has code 2.

2. **Serial Number**

   This is an integer which includes the name of the issuing CA. This uniquely identifies the certificate. No two certificates can have the same Serial Number.

3. **Signature Algorithm ID**

   This mentions the algorithm the CA used to sign the certificate. For example, SHA256, DSA, AES, sha256RSA etc.

4. **Issuer Distinguished Name**

   This is the unique name of the certification authority issuing the certificate.

5. **Validity Period**

   Certificates usually expire a certain period of time. This field denotes that life span before it expires. It is denoted by number of days.

6. **Subject Distinguished Name**

   This field contains the name of the subject that the certificate is issued to.

7. **Subject Public Key**

   This field holds the public key associated with the subject distinguished name.

8. **Extension**

   This is an optional field may contain any information the CA wants to store in the certificate. It is only applicable when using version 3 certificates.

The HCAP implementation uses X.509 certificates where the clients and the servers mutually authenticate each other using these certificates. After authenticating them, the entities can start an encrypted communication channel with one another.

2.6.4 Java Cryptography Architecture (JCA)

Java has been used as the base programming language in our implementation. We have used Java Cryptographic Architecture (JCA) to encrypt messages to and from the several entities interacting with one another. The JCA is a built in framework used to make use of several cryptographic algorithms and was introduced in with the Java Development Kit (JDK) version 1.1 as a part of the java security API in the `java.security` package.

JCA is a very vast platform containing APIs that can be used for hashing, certificate creation and validation, symmetric and asymmetric encryption algorithms, key generation and management, secure random number generation and many more. These APIs introduce security to be used with Java, and is easily integrable to an application with Java [3]. The JCA was designed with complementary algorithms, i.e., cryptographic services such as digital signatures and generating random numbers are readily available to be used without the worry of any implementation details.

Java Cryptographic Extension (JCE) used to be a separate component besides the JCA before JDK version 1.4. Presently JCE is bundled with JDK and is considered to be a part of the

JCA. The JCA framework consists of packages such as `java.security`, `javax.crypto`, `java.security.spec`, `java.crypto.spec`, etc. The `Signature` class from package named `java.security` was used in the implementation. A brief of the `Signature` class is described in the following.

The Signature class

The Signature class is an engine class which provides several cryptographic algorithm implementations such as DSA, RSA or SHA256withRSA. A user can choose an algorithm from a whole range of algorithms provided by the class. Using a cryptographically secure algorithm from this class we can sign an arbitrary-length string with a private key [2]. The resulting string of bytes is often of fixed length and is called *signature* [3].

As the signature is created by the private key, only the owner of the private/public key pair is able to create the signature. Using the signature and the public key, it is possible to find the authenticity of the input string. After initializing the `Signature` object, the `update` method is used to provide input to the algorithm. After that, the methods, `sign` and `verify` may be used to sign and verify the input string respectively. The algorithm name must be sent as a parameter while initializing the `Signature` object while signing. There are a wide range of algorithms that can be used and they are standardized in the JCA Standard Algorithm Name Documentation in [4].

In the next Chapter we will present Context-Aware History-Based Capability system in details.

[2] A private key and its counterpart, the public key can also be generated by a Key Pair generator also provided by the JCA framework.

Chapter 3

Context-Aware History-Based Capability System

This chapter presents the Context-Aware History-Based Capability system. It explains the syntax, semantics and the mathematical background of the context-aware history-based capability system. Firstly, I introduce "context" in distributed authorization scheme, then I go over the syntax and semantics of an automaton I designed for the purpose of representing "context" and how it is used for authorization decisions (Section 3.2). Secondly, I present a security principle and mathematically prove that the context-aware history-based capability system follows this security principle (Section 3.3). Thirdly, I present a conversion mechanism to convert my automaton to a deterministic variant (Section 3.4) and prove the determinism of the converted automaton and also show the equivalence of the converted automaton to the original automaton (Section 3.5). The chapter ends with a brief discussion on the space complexity of the converted automaton.

3.1 Introduction

In a distributed authorization scheme, there is no central entity which makes access control decisions. Entities move about performing operations on objects at random. Schneider [52] addressed the task of enforcing security policies by monitoring execution steps. This mechanism of enforcing security policies is known as Execution Monitoring (EM). Schneider introduced a Büchi-like automaton, which recognizes safety properties, and he named this automata model, **security automata**. Schneider's "security automata" is similar to a non-deterministic finite automata [29] and can be defined as a quadruple (Q, Q_0, I, δ) where,

- Q is a set of automaton states,
- $Q_0 \subseteq Q$ is a set of initial automaton states,

- I is a set of input symbols, and
- $\delta : (Q \times I \rightarrow 2^Q)$ is a transition function.

The set I models the set of actions that are guarded by the EM. In order to process a sequence $e_1 \cdot e_2 \cdots e_n$ of input symbols in I, let the current set of states be Q' which is equal to the set of initial automaton states Q_0 before any input is read. The security automaton updates Q' as each input e_i is read. With each update, the current set of states Q' changes to a new current set of states Q'' where $Q'' = \bigcup_{q \in Q'} \delta(q, e_i)$. As the inputs are read, if at any point, Q'' is an empty set, then the input sequence is rejected.

Significance of an SA State. According to Hopcroft *et. al.* [29] an on/off switch is an example of a nontrivial finite automata. This switch "remembers" if it is the 'on' state or the 'off' state. A user interacting with the switch grants a state change, meaning that a state also defines what is possible.

Consider Example 1 on Page 2, Alice has to go through doors A, B and C in that specific order. This can be represented in the following SA.

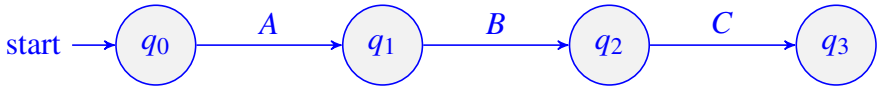

Figure 3.1: **An SA Representing Example 1 in Page 2.**

Before Alice has to go out of her lab door A, we say she is in state q_0. This state represents what is possible in the future: i.e., she is allowed to go through door A and once she has done that she can access door B. Her state changes to q_1 after going through door A. States also signifies what Alice has done in the past i.e., her history of access. Thus state q_1 signifies that Alice has gone through door A. So state q_1 signifies both the history of access for Alice (door A) and what she can do in the future (door B and then door C). Herein lies the significance of states in an SA.

Example 2 on Page 2, is an example of process awareness where accesses need to comply with a workflow specification where there are two steps S_1 and S_2 (S_1 to be done before step S_2).

Permissions p_1 and p_2 is allowed in S_1 and permissions p_3 and p_4 are allowed in S_2. The following figure presents an SA which enforces this sequencing constraint.

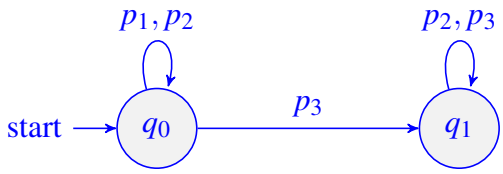

Figure 3.2: **An SA Representing Example 2 in Page 2.**

Each state signifies two things: 1. what is possible, and 2. what has been accessed in the past. In Figure 3.2, state q_0 signifies that permissions p_1, p_2 and p_3 are possible. State q_1 signifies that only permissions p_2 and p_3 are possible and that p_3 has been exercised in the past. This guarantees that the steps S_1 and S_2 conforms to the ordering of the steps and the sequencing contraints.

3.2 Context-Aware Security Automaton

This section introduces a language I have used to formally explain the notion of context-awareness in HCAP. I derive from the idea of Schneider [52] for enforcing security principles on a sequence of events using the notion of Security Automata (SA). Security Automata are structurally close to a non-deterministic finite automata as mentioned before.

To introduce context to HCAP, I define Context-Aware Security Automata (CASA) as an extension to the Security Automata (SA). A CASA not only identifies a **permission** for state transition, but also requires a **set of conditions** to be true at the time of requesting execution of the said permission. Entities prove conditions to be true by the presentation of certificates. Certificates are assertions signed by a "trusted entity" who has the ability to certify that some conditions are met. For example, "The temperature is greater than 20 degrees" is a condition that can be certified to be true by a thermometer. Conditions are signed by a trusted entity if they are true. These signed conditions are called certificates. Entities who are trusted to sign assertions are named Sensor Integration Centre (SIC). SICs are servers and are considered to have access to several sensors.

Therefore, these SICs can understand the state of the environment. In this way, SICs possess information about the environment at a specific time in the physical world. In other words, an SIC can look at several sensor data and can decide if the SIC should provide a signed condition to a requesting entity. The formal definition of a CASA is as follows.

Definition 3.2.1. A ***Context-Aware Security Automaton (CASA)*** can be represented as a 5-tuple $(\Sigma, Q, \Gamma, q_0, \delta)$ where,

- Σ is a finite set of permissions
- Q is a finite set of states
- Γ is a finite set of conditions
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \times 2^{\Gamma} \nrightarrow Q$ is a *partial transition function* mapping a state, a permission and a set of conditions to a state, i.e., $\delta(q, p, C)$ maybe undefined for some triples of $q \in Q$, permission $p \in \Sigma$ and set of conditions $C \subseteq \Gamma$.

Following the explanation of δ, it is to be noted that, the third argument in the mapping can be an empty set and would still yield a state. For example, $\delta(q, p, \emptyset) = q$ for some $q \in Q$ and $p \in \Sigma$. In the definition, a permission $p \in \Sigma$, is an operation performed on a resource, enabled by an entity named as Resource Server, as we shall observe in a later chapter. A resource could be any IoT device, such as a smart door, a smart TV or a smart thermostat, and operations on those resources could be "open", "off" or "set" respectively. Therefore, a permission is a pair consisting of an operation and a resource. An operation performed on a resource is hence represented by a permission. A condition $c \in \Gamma$ follows the discussion from a previous paragraph where, a condition is a requirement that needs to be true at the time of requesting to execute a permission. We deal with sets of conditions $C \subseteq \Gamma$ and require them to be true at the time of requesting to execute a permission.

Definition 3.2.2. The ***"behavior" of a CASA*** can be captured by a ternary relation $\cdot \dot{\rightarrow} \cdot \subseteq Q \times (\Sigma \times 2^{\Gamma}) \times Q$. This is a single-step transition relation. This means, we write, $q_1 \xrightarrow{(p,C)} q_2$ iff $q_2 = \delta(q_1, p, C')$ for some $C' \subseteq C$, where $q_1, q_2 \in Q$, $p \in \Sigma$ and $C, C' \subseteq \Gamma$.

We can also extend this ternary relation to $\cdot \dot{\rightarrow} \cdot \subseteq Q \times (\Sigma \times 2^{\Gamma})^* \times Q$. So, we can write $q \xrightarrow{\varepsilon} q$ where $q \in Q$ and ε is the empty string. This means that with an empty input sequence ε the CASA remains in the same state. And we can also write $q_1 \xrightarrow{\Theta_1 \cdot \Theta_2} q_3$ iff $\exists q_2 \in Q \,.\, q_1 \xrightarrow{\Theta_1} q_2$ and $q_2 \xrightarrow{\Theta_2} q_3$, where $\Theta_1, \Theta_2 \in (\Sigma \times 2^{\Gamma})^*$.

Definition 3.2.3. An **execution trace** is a finite sequence of pairs denoted by $\Theta \in (\Sigma \times 2^{\Gamma})^*$. Each pair takes the form (p, C) where $p \in \Sigma$, and $C \subseteq \Gamma$. A **single pair** as an execution trace is named a **unit trace**, denoted by $\tau \in (\Sigma \times 2^{\Gamma})$. A CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$ can process an execution trace $\Theta = \tau_1 \cdot \tau_2 \cdots \tau_n$. I use "traces" and "execution traces" interchangeably throughout this thesis.

Comparison of traces. To compare two execution traces, both the execution traces **must** be of equal length. Let Θ and Θ' be two traces of **equal** length. Let $\Theta = (p_1, C_1) \cdot (p_2, C_2) \cdots (p_n, C_n)$ and $\Theta' = (p_1, C_1') \cdot (p_2, C_2') \cdots (p_n, C_n')$ be two traces of length n and matching permissions. We write $\Theta \sqsubseteq \Theta'$ if and only if $C_1 \subseteq C_1', C_2 \subseteq C_2', \ldots, C_n \subseteq C_n'$ and the permissions $p_1, p_2, \ldots, p_n$ are the same in both Θ and Θ'. That means, for each corresponding unit trace in Θ and Θ', the set of conditions on the left side of $\sqsubseteq$ must be a subset of the set of conditions on the right side of $\sqsubseteq$ and the permissions must be the same. To be noted here once more, that execution traces of unequal lengths are not comparable. Whenever $\Theta \sqsubseteq \Theta'$, we say that Θ' **dominates** Θ, or Θ is **dominated** by Θ'. Intuitively, $\Theta \sqsubseteq \Theta'$ means that Θ' provides at least as much information as Θ.

Acceptance of a trace. Not all execution traces can be accepted by CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$. M accepts an execution trace Θ if there exists a Θ' such that $\Theta' \sqsubseteq \Theta$ and $q_0 \xrightarrow{\Theta'} q$, where $q \in Q$. If there exists no such trace Θ', then the execution trace Θ is rejected. To be noted here, there could be multiple such traces Θ' such that $\Theta' \sqsubseteq \Theta$. This is due to the non-deterministic nature of a CASA. The following example demonstrates acceptance or rejection of traces and how non-determinism leads to multiple execution traces.

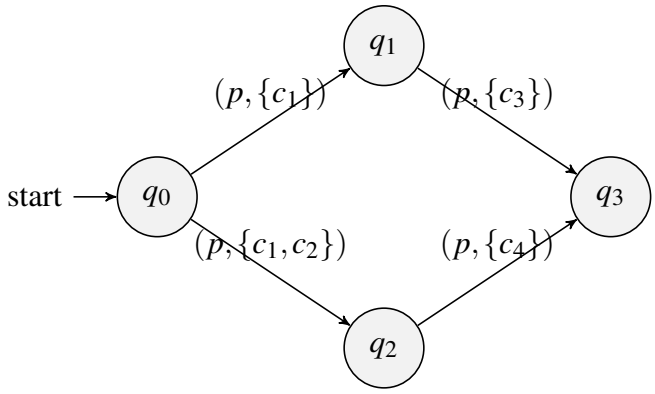

Figure 3.3: **A representation of CASA M.**

Let $M = (\Sigma, Q, \Gamma, q_0, \delta)$ be a CASA and $\Sigma = \{p\}$, $\Gamma = \{c_1, c_2, c_3, c_4\}$, $Q = \{q_0, q_1, q_2, q_3\}$, and $\delta(q_0, p, \{c_1\}) = q_1$, $\delta(q_0, p, \{c_1, c_2\}) = q_2$, $\delta(q_1, p, \{c_3\}) = q_3$ and $\delta(q_2, p, \{c_4\}) = q_3$. Figure 3.3 depicts CASA M.

CASA M accepts execution traces $\Theta_1 = (p, \{c_1\}) \cdot (p, \{c_3\})$ and $\Theta_2 = (p, \{c_1, c_2\}) \cdot (p, \{c_4\})$. Let an execution trace be $\Theta_4 = (p, \{c_3\}) \cdot (p, \{c_4\})$. We can infer that $\Theta_1, \Theta_2 \not\sqsubseteq \Theta_4$. So we can say that execution trace Θ_4 is rejected by M. On the other hand, let another execution trace be $\Theta_3 = (p, \{c_1, c_2\}) \cdot (p, \{c_1, c_3, c_4\})$. We can see that both $\Theta_1 \sqsubseteq \Theta_3$ and $\Theta_2 \sqsubseteq \Theta_3$, therefore, Θ_3 is an acceptable execution trace by CASA M. To be noted here, that the last unit trace of Θ_3 contains condition c_1, which is redundant, but is permitted as per the definition of domination ($\sqsubseteq$) in the previous section. In this example, there are multiple execution traces which are dominated by Θ_3, which means that there can be multiple execution paths that can be taken. This is a result of the non-deterministic nature of CASAs.

Definition 3.2.4. An *execution trace-output* Δ_M for a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$ is defined to be the set of all states that is reachable from a given state $q \in Q$ using an execution trace Θ. Formally we write the following.

$$\Delta_M(q, \Theta) = \{q' \in Q \mid q \xrightarrow{\Theta} q'\}$$

For the CASA represented by Figure 3.3, if $\Theta = (p, \{c_1, c_2\})$, then $\Delta_M(q_0, \Theta) = \{q_1, q_2\}$. If $\Theta = (p, \{c_1, c_2\}) \cdot (p, \{c_4\})$, then $\Delta_M(q_0, \Theta) = \{q_3\}$ and so on and so forth.

In case $\Theta = \varepsilon$, the execution trace-output would yield the given state q as no transition is involved. So, $\Delta_M(q, \Theta) = \{q\}$.

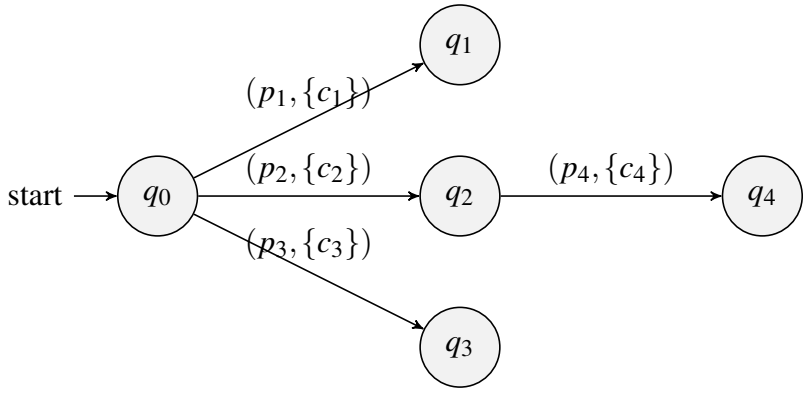

Figure 3.4: **Traces from a state example.**

Definition 3.2.5. An *execution trace-set from a state* $q \in Q$ is defined as the set of all execution traces $\Theta \in (\Sigma \times 2^{\Gamma})^*$ for which $q \xrightarrow{\Theta} q'$ for some $q' \in Q$. We write $Tr(q)$ to be the set of all execution traces from state q. Formally we can write the following.

$$Tr(q) = \{\Theta \in (\Sigma \times 2^{\Gamma})^* \mid \exists q' \in Q \,.\, q \xrightarrow{\Theta} q'\}$$

For example, consider the CASA in Figure 3.4. Here, CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, if $\Sigma = \{p_1, p_2, p_3\}, \Gamma = \{c_1, c_2, c_3\}, Q = \{q_0, q_1, q_2, q_3\}, \Theta_1 = (p_1, \{c_1\}), \Theta_2 = (p_2, \{c_2\}), \Theta_3 = (p_3, \{c_3\})$ and $\Theta_4 = (p_2, \{c_2\}) \cdot (p_4, \{c_4\})$, then $\Delta_M(q_0, \Theta_1) = \{q_1\}$, $\Delta_M(q_0, \Theta_2) = \{q_2\}$, $\Delta_M(q_0, \Theta_3) = \{q_3\}$, $\Delta_M(q_2, \Theta_4) = \{q_4\}$, and the execution trace-set from q_0 will be $Tr(q_0) = \{\varepsilon, \Theta_1, \Theta_2, \Theta_3, \Theta_4\}$. In this example $\Theta_1, \Theta_2, \Theta_3$ and Θ_4 are all acceptable execution traces by CASA M. The cardinality of the set $Tr(q)$ can be countably infinite.

I would like to extend the definition of the execution trace-set to an length-n execution trace-set from state q, which take the form $Tr_n(q)$. The length-n execution trace-set from a state q is the set of all execution traces $\Theta \in (\Sigma \times 2^{\Gamma})^n$ of length $n \in \mathbb{N}$. Formally we can write the following.

$$Tr_n(q) = \{\Theta \in (\Sigma \times 2^{\Gamma})^n \mid \exists q' \in Q \,.\, q \xrightarrow{\Theta} q'\}$$

Let us consider the previous example (Figure 3.4). We can say a length-1 execution trace-set from state q_0 is $Tr_1(q_0) = \{\Theta_1, \Theta_2, \Theta_3\}$ and a length-2 execution trace-set from state q_0 is $Tr_2(q_0) = \{\Theta_4\}$. A length-0 execution trace-set from state q_0 is $Tr_0(q_0) = \{\varepsilon\}$.

We have talked about the acceptance of execution traces before, and I would like to point out that given a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, an execution trace Θ is only accepted by CASA M, if and only if $\Theta \in Tr(q_0)$. For CASA M, the set of all acceptable execution traces is the set $Tr(q_0)$.

It is to be observed here that all the traces that are valid from the start state is accepted by a CASA. Therefore, **the language accepted by a CASA** M is the set of all execution traces from the start state of M. The language accepted by a CASA is denoted $L(M)$ which is defined to be $Tr(q_0)$.

3.3 Security Property

We have defined execution traces in Definition 3.2.3. There are many ways to define the notion of acceptance or rejection of a trace. The reason why the acceptance of a trace is defined in the specific way of Definition 3.2.3 is to prevent a certain form of attack by a malicious entity. This section examines and explains this form of attack and articulates a security property that states the conditions under which this form of attack can be prevented.

We know that clients go about exercising permissions by providing certificates of conditions. A client trying to execute permission p and providing certificates for condition set C can indeed execute the permission if the unit trace (p, C) is accepted from the current state the client is in. In a distributed authorization system such as this, the certificates for these conditions might be provided by a plethora of entities ranging from an entity certifying a light is on to certifying a traffic light is red on a specific street. The client therefore needs to gather and present all these evidence in order to exercise permission p.

Naturally we must ask ourselves what would happen if a client presents 3 certificates and is denied access, but then trying to manipulate the system, throws away 1 certificate and gets access by only presenting 2 certificates instead of 3. An adversary may intentionally withhold certificates;

because by telling half-truths, the adversary gains access. This is the intuitive explanation of an adversarial attack by withholding information.

Tying this intuition to the concept of traces, we can say that if Θ and Θ' are two execution traces such that $\Theta \sqsubseteq \Theta'$, and a system accepts Θ, then Θ' should also be accepted. We call such distributed authorization schemes "***tamper-proof against information withholding***". In that way, when Θ' is denied access, Θ will be denied as well. Consequently, withholding information never gives an attacker any advantage.

Security Property. Given two traces Θ_1, Θ_2 such that $\Theta_1 \sqsubseteq \Theta_2$, the system which accepts Θ_1, also accepts Θ_2 is said to be tamper-proof against information withholding.

The contrapositive of this security property states that if a trace Θ_1 is not accepted, then any trace Θ_2 less informed than Θ_1 is also not accepted. That means if an adversary fails to gain access by presenting a certain set of proof of conditions, then withholding such conditions will not give her any advantage. This is a direct consequence of introducing monotonicity in the definition of trace acceptance. This is why the acceptance of execution traces are defined in this certain way in Section 3.2.3.

Theorem 3.3.1. *A CASA is tamper-proof against information withholding.*

Proof. We want to demonstrate that, given a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, for all traces Θ, Θ' so that $\Theta \sqsubseteq \Theta'$, if M accepts Θ, then M accepts Θ'. We would like to strengthen the claim and prove a stronger statement: "Given a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, for all traces Θ, Θ' so that $\Theta \sqsubseteq \Theta'$, for every $q, q' \in Q$, if $q \xrightarrow{\Theta} q'$, then $q \xrightarrow{\Theta'} q'$. "

We prove this stronger claim by induction on the length of traces Θ and Θ'.

Base Case. If the trace is empty, that is, if $\Theta = \Theta' = \varepsilon$ or $n = 0$, and $q \xrightarrow{\Theta} q$ then $q \xrightarrow{\Theta'} q$. This means that Θ and Θ' are both accepted by CASA M from state q. So, for $n = 0$, the claim holds.

Induction Hypothesis. For length-n traces Θ_n, Θ'_n so that $\Theta_n \sqsubseteq \Theta'_n$, for every $q, q' \in Q$, if $q \xrightarrow{\Theta_n} q'$, then $q \xrightarrow{\Theta'_n} q'$.

Inductive Step. Consider two length-$(n+1)$ traces $\Theta_{n+1}, \Theta'_{n+1}$ where $\Theta_{n+1} \sqsubseteq \Theta'_{n+1}$. Now

Θ_{n+1} must be of the form $\Theta_n \cdot (p, C)$ where Θ_n is an length-n trace. Since $\Theta_{n+1} \sqsubseteq \Theta'_{n+1}$, the trace Θ'_{n+1} must be of the form $\Theta'_n \cdot (p, C')$, where $C \subseteq C'$ and $\Theta_n \sqsubseteq \Theta'_n$.

Suppose $q \xrightarrow{\Theta_{n+1}} q'$ for some $q, q' \in Q$, then for some $q'' \in Q$, we can say $q \xrightarrow{\Theta_n} q'' \xrightarrow{(p,C)} q'$. By the induction hypothesis, we know the following.

$$q \xrightarrow{\Theta'_n} q'' \tag{3.1}$$

Moreover, since $C \subseteq C'$, it must be the case that the following is true.

$$q'' \xrightarrow{(p,C')} q' \tag{3.2}$$

Therefore, by Equations 3.1 and 3.1, we have $q \xrightarrow{\Theta'_n} q'' \xrightarrow{(p,C')} q'$, or equivalently, $q \xrightarrow{\Theta'_{n+1}} q'$.

Therefore, by the principle of mathematical induction, given a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, for all traces Θ, Θ', such that $\Theta \sqsubseteq \Theta'$, for every $q, q' \in Q$, if $q \xrightarrow{\Theta} q'$, then $q \xrightarrow{\Theta'} q'$.

Since it is true for all $q, q' \in Q$, it is also true for $q_0, q' \in Q$, where q' is any state in the CASA. The language of CASA M accepts all traces contained in $Tr(q_0)$. Since we proved the claim holds for the any two states $q, q' \in Q$ including the start state, we have successfully proven that CASA is tamper-proof against information withholding. □

In the beginning of this section we mentioned that execution traces and their acceptance criteria are defined in a certain way in order to prevent a withholding attack by a malicious party. It is only possible to prevent this attack because of the monotonic definition of acceptance of execution traces. Then we proved that CASA is tamper-proof against information withholding. Therefore, CASA has that security property. In the next section we explore determinism and non-determinism in CASAs and their implications.

3.4 Determinism Versus Non-Determinism

Before we talk about determinism and non-determinism we need to talk about fragments. We have seen fragments designed in [54] in Section 2.1.4. An automaton may have a large number of states in the design of [54] as well as my own design. Therefore, it is more economical to provide a partial automaton to a client, which is then embedded in an authorization token. A client uses this authorization token to gain access. If the full automaton was embedded, it would add to the latency during the communication between a client and an authorizing entity. In a distributed authorization scheme, it is desirable to keep network packets in communications at a minimum. In this work, we are still faced with the same issue and therefore have designed a fragment called CASA fragment which is defined below.

Definition 3.4.1. Given a CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, a fragment F of M is a 5-tuple $(\Sigma, Q', \Gamma, q_0', \delta')$ such that,

- $Q' \subseteq Q$
- $q_0' \in Q'$
- For every $q \in Q'$, $p \in \Sigma$, $C \subseteq \Gamma$, if $\delta(q,p,C)$ is defined and $\delta(q,p,C) \in Q'$, then $\delta'(q,p,C)$ is defined and $\delta'(q,p,C) = \delta(q,p,C)$.

In other words, a fragment F only has a subset of states present in the CASA M. The transition function δ' in the fragment only has transitions among the states in Q'.

A CASA fragment does not contain all the states present in the CASA. It is only a partial specification of a CASA. But a fragment contains all the transitions for every state contained in the fragment.

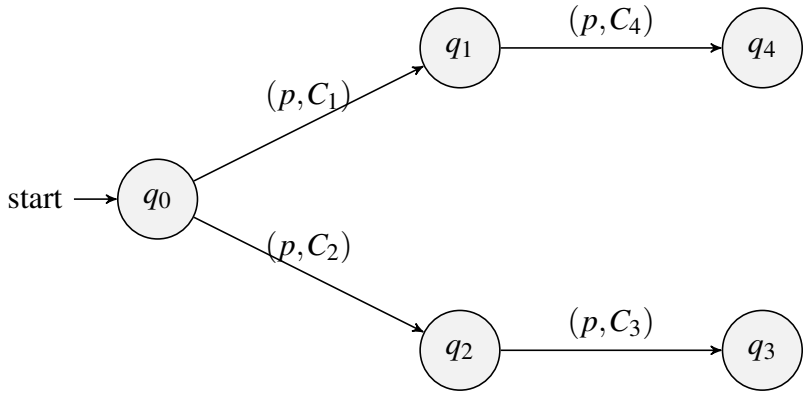

Figure 3.5: **Traces from a state example.**

Local Execution of a Trace. As we briefly mentioned, a CASA is non-deterministic even when the same conditions are provided. If a full CASA is provided with a trace then it is possible to determine if that trace is accepted or rejected. On the other hand, if a fragment of a CASA is provided then the decisions have to be made locally, i.e., with the information present in the fragment. Using only a CASA fragment it is impossible to determine the acceptance or rejection of a trace, because not all the information regarding the CASA is available in the fragment.

This becomes an issue because of the non-deterministic nature of the CASAs. By that we mean, a state q_0 in a CASA may have two transitions $\delta(q_0, p_1, C_1)$ and $\delta(q_0, p_1, C_2)$, which are both defined. Given the two condition sets C_1 and C_2, there are three possible cases: $C_1 \subseteq C_2$, $C_2 \subseteq C_1$, or they are incomparable (i.e., $C_1 \not\subseteq C_2$ and $C_2 \not\subseteq C_1$). For two incomparable sets of conditions C_1 and C_2, if a proof is provided such that it satisfies both C_1 and C_2, then without knowing the full CASA, we cannot take a decision which transition to choose.

Consider a fragment F of the CASA represented in Figure 3.5 where condition set C_1 and C_2 are incomparable, and $C_3 \not\subseteq C_4$. The fragment F houses states q_0, q_1 and q_2 and transitions $\delta(q_0, p, C_1)$ and $\delta(q_0, p, C_2)$. Given trace $\Theta = (p, C_1 \cup C_2) \cdot (p, C_4)$ using the fragment F, without knowing the full CASA, it becomes impossible to choose which transition to make where both $\delta(q_0, p, C_1)$ and $\delta(q_0, p, C_2)$ are valid for the first transition decision. This is because with a limited horizon, if transition $\delta(q_0, p, C_2)$ is taken, then in a future fragment, there would be no transition from q_2 for (p, C_4) and Θ would be rejected. On the other hand, if $\delta(q_0, p, C_1)$ transition was taken,

the trace would be accepted in a future fragment. Therefore, making the local decision to pick transition $\delta(q_0, p, C_2)$ with fragment F, leads to regret in the future even though $\delta(q_0, p, C_2)$ is a valid transition.

CASA by Definition 3.2.1 is non-deterministic, and it is called Non-Deterministic CASA (NCASA). A Deterministic CASA (DCASA) is a special kind of CASA. This next definition outlines the properties that a CASA must have in order to qualify to be a DCASA.

Definition 3.4.2. A CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$ is a **Deterministic Context-Aware Security Automaton (DCASA)** if and only if the CASA has the following properties.

(a) if $\delta(q, p, C_1)$ and $\delta(q, p, C_2)$ are defined, then $\delta(q, p, C_1 \cup C_2)$ is also defined, where $q \in Q$, $p \in \Sigma$ and $C_1, C_2 \subseteq \Gamma$.

(b) if $\delta(q, p, C_1) = q_1$, $\delta(q, p, C_2) = q_2$ and $C_1 \subseteq C_2$ then $Tr(q_1) \subseteq Tr(q_2)$ where $q, q_1, q_2 \in Q$, $p \in \Sigma$ and $C_1, C_2 \subseteq \Gamma$.

Condition (a) requires that given a state and a permission, if there are two valid transitions containing two condition sets, there must be another transition from that state, using the same permission, and demanding the union of the two condition sets. In other words, if $\Theta_1 = (p, C_1)$, $\Theta_2 = (p, C_2)$ and $\Theta_3 = (p, C_1 \cup C_2)$ and if $\Theta_1, \Theta_2 \in Tr(q)$, then $\Theta_3 \in Tr(q)$. Our intention is that, if a client presents a proof of both C_1 and C_2, we take the transition $\delta(q, p, C_1 \cup C_2)$. With this convention, there will be no ambiguity in the decision. A regular CASA cannot guarantee the existence of this third transition, so we felt it necessary to introduce a new kind of CASAs, CASAs which guarantee unambiguity in terms of choice of transitions are DCASAs.

Condition (b) requires that if there are two valid transitions $\delta(q, p, C_1) = q_1$ and $\delta(q, p, C_2) = q_2$ and if $C_1 \subseteq C_2$ then the traces from q_1 is a subset of the traces from q_2, that is, $Tr(q_1) \subseteq Tr(q_2)$. This means, due to the fact that $C_1 \subseteq C_2$, any transition that can be done from q_1, must be possible from q_2 as well. Combined with *Condition (a)*, *Condition (b)* implies that if $\delta(q, p, C_1) = q_1$ and $\delta(q, p, C_2) = q_2$ and a client presents a proof for $C_1 \cup C_2$, the client **must** be allowed all

future transitions from both q_1 and q_2. So, if $\delta(q,p,C_1 \cup C_2) = q_3$, *Condition (b)* implies that $Tr(q_3) \supseteq Tr(q_1) \cup Tr(q_2)$.

If a CASA possesses these two properties for all states, it is a DCASA. As we have seen in a previous paragraph, in a distributed authorization scheme, authorization decisions need to be made with a partial specification of a CASA because it is inefficient to embed the entire CASA in an authorization token. We need authorization tokens because there is no central authorizing entity. Authorization tokens are given to the client and the client presents it to an authorizing entity. So there is a lot of passing back and forth of the token. If the token is of a large size, the latency will be worsened. Therefore, fragments are embedded in an authorization token and transition decisions are to be made with whatever information is available in the fragment. This decision is called a local decision. The problem with local decision and fragments were mentioned before: we do not know which transition should be taken because of non-determinism. But this issue does not occur in a DCASA because we know exactly which transition to make.

Most specific transition. Suppose $q \xrightarrow{(p,C)} q'$. That means, in state q, presenting the conditions in C is sufficient for granting permission p. The DCASA may actually support more than one such transition. To that end, let us define the following for a DCASA.

Definition 3.4.3. A condition set of a state q is defined as the set of all condition sets that support a transition of (p,C) at state q which belongs to a DCASA. Formally, we write the following.

$$Cond(q,p,C) = \{C' \in 2^{\Gamma} \mid \delta(q,p,C') \text{ is defined and } C' \subseteq C, q \in Q, p \in \Sigma\} \tag{3.3}$$

We know that, because of Condition (a) in Definition 3.4.2, this set $Cond(q,p,C)$ is closed under set union: i.e., if $C_1, C_2 \in Cond(q,p,C)$, then $C_1 \cup C_2 \in Cond(q,p,C)$.

Let $C^* = \bigcup Cond(q,p,C)$. Here, C^* is called the *maximal subset* and is a set which is the union of all the sets contained in $Cond(q,p,C)$. As $Cond(q,p,C)$ is closed under set union, C^* is also a member of $Cond(q,p,C)$, meaning $\delta(q,p,C^*)$ is defined. By construction, C^* is the maximum set in $Cond(q,p,C)$, in that sense that every set in $Cond(q,p,C)$ is a subset of C^*. This transition,

$\delta(q,p,C^*)$, where $C^* = \bigcup Cond(q,p,C)$, is known as ***the most specific transition*** for (p,C) at state q.

To illustrate the notion of *the most specific transition*, let us look into the following example. Consider CASA $M = (\Sigma, Q, \Gamma, q_0, \delta)$, where $\Sigma = \{p_1\}$, $\Gamma = \{c_1, c_2, c_3, c_4\}$, $Q = \{q_1, q_2, q_3, q_4\}$ and $\delta(q_1, p_1, \{c_1\})$, $\delta(q_1, p_1, \{c_1, c_2\})$, $\delta(q_1, p_1, \{c_2, c_3\})$ and $\delta(q_1, p_1, \{c_1, c_2, c_3\})$ are defined. Notice that condition (a) of Definition 3.4.2 is satisfied for state q_1. Let's suppose the underlying CASA is in fact a DCASA.

Considering DCASA M, let us look at some different scenarios, given a client is in state q_1 and wants to exercise permission p_1.

Scenario 1. If the client presents proof for condition set $\{c_1\}$, then we have $Cond(q, p, \{c_1\}) = \{\{c_1\}\}$ and so, only the transition $\delta(q_1, p_1, \{c_1\})$ is valid and the DCASA would choose that transition.

Scenario 2. If the client presents proof for condition set $\{c_1, c_2\}$, then we have $Cond(q, p, \{c_1, c_2\}) = \{\{c_1\}, \{c_1, c_2\}\}$. Then transitions $\delta(q_1, p_1, \{c_1\})$ and $\delta(q_1, p_1, \{c_1, c_2\})$ are both valid. But $C^* = \{c_1, c_2\}$, and therefore the most specific transition is $\delta(q_1, p_1, \{c_1, c_2\})$ and the DCASA would choose this transition.

Scenario 3. If the client presents proof for condition set $\{c_1, c_3\}$, then we have $Cond(q, p, \{c_1, c_3\}) = \{\{c_1\}\}$. Then only transition $\delta(q_1, p_1, \{c_1\})$ is valid and the DCASA would choose that transition. Even though the client presented proof for c_3, there is no other valid transition than $\delta(q_1, p_1, \{c_1\})$.

Scenario 4. If the client presents proof for condition set $\{c_1, c_2, c_3, c_4\}$, then we have $Cond(q, p, \{c_1, c_2, c_3, c_4\}) = \{\{c_1\}, \{c_1, c_2\}, \{c_2, c_3\}, \{c_1, c_2, c_3\}\}$. In this case, all the transitions are valid but, $C^* = \{c_1, c_2, c_3\}$ and therefore the most specific transition is $\delta(q_1, p_1, \{c_1, c_2, c_3\})$ and the DCASA would choose this transition.

We would like to argue that, following the definition of a DCASA, when we have multiple valid transitions that can be performed for the same inputs in a given state, picking the *most specific*

transition would always result in a client having access. This means that for all the scenarios, picking one from all the valid transitions would ensure that in the resultant state, we will always have access to the transitions if the other choice for transitions were taken.

Let us now prove this intuition that by choosing the most specific transition from many choices, a client will always have access.

Theorem 3.4.1. *Given a DCASA, $M = (\Sigma, Q, \Gamma, q_0, \delta)$, if $\Theta = (p_1, C_1) \cdot (p_2, C_2) \cdots (p_m, C_m)$ is a trace accepted by M, then there exists $q_1, q_2, \ldots, q_m, q_{m+1} \in Q$ such that $q_1 = q_0$ and for $1 \leq i \leq m$, $\delta(q_i, p_i, C_i^*) = q_{i+1}$, where $C_i^* = \bigcup Cond(q_i, p_i, C_i)$.*

Proof. We prove the following stronger statement: For every $\Theta = (p_1, C_1) \cdot (p_2, C_2) \cdots (p_m, C_m) \in (\Sigma \times 2^{\Gamma})^*$ and for all $q, q' \in Q$, if $q \xrightarrow{\Theta} q'$, then there exists $q_1, q_2, \ldots, q_m, q_{m+1} \in Q$ such that $q_1 = q$ and for $1 \leq i \leq m$, $\delta(q_i, p_i, C_i^*) = q_{i+1}$, where, $C^* = \bigcup Cond(q_i, p_i, C_i)$.

It is easy to see that the Theorem is a corollary of this stronger statement. We prove the statement by induction on m, the length of Θ.

Basis. When $\Theta = \varepsilon$, $q \xrightarrow{\Theta} q$. Since $m = 0$, the requirement on δ is vacuously true.

Induction Step. Suppose the statement holds for every trace of length k. We proceed to show that it holds for traces of length $k+1$.

Consider a trace Θ of length $k+1$. Θ has the form $(p_1, C_1) \cdot \Theta'$, where Θ' is in turn a length-k trace of the form $(p_2, C_2) \cdot (p_3, C_3) \cdots (p_{k+1}, C_{k+1})$.

Suppose $q \xrightarrow{\Theta} q''$ for some $q, q'' \in Q$. Then we have $q \xrightarrow{(p_1, C_1)} q' \xrightarrow{\Theta'} q''$ for some $q' \in Q$. By condition (a) of Definition 3.4.2 (DCASA), there exists $q_2 \in Q$ for which $\delta(q_1, p_1, C_1^*) = q_2$ where $q_1 = q$ and $C_1^* = \bigcup Cond(q_1, p_1, C_1)$.

Since $q \xrightarrow{(p_1, C_1)} q'$, it must be the case that there exists $C' \subseteq \Gamma$ such that $C' \subseteq C_1$ and $\delta(q, p_1, C') = q'$. By construction, $C' \subseteq C_1^*$. Therefore, by condition (b) of Definition 3.4.2 (DCASA), $Tr(q') \subseteq Tr(q_2)$. Thus $\Theta' \in Tr(q_2)$. This means, $q_2 \xrightarrow{\Theta'} q'''$ for some $q''' \in Q$.

Since $q_2 \xrightarrow{\Theta'} q'''$, the induction hypothesis implies that there exists $q_3, \ldots, q_{k+2} \in Q$ such that for $2 \leq i \leq k+1$, $\delta(q_i, p_i, C_i^*) = q_{i+1}$, where $C_i^* = \bigcup Cond(q_i, p_i, C_i)$.

We have therefore proven the existence of those states $q_1, q_2, \ldots, q_{k+2}$ as required by the statement in the case when Θ is of length $k+1$. □

In a general sense, this proof demonstrates that when a client provides a proof for a set of conditions and wants to exercise a permission with a DCASA fragment, choosing the "maximal" subset from a set of valid sets of conditions for that state-permission pair would allow the client to perform all the transitions if the sets other than the maximal subset were chosen. The client would not miss out on any future transitions.

3.4.1 The Nature of Determinism and Non-Determinism in Finite Automata and CASAs.

Finite Automata is a well known concept in computer science [29, p 37]. There are two versions of finite automata: a deterministic version (DFA [29, p 45]) and a non-deterministic version (NFA [29, p 55]). CASA is also a form of non-deterministic automata (Section 3.2.1). The deterministic variant of a CASA is a DCASA (Section 3.4.2). We compare CASA with finite automata.

There are some primary differences right off the bat between a CASA and an NFA. Even though they are both represented as quintuples, a CASA does not have final/accepting states like an NFA [29, p 57]. Instead, among the quintuples, a CASA includes a finite set of conditions (Section 3.2.1). Instead of a finite set of input symbols in an NFA, a CASA has a finite set of permissions. The transition function is where an NFA and a CASA differ the most.

The transition function in an NFA takes in a state and an input symbol and returns a subset of states. On the other hand, a CASA takes in a pair consisting of a permission and a set of conditions and returns a state. The non-determinism in an NFA comes from the fact that given the same inputs to the transition function there might be multiple states that the machine can go to, hence the output is a set of states. On the other hand, the non-determinism in a CASA comes from the fact that presenting a superset of the second input would enable that transition. But a superset of the second input would also enable another transition as well provided the first input is the same. The following example is provided to draw a clearer picture.

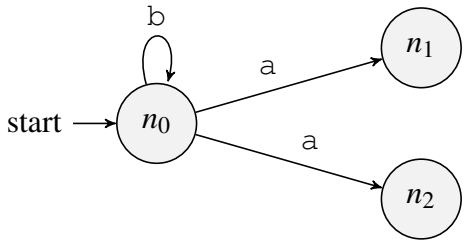

Figure 3.6: **An example of a NFA**

Figure 3.6 is represented by the NFA $N = (\{n_0, n_1, n_2\}, \{a, b\}, \delta', n_0, \{n_1\})$, where δ' is the transition function. In this NFA, $\delta'(n_0, a) = \{n_1, n_2\}$. That means, with input of a, there are two transitions which are taken. Both the states are tracked for all other future transitions. This is where non-determinism comes in NFA. This is not the case with a CASA.

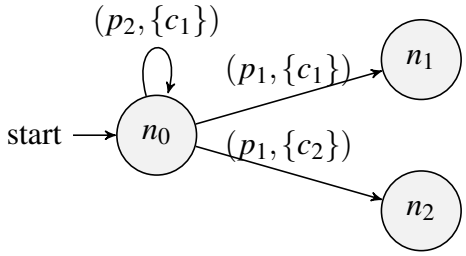

Figure 3.7: **An example of a CASA**

Figure 3.7 is represented by the CASA $M = (\{p_1, p_2\}, \{n_0, n_1, n_2\}, \{c_1, c_2\}, n_0, \delta)$, where $\delta(n_0, (p_1, \{c_1\})) = n_1$ and $\delta(n_0, (p_1, \{c_2\})) = n_2$. But for an input of $(p_1, \{c_1, c_2\})$, both the transitions are valid and the CASA has to choose between one and do future transitions with that. In this way, a user may lose out on future scopes of transitions if only one transition is taken. This is the nature of non-determinism in a CASA. The difference in the nature of non-determinism in a CASA and an NFA is pretty substantial. Even though they are both automata, their structure and nature is quite different.

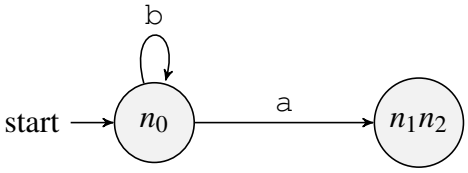

Figure 3.8: **Equivalent DFA of the NFA in Figure 3.6**

Moreover, if we take a look at the deterministic version of the NFA and the CASA we can also see differences. Consider DFA, D in Figure 3.8 which is the deterministic version of the NFA in Figure 3.6. D has only one output to an input of a from state n_0. The resulting state is a combination of states n_1 and n_2. It contains all the transitions from n_1 and n_2. It only returns ***one*** state instead of two in Figure 3.6. Determinism in a CASA is not the same as a DFA as well.

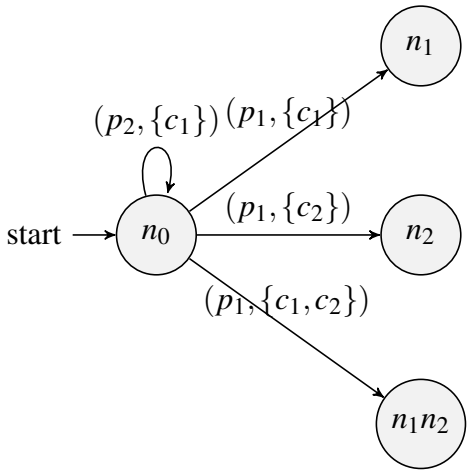

Figure 3.9: **Equivalent DCASA of the CASA in Figure 3.7**

DCASA M' is a deterministic variant of the CASA in Figure 3.7. M follows both the conditions mentioned in Definition 3.4.2, where for state n_0, $\delta(n_0, p_1, \{c_1, c_2\})$ is defined [1]. The CASA is now deterministic, which means there is always a *most specific transition* (Definition 3.4.3) with respect to the evidence that is provided. This was proved in Theorem 3.4.1. The determinism in this DCASA is vastly different from the determinism in a DFA.

In the DFA D from the start state n_0 and an input of a, we have only one state as an output. On the other hand, in the DCASA M', from the start state n_0, with an input of $(p, \{c_1, c_2\})$, all three transitions out of n_0 are valid, but now we can pick transition $\delta(n_0, p_1, \{c_1, c_2\})$. Picking this transition, we would never miss out on any future access (Theorem 3.4.1). This is the difference in the nature of determinism in a DFA compared to a DCASA.

We have identified and defined a subset of CASAs which are deterministic; this solves the issue of making local decisions. If we only put the CASAs which are deterministic in an authorization

[1] Condition (b) does not apply here because pre-requisites of that condition are not met.

token, then we can make local decisions and the client would not miss out on any access. But it is unreasonable to expect a CASA designer to always design a deterministic CASA. So we propose a conversion mechanism which will allow us to convert a CASA to an equivalent deterministic CASA. Later we prove that the DCASA D created from CASA M is equivalent by proving $L(M) = L(D)$.

3.5 Conversion Mechanism

In the previous section we claimed it is unreasonable to expect a policy developer to only design DCASAs and it is easier to design a CASA. Let us illustrate that with an example of a CASA and what a DCASA would look like with the same sequencing constraints.

Consider the following CASAs in the figure below.

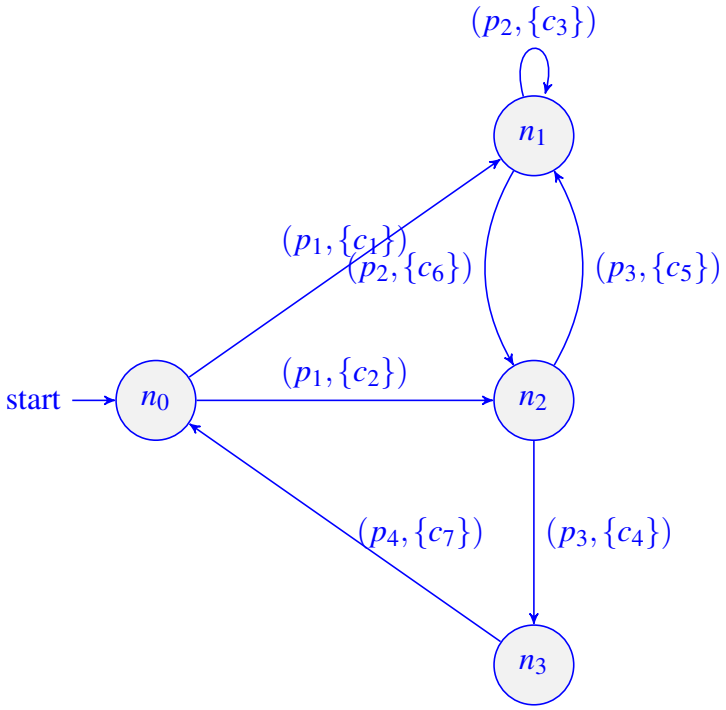

Figure 3.10: **A Non-Deterministic CASA.**

Figure 3.10 is a CASA and it is non-deterministic as the properties defined in Section 3.4.2 is not respected. This CASA is simple to design as it does not have a lot of states, neither does it

have a lot of transitions. Total states in this CASA is 3 and the total number of transitions is 7. If this CASA had to be designed with the deterministic properties in mind we would have the CASA below represented by Figure 3.11.

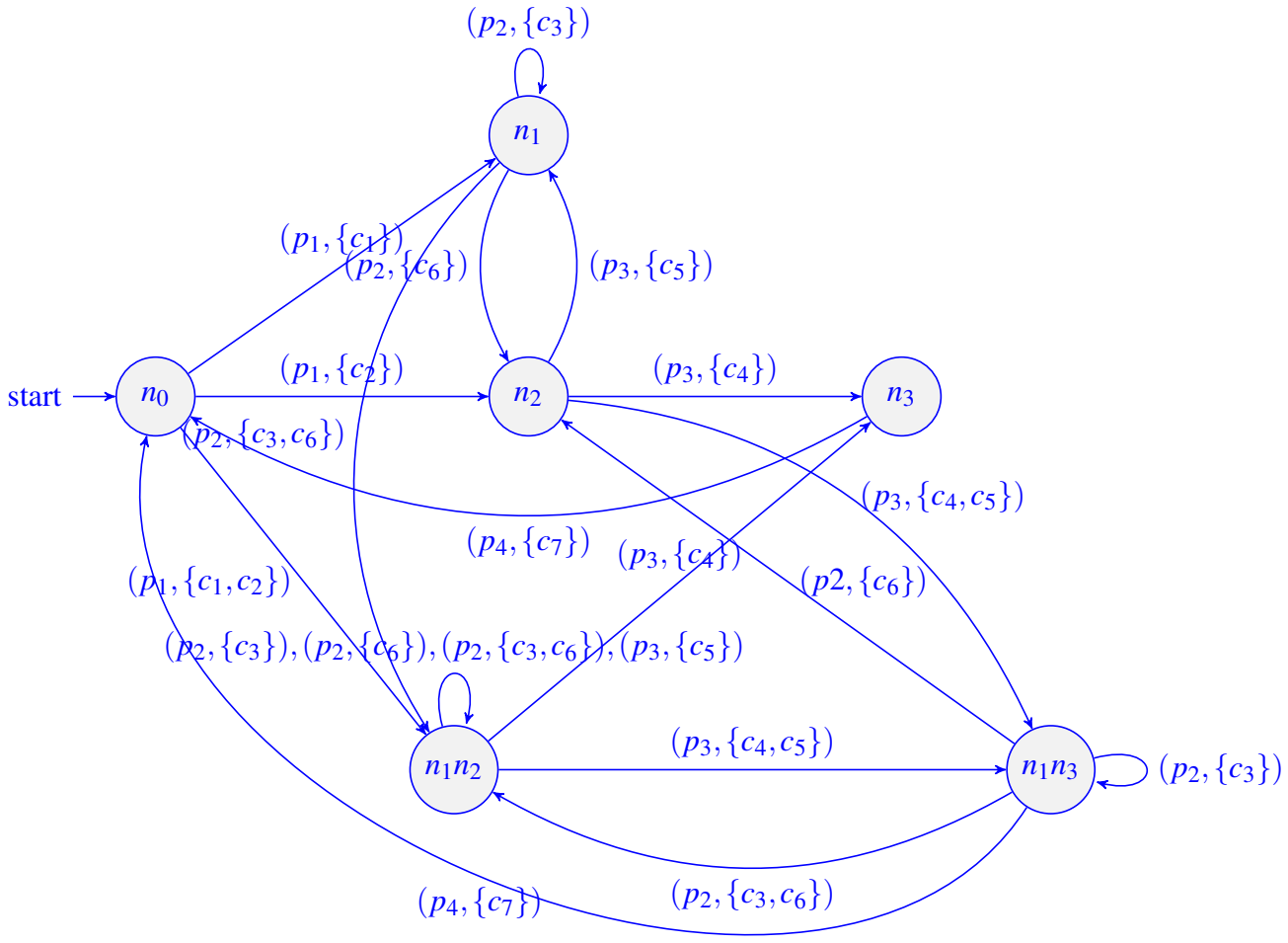

Figure 3.11: **A Deterministic CASA.**

Figure 3.11 is a deterministic CASA enforcing the same sequencing constraints as Figure 3.10. We notice here that this CASA is vastly more complicated than the CASA in Figure 3.10. This DCASA contains a total of 6 states and a total of 20 transitions. It is evident that designing a

CASA is much more preferable to a policy developer than to design a DCASA. On the other hand, with a CASA we have the issue of inability to make a local decision as discussed in Section 3.4. Therefore, it is desirable to have a conversion mechanism that automatically produces a DCASA out of a CASA written by the policy developer.

We, therefore, formulate a conversion mechanism to convert a CASA to a DCASA. We use a subset construction method [29, p. 60] for the conversion process where we will deal with sets of states and sets of sets of conditions. To facilitate the ease of understanding let us first define a few more functions.

Definition 3.5.1. A condition set of a state is defined as the set of all conditions for which $\delta(q,p,C)$ is defined for all $q \in Q$. Formally,

$$Cond(q,p) = \{C \subseteq \Gamma \mid \delta(q,p,C) \text{ is defined and } q \in Q, p \in \Sigma\} \tag{3.4}$$

Consider the CASA represented by Figure 3.5. For that CASA, we have $Cond(q_0,p) = \{C_1,C_2\}$ and $Cond(q_1,p) = \{C_4\}$.

Definition 3.5.2. A condition set for a set of states $S \subseteq Q$, is defined as the union of all sets of sets of conditions in $q \in S$ for which $Cond(q,p)$ is defined.

$$Cond(S,p) = \bigcup_{q \in S} Cond(q,p), \text{ where } S \subseteq Q \text{ and } p \in \Sigma$$

Given $Cond(q_1,p) = \{C_1,C_2\}$, $Cond(q_2,p) = \{C_3,C_2\}$, $Cond(q_3,p) = \{C_4\}$ and if $S = \{q_1,q_2,q_3\}$, then we can say, $Cond(S,p) = \{C_1,C_2,C_3,C_4\}$, where $C_1,C_2,C_3,C_4 \subseteq \Gamma$.

With Definition 3.5.1, we could get the set of sets of conditions **from one state** for a given permission, but with this definition we can get the same, but instead get it **for a multiple states**. This function is necessary in a later section where we have to deal with a set of states in the conversion mechanism.

Next we define a function *closure* which is needed for subset construction to deal with sets of sets of conditions.

Definition 3.5.3. If $T \subseteq 2^{\Gamma}$, we define *closure*(T) to be the smallest $T' \supseteq T$, such that if $C_1 \in T'$ and $C_2 \in T'$ then $C_1 \cup C_2 \in T'$.

For example, if $T = \{\{c_1, c_2, c_3\}, \{c_4\}, \{c_2, c_3\}\}$, then *closure*$(T) = \{\{c_1, c_2, c_3\}, \{c_4\}, \{c_2, c_3\}, \{c_1, c_2, c_3, c_4\}, \{c_2, c_3, c_4\}\}$. This function is necessary because while conversion from a CASA to a DCASA, using subset construction, we have multiple states in CASA represented by one state in a DCASA. Due to this representation and to express all the transitions from all the states in one state in a DCASA, we will need *closure* function which let us get the smallest set of sets of conditions that satisfies Condition (a) in the definition of DCASA in Definition 3.4.2.

We require *closure*(T) to be the "smallest" superset of T that is closed under set union because there are in general many such supersets. For example 2^{Γ} is such a superset. The requirement of "smallest" ensures that *closure*(T) is closed under set union without introducing redundant condition sets.

In this next section we look at how to convert a CASA to an equivalent DCASA accepting the same language.

3.5.1 A Mechanism to Construct a DCASA from a CASA

This section details a process to convert a CASA to a DCASA by the mechanism of subset construction [29, p. 60]. If a CASA $M_1 = (\Sigma, Q, \Gamma, q_0, \delta)$ is given, we can construct an equivalent DCASA M_2. By equivalent CASA I mean $L(M_1) = L(M_2)$. Subset construction involves constructing all the subsets of all the states in M_1. The target of this construction is to create a DCASA $M_2 = (\Sigma, Q', \Gamma, \{q_0\}, \delta')$ such that $L(M_1) = L(M_2)$. The inputs to a DCASA (the set of permissions Σ and the set of conditions Γ) stays the same and the start state of M_2 is a singleton set containing the start state of M_1. The other members of M_2 are constructed in the following way.

- $Q' = 2^{Q}$; Q' is the set of all subsets of Q. Note that it might be the case that not all of these states in M_2 can be reached from the start state $\{q_0\}$ of M_2. As an optimization, one can omit the unreachable states from Q'. We leave this optimization

as an exercise for the reader.

- For every $S \in 2^Q, p \in \Sigma, C \in 2^\Gamma$,

$$\delta'(S,p,C) = \begin{cases} \{q' \in Q \mid \exists q \in S \,.\, \exists C' \subseteq C \,.\, q' = \delta(q,p,C')\} \\ \qquad \text{if } Cond(S,p) \neq \emptyset \text{ and } C \in closure(Cond(S,p)) \\ \text{undefined otherwise} \end{cases}$$

This is the way to construct an equivalent DCASA from a CASA. We claim that the CASA that is constructed as above is deterministic.

Theorem 3.5.1. *A CASA returned by the subset construction method is a DCASA.*

Proof. We begin by defining CASAs $M_1 = (\Sigma, Q, \Gamma, q_0, \delta)$ and $M_2 = (\Sigma, Q', \Gamma, \{q_0\}, \delta')$ where M_2 was constructed from M_1 by the subset construction method defined above.

By construction, for every $S \in 2^Q, p \in \Sigma$ and $C \in closure(Cond(S,p)), \delta'(S,p,C)$ is defined so long as $Cond(S,p) \neq \emptyset$. Therefore condition (a) of Definition 3.4.2 (DCASA) is satisfied by M_2. What remains to be demonstrated is that condition (b) of Definition 3.4.2 is also satisfied by M_2. To that end, we prove the following claim.

Lemma 3.5.1. *Suppose $S_1, S_2 \in 2^Q, p \in \Sigma$, and $C \in 2^\Gamma$. If $\delta'(S_1,p,C)$ is defined and $S_1 \subseteq S_2$, then $\delta'(S_2,p,C)$ is defined and $\delta'(S_1,p,C) \subseteq \delta'(S_2,p,C)$.*

Proof. Suppose $\delta'(S_1,p,C)$ is defined and $S_1 \subseteq S_2$. Observe two facts.

1. Since $\delta'(S_1,p,C)$ is defined, there must be an M_1 state $q' \in S_1$ and $C' \subseteq C$ for which $\delta(q',p,C')$ is defined. Let $q = \delta(q',p,C')$.
2. Since $S_1 \subseteq S_2$, we know that $Cond(S_1,p) \subseteq Cond(S_2,p)$, and thus $closure(Cond(S_1, p)) \subseteq closure(Cond(S_2,p))$. As we have $C \in closure(Cond(S_1,p))$ we also have $C \in closure(Cond(S_2,p))$.

By (1) and (2), $\delta'(S_2,p,C)$ is defined. In addition, the M_1 state q (as defined in (1)) is also a member of $\delta'(S_2,p,C)$. Therefore, $\delta'(S_1,p,C) \subseteq \delta'(S_2,p,C)$. □

Continuing the proof of Theorem 3.5.1. Condition (b) of Definition 3.4.2 requires that, if $C_1 \subseteq C_2$, and both $C_1, C_2 \in closure(Cond(S,p))$, then $Tr(\delta'(S,p,C_1)) \subseteq Tr(\delta'(S,p,C_2))$. This can be demonstrated easily from an inductive argument that makes use of two facts below:

1. $S_1 \subseteq S_2$ implies $\delta'(S_1,p,C) \subseteq \delta'(S_2,p,C)$ [from Lemma 3.5.1].
2. $C_1 \subseteq C_2$ implies $\delta'(S,p,C_1) \subseteq \delta'(S,p,C_2)$ [follows directly from the definition of δ'].

□

Now that we have established that the conversion mechanism indeed produces a DCASA, we proceed to show that the DCASA accepts the same language as the original CASA.

The definition of execution trace-output Δ_M in Definition 3.2.4 can be applied to any CASA M, even if M is deterministic. That means, if N is a CASA and D is a DCASA, then Δ_N and Δ_D are their corresponding execution trace-output. For the next parts, we overload the definition of Δ_M. We overload the notation so that Δ_M can take a set of M-states as its first argument.

Suppose $S \subseteq Q$. Then we overload Δ_M as follows:

$$\Delta_M(S,\Theta) = \{q' \in Q \mid \exists q \in S \,.\, q \xrightarrow{\Theta} q'\}$$

We start with 2 technical lemmas before we state and prove the correctness of subset construction.

Lemma 3.5.2. *Suppose DCASA* $D = (\Sigma, 2^Q, \Gamma, \{q_0\}, \delta_D)$ *is constructed out of CASA* $N = (\Sigma, Q, \Gamma, q_0, \delta_N)$ *by subset construction. Suppose further that* $S_1, S_2 \in 2^Q, p \in \Sigma, C \in 2^\Gamma$*, then* $S_1 \xrightarrow{(p,C)}_D S_2$ *iff* $\Delta_N(S_1,(p,C)) = S_2$ *and* $S_2 \neq \phi$.

Proof. ($\Rightarrow$) Suppose $S_1 \xrightarrow{(p,C)}_D S_2$. We want to show that, $S_2 = \{q' \in Q \mid \exists q \in S_1 \,.\, q \xrightarrow{(p,C)}_N q'\}$ and that $S_2 \neq \phi$.

By Definition 3.2.2, there exists $C' \subseteq C$ such that $\delta_D(S_1, p, C') = S_2$. By the construction of δ_D as seen in Section 3.5.1, this means, $S_2 \neq \phi$ and the following.

$$\begin{aligned} S_2 &= \{q' \in Q \mid \exists q \in S_1 \,.\, \exists C'' \subseteq C' \,.\, q' = \delta_N(q, p, C'')\} \\ &= \{q' \in Q \mid \exists q \in S_1 \,.\, q \xrightarrow{(p,C)}_N q'\} \end{aligned}$$

($\Leftarrow$) Suppose $\Delta_N(S_1, (p, C)) = S_2$ and $S_2 \neq \phi$, then we have:

$$\begin{aligned} S_2 &= \{q' \in Q \mid \exists q \in S_1 \,.\, q \xrightarrow{(p,C)}_N q'\} \\ &= \{q' \in Q \mid \exists q \in S_1 \,.\, \exists C' \subseteq C \,.\, \delta_N(q, p, C') = q'\} \\ &= \delta_D(S_1, p, C) \end{aligned}$$

□

Lemma 3.5.3. *Suppose DCASA* $D = (\Sigma, 2^Q, \Gamma, \{q_0\}, \delta_D)$ *is constructed out of CASA* $N = (\Sigma, Q, \Gamma, q_0, \delta_N)$ *by subset construction. For every non-empty trace* $\Theta \in (\Sigma \times 2^\Gamma)^+$*, for every* $S_1, S_2 \in 2^Q$*,* $S_1 \xrightarrow{\Theta}_D S_2$ *iff* $\Delta_N(S_1, \Theta) = S_2$ *and* $S_2 \neq \phi$.

Proof. We prove Lemma 3.5.3 by induction on the length of Θ.

Basis. The case of $\Theta = (p, C)$ follows immediately from Lemma 3.5.2.

Induction Step. Suppose for every length-n trace Θ, $n \geq 1$, we have for every $S_1, S_2 \in 2^Q$. $S_1 \xrightarrow{\Theta}_D S_2$ iff $\Delta_N(S_1, \Theta) = S_2$ and $S_2 \neq \phi$.

Consider a length-$(n+1)$ trace $\Theta' = (p, C) \cdot \Theta$, and $S_1, S_3 \in 2^Q$. $S_1 \xrightarrow{\Theta'}_D S_3$. We have the following.

$S_1 \xrightarrow{\Theta'}_D S_3$

$\Leftrightarrow \exists S_2 \in 2^Q \,.\, S_1 \xrightarrow{(p,C)}_D S_2$ and $S_2 \xrightarrow{\Theta}_D S_3$

$\Leftrightarrow \exists S_2 \in 2^Q \,.\, \Delta_N(S_1,(p,C)) = S_2 \neq \phi$ and $\Delta_N(S_2,\Theta) = S_3 \neq \phi$

[by Lemma 3.5.2 and induction hypothesis]

$\Leftrightarrow \Delta_N(\Delta_N(S_1,(p,C)),\Theta) = S_3$ and $S_3 \neq \phi$

$\Leftrightarrow \Delta_N(S_1,\Theta') = S_3$ and $S_3 \neq \phi$

□

Theorem 3.5.2. *If $D = (\Sigma, 2^Q, \Gamma, \{q_0\}, \delta_D)$ is the DCASA constructed from the CASA $N = (\Sigma, Q, \Gamma, q_0, \delta_N)$ by subset construction, then $L(D) = L(N)$.*

Proof. We want to prove that, for every $S \in 2^Q$,

$$Tr_D(S) = \bigcup_{q \in S} Tr_N(q) \tag{3.5}$$

The implication of the above claim is that $Tr_D(\{q_0\}) = Tr_N(q_0)$, meaning $L(D) = L(N)$, which establishes the theorem.

Note that ε belongs to both the left and right hand sides of Equation 3.5. Our goal now is to show that the 2 sets on the left and right sides of Equation 3.5 contain exactly the same non-empty traces.

A non-empty trace $\Theta \in Tr_D(S_1)$ iff $S_1 \xrightarrow{\Theta}_D S_2$ for some $S_2 \in 2^Q$. A non-empty trace $\Theta \in \bigcup_{q \in S_1} Tr_N(q)$ iff $\Delta_N(S_1,\Theta) \neq \phi$.

Lemma 3.5.3 ensures that $S_1 \xrightarrow{\Theta}_D S_2$ iff $\Delta_N(S_1,\Theta) = S_2$ and $S_2 \neq \phi$. Therefore, Equation 3.5 holds.

□

To recap, we have defined a deterministic CASA which allows us to make local decisions and therefore can be put into fragments. We have provided an algorithm of converting a given CASA to a DCASA. We have proven that the algorithm does indeed provide a CASA that is deterministic. At the end we proved that the DCASA converted from a CASA is equivalent to the original CASA by proving that both the CASAs accept the same language.

3.6 Discussions on Space Complexity of a DFA and a DCASA Converted Using Subset Construction

The conversion from a non-deterministic finite-state automata (NFA) to a deterministic finite-state automata (DFA) is a very well known concept in automata theory. The conversion follows the subset construction method. The conversion takes an NFA and, using subset construction, constructs an equivalent DFA which accepts the same language [29, Section 2.3.5]. The subset construction mechanism takes as input the set of all the states in the NFA in question, and creates all the subsets of the set of states. That is if the NFA has n states, then the equivalent DFA will have at most 2^n states. That means that the state space in the worst case is $O(2^n)$ where, n is the number of states in the NFA. Although in practice, it is rare that the DFA converted from an NFA has exponential states [29, Section 2.3].

Since my conversion mechanism to convert from a CASA to a DCASA also follows a subset construction mechanism, it also produces, in the worst case scenario, $O(2^n)$ states where n is the number of states in the CASA.

In conclusion, this chapter presents a new kind of automaton called Context-Aware Security Automaton (CASA) which takes into consideration the environment and requires a certain proof of conditions to be presented while requesting access. Then a formal definition is provided of a CASA. A concept of "tamper-proof against information withholding" is presented and it was proved that CASA is in fact tamper-proof against information withholding. Fragments were defined to be a partial specification of a CASA which can be used in the real world in authoriza-

tion tokens. Fragments require a notion of local execution which is not possible due to the non-deterministic nature of CASAs was argued. Two properties were defined which, if possessed by a CASA, would qualify it to be deterministic. The notion of a maximal subset of conditions and the most specific transition were explained. A conversion mechanism to transform a CASA to a DCASA was presented and proved that the resulting CASA was deterministic. It was also established that the converted CASA is equivalent to the original one, meaning that the input CASA and the output CASA accepted the same language. The chapter concludes with a brief discussion on the space complexity of the converted DCASA.

Chapter 4

The Design Specifications of the Context-Aware History-Based Capability System

In this chapter, I present the overall system design of the Context-Aware History Based Capability System. I will go over the participating entities of a context-aware history-based capability system (Section 4.2), the building blocks of the system 4.4, and the different protocols that drive the system (Section 4.5).

4.1 Conditions

We are moving into a world of sensors and IoT devices all around us. Sensors being small in size have the potential to have a wide deployment. It is a natural step to include their inputs in an authorization scheme. Currently most of our IoT devices and handheld smartphones can be connected to the internet and can efficiently gather data from these sensors and fulfill the conditional requirements in this authorization scheme. I will talk more about the efficiency in the following chapter.

Conditions dictate the state of the environment and they are factors in the decision making process of an authorizing entity [46]. These conditions are provided as a part of the Context-Aware Security Automaton, which are to be signed by a trusted entity. These trusted entities are requested to confirm the conditions to be true at the time of requesting. If the conditions are true at the time of requesting, the entity provides a certificate proving the condition to be true. The condition may require a numerical or boolean assessment by a sensor or a group of sensors. For simplicity, the assessment of a condition may only yield a boolean result. I have tried to model the conditions after the $UCON_{abc}$ model in [46], where conditions requirements are immutable. That

means that conditions are only effected by environmental changes and evaluations of conditions does not update the internal state of any entity.

A condition may require a sensor to say that a person is present in a room by using a motion detecting sensor. Such a condition would only require a boolean operative. The proof of this condition may be required by a resource server if an entity wishes to access a resource, such as, a light or an air conditioner. Other examples of conditions may include, location at the time of requesting, the current time (e.g., no access after 5 p.m.), security status of the system [46], temperature at room 624, emergency alarm status, velocity of a vehicle is above 80 miles per hour, and so on and so forth. Policies may be set by an administrator, which will dictate which access requires which set of conditions to be true.

4.2 Entities in the Context-Aware HCAP Ecosystem

In this section, I will discuss in details the objective, functionality and the design of the entities (i.e., components) involved in the context-aware HCAP system. The groundwork of the core HCAP has been laid out by Tandon *et al.* [54]. I have reused the code from [54] and extended the code to incorporate context-awareness. The system has four participants: the authorization server, resource servers, sensor integration centers and the clients. The authorization server, the resource server and the clients were inherited from Tandon *et al.* [54] and I have introduced the Sensor Integration Center. With the inclusion of the Sensor Integration Center, changes were made to all the other servers as well. In this section, I shall go over the participants in brief and point out the major upgrades that were made to the base HCAP model in [54].

4.2.1 Resource Server

The resource server that I designed is an extension of the design made by Tandon *et al.* [54] which I described in 2.1.2. In the original HCAP, the resource server is a computationally constrained IoT device controlling access to several resources via an authorization algorithm in place. Clients

request the resource server to do operations on resources, and the resource server accepts or rejects the request based on the authorization algorithm in place.

On the other hand in my design of the context-aware HCAP, in addition to the authorization algorithm in HCAP, the resource server needs to do some proof checking as a part of the authorization scheme. The proof is presented by the client as a part of the request sent to the resource server. These proofs are, in a very basic sense, statements signed by a trusted entity, and it is the duty of the client to gather the proof. This is different from the original HCAP model. I have taken this extra proof checking into consideration while designing the context-aware extension to HCAP. I will discuss more about this in the upcoming sections.

4.2.2 Client

Clients are the end users of the system, who go about requesting access to resources with authorization tokens. Their roles in both original HCAP and context-aware HCAP is similar with some changes. In the context-aware HCAP, the client has to talk to one or multiple sensor integration centers to construct a proof of compliance before actually requesting to exercise a permission. There is a proof caching feature added into the client side, which can be turned on and off (Section 4.5.4). This feature, when turned on, requires some memory on the device in the part of the client but makes up for it in faster access decisions.

In the current world of technology, even the smallest devices have a decent amount of memory. The client is expected to be running the application without any obstacles. The design accounts for client applications to be running on multiple client devices, performing random actions.

4.2.3 Sensor Integration Center

A Sensor Integration Center (SIC) is a server which has access to the readings of several sensors. A Sensor Integration Center may have access to the temperature reading of all rooms and hallways of a floor of a building. It may also have access to the reading of all the motion detecting sensors of a whole floor, or building. SICs only have access to sensors and their data. There might be many

SICs, each of which may have access to different sensor reading. Their main task is to provide proof of the condition requirement requested by a client at the time of request. Each SIC have unique signatures which are part of a Public-Key Infrastructure in place.

Clients can request the Sensor Integration Centers to check if a condition is true prior to the time of the request. The client does it by requesting a specific type of certificate that I designed, called **Condition Certificate**. The SIC gets the readings from the sensors and if the requirement is fulfilled, the SIC signs the condition and gives it back to the client in the form of a Condition Certificate. If the sensor is not accessible to the requested SIC, then the SIC suggests another SIC the client can talk to, again, in the form of Condition Certificates. The SICs were designed to respond with three different kinds of Condition Certificates, which I will discuss in Section 4.4.1. If the sensor pertaining to the conditional requirement is accessible by the SIC in question, the SIC may or may not provide a certificate depending on, if the condition *C* is met.

To sum up, the SIC may respond in the following ways.

- Respond with a Condition Certificate asserting, *"The condition C is true, if a delegate of SIC-B says its true."*
- Respond with a Condition Certificate asserting, *"The condition C is true, if SIC-B says its true."*
- Respond with a Condition Certificate asserting, *"The condition C is true."*
- Respond null if the conditions are not met.

The details of the structure of the types is discussed in Section 4.4.1 and the SIC protocols are discussed in Section 4.5.3.

4.2.4 Authorization Server

An administrator defines Context-Aware Security Automata (CASA) for each client session which acts like a reference monitor [52]. CASAs specifies not only the permissions the client can exercise,

but also the order in which they can be exercised. Furthermore, each permission may or may not require a condition to be proven true by the client at the time of request. All of these are encoded in a CASA. But as we have seen in Section 3.4, CASAs are naturally non-deterministic. Therefore, an algorithm is in place to convert a CASA to a Deterministic Context-Aware Security Automaton (DCASA). This conversion process is the duty of the Authorization Server. This conversion is done only once before the first capability is processed to be given to a client.

After an administrator defines the policies and stores client information, the clients are free to communicate with the authorization server. The authorization server upon receiving a request from a client, authenticates the client and provides the client with a capability and a set of Condition Certificates. The capability is an access token which need to be used to get access to resources from the resource server. The Condition Certificates provided by the authorization server, on the other hand, are either type-1 or type-2 Condition Certificates. These Condition Certificates contain information on which SIC to talk to in order to gather proofs for the condition requirements specified in the capability. To be noted here, that the authorization server does not provide type-3 Condition Certificate. This process is discussed in details in Section 4.5.2.

4.3 Trust preliminaries between entities

Before the discussion about the building blocks and protocols, we need to discuss which entities are trusted to do what. Several assumptions are made during the design of the context-aware HCAP framework. This is necessary in order to determine which components are trusted to work securely and which components have the capacity to act in a malicious fashion.

Since the implementation work builds on top of the work done by Tandon *et al.* [54], the trust assumptions in the original HCAP still hold in the context-aware extension of the work. Trust Assumptions are assertions which state which entity or what are trusted. Trust Assumption (TA) [1-6] are unchanged from [54] and are in effect on the context-aware extension of HCAP. These trust assumptions have been mentioned in Section 2.1.3. I have added two extra Trust Assumptions, TA-

7 and TA-8. They both involve assumptions based around the SICs. These extra trust assumptions are necessary for the smooth operation of the context-aware variant of HCAP.

TA-7 The SICs are all trusted by the Authorization Server and the Resource Servers to accurately provide Condition Certificates.

TA-8 The identities of the SICs are also part of the Public Key Infrastructure (PKI) in place so that their public keys can be verified.

4.4 Building blocks

This section explains the building blocks of the context-aware HCAP. These building blocks enable explaining the features provided by the context-aware HCAP. Using these concepts, features such as, enforcing the permission sequence constraints, condition proofs and capabilities as tokens issued to the client are discussed.

4.4.1 Condition Certificate Structure

A certificate is an assertion signed by a trusted entity. In the scope of context-aware HCAP, the assertions are the conditions specified in the CASA. In order to avoid confusion and give a distinctive identity, I have named these signed assertions *Condition Certificates*. This subsection describes these Condition Certificates which were specifically designed with the context-aware nature of HCAP in mind. To sign the conditions requested by the client, the Condition Certificates have a well structured domain which was carefully chosen to detail several information pertaining to the condition in question. The domain includes a certificate validity period, which is represented by a start time and an end time. It also includes the issuer identification information, type of certificate, the condition itself and an encrypted string for authentication purposes. Depending on the type of certificate, the fields maybe different. The range of the certificates are respective values which are described in later sections.

In [7], Abadi et al. studied access control in distributed computing systems. Their work accounts for how one principal may believe another principal making a statement by itself or on behalf of someone else. This method of speaking on behalf of another principal is delegation. In context-aware HCAP, we have a delegation structure realized by Condition Certificates. Only a basic assertion of a condition being true is not enough because if there are only a number of entities certifying conditions are true, then with a lot of clients, they quickly become bottlenecks.

To support delegation, my scheme follows Abadi *et al.* [7], and features 3 types of certificates. First, there are basic certificates, in which an SIC asserts a statement x. Then there are certificates that allow a directory service to refer the client to different SICs. Lastly, there are certificates that a higher-level directory service uses to refer a client to lower-level directory services. Assuming two entities: *SIC-A* and *SIC-B*, the 3 types of certificates are the following.

- *SIC-A* says, in matters of x, trust a delegate of *SIC-B*, if *SIC-B* says so. This is a ***Type-1 Condition Certificate***.
- *SIC-A* says, in matters of x, trust *SIC-B*. This is a ***Type-2 Condition Certificate***.
- *SIC-A* says x. This is a ***Type-3 Condition Certificate***.

In the context-aware HCAP, one SIC may delegate the power to certify a condition to another SIC. For example, SIC-A may certify that, in matters of the temperature of room 624, believe whomever SIC-B says to believe by providing a *type-1 Condition Certificate*. Or, SIC-A may say, in matters of the temperature of room 624, believe what SIC-B says about this matter by providing a *type-2 Condition Certificate*. The last case may be that, SIC-A may certify that the temperature of room 624 is less than 18 degrees by providing a *type-3 Condition Certificate*.

The following are the detailed description of the types of Condition Certificates and the domains of the Condition Certificates. It should be mentioned here that, since there is a PKI system in place, when we talk about IDs, we mean the *Issuer Name*, as they appear on the X.509 Certificates used for authentication.

1. **Type-1 Condition Certificates.** Type-1 Condition Certificates are delegation certificates which is representative of SIC-A saying, in matters of *x*, trust whoever SIC-B tells you to trust. The following is the domain of type-1 Condition Certificates.

    (a) Condition. The condition for which one entity trusts another entity to say something about.

    (b) Issuer ID. The identification of the entity that has signed this Condition Certificates. In the example previously, that is SIC-A.

    (c) Type. The type of the Condition Certificates. For this, this type is fixed to be 1.

    (d) Start Time. The time at which the Condition Certificates started to be valid. This is usually a *long* in milliseconds. The range of this is the time at which the Condition Certificates starts being true.

    (e) End Time. The time at which the Condition Certificates ceases to be valid. This is usually a *long* in milliseconds. Since this Condition Certificates is a delegation certificate, and not certifying a condition, it is usually valid for a longer time. Depending on the issuer of this certificate (SIC-A in this case), the lower bound for the End Time is usually 15 minutes from the Start Time and the upper bound could be at most a week from the Start Time.

    (f) Next ID. The identifying name, as it appears in the X.509 Certificate of the delegated entity. In the example previously, that is SIC-B.

    (g) Encrypted String. An encrypted string signed by the private key of the issuer.

2. **Type-2 Condition Certificates.** Type-2 Condition Certificates is also a delegation

certificate which is representative of *SIC-A* saying, in matters of *x*, trust *SIC-B*. The following are the domain of type-2 Condition Certificates.

(a) Condition. The condition for which one entity trusts another entity to say something about.

(b) Issuer ID. The identification of the entity that has signed this Condition Certificates. In the example previously, this is *SIC-A*.

(c) Type. The type of the Condition Certificates. This type is fixed to be 2.

(d) Start Time. The time at which the Condition Certificates started to be valid. This is usually a *long* in milliseconds. The range of this is the time at which the Condition Certificates starts being true.

(e) End Time. The time at which the Condition Certificates ceases to be valid. This is usually a *long* in milliseconds. Since this Condition Certificates is a delegation certificate, and not certifying a condition, it is usually valid for a longer time than a *type-3 Condition Certificate*, but shorter than a *type-1 Condition Certificate*. Depending on the issuer (SIC-A), the lower bound for the End Time is usually 5 minutes from the Start Time and the upper bound could be at most a few days from the Start Time.

(f) Next ID. The identifying name, as it appears in the X.509 Certificate of the delegated entity. In the example previously, that is SIC-B.

(g) Encrypted String. An encrypted string signed by the private key of the issuer.

3. **Type-3 Condition Certificates.** Type-3 Condition Certificates are representative of entity *A* saying *x*. This means the issuer of this Condition Certificates has access

to the data of the sensors, using which an entity can verify the condition. The following are the domain of type-3 Condition Certificates.

(a) Condition. The condition for which an entity is providing an assertion for.

(b) Issuer ID. The identification of the entity that has signed this Condition Certificate. In the example, this is SIC-A.

(c) Type. The type of the Condition Certificates. For this, this type is fixed to be 3.

(d) Start Time. The time at which the Condition Certificates started to be valid. This is usually a *long* in milliseconds. The range of this is the time at which the Condition Certificates starts being true.

(e) End Time. The time at which the Condition Certificates ceases to be valid. This is usually a *long* in milliseconds. Since this Condition Certificates is saying that a condition is true, it is usually valid for a shorter time in comparison to the type-1 or type-2 Condition Certificates. It also depends on the policy of the SIC which provides type-3 Condition Certificates. For example, the lower bound for this Condition Certificate can be 5 to 10 seconds and the upper bound could be a few minutes from the Start Time.

(f) Encrypted String. An encrypted string signed by the private key of the issuer.

Using these types of Condition Certificates, the client forms a proof and presents it to the resource server (described in Section 4.5.4). To be noted here that the encrypted string is basically a concatenation of the information of the Condition Certificate, signed by the private key of the

issuer. The resource server checks this encrypted string with the public key of the issuer to validate that the Condition Certificate was indeed signed by the issuer.

4.4.2 Tickets

Tickets are tokens that the clients use in order to request to exercise a permission. There are two kinds of tickets: (1) Capabilities and (2) Update requests. The **capability** type tickets are generated by the authorization server, and the resource server can generate both the **capability** and the **update request** type of tickets. Clients get their first ever ticket from the authorization server. Tickets are presented to the resource server by the clients in order to request to exercise a permission[1]. The resource server then may or may not provide the client with a new updated ticket in the form of either capability or update request. The tickets are structurally very similar to the tickets in the original HCAP explained in Section 2.1.4.

The only difference in the structure of **capabilities** is the CASA fragment which is dissimilar to a SA fragment. The ticket α is concatenated with a secret key k for a client whose identification tag is *cid*. The concatenated value is then hashed. The hashed value is appended as the authentication tag and is used by the resource server to authenticate the ticket. The clients cannot use tickets issued to other clients because the *cid* is authenticated by X.509 certificates. Capabilities have the form $\langle\langle sessid : cap(t_{ser}, F)\rangle\rangle_{k,uid}$, where t_{ser} is the time when the CASA entered into the current state (is also used as a serial number for the capability) and F is a CASA fragment (as opposed to a SA fragment in original HCAP). Other than the CASA fragment, this process and form of capabilities is exactly like original HCAP.

On the other hand, the structure of the **update request** type ticket is different as well. The update request has the form $\langle\langle sessid: upd(e)\rangle\rangle_{k,cid}$. Here, previously e was the exception list containing transitions performed by the client, whereas in my design it contains a bit more information which is explained in the next section.

[1] Clients may also provide a proof, as a supplement to the tickets, explained in Section 4.4.4.

4.4.3 Exceptions

The exception e in my design is a quadruple containing a transitioning permission name, a timestamp, a set of conditions and another exception. In the work done by Tandon *et al.* [54], the exceptions are only missing the set of conditions. When the client presents a set of valid Condition Certificates for a set of conditions and requests to exercise a permission, and it gets accepted, the transition taken by the resource server in the CASA fragment is recorded as an exception. A transition in a CASA is a permission and a set of conditions, which are both stored in an exception. Drawing from the inductive definition of exceptions in original HCAP explained in Section 2.1.4, the following is the definition of exceptions in context-aware HCAP.

$$e ::= nil(t) | ex(p, C, t, e)$$

By this definition, exception list e is either $nil(t)$, where $t \in \mathbb{N}$, or it is a triple containing $p \in \Sigma$, $C \subseteq \Gamma$ and a timestamp t. Since it is stored in a recursive manner, the extra e is another exception list. Just like the design mentioned in Section 2.1.4 under Exceptions, it is ordered in a descending manner based on the timestamps.

The resource server maintains separate exception lists for each client. Initially the authorization server provides a capability with a start state and stores that start state for a client as the current state. As the client keeps on exercising permissions, the current state changes. Resource servers keeps track of the transitions taken. The authorization server can deduce the current state by applying the transitions stored in the exception list in a chronological order.

4.4.4 Structure of Proof

The clients may need to present a proof for a number of conditions, as a part of a request to exercise a permission. The transitions in a CASA not only specifies a permission, but also, a set of conditions that need to be true at the time of requesting to exercise the permission. The clients need to gather assertions about conditions in order to exercise certain permissions of their choice.

These assertions take the form of type-3 Condition Certificate, which were explained in Section 4.4.1.

In the base case, a proof is a type-3 Condition Certificate. On the other hand, the client may receive a type-1 or a type-2 Condition Certificate and may need to refer to a different SIC in order to eventually gather a type-3 Condition Certificate. In this scenario, a client needs to gather Condition Certificates and form a proof tree. To facilitate the discussion let us represent Condition Certificates in the following ways and lay down some rules for natural deduction [31, p.6].

Let us represent Condition Certificates in the following ways. They follow the definition mentioned in Section 4.4.1.

- Type-1 Certificate: $A.x \rightarrow B.x$
- Type-2 Certificate: $A.x \rightarrow B$
- Type-3 Certificate: $A \; says \; x$

The rules for natural deduction are as follows.

- Rule 1:

$$\frac{A.x \rightarrow B \qquad B \; says \; x}{A \; says \; x}$$

- Rule 2:

$$\frac{A.x \rightarrow B.x \qquad B.x \rightarrow C}{A.x \rightarrow C}$$

- Rule 3:

$$\frac{A.x \rightarrow B.x \qquad B.x \rightarrow C.x}{A.x \rightarrow C.x}$$

Consider an authorizing entity, lets say a resource server, trusts entity A to say something about condition x. The client is in possession of the following Condition Certificates.

1. $A.x \rightarrow B.x$
2. $B.x \rightarrow C.x$
3. $C.x \rightarrow D$
4. D *says* x

Considering the facts that, 1. the resource server only trusts entity A to say something about condition x, and 2. the Condition Certificates presented are valid and are premises, the following proof tree can be constructed with the natural deduction rules mentioned previously.

$$\dfrac{\dfrac{\dfrac{A.x \rightarrow B.x \qquad B.x \rightarrow C.x}{A.x \rightarrow C.x} \qquad C.x \rightarrow D}{A.x \rightarrow D} \qquad D \text{ says } x}{A \text{ says } x}$$

We can conclude that indeed A *says* x. These proof trees are skewed to one side and that is why it looks like a linear structure. In this design, the resource server trusts the authorization server. By providing a proof of a condition which concludes with the authorization server saying a condition is true, the client proves to the resource server that, that condition is true. The client provides one proof tree for each condition for the permission the client is requesting and is evaluated by the resource server.

4.5 Context-Aware HCAP Protocols

The previous section talked about the building blocks on which the context-aware HCAP is built on. In this section, I will talk about how each entity talks to one another, what information each holds, how the SIC function and with the addition of context-awareness, how clients behave. I will also explain how the resource servers authorize a permission request. At the very end, I will talk about some extra features such as garbage collection and proof caching.

4.5.1 Server States

There are 3 servers in my design of the context-aware HCAP, authorization server, resource server and a sensor integration server. The authorization server and the resource server needs to keep some information related to the client and the activities of the client enclosed in sessions. The server states are named accordingly and stores pertinent information under client sessions.

Just like original HCAP, in context-aware HCAP the authorization server and the resource server saves and maintains a shared key, using which, they hash tickets and authenticates each others messages traveled through the client. In that way, the client cannot modify the ticket or any Condition Certificates it may be given. Secondly, the SIC does not maintain any information regarding the clients or the authorization server. The SICs simply receive requests from the client asking for Condition Certificates. A Public-Key Infrastructure is in place so that the client knows the public keys of the SICs and the SICs knows the public keys of the clients in the form of X.509 certificates.

Let us look at the information the **Authorization Server** maintains. The authorization server maintains four maps: 1. *monitor[sessid]*, 2. *state[sessid]*, 3. *serial[sessid]*, 4. *fragment[M,q]*. Maps *state[sessid]* and *serial[sessid]* are similar to Section 2.2 following original HCAP [54]. The *monitor[sessid]* map differs from original HCAP because it maps a session id to a DCASA instead of a SA. The *fragment[M,q]* maps a DCASA M and a state q to a fragment instead of a SA in original HCAP. The usage and functionality of these maps are the same as original HCAP.

The ***fragment*** map assigns a fragment to every state of every DCASA used by the authorization server. A fragment is pre-computed for each state in each possible DCASA and is stored in this mapping. This will be touched on again in Section 4.5.2. This is done so that when a client needs a new fragment from the authorization server, there is no delay. The trade-off for this pre-computation is some delay when a new CASA is formulated as an access control policy. So, when a new client is added and the client requires a CASA not already present in the map, then pre-computation is incurred to make that CASA deterministic as well as creating fragments for each

state in the DCASA. This is done only once and only for the first time. If several clients share the same CASA, this computation can be shared.

The **Resource Server** maintains the same information as the original HCAP: 1. a timestamp t_{rs} and 2. a map $ex[\cdot]$ that records exceptions. These are explained in Section 2.2.2. The only difference is the structure of exceptions. The exceptions used by the resource servers in context-aware HCAP uses a structure explained in Section 4.4.3.

The **Sensor Integration Centers** *do not* store or maintain any information.

4.5.2 Authorization Server Initialization and Protocols

An administrator needs to set up the Public-Key Infrastructure so that the client and the authorization server recognize each other. The administrator also needs to setup which CASA will be given to which client. There might be multiple CASAs which might be distributed to different clients in the form of Fragments. The authorization server in the work by Tandon *et al.* [54], maintains some information about the clients in the form of maps through client sessions which are mentioned in Section 2.2.1. In this work, the authorizations server maintains the same maps with two changes.

Firstly, in this work, the *monitor[sessid]* is a map which maps each session identification of a client to a DCASA for that specific client instead of a Security Automaton (SA). *Secondly*, in this work, the *fragment[M,q]* map, maps a pair of DCASAs and a state to a DCASA fragment instead of SAs. For all states in M, a fragment is pre-computed in the event a client revisits a state during the traversal of the CASA.

Before the fragments are computed, the authorization server runs a conversion algorithm to convert the CASA into a DCASA using the conversion mechanism described in Section 3.5.1. This is only done when the client is assigned a CASA which is not already present in the fragment map. In case a deterministic version of the CASA is not present in the fragment map, the CASA needs to be converted to a DCASA before the fragment for this DCASA is put in the map. The structure of the other maps are left unchanged.

When the client asks for a capability for the first time, the authorization server runs the conver-

sion mechanism and initializes the maps for the client session. Each client is issued a *sessid*. My design diverges here from the work by Tandon *et al.* [54]. In my design, the authorization server extracts all the conditions present in the fragment $fragment[M, q_0]$ and creates a set of type-1 or type-2 Condition Certificates for all of these conditions. This is created by the authorization server for three reasons, 1. So that the client can form a proof tree for any condition to present to a resource server, 2. So that the resource server trusts the condition to be true as the root of the proof tree is the authorization server and the resource trusts the authorization server, 3. So that the client can identify the entity which can provide a type-3 Condition Certificate, or provide a means to find an SIC to provide a type-3 Condition Certificate using the PKI system in place.

After this, the authorization server initializes the maps for the user in session *sessid*, the authorization server provides a capability. The form of the capability is the same as the capability in the work by Tandon *et al.* in [54], described in Section 2.2.1, with one subtle difference in the fragment in the capability. The fragment in this work is a DCASA fragment instead of an SA fragment. The authorization server sends the capability with the DCASA fragment and the set of Condition Certificates to the client.

4.5.3 SIC protocols

The Sensor Integration Center or SIC is a server which only serves one purpose, that is, to provide Condition Certificates to the clients as a part of the clients proof of a condition. SICs have the authority over providing any type of Condition Certificates. SICs also have access to different sensor values which it can read at any time. SICs do not require to store information about the clients, although they are required to be a part of the PKI in place, in order to securely communicate with the clients.

A condition is associated with the reading of a sensor. These sensor readings provide the latest and current environmental status. On the other hand, conditions outline a threshold or range of readings from a specific environment which can be verified by an SIC. For example, condition requirement could be a time period (till 5 p.m.), could be a location verification, could be a temper-

ature range of a certain room, system load and so on [46]. If the condition in question is satisfied, the SIC provides a Condition Certificate of type-3. Type-3 Condition Certificates are the only certificates which can verify that a condition is met. SICs who provide type-3 Condition Certificates receive requests from the client, read the relevant sensor value, evaluate if the condition is valid. If so, responds with a type-3 Condition Certificate, otherwise responds null.

On the other hand, if the SIC receives a request for the certification of a condition which the SIC is unable to provide for whatever reason, the SIC returns a type-1 or a type-2 Condition Certificate. The decision for which SIC responds with which type of Condition Certificate are left for the SIC administrators and designers and are not within the scope of this work. What I have done, is create the framework for the clients to gather the Condition Certificates, in order to successfully execute a permission. The SICs only sign Condition Certificates and provide them to the clients. They do not connect with the authorization server or the resource server and do not store any information about the requests and the clients.

4.5.4 Client Construction of Proof

For every condition appearing in the DCASA, the authorization server produces at least one certificate that is of either type-1 or type-2. Not only that, when an SIC receives a request to find out if a condition holds, it will return *at least* one certificate: type-1 or 2 or 3. For the client to construct the proof tree, the client has to go to different SICs and try and complete a proof (Section 4.4.4).

When a client is constructing a proof it is quite like doing a graph walk (vertices being the SICs, and edges being type-1 certificates). The proper algorithm to construct a proof is a graph search (e.g., DFS, BFS) and therefore the running time of such a graph search algorithm is $O(V+E)$. Identifying one potential path in a graph structure is efficient. Each path in the graph corresponds to a *potential* proof. The issue with this is that the client may not succeed in obtaining a type-2 or a type-3 Condition Certificate needed to complete a proof.

If the client cannot form a proof following one path of the proof tree, the client has to backtrack and try out a different path. Backtracking repetitively would lead to exponential runtime. If the

number of type-1 Condition Certificates returned by the authorization server and SICs for a single condition is bounded by a positive integer b and the size of the proof is n, then the runtime would be $O(b^n)$. In order to control this exponential nature of proof construction, we can make some structural assumptions on the graph so that it is much more manageable. Specifically, one of the following assumptions can be adopted:

Assumption 1: **Proofs are short.** For example, mandating that only the Authorization Server can issue type-1 certificates, then $n \leq 3$. In this scenario, the SICs would only deal with type-2 and type-3 certificates and type-1 certificates would be the monopoly of the Authorization Server. The runtime in this case would be cubic instead of exponential.

Assumption 2: **Fixing $b = 1$.** This means that the Condition Certificate provided by Authorization Server or the SICs for one condition is always exactly 1. This means that the SICs would be acting like a hierarchical directory structure where the client essentially does a look-up for the next SIC to go to. For example, in order to get a Condition Certificate for a room in the ICT building at the University of Calgary, the client might have to go to SIC-MainCampus covering the main campus, who might provide a type-1 certificate directing to go to SIC-ICT. Next, SIC-ICT may provide a type-2 certificate directing to SIC-FifthFloor and the SIC-FifthFloor may provide a type-3 certificate. In fact, in this implementation and experiments, this kind of hierarchical structure is at play and the authorization server and the SICs all provide *only one certificate* for each condition. This is also the reason for a linear proof tree structure.

There can be a hybrid assumption as well where only the Authorization Server can provide multiple type-1 certificates but the SICs can provide at most one type-1 certificate. In this way, elements from both the aforementioned assumptions are present. The graph is not a linear structure but the client does not have to try many paths. The runtime for this structure is $O(b+n)$. This

is far better than the exponential runtime of backtracking and provides a middle ground between assumptions 1 and 2.

Caching. As directed by the authorization server, the client goes to an SIC and keep going to new SICs, referred to in the last Condition Certificate until eventually they find a type-2 and then a type-3 Condition Certificate. The proof tree may contain several type-1 Condition Certificates ending with a type-2 and then type-3 Condition Certificate. This is discussed in Section 4.4.4. The root of the proof tree *must be* signed by the authorization server. In this way, the client collects the Condition Certificates culminating to a proof of a condition. When all the pertinent conditions has a proof, the client is ready to request a resource server for access to a resource. The structure of presentation of these proofs is in the form of a map which maps a condition to a proof tree.

The client discards all the Condition Certificates except the ones provided by the authorization server after the request was granted by a resource server. This is done to conserve memory in the client device. In case there are a large number of Condition Certificates that the client needs to store, and if the Condition Certificates have a long validity period, then the client device may be bloated. Therefore, for every condition, before every request, the client gathers necessary certificates and forms proof trees by contacting SICs.

On the other hand, the client may decide to **cache** the Condition Certificates. In this case, the client does not discard the certificates. Instead, before every request, the client checks the validity of each certificate for each condition starting from the root certificate from the authorization server. In the event a certificate is over its validity period, that certificate along with all the certificates succeeding that certificate are discarded. The client builds the rest of the proof contacting the next SICs. This method is called **proof caching**. This is the way the client constructs the proof before contacting the resource server. The effects for *proof caching* is discussed in a later chapter.

4.5.5 Resource Server Authorization

Clients need access to resources. Resource servers are the safeguards to the resources. The clients need to ask the resource server to authorize the permission it requests. The client needs to contact

the resource server after the client has successfully obtained a capability and relevant proofs of conditions and request the resource server to exercise a permission. The resource server evaluates the capability and the proofs and depending on the evaluation, accepts or denies access and returns a ticket to the client. The request is authorized according to Algorithm 4. Let us discuss Algorithm 4 in more details.

The client provides a quadruple to the resource server, $(uid, p, \langle\langle sessid : cap(t_{ser}, F)\rangle\rangle_{?,?}, proof)$, where *uid* is the clients unique identification number, p is the permission the client wants to exercise, the third item is the capability. These three items serve the same purposes as original HCAP (Section 2.2.2). The fourth item *proof* is a map which maps a condition to a proof tree (discussed in Section 4.4.4), with a Condition Certificate from the Authorization Server serving as the root of the proof. The client only sends the relevant proofs pertaining to the permission in question as per the fragment in the capability.

Lines $1-11$ are exactly the same as original HCAP for multiple resource servers (Algorithm 2). After the resource server receives this quadruple from a client, it validates the capability using steps 1 and 2 mentioned in Section 2.3.2 following original HCAP for multiple resource servers (Algorithm 2) as can be seen from lines $1-11$. After that the resource server checks if the permission requested is in the fragment (line 12). Then the resource server checks each mapping in *proof* for each condition provided by the client (line 13) and extracts a set of valid conditions which are proved to be true by the client at that time. Checking the validity of the proofs is not present in original HCAP and is specific to context-aware HCAP. If the set of valid conditions is C', the resource server calculates the maximal condition set, given state q and permission p according to Definition 3.4.3 in line 14. Finally if the triple (p, q, C_{max}) is in the domain of δ, the permission is granted (lines 15-16). If the current state q is the same as the next state $\delta(q, p, C_{max})$, no state change occurs and nothing is returned (line 17-18). Otherwise, the current time is recorded and the exception list is updated (lines 19-20). Depending on the existence of the next state in the fragment, the client is returned an update request or a new capability (lines 21-25).

Algorithm 4: Authorize Access Request in Context-Aware HCAP (multiple resource servers).

Input: A client access request $(uid, p, cap, proof)$, where *uid* is the client's authenticated identity, p is the permission to be exercised, and *cap* is a capability $\langle\langle sessid : cap(t_{ser}, F)\rangle\rangle_{?,?}$ for session *sessid*, such that $F = (\Sigma, Q, \Gamma, q_0, \delta)$ and *proof:* $\Gamma \rightarrow proofTree$.

Output: A set of tickets, or a failure response.

if $RS(p) \neq rsid$ **then return** *failure* ;
if $vid = rsid$ **then**
    Invoke Algorithm 3 locally on *rsid* to validate *cap* ;
    if *Algorithm 3 fails* **then return** *failure*;
else
    Request *vid* to run Algorithm 3 to validate *cap* ;
    if *Algorithm 3 succeeds* **then**
        vid sends *rsid* the contents of $ex[sessid]$;
        vid deletes its copy of $ex[sessid]$;
        rsid stores the received contents locally in $ex[sessid]$;
    else return *failure* ;
if $p \in \Sigma$ **then**
    $C' \leftarrow \mathsf{validityCheck}(proof)$;
    $C_{max} \leftarrow \mathsf{findMaximal}(Cond(q, p, C'))$;
    if $(q, p, C_{max}) \in dom(\delta)$ **then**
        Exercise permission p ;
        if $\delta(q, p, C_{max}) = q$ **then**
            return $\emptyset$;
        $t \leftarrow current\_time()$;
        $ex[sessid] \leftarrow ex(p, C_{max}, t, ex[sessid])$;
        if $\delta(q, p, C_{max}) = \circ$ **then**
            return $\{\langle\langle rsid, sessid : \mathsf{upd}(ex[sessid])\rangle\rangle_{k_{rsid}, uid}\}$
        else
            $F' \leftarrow (\Sigma, Q, \Gamma, \delta(q, p, C_{max}), \delta)$;
            return $\{\langle\langle rsid, sessid : \mathsf{cap}(t, F')\rangle\rangle_{k_{rsid}, uid}\}$
    else return *failure* ;
else return *failure* ;

4.5.6 Putting Everything Together

Next, let us look at an example, which follows the sequence of events from the time a new client is added to the infrastructure. I will explain two scenarios, first of which is visualized by Figure 4.1 and the second by Figure 4.2.

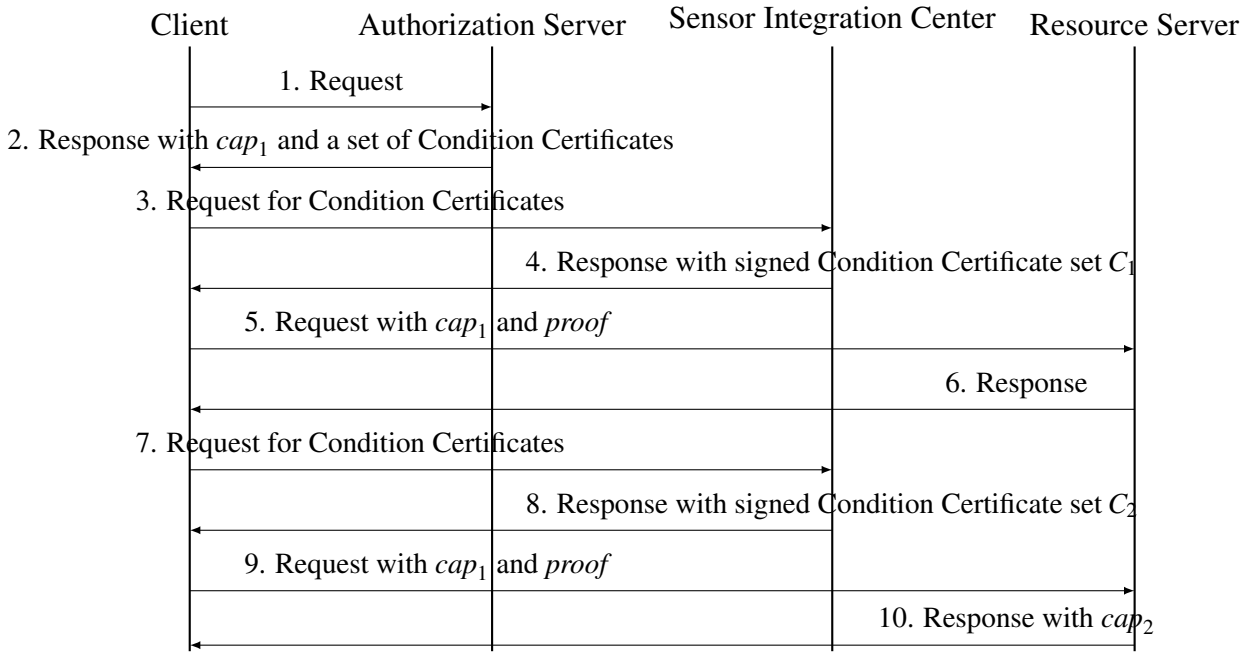

Figure 4.1: **Sequential exercise of permission with and without new capability**

Scenario 1.

Figure 4.1 shows the sequential series of events when a client needs access to a resource. In the figure, the first step is asking the authorization server for a ticket, the authorization server returns a capability cap_1 and an initial set of Condition Certificates. In step 3 of the figure, the client requests the SIC for a set of Condition Certificates of some conditions it needs, to get access to a resource. And in step 4, the SIC responds with a set of signed Condition Certificates. Step 3 and 4 might be recurring as the client may have to contact some other SICs for the completion of the proof.

In the next step, the client requests the resource server for accessing a resource and provides

capability cap_1 and *proof*. In this instance, the resource server grants access but does not return a new capability, so the client can use the old capability for further access.

Once the client needs to access another resource, it contacts SICs once more in steps 7 and 8 and receives Condition Certificate set C_2. These two steps are repeated until the client acquires enough Condition Certificate to form another proof. The client then requests a resource server with cap_1 and *proof*. The resource server grants the request and in this case creates a new capability cap_2 and provides it to the client. The client then can use this new capability for further access.

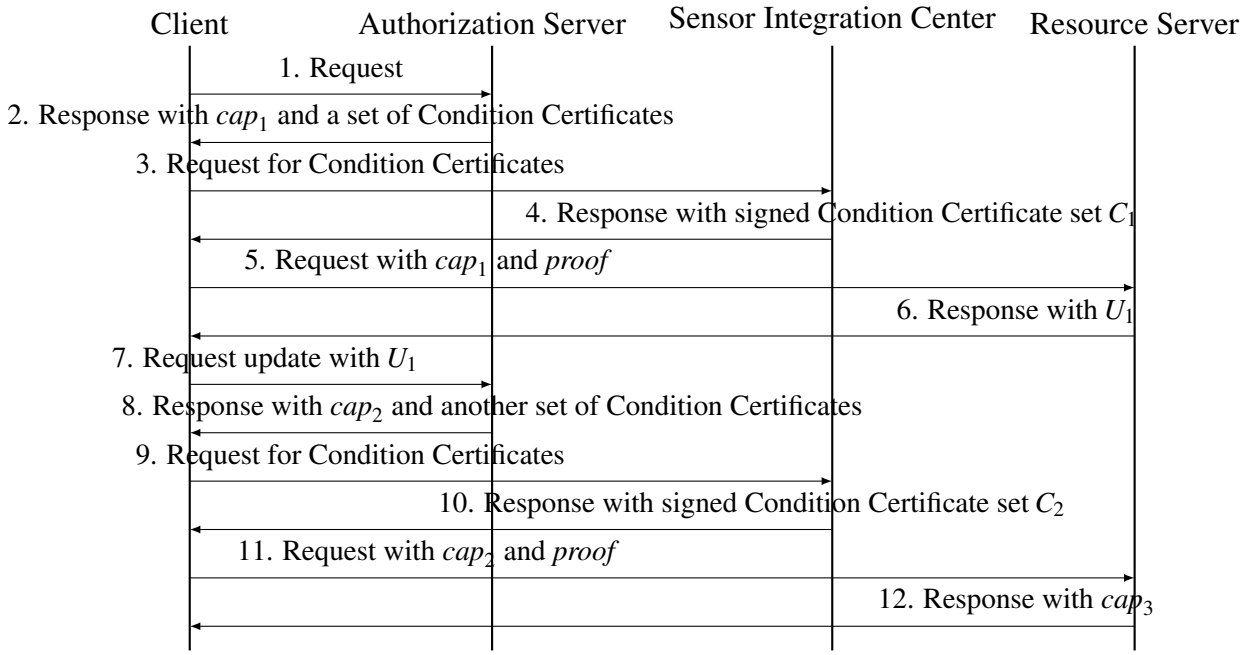

Figure 4.2: **Sequential exercise of permission with update requests**

Scenario 2.

Figure 4.2 is an example of the sequence of steps by a client who receives an update request. We see similar sequence of steps until step 4 just like scenario 1 visualized by Figure 4.1. In step 5, when the client requests for access, the resource server after giving access returns an update request U_1, meaning the resource server does not have enough information to construct a new capability. The client now has to gather a new capability and a new set of Condition Certificates from the authorization server, shown in steps 7 and 8. This is an extra communication that is done with the

authorization server. After the client gets a new capability and a new set of Condition Certificates, it can contact SICs again (steps 9 and 10) to complete proofs of conditions it requires to request permission. Steps 11 and 12 shows a request to the resource server and then the resource server replies with a new capability cap_3. For the next request, the client may use capability cap_3 to gain access.

Resource servers are generally memory constrained devices which can only hold a set amount of information. If it keeps on building, then the memory will be full. Therefore, the garbage collection method is introduced. The garbage collection method is the same as original HCAP explained in Section 2.2.4.

In this chapter, we discussed the different entities present in the context-aware history based capability system, what each entity is responsible for, what information each entity stores and an brief description on the algorithms they use. We ended with two examples, which summarizes the flow of connections between entities. In the next chapter we dive deeper into the technical details of what we intuitively presented in this chapter.

Chapter 5

Implementation

This chapter presents the implementation overview of the context-aware HCAP. I have built on top the work done by Tandon *et al.* in [54] and made my work as reusable software components. The Java programming language has been used to implement this work. Several third party Application Programmable Interfaces (APIs) were used to for this purpose.

This chapter goes into deep details about the implementation of Context-Aware History-Based Capability System. This is done so that any interested parties might be able to replicate this work consulting this chapter. This chapter goes into details of the internal workings of the Authorization Server, Resource Server, Client and the Sensor Integration Centre in such a way that there might be no issue if any parties may be interested in implementing this work on their own.

Section 5.1 goes over the background technologies that were used. Section 5.2 goes over the architecture of the context-aware HCAP system. The subsections under Section 5.2 goes over the Client API, Authorization Server API, Sensor Integration Centre API, and the Resource Server API one by one. Each subsection goes into details about the code of the implementation of the context-aware HCAP system.

5.1 Background Technologies Used

During the implementation of the context-aware HCAP, several third-party APIs were used. In order to successfully establish connections between different parties such as the client, the Resource Server, the SIC and the Authorization Server, CoAP protocol was used which provided a secure communication channel using encrypted network packets. CoAP protocol also has provisions for different interchangeable dataformats and different encoding mechanisms that can be used to transfer data between parties. The Condition Certificates are custom made and uses the Java Security

Signature which are all described in the subsections below.

A total of approximately 38 Java source files that were edited and 13 newly created Java source files created. Approximately $11,000$ lines of code were written and among them approximately 4000 lines are newly written and the rest are modified. The implementation consists approximately $17,900$ lines of code.

5.1.1 Californium-core API

The Californium-core API is a CoAP framework which is used to build Internet of Things applications. Californium is implemented using Java and it provides this API as a RESTful Web Service [44] and supports CoAP features. It has provisions for creating servers which the clients can communicate with, using an exposed endpoint which is bound to the IP address and the port of the server. The API also provides ways to make a client app which can send requests to the servers endpoint. Since Californium is a RESTful Web Service, it provides each resource in a server with a Uniform Resource Identifier (URI) and the clients can make requests to the servers directly.

5.1.2 Scandium API

Scandium is a Java implementation for *Datagram Transport Layer Security 1.2* for the Californium CoAP framework. It allows entities in the CoAP framework to securely communicate with one another and also has provisions for mutual authentication. To generate the keys for encryption and for the DTLS handshake, Scandium uses the Elliptic-Curve Diffie-Hellman algorithm. X.509 certificates are used for mutually authenticate one another. Scandium APIs also allows scripts to be used for generating our very own X.509 certificates which can be validates by a certification authority (CA), which can be created as well using Scandium.

5.1.3 Jackson-Dataformat JSON

In my implementation, I have used the Jackson library to serialize Java objects to JavaScript Object Notation (JSON) and vice-versa. Jackson is an open-source Java-based library and provides a fast,

clean and compact JSON encoding. I have used the `JSONFactory` class in the Jackson library to create an instance of the `ObjectMapper` class. The `ObjectMapper` class was used to serialize Java objects to a byte array which can then be sent over the network allowing communication. The same class is used on the receiving end to deserialize the byte array using a reader found in the `JSONFactory` class.

5.1.4 Jackson-Dataformat CBOR

I have also used Concise Binary Object Representation (CBOR) as a data interchange format. This is a variant of JSON and the Jackson library has support for this as well. The process is very similar to the the previous section but here the `CBORFactory` is used to create an instance of the `ObjectMapper` and the rest follows similar process as Section 5.1.3 to serialize and deserialize Java objects to a byte array and vice versa.

5.1.5 Java Security Signature SPI

I have a custom certificate which is used as a part of the proof of conditions when a client needs to request a permission to be executed. I have named that certificate Condition Certificate. There are three types of Condition Certificates as explained in Section 4.4.1. The `Encrypted String` field is present in all the types of Condition Certificate and it is the concatenation of the values of the other fields, signed by the private key of the issuing entity. The verification is done using the public key of the signing entity. Here, the `Signature` class provided by `java.security` is used to digitally sign and validate the concatenated data. This is done by the SICs so that the integrity of the Condition Certificate is not violated. The Resource Server uses the same class to verify the signature.

To sign data or to verify the signature, there are three steps. To sign the data I use `initSign`, `update` and the `sign` method in the `Signature` class. To verify the signature, I have used `initVerify`, `update` and the `verify` method in the same class. The signature algorithm that I have used is `SHA256WithECDSA`.

5.2 An Overview of the Architecture of the System

The architecture of the context-aware HCAP system is an extension of the implementation done by Tandon *et al.* [54]. The implementation of their work consisted of three distinct components. They are the Authorization Server, the Resource Server and the Client. I have one extra component named the Sensor Integration Centre. Although the components match with the previous work, there are some major differences in their operations, including the inception of the sensor integration centre.

The clients are the end users who need access to resources and there can be multiple clients. These clients randomly move around and contact respective entities to exercise permissions. The clients can contact different entities for different reasons and are able to operate the following operations.

1. Contacts the Authorization Server for the following reasons.
    (a) To obtain a new capability.
    (b) To obtain a new set of Condition Certificates (only Type-1 and Type-2 Condition Certificates can be obtained from the Authorization Server).
    (c) To update the internal state of the Authorization Server and in turn get a new capability.
    (d) To retrieve a lost capability.
2. Contacts the Resource Server for the following reasons.
    (a) To request access to a resource.
    (b) To retrieve a lost capability.
3. Contacts the sensor integration centres for the following reasons.

(a) To get a Condition Certificate in order to construct the proof for a condition.

The HCAP Client API is in place between the Californium library and the code written for the end users. The code for the context-aware HCAP protocols are written and are at work in the client side in some device, for example, a smartphone. The details of the client side code is written in Section 5.2.1. Although the client contacts the Authorization Server for multiple reasons, the frequency of contact are less in number when compared to the frequency of contact with the Resource Server and the Sensor Integration Centre. In this design, the Authorization Server is a central entity and there is only one Authorization Server within the organization. The client primarily contacts the SICs and the Resource Servers in a distributed manner.

The Authorization Server on the other hand uses the Californium library to host the context-aware HCAP authorization code at a specific port in the system. The Californium library is responsible for the implementation of the CoAP side of things, for example, receiving and responding the requests made by the client. The context-aware HCAP authorization code works out what to respond to which client. The Context-Aware HCAP Authorization Code works alongside the Californium library to form the context-aware HCAP Authorization Server API. The Authorization Server uses the Californium library to receive requests from the client or the Resource Server. After that the authorization code parses the request following the process mentioned in the previous chapter, resulting in decisions deemed appropriate by the authorization code. The Californium library is responsible for responding to the requester. The details of the Authorization Server code is discussed in Section 5.2.2. The Authorization Server only responds to requests from the client or the Resource Server.

The Sensor Integration Centre API is also written using the Californium library to host the SICs code at a specific port. The SIC only responds to clients requesting Condition Certificates. The SIC has the power to create new Condition Certificates as per the programming of an administrator. Therefore, it has to do some slight calculations and form an agreed upon data structure to

be returned to the client. The Condition Certificates are created using an internal administrative guidance, coded into the API. This is described in details in Section 5.2.3.

The Resource Server is the entity which serves access decisions to the clients. In a distributed authorization scheme such as this, the Resource Server does not contact the Authorization Server for access decisions. The Resource Server checks the validity of the capability and also the proof provided by the client and provides access decisions as per the authorization code. The authorization code for the Resource Server is written within the Californium Library, using an interface in the network stack of the Californium library called `MessageDeliverer`. In this way, the access control mechanism is enforced within the Californium library. All the checks of validity are done before the request leaves the network stack of the Californium library. The Resource Server also has to create new capabilities and present it to the client. All these activities are detailed in Section 5.2.4.

5.2.1 Client API

The end users of context-aware HCAP are clients who use the Client API. The Client API sits between the end user code and the Californium library. The functions and operations that are available are described as follows.

- **Client Initialization**

  The client is initialized using the `ClientBuilder` class in the Client API. It contains all the variables and the methods to run the Client API. There is a client properties file which hold some information regarding some configuration details. The properties include information about the data format being used, if its JSON or CBOR, the Authorization Server address and port, the Resource Servers addresses, also information regarding the X.509 certificates being used for mutual authentication and also the location of a trusted authority. This properties file is usually readable by humans and can be modified using any text editor for the purposes for

easier comprehension and easier editing should it need modification.

The constructor for the `ClientBuilder` class takes in the location of the properties file as an argument. The sequence of steps the constructor does are as follows.

1. Parses the properties file.
2. Stores the attributes read from the properties file to appropriate local variables.
3. Creates secure communication channels by building DTLS connectors with the help of the information specified in the properties file, specifically, using a trust store where X.509 certificates of the servers are stored.
4. The constructor also creates a CoAP client using a built-in class called `CoapClient` in the Californium library. The URIs of the authorization and the Resource Servers are also fed as parameters. Using this CoAP client, the client would be able to send and receive requests.
5. Sets up the endpoint for this client using a DTLS connector from the Scandium API.

- **Connections with the Authorization Server**

  The client communicates with the Authorization Server for many reasons described previously. The following are each connections the client is capable of making with the Authorization Server.

  1. Getting a capability.
  2. Recovering a capability.
  3. State Update requests.

4. Getting updated Condition Certificates.

Let us look into these communications in details.

Getting a Capability. The client needs to contact the Authorization Server as a first point of contact to gain a capability and some Condition Certificates in relation to the capability. This capability is used to gain access to resources from the Resource Server. In order to gain a capability, the client contacts the Authorization Server using the `getCapability` method located in `ClientBuilder` class. The `Request` class from the Californium API is used to construct a request to be sent to the Authorization server. As instructed by the Californium API documentation, the modes of request can be of types GET, POST, PUT or DELETE. In this case, *GET* was used while creating an instance of the class. Next the URI is set in accordance [1, Section 6]. The URI is a superset of the much used URLs [6]. The client needs to form each URI in accordance with what resource it wants to access. Regarding getting a capability the client needs to request for `issueNew` resource from the Authorization Server. That resource on the Authorization Server end is responsible for creating a capability and giving to the client. Therefore, the URI setup is very important. A URI in this scheme has three parts, namely, the IP address of the server, the port number and the location of the resource that the client is trying to access. Through this communication, the client does not attach a payload. The Authorization Server checks several things which is described in the Authorization Server API (Section 5.2.2), and either returns null if the evaluation of the request is negative, otherwise sends a **capability** and **a set of Condition Certificates**. The client stores the capability in a data structure in the `capabilityStore` class and the Condition Certificates are stored in the `proofMap` data structure in the `CertificateStore` class. These are later used during requesting to exercise a permission from the Resource Server.

State Update Requests. After a client requests to exercise a permission to the Resource Server, the Resource Server might respond with a state update, meaning, either the current capability does not have enough information to create a new capability, or maybe a process called garbage collection was triggered. In either case, the client needs to contact the Authorization Server and provide the state update request in order to gain the next capability. The `sendUpdateRequestAndGetCapability` method found in the `ClientBuilder` class does this job. To use this, the state update request received from the Resource Server must be passed as a parameter to this method. The associated URI for this points to a resource of `update` in the Authorization Server. What the Authorization Server does after receiving this is left for later when we talk about the Authorization Server API. Once the Authorization Server processes the request and goes through the necessary algorithms, the Authorization Server sends back a new capability and a new set of Condition Certificates. The client uses this new capability as the current capability. For the new set of Condition Certificates, it updates the `proofMap` data structure in the `CertificateStore` using the `updateProofMap` method in the same class.

Updates Condition Certificates. After a considerable amount of time, the original Condition Certificates provided by the Authorization Server will expire. When that happens, the client needs to request the Authorization Server for a new set of Condition Certificates. Usually, the Condition Certificates provided by the Authorization Server has a considerably large validity period, so this communication is very much less. This event may be triggered before the client requests to exercise a permission, when collecting the proofs. A proof is a map mapping a condition to a proof tree represented by the data structure `proofMap` in the `CertificateStore`. So before the client tries to collect Condition Certificates for specific conditions, it

checks every mapping in the `proofMap`; if the first Condition Certificate of any condition is invalid, that means the Condition Certificate from the Authorization Server for that condition is invalid. At that time, updating the Condition Certificates gets triggered and the `proofMap` data structure is iterated and all the Condition Certificates from the authorization server is checked for validity. The method `updateCertificateFromAuth` is called with all the invalid conditions as parameters. This method sends a request to the `updateCertificate` resource in the Authorization Server with the latest capability the client has and the list of invaid conditions as payload. The Authorization Server extracts all the conditions from the capability and checks if the conditions in the list sent are present in the capability. Then responds with the Condition Certificates corresponding to the conditions. These Condition Certificates are then added to the `proofMap` and the next SICs are contacted to complete the proof. A proof is completed when type-3 Condition Certificates for all the required conditions are found.

Recovering a Lost Capability. A client may lose the current capability that was given, at any time. In that case, there are processes in place to recover said capability. This process works much like the process used in the original HCAP [54]. The client takes two actions in order to retrieve a lost capability. Firstly, it contacts the Authorization Server using the `reIssueCapabilityFromAuthServer` in the `ClientBuilder` class. The client makes a request to the `issueLost` resource of the Authorization Server with the session ID as payload. The Authorization Server evaluates the request and if it is a valid request, responds with a capability with the last known state as the start state. But, that may not be the most recent state of the client as the Authorization Server not always knows the most recent state. That is why the client has to take another action, that is, to contact the Resource Server, which we will discuss in the next part.

- **Connections with the Resource Server**

  The client communicates with the Resource Server for two reasons mentioned in Section 5.2. Resource servers are primarily in charge of resources and the clients need to go to them every time it needs to exercise a permission. A huge numbers of clients using context-aware HCAP will randomly contact random Resource Servers for access to resources, therefore, there are a lot of communications with the Resource Servers. The primary task of the Resource Server is to grant or reject access to resources, a minor secondary task is to provide assistance in recovering a lost capability by the client. Let us look at the two communications into details.

  Requesting Access. The client requests to exercise a permission from the Resource Server using the `requestAccess` method found in the `ClientBuilder` class. In context-aware HCAP system, the client has to gather relevant Condition Certificates before contacting the Resource Server to exercise a permission. Let us consider that the relevant proofs for conditions have been gathered by contacting relevant SICs and are stored in an appropriate data structure in the `CertificateStore` class. The process of gathering Condition Certificates from the SIC is discussed in a later section. The `requestAccess` method is responsible for requesting access to resources. The method in question takes in five parameters. The parameters are: 1. a capability received from a trusted entity (Authorization Server or Resource Server), 2. a payload to be included with the request, 3. a resource id of the resource that the client wants to access, 4. a set of condition names and 5. an instance of `CertificateStore` in order to construct proof. The first parameter, **capability**, is used to evaluate if the client is authorized to access that resource and under what conditions. The second parameter, **payload** is something that the client may want to send to the Resource Server. The **resource** is a string, which points to a resource ID. This string is used to make the URI, because any resource the client

may want to access to, exists as a resource in the Resource Server accessible in the form of a URI. The fourth parameter is a **set of condition names** whose proof the client chose to present to the Resource Server. The client may have several transitions with the same permission but differ in requirement of proofs of conditions. The client can choose to present whichever proof of conditions they desire here as it does it violate the security principle described in Section 3.3. The last parameter is an **instance of a `CertificateStore` class** which contains the proofs for each condition in the previous parameter.

The `requestAccess` method first extracts the proofs for the conditions in the fourth parameter from the data structure `proofMap` in the `CertificateStore` instance. In practice, not all proofs for all conditions are necessary for all requests all the time. The proof for the conditions in the fourth parameter are gathered from the `CertificateStore` and a proof map is generated, which is then used in the request. Then the method puts the capability and the newly created proof map in the payload to the request made to the Resource Server. The `Request` class from the Californium API is used to form the request, with the Resource Server URI and the resource name concatenated together with a "\" in between. Depending on the Resource Server, the Resource Server can respond to the request in four ways, they are, 1. access granted with a new capability, 2. access granted with no new capability, 3. access granted with a state update request, and 4. rejected access request. The new capability may be used for further access. When no new capabilities are sent back from the Resource Server, the client may use the previous capability for further access. This is because no state change occurred and no new capability was needed. The state update request is created by the Resource Server to be presented to the Authorization Server in order to get the most recent capability with a new state.

Recovering a Lost Capability. This is in continuation of the recovering a lost capability mentioned previously. After the client gets a capability from the Authorization Server, the client presents that capability to the Resource Server. The client contacts the `recover` resource of the Resource Server. The Resource Server calculates the current state by referring to the exception list and then provides a new capability which shall be the current capability. The client then can go about exercising permissions using this new capability.

- **Connections with the Sensor Integration Centre**

  The client communicates with the sensor integration centres to gather Condition Certificates to eventually form the proof of a condition. This is the only reason the clients communicate with the sensor integration centre. The client may contact several SICs to form a proof for a condition and has to do this for multiple conditions. In this part I will explain this process in details.

  Requesting Condition Certificates. The client needs to request several SICs to gather Condition Certificates for conditions and form a complete proof. The client can either record all the Condition Certificates gathered from the SICs, or choose to delete the gathered Condition Certificates after exercising a permission as explained in Section 4.5.4. A client may choose to delete the Condition Certificates if the client device is memory constrained and is bloated with a lot of Condition Certificates. This is a feature which one can turn off or on depending on preferences. When the client turns off the feature, the collected Condition Certificates are deleted except for the Condition Certificates provided by the Authorization Server.

  In order to connect to the Sensor Integration Centre the client uses the `requestCertificates` method when the feature is turned off and `requestCertificateSaved` when the feature is turned on. Both of these methods exist in the `ClientBuilder` class. Both methods take in a set of strings representing the

conditions as parameters. These are the conditions that require a proof for a specific request. These two methods are called from the `requestAccess` method of the same class. To process the Condition Certificates and store the Condition Certificates the `proofBuilder` class is used here whose primary goal is to store the Condition Certificates in the `proofMap` data structure. It also helps to figure out which proof is complete and to keep a track of all the remaining proofs for conditions. Let us look into both the `requestCertificate` method and the `requestCertificateSaved` method in the next paragraphs.

In the `requestCertificate` method, the `proofBuilder` class is initialized with two parameters. Firstly, the conditions that need to be fulfilled before requesting access and secondly, the Condition Certificates provided by the Authorization Server are stored in two distinct data structures within the class. The `requestCertificate` method goes on to run a while loop in `isProofComplete` method in the `proofBuilder` class. This loop will run until all the conditions have found a type-3 Condition Certificate. Next, using the `getConditions` method in the `proofBuilder` class, the client will gather the remaining conditions whose proofs are incomplete. Extracting the latest valid Condition Certificate for each of the condition from data structure `proofMap` in `proofBuilder` class, the method groups the requests by which Sensor Integration Centre it needs to contact. The client starts requesting the respective Sensor Integration Centres using the "getConditionCertificate" resource of the Sensor Integration Centres. All the Sensor Integration Centres have the same resource as they do not have any other utilities. Once the response is provided, the `processCertificate` method in the `proofBuilder` class is used to evaluate the Condition Certificates and then store in the `proofMap` data structure. If the Condition Certificate is type-3 for a condition, then they are taken off the needed list of proofs. If all type-3 Condition

Certificates are found, then the loop is broken and the proof map is returned. The client goes on to provide this proof map in the payload to the Resource Server while asking to exercise a permission. After the permission is exercised, the client does not store the gathered Condition Certificate. The only Condition Certificate stored is from the Authorization Server.

When the storing the Condition Certificate feature is turned on, the `requestCertificatesSaved` method in the `ClientBuilder` class is utilized. This method takes in a set of conditions to be fulfilled and an instance of the `CertificateStore` class. Next, the relevant Condition Certificates are derived from the `CertificateStore` class. The Condition Certificates are stored in a chronological list in the `CertificateStore` class. The Condition Certificate list is evaluated chronologically. While the chronological check, the first Condition Certificate which has surpassed its validity period along with every subsequent Condition Certificate is deleted. After that, the `proofBuilder` class is initialized with the valid proofs and the list of incomplete conditions. The next steps are the same as the previous paragraphs. After the client gathers all the proofs, the `CapabilityStore` is updated. The client then uses the `CertificateStore` class to form a proof map and send it as a payload when trying to exercise a permission. So, we can see that the `CapabilityStore` class stores the Condition Certificates when the feature is turned on and evaluates them before requesting to exercise a permission.

These are the tasks the client can perform. It was designed so that the client does not have to do a lot of computation and has the option to store or not store the gathered Condition Certificates. The clients can seamlessly communicate with all the other entities for different reasons. Next we look at the Authorization Server API and the operations the Authorization Server can perform.

5.2.2 Authorization Server API

The Authorization Server is a central entity which services the client and enforces protocols set out by the administrators. The Authorization Server API is in use by the administrators of the system and is built using the Californium and Scandium APIs just like the other entities. The Authorization Server is hosted on a specific IP address and port number. The clients and the Resource Servers contacts the Authorization Server with different requests. The API refers to the authorization code and provides responses to the requests as per the authorization code written up by an administrator.

The Authorization Server is in charge of the following activities.

1. Providing new capabilities and Condition Certificates to new clients.
2. Providing an updated capability with updated Condition Certificates.
3. Helping form lost capabilities.
4. Providing updated Condition Certificates.
5. Executing garbage collection.

The design of the Authorization Server API is based on the REST architecture [48] that it follows to interact with the Resource Server and the client. Let us look into some details of the Authorization Server API.

- **Server Initialization**

  The Authorization Server need to be initialized and hosted at an ip address and a port. An administrator is responsible to set up the Authorization Server. The API uses the `HCAPAuthorizationServer` Java class to initialize and setup hosting. This class contains some important data structures necessary for the smooth operation of the context-aware HCAP system including several keystore locations and passwords in order to access the X.509 certificates to open encrypted communication channels between entities, several maps storing client, sessions, CASAs

and so on. Let us look into some of the important data structures and go over them in brief.

- *SessionMap.* A map mapping session ids to CASAs, where CASAs are an instance of a Java class named CASA. With every new session created, a new entry is added to the session map.
- *UserMap.* A map mapping session ids to users as objects in the form of `ManageUser` Java class. `ManageUser` classes are used to represent clients, i.e., each client that gets added to the system, an instance of this class is initialized and added to the UserMap data structure. `ManageUser` class stores several client information such as the name, current state, the time the client entered into the current state etc. It also contains a map mapping each state to its corresponding fragment. This saves time when the Authorization Server has to make a new fragment for the client. So the every possible fragment is pre-made in order to facilitate the process.
- *SharedSecretMap.* A map mapping Resource Server ids to their respective shared secrets. Since a capability needs to be hashed with a shared secret between the Authorization Server and a Resource Server, the Authorization Server needs to know each shared secret for each Resource Server.
- *StateRSMap* A map mapping each state to a Resource Server. Using this the Authorization Servers make the capability and then use the Resource Server information to hash the capability.
- *Keys.* This is a set of string type data structures. There are 4 strings: locations of the trustStore and the keyStore and their passwords.
- *Port.* The port at which to hold the Authorization Server.

The constructor for the `HCAPAuthorizationServer` class takes in a properties file location as a parameter. The properties file contains information regarding the location of the key store, the trust store and their passwords, the port number and if the data exchange follows CBOR or JSON.

The next method we need to look at is the `startHCAPServer` method in the same class. This method is used to start the Authorization Sever. First of all, the method sets up the keystore which houses the certificates which allows encrypted communication with the client and the Resource Servers. Secondly, a DTLS connector is instantiated which helps set up the Authorization Server identity and sources the keystore and the trusted certificates. Next, the method makes use of the `CoapServer` class from the Californium API with the DTLS connector and starts the server at an endpoint binding it to the IP address of the machine and a port number retrieved from the properties file.

The administrator can set up CASAs for clients using the `casaGenerator` class. There are a few ways for an administrator to specify CASAs for clients. First of all, the administrator can make use of the `state` and `casa` classes and code in each CASA. This may be done for smaller CASAs. The `casa` class is a blueprint of a CASA which when initialized, represents a CASA. Secondly, an administrator may generate XML files defining the permissions, states, conditions and the transitions of a CASA in a specific structure which can then be read and converted to instances of `casa` classes. CASAs can be either deterministic or non-deterministic in nature. To that end, the `casa` class also contains a deterministic boolean variable, and a `CASAConverter` class. The boolean variable tells if the CASA is deterministic or not and the `CASAConverter` class takes in an instance of the `casa` class as a parameter of the constructor. The `convert` method converts that CASA into a deterministic CASA and returns a casa type object which is the converted

deterministic CASA.

For the purposes of the implementation, I have created a CASA generator, which creates CASAs at random using the class `RandomCasa`. This instance of a deterministic CASA would go as a parameter to the `addClientStateMachine` method in the `HCAPAuthorizationServer` class along with a client id and a fragment length as a parameter. The client id must match the client identity information in the X.509 Certificates in the trust framework. The **fragment length** points to the amount of states to be put in a capability. The number of states to be put in the capability are all the states reachable from the start state. The `addClientStateMachine` stores the CASA for the client and the fragment length by mapping the client id to this pair and storing them in the variable `SAMap`. Fragments are created using a `Fragment` class before they are put in a capability. This is touched on at a later section.

- **`issueNew` Resource**

  This resource is a part of the Authorization Server code. When the clients contact the Authorization Server for the first time, this piece of code is run. The task of the Authorization Server is to create a first capability by looking at the DCASA map created beforehand. This resource overrides the `handleGET` method of the `CoapResource` from the Californium API. The Authorization Server then makes use of the `HCAPProcessExchange` class to parse the request made by the client and extract information about the client including identification information. The client ID is extracted from the X.509 Certificate that the client used to authenticate themself. Using that information, the CASA for the client is extracted from the `CASAMap` in the `HCAPAuthorizationServer` class.

  In order to build the capability, the `HCAPRequestHandler` class is used. The payload from the client, the client ID and the fragment length is passed as pa-

rameters to the class. The payload is null for the first time the client requests the Authorization Server for a capability. This will come in handy once the client makes an update request. If the payload is null, the Authorization Server calls `generateFirstCapability` method in the `HCAPRequestHandler` class. The first thing the method does is converts the CASA. The CASA is run through a conversion algorithm to convert the CASA to a DCASA. This is achieved through the `convert` method in the `CASAConverter` class. Using the new DCASA, all possible fragments are created with a set number of states in each fragment and the `fragmentMap` is populated in the `HCAPAuthorizationServer`. This is done so that fragments do not have to be re-made every time client comes to the Authorization Server with an update request.

The `ManageUser` object of the client is recalled and the current state and the entry time is set. The current state is extracted from the CASA which is the initial state. The data types `userMap` and the `sessionMap` are maps which are populated with the session ID mapping to a `ManageUser` object and a CASA object respectively. These data types are found in the `HCAPAuthorizationServer` class. Finally to create the capability, `CreateCapability` class is initialized by passing Resource Server ID, session ID, current state, time of entry to the current state and the fragment map as parameters. Using the `returnCap` method, a capability along with the initial Condition Certificates are created and now can be used to return to the client as response to the issue new request as the payload. The payload is basically a map with two items. The first item is the string "capability" mapping to the capability, the second item is the string "certificates" with a list of Condition Certificates. This will go as the payload to the client. This payload in the form of a map is then encoded in either CBOR or JSON and sent back to the client.

- **`update` Resource**

This resource is used when the client needs an updated capability from the Authorization Server. This updates the state of the CASA for the session of this client. The client presents a state update request to the Authorization Server in the payload of the update resource request. The Authorization Server decodes the update request and then checks the hash using the shared secret of the Resource Server who signed off on that state update request.

The update request is essentially the exception list containing all the permission condition pairs that were accepted by the Resource Servers. To execute this request, the `HCAPRequestHandler` class is initiated with the update request and client ID. The `handleUpdateRequest` method is used which updates the state. The method applies the permission condition pairs to the known state chronologically to arrive at a new state and updates the state known by the reference monitor. Then the `generateCapability` is used to create a capability using the fragment pre-created whose start state is this new state in addition to the relevant Condition Certificates. The newly made payload is then sent back to the client after encoding it as per the data interchange format.

- **`issueLost` Resource**

  The issueLost resource is another resource in the Authorization Server API. Due to whatever reason a client loses the current capability, the client needs to contact the Authorization Server for a new capability and Condition Certificates. The Authorization Server maintains a DCASA and the last known state for each client session. The Authorization Server may not know the latest state the client is in. Therefore the process is twofold and this contacting the Authorization Server is the first step. The Authorization Server overrides the `handlePOST` method in the `CoapResource` class and uses the `HCAPProcessExchange` to get the client ID. It also extracts the session ID which the client needs to send as a pay-

load to the request. The client ID and the session ID is passed as parameters to the `HCAPRequestHandler` class and the `generateCapabilityLost` method is used to generate a new capability to be sent back to the client after hashing and encoding it with the data interchange format agreed upon beforehand. The client then takes that capability and further contacts the Resource Server for the most updated capability. The part of the Resource Server is explained at a later section.

- **`flush` Resource**

  This resource is used when the Resource Server wants to perform garbage collection. This part of the Authorization Server API overrides the `handlePOST` method in the `CoapResource` class of the Californium API just like all the other resources provided by the Authorization Server API. The Resource Server must send some data in the payload consisting of an exception map and a baton map. The keys in the exception map and the baton map correspond to session IDs of clients and the values correspond to exception list and a boolean value pointing to if the Resource Server holds the baton for that session respectively.

  After the payload is extracted and decoded, the authorization starts processing the information. It processes each key-value pair in the exception map. For each session ID, the Authorization Server gets the exception list and calculates the current state of the client in this session with the help of the CASA stored for this session. It does not calculate or update the state stored by the Authorization Server right away. The updating of the state requires a lock to be put on the Authorization Server for that session. This is done to avoid a race condition [30] whereby a client may have a state update request and ask the Authorization Server for a capability, while the Authorization Server is trying to update the last known state for the session. This internal variable of current state of the CASA, if changed in a disordered manner may lead to bugs and must be only changed in a critical section-like manner. There-

fore, `sessionLockMap` is introduced, which maps a session to a boolean value. Whenever the client arrives with a state update request, it must check if the session is locked.

Finally, once the Authorization Server locks the session, it calculates the current state of the CASA in this session by traversing through the permission and accepted condition pairs. Then updates it in the internal variable maintained in order to keep track of the current state in an instance of `ManageUsers` class representing the client. After doing so, sets the baton to false in the same instance of client representation.[1] The Authorization Server proceeds to unlock the session and moves on to the next instance of exception list from the exception map. As soon as the Authorization Server completes it for all the pairs in the exception map, it sends back a success notification to the Resource Server and the resource can restart its other activities.

When the client comes to the Resource Server to exercise a permission, it checks to see that no Resource Server holds the baton for the client in this session by contacting the Authorization Server only to see the baton flag was set to false. That means, flush resource was used. The Resource Server then initializes a new exception list for that client in that session.

- **`confirmation` Resource**

  This resource is used by the Resource Server to contact the Authorization Server in order to validate a capability. This resource is used when the Resource Server

[1] A baton [54] is an indication as to which Resource Server is keeping track of the exception list for a specific session. Multiple Resource Servers cannot hold the same exception list. When flush resource is used by a Resource Server, a baton map within the Authorization Server API is used to keep track of the flushed exception lists as the Resource Server deletes all the exception lists. The boolean values are set to `false` whose keys correspond to the session IDs.

receives a request from a client and the Resource Server does not hold the baton for that session. The Resource Server requests the Authorization Server to confirm two things using the `confirmation` resource and must send session ID and the serial number of the received capability in the payload of the request. The Authorization Server overrides `handlePOST` method of the `CoapResource` to receive this request and then proceeds to decode it. After decoding the Authorization Server checks the following two conditions.

1. Checks baton map for the session ID sent. It is required to be false.
2. Checks if the serial number of the capability is the same value as the entry time for that client in that session to the current state.

If any of the conditions are false then the Authorization Server responds failure in the form of a flag being set to `false`. Otherwise, the flag is set to `true`. The Authorization Server responds with the flag as payload.

- **`ping` Resource**

  The clients, the SIC and the Resource Servers can ping the Authorization Server. The first contact the Resource Server and the clients make is with the Authorization Server. The `ping` resource is used in the beginning so that the clients, SIC and the Resource Server can mutually authenticate one another. By mutually authenticating one another, the client, the SIC and the Resource Server establishes a secured connection with the Authorization Server and is represented by a session. The HCAP API is used to establish a session with the Authorization Server and the connection is usually open 30 minutes after the first ping is executed.

- **`parent` Resource**

  This resource acts as a parent resource to all the other resources available in the Authorization Server API. The default resources available to the Authorization Server

are all available under a single umbrella and that is the resource named “hcap”. Due to this, hcap needs to be written as a prefix to all the URIs of the resources. For example, for issuing a new resource, the URI should be ‘`.../hcap/issueNew`’.

These are the resources available in the Authorization Server. These resources are then attached to the hcap Authorization Server using the add method in the `CoapServer` class in the Californium library.

5.2.3 Sensor Integration Centre API

The Sensor Integration Centre API is needed by the clients to access Condition Certificates which are needed to form proof as a part of an authorization request to the Resource Server. The Sensor Integration Centre code has been written to be run within the Californium Library. The SIC works as a Condition Certificate provider and its only task is to provide Condition Certificates to the clients. The SIC does not talk to the Resource Server or the Authorization Server, thus providing a large degree of decentralization.

The SIC has access to readings for several sensors and has the power to evaluate the request of a client. The objective of this work is not centered around the actual reading measure of sensors, therefore, the SICs are designed to usually return a signed Condition Certificate. It has provisions for checking if the sensor readings are correct or incorrect. Depending on the result, it has the power to reject the request for Condition Certificates.

On the other hand, the type of Condition Certificate provided is dependent on the admins of the certain SIC and is subject to the whimsy of the administrators. The administrators may decide to use any type of Condition Certificates as a response to a request made by the client. Although, type-3 Condition Certificates may be only issued if the SIC has access to the sensor reading in question. The SIC is contacted only by the client. We assume that the SIC is a part of the public key infrastructure that the rest of the entities are operating on and can verify the public key of all the clients in order to communicate via an encrypted channel. There are only two resources

working under the HCAP parent resource and room for more.

Let us take a deeper dive in the HCAP Sensor Integration Centre API.

- **Initialization**

  In order to start the SIC, the `HCAPSensorIntegrationCentre` class is initiated. The `CoapServer` class from the Californium API is used to start the server. Upon initialization of the `HCAPSensorIntegrationCentre` class, a properties file is read just like the Authorization Server API and the client API. The properties file has the information of the trust store which is necessary to perform encrypted communication with all the clients. It also stores information on which port to host the SIC. It is the responsibility of the administrator of the SIC to ensure the proper information is available in the properties file.

- **`ConditionCertificate` class**

  The `ConditionCertificate` class acts as a blueprint of the Condition Certificates. This class has variables such as `condition`, `location`, `issuer_id`, `encryptedString` as Strings, `certificate` as a Certificate type from the Java Security package, `type` as an integer, `startTime` and `endTime` as long type data structure and an instance of a `PrivateKey` from the Java Security package.

  The SIC can choose to create instances of this `ConditionCertificate` class and then can use `getConditionCertificate` method in the class to get a Condition Certificate to be sent to the client to be used. The SIC may use the constructor in the class to form the Condition Certificates. The `encrypt` method does the encryption for this custom certificate. Depending on the type of Condition Certificate the method concatenates some of the variables and signs them using the private key and stores the signed string to the variable `encryptedString`. This

is done in the `encrypt` method in this class. Finally, the `getCondition-Certificate` method is used to construct a map with some of the important variables including the condition, issuer, type, startTime, endTime and the encrypted string.

- **Processing Requests from the Client**

  The SIC processes the client request using `HCAPMessageDeliverer` class. The `MessageDeliverer` class is an interface found in the Californium library and I have designed a custom message deliverer named `HCAPMessageDeliverer` and its primary duty is to check if the request can be processed at the current time and then passes them on to the resource requested. The SIC has only two resources, `PING` and `getConditionCertificates`. The request goes through the network stack of Californium and then reaches this class. The ping request operates similar to the Authorization Server ping so it will not be discussed further. On the other hand, if the client requests `getConditionCertificates` resource, the `HCAPAuthorize` class is initiated with the user id and the payload of the request as parameters. Then the `evaluateRequest` method is called. This method firstly, checks all the condition requested and the types requested by the client. Then it refers to the administrative guidelines on which type Condition Certificates it can provide.

  The SIC then generates the Condition Certificates and signs them using the `encrypt` method in the `ConditionCertificate` class. Once the SIC has generated all the Condition Certificates, it puts them in a HashMap mapping the condition to the Condition Certificate. This HashMap is then sent back to the client as a response to the `getConditionCertificate` request. Now, the client uses these Condition Certificates to form a proof tree which was discussed earlier.

The SIC does not do any form of computation and is only tasked with one task, to serve the

requests made by the client. It does not communicate with the Resource Servers, Authorization Server or other SICs. It only refers to the administrator directives and provides Condition Certificates to the clients accordingly.

5.2.4 Resource Server API

The Resource Server API allows the vendors of IoT devices to create context-aware HCAP access control features to a Resource Server running on a Californium framework. The API is within the Californium Library by design.

The Californium framework provides a message deliverer which sits in between the Californium network stack and the resources under the Resource Server. In this design of context-aware HCAP, access control is done by first checking the validity of the capability presented by the client and then checking the Condition Certificates provided by the client. Only after the capability and the proof consisting of Condition Certificates are checked does the Resource Server allows a permission to be executed on a resource. The message deliverer is the perfect place to put custom code because it sits between the Californium network stack and the resource. Even before getting out of the network stack, the context-aware HCAP access control mechanism takes effect.

The Californium framework contains a default message deliverer called `ServerMessageDeliverer` whose task is to deliver a request to a resource. On the other hand, the Californium library provides an interface called `MessageDeliverer`. Using this interface, I have designed my own `HCAPMessageDeliverer` class. Elements of this class has been reused from the work done by Tandon *et al.* [54] following the principles of code re-usability [57]. If the context-aware HCAP protocol is used to secure the server, this message deliverer class must be used in the Resource Server.

The client requests the Resource Server to execute a permission and presents a capability and a proof consisting of Condition Certificates. The request goes through the Californium's network stack and then reaches the message deliverer. Here, the proof is checked and then the capability is validated. It is then decided if the Resource Server should allow the IoT device to perform the

action requested by the client. If the request is rejected, the Resource Server sends back a canceled response to the client. There are two events which the message deliverer is responsible for: garbage collection and baton compression. These are also reused code which were reformed to account for the "context-awareness" of the HCAP protocol.

Besides the `HCAPMessageDeliverer` class which acts in lieu of the default message deliverer of the Resource Server, there is an extra resources of the context-aware HCAP, which is a validation resource. The validation resource is necessary when we have multiple Resource Servers. We will discuss this is a later section in details. Let is look into the inner workings of the HCAP Resource Server API in details in this next section.

- **Initialization**

  The user of this system needs to first initialize the Resource Server using the `Coap Server` class provided by the Californium library to create the server and host it at a specific IP address and a port. This is different from the Authorization Server's API in the sense that the Authorization Server API is integrated with the Californium framework, but here the user needs to include the Resource Server API code before starting the server. The user needs to write some code to create an instance of the `HCAPMessageDeliverer` class and make it the default message deliverer instead of the library provided deliverer. This is done using the `setMessageDeliverer` method provided in the `CoapServer` class.

  Before we initialize the Resource Server, we need to create some demo resources. This is done in order to demonstrate the workings of the server. Actual IoT devices were not used. Ideally we would have an Administrator API in place which would create a permission-reference map which would signify which permission points to which resource and which operation. Since there is no Administrator API, I have created some demo resources to render the inner workings using `Construct-ResourcesTestMachine` class. Then a map is created mapping a CoAP method-

Resource ID pair to a permission ID. This map is called a permission map. The CoAP method is the type of request: GET, POST, DELETE or PUT. The Resource ID is a part of the request URI which points to the resource a client may want to access. Next, the `HCAPResourceServer` is initiated with the permission map and the location of a properties file. This class extracts information of the trust store and the keystore to communicate with the clients and the Authorization Server properly and stores the permission map within the class. The properties file also has information pertaining to the garbage collection and the baton compression. Finally, the `startHCAPServer` method is called to start the server.

- **HCAP Message Deliverer**

  The `HCAPMessageDeliverer` class is a crucial part of the context-aware HCAP protocol and every request made by the client passes through this class in order to execute a permission. The class has two constructors accepting a `Resource` type and a `Map` type. The second constructor has one extra parameter, that is an instance of `MessageDeliverer` class. This third parameter is necessary if the user wants to add any other functions after the context-aware HCAP protocol has made all its checks. The **resource** type is the root resource of the server containing all the demo resources that were created in the initialization phase. The **map** is the permission map. The third parameter is in case a user wants to add a custom message deliverer. The user would have to create a new class implementing `MessageDeliverer` from the Californium library and the pass this as a third argument.

  After the request from a client reaches `HCAPMessageDeliverer`, the method: `deliverRequest` is called. First, a **current time** is recorded. Then this method parses the request to obtain the CoAP method and the resource ID and refers to the permission map to get the permission ID. It also extracts the user ID and the payload. Then it passes the permission ID, user ID and the payload as parame-

ters to the constructor of the `HCAPAuthorize` class. Let us look into what the `HCAPAuthorize` class does next.

`HCAPAuthorize` class. This class is used to check the validity of the capability, if the permission requested is a valid request and if the proof provided by the client is legitimate. After its construction, the `HCAPMessageDeliverer` calls the `evaluateRequest` method. Let us look into work of `evaluateRequest` method in the following.

1. The first task of the method is to extract the capability from the payload and validate it. This is done using the `runValidationAlgorithm` method. The validation of the capability is done using the validator's identity or vID that is present in the capability. vID is the identifying element used to do a hash check. If the vID belongs to a different Resource Server then the capability is sent to the Resource Server which the vID belongs to and is checked remotely using the `CoapValidateResource`. Otherwise, it is done locally. The `ValidationAlgorithm` class runs this check by comparing the hash stored in the capability and the hash computed using the servers shared secret and user ID. If the validation failed, the client is responded with a canceled. If the validation is done remotely, the remote Resource Server sends the exception list and is stored in the exception list data structure for this client and this Resource Server holds the baton for this client. If the validation is done locally, the current Resource Server already has the exception list for that client in that session. If the vID is of the Authorization Server then the Resource server will create the exception list for that client in this session.

2. Next, the `HCAPAuthorize` class checks if the permission exists from the current state of the client. If it does not exist, the client is responded with a canceled. If it exists, then the proof map is extracted from the payload. The proof map is a map mapping conditions to a proof tree. The proof map is checked using the `evaluateCertificates` method in the `CertificateHandler` class. In that method, each mapping is checked in an iterative manner where each Condition Certificate is validated in the `checkValidity` method. If the `endTime` field in the Condition Certificate is less than the current time (which was recorded at the time of receiving the request), that Condition Certificate is valid. The first Condition Certificate in the list must be signed by the Authorization Server. Every Condition Certificate succeeding the first Condition Certificate must be issued and signed by the entity referred to in its preceding Condition Certificate's `nextID` field. The signature for each Condition Certificate is verified using the `verify` method provided in `java.security.Signature` in the `verify` method of `CertificateHandler` class. The last Condition Certificate must be type-3. If all these are found to be correct, then that condition is considered valid. The `CertificateHandler` class does this for all the mappings and returns a list of accepted conditions.

3. Next, the `HCAPAuthorize` class makes an instance of the `HCAPHandleRequest` class with the capability, permission and the accepted conditions as parameters of the constructor. The `computeResponse` method is then called which checks if the permission and accepted condition pair is in the capability [2]. If it does not exist,

[2]For the accepted conditions, there could be multiple valid transitions. The code picks the maximal subset from

the server cancels the request of the client. Otherwise, this is a valid request. In this case, the permission and the maximal condition set are stored in the exception list for this client. The accepted conditions are those conditions which has a complete proof sent by the client. The chosen condition set is the maximal set out of all the valid condition sets.

4. After the permission is executed, the next state is extracted from the capability. If it is the same as the current state, then a success is responded back to the client. If it is a different state, the server tries to make a new capability with the new state as the current state and returns the new capability to `HCAPMessageDeliverer`. If the server does not have enough information to create a new capability, `createUpdReq` method is called which creates a state update request and returns it to the `HCAPMessageDeliverer` to be put in the response payload.

5. The server then replaces the ticket (either a state update request or a new capability) in the request payload using the `changeTicket` method in the `HCAPMessageDeliverer` class and passes to the resource. This means the permission will now be executed by the resource. After that, depending on the situation, the client may get a success response, a new capability or a state update request based on what was put in the response payload.

- **Garbage Collection and Baton Compression**

  One of the work of the `HCAPMessageDeliverer` is to do garbage collection. Garbage collection is done so that the Resource Server does not get bloated with

the set of all valid condition sets.

the size of the exception list. So, to relieve the Resource Server, garbage collection is done when either a certain time has passed or it has reached a number of entries to the exception list. This is done the same way as the original HCAP.

Another important activity the `HCAPMessageDeliverer` class performs is called Baton Compression. It is a recommended feature that can be turned on or off. If it is indeed turned on, the Resource Server checks if the number of entries in the exception list exceed the number of states in the fragment in the capability. If it is, baton compression is done using the `compress` method in `batonCompression` class and it effectively reduces the size of the exception list. This is done after the client performs a transitioning permission. This is also done the same way as the original HCAP.

This chapter goes over the third party APIs that were used in the implementation of the context-aware HCAP. Then it goes over the details of the how the code works by explaining the major classes and methods. The objective of this detailed account of the implementation in this Chapter is such that it is possible for others to replicate this work. The next chapter discusses the performance of the context-aware HCAP API that has been implemented using the process in this chapter.

Chapter 6

Empirical Evaluation

This chapter presents a performance evaluation of the context-aware HCAP protocol. In the evaluations, the Authorization Server, the Resource Server, the Sensor Integration Centre and Clients were all involved. The study was done in a simulated environment, where actual IoT devices were not used, instead these devices and their readings were simulated. The CASAs and the access requests the client would make were randomly generated (Sections 6.1.1 and 6.1.2) before starting each round of experimentation. Two experiments were conducted; the first evaluated the impact of varying sizes of CASA fragments on average authorization times (Section 6.2). The second experiment (Section 6.3) assesses the impact of the caching mechanism on the authorization times, which was explained in Section 4.5.4. Section 6.1 describes the configuration of the system on which the experiments were run on and also describes the setup of the experiments. The experimental results are discussed in Sections 6.2 and 6.3.

6.1 Preliminaries

Both experiments involved all three of the servers and the client. The Authorization Server, Resource Servers, Sensor Integration Centres and the Clients were run on separate machines. These machines had 3.0 GHz Intel Xeon(R) 5160 processors with 16 GB of RAM and ran on Fedora 28 operating system. The servers and the clients were all connected with a wired LAN with 900 MBps line.

In both the experiments, random CASAs are needed to simulate a real life scenario. In the real world, random fragments from a deterministic version of random CASAs would be given to the clients. The clients would then exercise random permissions by providing proofs of conditions. I have generated random CASAs and random execution traces (Definition 3.2.3) which dictate which

permissions would be requested by the client providing which proofs. In the next subsections, I will describe how random CASAs and random execution traces are generated.

6.1.1 Random CASA Generation

To construct a random CASA, I have outlined a few parameters that are needed to control the structure of a CASA. Let us discuss these parameters.

Parameters for CASA generation. Every CASA could have Q_{total} number of states, P_{total} number of permissions and C_{total} number of conditions. In addition to this, the number of transitions out of each state is between T_{min} and T_{max}, inclusive. In each transition, the number of condition is between C_{min} and C_{max}. Table 6.1 summarizes the values chosen for all the parameters.

Table 6.1: Parameters for generating random CASAs.

Parameter	Description	Value
Q_{total}	Total number of states	15
P_{total}	Total number of permissions	5
C_{total}	Total number of conditions	5
T_{min}	Min number of transitions out of each state	2
T_{max}	Max number of transitions out of each state	7
C_{min}	Min number of conditions required on each transition	0
C_{max}	Max number of conditions required on each transition	3

I have generated 100 random CASAs. All of them have been generated using the parameters in Table 6.1. In each CASA, there are Q_{total} states, although they might not all be reachable from the initial state. State q_0 is always the initial state and is always a part of the CASA. Each CASA has P_{total} permissions and C_{total} conditions.

The total number of permissions P_{total} was chosen to be a smaller number 5. This is because regardless of the number of conditions, a higher variability in permissions biases generation of CASAs which are largely deterministic from inception. I want to design CASAs without any

biases. Since choosing a bigger number of permissions P_{total} generates CASAs with an inherent bias, a smaller number was chosen. On the other hand, if C_{total} was chosen to be a large number, and P_{total} is a small number, it would bias the type of CASAs whose deterministic version would have an exponential number of states. These versions of CASAs would be generated with a greater probability and would skew the data. So, as to generate random CASAs without biases, total numbers of conditions (C_{total}) and permissions (P_{total}) are chosen to be a smaller number, that is, 5.

Generating one CASA. Let us now look into the process to generate one CASA. Generation of a CASA is done in two steps. Firstly, transitions are randomly generated. Secondly, each transition is labeled with a permission and a set of conditions.

Before the first step, I generate all $Q_{total} = 15$ states. I maintain a worklist of states. Initially, I add the start state q_0 to the worklist. Then I enter into the main loop. In each iteration, I remove one state from the worklist and work on it. Let's call that state the current state. I randomly generate transitions from the current state to the other states. If a destination state has not been added to the worklist in the past, then it is added. This concludes one iteration of the main loop. This process is repeated until the worklist is empty. Not all states are reachable from the start state. Unreachable states are ignored. Let us now examine the steps in greater details.

Step 1. In this step, I describe the process of randomly picking transitions from the current state to other states. I am picking a random subset of states out of Q_{total} number of states. There are 3 mechanisms that have been explored for this process. They are described in the next paragraphs.

The first mechanism to pick a random subset of states is to flip a coin for each of the 15 states. I could then draw a transition from the current state to the states which received a head in the coin toss. In this way, I would be generating any subset between the sizes of 0 and 15. But I do not want any arbitrary subset. I want a subset of a particular size, specifically a size between T_{min} (2) and T_{max} (7). Not only that, I want to generate a subset of states which has an equal probability of

being chosen out of all subsets of states of sizes between T_{min} and T_{max}. This brings us to the second mechanism.

The second mechanism to pick a random subset of states with equal probability can be achieved by first generating all subsets of states of sizes between T_{min} and T_{max} inclusive and then picking one subset. In this way, we would bias towards subsets of bigger sizes. Biasing towards bigger subsets is not a general phenomenon. But, if we set up the choices for Q_{total}, T_{min} and T_{max} to be 15, 2 and 7 respectively, then, for this specific configuration, the bias would be towards subsets of bigger sizes. This is why, this mechanism is not a viable option to pick a random subset of states and that brings us to the third mechanism.

In the third mechanism I pick a random number T which is between T_{min} and T_{max} inclusive. Next I sample T number of states from Q_{total} number of states. In this way, every subset size between T_{min} and T_{max} has an equal probability of being generated. This is the third method to pick random subset of states.

After the random subset of states are generated using the third mechanism, I add those states to the worklist if they have not been added to it in the past. Now, what we have generated is essentially a directed graph. The second step labels each directed edge with a permission and a set of conditions. In the next step, I will describe this labeling process.

Step 2. In this step, I describe the process of putting labels on each directed edge. The label consists of a permission and a set of conditions. For each edge that was generated in step 1, I pick a random permission out of P_{total} number of permissions where each permission has an equal chance of being picked. To pick a set of random conditions, I use the same process as mechanism 3 where I pick a random number C_{pick} which is in between C_{min} and C_{max} inclusive and then sample C_{pick} conditions out of C_{total} conditions. To be noted, since C_{min} can be 0, that means, the condition set could

also be empty.

At the end of these steps, we now have a CASA. In the same way a total of 100 CASAs are generated. By generating a 100 CASAs, I have tried to account for variability in terms of the structure of CASAs.

6.1.2 Trace Generation

Now that the CASAs have been generated, before we begin the experimentation, a design issue arises. If I simulate a client making a sequence of random requests, and the client is choosing these random requests in real time during the experiment, then that kind of deliberation would create noise (i.e., unnecessary delays) in the data being collected. I only want to measure the efficiency of this protocol. I do not want to collect the data of the time a simulated client takes to randomize the next decision for transitions. Therefore, I have decided that I will generate the sequence of transitions before I start the experiments.

I generate execution traces (Section 3.2.3) of length 100. Generally, not all execution traces are accepted by a CASA. But these experiments has nothing to do with unacceptable execution traces and I am only measuring the performance of this protocol. Therefore, I only generate execution traces of length 100, which are accepted by the respective CASA. I generate only one trace of length 100 for each CASA.

Generating random execution traces for the client is trivial. Starting with the initial state in the current CASA we can do a random walk up to a 100 transitions. That means, from the initial state, I pick a random transition and record the permission and the set of conditions as a part of the trace. The resulting state becomes the current state. From that state, I pick another transition. This is done until the length of the trace is 100. This process never gets "stuck" in a "sink" state because every state in the randomly generated CASA has at least $T_{min} = 2$ outgoing transitions to two different states in the CASA. You can always make a transition leading to a different state. This process is used to generate execution traces for all 100 CASAs.

6.1.3 Controlling the Length of Proof

The client needs to gather a proof for each condition if a specific transition requires it. In this implementation, the Authorization Server and the SICs all work in a hierarchical structure with the Authorization Server at the top of the hierarchy. This means $b = 1$ in accordance with the discussions in Section 4.5.4. The AS and the SICs all provide *at most one* Condition Certificate for each condition required.

To complete the proof, the client refers to the Condition Certificate provided by the Authorization Server for each condition and contacts the SICs necessary to complete a proof for each condition. For the experiments, we would like to control what type of Condition Certificate goes in each proof. For example, for a proof of length-3, it would need a type-1 and then a type-2 and then a type-3 Condition Certificate. For a proof of length-4, it would need two type-1 Condition Certificates and then a type-2 and a type-3. The first experiment was run where the proof lengths were $0, 2, 3, 4$ (represented by Figures 6.1 and 6.2). For the second experiment, the proof lengths ranged from $2 - 10$ (represented by Figure 6.3).

6.2 Experiment 1: Average Authorization Time

The context-aware HCAP protocol was designed in such a way that the Resource Servers do not have to do many computations and can provide authorization decisions without much lag. Therefore, authorization times are primarily dependent on the configuration and the speed of the network. The network packets between context-aware HCAP and original HCAP are different because their contents are different and because Condition Certificates were introduced. This is why we felt the necessity to measure how long it takes for a client to request access and get a response. We call this authorization time.

The authorization time can be broken down into two parts: 1. the time it takes for the client to gather the proof for a request and, 2. the time it takes for the client to make a request to the resource server and get a response. This experiment measures the average authorization time and

separately assesses the two parts above. The independent variables are: 1. the size of each fragment (Definition 3.4.1) and, 2. the length of proof for each request. The size of each fragment is varied by changing the number of states put in the fragment. The length of proof is varied such that each proof is the exact same in each iteration of the experiment.

6.2.1 Independent Variables of the Experiment

The clients are given fragments in capabilities which are partial specifications of a DCASA. This experiment checks the average time for *one* authorization decision when the fragment size is varied. The more states we put in a fragment, the bigger the fragment will be. Fragment sizes were varied by putting 1 to 7 states. I also varied the length of the proof the client has to gather. The lengths are: $0, 2, 3, 4$. This means, the number of Condition Certificates in each proof of a condition in each request can be $0, 2, 3, 4$. 0 meaning there are no proof provided for any condition.

Therefore, the independent variables in this experiment are the *fragment size* and the *size of each proof* of each condition the client has to gather. To be noted here that proof size cannot be 1 as the first Condition Certificate is provided by the Authorization Server and that certificate cannot be a type-3 Condition Certificate. So proof sizes were chosen to be $0, 2, 3$ and 4. I have also recreated the experiment done by Tandon *et al.* with the original HCAP [54], where there are no Condition Certificates. This was done to draw some comparisons on the average authorization time for original HCAP and context-aware HCAP with no proofs.

6.2.2 Experimental Setup

For this experiment, the servers are started. There are 2 Resource Servers, 3 Sensor Integration Centres and 1 Authorization Server. We measure two kinds of communication. Firstly, how long it takes to gather proof before each request and secondly, how long it takes to get a response from the resource server.

The client gathers a capability, with a set fragment size and a set of Condition Certificates from the Authorization Server. The client then starts a timer and communicates with different SICs to

construct a set of proofs before requesting to exercise a permission. When the proofs have been constructed for a set of conditions, the timer is stopped and the time difference is recorded. Before contacting a Resource Server, another timer is started and after the Resource Server has responded, the timer is stopped and the time difference is again recorded.

This process is repeated for 100 transitions and for 100 CASAs (a total of $10,000$ communications) with a set fragment size and proof length and an average time is recorded for both kinds of communication. After noting the average times, the fragment size is varied and the experiment is run again for all 100 transitions and all 100 CASAs. After the fragment size reaches 7, its reset to 1 and the length of proof is increased and the process starts again until the length of proof reaches 4.

Figure 6.1 reports average times to get one access decision when the length of proof is predetermined to be $2, 3$ and 4. Figure 6.2 is a comparison of the average time to get one access decision between original HCAP and context-aware HCAP. Let us look into the results of the experiments.

6.2.3 Results

The results of this experiment are condensed into three findings which are discussed in the following sections.

Finding 1: Authorization time decreases as fragment size increases. As the states in the fragments are increased from 1 to 7, there is a decrease in average access times; as seen by the pink, gray and green bars in Figure 6.2. This is counter-intuitive because as we increase the fragment size, the payload gets bigger and access decisions should take longer. But in reality this is not the case. In Figures 6.1 and 6.2 if we turn our attention to the average time to get access decision, we realize that as the size of the fragments increases, the authorization times from the Resource Server decrease in a linear fashion, even though the payload gets bigger.

This phenomenon is a result of the **update requests** (Section 4.4.2) that are issued by the Resource Server. When the Resource Server does not have enough information to create a new capability, it issues an update request. Now, before a client can request to exercise a new permission,

the client has to go to the Authorization Server to get a new capability. That extra communication with the Authorization Server is costly as we can see that in the charts in figures mentioned previously, the average time to get access decisions from the Resource Server is the highest when the fragment size is 1. When there is 1 state, the probability that the client will exercise a transitioning request is fairly high and therefore, the client will need to do an extra communication with the Authorization Server to get a new capability. That communication is the highest when the fragment size is the smallest.

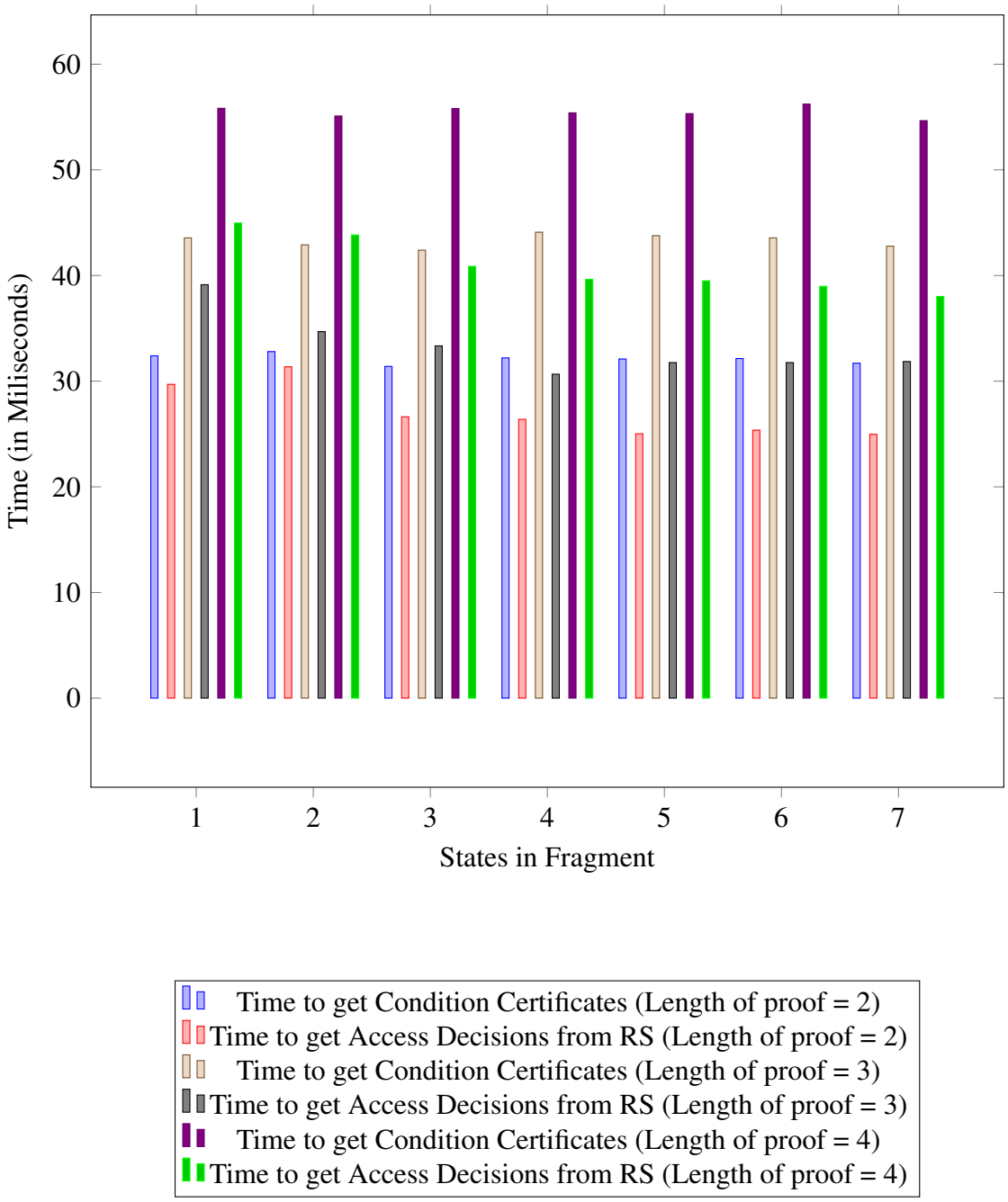

Figure 6.1: **Average times to get Condition Certificates and Access**

The size of the proof makes an overall impact as well. This can be seen in Figure 6.1; for the same fragment size, the average time to get access decision from the Resource Server increases as length of proof increases (green bars bigger than gray bars and gray bars bigger than pink bars). When length of proof is 2, time to get one access decision from the resource server is around

$25-30$ milliseconds (pink bars), and when length of proof is 3, the time increases to $31-38$ milliseconds (gray bars), and when length of proof is 4, the time increases up to about $39-44$ milliseconds (green bars). This is due to two reasons: 1. A bigger size of proof needs to be sent to the Resource Server leading to bigger network packets, and 2. The resource server has to validate a longer proof. That is why, as the size of the proofs increase, the average authorization time increases.

Finding 2: Addition of context is still within an acceptable range of authorization time. Adding context to HCAP increases the authorization times because the clients has to gather Condition Certificates and complete certain proofs before requesting to exercise a permission from the Resource Server. As the length of proof for each condition increases we realize that the time required to gather the proof is linearly increasing. It is around 34 milliseconds when length of proof is 2 (pink bars), 44 milliseconds when length of proof is 3 (gray bars) and 55 milliseconds when length of proof is 4 (green bars). On the other hand, as fragment sizes increase, the authorization time decreases. On the whole, the average authorization times in different configurations is between 10 and 100 milliseconds[1]. According to [45], the limit for having a user feel a system is reacting instantaneously is 0.1 second, or 100 milliseconds. Therefore, the authorization times reported in this experiment are well within an acceptable and tolerable range for a client.

[1]To get proof and access decision from Resource Server for fragment size 1 and proof size 4 is 45 and 55 milliseconds, which is a total of 100 milliseconds as per Figure 6.1. To get access decision from the Resource Server when proof size is 0 and fragment size is 7 is 10 milliseconds as per Figure 6.2. These are the best and worst configurations for context-aware HCAP.

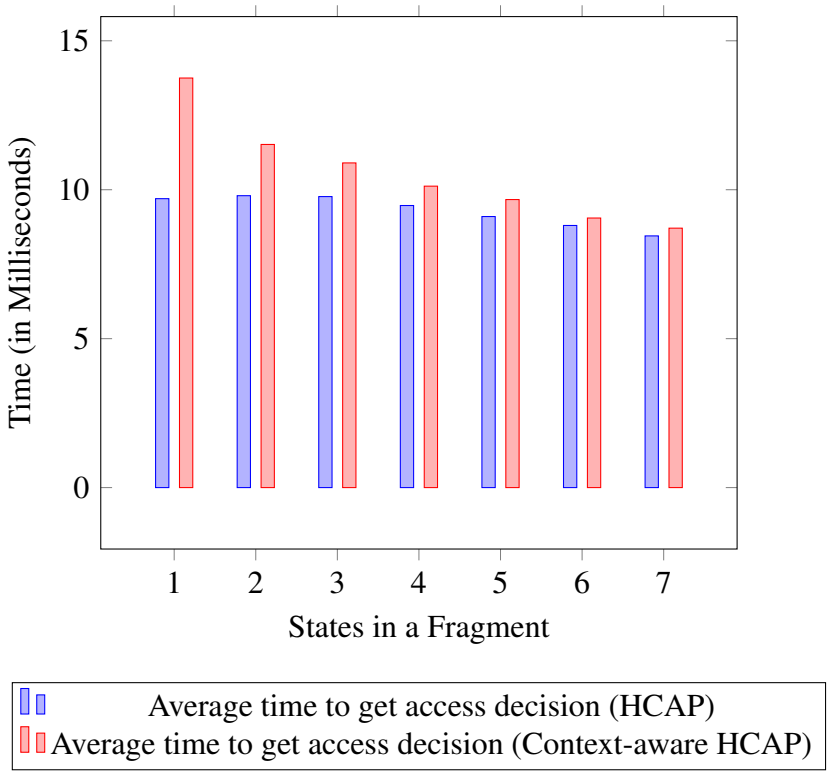

Figure 6.2: **Comparison of access times in HCAP and context-aware HCAP**

Finding 3: Competitive authorization times between HCAP and context-aware HCAP. As represented by Figure 6.2, we can see it is a comparison of the authorization times for HCAP and context-aware HCAP for fragment sizes ranging from 1 to 7. The proof size for the context-aware HCAP is 0 here and that means that it is comparable to the original design of HCAP where there is no concept of a proof. What we can see from Figure 6.2 is that as the fragment size increases, the authorization time decreases due to the decrease of the frequency of the update requests. Although the context-aware HCAP has a higher authorization time than original HCAP, the difference decreases over the increase of fragment sizes. Moreover, the difference between the average authorization times is negligible ranging from 5 milliseconds to 1 millisecond. The difference in the authorization times is due to the structure of the capabilities between the two schemes. Therefore, it is safe to say that context-aware HCAP has very competitive authorization times compared to original HCAP.

6.3 Experiment 2: Effect of the Proof Caching Mechanism

In the context-aware HCAP protocol, a client has to gather Condition Certificates from different SICs. There are two ways the client can go about processing the Condition Certificates. The client can either store the gathered Condition Certificates, or can forget the gathered Condition Certificates from the SICs by the caching mechanism described in Section 4.5.4. The proof caching mechanism allows clients to have a degree of control over the storage of the Condition Certificates. In the event a certain Condition Certificate in a proof tree is obsolete, the obsolete Condition Certificate and the subsequent Condition Certificates in the proof tree are deleted and the client needs to build up the proof tree from the last valid Condition Certificate as explained in Section 4.4.4.

In the naive method, the client has to construct proofs every time before requesting access. In this way, the client is penalized with longer authorization times for having restricted memory as observed in Section 6.2.3 [2]. On the other hand, enabling the caching mechanism would lessen the contacts with additional SICs and it is expected that gathering the Condition Certificates would take less time and would lead to lower authorization times. The purpose of this experiment is to see how the caching mechanism effects the time to gather the Condition Certificates when compared to the original method. The experimental results are represented in Figure 6.3.

6.3.1 Experimental Setup

The experiment is run with all the CASAs generated previously, in the same manner as explained before (Section 6.2.2). The length of proof is varied from 2 to 10 and so, 1 to 9 SICs are started to account for the different lengths of proofs. The only timer that is turned on is on the client machine before the client machine starts contacting the SICs, and it stopped once the proofs are completely constructed prior to issuing a request to a Resource Server. The graph in Figure 6.3 consists of the length of proof of each condition on the x-axis and the average time to gather a proof for one

[2] These experiments are run in the original methods where proof caching is turned off.

request, in milliseconds, in the y-axis. In each iteration of the experiment, the length of proof was kept constant, and fragment size was varied. With this setup, the average time to construct proofs for one request was recorded with the cache feature on, and then off. Two data points are recorded for each choice for length of proof. Then the length of proof is varied and the experiments are run again. The length of proof is changed from 2 to 10.

6.3.2 Results

The results of the second experiment is discussed in the following sections.

Finding 1: Caching mechanism significantly reduces the time to complete proof. It was expected that the caching mechanism will reduce the average time for a client to gather the proof before each request. The purpose of this experiment was then to draw some comparisons and to what extent the cache feature reduces the time to complete the proof. This might give some idea to the user and the administrators on how to better handle this feature.

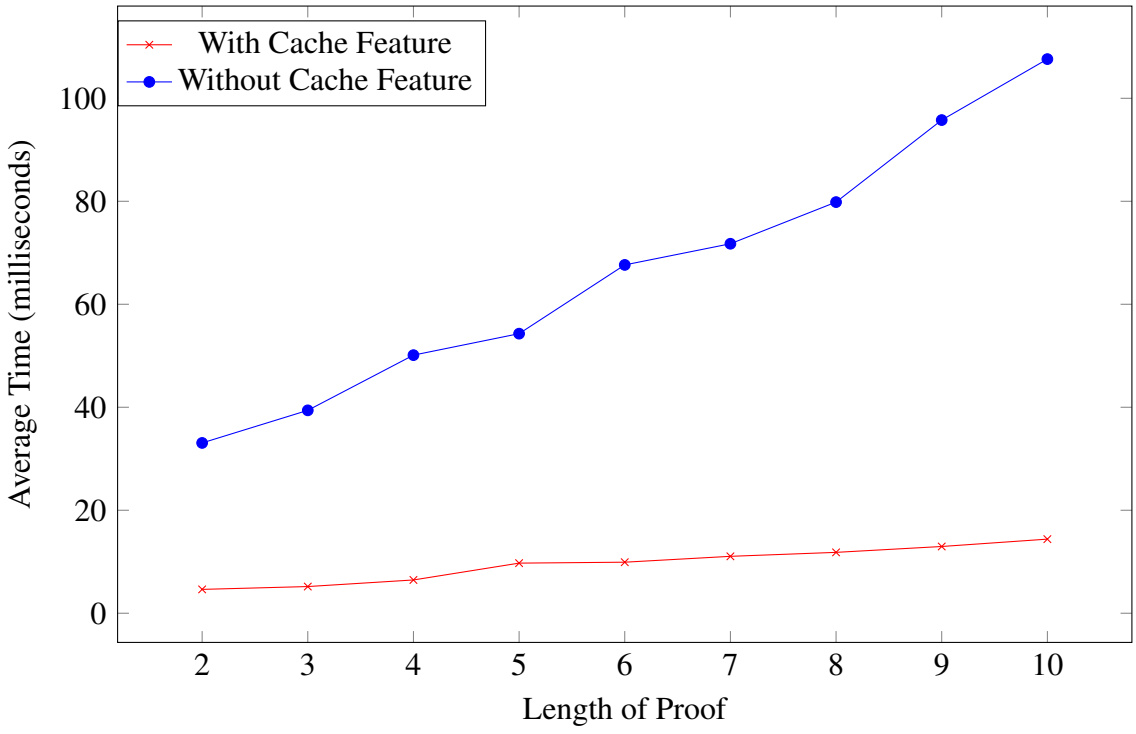

Figure 6.3: **Effect of proof caching feature (Fragment Size =** 1 **to** 7**)**

What is quite noticeable from Figure 6.3 is that, as the length of the proof increases, the difference between the lines increases. The difference is as little as 30 milliseconds when length of each proof is 2 and as large as 95 milliseconds when the length of each proof is 10. It is then apt to assume that as the length of proof gets longer, the difference between the average time to complete proof with and without the cache feature will get bigger. It is easy to imagine a hybrid system where the cache feature will work up to a certain size of the proof and then begin dropping certificates. This idea is left as a future implementation. It would be interesting to see the effects of a hybrid system.

Finding 2: With caching on, the time to construct a proof grows very slowly as the length of proof increases. We can also notice that as the length of each proof increases, the line representing the "on" state (the red line) does not increase as much in the y-axis. The slope of the red line is very small (slope = 1.381) compared to the slope of the blue line (slope = 9.07) [3]. The difference between completing a length-2 proof and a length-10 proof is as little as 10 milliseconds with the cache feature on. This shows that as we turn on the caching mechanism, even when the proof length is increased, the authorization time is increasing very slowly.

In this chapter, we looked at the average authorization times in the first experiment. We saw a counter-intuitive idea how even though fragment sizes increase, authorization times decrease due to the phenomenon of update requests. The smaller the fragments, the higher the frequency of update requests, the higher the authorization times. We also saw the effects of the size of proofs; as proof size increase, the authorization times increase without a caching feature (Figure 6.1). A comparison was also drawn between context-aware HCAP and original HCAP with the context-aware HCAP featuring a proof size of 0. It was concluded that context-aware HCAP with 0 proof size is very close to the authorization times of original HCAP (Figure 6.2). The second experiment looked at the effects of the caching mechanism and the magnitude of difference of completing a proof with and without it (Figure 6.3). We also saw that as length of each proof increases, the caching mechanism is largely unaffected. In the next chapter, I conclude this work with some ideas

[3] Slope calculated from best-fit value.

on areas to improve.

Chapter 7

Conclusion

7.1 Conclusion

The era of IoT devices is upon us. We have incorporated these devices in our daily lives from having personal uses to controlling the environment of an office space and even an industry line. The security and privacy implications of these devices cannot be ignored as they collect our personal data and control the environment around us. Tandon *et al.* [54] introduced *History-Based Capability System (HCAP)*, developing on the idea of IoT devices being physically embedded over a wide area (Chapter 2). This characteristic of IoT devices means that permissions have to be exercised in sequences. Using this, HCAP supports the imposition of sequencing constraints on the permissions that need to be exercised one after another in a certain order.

This research introduced *Context-Aware History-Based Capability System (Context-Aware H--CAP)* as an extension of HCAP put forward by Tandon *et al.* [54]. Context-Aware HCAP extends HCAP by introducing contextual constraints as an added layer on top of sequencing constraints of HCAP (Chapter 3). "Context" in this research referred to the state of an entity (a person, place or object). For example, location of a person, humidity in a room and so on. A "Context-Aware" system takes the situation of an entity into account, and uses that information for smooth operation of the system. Context-Aware HCAP is a protocol that uses context information in the authorization decisions of resources. At the time of requesting access, a user has to present a proof of context conditions bundled with a capability, in order to get access.

HCAP enforces sequencing constraints using *Security Automata (SA)*, which are akin to deterministic finite automata (DFA), where each transition is labeled with a permission and every state is a final state. To introduce context-awareness, this research introduced *Context-Aware Security Automata (CASA)* (Section 3.2), which is a modified version of the SA. Transitions in CASAs are

not only labeled by a permission but also a set of context conditions or just "conditions". Conditions dictate the precise state of an entity as a prerequisite to exercising a permission, for example, the temperature of room 628 is 24.8 degrees. A set of conditions together form a context. These set of conditions need to be true at the time of requesting access. In Context-Aware HCAP, the requestor is responsible for collecting and providing the proof of conditions.

Since the requestor is not trusted and since it is tasked with gathering the proof of conditions, there is a very real possibility the requestor can act with an ill-will. Let's say a requestor provides a number of proofs of conditions and is denied access. But then, trying to act maliciously, throws away certain proofs and gains access. In short, the requestor gains access by not telling the whole truth. To prevent this from happening, a security property is introduced: *"Tamper-Proof Against Information Withholding (TPIW)"* (Section 3.3). TPIW makes sure that by willfully withholding proof of certain conditions, adversaries will not gain access. It was proved that CASAs have this security property (Theorem 3.3.1).

CASAs are embedded in an authorization token called a capability which is provided to the clients. CASAs may contain a large number of states and transitions; it is infeasible to put the full CASA in a capability. Therefore, a partial specification of a CASA: a CASA fragment or just "fragment" is introduced (Section 3.4.1). A fragment only has a subset of states and transitions of the full CASA which is much more economical to embed in a capability. A resource server needs to make transition decisions based on whatever information is in the fragment. This is called a local decision. There can be multiple valid transitions from one state in a CASA. With the limited information in the fragment, it is impossible to choose one out of many. Choosing one may lead to a future access request being denied (which should be accepted). It cannot be guaranteed that choosing one will not have future repercussions. This is due to the non-deterministic nature of CASAs.

Because of this, a special class of CASAs are introduced called Deterministic CASAs or DCASAs which are just regular CASAs with two special properties (Definition 3.4.2). It was

proved that DCASAs do not have the problem of making local decisions (Theorem 3.4.1). Choosing a "most specific transition" would always allow all future transitions which would have been valid if another transition was chosen.

It is not always convenient to design only DCASAs, so a conversion mechanism is formulated which converts a CASA into a DCASA using standard subset construction (Section 3.5.1). It is proved that the output of the conversion mechanism is a DCASA (Theorem 3.5.1). It is also proved that the input CASA and the output DCASA accept the same language (Theorem 3.5.2). A comparison of the nature of non-determinism in NFA and CASAs is drawn (Section 3.4.1). In an NFA, the non-determinism comes from having multiple arcs emanating from a state having the same label. On the other hand, multiple arcs emanating from a specific state in a CASA can having the same permission label but different sets of condition requirements. Let an input be given to a CASA with that permission but a condition set which is a superset of some of them. That would mean that all of those transitions are valid. That is where non-determinism comes from in a CASA. Next, the space complexity of a DFA and a DCASA converted using a subset construction method are discussed (Section 3.6).

The Clients need Condition Certificates to formulate a proof of a condition (Section 4.1). There are three types of Condition Certificates but only type-3 Condition Certificate proves a condition is true (Section 4.2.3). The Authorization Server provides some type-1 and type-2 Condition Certificates. The following example is a brief on each type of Condition Certificates (Section 4.4.1). Let there be the two SICs: SIC-A and SIC-B. If x is an assertion then a type-3 Condition Certificate states "*SIC-A* says x". Meaning only a type-3 Condition Certificate is certifying a condition. A type-2 SIC Condition Certificate is, "*SIC-A* says, in matters of x, trust *SIC-B*" and a type-1 Condition Certificate states, "*SIC-A* says, in matters of x, trust a delegate of *SIC-B*, if *SIC-B* says so".

Before the client can request access, the Client has to gather proofs of conditions by contacting relevant SICs. After the Client completes the proofs of conditions, it contacts the RS providing

the capability and the Condition Certificates. The RS checks the integrity of the capability and validates the proofs. If it holds the baton for that client, for that session, then the RS provides the Client with a new capability or an update request according to the fragment in the capability. Otherwise, the RS executes remote capability validation. After an access decision is made, the Clients may choose to remember the collected Condition Certificates, or forget them. This is a feature introduced to Context-Aware HCAP called "caching" (Section 4.5.4). With caching on, the Client device will need more memory but will achieve faster authorization decisions. With caching off, the Client device will not need a large memory storage but will have lags in authorization decisions because they have to construct proof from scratch every time access is required.

Techniques such as baton compression, hard GC and soft GC are all inherited from the work of Tandon *et al.* [54] and modified to accept new data structures and work with conditions. Context-Aware HCAP was implemented with an AS and a number of RS, SIC and Clients and its performance was measured (Chapters 5 and 6).

7.2 Future work

Although this work expanded on the idea of original HCAP and introduced Context-Awareness in HCAP, there are still opportunities for future works. Some of them are discussed below.

1. Context-Aware HCAP has no assurance of proper functioning if SICs go offline. With SICs offline, the Clients will not be able to complete proofs and may not be able to get proper access. Context-Aware HCAP can be extended such that any failure on the side of the SICs do not hamper regular mechanics of the protocol.

2. A caching mechanism was introduced in Section 4.5.4. The caching mechanism can be turned on or off depending on the situation. But a hybrid caching mechanism might be developed as well. It can be designed in such a way that the Client will start throwing away Condition Certificates when a threshold amount of Cer-

tificates have been stored. Moreover, it can perform regular checks on the stored Condition Certificates and do a "garbage collection" for them as well so that invalid Certificates are thrown away.

3. Some standard issues in access control are: delegation and revocation. Context-Aware HCAP does not include an opportunity for delegation or revocation of capabilities. Delegation and revocation of capabilities are commonplace in other access control paradigms [12, 20, 59, 28] but Context-Aware HCAP does not fully support delegation and revocation in its true sense. For example, entity *A* can temporarily transfer its own capability to entity *B*. *B* can only *temporarily* perform tasks *A* is capable of. A Teaching Assistant responding to students emails on behalf of an Instructor is an example of *delegation*. A capability in context-aware HCAP is client-specific and cannot be used by any other client. On the other hand, if for some reason access control policies are changed or a client is revoked all privileges, then revoking the capabilities already with the client is not within the scope of this work. For example, if an employee is let go then the capabilities and all the access to the company must be revoked. Context-aware HCAP does not have options to perform revocations like this. Extensions can be made to Context-Aware HCAP to support delegation and revocation of capabilities.

   There are also work that can be done to revoke Condition Certificates similar to the example for capabilities. There is *some* form of revocation of Condition Certificates. After a certain time period, Condition Certificates are expired and rendered invalid. But deliberate revocation of these certificates are not within the scope of this work. Extensions can be made to incorporate revocation of Condition Certificates as well.

4. There are no easy-to-understand specification language/code to specify access control policies in the form of CASAs. In my implementation, there are provisions to

input a CASA from an XML file which needs to be typed up in a certain configuration. But it is inefficient and complicated for practical usage by policy makers from a non-technical background. Therefore, it is recommended that a specification language be designed which is easy-to-understand and that can specify CASAs for resource owners or administrators.

5. The access control policies are in general static and are not subject to change because these policies rarely change. This becomes a problem in certain situations in healthcare, some form of emergency or disaster management [16, 42]. Some flexibility is necessary in Context-Aware HCAP, for example, designing a "break-glass" policy for access in dire situations. There should be provisions for documenting these special accesses and would need to be audited in case such access is granted. Context-Aware HCAP can be expanded to include a break-glass policy for emergency situations.

This chapter concludes the work in this thesis and presents some future work ideas and recommendations to improve upon Context-Aware History-Based Capability System.

Bibliography

[1] The constrained application protocol (coap). https://tools.ietf.org/html/rfc7252. Accessed: 2019-04-19.

[2] Department of economic and social affairs population dynamics world population prospects 2019. https://population.un.org/wpp/Download/Standard/Population/. Accessed: 2021-09-23.

[3] Java cryptography architecture (jca) reference guide. https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html. Accessed: 2019-09-23.

[4] Java cryptography architecture standard algorithm name documentation. https://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#Signature. Accessed: 2019-09-25.

[5] Spki requirements. https://www.rfc-editor.org/rfc/rfc2692.html. Accessed: 2021-06-04.

[6] Uniform resource identifier (uri): Generic syntax. https://tools.ietf.org/html/rfc3986. Accessed: 2019-06-19.

[7] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS) 15*, 4 (1993), 706–734.

[8] ABOWD, G. D., DEY, A. K., BROWN, P. J., DAVIES, N., SMITH, M., AND STEGGLES, P. Towards a better understanding of context and context-awareness. In *International symposium on handheld and ubiquitous computing* (1999), Springer, pp. 304–307.

[9] ALAGAR, V., ALSAIG, A., ORMANDJIVA, O., AND WAN, K. Context-based security and privacy for healthcare iot. In *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)* (2018), IEEE, pp. 122–128.

[10] ALMOTIRI, S. H., KHAN, M. A., AND ALGHAMDI, M. A. Mobile health (m-health) system in the context of iot. In *2016 IEEE 4th international conference on future internet of things and cloud workshops (FiCloudW)* (2016), IEEE, pp. 39–42.

[11] ALMUSAYLIM, Z. A., AND ZAMAN, N. A review on smart home present state and challenges: linked to context-awareness internet of things (iot). *Wireless networks 25*, 6 (2019), 3193–3204.

[12] ANGGOROJATI, B., MAHALLE, P. N., PRASAD, N. R., AND PRASAD, R. Capability-based access control delegation model on the federated IoT network. In *Wireless Personal Multimedia Communications (WPMC), 2012 15th International Symposium on* (2012), IEEE, pp. 604–608.

[13] BARI, L., AND O'NEILL, D. P. Rethinking patient data privacy in the era of digital health. *Health Affairs 12* (2019).

[14] BENDAVID, Y., BAGHERI, N., SAFKHANI, M., AND ROSTAMPOUR, S. Iot device security: Challenging "a lightweight rfid mutual authentication protocol based on physical unclonable function". *Sensors 18*, 12 (2018), 4444.

[15] BHATTI, R., BERTINO, E., AND GHAFOOR, A. A trust-based context-aware access control model for web-services. *Distributed and Parallel Databases 18*, 1 (2005), 83–105.

[16] BRUCKER, A. D., AND PETRITSCH, H. Extending access control models with break-glass. In *Proceedings of the 14th ACM symposium on Access control models and technologies* (2009), pp. 197–206.

[17] CHAPIN, P. C., SKALKA, C., AND WANG, X. S. Authorization in trust management: Features and foundations. *ACM Computing Surveys (CSUR) 40*, 3 (2008), 1–48.

[18] CONNER, W. S., KRISHNAMURTHY, L., AND WANT, R. Making everyday life easier using dense sensor networks. In *International Conference on Ubiquitous Computing* (2001), Springer, pp. 49–55.

[19] CONSOLVO, S., ROESSLER, P., SHELTON, B. E., LAMARCA, A., SCHILIT, B., AND BLY, S. Technology for care networks of elders. *IEEE pervasive computing 3*, 2 (2004), 22–29.

[20] CRAMPTON, J., AND KHAMBHAMMETTU, H. Delegation in role-based access control. *International Journal of Information Security 7*, 2 (2008), 123–136.

[21] DA SILVA, D. M. A., AND SOFIA, R. C. A discussion on context-awareness to better support the iot cloud/edge continuum. *IEEE Access 8* (2020), 193686–193694.

[22] DAGLISH, D., AND ARCHER, N. Electronic personal health record systems: a brief review of privacy, security, and architectural issues. In *2009 world congress on privacy, security, Trust and the Management of e-Business* (2009), IEEE, pp. 110–120.

[23] ELAYAN, H., ALOQAILY, M., AND GUIZANI, M. Digital twin for intelligent context-aware iot healthcare systems. *IEEE Internet of Things Journal 8*, 23 (2021), 16749–16757.

[24] FONG, P. W. Access control by tracking shallow execution history. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on* (2004), IEEE, pp. 43–55.

[25] GOCHHAYAT, S. P., KALIYAR, P., CONTI, M., TIWARI, P., PRASATH, V., GUPTA, D., AND KHANNA, A. Lisa: Lightweight context-aware iot service architecture. *Journal of cleaner production 212* (2019), 1345–1356.

[26] GONG, L. A secure identity-based capability system. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on* (1989), IEEE, pp. 56–63.

[27] GUO, B., SUN, L., AND ZHANG, D. The architecture design of a cross-domain context management system. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)* (2010), IEEE, pp. 499–504.

[28] HAYTON, R. J., BACON, J. M., AND MOODY, K. Access control in an open distributed environment. In *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No. 98CB36186)* (1998), IEEE, pp. 3–14.

[29] HOPCROFT, J., AND J.D. ULLMAN. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Cambridge, 1979.

[30] HUFFMAN, D. A. The synthesis of sequential switching circuits. *Journal of the franklin Institute 257*, 3 (1954), 161–190.

[31] HUTH, M., AND RYAN, M. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[32] HYUN-WOOK, K., HOQUE, R. M., HYUNGYU, S., AND YANG, S.-H. Development of middleware architecture to realize context-aware service in smart home environment. *Computer Science and Information Systems 13*, 2 (2016), 427–452.

[33] INTERNATIONAL, E. *The JSON Data Interchange Format*. October 2003.

[34] JIH, W.-R., CHENG, S.-Y., HSU, J. Y.-J., TSAI, T.-M., ET AL. Context-aware access control in pervasive healthcare.

[35] KAYES, A., RAHAYU, W., DILLON, T., CHANG, E., AND HAN, J. Context-aware access control with imprecise context characterization through a combined fuzzy logic and ontology-based approach. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (2017), Springer, pp. 132–153.

[36] KORTUEM, G., KAWSAR, F., SUNDRAMOORTHY, V., AND FITTON, D. Smart objects as building blocks for the internet of things. *IEEE Internet Computing 14*, 1 (2009), 44–51.

[37] KUMAR, A., KARNIK, N., AND CHAFLE, G. Context sensitivity in role-based access control. *ACM SIGOPS Operating Systems Review 36*, 3 (2002), 53–66.

[38] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Design of a role-based trust-management framework. In *Proceedings 2002 IEEE Symposium on Security and Privacy* (2002), IEEE, pp. 114–130.

[39] LI, N., WINSBOROUGH, W. H., AND MITCHELL, J. C. Distributed credential chain discovery in trust management. *Journal of Computer Security 11*, 1 (2003), 35–86.

[40] MAHALLE, P. N., ANGGOROJATI, B., PRASAD, N. R., AND PRASAD, R. Identity driven capability based access control (ICAC) scheme for the internet of things. In *Advanced Networks and Telecommuncations Systems (ANTS), 2012 IEEE International Conference on* (2012), IEEE, pp. 49–54.

[41] MAHALLE, P. N., ANGGOROJATI, B., PRASAD, N. R., AND PRASAD, R. Identity authentication and capability based access control (iacac) for the internet of things. *Journal of Cyber Security and Mobility 1*, 4 (2013), 309–348.

[42] MARINOVIC, S., CRAVEN, R., MA, J., AND DULAY, N. Rumpole: a flexible break-glass access control model. In *Proceedings of the 16th ACM symposium on Access control models and technologies* (2011), pp. 73–82.

[43] MASOOD, I., WANG, Y., DAUD, A., ALJOHANI, N. R., AND DAWOOD, H. Towards smart healthcare: patient data privacy and security in sensor-cloud infrastructure. *Wireless Communications and Mobile Computing 2018* (2018).

[44] MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[45] NIELSEN, J. *Usability engineering*. Morgan Kaufmann, 1994.

[46] PARK, J., AND SANDHU, R. The uconabc usage control model. *ACM Transactions on Information and System Security (TISSEC) 7*, 1 (2004), 128–174.

[47] PERLMAN, R., KAUFMAN, C., AND SPECINER, M. *Network security: private communication in a public world*. Pearson Education India, 2016.

[48] RICHARDSON, L., AMUNDSEN, M., AMUNDSEN, M., AND RUBY, S. *RESTful Web APIs: Services for a Changing World*. " O'Reilly Media, Inc.", 2013.

[49] RIVEST, R. L., AND LAMPSON, B. Sdsi-a simple distributed security infrastructure. Crypto.

[50] SAADEH, M., SLEIT, A., SABRI, K. E., AND ALMOBAIDEEN, W. Hierarchical architecture and protocol for mobile object authentication in the context of iot smart cities. *Journal of Network and Computer Applications 121* (2018), 1–19.

[51] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[52] SCHNEIDER, F. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC) 3*, 1 (2000), 30–50.

[53] SHUAI LI, A., SAFAVI-NAINI, R., AND FONG, P. W. A capability-based distributed authorization system to enforce context-aware permission sequences. *arXiv e-prints* (2022), arXiv–2211.

[54] TANDON, L., FONG, P. W., AND SAFAVI-NAINI, R. HCAP: A History-Based Capability System for IoT Devices. In *Proceedings of 23rd ACM Symposium on Access Control Models and Technologies* (2018), ACM, p. 13 pages.

[55] TONINELLI, A., MONTANARI, R., KAGAL, L., AND LASSILA, O. A semantic context-aware access control framework for secure collaborations in pervasive computing environments. In *International semantic web conference* (2006), Springer, pp. 473–486.

[56] VAHDAT-NEJAD, H., ZAMANIFAR, K., AND NEMATBAKHSH, N. Context-aware middleware architecture for smart home environment. *International journal of smart home 7*, 1 (2013), 77–86.

[57] VLISSIDES, J., GAMMA, E., HELM, R., AND JOHNSON, R. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Professional, 1994.

[58] WAN, K., AND ALAGAR, V. Integrating context-awareness and trustworthiness in iot descriptions. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing* (2013), IEEE, pp. 1168–1174.

[59] WANG, Q., LI, N., AND CHEN, H. On the security of delegation in access control systems. In *European Symposium on Research in Computer Security* (2008), Springer, pp. 317–332.

[60] YANG, C., SHEN, W., AND WANG, X. The internet of things in manufacturing: Key issues and potential applications. *IEEE Systems, Man, and Cybernetics Magazine 4*, 1 (2018), 6–15.

[61] ZHANG, G., AND PARASHAR, M. Context-aware dynamic access control for pervasive applications. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference* (2004), pp. 21–30.