TECHNIQUES FOR BUILDING ROBUST AND USER CUSTOMIZABLE IOT SYSTEMS

by

Trusit Sandipbhai Shah

APPROVED BY SUPERVISORY COMMITTEE:

Subbarayan Venkatesan, Chair

Dinesh Bhatia

Kamil Sarac

Murat Kantarcioglu

This dissertation is dedicated to

my parents and my beloved wife, Pooja,

for their constant support,

inspiration and guidance.

TECHNIQUES FOR BUILDING ROBUST AND USER CUSTOMIZABLE IOT SYSTEMS

by

TRUSIT SANDIPBHAI SHAH, BE, ME, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2019

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Venky for his constant guidance throughout my PhD. He has always encouraged me to solve a research problem with great efficiency. His mentorship was not limited to my research and also provided me guidance for the other aspects of my life.

I would like to thank Dr. Kamil Sarac and Dr. Murat Kantarcioglu for providing critical suggestions on the security aspects of this dissertation. I also thank Dr. Dinesh Bhatia for his suggestion for the hardware design.

During my PhD, I have worked with many undergraduate and graduate students and I would like to thank them for their help in designing and implementing several modules of this user-customizable IoT system. They include Somasundaram Ardhanareeswaran, Vishwas Lokesh, Tifanny Ngo, Rucha Rajurkar, Harsha Reddy, Siddharth Satyapriya and Harprit Singh. I would also like to thank my fellow students Shreyas Gokhale, Hemant Malik, Pinak Patel, Hrishikesh Prajapati, Devangsingh Sankhala and Vidarsh Shah for their constant support.

November 2019

TECHNIQUES FOR BUILDING ROBUST AND USER CUSTOMIZABLE IOT SYSTEMS

Trusit Sandipbhai Shah, PhD
The University of Texas at Dallas, 2019

Supervising Professor: Subbarayan Venkatesan, Chair

In the last decade, billions of new IoT devices have been developed by different providers using either their proprietary architecture or standard architecture like IoT M2M, AWS IoT, and AT&T M2X. Many of these IoT systems are rigid with no user customization, and a user can configure only a few parameters in the IoT system. We present a user-customizable IoT architecture that enables users to build and customize IoT systems based on their requirements without writing any code.

User customization is performed in two ways: hardware customization and software customization. In hardware customization, a user can add/remove IoT sensors/actuators, to/from the IoT system without knowing any hardware details of those sensors/actuators. For software customization, the user can create tasks without writing any code. We have designed a rule engine, which converts user-desired actions into computer code as a part of software customization. We design a concept called self-aware sensor/actuator to achieve user customization. A self-aware sensor/actuator is a sensor/actuator aware of its type, attributes, capabilities, and constraints and provides a mechanism to the users to customize tasks using these parameters.

In the later part of this dissertation, we describe the robustness and security of the user-customizable IoT system. An IoT environment containing a large number of sensing and

actuating devices increases the chances of creating conflicting and incomplete rules. We introduce a new conflict called independent conflict and incompleteness in a boolean function based rule structure. We have developed conflict resolution methods for different types of rule conflicts. We have tested our user-customizable IoT architecture and its robustness in several real-life scenarios.

Authentication of the IoT device is one of the important aspects of securing an IoT system. We design mechanisms to secure the authentication process between 1) an IoT server and an IoT device and 2) a user and an IoT device. Our approach, secure vault, prevents side-channel attacks on mutual authentication between an IoT server and an IoT. We introduce a variant of the client puzzle named login puzzle, which prevents a mass capture of IoT devices from the DDoS malwares. We calculated that the famous Mirai attack, which acquired 64500 devices in 20 hours, would have taken two months if all the IoT devices were using the login puzzle.

TABLE OF CONTENTS

ACKNOWLEDGMENTS v
ABSTRACT vi
LIST OF FIGURES xi
LIST OF TABLES xii
CHAPTER 1 INTRODUCTION 1
1.1 Our Contributions 4
1.2 Dissertation Roadmap 5
CHAPTER 2 SYSTEM ARCHITECTURE 6
2.1 User Customization 6
2.2 Mutual Authentication 8
2.2.1 Assumptions 8
2.2.2 Threat Model 9
2.2.3 Our Authentication Architecture 10
CHAPTER 3 PREVIOUS WORKS 12
3.1 Related to User Customizable IoT Systems 12
3.2 Related to Rule Conflict 14
3.3 Related to Mutual Authentication 14
3.4 Related to Login Puzzle 16
CHAPTER 4 USER CUSTOMIZATION 18
4.1 Self-aware sensor/actuator 18
4.2 Convert an arbitrary sensor/actuator into self-aware sensor/actuator 19
4.2.1 Select a suitable processing element 19
4.2.2 Interface sensor/actuator to processing element and program parameters of sensor/actuator into memory 19
4.2.3 Communication interface between self-aware device and IoT gateway device 20
4.2.4 Self-aware device discoverable code for IoT gateway device 20
4.3 Hardware customization 21

4.3.1 Validation of sensor/actuator value . . . 22
4.3.2 Working Status of Sensor/Actuator . . . 22
4.3.3 Software Customization . . . 23
4.3.4 Self-aware sensor/actuator properties . . . 25
CHAPTER 5 RULE CONFLICT . . . 27
5.1 Our Approach . . . 27
5.1.1 Rule Structure . . . 27
5.1.2 Rule Conflict and Incompleteness Detection . . . 29
5.1.3 Rule Decomposition to DNF . . . 31
5.1.4 Rule Conflict Classification . . . 32
5.1.5 Rule Incompleteness . . . 36
5.2 Rule Conflict Resolution . . . 39
5.2.1 Priority Method . . . 40
5.2.2 Adjusting Rule Sensor Values . . . 40
CHAPTER 6 MUTUAL AUTHENTICATION . . . 42
6.1 3-way mutual authentication . . . 42
6.2 Our Approach . . . 43
6.2.1 Secure Vault . . . 44
6.2.2 Challenge Response . . . 44
6.2.3 Changing the secure vault . . . 47
6.2.4 Securing vault on hardware . . . 48
6.3 Security analysis of our approach . . . 49
6.3.1 Man in the middle attack . . . 49
6.3.2 Next password prediction . . . 50
6.3.3 DoS attack . . . 51
6.4 Performance Analysis . . . 51
6.4.1 Power Analysis . . . 51
6.4.2 Memory Analysis . . . 53
6.4.3 Security Analysis . . . 54

CHAPTER 7 LOGIN PUZZLE 56
7.1 DDoS Attack Overview 58
7.2 Our Methodology 59
7.2.1 Login Puzzle 60
7.2.2 Complexity Increment of the Login Puzzle 61
7.2.3 Login Mechanism 61
7.3 Resource and Timing Analysis 62
7.4 Evaluation and Performance Measure 65
7.4.1 Comparison with Blocking Method 67
7.5 Comparison with Existing Client Puzzles 68
7.6 Discussion and Future Work 69
CHAPTER 8 IMPLEMENTATION 71
8.1 User Customizable IoT System 71
8.1.1 IoT Server 71
8.1.2 IoT Device Gateway 72
8.1.3 Self-aware Device 72
8.1.4 User Interface 73
8.2 Rule Conflict 74
8.2.1 System Architecture 74
8.2.2 Complexity Analysis 74
8.2.3 Smart Agriculture System 76
8.3 Secure Vault 77
CHAPTER 9 CONCLUSION 79
REFERENCES 80
BIOGRAPHICAL SKETCH 86
CURRICULUM VITAE

LIST OF FIGURES

2.1 Gateway based architecture. 6
2.2 Standalone architecture. 7
4.1 Structure of a rule engine 24
5.1 Visual representation of a rule with an anti-action. 28
5.2 This figure is an expression tree for $(t_1 \vee t_2) \wedge (t_3 \vee t_4)$. 30
6.1 Message exchange for 3-way mutual authentication 43
6.2 Message exchange for secure vault mutual authentication 45
8.1 This figure represents JSON structure of a simple rule. The rule is "if soil moisture is less than 30 % turn on the sprinkler and keep it on till the soil moisture reaches 40 %". 75

LIST OF TABLES

4.1 Example properties of self-aware sensors/actuators 25
4.2 Properties of a soil moisture sensor 26
5.1 Rule description 29
5.2 Clauses Relationship Classification 30
5.3 Trigger Condition Range Relationship 31
5.4 Conflict Definitions 31
6.1 Energy consumption of different protocols 52
6.2 Energy consumption of different authentication schemes 52
6.3 Data memory required for different authentication schemes 53
6.4 Data memory required for different authentication schemes 53
6.5 Password prediction complexity for different values of m and n 54
7.1 Time required to solve login puzzle for different type of devices 65
7.2 Time required to capture 64,500 devices with login puzzle using different environment 67
7.3 Login puzzle comparison 69
8.1 Smart Agriculture Rules 76
8.2 Smart Agriculture Rule Conflict Analysis 76

CHAPTER 1

INTRODUCTION

With the advancements in the field of embedded systems, the number of IoT devices increased exponentially and provided a convenient way to monitor and control sensing remotely and actuating systems. Current IoT solutions, offered by many different providers, typically have a proprietary system including user interfaces (both smartphone and web). The lack of standardization for IoT products makes them rigid and limits the users ability to customize the product. Most of these architectures are developed for specific use cases and supports a limited number of sensors and actuators. If a user wants to add a new type of sensor to the system, it requires a change in the code of the architecture. This limits a user without technical knowledge of coding or hardware from adding a new type of sensor to the system. A standardization way to define all types of sensors and actuators can solve the above problem.

Platforms like AT&T M2X [4], AWS IoT [43] and Samsung SmartThings [54] provide architectures such that any developer can import any sensor or actuator to the IoT system without changing any code in the IoT architecture. AT&T M2X and AWS IoT store sensor values as a stream of numbers or strings. Each value in the stream contains a timestamp and data (data must be either a string or a number). Any sensor type that provides numeric or string value can be easily integrated into these architectures. Sensor types like camera that provide encoded streams as data output, require additional coding to integrate into these architectures. On the other hand, Samsung SmartThings can integrate any type of sensor data, and it is responsibility of a developer to add sensor data structure while adding the sensor type. All these architectures can integrate into any type of sensors, but they provide limited user-customizability. In AT&T M2X and AWS IoT, a user can create customizable tasks only on sensor data. Parameters other than sensor data can not be used to create customized tasks. Samsung SmartThings allows user customization on a parameter other

than the sensor data, but for each automated task on a parameter, a separate code must be written by the developer.

Our self-aware sensor/actuator approach provides a unique mechanism for user customization. All the sensors/actuators are self-aware and can provide all the parameter details (including how to customize a task) to the IoT architecture. Thus, no additional coding is required to customize an automated task on a parameter. The advantage of this approach over the SmartThings is, an additional code is required only for each parameter, not for each task. Hence, to customize a task on an IoT system, a user does not need to write additional code.

The safety of an IoT system becomes crucial when a user can customize tasks on an IoT system. It is possible that a user may create one or more tasks that are conflicting in actions and has the potential to damage the IoT system. Consider the following two tasks as an example of unsafe customization of an IoT system.

Task 1: Temperature > 100 °F $\longrightarrow$ turn on the fan.

Task 2: Temperature < 110 °F $\longrightarrow$ turn off the fan.

For the temperature values between 100 °F and 110 °F, the fan has two states on and off. The IoT system executes both the actions and the fan continuously changes the state between on and off. These frequent changes in the state may reduce the efficiency and lifespan of the fan.

In this thesis, we use the term rules for tasks. According to research conducted by yang et. al [67], a system consisting of multiple rules is in a perfectly working situation if the rules do not have conflicts, incompleteness, and redundancies. A system with no conflicts and incompleteness is a safe system, and removal of redundant rules from the system reduces workload. We have designed a conflict detection system that detects conflicts, redundancies, and incompleteness. Our system performs three types of conflict and incompleteness checks to provide a safe environment for the IoT system. Our work detects the conflicts and

incompleteness in a boolean function based rule system, where a rule can be represented in a form of a boolean function.

Next, we shall consider the security issues in the user-customizable IoT system. Just like any other internet connected system, an IoT system requires confidentiality, authenticity, integrity, and privacy. Due to the limitations of computation power, providing security to an IoT device is a major challenge. In this thesis, we assume that the IoT server is secured and we only focus on securing the IoT device. We worked on the following two security problems:

- Side-channel attack resilient mutual authentication system between an IoT device and an IoT server.
- DDoS prevention technique for IoT devices

The mutual authentication between an IoT device and an IoT server is categorized into three types: RSA based mutual authentication, ECC based mutual authentication, and shared key-based mutual authentication. All of these methods are vulnerable to different side-channel attacks. Researchers have exhibited electromagnetic based side-channel attacks to steal keys of RSA and ECC based encryption [39][17]. Using side-channel attacks AES encryption keys can be stolen from IoT devices [64] [45].

In this dissertation, we propose a secure authentication protocol to authenticate the IoT device and the server. Some of the current authentication mechanisms, which are mostly based on single password-based mechanism, are vulnerable to side-channel and dictionary attacks. We have designed a multi-key authentication mechanism, such that, even if the secret key (or a combination of keys) used for ongoing authentication is retrieved successfully by the attacker, the attacker cannot gain access to the unused authentication keys and the authentication system is secure from the side-channel attack or similar attacks. The key values keep changing over time to prevent dictionary attacks.

Since the IoT devices are connected to the internet, such devices are vulnerable to attacks through the network. Mirai malware is an example of such attacks, where many IoT devices have been captured using default username and password to attack other internet based services [32]. Perakovic, et al. [46] have discussed the increased amount of DDoS attacks using IoT devices. DDoS attacks based on protocols like SSDP (a universal plug and play protocol), which is widely used in IoT devices, have increased significantly after 2013.

We present a method called login puzzle to prevent the unrestricted login attempts performed by a malicious script. The login puzzle is a collection of mini-puzzles, which requires a fixed amount of time and computation power to solve. When a malicious script tries to brute-force an IoT device, it needs to allocate a fixed amount of time and resources to solve the login puzzle, and every unsuccessful attempt increases the complexity of the login puzzle, hence increasing the time and computation power required to solve the next login puzzle. At a certain point, the login puzzle is hard enough that it is not practical for the script to solve. This method not only limits the number of attempts but also slows down the rate of attempts.

1.1 Our Contributions

In this work, we demonstrate mechanisms to create a robust user-customizable IoT architecture. First, we describe a method to called, self-aware sensor/actuator, to design user-customizable IoT architecture. Unlike most of the existing architecture, our architecture supports both hardware and software customization. Our method provides a unique way to represent parameters of sensors/actuators and let the user customize tasks using these parameters.

Later, we present a way to detect rule conflicts for the user-customizable IoT systems. We created an algorithm to detect conflicts, redundancy, and incompleteness in the rules. We discovered a new conflict called Independent Conflict, occurs when two different sensors, part

of two different rules and performing conflicting actions. We also introduced a mechanism to detect incomplete rules in the IoT system.

In the end, we introduced two methods to secure IoT authentication between 1) IoT device and server and 2) IoT device and user. For the first one, we have created a method called Secure Vault, which is a collection of passwords, and only a subset of them are used for each authentication, provides security against the side-channel attacks. Vulnerabilities in authentication protocol between the IoT device and user raise the chances of capturing IoT devices using DDoS malwares. We design a variant of the client puzzle, Login Puzzle, to slow down the capture of IoT devices. Our method ensures continuous resource allocation and minimal benign penalty.

1.2 Dissertation Roadmap

The rest of the text is organized as follows. We first describe the preliminaries & system architecture necessary to understand our work in Chapter 2. In Chapter 3, we discuss prior related work on user customization and robustness of the IoT systems. In Chapter 4, we describe our self-aware sensor/actuator method to create a user-customizable IoT system. In Chapter 5, we introduce methods to detect conflicts, redundancy, and incompleteness in user-customizable IoT rules. Chapter 6 and 7 mention about two authentication mechanisms, secure vault, and login puzzle. In the end, Chapter 8 provides the implementation details of the whole system.

CHAPTER 2

SYSTEM ARCHITECTURE

Researchers have proposed different ways to develop an IoT architecture. These architectures can be classified into two types: standalone and gateway based. In the first type, the IoT sensors/actuators directly communicates with the server over the internet [68]. In the second type, all the sensors/actuators are connected to a gateway device and that gateway device communicates with the server using a WAN interface [63] [12] [51] [23]. The first type is suitable for the application where the number of sensors and actuators are low.

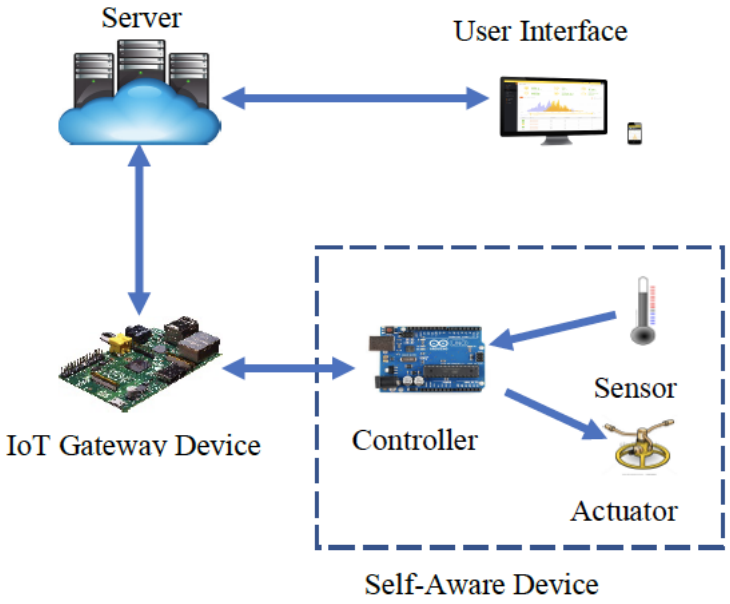

Figure 2.1. Gateway based architecture.

2.1 User Customization

We have designed user customizable IoT system, which supoorts both IoT gateway based and standalone architecture. The IoT gateway based architecture has four main components: One or more self-aware sensor(s)/actuator(s), an IoT device, a server (on the cloud) and a user interface. Fig. 2.1 shows the different components and their interactions.

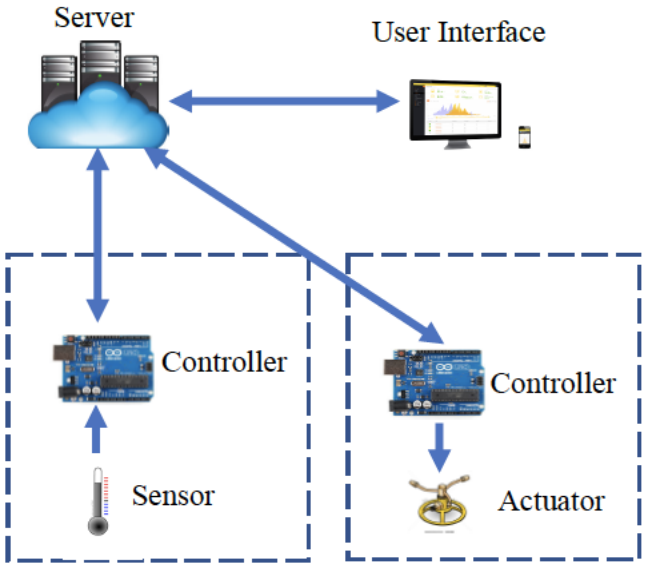

Figure 2.2. Standalone architecture.

Each self-aware sensor/actuator consists of a sensor/actuator with an additional processing element and related support components. The processing element stores basic details about the sensor/actuator, such as the id of sensor/actuator, type of sensor/actuator, parameters of sensor/actuator, etc. When the self-aware device is connected to the IoT device, all the details of sensor/actuator is communicated to the IoT device. The IoT device uploads these details to the server and obtains further instructions from the server on how to handle the newly connected sensor/actuator. Any number of self-aware sensors/actuators can be connected to the IoT device. The user can monitor all the connected sensors/actuators through the user interface and create tasks for those connected sensors/actuators.

The standalone self-aware architecture does not have the IoT gateway. The self-aware device (which can be either a self-aware sensor or a self-aware actuator) directly communicates with the server. The self-aware device must contain a network interface such as Wi-Fi/Cellular/Satellite modem to communicate with the server. In this case, the software (that processes the task/rule) runs on the server and communicates with the IoT device using web API. Fig. 2.2 shows the representation of this architecture.

2.2 Mutual Authentication

Fig. 2.1 and Fig. 2.2 illustrate typical IoT systems. These systems consist of three major components: An IoT device, an IoT server and a user interface. The IoT device is responsible for collecting the data generated by the sensors connected to it and uploading them to the server. In many cases, it also processes the data before uploading to the server. The IoT device communicates with the IoT server through a wide area network. This IoT system is accessible to the user using a web and/or a mobile interface. A secure communication is required (1) between the server and the user interface and (2) between the server and the IoT device. For the first link, https-based username and password method can be used. For additional security, a two-factor mechanism with one-time password can be used. (A multi-factor mechanism involving first two-factor authentication along with users fingerprint can be used to make the authentication more reliable and secure.) Providing a secure communication between the IoT device and the server is challenging because of memory and computational constraints of the IoT device. In addition, in many deployments, there is a constant traffic between the server and the device. Most of the public key encryption algorithms require high computation power and memory. SSL or any public key-based mechanisms require computational power that are beyond the capabilities of small IoT devices. Shared key encryption, which requires comparatively less computational power and memory is preferable.

2.2.1 Assumptions

1. We use an initial vault consisting of n keys of m bits each. Parameters m and n can be chosen by the developer based on security requirements and memory constraints.

2. IoT devices are constrained devices and have low memory, computational capabilities and low energy availability.

3. Each IoT device has a unique identification number assigned.

4. An initial secure vault is stored in the IoT device and this initial vault is shared with the server.

5. The server has a well-protected database.

6. Side-channel attacks can be performed by an adversary.

7. All the data exchanged between the server and the IoT device is over a reliable channel, so there is no data loss and every message is delivered exactly once.

2.2.2 Threat Model

The IoT server is deployed in the cloud and communicates with the client (or IoT device) over WAN. Single key based authentication mechanisms are not sufficient to authenticate the IoT device to the server. There are some side-channel attacks possible to retrieve the shared key during the communication between the IoT server and the IoT device. If the password does not change over the time, it is vulnerable to the dictionary attack. Once the adversary has the shared key, a fake device can be created using that shared key.

If an adversary can impersonate an original IoT device with a fake device, the fake device can inject fake data into the server. If the adversary impersonates an actuator, by knowing the incoming values to the actuator, the adversary can predict certain values of some sensors. For example, an IoT system may be configured as follows: if the temperature goes above 100 °F, turn on the fan controller. If the adversary impersonates the fan controller, every time it gets a signal from the server to turn on, it knows that the temperature value is above 100 °F. If the fake device acts as a sensor, the fake device can upload wrong sensor values and can trigger the actuator during undesired situations possibly causing a hazard. Not only the IoT device but also the server can be compromised as an attacker can impersonate itself as the server and can perform both attacks explained earlier. There are several ways an attacker can attack an authentication protocol; few of them are as follows:

- **Password Prediction:** An adversary can predict the next password from previously exchanged messages between the server and the IoT device. If the password does not change with time, the attacker can retrieve the password either via side channel attacks or dictionary attacks. If the password changes with time and if the newly selected password is not random enough from the previous password, an adversary can detect the pattern of the password change and can predict the next password.

- **Man in the middle attack:** An adversary acts as a middleman between the server and the IoT device and acts as a server to the IoT device and as an IoT device to the server. Using this attack, the adversary can authenticate itself as a server to the IoT device and as an IoT device to the server, thereby gaining access to the data exchanged between them. The adversary can also modify the data and inject wrong data to either the IoT device or the server.

- **Replay attack:** The adversary captures the authentication messages exchanged between the server and the IoT device and can use these captured messages to authenticate itself, if the security protocol doesnt consider the liveness and freshness of the message.

- **DoS attack:** The adversary floods the server with a large number of requests. The server may deploy a large amount of resources to respond to all the arriving requests and, after a certain limit, the server can crash due to high load. A good authentication protocol should not assign resources to the request before authenticating the client.

2.2.3 Our Authentication Architecture

In our protocol, we use a set of keys, called a secure vault, to authenticate both the server and the IoT device. This secure vault is initially shared between the server and the IoT device and it changes its values based on data exchanged between the IoT server and the IoT device.

Thus, contents of the vault change constantly. No additional message is exchanged between the IoT server and the IoT device to change the value of the secure vault. We use the standard 3-way mutual authentication for authenticating the IoT server and the IoT device. The communication is initiated by IoT device by sending a connection request to the server. When this request is received by the IoT server, it sends back a challenge to the IoT device; the IoT device responds to the IoT servers challenge and sends an authentication challenge to the IoT server. The IoT server verifies the response and, if it is valid, the server responds back to the IoT devices challenge.

During the authentication phase, the IoT server and the IoT device establish a shared secret, called a session key. This session key is used for two purposes. First, it is used to encrypt the messages exchanged between the server and the IoT device. It is also used as an encryption key for the message authentication code, which is used for message authentication. All the messages exchanged between two authentications considered as a session. The session key remains unchanged throughout a single session, but different sessions use different session keys.

CHAPTER 3

PREVIOUS WORKS

3.1 Related to User Customizable IoT Systems

There have been various IoT architectures introduce by researchers and industries provide different types customization. Customizable IoT systems is further divided into different categories. One way to categorize IoT system is level-based categorization. There are three levels of customization architecture level, developer level and user level. In architecture level customization, any change in IoT system, requires change in the architecture code. This is most basic level of customization. Next level is developer, where a developer can create, edit and delete new(or existing) sensor, actuator and gateway types. Most of these architecture provide a way for a user to configure simple automation on the customization created by a developer. In user customizable IoT system, a user can add, edit or remove new (or existing) sensor and actuator without writing any code.

A user customizable IoT system is divided into two types of customization: hardware and software. In hardware customization, a user can add/remove sensor or actuator to/from the system without knowing any hardware specific details of the newly added (or existing) sensor or actuator. In software customization, a user can specify tasks on existing sensors or actuators. A user doesn't need to write any additional code to create, edit or remove tasks from the system.

There have been several ways that researchers have conceptualized the idea of user customizable IoT architecture. Kortuem, G. et al. discuss the awareness of smart IoT objects [33]. In that paper, the authors describe how a normal sensor or actuator can be aware of its surrounding and can perform tasks according to the environmental change. They have presented ways using which a developer can pre-program the IoT sensors/actuators with environmental constraints. The user doesn't have any control over customization of an IoT

device. Several approaches have been presented on the unification of IoT devices. JADE architecture is an example of one such architecture where the developer can configure and customize the IoT service [18]. JADE provides an easy way to create an IoT device using their framework. The developer needs to write some code to define the IoT system and JADE creates the IoT system from the defined script. A restful service is created by Stirbu et. al. [61] for unified discovery, monitoring and controlling of smart things. Sarar et. al [55] have introduced an IoT architecture with virtual object layer. This virtual object layer is responsible for unifying heterogeneous IoT hardware. Alam et. al [2] have proposed an architecture named SenaaS, which creates a virtual layer of IoT on the cloud. Here, IoT sensors are considered as sensor as a service and it hides all the hardware specific details of sensor to the user. Kiran et. al [30] have designed a rule based IoT system for remote healthcare application. They have created a virtual software layer to execute rules on the sensor values. As their work focuses only on a single application (healthcare), they dont have any virtualization on hardware. There has been similar work done on rule-based IoT systems. An If-Then based rule implementation architecture is explained by Zeng, D. et. al. [14] The authors discuss the user configurable triggers for IoT systems. Amazon AWS IoT, AT&T M2X, IBM IoT, and Samsung SmartThings are examples of IoT architectures customizable by developers[4, 3, 54, 25]. Mainetti et al. [38] present an IoT architecture for the smart house, which can be configured and customized by a user. In this approach, the user can develop location-aware service. Khan et al. [29] have designed a DIY (Do it yourself) based IoT system, where a developer can build an IoT system using a drag and drop based user interface. Yang et al. [66] present an IoT system which enables users to customize and personalize IoT device.

3.2 Related to Rule Conflict

Yang et al.[67] present a petri-net based approach to detect redundant, conflicting and incomplete boolean expression based rules. Their work conceptualize that any boolean rule based system must detect conflicts,redundancies and incompleteness in order to work seamlessly. arreira et al. [9] introduce a policy based conflict detection and resolution system for home and building automation systems. ECA (event-condition-action) is a popular mechanism to execute a rule in an IoT system. In an ECA based rule architecture, events change the state of the IoT system and if this change triggers the condition, the actuator performs needed actions. Perumal et al. [47] present a framework to detect rule conflicts in a smart home system for ECA based rules. Oh et al. [44] have designed a framework to visualize ECA based rule conflict. Liang et al. [36] present an architecture named "SIFT", which provides a safe way to configure IoT devices. Shehata et al. [60] propose a semiformal method, called IRIS (Identifying Requirements Interactions using Semiformal methods), for detecting interactions between policies in the smart home domain. Hu et al. [24] propose a semantic web-based methodology, named "SPIDER", for automated detection of interactions for policies in smart homes. Both "IRIS" and "SPIDER" consider the rule as a policy defined by a user and, using policy management, detect conflicting and redundant policies (or rules) in an IoT system. Su et al.[62] present a scheme, called "UTEA" (user, trigger, environment and action), to detect rule conflicts in a smart house application. UTEA categorizes conflicts in 5 categories and rule relationships in 11 categories. Unlike other methods, UTEA considers user priority in conflict detection.

3.3 Related to Mutual Authentication

Many devices that are part of IoT system have constraints (such as computational, memory, energy, etc). IEFTs RFC 7228[7] has proposed an application layer protocol, CoAP for low

end IoT devices to connect to the internet. IEFT has also introduced low power security mechanism, DTLS (Datagram Transport Layer Security) [52] for secure communication over the CoAP protocol. Kothmayr et. al. [34] have proposed a two- way authentication system over the DTLS. The authentication system is based on the well-known RSA scheme, which is hard to deploy in severely constrained devices. Raza, et. al. [50] also present an approach based on the DTLS. They have adopted a shared key approach rather than RSA, which makes it more feasible for resource-constrained devices. Another approach suggested by researchers for IoT authentication is the use of ECC public key encryption [28]. Even though ECC requires less memory and computational power than other public key encryption methods, it requires more memory and computational power than shared key encryption. Also, some side channel attacks are possible for ECC . Barreto et al. [11] present an SSO based authentication system for authenticating the IoT device, the IoT cloud and the user. They create an authentication mechanism where the server authenticates the user using a user credential and then the server communicates with the IoT device (they mutually authenticate each other). After both end points (IoT device and the server) successfully authenticate each other, the server grants access. Porambage, et al. [48] introduced an ECC based two phase authentication algorithm. Butun, et al. [8] present an end to end cloud centric authentication mechanism for the IoT devices. They use ECC for authentication of the user to the IoT device. All the above authentication mechanisms use ECC based public certificate or ECC based Diffie and Heilman mechanism for authentication, which makes the authentication mechanism secure. Jan et al. [26] introduce a robust authentication mechanism using which the server and the IoT device can mutually authenticate each other. They use shared key approach to authenticate the server and the IoT device and a secret is shared between them. Their algorithm uses AES encryption for mutual authentication. If this single key is lost (or compromised), the two end points must replace their shared keys.

3.4 Related to Login Puzzle

Researchers have proposed few methods to use digital problems to avoid DDoS attacks. Most of these works secure server or web services from a DDoS attack. The very first work was introduced by Aura et al. as client puzzle [5]. In this paper, the authors introduced the concept of resource commitment at the client side. The client solves a puzzle given by the server with the authentication, before the server allocates any heavy resource for the client. While this method is very effective in stopping the DDoS attack at server side, it doesnt provide significant improvement at the host scanning phase because it uses problems of same complexity every time. If the complexity is too high, the benign user suffers from getting access of the IoT device and if the complexity is too low, the attacker can solve it easily and can brute force credentials to get the access of the IoT device. A varying complexity method helps benign user to access IoT device with low computation and restricts an attacker from brute forcing the credentials on the IoT devices. Michalas et al. [40] presented a similar concept in client puzzle to prevent DoS attack in an ad-hoc network. Another client puzzle was introduced by Abliz et al. [1] which protects a server from DDoS attack based on traffic created by the clients. Many approaches for creating client puzzle have been proposed by researchers. Chen et al. [10] have proposed a generic way to represent a client puzzle. In the paper authors have defined client puzzle as a tuple algorithm, which provides a formal setup to generate puzzle, solve puzzle and formal verification of the puzzle. Bogdan Groza et al. [20] have presented a formal verification for the puzzle hardness and bounds. Jing Yang Koh et al. [31] have used repeated squaring and hash reversal client-puzzle with the leaky bucket algorithm. Other than client puzzle, various security strategies at different stages of DDoS attack have been proposed by researchers to prevent the DDoS attacks [41] [27]. At the discovery stage, a DDoS attack can be prevented by closing the unused ports. At the network level, construction of a firewall restricts the spread of malware. IoT devices are made for performing specific tasks and require communication with specific domains only.

Continuous monitoring of running processes and network traffic of an IoT device provides enhanced security. This method constrains device activities and network traffic and causes false alarms when benign but unknown activities are happening. Various methods have been introduced by different researchers to prevent the DDoS attack on the victim side. There are methods called intrusion detection to detect a potential DDoS attack. There are other signature-based methods to detect the intrusion [22] [39]. Recently, machine learning based detection methods have been proposed to check the spread of the DDoS attack [49] [53].

CHAPTER 4

USER CUSTOMIZATION

User customization is a prominent feature of an IoT system. A fully customizable IoT system supports two types of customization: hardware customization and software customization. In hardware customization, a user can add/remove a sensor or an actuator to/from IoT system without knowing any hardware details of the sensor/actuator. Using software customization, a user can create automated tasks on the connected sensors and actuators. In this chapter, we describe an approach to achieve user customization using self-aware sensor/actuators.

4.1 Self-aware sensor/actuator

A self-aware sensor/actuator is basically a sensor/actuator with awareness of its

- type
- attributes
- capabilities
- constraints

A self-aware sensor/actuator requires additional components to be aware of the above properties. Typically, a self-aware sensor/actuator has the following components:

- Sensor/Actuator
- Processing Element
- Driver circuit for an actuator
- External memory (if processing element doesnt have sufficient memory)
- Communication interface to communicate with IoT gateway device
- Power supply (if sensor/actuator cant be powered by IoT gateway device)

4.2 Convert an arbitrary sensor/actuator into self-aware sensor/actuator

Steps to convert an arbitrary sensor/actuator into self-aware sensor/actuator:

- Select a suitable processing element
- Interface the sensor/actuator to the processing element
- Store properties of the sensor/actuator in the memory
- Create a communication interface for communication between the self-aware device and the IoT gateway device connected to it
- Create and embed self-aware device discovery code on the IoT gateway device and self-aware device to detect connected self-aware devices

4.2.1 Select a suitable processing element

Depending on the type of sensor/actuator, the developer selects the processing element. For example, if the type of sensor is analog, it is advisable to select a processing element which has an inbuilt analog to digital converter. Some sensors such as crack detection sensor or fingerprint sensor require a complex algorithm to read sensor value, and hence a processing element with significant memory and processing power is preferable. The power consumption of processing element is also important, as some installations may require a battery powered solution. For those applications, processing elements with power saving capabilities should be selected.

4.2.2 Interface sensor/actuator to processing element and program parameters of sensor/actuator into memory

The developer interfaces the sensor/actuator to the processing element depending on the type of the sensor/actuator. After that the developer programs the parameters of sensor/actuator into the memory. The parameter can be of two types: fixed parameters and user

configurable parameters. Fixed parameters dont change over time (such as the unique id of a sensor/actuator, type of a sensor/actuator or manufacturing date of sensor/actuator). User configurable parameters are the parameters which the user can define according to the application (for example, the time interval between two consecutive sensor readings). The user can define these user configurable parameters from the user interface.

4.2.3 Communication interface between self-aware device and IoT gateway device

The communication interface between the self-aware device and the IoT device should be user-friendly and known to the user. WiFi, USB, and Bluetooth are some examples. The processing element is connected to one such communication interface to communicate with self-aware sensor/actuator. The IoT device and self-aware sensor/actuator communicate using two methods: query-response method and interrupt method. In the query-response method, the IoT device queries the self-aware sensor/actuator and self-aware sensor/actuator responds it. In the interrupt method, the IoT device turns on the interrupt mode where the self-aware device sends messages to the IoT device without any query. We have developed a library for the query response model in embedded-c which is suitable for most embedded hardware used for self-aware device.

4.2.4 Self-aware device discoverable code for IoT gateway device

An IoT gateway device should be able to discover all the self-aware devices near it (for wireless connection) and all the devices connected to it (for wired connection). For different types of communication protocols, the method of discovering self-aware sensor/actuator changes. For example, for USB, the developer just needs to check the /dev/ttyUSB ports for checking connected USB device. For WiFi, a multicast signal with a certain message can be sent and all the nearby self-aware devices respond back. The developer should write code to enable

discovery for all the communication interfaces available on that IoT device. The developer also writes the code for self-aware device, so that it responds back to the IoT devices device discovery query.

4.3 Hardware customization

The goal of hardware customization is to enable the user to add or remove sensors/actuators as the needs change (instead of replacing the whole IoT system). To achieve such a goal, every time a new self-aware sensor/actuator is connected to the IoT gateway, the self-aware sensor/actuator should be able to identify itself to the gateway. The sensor/actuator knows the basic details about itself and, once connected to the IoT gateway, it communicates those details to the IoT gateway device.

As a part of hardware customization, the self-aware sensor/actuator provides its properties, which is categorized into the following three categories: attributes, capabilities and constraints. An attribute provides details about the sensor/actuator. Physical unit of a sensor is an example of an attribute. A capability defines the ability of a sensor or an actuator. For a sensor, minimum and maximum value a sensor can sensor read is capability of the sensor. For an actuator, capability is associated with the physical output of the actuator. For example, a motor runs with the maximum speed of 100 rpm; here, 100 rpm is the capability of the fan. Constraint is the opposite of the capability. A constraint defines the limitations of the sensor/actuator. For example, an air-conditioner requires 30 seconds of delay between two consecutive triggers. For a GPS sensor, it requires connection with at least four satellites.

Once the self-aware architecture knows the capabilities of a sensor, if a user creates a rule out of these capabilities, the rule engine can detect it and notify the user about the error. These additional properties provide more customization for the user. Thus, these properties makes the software customization more flexible and safe.

4.3.1 Validation of sensor/actuator value

Validation of sensor values is an important feature of the self-aware architecture. As the sensors/actuators are self-aware, they know their typical range of readings of the sensor outputs. These will also be stored as part of the sensor-specific data. If any sensed value is out of this range, the self-aware sensor notifies the user about the deviation. For example, a ds1820 temperature sensor [42] is connected to the self-aware device and the maximum value ds1820 temperature sensor can have is 125 °C. If the sensor reads more than 125 °C, the self-aware sensor itself needs to generate an error message (in addition to possibly sending the error reading to the server). Another example of self-validation is related to a self-aware actuator. Suppose we have a relay that controls a compressor as the actuator, which has been made self-aware. For compressor longevity, the compressor should not be turned on and off frequently. There should be a minimum elapsed time between two successive times when it is turned on [43]. The self-aware actuator knows this constraint and, if it receives too many commands for turning on and turning off the relay (actuator), it disregards the received commands and generates an error to inform the user. This property makes sure that neither the user nor the server needs to be aware of actuator-specific constraints. Instead, the self-aware device has the knowledge of the constraints and has checks and balances built in.

4.3.2 Working Status of Sensor/Actuator

It is important to know if the connected sensors/actuators are actually working or not. For a sensor, we can detect its working condition by its current value. If a sensor stops sending values or sends values that are out of range, we can conclude that that sensor is not working properly assuming that the communication channel is not faulty. This type of property only works on sensors which give analog values (e.g. temperature sensor or pressure sensor which gives output over a range). It is not possible to detect working status for some of the digital

sensors (for example touch sensor which gives either 1 or 0 as its output). We cannot decide whether the sensor is stuck at a single value or it is providing normal input. We can define such analog and digital sensors in the device property part of the unique id of the sensor and notify the user whether the user can get the working status of the sensor or not. Checking the health of an actuator is harder than that of a sensor. Many actuators may not provide any feedback to the processing element. However, this can be rectified by using auxiliary parts. For example, we can connect an appropriate new sensor to the actuator and retrieve values from the sensor. Using the output of the new sensor, we can check whether the actuator is working or not. Failure of the feedback sensor can raise a false alert.

4.3.3 Software Customization

Our architecture provides user customization of software where the user can create tasks/rules and set the user defined parameters for self-aware sensors/actuators. For example, the user can set how frequently the user wants to read the sensor values as a user-defined parameter. Rules are divided into two main components:

- Trigger Condition: When the trigger condition is true, the rule is executed by either IoT gateway device or the server. A single rule can have multiple trigger conditions. When a rule has more than one trigger condition, the rule is executed when all the trigger conditions are true or a specific combination of triggers occurs.
- Actions: This represents the actions to be taken when the conditions are true.

Some examples of actions our architecture supports are as follow.

- Send Text Message / App Notifications.
- Send Email
- Make phone calls and play notification message

- Turn on/off Actuator

A single rule can have more than one action. All the actions are executed when the rule is true. An example of full rule is: If (temperature $\geq$ 82 and humidity $\geq$ 40) then turn on the fan controller and send message and email This rule will turn on fan controller (a self-aware actuator) and send out the notifications when the value of temperature (a self-aware sensor) is more than 82 and the value of humidity (a self-aware sensor) is more than 40. While this is a simple if then else type of rule more complex rules are also possible.

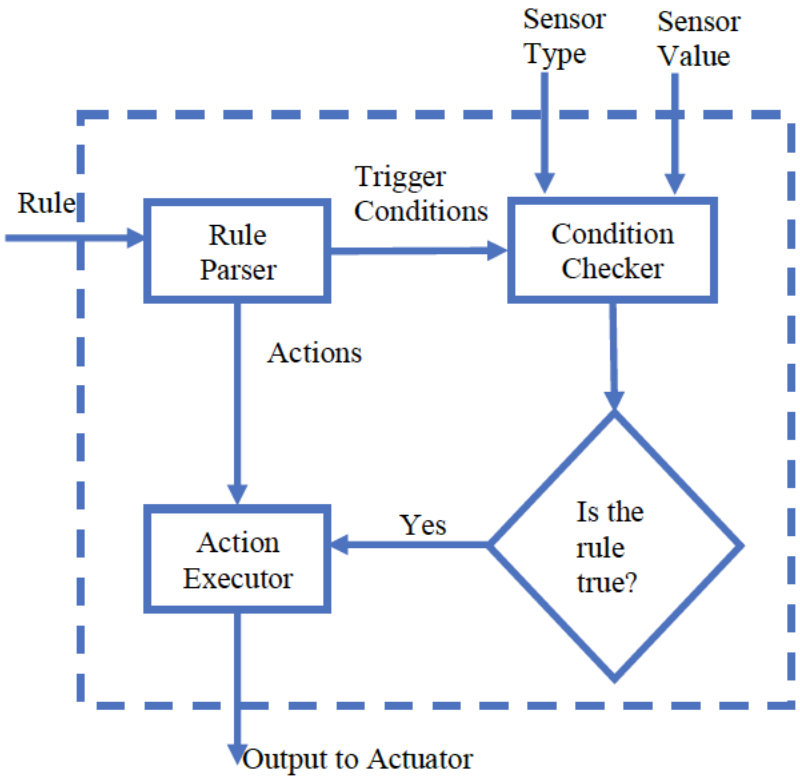

Figure 4.1. Structure of a rule engine

Execution of a rule

A rule can be executed either on the server or on the IoT device. For the standalone self-aware architecture, a rule must run on server because the standalone self-aware devices may

not be capable of running the rule engine. For the gateway based architecture, a rule can run either on the server or on the IoT gateway device. When all the self-aware sensors/actuators attached with the rule are connected to the same IoT gateway device, the rule can be executed on the IoT gateway device. When the self-aware sensors/actuators are connected to different devices, the rule is executed on the server. When a rule is executed on the server, slow or intermittent internet connectivity can cause problems in execution of the rule. When a user creates a rule, the rule is stored in our database in the form of different tables. Execution of the rule is done as follows:

If the server is executing the rule, it fetches the rule from the database and sends it to the rule engine. The rule engine is a service in our architecture, which takes the rule as input, fetches sensor values related to the rule and outputs appropriate commands to the actuator. If the IoT device is executing the rule, it fetches the rule from the server using REST calls and after that, it sends the rule to the rule engine running on the IoT device. The architecture of rule engine is shown in fig. 4.1.

The code executed by the IoT device or the server is generated automatically by our backend. The user doesnt write any code.

Table 4.1. Example properties of self-aware sensors/actuators

property	**type**	**datatype**	**readonly**
physical unit	parameter	string	yes
min value	capability	number	yes
max value	capability	number	yes
reading interval	parameter	number	no
manufacturing date	parameter	date	yes

4.3.4 Self-aware sensor/actuator properties

As described in the previous section, a self-aware sensor/actuator is aware of its properties. These properties can be constraints, capabilities or parameters of the sensor/actuator. A

Table 4.2. Properties of a soil moisture sensor

property	**type**	**datatype**	**readonly**
physical unit	parameter	string	yes
min value	capability	number	yes
max value	capability	number	yes
reading interval	parameter	number	no
precision	parameter	number	yes
min calibration value	parameter	number	no
max calibration value	parameter	number	no

property has the following attributes: type, datatype and readonly. Table 4.1 provides some examples of such properties. The type defines whether the attribute is of type constraint, capability or parameter. A readonly property can not be changed by the user. A user can customize task on only customizable properties. We have analyzed 36 different sensors and actuators and found 31 different properties. As an example, Table 4.2 shows properties for the soil moisture sensor.

The work discussed in this chapter is published in ICCOTWT 2018 conference [59].

CHAPTER 5

RULE CONFLICT

5.1 Our Approach

Users create automated tasks on IoT systems in the form of a rule. Usually, a rule is in "if-condition-then-action" form [35]. In the following section, we provide more details.

5.1.1 Rule Structure

A rule is a combination of trigger conditions and actions; when trigger conditions are true, the IoT system performs the actions. In an IoT system, most of the actions change the state of the IoT system from $S_{previous}$ to S_{next} and later, the system is required to return to $S_{previous}$ state. An action that changes the state of the system to $S_{previous}$ is called an anti-action and a trigger condition that triggers the anti-action is called an anti-trigger condition.

Fig. 5.1 shows an example of an action and an anti-action. In this paper, a rule is a combination of two sub rules: a trigger rule and an anti-rule.

- Trigger rule: This is the "if" part of the rule. It consists of trigger conditions and actions. Every rule must have a trigger rule.

- Anti-rule: This is the "else" part of the rule. It consists of anti-trigger conditions and anti-actions. A user can create a rule without an "else" part, so the anti-rule is optional.

The following is an example of a rule with an anti-rule: If temperature $\geq$ 100 °F, turn on the air-conditioner and turn off the air-conditioner when the temperature $\leq$ 80 °F. This rule contains the four components: a trigger condition, an action, an anti-trigger condition and an anti-action.

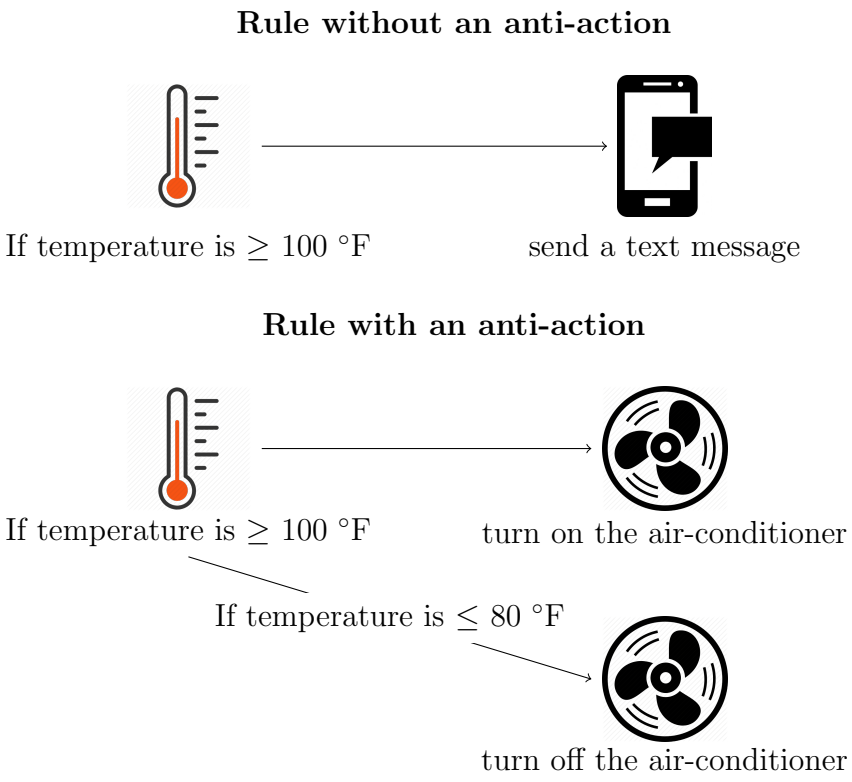

Figure 5.1. Visual representation of a rule with an anti-action.

- Trigger condition: The simplest trigger condition has three parameters: {sensor value, operator, threshold}. For the above rule, the trigger condition is "temperature ≥ 100 °F" with temperature as the sensor value, ≥ as the operator and 100 °F as the threshold.
- action: In the above example, the action is "turn on the air-conditioner".
- Anti-trigger condition: It is similar to the trigger condition, except it is part of the anti-rule. For the above example, the anti-trigger condition is "temperature ≤ 80 °F".
- anti-action: In the above example, "turn off the air-conditioner" is the anti-action.

Table 5.1 summarizes trigger-rule and anti-rule part of the above rule.

A rule is a combination of a trigger-rule and an anti-rule and both of them can be defined as $f(t_1,t_2,...,t_n) \implies a_1 \wedge a_2 \wedge ... \wedge a_n$. The function f is called trigger function and

Table 5.1. Rule description

	trigger-rule	**anti-rule**
trigger/anti-trigger conditions	temperature $\geq$ 100 °F	temperature $\leq$ 80 °F and rule is active
action/anti-action	turn on air conditioner	turn off air conditioner

it consists $\vee$, $\wedge$ and $\neg$ operations. $\vee$ represents a 'logical or', $\wedge$ represents a 'logical and' and $\neg$ represents a 'logical not' operation. For example, $(t_1 \wedge t_2) \vee (t_3 \wedge t_4) \vee t_5$ is a trigger function; where t_i represents a trigger condition consisting of a sensor value, an operator and a threshold. a_i represents an action and, when the function f returns a true value, all the actions, $a_1, a_2, ..., a_n$ are performed by the IoT system.

In our approach, we support the following operators for a trigger condition: $\leq$, $<$, $\geq$, $>$, $=$, $\neq$ and $\in$. All the other operators except $\in$ have only one threshold value and $\in$ has two threshold values representing a range.

5.1.2 Rule Conflict and Incompleteness Detection

Two rules conflict with each other when, for a certain range of sensor values, actions taken by both the rules conflict with each other. Actions are conflicting, if they satisfy one of the following conditions:

- Both the actions are associated with the same actuator and the actuator cannot perform both of them at the same time.
- Both the actions are associated with different actuators and one action is nullifying the impact of the other action.

Two actions, "turn on the fan" and "turn off fan" are examples of a conflict caused by the first condition. Turning on the heater and the air-conditioner at the same time is an

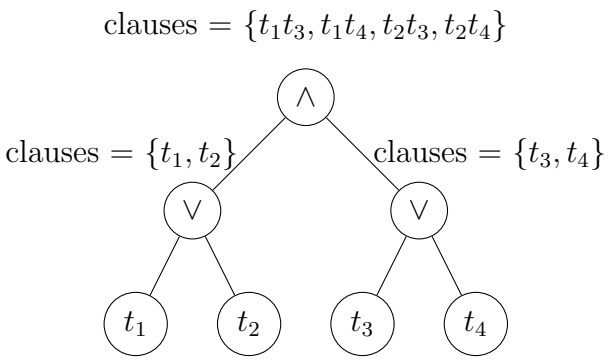

Figure 5.2. This figure is an expression tree for $(t_1 \vee t_2) \wedge (t_3 \vee t_4)$.

example of the second condition; turning on air-conditioner decreases the temperature and turning on heater nullifies its effect by increasing the temperature.

A set of rules are considered incomplete if, for any actuator, a sensor value range exists for which the actuator's action is not defined. In our rule structure, the anti-rule provides a method to create a complete rule.

In the following sub-sections, we describe the steps required to detect rule-conflicts and rule-incompleteness. First, we convert the rule to its DNF form using the distributed law property [13]. After that, for each actuator, we find overlapping DNF terms from the rules, and, if those rules perform conflicting actions, those rules are marked as conflicting rules. We next detect rule incompleteness. For an actuator, if all possible sensor value ranges are not defined, all the rules associated with that actuator are considered incomplete. To detect the incomplete rule, we use a variant of the covering polygon problem [19].

Table 5.2. Clauses Relationship Classification

relation	**abbreviation**	**definition**
Similar Sensors	sim_sensors(C_A, C_B)	$sensors_A = sensors_B$
Subset Sensors	sub_sensors(C_A, C_B)	$sensors_A \subset sensors_B$
Overlapping Sensors	overlap_sensors(C_A, C_B)	$\forall$sensor such that $(sensor \in A \wedge sensor \in B)$
Similar Action	sim_action(C_A, C_B)	$action_A = action_B$
Conflicting Action	conflict_action(C_A, C_B)	$action_A \neq action_B$

Algorithm 1 Algorithm to find DNF from an expression tree

```
1: procedure FindDNFClauses(node)
2:     if node → left is null and node → right is null then
3:         return node
4:     end if
5:     leftClauses ← FindDNFClauses(node → left)
6:     rightClauses ← FindDNFClauses(
                          node → right)
7:     clauses ← {}
8:     if node → operator is ∨ then
9:         clauses ← clauses.append(leftClauses)
10:        clauses ← clauses.append(rightClauses)
11:    else if node → operator is ∧ then
12:        clauses ← crossProduct(leftClauses,
                       rightClauses)
13:    end if
14:    return clauses
15: end procedure
```

Table 5.3. Trigger Condition Range Relationship

relation	**abbreviation**	**definition**
Overlapping Range	overlap_range(R_A, R_B)	$R_A \cap R_B \neq \varnothing$
Subset Range	sub_range(R_A, R_B)	$R_A \subset R_B$
Superset Range	sup_range(R_A, R_B)	$R_A \supset R_B$

Table 5.4. Conflict Definitions

conflict	**definition**	**clauses required**
Execution(C_A, C_B)	$[\forall (R_A, R_B) \in overlap\_sensors(C_A, C_B)$ overlap_range$(R_A, R_B)]$ $\wedge conflict\_action(C_A, C_B)$	any one
Shadow(C_A, C_B)	$sim\_sensors(C_A, C_B) \wedge sub\_range(R_A, R_B)$ $\wedge sim\_action(C_A, C_B)$	all
Independent(C_A, C_B)	$sim\_sensors(C_A, C_B)$ $\wedge conflict\_action(C_A, C_B)$	any one

5.1.3 Rule Decomposition to DNF

DNF (disjunctive normal form) is disjunction of conjuctive clauses. In other words, it is 'boolean or' (disjunction) of clauses which have all the terms in 'boolean and' (conjunction)

relationship. For example, $(t_1 \vee t_2) \wedge t_3$ is a function and the DNF of this function is $(t_1 \wedge t_3) \vee (t_2 \wedge t_3)$. In the above example, we have two clauses: $(t_1 \wedge t_3)$ and $(t_2 \wedge t_3)$. We use the distributed law of the boolean algebra to convert a rule into its DNF form. To convert any boolean function into its DNF, we first map the boolean function into an expression tree. Fig. 5.2 shows the expression tree for $(t_1 \vee t_2) \wedge (t_3 \vee t_4)$. The expression tree has $\wedge$ and $\vee$ operations as internal nodes, and the leaf nodes are the terms (in our case, condition statements) of the expression. DNF is created using depth first search traversal starting at the root node of the tree with the following criteria;

- If the current node is $\vee$, create two separate clauses, one represents the left sub-tree of the node and the other represents the right sub-tree of the node.
- If the current node is $\wedge$, the current clause is the cross-product of left sub-tree clauses and right sub-tree clauses.
- If the current node is a leaf node, return the node.

Algorithm 1 describes the recursive traversal for obtaining the DNF clauses.

A clause is a collection of trigger conditions; each trigger condition consists of a sensor value, a threshold and an operator. Based on the operator and the threshold, a trigger is associated with a sensor value range, for which boolean value of the trigger condition is true. For operators $>$, $\geq$, $<$ and $\leq$, the sensor value ranges are (threshold, ∞), [threshold, ∞), ($-\infty$, threshold) and ($-\infty$, threshold] respectively. For the $\in$ operator, the sensor value range is (lower_threshold, upper_threshold). Lastly, for $=$ and $\neq$ operators the sensor value ranges are (threshold, threshold) and ($-\infty$, ∞) - $\{threshold\}$ respectively.

5.1.4 Rule Conflict Classification

We divide rule conflicts into three categories: execution conflict, independent conflict and shadow conflict. In the execution conflict, for one sensor value, two different rules have

conflicting actions. Independent conflict is similar to execution conflict except instead of having overlapping ranges, sensors associated with both the rules are distinct. When distinct sensors from two rules performs conflicting actions, there are chances that they both independently trigger their rules at the same time and, it results in a conflict. Shadow conflict detects redundancy; both the rules have similar actions and sensor value range of one rule is a subset of another.

Execution Conflict

Two rules are in execution conflict if they have a) overlapping sensor value ranges and b) conflicting actions. To detect overlapping sensor value ranges, all the clauses of both the rules are compared with each other and if one pair has an overlapping range, both the rules share an overlapping range. The following example shows an execution conflict between two rules.

Rule 1: if soil moisture is less than 40%, turn on the sprinkler.
Rule 2: if soil moisture is greater than 30%, turn off the sprinkler.

For the values between 30 and 40 %, rules are in execution conflict, as both the rules are performing conflicting actions.

Shadow Conflict

Shadow conflict detects redundant rules. If one rule is the superset of the other rule, eliminating the second rule will not affect the system [62]. Two rules are in shadow conflict if a) trigger conditions of one rule is a subset of another rule and b) both rules have same action. For the shadow conflict, all the clauses of the redundant rule must be a subset of the other rule. The following example shows a shadow conflict between two rules.

Algorithm 2 Algorithm to Detect Clause Conflicts

```
1: procedure DetectClauseConflict(clause1,clause2)
2:     executionConflict ← False
3:     shadowConflict ← False
4:     independentConflict ← False
5:     sensors1 ← clause1.getSensors()
6:     sensors2 ← clause2.getSensors()
7:     action1 ← clause1.getAction()
8:     action2 ← clause2.getAction()
9:     conflictingAction ← conflict(action1, action2)
10:    similarAction ← similar(action1, action2)
11:    if conflictingAction then
12:        if sensor1.range ⊂ sensor2.range then
13:            executionConflict ← True
14:        end if
15:        if overlap(sensor1.range, sensor2.range) then
16:            executionConflict ← True
17:        end if
18:        if sensor1 ∩ sensor2 ≠ ∅ then
19:            independentConflict ← True
20:        end if
21:    end if
22:    if similarAction then
23:        if sensor1.range ⊂ sensor2.range then
24:            shadowConflict ← True
25:        end if
26:    end if
27:    return executionConflict, shadowConflict, independentConflict
28: end procedure
```

Rule 1: if soil moisture is less than 30%, turn on the sprinkler.

Rule 2: if soil moisture is less than 20%, turn on the sprinkler.

The second rule is redundant. The sprinkler is to be turned on for all the values below 30% based on the first rule; hence, all the values for the second rule are covered by the first rule.

Algorithm 3 Algorithm to Detect Rule Conflicts

```
1: procedure DetectConflict(rule1,rule2)
2:     clauses1 ← rule1.getClauses()
3:     clauses2 ← rule2.getClauses()

4:     executionConflict ← False
5:     shadowConflict ← True
6:     independentConflict ← False

7:     for c1 in clauses1 do
8:         for c2 in clauses2 do
9:             conflicts ← CLAUSECONFLICT (c1, c2)
10:            if conflicts.executionConflict then
11:                executionConflict ← True
12:            end if
13:            if conflicts.independentConflict then
14:                independentConflict ← True
15:            end if
16:            if not conflicts.shadowConflict then
17:                shadowConflict ← False
18:            end if
19:        end for
20:    end for
21:    return executionConflict, shadowConflict, independentConflict
22: end procedure
```

Independent Conflict

Two rules are in independent conflict if they have a) distinct sensors and b) conflicting actions. As in execution conflict, if any clause from the first rule has an independent conflict with any of the clause from the second rule, both the rules are in independent conflict. The following example shows the independent conflict.

Rule1: if weather forecast predicts more than 5 mm rain within an hour, turn off the sprinkler.

Rule2: if soil moisture is less than 20%, turn on the sprinkler.

When soil moisture value is less than 20% and weather forecast predicts more than 5 mm rain, the sprinkler has to be both on and off, which is a conflict.

After converting the rule into the DNF, it is a composition of multiple clauses and the relation between these clauses decide the conflict between rules. Table 5.2 presents possible relations between a pair of clauses. These clauses are collections of trigger conditions and the boolean value of a clause is true when all of its trigger conditions are true. Table 5.3 describes relationship between sensor value ranges.

Table 5.4 defines the conflicts and the third column presents how many clauses must conflict to have a rule conflict. In the execution conflict, all the common sensors between two clauses must have overlapping ranges and rules must have conflicting actions. Any clause can trigger a rule independently; hence any clause from the first rule conflicts with any clause from the second rule resulting in a rule conflict. Shadow conflict detects the redundancy of a rule. If all the ranges of the first clause are subsets of the second clause, and they both perform the same action, then we can remove the first rule because the second rule covers all the cases of the first rule. Unlike an execution conflict, shadow conflict requires all of the clauses of the first rule to be a subset of at least one of the clauses of the second rule. Independent conflict, like the execution conflict, requires only one clause from the first rule in conflict with any of the clauses from the second rule. Algorithm 2 and algorithm 3 show the psuedo-code for detecting the clause and the rule conflicts, respectively.

5.1.5 Rule Incompleteness

Rules are considered incomplete if they do not cover all the possible sensor values. The following examples demonstrate incomplete rules.

Example 1

Rule 1: if temperature is greater than 70 °F, turn on air-conditioner.

Example 2

Rule 2: if temperature is greater than 70 °F, turn on the air-conditioner and when temperature reaches 60 °F, turn on the air-conditioner.

The first rule is clearly incomplete because the rule is not covering all the possible values of the temperature sensor. For the second rule, by mistake the user provides a wrong anti-action, and instead of turn off, the anti-action is turn on. The following example demonstrates a complete rule.

Rule 3: if temperature is greater than 70 °F, turn on the air-conditioner and when temperature reaches 60 °F, turn off the air-conditioner.

Rule 3 covers the whole range, and provides a condition to perform the opposite action. In rule 3, the actuator maintains its previous state between 60 °F and 70 °F. We define this range as the guard range. If the rule does not have a guard range and values continuously oscillate near the threshold value, the actuator state may continuously change. The rapid change in the actuator state may decrease the lifetime of the actuator. A guard range provides a way to change actuator's state less frequently without affecting the functionality.

When multiple sensors are associated with a rule, it is not required to cover the whole range of all of the sensors. The range coverage depends on the boolean function of the trigger conditions. To understand this, consider the following examples:

Example 1:

Rule 1: If soil moisture is less than 20% and time = 7 AM, turn on sprinkler.

In the above example, both sensors, soil moisture and time must be in a specific range to turn on the sprinkler. To complete the rule, the anti-rule requires only one sensor's range to complete. One of the following two rules completes the above rule.

Rule 2: If soil moisture is less than 20% and time = 7 AM, turn on the sprinkler and turn off the sprinkler when soil moisture is greater than 40%

Rule 3: If soil moisture is less than 20% and time = 7 AM, turn on the sprinkler and turn off the sprinkler at 7:10 AM

Example 2
Rule 1: If soil moisture is less than 20% or time = 7 AM, turn on the sprinkler.

For the above rule, both soil moisture and time can trigger the sprinkler independently; hence, we need both sensors to cover the whole range. Adding the following rule in the system completes the above rule.

Rule 2:
If soil moisture is greater than 30% or time is 7:10 AM, turn off the sprinkler.
From the above two observations, we can conclude,

- If multiple sensors are in 'boolean and' condition with each other, covering the whole range of only one of the sensors is enough to cover all possible sensor values.
- If multiple sensors are in 'boolean or' condition with each other, all the sensors need to cover whole range.

If we combine the above conclusion with the DNF form of a boolean function, a complete rule has the following properties.

- Complete clause: For a complete clause, at least one sensor is required to cover the whole range.
- Complete rule: Each clause of a rule must be complete.

The following expression defines the completeness of a clause C.

$\exists C_1, C_2, .., C_n$ such that $\bigcup C, C_1, C_2, ..., C_n$ covers all the possible values for one or more sensors $\wedge$ $\forall i$ $sub\_sensors(C_i, C)$.

The C_i can be part of the anti-rule of the given clause or other rules defined in the system. $sub\_sensors(C_i, C)$ defined as C_i contains all the sensors from C.

Finding whether such subsets of clauses exist, which complete a clause is a variant of well-known computational geometry problem known as covering polygon problem [19]. In this problem, the goal is to find a covering of the given polygon with the smallest number of rectangles. To check completeness, instead of finding smallest number of rectangles that cover the given polygon, we check whether a given set of rectangles cover the polygon or not. We keep adding non-overlapping clauses to the covering set until all the possible ranges of one or more sensors are covered. If, after adding all the possible clauses, the whole range of at least one sensor is not covered, then the given clause is incomplete. As the covering polygon is a separate topic for the research and has well-established solutions [16] [37], in this paper, we will not include the implementation of the covering problem.

5.2 Rule Conflict Resolution

The resolution of the rule conflict is an essential part of a rule engine. The naive approach to resolve a rule conflict is removing or disabling the conflicting rules. In this section, we discuss few methods to avoid conflicts between the rules: priority method and adjusting rule sensor values.

5.2.1 Priority Method

In the priority method, every rule has a priority and the one with the higher priority is executed in the event of a conflict. The priority method changes the definition of the rule conflicts. All the rule conflict definitions require an additional condition named 'similar_priority'. This condition is true, when both rules have the same priority. If both the rules have different priorities, the one with higher priority wins and there is no conflict. $sim\_priority(C_A, C_B)$ is added at the end of each rule conflict definition in Table 5.4.

Priority method is suitable for the IoT systems with the overriding triggers. Consider a smart home system with the following rules:

Rule 1: Close the window after 10 PM and open it again after 8 AM.

Rule 2: Open the window when there is a fire at home.

The above rules exhibit an independent conflict. A fire hazard between 10 PM and 8 AM leads two state values for the window at the same time. Clearly, the second rule must override the first rule, hence by providing higher priority to the second rule avoids the conflict.

5.2.2 Adjusting Rule Sensor Values

For some cases, by adjusting one or more sensor values, the conflict can be resolved. We next discuss the resolution of execution conflicts by adjusting the rule sensor values. Consider the following two cases.

Case 1: If temperature is $> 60°$F turn on the air-conditioner and if temperature is $< 65°$F

turn off the air-conditioner. In this case, adjusting the range of one of the ranges can avoid the conflict.

Any of the following changes can avoid conflict.

1. The first condition changes to temperature is $> 65°$F.
2. The second condition changes to temperature is $< 60°$F.
3. change the ranges of both the conditions and meet in the middle; both conditions change to temperature is $> 62.5°$F.

Case 2: If temperature is $> 60°$F turn on the air-conditioner and if temperature is $> 65°$F turn off the air-conditioner. In this case, changing the 'greater than' symbol to 'less than' in the first condition changes the range of the first condition to $(-\infty, 60)$, which is not overlapping with the range of the second condition.

The suitable method to resolve the independent conflict is priority method and for the execution conflict suitable method is adjusting the rule sensor values. In the case of shadow conflict, the resolution is simple: remove the rule which is a subset of the super rule.

The work discussed in this chapter is accepted in IEMCON 2019 conference and will be published in IEEE digital library [57].

CHAPTER 6

MUTUAL AUTHENTICATION

Mutual Authentication between the IoT device and the IoT server is an essential part of securing an IoT device. In this chapter, we present a novel approach to secure a mutual authentication from side channel attacks. We use 3-way mutual authentication as a template for our approach and provided solution to make it secure from side-channel attacks. We use a set of passwords (or keys), which we define as a 'secure vault' to mitigate the side-channel attacks. Every authentication request uses a subset of secure vault to authenticate, which prevents a side-channel attack from getting the whole secure vault. The value of secure vault is changed frequently and the design of the secure vault prevents an attacker from retrieving the whole secure vault from the partial values collected from the side-channel attack.

6.1 3-way mutual authentication

3-way mutual authentication is a well-known state of art technique to mutually authenticate two parties. It uses shared key encryption, which are generally lightweight in terms of computation and energy consumption, makes it a suitable candidate for authentication of IoT system. 3-way authentication uses the challenge-response method to verify the authenticity of the other party. In challenge-response method, one entity challenges the other entity by sending a random number and other entity encrypts that random number with the secret key shared between these two entities. In 3-way mutual authentication, both parties uses challenge-response mechanism to mutually authenticate each other. The 3-way mutual authentication requires 3 messages.

- Message 1: Entity 1 challenges entity 2
- Message 2: Entity 2 responds to entity 1 and challenges entity 1 with another challenge

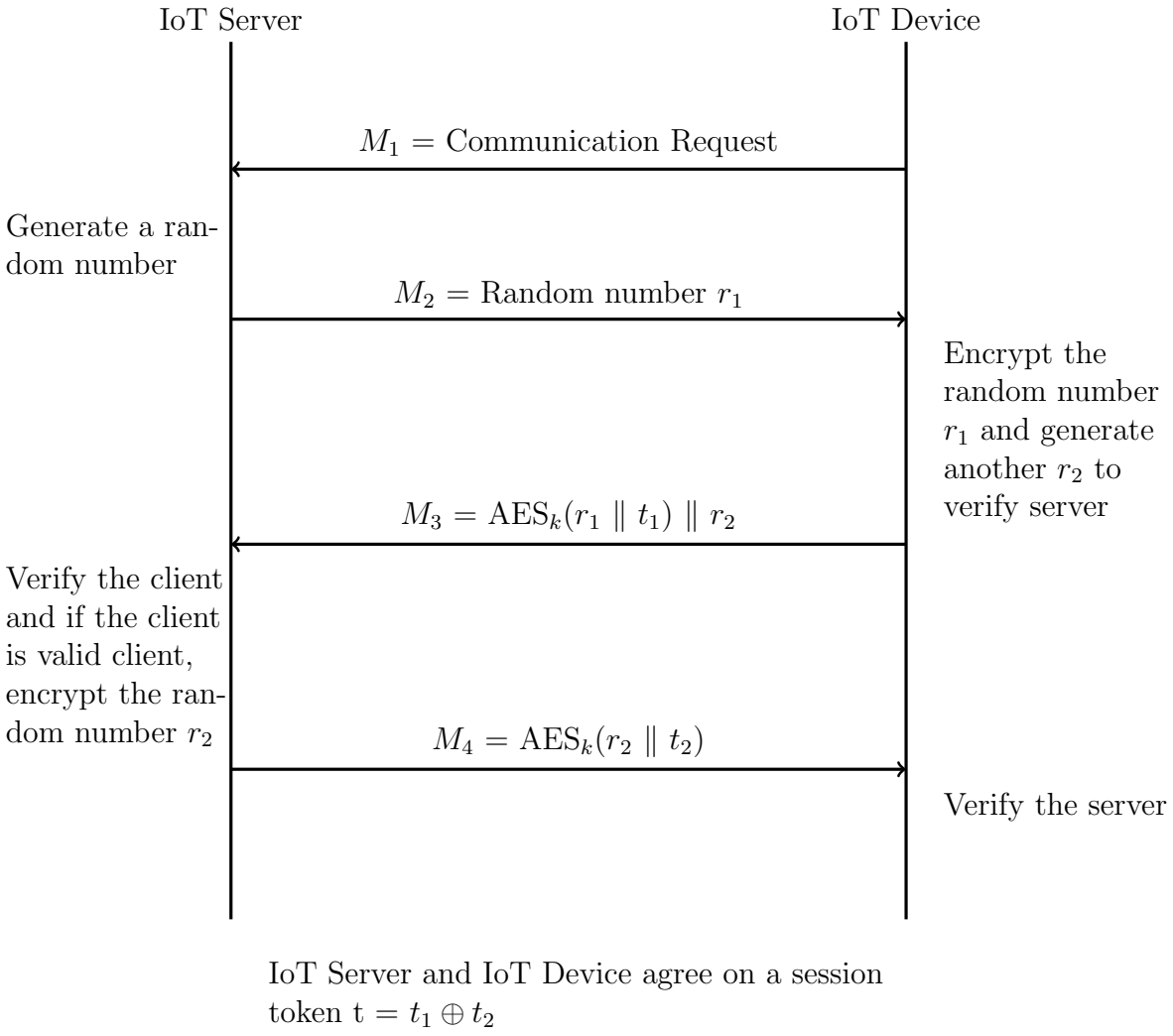

Figure 6.1. Message exchange for 3-way mutual authentication

- Message 3: Entity 1 responds back entity 2

The IoT server and the IoT device exchanges 4 messages to mutually authenticate each other. Figure 6.1 shows the message exchange for the 3-way mutual authentication.

6.2 Our Approach

Our protocol uses a set of keys, called a secure vault, to authenticate both the server and the IoT device. This secure vault is initially shared between the server and the IoT device

and it changes its values based on data exchanged between the IoT server and the IoT device. Thus, contents of the vault change constantly. No additional message is exchanged between the IoT server and the IoT device to change the value of the secure vault. We use the same 3-way mutual authentication for authenticating the IoT server and the IoT device. The communication is initiated by IoT device by sending a connection request to the server. When this request is received by the IoT server, it sends back a challenge to the IoT device; the IoT device responds to the IoT servers challenge and sends an authentication challenge to the IoT server. The IoT server verifies the response and, if it is valid, the server responds back to the IoT devices challenge.

During the authentication phase, the IoT server and the IoT device establish a shared secret, called a session key. The session key is used to encrypt messages exchanged between the server and the IoT device after the authentication phase. All the messages exchanged after the authentication phase are considered as a session. The session key remains unchanged throughout a single session, but different sessions use different session keys.

6.2.1 Secure Vault

The secure vault contains n keys each key being m bits long. The value of m is the key size. We denote all the keys as $k[0], k[1], k[2], ..., k[n-1]$. During the time of deployment of the IoT device, the secure vault is shared between the IoT device and the server. On the IoT device, the secure vault should be stored in an encrypted format. On the server, secure vaults are stored in a secure database.

6.2.2 Challenge Response

Our protocol uses a variant of the well-known three-way authentication mechanism to mutually authenticate the IoT server and the IoT device. Figure 6.2 represents the messages exchanged between the server and the IoT device. The IoT device initiates the process by

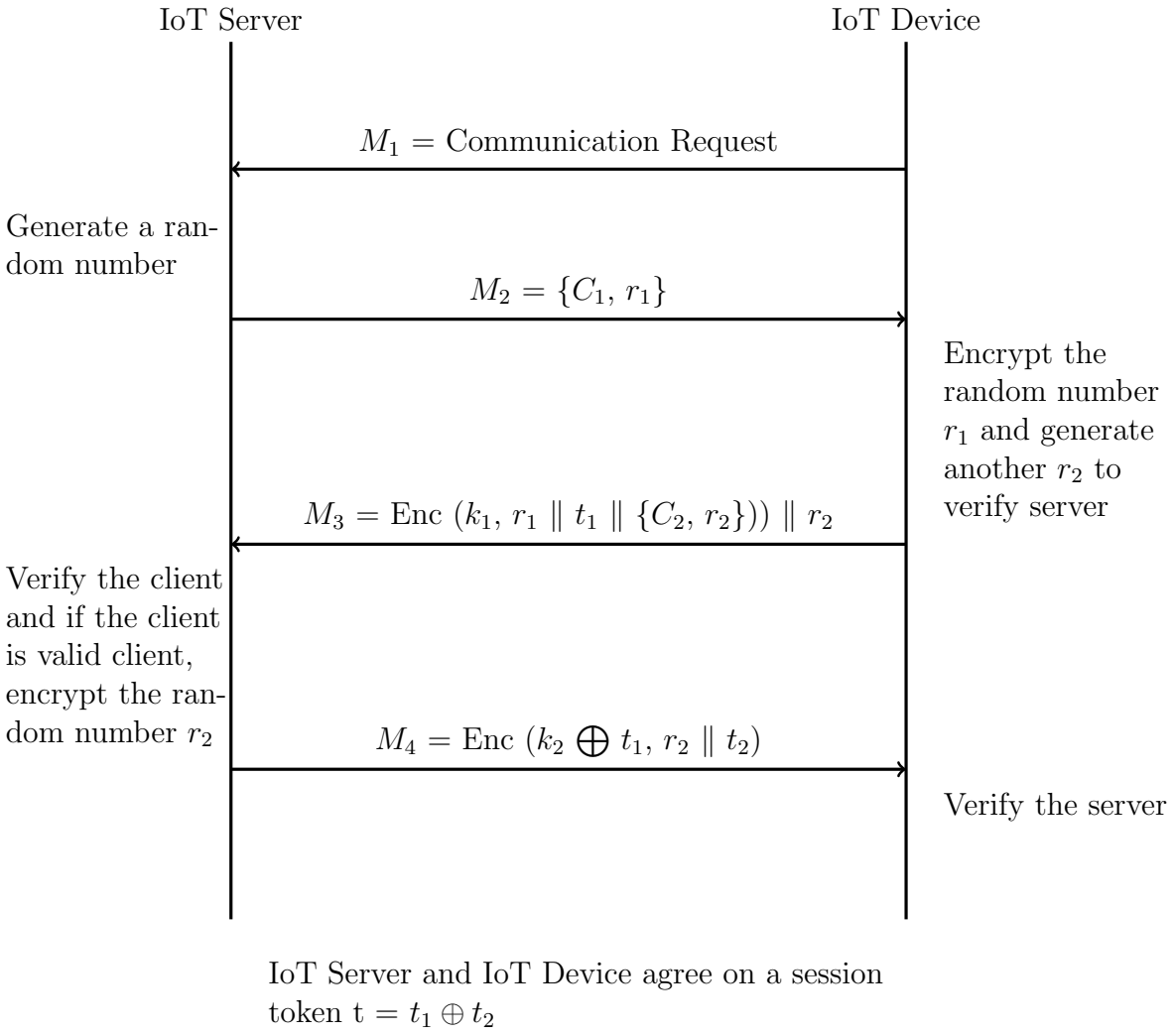

Figure 6.2. Message exchange for secure vault mutual authentication

sending the request message M_1 to the server. The request message contains the unique id of the IoT device and a session id to maintain the authentication session. This message doesnt contain any sensitive information and, the message is not encrypted. The server verifies the unique id of requesting IoT device and, if the message contains the valid unique id, the server sends back a challenge message M_2 to the IoT device. The challenge message contains a challenge C_1 and a random number r_1. C_1 is a set of p distinct numbers, and each number represents an index of a key, stored in the secure vault. C_1 is denoted as $c_{11}, c_{12}, c_{13}, , c_{1p}$. The value of p should be less than n.

$M_2 = \{C_1, r_1\}$ The values contained in C_1 are between 0 and n-1. Again, each element in C_1 represents an index of a key stored in the secure vault. The IoT device generates the response as follows: First, it generates a temporary key k_1 of size m bits by performing XOR operation on all the keys whose indices are in C_1. Thus, $k_1 = k[c_{11}] \bigoplus k[c_{12}] \bigoplus ... \bigoplus k[c_{1p}]$.

The IoT device creates the response for the challenge by performing shared key encryption on $r_1 \parallel t_1$ using k_1 as the encryption key. Note that $\parallel$ is the concatenation operation and $\bigoplus$ is the bitwise xor (exclusive OR) operation. Here, t_1 is a random number generated by the IoT device, which is further used to generate a session key t. This session key t will be part of the authentication key for the given session. In the security section, we discuss how the session key t helps in preventing man in the middle attacks.

The IoT device also generates a separate challenge for the server using the same mechanism. The IoT device generates a challenge C_2, another set of p distinct random numbers, each number being between 0 and n-1 and a random number r_2. Set C_1 and C_2 are different. If C_1 and C_2 are same, an attacker can get the key used for the C_1 challenge, and he can reuse that key for C_2.

The IoT device concatenates the response and the challenge and sends it back to the server. Message from the IoT device to the Server: $M_3 = \text{Enc}\ (k_1, r_1 \parallel t_1 \parallel \{C_2, r_2\})$
Where,
Enc: shared key encryption
$k_1 = \text{P}[c_{11}] \bigoplus \text{P}[c_{12}] \bigoplus ... \bigoplus \text{P}[c_{1p}]$ is the key for the encryption
$C_2 = \{c_{21}, c_{22}, , c_{2p}\}$
r_2 = random number for the challenge C_2
t_1 = random number for session key generation

Once the server receives the message M_3, the server verifies the message sent by the IoT device by generating the key k_1 from its secure vault. If the server retrieves r_1 from the

received message, it generates a response to the challenge C_2.

The message sent by the server back to the IoT device is:

M_4 = Enc ($k_2 \bigoplus t_1$, $r_2 \parallel t_2$)

Where,

k_2 = P[c_{21}] $\bigoplus$ P[c_{22}] $\bigoplus$ ... $\bigoplus$ P[c_{2p}] and $k_2 \bigoplus t_1$ is the key for the encryption

t_2 = random number for session key generation

The IoT device receives the message M_4 and it verifies the identity of the server by getting back the value of r_2 by decrypting the message M_4 using $k_2 \bigoplus t_1$. Once the server and the IoT device authenticate each other, they decide on a session key t = $t_1 \bigoplus t_2$ and all the further communication for this session is securely encrypted using this session key.

6.2.3 Changing the secure vault

The duration of the session is determined by the user: shorter duration gives high security, while resulting in frequent invocation of the 3-way authentication message exchange. After every session, the value of secure vault is changed based on the data exchanged between the server and the IoT device. The new value of secure vault is generated by performing HMAC on the contents of the current secure vault value. HMAC [23] is a key based hashing algorithm. Following are the steps to change the secure vault:

- We take the HMAC of the current secure vault and the key for the HMAC is data exchanged between the server and the IoT device. The hash function used here provides the output of k bits. This HMAC value is denoted as h.

  h = HMAC (current secure vault, data exchanged)

- The current value of secure vault is divided into j equal partitions of k bits, called vault partitions. All these partitions are xored with the h $\bigoplus$ i to generate the new secure

vault (here i is the index of the vault partition). If the size of the secure vault is not divisible by k bits, 0 is padded at the end to create j equal partitions.

Newly generated secure vault is random from the previous secure vault because of the following reason:

- The h value is created using a hash function, which generates a random number based on data exchanged between the server and the IoT device.
- The one-time pad theorem says that If data is xored with a random number, the generated output will be a random number. The contents of the new secure vault are generated by performing xor operation with a random number(h), and hence the newly generated data is random.

6.2.4 Securing vault on hardware

If an intruder gets the access to an IoT device, the intruder can access the secure vault stored in the IoT device and will have ability to create a duplicate device using the stolen secure vault. To prevent this attack, secure vault should be encrypted. Our algorithm encrypts the secure vault with a composite encryption key, which is a combination of three different keys at three different levels.

- Hash of hardware specific key (i.e. mac address, processor serial number or separate key stored in processor memory)
- File specific key (key stored in a specific file)
- Firmware specific key (key embedded in the firmware)

The Firmware specific key and hardware specific keys are static and do not change over time. The file specific key changes over time by the firmware, which makes the final key

random over time. To decrypt the secure vault, the intruder must know the hardware specific key and there are hardware modules known as trusted platform modules, which stores hardware specific keys securely. AT97SC3204/AT97SC3205 from Atmel company is an example of such a module. Thus,the final key is also a combination of three random keys, out of which one key changes over the time, thereby making dictionary attack harder for the attacker.

6.3 Security analysis of our approach

There are multiple ways to breach the security of an authentication protocol. Depending upon the nature of the authentication protocol, different types of attacks are possible. The most common attacks on an authentication protocol are man-in-the-middle, next password prediction, side channel attack and DoS attack. In this section, we show that our authentication mechanism is safe from all these attacks.

6.3.1 Man in the middle attack

The man-in-the-middle can capture all the messages exchanged between the server and the IoT device using network spoofing. After spoofing all the messages exchanged for the authentication, it can identify itself as the server to the IoT device and as the IoT device to the server. In our protocol, we are using a session key t to authenticate all the messages exchanged after authentication. This session key t is generated using two separate random numbers t_1 and t_2, which are exchanged between the server and the IoT device in encrypted messages. The key for those encrypted messages is a part of the secure vault, which is secretly shared between the server and the IoT device. Hence, the man-in-the-middle cant retrieve session key t from the messages exchanged between the server and the IoT device and is unable to retrieve or modify any messages exchanged between the server and the IoT device after the authentication.

6.3.2 Next password prediction

After every successful session, the IoT device and the server change the values of the secure vault based on the data exchanged between them. The new value of the secure vault must be random from the previous vault value. If some passwords from a secure vault are predicted/retrieved by the adversary, the adversary should not able to predict any other password of the next secure vault. We prove that the next password prediction is not possible using random oracle model.

In the random oracle model, hash functions are assumed as random oracles. A random oracle has following properties:

- it takes an input x and generates a random output y.
- For every different value of x, it generates a different value y.
- Every time input x is provided to the random oracle, it generates the same output y.

Thus, all the outputs provided by the random oracle are random and from the output provided by the random oracle, it is not feasible to predict the input. When creating a new secure vault, first we take the keyed hash(HMAC) of previous secure vault with the data exchanged between them as the key. The random oracle generates a new random value, every time a secure vault value is provided to it. This newly generated random value is xored with the previous vault values. According to the one-time pad theorem, if we xor any value with a random value, the generated output value will be random. Thus, the newly generated secure vault values are random from previous values and the attacker cant predict any value of next oracle even if attacker knows a part of previous secure vault.

Side channel attack

Some side channel attacks exist in well-known shared key encryption. For example, side channel attacks based on power analysis, temperature analysis and memory analysis can

break AES [13,21]. For the single password based authentication system, the attacker can retrieve the AES encryption key involved in the challenge-response using the side channel attack. In our protocol, the AES encryption key is the combination of multiple keys xored with each other and it is not possible to retrieve back those keys from the encryption key. There is no way an attacker can know values of the keys involved in the authentication by just knowing the encryption key. Hence, it is not possible to retrieve the whole secure vault from the side channel attack and create a duplicate IoT device or inject a false message to the channel.

6.3.3 DoS attack

An attacker can either flood the server or the IoT device with a large number of fake requests and crash it due to resource constraint. In our architecture, we are not assigning any resource before the authentication, so the DoS attacks are not possible.

6.4 Performance Analysis

We did power, memory and security analysis for our algorithm. We compared our algorithm with ECC (Elliptic Curve Cryptography) based public key encryption mechanism and a simple 3-way authentication mechanism with changing keys after every successful data exchange using energy consumption as the comparison measure. ECC is a light weight public key encryption scheme often used for IoT devices.

6.4.1 Power Analysis

We used the method described by Prasithsangaree, et. al. [20] to measure the power consumption. Their method stipulates that the total energy consumed is the product of average current drawn by the hardware, voltage provided to the hardware and the average time taken by the algorithm to execute. Arduino uses 19.9 mA of average current when supplied with

5V voltage. We have tested the average time for different algorithms to measure the energy consumed by them. Table 6.1 shows the results.

Table 6.1. Energy consumption of different protocols

Algorithm	**Average time to execute**	**Energy consumed**
AES 128	2.5 ms	248.75 uJ
SHA 512	1 ms	99.5 uJ
SHA 512 with HMAC	1.5 ms	149.25 uJ
ECC	1105 ms	109.95 mJ
AES 256	4 ms	398 uJ

Table 6.2. Energy consumption of different authentication schemes

Algorithm	**Energy consumed**
secure vault	646.75 uJ
ECC	109.95 mJ
single rotating password	746.25 uJ

Our algorithm requires one AES encryption and one AES decryption at the IoT device side. For changing the secure vault value, one HMAC operation is required. Thus, the total energy consumed by our protocol is 646.75 uJ. If we compare our algorithm with ECC based authentication, the energy consumption is fairly low. The simplest version of authentication with one password and AES-128 encryption requires 2 AES operations and it consumes 497.5 uJ of energy. Single rotating password-based scheme requires 3 AES operations, first two AES operation for authentication and the last one for the exchanging new password using previous password as encryption key. The total energy consumed by single rotating passwords is 746.25 uJ. Table 6.2 shows the comparision between different authentication algorithms.

6.4.2 Memory Analysis

We compared our algorithm with single key based algorithm for this analysis. We also illustrate memory requirement for different values of m and n. For the memory analysis, we divided the memory usage in two sub categories: program memory and data memory. Program memory is the memory required to store the program for the authentication algorithm. As single password and our algorithm both use the same encryption algorithm and same 3-way communication method, hence they require same amount of memory for both the cases. Data memory is the memory used to store the actual keys. Table 6.3 compares the data memory usage for different values of m and n.

Table 6.3. Data memory required for different authentication schemes

Algorithm	**n**	**m**	**Data memory**
single key	1	1	128 bits
secure vault	4	128	512 bits
secure vault	4	256	1024 bits
secure vault	8	128	1024 bits

Table 6.4. Data memory required for different authentication schemes

attack	**our algorithm**	**single non-rotating password**	**single rotating password**
Man in the middle	Yes	Yes	Yes
Dictionary attack	Yes	Yes	No
DoS attack	Yes	Yes	Yes
Side-channel attack	Yes	No	No

Table 6.5. Password prediction complexity for different values of m and n

n	m	Data memory	Password prediction complexity
1	128	128	1
2	128	256	2^{128}
2	256	512	2^{256}
4	128	512	$3 * 2^{128}$
4	256	1024	$3 * 2^{256}$
8	128	1024	$7 * 2^{128}$
8	256	2048	$7 * 2^{256}$

6.4.3 Security Analysis

Security of our algorithm is analyzed based on following factors: safety against known security attacks and password prediction complexity. Table 6.4 shows the comparison of our algorithm with other algorithm for known attacks. Yes in the table indicates algorithm is secure for the given attack.

In this security analysis, we assume that an attacker can retrieve the password being used during the authentication mechanism using a side-channel attack. We compare password prediction complexity for different values of n and m, where n = number of keys in the vault and m = size of each key. Single rotating and non-rotating based password methods can be considered as n = 1. For the authentication system with n = 1, the attacker can get the actual password used for the authentication using a side channel attack. For n = 2, the attacker has to perform brute force 2^m hash operations to retrieve both the passwords. In general, for a secure vault with m bits long n keys, requires (n - 1) * 2^m brute force hash operations to predict the whole secure vault. Table 6.5 shows the comparison of data memory required and password prediction complexity for different values of m and n.

Table 6.5 shows that higher the value of n better the security. This better security comes with the cost of high power consumption and more memory. From Table 6.1, we can observe

that the energy required for 256 bits AES is around 1.5 times more than the energy required for the 128 bits.

In the previous section, we discussed that after every session, it is recommended to change the value of secure vault. This is the highest rate a secure vault value can be changed. Alternatively, the secure vault contents can be changed less often based on the values of m and p,where m is the number of keys in the secure vault and p is the number of keys involved in the authentication process. If an attacker is using side-channel attack to retrieve the key (generated from sub-keys of secure vault) used in the authentication process, the same set of sub-keys cannot be used during the authentication phase without changing the secure vault value. For an authentication algorithm having m keys and p keys being used in authentication, $\binom{m}{p}$ distinct sessions are supported without changing the secure vault value.

The work discussed in this chapter is published in IEEE TrustCom 2018 conference [56].

CHAPTER 7

LOGIN PUZZLE

In the last decade, the rapid growth of IoT products in the market have connected billions of small-scale devices to the internet. To survive in such a fast-growing market, developers often release products with security vulnerabilities. The attackers use such vulnerabilities to perform large-scale DDoS attacks and steal personal information of the user. Mirai is a well-known DDoS attack, which represents the first type of attack using vulnerable IoT devices. A casino at Las Vegas, which lost some of its critical information due to a cyber-attack enabled by a vulnerable internet enabled fish tank1, is an example of another attack.

IoT devices using either default or easy to guess credentials allow the attackers to access these devices remotely with root permissions. An attacker can log in using open Telnet or SSH ports with a pre-defined set of credentials. Most of the time, around 100 credentials are sufficient to capture a massive number of IoT devices and only a few seconds are needed to brute-force 100 credentials. Thus, a self-replicating attacker script can brute-force a few hundred thousand devices in less than a week. The ability to brute force devices without any restriction enables attackers to capture a large number of IoT botnets.

A potential solution for this problem can be to restrict a single IP from trying a large number of login attempts. This solution is vulnerable to IP address hopping: an attacker can change the IP address after every unsuccessful login. Another solution is to implement a blocking mechanism, which allows a small number of login attempts before it blocks further login attempts. This method certainly reduces the number of devices being captured, but the attacker script can get an instantaneous access till it reaches the blocking number. A selective approach with different login credentials for different kind of devices can capture a large number of devices. A timeout-based method is also useful in preventing multiple abusive login attempts. The device enters into a timeout mode after every unsuccessful

attempt and every unsuccessful attempt increases the timeout period. The downside of the timeout method is, the attacker script can capture other devices in parallel.

In this chapter, we present a method called login puzzle to prevent the unrestricted login attempts performed by a malicious script. Login puzzle is a collection of mini puzzles, which requires a fixed amount of time and computation power to solve. When a malicious script tries to brute-force an IoT device, it needs to allocate a fixed amount of time and resource to solve the login puzzle, and every unsuccessful attempt increases the complexity of the login puzzle, hence increasing the time and computation power required to solve the next login puzzle. At a certain point, the login puzzle is hard enough that it is not practical for the script to solve. This method not only limits the number of attempts but also slows down the rate of attempts. It is a combination of both blocking and timeout method and provides better performance than these individual methods.

Another approach is to send a single puzzle to the login entity instead of multiple mini puzzles, it leads to a problem in changing the complexity of the puzzle. If we change the complexity every time a wrong attempt happens, an adversary can request multiple parallel sessions with a same complexity. On the other side, a malicious script can perform multiple attempts just to increase the complexity to block a benign user. Use of login puzzle solves these problems.

Login entity must solve all the mini puzzles of the login puzzle to get the login access to the IoT device. All of these puzzles are sent sequentially; login entity will get the next mini puzzle, once it solves the previous one. In case of parallel requests, once a request solves all the puzzle and attempts an unsuccessful login attempt, number of mini puzzles for the other parallel requests will increase, consequently increasing the complexity of the subsequent login attempts. If an adversary just requests multiple sessions without actually solving mini puzzles, the complexity of the puzzle wont increase, hence benign user wont be block by any malicious script.

We have calculated the improvement achieved by using login puzzle. We have measured this scheme with Mirai and computed the amount of additional time and resource required to capture the IoT devices. The main contribution of our work includes:

- A practical mechanism to extend the capture time of an IoT device as botnet
- Introduction to the login puzzle and provide calculations for its performance efficiency
- A novel complexity changing mechanism, which ensures constant resource allocation from the client side and provides minimum penalty to benign users

7.1 DDoS Attack Overview

A denial of service is characterized as attempt to prevent the use of a network service to the legitimate users. In a DDoS, an attacker uses multiple devices to achieve denial of service. As more IoT devices are being manufactured with limited security considerations, new methods are being introduced to use IoT devices as bots for DDoS attacks. A study by Dragan Perakovi et al. shows that the number of DDoS attacks has increased with the growth in numbers of IoT devices [46].

A DDoS attack using IoT botnets typically consists of three phases: host scanning, device acquisition and denial of service attack. In the host scanning phase, the attacker scans for the IoT devices with open Telnet or SSH ports. After discovering a vulnerable device, the attacker script tries to log into the device using a set of pre-defined credentials.

If the attacker gets the access to the IoT device using one of the pre-defined credentials, the attacker executes the second phase. In the device acquisition phase, a selfreplicating script is injected inside the IoT device by the attacker. The self-replicating script scans through the network to infect more IoT device using the same host scanning mechanism and injects the same script into the newly infected device. Meanwhile, this script continuously

communicates with the attacking server and waits for the commands to perform a DDoS attack. After acquiring enough IoT devices, the attacking server commands all the infected IoT devices to attack a specific service. A large volume of internet traffic is generated for the victim with the potential to disrupt the service.

The four main components for a DDoS attack using IoT devices are an attacker, a malware server, a command and control (CnC) server and IoT botnets. The attacker initiates the host scanning phase and acquires an ample amount of IoT devices. Those discovered IoT devices download the self-replicating script from the malware server. In some cases, the malware server is also known as report server. The script registers the IoT device to the CnC server and waits for the command from the CnC server. Finally, after acquiring adequate bots, the attacker commands the CnC server to send a request to botnets to attack the victim service.

7.2 Our Methodology

We present a mechanism to provide security against the DDoS attack at the device scan phase. The malware could acquire a large number of IoT devices because there is no restriction on trying multiple guessed credentials using a brute force attack. Our method called Login puzzle', prevents an attacker from brute-forcing credentials. Login puzzle provides an extra layer of a challenge-response mechanism by generating NP-hard puzzles and asks the login entity to solve it before being logged in. The difficulty of question gets harder with the number of false attempts, hence stopping the intruder from using the brute-force method to capture an IoT device.

The login attempts to an IoT device can be divided into three main categories: log in using user interface having username-password as security credentials, log in using script having username-password as security credentials and log in using script using the digital certificate as security credentials. The third category is not vulnerable to brute force attack as it is computationally impossible to create a brute-force attack on a digital certificate,

hence no additional security required for this category. For the first and second categories, an additional mechanism is required to prevent the unrestricted brute-force login attempt. In our methodology, we use regular captcha and login puzzle for the first and second category, respectively.

Both first and second category use: username-password as security credential: To differentiate them, a flag bit indicating the type is sent along with the login request. In this paper, we assume that a computer program is not capable of solving the regular captcha used for login. Hence, if a script tries to fake itself as a human to avoid the login puzzle, it won't be able to login to the system.

7.2.1 Login Puzzle

A login puzzle is a collection of mini puzzles with a varying complexity, which require a fixed amount of time and resource to solve depending on the complexity. A login puzzle is defined as P (n), where n defines the complexity of the puzzle. Complexity of a login puzzle is addition of all of its mini puzzles. Mini puzzle is expressed as M (x, n), where x is the set of puzzle parameters and n is the complexity of the mini puzzle. A login puzzle can have mini puzzles with different complexities. The time required to solve the login puzzle P and mini puzzle M at complexity n is t_p (n) and t_m (n) respectively.

The login puzzle has the following properties.

- Based on the complexity n, it requires a fixed amount of time and computation power to solve.
- For a puzzle function P, if $n_1 > n_2$ then $t_p(n_1) > t_p(n_2)$.
- For every device with a finite computational capacity C, $\exists$n at which computational capacity C requires unreasonably more time and resource to solve the login puzzle. This value of complexity is called critical complexity for the computational capacity C. The time require to solve it called critical time t_c (n).

7.2.2 Complexity Increment of the Login Puzzle

After every unsuccessful attempt, the complexity of the login puzzle is increased. There are two ways the complexity can be increased: linear increment and exponential increment. In the linear increment, the complexity of login puzzle is increased exponentially. This can be achieved by either increasing the number of mini puzzles, without changing the complexity of mini puzzles or increasing the complexity of only one mini puzzle. In exponential increment, the complexity of each mini puzzle is incremented exponentially, consequently increasing the overall complexity exponentially. In general, after every unsuccessful attempt complexity of the puzzle is incremented exponentially and in case client stops attempting the puzzle, the complexity is increased linearly.

7.2.3 Login Mechanism

At each login attempt by a remote device, a login puzzle is sent to the device. The initial complexity of the login puzzle is 1, and each unsuccessful login attempt to the IoT device exponentially increases the complexity. As per the third property of the login puzzle, at some point, the login attempt requires immense resources, which cannot be handled by the remote login-device. The complexity increment mechanism is as follow:

- Normal attempt: When a remote login entity solves all the puzzles, it is considered as a normal attempt. Once the remote device provides solutions for all the puzzles, it has access to the login and if it performs an unsuccessful login attempt, the complexity of login puzzle is increased exponentially. On the other side, for successful login attempt the complexity will reset to 1.

- Midway give up attempt: When a login entity stops solving mini puzzles in the middle and doesnt arrive at login attempt, the login puzzle complexity is increased linearly. This can be performed by either increasing number of mini puzzles or increasing complexity of single mini puzzle.

- Parallel login attempts: This situation arises when a login entity requests for the login access during an existing login entity solving login puzzle. At the initial stage the new login entity will have same complexity as the previous one. In case the previous one does the unsuccessful login attempt, the complexity of login puzzle is doubled.

7.3 Resource and Timing Analysis

The resources and time required to solve a login puzzle is addition of time and resource required to solve all mini puzzles. Every device needs to allocate a fixed computational capacity to solve a login puzzle. The required computational capacity varies based on the type and complexity of the login puzzle and, as per the 3^{rd} property of the login puzzle, after m tries, the login puzzle reaches to the critical complexity and it is not feasible for a device to solve the puzzle. Thus, a device with a computational capacity C is limited to only m tries. All these m tries are not instantaneous; for every try, the device needs to solve a login puzzle, which requires a fixed amount of time to solve. In this section, we will measure the effectiveness of login puzzle on a botnet capture script, which uses x number of credentials to capture an IoT device.

Consider the time required to try one credential without login puzzle be t_d. This time includes propagation and transmission delay caused by the network and processing delay due to login procedure; it is few milliseconds in duration. If we consider a uniform distribution of the credentials, it requires on an average $x/2$ attempts for an attacker script to capture an IoT device as a botnet if the devices credential is from one of the x credentials. The device needs to perform x failed attempts if the devices credential is not from the list. Assuming that the ratio of number of devices whose credentials are from x credentials to the total number of devices scanned is p, the average time required to capture one device is:

$$t_{avg} = t_d * \frac{x}{2} * p + t_d * (1 - x) * p$$

This can be further simplified as:

$$t_{avg} = x * t_d * \left(1 - \frac{p}{2}\right) \tag{7.1}$$

Here,

$$p = \frac{number\ of\ devices\ whose\ credentials\ are\ from\ x\ credentials}{total\ number\ of\ devices\ scanned}$$

If we consider an exponentially increasing login puzzle, where the complexity of $t_p(n)$ increases a times with the increment of complexity of login puzzle by 1. Considering the time required to solve the login puzzle at complexity 1 is t_c. As per the 3^{rd} property of login puzzle, after m attempts the problem becomes incomputable for the device. The total time required to solve login puzzle for first m attempts is,

$$t_m = \sum_{i=0}^{m-1} t_c * a_i \tag{7.2}$$

If the devices credential is from one of the pre-defined credentials, it takes x/2 at-tempts to capture the device, otherwise the attacker script tries x credentials. Considering there are k mini puzzles in one login puzzle, average time required by a device to capture one device after using login puzzle is,

$$t_{avg}(login\_puzzle) = p * \left(t_{\frac{x}{2}} + \frac{x * k * t_d}{2}\right) + (1 - p) * (t_x + x * k * t_d) \tag{7.3}$$

Where,

$$t_{x/2} = \sum_{i=0}^{\frac{x}{2}-1} t_c * a^i \quad and \quad t_x = \sum_{i=0}^{x-1} t_c * a^i$$

For simplicity, if we consider the time complexity of login puzzle increases with a factor of 2. The values of t_x and $t_{\frac{x}{2}}$ is as follow.

$$t_{x/2} = \sum_{i=0}^{\frac{x}{2}-1} t_c * 2^i \quad and \quad t_x = \sum_{i=0}^{x-1} t_c * 2^i$$

If we assume the factor of tc to td is f. The average required time to capture a device using login puzzle is,

$$t_{avg}(login\_puzzle) = p * t_c * \left(2^{\frac{x}{2}} - 1 + \frac{x * k}{2 * f}\right) + (1 - p) * t_c * \left(2^x - 1 + \frac{x * k}{f}\right) \quad (7.4)$$

Case 1:

For the value of $\frac{x}{2} < m \leq x$, the t_x term in equation 7.3 becomes t_m, because after m attempts the login puzzle is incomputable.

$$t_{avg}(login\_puzzle) = p * t_c * \left(2^{\frac{x}{2}} - 1 + \frac{x * k}{2 * f}\right) + (1 - p) * t_c * \left(2^m - 1 + \frac{m * k}{f}\right) \quad (7.5)$$

$$t_m = \sum_{i=0}^{m-1} t_c * a_i$$

Case 2:

If $m \leq \frac{x}{2}$, the average time required to capture a device using login puzzle is,

$$t_{avg}(login\_puzzle) = p * t_c * \left(2^m - 1 + \frac{m * k}{f}\right) + (1 - p) * t_c * \left(2^m - 1 + \frac{m * k}{f}\right) \quad (7.6)$$

Equation 7.6 can be furher simplfied to followng equation:

$$t_{avg}(login\_puzzle) = t_c * \left(2^m - 1 + \frac{m * k}{f}\right) \quad (7.7)$$

The probability of an IoT device getting captured for the above case is,

$$\frac{p * x}{m}$$

Table 7.1. Time required to solve login puzzle for different type of devices

Environment	**n**	t_n
Raspberry Pi	16	10 s
	24	200 s
	32	Incomputable
Macbook Pro	16	Less than a second
	24	20 s
	32	85 mins
	40	Incomputable
AWS Server	24	Less than a second
	32	1 min
	40	250 mins
	48	Incomputable

7.4 Evaluation and Performance Measure

We evaluated our approach on the different environments with different login puzzle conditions for Mirai DDoS attack. We have used three different environments with completely different capabilities to measure the effectiveness of our approach. The first environment is a small scale IoT device and we have used Raspberry Pi 3. The second environment is a MacBook Pro laptop with quad-core Intel i7 processor and 16GB of RAM. The final environment is a cloud server hosted on AWS cloud ser-vice. The AWS instance contains 36 cores with 60GB RAM. The first environment represents the normal spread of botnets, where infected IoT devices scan other IoT devices and solve the login puzzles themselves. The second and third environments provide the insight for the case, when IoT devices dont solve the login puzzles but send it over to the server to solve them. The primary difference between the second and third environment is the processing power of the server. The login puzzle we are using for the evaluation is as follow:

D (x, n): find a number m, for which SHA512(x || m) has all last n bits are 0

Table 7.1 shows the typical time required for all three environments to solve the login puzzle for complexity n.

We use the following method to simplify the performance evaluation of the login puzzle. First, we calculate the average time required to capture a single device with and without using login puzzle. Using these values, we measure the performance enhancement achieved by login puzzle. Finally, we multiply the performance enhancement ratio and the actual time taken by Mirai to spread over the network and compute the time required to spread Mirai when all the IoT devices are using login puzzle. We also made another assumption that the number of mini puzzles, k = 16.

Number of credentials used in Mirai was 62. (value of x in the equation 7.1 is 62). Thus, the average time to capture one device is:

$$62 * t_d * \left(1 - \frac{p}{2}\right) \tag{7.8}$$

We have tested the values of t_c and t_d for the raspberry pi, and the values are 10 μs and 100 ms respectively.

$$factor\ f = \frac{t_c}{t_d} = 10^{-4}.$$

To calculate value of m, we assume that the attacker script stops solving login puzzle, once the time taken for solving captcha reaches 100s.

$$m = log_2\left(\frac{100}{t_c}\right) \sim 24$$

As value of the x/2 is greater than m, the average time required to capture one IoT device is calculated using equation 7.7. The average time required to capture one device is $\sim$ 200 s. Due to limitation of only m tries can be performed, not all devices with the default credentials can be captured.

The probability of device being captured is $\frac{m}{x} = 0.39$.

Time required to capture one device using login puzzle with probability of device being captured is 1,

$$t_{(p=1)} = \frac{time\ required\ to\ capture\ one\ device\ using\ login\ puzzle}{probability\ of\ device\ being\ captured} = 512.82s$$

In conclusion, use of login puzzle increases the time required to capture a device by a factor of,

$$\frac{t_{(p=1)}}{time\ required\ to\ capture\ single\ device\ without\ login\ puzzle} = 82.72$$

Here, the time required to capture single device without login puzzle is calculated from equation 7.8 by setting p = 0, which represents the maximum value for the equation. This implies that the total time required to cover the spread of 64,500 devices with login puzzle = 20 hours * 82.72 $\sim$ 69 days.

Similar calculation can be performed on 2^{nd} and 3^{rd} type of environment and we can find time required to capture 64,500 devices if all the devices were using login puzzle and the login puzzles are solved on cloud with capabilities mentioned in Table 7.1. Table 7.2 mentions these times.

Table 7.2. Time required to capture 64,500 devices with login puzzle using different environment

Environment	**Time required**
Raspberry pi	59 days
Macbook pro	42 days
AWS Server	26 days

7.4.1 Comparison with Blocking Method

In the blocking method, after a certain amount of attempts the IoT device blocks further login attempts until the user approves those attempts. At first, it looks like a blocking method with a small number of login attempts performs better than the login puzzle. In

fact, based on our calculations, our method performs 70 times better than the blocking mechanism.

Lets assume that the IoT device blocks login attempts after b attempts. The average time required to capture one device is,

$$t_d * \frac{b}{2} * \frac{b*p}{x} + t_d * b * \left(1 - \frac{b*p}{x}\right)$$

The first term in the equation represents that one of the randomly guessed credentials in first b tries is the credential of the device. On an average it requires b/2 tries to achieve it with the probability b/x. This probability is finally multiplied with probability p, which represents the number of devices with the credentials used by the malware. The second term represents the scenario when the malware is not able to guess the correct credentials in first b login attempts. The above equation can be further simply to 300 7.25p milli-seconds for the value of b = 3 and x = 62. The ratio of number of devices captured to total number of devices with given credential is,

$$\frac{b}{x} = 0.05$$

The value of p varies from 0 to 1 does not have a big impact on time required to capture the device. Thus, we can assume that the time required to capture one device is 300 ms. So, the average time to capture one device is = 300/0.05 milli-seconds = 6 seconds. The average time to capture one device using login puzzle is = 170/0.39 seconds = 436.34 seconds. This implies that the login puzzle method is roughly 70 times better than blocking method.

7.5 Comparison with Existing Client Puzzles

In this section, we will compare our approach with existing client puzzles and showcase its suitability to prevent the IoT device capture on a large scale. We compare our scheme with

simple client puzzle and adaptive client puzzle. A simple client puzzle is the most basic form of client puzzle and adaptive puzzle is a type of puzzle which changes its complexity based on the situation. We used three parameters to compare these client puzzles: login delay, resource assurance, benign penalty.

Table 7.3. Login puzzle comparison

	Client Puzzle	**Adaptive Puzzle**	**Login Puzzle**
Login delay	Very low	High	High
Resource allocation	No	No	Yes
Benign penalty	Low	Very high	Minimal

Table 7.3 shows a comparison between these three client puzzles. As normal client puzzle doesnt change its complexity over the time, it is not providing any significant delay in login process. At the same time, it is not creating any high waiting time for the benign user, so benign penalty is minimum in this case. For the case of adaptive puzzle, the login delay can go really high, as the complexity go higher. As discussed in the previous sections, a malicious user can increase the complexity for the benign user by sending multiple fake requests. Just like adaptive puzzle, login puzzle has high login delay and increases with every wrong login attempt. In login puzzle, as discussed in the previous sections, puzzle ensures the resource assurance and has different policies to increase complexity for benign and malicious request, hence benign penalty is minimal in this case.

7.6 Discussion and Future Work

we demonstrated how adding an extra layer of login puzzle during an authentication phase can reduce the number of IoT devices being captured as botnets. A login puzzle is an NP-hard problem, whose complexity increases at every unsuccessful login attempt. In this chapter, we used searching a bitstream of zeros at the end of the inverse of a one-way hash

function as the login puzzle problem. However, this method does not prevent capturing of every IoT device, it reduces the number of devices being captured and increases the time and resource required to capture those devices. Based on the calculations in section 7.4, the additional time required to capture IoT devices is enough to detect the spread of the malware through the network traffic and prevent it from acquiring more devices.

Unlike normal adaptive puzzle, login puzzle doesnt have problem of benign penalty. Login puzzle provides minimal penalties to benign users. These penalties can be further reduced, if someway a benign request is differentiated from malicious request. As a future work to reduce the benign penalties, a method to provide the login puzzle only to the suspicious scripts can be implemented. There are researches explain multiple ways to differentiate a bot attack from a regular user login [65] [21] [6]. This paper assumes the distribution of the passwords is uniform, but in a real-world scenario, it is not the case. A more detailed study with a non-uniform password distribution provides a more accurate performance analysis of the login puzzle.

The work discussed in this chapter is published in ICIoT 2019 conference [58].

CHAPTER 8

IMPLEMENTATION

We have implemented a user-customizable IoT system to provide an insight to our work. Our system supports both IoT architectures, standalone IoT architecture and gateway based architecture, which are defined in Chapter 2. Fig. 2.2 and Fig. 2.1 show a representation of the systems. The user interface and the server are same in both the architectures. In this chapter, we discuss the implementation details of the gateway based system and at the end of the chapter, we explain the changes needed to support the standalone system. We have implemented a rule conflict and incompleteness detection system. For the the standalone systems, the rule detection system runs on the server and for the gateway based systems, it runs on the IoT gateway device. For the mutual authentication, a separate implementation has been built using an esp8266 [15] based system.

8.1 User Customizable IoT System

our user customizable IoT system contains four basic elements: Server, IoT device gateway, self-aware device and user interface.

8.1.1 IoT Server

We have hosted our server on AWS EC2 instance. The server is divided into three main components: database, web api calls and business logic. The database contains all the data related to IoT devices, self-aware devices and users. We use mysql a structured database as the database system. We use nodejs as the server-side coding language. Nodejs is flexible, scalable and event-driven server-side library designed on the javascript. We use express package of nodejs to build web api calls. We use sequelize package to communicate with the database. The use of sequelize provides a better security and flexibility. It secures the web api calls from database attacks like sql injection.

8.1.2 IoT Device Gateway

We have used a Raspberry Pi as the IoT device. We have created four threads to perform the following tasks independently:

- Detect any new USB device connection or removal of a self-aware device.
- Communicate with connected self-aware sensors/actuators via USB. When there are more than one sensors/actuators, the IoT device communicates with them one by one.
- Get push notification from the server via MQTT.
- Run the rule engine and rule conflict system.

When any newly connected device is detected by the first thread, the thread checks for the type of new device connected to it and if the new device is a self-aware device, it configures it and adds it to the list of connected devices.

According to the constraints of self-aware sensor/actuator, the second thread will communicate with the sensor and upload data through the services provided by the server.

If the user wants to manually turn on/off the actuator, the user can send a request to the server via the user interface. The server sends the same request to the appropriate IoT device using MQTT notification. The third thread running on IoT device receives this request and sends the appropriate command to turn on/off the actuator. If all the self-aware sensors and self-aware actuators related to a rule are attached to the same IoT device, the last thread of IoT device code executes the rule and sends appropriate actions to the self-aware actuator.

8.1.3 Self-aware Device

For proof of concept, an Arduino board has been used to build the self-aware sensor (or actuator). [A custom hardware will make the device minimal and aesthetically appealing,

but will be time-consuming to build.] Temperature sensor ds1820 is connected to it. The sensor ds1820 works on 1 wire protocol. When the Arduino is connected to the Raspberry Pi it sends all details such as unique id, sensors connected to it, the range of the sensor, sensors other constraints, etc. All these details are stored in nonvolatile memory of Arduino. After getting all these values, the Raspberry Pi will periodically request values from Arduino using AT commands. The Arduino will read the value from ds1820 via 1 wire protocol, and send the value to the pi. Before sending data to the pi, the Arduino performs preliminary check on the data to ensure that the data being provided to the Raspberry Pi is valid.

8.1.4 User Interface

We have created a simple web interface. It consists of the following web pages.

- **Summary Page:** It displays basic details of the user. It also displays all the IoT devices owned by the user. The user can import a new IoT device and delete an existing IoT device from this page.
- **Rule Page:** User can create a new rule from this page. The user can select the IoT device to program, see what sensors and actuators are connected to it, and then from a set of pull down menus, select the desired behavior. This page is responsible for taking all the data from the user to create the needed tables which will be used by our backend to automatically create code to implement the rule.
- **Data Display Page:** All the sensor values are displayed on this page.

We have also deployed one system at a local organic farm named Profound Microfarms and the system operated well for 9 months. The system consists of two self-aware sensors measuring air and water temperature respectively. The user can customize text message alert or email alert on this system. The farmer at Profound Microfarms have customized

alerts according to their requirements. For example, during winter they set an alert for the air temperature value greater than 70 °F and during summer they set an alert for the air temperature value greater than 85 °F.

8.2 Rule Conflict

We have setup multiple IoT systems working in different environments and constraints to check the validity and scope of our algorithm. We next illustrate our approach on a smart agriculture system.

8.2.1 System Architecture

We use an IoT system with the following components: an IoT server, an IoT device and a user interface. The IoT server is the main component of the IoT architecture and is responsible for storing IoT device information, sensor data and rules. It also creates a communication link between the IoT device and the user interface. The IoT device stores, maintains and executes the rules for the IoT system. It also checks for the rule conflict and incompleteness.

Fig. 8.1 shows a simple rule structure. The rule contains a trigger rule part and an anti rule part. If the trigger is based on sensor values only, its relation operator is '.'. If the trigger is a combination of more than one trigger, the relation operator is the relation between those triggers.

8.2.2 Complexity Analysis

The IoT system checks for the rule conflicts and incompleteness when, a) a new rule is added, b) an existing rule is deleted or c) an existing rule is edited. For the rule conflict, all the clauses of the newly added rule are compared with all existing clauses. If there are n existing clauses and the newly added rule has m clauses, the number of overlap checks are n * m. Each overlap check requires us to check overlap for individual sensor value and if there are

```
[{
  "ruleName": "testRule",
  "trigger-conditions":[
    {
      "symbol": "A",
      "relation_op": ".",
      "sensor": "SoilMoisture",
      "value": 30,
      "op": "<="
    }
  ],
  "actions":[
    {
      "actuator": "Sprinkler",
      "param": "On"
    }
  ]
  "anti-triger-conditions":[
    {
      "symbol": "B",
      "relation_op": ".",
      "sensor": "SoilMoisture",
      "value": 40,
      "op": ">="
    }
  ],
  "anti-actions":[
    {
      "actuator": "Sprinkler",
      "param": "Off"
    }
  ]
}]
```

Figure 8.1. This figure represents JSON structure of a simple rule. The rule is "if soil moisture is less than 30 % turn on the sprinkler and keep it on till the soil moisture reaches 40 %".

k sensors in the system, the worst-case time required to check overlap for a single clause is O(k). Time required to detect the rule conflict is O(n * m * k).

Table 8.1. Smart Agriculture Rules

rule name	**trigger conditions**	**action (on sprinkler)**	**anti-trigger conditions**	**anti-action (on sprinkler)**
rule 1	soil moisture $< 20\%$	turn on	soil moisture $> 50\%$	turn off
rule 2	soil moisture $< 30\% \wedge$ time is 7 AM	turn on	soil moisture $> 50\%$	turn off
rule 3	rain sensor = raining	turn off	NA	NA
rule 4	rainTomorrow $\wedge$ soil moisture $\geq 25\%$	turn off	NA	NA
rule 5	soil moisture < 10 %	turn on	soil moisture $> 50\%$	turn off

Table 8.2. Smart Agriculture Rule Conflict Analysis

conflicting rules	**conflict type**	**conflicting range**	**conflict avoid technique**
rule 1 & rule 5	shadow	soil moisture < 10 %	remove rule 5
rule 2 & rule 4	execution	rainTomorrow $\wedge$ time is 7 AM $\wedge$ soil moisture $\in$ [25,30)	adjust ranges of trigger conditions
rule 1 & rule 3	independent	it is raining when soil moisture < 20 %	assign priorities
rule 5 & rule 3	independent	it is raining when soil moisture < 10 %	assign priorities
rule 2 & rule 3	independent	it is raining when soil moisture < 30 % $\wedge$ time is 7 AM	assign priorities

The incompleteness is detected in a similar way. It takes O(n) time to check the polygon cover for a rectangle based problem [19], where n is the dimension of the polygon. Incompleteness check requires n * m iterations, hence the complexity of detecting incompleteness is O(n * m * k).

8.2.3 Smart Agriculture System

We have implemented a rule-based system for smart agriculture system with the following sensors and actuators:

- soil moisture sensor

- temperature sensor
- time - virtual sensor
- location - virtual sensor
- rain sensor
- weather feed - virtual sensor
- sprinkler controller

Virtual sensors are that sensors provide a value without a physical device being present.

Table 8.1 describes a set of rules for smart agriculture. Table 8.2 describes the conflicts for the rules mentioned in Table 8.1.It explains the rule type, conflicting range and potential solution to avoid the rule conflict.

Rule 1 and rule 5 are in shadow conflict; the range of rule 5 is a subset of rule 1 and they both are performing the same action. Removal of rule 5 will not affect the overall system behavior. Rule 2 and rule 4 have execution conflict when the time is 7 AM, tomorrow's weather feed is predicting rain and soil moisture is between 25 and 30. It can be avoided by either changing the trigger value for soil moisture either in rule 4 to $\geq 30\%$ or in rule 2 to $\leq 25\%$.

8.3 Secure Vault

We have implemented our protocol to test the feasibility of our authentication mechanism on constrained IoT devices. We have used Arduino as the IoT device. Arduino consists of an ATMEGA 328 processor, 32 MB of Flash memory, 2KB of SRAM and 16 MHz of clock frequency. An Arduino is connected with esp8266 [15] Wi-Fi module to connect to the internet. An Intel i7 based machine is used as the IoT server. The IoT server and the IoT

device communicate using TCP socket connection. The IoT device contains a temperature sensor ds1820 [42], which provides output in digital format to the IoT device. The IoT device sends this temperature data to the server using the following format.

$< device\_id > - < sensor\_id >:< sensor\_value >$

Here, we are assuming *device_id* and *sensor_id* are 64bit and 128bit in length respectively and these id are transferred in hex format. The sensor value is a string of digital output of temperature value with 2 decimal precision. For the securevault the number of keys (n) is 8 and the size of keys (m) is 128 bits. We implemented this authentication mechanism using 128-bit AES encryption.

The IoT device sends a connection request to the server with request id and device id and the server responds back with a session id to maintain the session between the IoT device and the server. After that, they both perform the 3-way authentication as explained in the previous section.

CHAPTER 9

CONCLUSION

In this dissertation, we present mechanisms to create a robust user-customizable IoT system. First, we have designed and implemented a user configurable IoT architecture using which a user can add (or remove) one or more self-aware sensor/actuator to (or from) the IoT system without knowing any hardware knowledge. The user can also define rules that will govern the execution of the IoT system, without having to write any software. This architecture enables a novice user to build a highly customized IoT system. We have built several proof of concept prototypes to validate the idea. Later, We present a mechanism to detect conflicting and incomplete rules in IoT systems. Our approach is flexible and can be used in any IoT system. We define three ways a rule can conflict with another. In addition to that, we introduce rule completeness, which checks for the sensor ranges for which an action is not defined. Lastly, we design two mechanisms to provide better security for the IoT device. The first mechanism, 'Secure Vault', secures the communication between an IoT device and an IoT server from the side-channel attacks. The second mechanism, 'Login Puzzle', delays the spread of the DDoS malwares. The login puzzle is a variant of client puzzle with the following properties: continuous client resource assurance and minimal benign penalty.

As a future work of this dissertation, a rule conflict algorithm, which can detect rule conflict for more complex functions. As a extension of our work, a rule conflict algorithm for IoT systems that have a complex rule structure like the action is triggered based on not only current sensor value but also past sensor values or the action parameter is in a linear relationship with the sensor value. For these cases, rules can be defined as a program using an event-based programming language. For the 'Login Puzzle', we assume the the distribution of the passwords is uniform, but in a real-world scenario, it is not the case. A more detailed study with a non-uniform password distribution provides a more accurate performance analysis of the login puzzle.

REFERENCES

[1] Abliz, M. and T. F. Znati (2015). Defeating ddos using productive puzzles. In *2015 International Conference on Information Systems Security and Privacy (ICISSP)*, pp. 114–123. IEEE.

[2] Alam, S., M. M. Chowdhury, and J. Noll (2010). Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications*, pp. 1–6. IEEE.

[3] Amazon (2019). IoT Applications & Solutions — What is the Internet of Things (IoT)? — AWS. `https://aws.amazon.com/iot/`. (Accessed on 07/03/2019).

[4] AT&T (2019). AT&T M2X: Build solutions for the Internet of Things. `https://m2x.att.com/`. (Accessed on 07/03/2019).

[5] Aura, T., P. Nikander, and J. Leiwo (2000). Dos-resistant authentication with client puzzles. In *International workshop on security protocols*, pp. 170–177. Springer.

[6] Binkley, J. R. and S. Singh (2006). An algorithm for anomaly-based botnet detection. *SRUTI 6*, 7–7.

[7] Bormann, C., M. Ersue, and A. Keranen (2014). Terminology for constrained-node networks. *Internet Engineering Task Force (IETF): Fremont, CA, USA*, 2070–1721.

[8] Butun, I., M. Erol-Kantarci, B. Kantarci, and H. Song (2016). Cloud-centric multi-level authentication as a service for secure public safety device networks. *IEEE Communications Magazine 54*(4), 47–53.

[9] Carreira, P., S. Resendes, and A. C. Santos (2014). Towards automatic conflict detection in home and building automation systems. *Pervasive and Mobile Computing 12*, 37–57.

[10] Chen, L., P. Morrissey, N. P. Smart, and B. Warinschi (2009). Security notions and generic constructions for client puzzles. In *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 505–523. Springer.

[11] Danger, J.-L., S. Guilley, P. Hoogvorst, C. Murdica, and D. Naccache (2013). A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *Journal of Cryptographic Engineering 3*(4), 241–265.

[12] Datta, S. K., C. Bonnet, and N. Nikaein (2014). An iot gateway centric architecture to provide novel m2m services. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 514–519. IEEE.

[13] Donzellini, G., L. Oneto, D. Ponta, and D. Anguita (2019). Boolean algebra and combinational logic. In *Introduction to Digital Systems Design*, pp. 1–31. Springer.

[14] DAniello, G., M. Gaeta, V. Loia, and F. Orciuoli (2015). An ami-based software architecture enabling evolutionary computation in blended commerce: the shopping plan application. *Mobile Information Systems 2015*.

[15] ESP8266 (2019). Everything esp8266. `https://www.esp8266.com/`. (Accessed on 10/14/2019).

[16] Genc, B., C. Evrendilek, and B. Hnich (2011). Covering points with orthogonally convex polygons. *Computational Geometry 44*(5), 249–264.

[17] Genkin, D., L. Pachmanov, I. Pipman, and E. Tromer (2016). Ecdh key-extraction via low-bandwidth electromagnetic attacks on pcs. In *Cryptographers Track at the RSA Conference*, pp. 219–235. Springer.

[18] Ghosh, D., F. Jin, and M. Maheswaran (2014). Jade: A unified programming framework for things, web, and cloud. In *2014 International Conference on the Internet of Things (IOT)*, pp. 73–78. IEEE.

[19] Gluck, R. (2017). Covering polygons with rectangles. In *International Conference on Theory and Applications of Models of Computation*, pp. 274–288. Springer.

[20] Groza, B. and B. Warinschi (2012). Revisiting difficulty notions for client puzzles and dos resilience. In *International Conference on Information Security*, pp. 39–54. Springer.

[21] Gu, G., R. Perdisci, J. Zhang, and W. Lee (2008). Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection.

[22] Gu, G., P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee (2007). Bothunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security Symposium*, Volume 7, pp. 1–16.

[23] Hada, H. and J. Mitsugi (2011). Epc based internet of things architecture. In *2011 IEEE International Conference on RFID-Technologies and Applications*, pp. 527–532. IEEE.

[24] Hu, H., D. Yang, L. Fu, H. Xiang, C. Fu, J. Sang, C. Ye, and R. Li (2011). Semantic Web-based policy interaction detection method with rules in smart home for detecting interactions among user policies. *IET communications 5*(17), 2451–2460.

[25] IBM (2019). Explore the Internet of Things (IoT) — IBM Watson IoT. `https://www.ibm.com/internet-of-things`. (Accessed on 07/03/2019).

[26] Jan, M. A., P. Nanda, X. He, Z. Tan, and R. P. Liu (2014). A robust authentication scheme for observing resources in the internet of things environment. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 205–211. IEEE.

[27] Jiang, L., C. Xu, X. Wang, and Y. Zhou (2015). Analysis and comparison of the network security protocol with dos/ddos attack resistance performance. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1785–1790. IEEE.

[28] Kalra, S. and S. K. Sood (2015). Secure authentication scheme for iot and cloud servers. *Pervasive and Mobile Computing 24*, 210–223.

[29] Khan, M. S. and D. Kim (2015). DIY interface for enhanced service customization of remote IoT devices: a CoAP based prototype. *International Journal of Distributed Sensor Networks 11*(10), 542319.

[30] Kiran, M. S., P. Rajalakshmi, K. Bharadwaj, and A. Acharyya (2014). Adaptive rule engine based iot enabled remote health care data acquisition and smart transmission system. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 253–258. IEEE.

[31] Koh, J. Y., J. T. C. Ming, and D. Niyato (2013). Rate limiting client puzzle schemes for denial-of-service mitigation. In *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1848–1853. IEEE.

[32] Kolias, C., G. Kambourakis, A. Stavrou, and J. Voas (2017). Ddos in the iot: Mirai and other botnets. *Computer 50*(7), 80–84.

[33] Kortuem, G., F. Kawsar, V. Sundramoorthy, D. Fitton, et al. (2009). Smart objects as building blocks for the internet of things. *IEEE Internet Computing 14*(1), 44–51.

[34] Kothmayr, T., C. Schmitt, W. Hu, M. Brünig, and G. Carle (2013). Dtls based security and two-way authentication for the internet of things. *Ad Hoc Networks 11*(8), 2710–2723.

[35] Leong, C. Y., A. R. Ramli, and T. Perumal (2009). A rule-based framework for heterogeneous subsystems management in smart home environment. *IEEE Transactions on Consumer Electronics 55*(3), 1208–1213.

[36] Liang, C.-J. M., B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu (2015). SIFT: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pp. 298–309. ACM.

[37] Liou, W., C. Y. Tang, and R. C. T. Lee (1991). Covering convex rectilinear polygons in linear time. *International Journal of Computational Geometry & Applications 1*(02), 137–185.

[38] Mainetti, L., V. Mighali, and L. Patrono (2015). An IoT-based user-centric ecosystem for heterogeneous smart home environments. In *2015 IEEE International Conference on Communications (ICC)*, pp. 704–709. IEEE.

[39] Masud, M. M., T. Al-Khateeb, L. Khan, B. Thuraisingham, and K. W. Hamlen (2008). Flow-based identification of botnet traffic by mining multiple log files. In *2008 First International Conference on Distributed Framework and Applications*, pp. 200–206. IEEE.

[40] Michalas, A., N. Komninos, N. R. Prasad, and V. A. Oleshchuk (2010). New client puzzle approach for dos resistance in ad hoc networks. In *2010 IEEE International Conference on Information Theory and Information Security*, pp. 568–573. IEEE.

[41] Mirkovic, J., S. Dietrich, D. Dittrich, and P. Reiher (2004). *Internet denial of service: attack and defense mechanisms (Radia Perlman Computer Networking and Security)*. Prentice Hall PTR.

[42] National-Instruments (2019a). Ds18s20.pdf. `https://datasheets.maximintegrated.com/en/ds/DS18S20.pdf`. (Accessed on 10/13/2019).

[43] National-Instruments (2019b). How to choose the right relay - national instruments. `http://www.ni.com/en-us/innovations/white-papers/06/how-to-choose-the-right-relay.html`. (Accessed on 10/13/2019).

[44] Oh, H., S. Ahn, J. K. Choi, and J. Yang (2017). Mashup service conflict detection and visualization method for Internet of Things. In *2017 IEEE 6th Global Conference on Consumer Electronics (GCCE)*, pp. 1–2. IEEE.

[45] Pammu, A. A., K.-S. Chong, W.-G. Ho, and B.-H. Gwee (2016). Interceptive side channel attack on aes-128 wireless communications for iot applications. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 650–653. IEEE.

[46] Peraković, D., M. Periša, and I. Cvitić (2015). Analysis of the iot impact on volume of ddos attacks. *XXXIII Simpozijum o novim tehnologijama u poštanskom i telekomunikacionom saobraćaju–PosTel 2015*, 295–304.

[47] Perumal, T., M. N. Sulaiman, S. K. Datta, T. Ramachandran, and C. Y. Leong (2016). Rule-based conflict resolution framework for Internet of Things device management in smart home environment. In *2016 IEEE 5th Global Conference on Consumer Electronics*, pp. 1–2. IEEE.

[48] Porambage, P., C. Schmitt, P. Kumar, A. Gurtov, and M. Ylianttila (2014). Pauthkey: A pervasive authentication protocol and key establishment scheme for wireless sensor networks in distributed iot applications. *International Journal of Distributed Sensor Networks 10*(7), 357430.

[49] Ranjan, S., J. Robinson, and F. Chen (2014, June 24). Machine learning based botnet detection using real-time connectivity graph based traffic features. US Patent 8,762,298.

[50] Raza, S., H. Shafagh, K. Hewage, R. Hummen, and T. Voigt (2013). Lithe: Lightweight secure coap for the internet of things. *IEEE Sensors Journal 13*(10), 3711–3720.

[51] Ren, J., H. Guo, C. Xu, and Y. Zhang (2017). Serving at the edge: A scalable iot architecture based on transparent computing. *IEEE Network 31*(5), 96–105.

[52] Rescorla, E. and N. Modadugu (2012). Datagram transport layer security version 1.2.

[53] Saad, S., I. Traore, A. Ghorbani, B. Sayed, D. Zhao, W. Lu, J. Felix, and P. Hakimian (2011). Detecting p2p botnets through network behavior analysis and machine learning. In *2011 Ninth annual international conference on privacy, security and trust*, pp. 174–180. IEEE.

[54] Samsung (2019). SmartThings. Add a little smartness to your things. `https://www.smartthings.com/`. (Accessed on 07/03/2019).

[55] Sarkar, C., S. A. U. Nambi, R. V. Prasad, and A. Rahim (2014). A scalable distributed architecture towards unifying iot applications. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pp. 508–513. IEEE.

[56] Shah, T. and S. Venkatesan (2018). Authentication of iot device and iot server using secure vaults. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pp. 819–824. IEEE.

[57] Shah, T. and S. Venkatesan (2019a). Conflict detection in rule based iot systems. In *2019 10th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE. (in press).

[58] Shah, T. and S. Venkatesan (2019b). A method to secure iot devices against botnet attacks. In *International Conference on Internet of Things*, pp. 28–42. Springer.

[59] Shah, T., S. Venkatesan, H. Reddy, S. Ardhanareeswaran, and V. Lokesh (2018). User customizable iot systems using self-aware sensors and self-aware actuators. In *International Conference on Cloud of Things and Wearable Technologies 2018*, pp. 12–20.

[60] Shehata, M., A. Eberlein, and A. Fapojuwo (2007). Using semi-formal methods for detecting interactions among smart homes policies. *Science of Computer Programming 67*(2-3), 125–161.

[61] Stirbu, V. (2008). Towards a restful plug and play experience in the web of things. In *2008 IEEE International Conference on Semantic Computing*, pp. 512–517. IEEE.

[62] Sun, Y., X. Wang, H. Luo, and X. Li (2014). Conflict detection scheme based on formal rule model for smart building systems. *IEEE Transactions on Human-Machine Systems 45*(2), 215–227.

[63] Tan, L. and N. Wang (2010). Future internet: The internet of things. In *2010 3rd international conference on advanced computer theory and engineering (ICACTE)*, Volume 5, pp. V5–376. IEEE.

[64] Tempest (2015). Tempest_attacks_against_aes.pdf. `https://www.fox-it.com/nl/wp-content/uploads/sites/12/Tempest_attacks_against_AES.pdf`. (Accessed on 10/19/2019).

[65] Walid, A., A. Mostafa, and M. Salama (2017). Malnod: malicous node discovery in internet-of-things through fingerprints. In *2017 European Conference on Electrical Engineering and Computer Science (EECS)*, pp. 280–285. IEEE.

[66] Yang, C., S. Lan, W. Shen, G. Q. Huang, X. Wang, and T. Lin (2017). Towards product customization and personalization in IoT-enabled cloud manufacturing. *Cluster Computing 20*(2), 1717–1730.

[67] Yang, S. J., A. S. Lee, W. C. Chu, and H. Yang (1998). Rule base verification using Petri nets. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*, pp. 476–481. IEEE.

[68] Yashiro, T., S. Kobayashi, N. Koshizuka, and K. Sakamura (2013). An internet of things (iot) architecture for embedded appliances. In *2013 IEEE Region 10 Humanitarian Technology Conference*, pp. 314–319. IEEE.

BIOGRAPHICAL SKETCH

Trusit Shah is a PhD in Computer Science at the University of Texas at Dallas. His current areas of research are security and privacy issues in IoT and User Customizable IoT architecture.

He completed his master's degree in Embedded Systems from BITS Goa, India. His area of research for his master's thesis was cross-layer wireless sensor network protocol. He was part of a research group at Wireless Sensor Lab in Birla Institute of Technology & Science, Goa, where he led a team of 5 undergraduate students in designing a wireless network for gas pipelines. His contribution helped the project to raise a fund of 9 million Indian Rupees (150k USD). He also provided solutions to local Embedded System based industries as a consultant while pursuing his undergraduate and master's degrees.

CURRICULUM VITAE

Trusit S. Shah

October 20, 2019

Contact Information:

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Email: trusit.shah@utdallas.edu

Educational History:

BE, Electronics and Telecommunications, Gujarat Technological University, India, 2012
ME Embedded Systems, Birla Institute of Technology and Science, India, 2014
MS, Computer Science, University of Texas at Dallas, 2019
PhD, Computer Science, University of Texas at Dallas, pursuing

TECHNIQUES FOR BUILDING ROBUST AND USER CUSTOMIZABLE IOT SYSTEMS
PhD Dissertation
Computer Science Department, University of Texas at Dallas
Advisors: Dr. S. Venkatesan

Design and development of Cross layered protocol stack for Condition Based Pipeline monitoring system using WSN
Master Thesis
Birla Institute of Technology and Science
Advisor: Dr. K.R. Anupama

Employment History:

Teaching Assistant, The University of Texas at Dallas, January 2015 – Present
Research Assistant, Birla Institute of Technology and Science, August 2013 – May 2014

Professional Memberships:

Institute of Electrical and Electronics Engineers (IEEE), 2015–present
Association of Computing Machinery (ACM), 2015–present