MODELING TRIGGER-ACTION IOT ATTACKS AND DEVISING REAL-TIME PROBABILISTIC DEFENSE MECHANISMS

by

Md Morshed Alam

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2024

Approved by:

Dr. Weichao Wang

Dr. Mohamed Shehab

Dr. Jinpeng Wei

Dr. Yonghong Yan

ABSTRACT

MD MORSHED ALAM. Modeling Trigger-Action IoT Attacks and Devising Real-time Probabilistic Defense Mechanisms. (Under the direction of DR. WEICHAO WANG)

Trigger-action Internet of Things (IoT) platforms allow users to leverage functional dependencies between IoT event conditions and actions to set up trigger-action rules in a rule engine, where event conditions act as *triggers* to the corresponding actions. When these user-defined rules are executed, they create a chain of interactions. IoT hubs utilize this chain to automate network tasks, invoke actions in various IoT devices based on triggers, and communicate with users about the physical changes in the network. However, adversaries exploit this chain to maliciously inject fake event conditions in the network and perform remote injection attacks. The objective here is to force the hubs to invoke invalid actions in target IoT devices violating rule integrity. Security mechanisms in the existing literature attempt to address this vulnerability either by deploying event verification systems to verify the physical occurrence of IoT events or by enforcing security compliance mechanisms to prevent unsafe and insecure event transactions in the network. Although these mechanisms are well suited for offline protection, they can barely provide realtime defense against agile remote injection attacks. Additionally, some of the mechanisms require modification of the source code of IoT mobile apps, making the defense solutions platform dependent.

In this dissertation, we present three novel research works to address this gap. First, we propose *IoTMonitor*, a Hidden Markov Model based security analysis system that discovers optimal attack paths from a set of physical evidence generated due to attack actions in the network. IoTMonitor learns attack behavior by continuously observing physical changes caused by event occurrences and determines the most likely IoT devices compromised by attackers. Second, we develop *IoTWarden*, a deep reinforcement learning (deep RL) based defense system that profiles attack actions

at runtime and takes necessary defense actions to obstruct the progression of ongoing attacks. We implement an LSTM-based recurrent neural network (RNN) to infer attack behavior at runtime and a Deep Q-Network (DQN) based function approximator to obtain optimal defense policies. The objective here is to train a defense agent with an optimal action policy so that the agent takes defense actions at runtime yielding maximum security gain. Third, we develop *IoTHaven*, a realtime defense system to mitigate remote injection attacks in partially observable IoT networks. In IoTHaven, the defense agent takes optimal actions at runtime against ongoing remote injection attacks under the uncertainty of actual network states maximizing the overall security gain. We design the decision process of the defense agent as a Partially Observable Markov Decision Process (POMDP). IoTHaven utilizes a Deep Recurrent Q-Network (DRQN) based function approximator to obtain optimal defense policies.

DEDICATION

I dedicate this dissertation to my mother, *Sabina Yasmin*, my uncle, *Kamrul Hassan Khan*, and my wife, *Israt Jahan*, for their continuous support, love, and sacrifice. I am deeply grateful for their contributions to my life and academic career.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my advisor, *Dr. Weichao Wang*, for his unwavering guidance, motivation, and support during my doctoral journey. He consistently encouraged and challenged me to improve as a researcher. His continued mentorship and assistance have positively impacted my academic career. I am fortunate to have such a talented, patient, and kind individual as my doctoral advisor.

I am deeply grateful to my dissertation committee members, *Dr. Mohamed Shehab*, *Dr. Jinpeng Wei*, and *Dr. Yonghong Yan*, for their dedicated efforts in providing valuable insights, suggestions, and feedback that have greatly enriched this dissertation. I would also like to acknowledge *Dr. Thomas Moyer* for his contributions to the initial stages of this dissertation.

I appreciate the love and support of my parents, family members, and friends, who have always helped me navigate life's challenges and experience significant and positive personal growth.

This dissertation wouldn't have been possible without the support of the UNC Charlotte community. I am grateful to everyone at UNC Charlotte for their assistance in completing this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: Introduction

In recent years, the deployment of Internet of Things (IoT) devices to facilitate home and industry automation has become a popular phenomenon. We also see an upward trend in the usage of IoT and smart devices in transportation, agriculture, and healthcare sectors. It is projected that we will have at least 41.6 billion connected IoT devices by 2025 [1]. The pervasiveness of IoT devices gives us the opportunity to make our workspace and living space autonomous, adaptive, and efficient. To make the automation easier, many IoT platforms now support the trigger-action capability. IFTTT [2], SmartThings [3], Microsoft Power Automate [4], Apple Home App [5], openHAB [6], Wink [7], and Zapier [8] are few such platforms.

1.1 Motivation

In trigger-action IoT platforms, IoT devices communicate with each other via IoT hubs. The occurrence of events in one IoT device invokes action(s) in another IoT device. Thus, event conditions and actions maintain functional dependencies, where actions in IoT devices are triggered based on the occurrence of relevant event conditions. Trigger-action IoT platforms allow users to set up trigger-action rules leveraging these functional dependencies. At runtime, when these rules are executed with the help of IoT hubs, we see a chain of interactions in the network representing a sequence of event conditions and corresponding actions. IoT hubs use this chain to automate network tasks and send notifications about the state of devices to users. However, adversaries can exploit this chain of interactions to perform remote injection attacks in the network and invoke safety-critical actions in target IoT devices [9] [10] [11]. For instance, an adversary can inject a fake thermometer reading of 120°F into a smart

home network faking a fire hazard situation to force the smart lock to open the front door in the absence of the homeowner violating rule integrity.

In the literature, we see two main types of security systems to address this vulnerability: 1) security compliance enforcement systems, and 2) event verification systems.

The former type of systems [9] [12] [13] ensure that all event transactions in IoT environments comply with the pre-defined security policies. They perform static analysis over the source code of IoT mobile apps to discover the underlying functional dependencies between event conditions and actions. The objective of performing static analysis is to identify hook points in the source code so that customized security codes can be instrumented. When these instrumented codes are executed at runtime, the security system generates a dynamic state machine representing the event transactions and verifies whether any event transaction violates the security policies defined for the network. It is also possible to generate a provenance graph to trace the flow of event transactions [14]. These systems are mainly designed to enforce the security compliance of event transactions and block any event transaction that doesn't comply with the policies. However, since these security systems rely on the static analysis of the IoT mobile apps to determine where to add customized security codes, they are pretty platform-dependent.

The event verification systems [15] [16] [17] [18], on the other hand, act as anomaly detection systems in IoT platforms. They utilize *physical event fingerprints* of IoT devices to verify the occurrence of IoT events in the physical environment and determine whether any existing event condition is maliciously injected into the network. They use rule-based and/or ML-based anomaly detection approaches to decide whether an existing event condition is anomalous [19]. To extract physical event fingerprints, these systems deploy a set of verifying sensors in IoT environments to continuously measure the physical channels of network devices and capture any noticeable physical changes (e.g., *light intensity*) due to the occurrence or injection of event conditions

(e.g., *light turned on*) in the network. When the physical event fingerprint of an IoT device is discovered, these systems harness that fingerprint to verify the corresponding event conditions and effectively determine whether the IoT device shows any anomalous behavior. These security systems work greatly as offline defense solutions against remote injection attacks, however, they barely provide real-time security against agile injection attacks with the capability of profiling defense actions and altering attack strategy at runtime. Additionally, they need full observability over the network states of the environment to function properly.

Based on the aforementioned discussions, it is evident that the existing literature needs platform-independent real-time security solutions against agile remote injection attacks. It also needs security solutions that can take effective defense actions at runtime in partially observable IoT networks. Moreover, it lacks research work on the utilization of statistical properties of IoT networks in detecting injection attacks and the development of probabilistic defense mechanisms.

1.2 Research Questions

In this dissertation, we aim to address the research gap in the state-of-the-art and solve the following three research questions:

- **RQ1**: How can we determine the optimal attack path from a stream of physical evidences emitted during a remote injection attack and identify the most frequently compromised IoT devices?
- **RQ2**: How can we design a security system that discerns the optimal defense policy for a fully observable IoT network to counter ongoing remote injection attacks at runtime?
- **RQ3**: How can we discern the optimal defense policy for a partially observable IoT network to counter ongoing remote injection attacks at runtime?

To solve RQ1, we propose *IoTMonitor* [11], a Hidden Markov Model (HMM) based security analysis system that determines the optimal attack path consisting of a sequence of event conditions based on a set of physical evidences collected by sensors during a remote injection attack. We use the Baum-Welch algorithm [20] to discover probabilistic relations between event conditions and the set of physical evidences. Later, we use the Viterbi algorithm [21] to discern the underlying optimal sequence of event conditions. IoTMonitor also identifies the IoT devices attackers mostly try to compromise to successfully implement remote injection attacks.

We solve RQ2 by developing a Deep Reinforcement Learning (DRL) based real-time defense system namely *IoTWarden* [19]. It allows a defense agent to profile attack actions based on their impacts on the IoT environment and takes defense actions following an optimal defense policy. To infer attack behavior, IoTWarden uses an LSTM-based [22] Recurrent Neural Network (RNN) and trains the RNN using a sequence of attack conditions. Additionally, IoTWarden implements a Deep Q-Network [23] to obtain optimal defense policy.

The final research work we cover in this dissertation is the design and implementation of *IoTHaven* [24], an online defense system to mitigate remote injection attacks in partially observable IoT networks with the trigger-action capability. We propose IoTHaven to solve RQ3. IoTHaven allows a defense agent to take optimal defense actions against an ongoing injection attack yielding maximum security gain. Hence, the agent discerns an optimal defense policy by observing a stream of physical evidences with the help of a Deep Recurrent Q-Network (DRQN) [25] function approximator. Since IoTHaven takes optimal defense actions under the uncertainties of network states, it can be deployed to secure heterogeneous and scalable trigger-action IoT platforms.

1.3 Dissertation Map

The rest of the dissertation is divided into 7 chapters. In Chapter 2, we share background knowledge of trigger-action IoT platforms and discuss some notable related works. In Chapter 3, we mathematically define the trigger-action based remote injection attack and provide a detailed threat model. We discuss IoTMonitor in Chapter 4 and explain how IoTMonitor extracts optimal attack paths from physical evidences. We also share an algorithm that identifies the most frequently targeted IoT devices in trigger-action IoT platforms. Later, we provide a detailed description of the IoTWarden defense system in Chapter 5 and the IoTHaven defense system in Chapter 6, respectively. In Chapter 7, we discuss some potential research works, and finally, we conclude the dissertation in Chapter 8.

CHAPTER 2: Background

In Chapter 1, we discuss how users create trigger-action rules to help automate network tasks and how IoT hubs execute these rules at runtime based on the occurrence of event conditions in the network. We also discuss how the generation of a chain of interactions enables attackers to exploit functional dependencies to implement remote injection attacks. In this chapter, we first provide an overview of a typical trigger-action IoT platform. Then, we show how event conditions and actions create a chain of interactions in a trigger-action platform. Later, we briefly discuss some challenges existing in popular trigger-action platforms. Finally, we introduce some relevant research works from the literature which attempt to address the vulnerability caused by the existence of chain of interactions.

2.1 Trigger-Action IoT Platform

Figure 2.1 shows the architecture of a typical trigger-action IoT platform. As we see in the figure, IoT devices are connected with a smart hub wirelessly, and the hub collaboratively works with a cloud backend and a companion app. The companion app (e.g., SmartThings mobile app [3]) is a single space to install SmartApps relevant to IoT devices in the network, and users can manage, automate, and control network IoT devices using these SmartApps. Whenever IoT devices communicate through the hub, the hub works with a cloud backend, where device handlers are utilized as software wrappers and API of the actual SmartApps. There is a permission system that can manage and control permission for the SmartApps and determine how privilege the SmartApps should be.

In a trigger-action platform, users install SmartApps based on the requirement

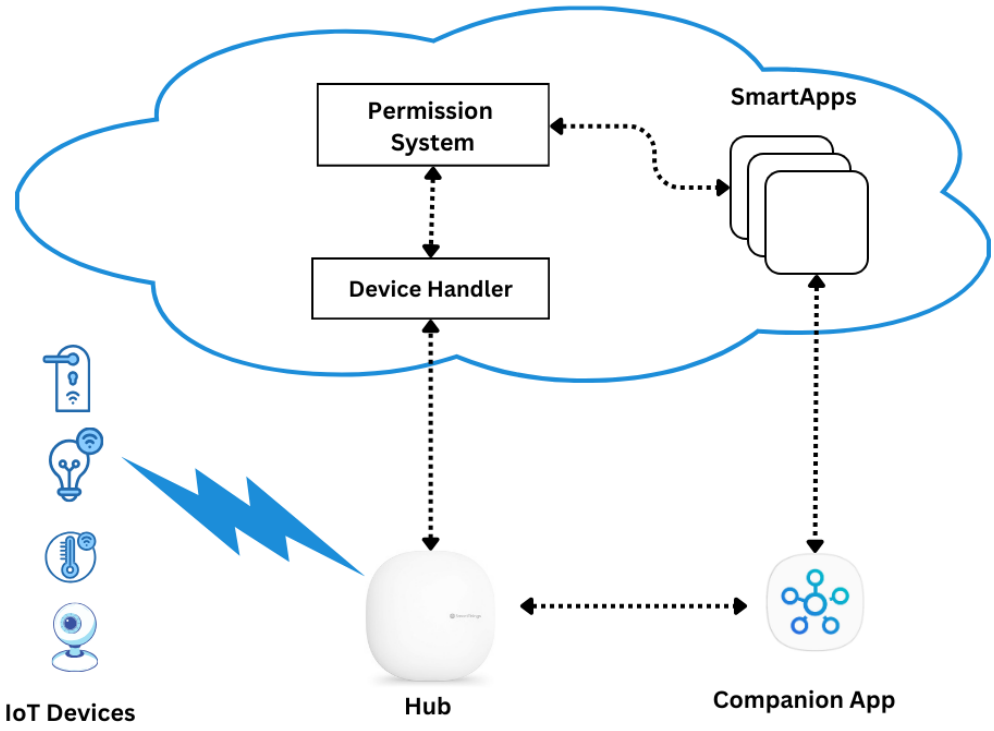

Figure 2.1: Architecture of a Trigger-action IoT Platform

of IoT devices and can manually set up the trigger-action rules. SmartApps then use device handlers to execute those rules at runtime and impact the functionalities of the network IoT devices. Since it is possible that a user buys IoT devices from multiple vendors, the user may need to install SmartApps developed by a diverse set of developers. Therefore, the companion app needs to support the installation of different types of SmartApps, even though the heterogeneity in SmartApps may end up creating some security vulnerabilities [26].

2.2 Chain of Interactions

In a trigger-action IoT platform, when the cyber state of an IoT device changes due to the occurrence of an event, the device reports the event condition to the smart hub to notify the new cyber state. Upon receiving the notification, the hub verifies the physical state of the device and invoke corresponding actions in relevant devices

dictated by user-defined rules. As we discuss in Chapter 1, the execution of user-defined rules creates a chain of interactions that enable the automation of network tasks. To understand how a chain of interactions is generated, let's consider the following use case:

"Alice has a smart home with five IoT devices: *smart lock (SLO)*, *motion detector (MD)*, *smart light (SLI)*, *smart coffee machine (COF)*, and *smart window (SW)*. She set the following rules in the rule engine:

- **R1**: When the motion detector senses motion, the smart light automatically turns on.
- **R2**: If the smart lock attached to the front door is unlocked, the smart light automatically turns on.
- **R3:** If the smart lock attached to the front door is unlocked and the motion detector senses motion, the smart coffee machine starts grinding coffee.
- **R4**: When the motion detector senses motion near the window and the smart light is turned on, the smart window automatically opens for ventilation.
- **R5**: When there is no motion detected by the motion detector for 5 minutes, the smart light turns off to save energy.
- **R6**: If the smart light is turned off and there is no motion at the home, the smart lock is automatically locked to secure the home from intruders.

Alice sets these six rules to automate tasks when she comes home from outside or leaves home for work. The ultimate objective is to let the IoT devices interact with each other and leverage the possible chain of interactions to perform automation."

In this use case, the execution of the rules creates a chain of interactions that enables automation in Alice's smart home. If we convert the rules using formal logic, we find the following statements:

$$R1 : MD_{motion\_detected} \rightarrow SLI_{on}$$

$$R2 : SLO_{unlocked} \rightarrow SLI_{on}$$

$$R3 : SLO_{unlocked} \wedge MD_{motion\_detected} \rightarrow COF_{grinding}$$

$$R4 : MD_{motion\_detected\_near\_window} \wedge SLI_{on} \rightarrow SW_{open}$$

$$R5 : MD_{no\_motion\_for\_5\_minutes} \rightarrow SLI_{off}$$

$$R6 : SL_{off} \wedge MD_{no\_motion} \rightarrow SLO_{locked}$$

When Alice comes home from outside and walks towards the window, the execution of the following chain of interactions ensures that the smart light is automatically turned on, the coffee machine is grinding coffee beans, and the window is open:

$$R2 \rightarrow R1 \rightarrow R3 \rightarrow R4$$

But when Alice leaves home for work, the following chain of interactions ensures that the smart home is secured and the energy used by the light is saved:

$$R5 \rightarrow R6$$

From the aforementioned use case, we can clearly see that when the user of a trigger-action IoT network carefully sets rules to represent trigger-action scenarios in the network, the sequential execution of a set of rules creates a chain of interactions that automates the network tasks and changes the status of IoT devices. Although the examples we see in this use case provide very simple chains, the real-life implementation of trigger-action capabilities in IoT networks may produce much more complex chain of interactions where multiple rules may concurrently impact single rule and vice versa.

2.3 Challenges in Trigger-action IoT Platforms

Trigger-action IoT platforms face challenges from multiple aspects, such as *risky inter-app interactions*, *design flaws in IoT platform permission systems*, and *design flaws in protocols*.

2.3.1 Risky IoT App Interactions

In trigger-action platforms, IoT apps interact with each other following user-defined rules. Users try to set up rules harnessing functional dependencies of the devices. It is possible that multiple user-defined rules are executed simultaneously letting the apps interact with each other and change the participating devices' status. This concurrent execution of IoT rules sometimes creates *conflicts* in device operation [27]. In smart home networks, a conflict can be defined as an unexpected situation where two or more competing environmental changes are invoked concurrently [28]. For example, when the temperature is beyond 100°F, a thermostat may ask a window to open, but an AC unit may ask the same window to close at the same time. We can also define conflict as an implicit interference between two rules trying to impact multiple devices with contradictory impacts on the network [29]. Conflicts in user-defined rules create inter-rule vulnerabilities [30], interference threats [31], and risky interaction chains [32].

In [30], we see that inter-rule vulnerabilities may cause 1) *condition bypassing* that allows security conditions to be bypassed by attackers to compromise the functionality of a target IoT device, 2) *condition blocking* that undesirably makes a trigger condition unsatisfied to invoke a necessary action in an IoT device next in the chain of interactions, and 3) *action looping* where trigger conditions can be manipulated to invoke same actions iteratively. In [31], *Cross-App Interference (CAI)* attack is explained. The primary vulnerability that makes CAI attacks possible comes from the interference of a rule's action towards the trigger conditions of another rule. Even

though IoT apps are given least privilege by the permission management system, the execution of user-defined rules from different apps and the implicit rule chain allow the adversaries to craft CAI attacks exploiting the dependency relations among IoT apps. Since a smart home network may deploy multiple trigger-action platforms together [33] [34], CAI attacks may also be performed in multi-platform multi-control-channel smart home environments [35]. IoT app interactions may also create risky interaction chains in the network [32], which can be exploited by attackers to violate rule integrity and perform safety-critical attacks.

2.3.2 Design Flaws in IoT Platform Permission System

We discuss in Section 2.1 that the cloud backend of a standard IoT platform contains a permission management system that manages and controls the SmartApps and the privileges they achieve. Since the permissions SmartApps granted to access sensitive resources of a network play a vital role in providing security of the network devices [36], any permission model that considers a coarse-grained definition of permission may undesirably end up giving *overprivilege* to SmartApps [37] [38] [39]. Jia et al. [40] argue that SmartApps users should be given *contexts* while granting permissions at runtime. They advocate for the fine-grained definition of permission so that the *overprivilege* issue can be handled effectively and the *contextual integrity* [41] of SmartApps can be maintained. If SmartApps are granted fine-grained context-specific permissions, it becomes easier to design an efficient access control system to manage the applications and their behavior [42]. It also becomes easier to extract contextual information from application source codes and design effective security policies to enforce at runtime [43] [9] [12] [44].

2.3.3 Design Flaws in Protocols

Zigbee [45] [46] and ZWave [47] are the two most popular communication protocols used in trigger-action platforms. MQTT [48], Matter [49], Thread [50], and

Bluetooth/BLE [51] are few other important communication protocols used in IoT platforms. In [52], we see that attackers near to a smart home can easily exploit the design flaws in ZWave protocol to perform attacks on an IoT network. We also see similar vulnerabilities caused by the design flaws of Zigbee protocol [53]. Similar to Mirai botnet attack [54], It is possible for malware to infect an IoT device first and then exploit the weakness of communication protocols to quickly spread payloads through the whole network enabling a possible DDoS attack [55] [56]. In [57], we see how worms can leverage the Zigbee chain to rapidly infect a whole IoT network. This type of vulnerability caused by the design flaws of IoT communication protocols make the widespread deployment of IoT devices in smart environments, including smart cities, quite challenging. The challenges trigger-action IoT platforms face due to the design flaws of communication protocols are well studied in [58], [59], [60], [61].

2.4 Literature Review

In this section, we discuss the relevant research works from the literature based on the following three criteria: 1) security analysis of trigger-action IoT platforms, 2) security policy compliance and rule integrity check, and 3) event verification and anomaly detection.

2.4.1 Security Analysis of Trigger-action IoT Platforms

The early research works in the literature mostly deal with the analysis of security vulnerabilities caused by benign or malicious execution of user-defined rules in trigger-action IoT platforms. In the literature, we see different types of security analysis to discover vulnerabilities caused by the execution of user-defined rules. Some focus on the detection of rules that prevent the execution of other important rules, while some focus on the rule conflicts or the presence of race conditions in user-defined rules. We also see research pertaining to the discovery of unexpected rule chains in the network.

Celik et al. [62] present an automated safety and security analysis of Samsung

SmartThings [3], one of the most popular trigger-action IoT platforms. The objective of the analysis is to determine how the execution of a user-defined rule unknowingly prevents the execution of a much needed rule and ends up damaging the safety and security of the user. For example, if there is a user-defined rule whose execution turns off the main water supply valve in the case of a water leak, the fire sprinkler may not be turned on later if there is a fire hazard situation. Authors proposed a static-analysis based rule validation approach, namely *Soteria*, to detect such types of rule prevention scenarios in trigger-action IoT platforms.

It is possible that an action in one particular device depends on the occurrence of multiple triggers, possibly in multiple devices. Therefore, user-defined rules pertaining to compound triggers also need to be investigated. In [63] and [30], authors attempt to find rule prevention issues in compound triggers by performing static code analysis over SmartApp source code. The objective is to extract triggers and corresponding actions by tracking information flow and identifying function calls. We see similar type of approach in [9], [12], and [13].

To detect rule conflicts and race conditions, researchers also adopt static analysis based approach, but the key focus is to determine rules with opposite effects that execute simultaneously in the same device. In [30], [31], and [64], authors perform security analysis to detect conflict between any two user-defined rules executed in a particular IoT device. To give an example of a possible rule conflict, consider the following two rules: *R1*: *unlock the front door when the temperature is 105˚F*, and *R2: when it is 8:00 pm, lock the front door*. Now, if the thermostat gives a reading of 110°F at 8:30 pm, a rule conflict will emerge in the network. The aforementioned research works attempt to discover these types of conflicts in the network. However, in smart homes and other IoT networks with trigger-action capability, the interactions among IoT devices often create a chain of interactions [37]. Therefore, the rules that create a chain of interactions need to be checked for possible rule conflict or race condition. [65]

and [66] offer two security analysis approaches to detect such conflicts in rule chain. In smart home network, sometimes the emergence of an unexpected chain of interactions does not fully reflect the status change of the network devices [67] [29]. Under that circumstance, the detection of conflict in rules is challenging. To detect such implicit rule chain, researchers propose NLP-based contextual information analysis of the trigger-action rules set by users [29]. It is also possible that any physical change in IoT environment may cause an undesired chain of interactions [68]. In that case, we have to consider environmental factors that may cause the unexpected chain of interactions when we perform security analysis. In [69], authors propose *IoTMon*, a security analysis tool to discover unexpected rule chains. They also discuss how the risk levels of each chain can be evaluated automatically. A more detailed overview of the relevant literature can be found in [70].

In the literature, we also see research works related to the security analysis of trigger-action platforms where security vulnerabilities are caused by other factors apart from the user defined rules. Fernandes et al. [37] perform a detailed security analysis on smart home platforms and find out that SmartApps often enjoy over-privilege. The privileges assigned to SmartApps are often coarse-grained, and it is possible for a SmartApp to have authorization for the access of two different IoT devices. In SmartThings platform, SmartApps are not allowed to directly access the IoT devices. They have to access the devices through device handlers located in the cloud backend. However, the coarse granularity of the *capability* assigned to SmartApps still make the access to IoT devices insecure and unsafe. This vulnerability can be addresses through the implementation of fine-grained privilege mechanism for the SmartApps [14] [9]. Trigger-action IoT platforms may also cause data leakage problems [71]. However, these security vulnerabilities are out of context for this dissertation.

2.4.2 Security Policy Compliance and Rule Integrity Check

To determine whether rule executions or event transactions in trigger-action IoT platforms comply with the defined security policy of the network and verify whether the integrity of user-defined rules is violated, researchers mostly utilize static analysis based approaches. In [62], authors adopted static analysis based model checking approach to detect undesired rule prevention or rule collision scenarios. The goal is to create a state machine from a set of extracted events/actions and check whether any state transition violates the defined security properties. Hence, security properties contain event constraints and required past event occurrences. The events/actions used to define states and transitions are usually extracted by performing static analysis over the source code of the IoT mobile apps developed to control the functionalities of IoT devices. We see similar approaches in IoTGuard [9] to detect unsafe and insecure state transitions, but IoTGuard instruments customized codes into app source code to generate a dynamic system model at runtime to perform model checking. The instrumented codes are carefully placed in hook points and API calls which are determined through the static analysis.

We also see model checking based rule integrity verification approaches in [30] [63] [72]. In [30], authors utilize a rule parser to convert IFTTT rules into fine-grained and pre-formatted rules which are further used to generate states in a state machine. Using model checking, they later verify whether any state transition violates the security properties. In [63], authors propose *IoTCom* that considers event conditions, actions, and a Behavioral Rule Graph (BRG) to detect rule prevention/collision and unexpected underlying rule chain using model checking. BRGs are created from Inter-procedural Control Flow Graphs (ICFGs) which are generated from IFTTT rules using a path-sensitive analysis method [73] at the first place. ICFGs usually represent the logical control flow of the rules. In [72], authors use a similar approach to IoTCom to generate a finite state machine (FSM) from ICFGs, but here the model

checking system also performs model slicing and state compression over the FSM.

Beyond model checking, we also see different types of approaches in the literature to detect rule integrity violation or security policy compliance check. In [31], we see a syntax-based conflict detection method for the SmartThings platfrom, while NLP-based methods are used to detect rule collision [29] and unexpected rule chains [69] [74] in trigger-action IoT platforms.

2.4.3 Event Verification and Anomaly Detection

Since actions in IoT devices are triggered by the event occurrences in the network, event verification plays a vital role in designing an effective anomaly detection system to determine the presence of any attack in the platform. The latest research works related to the detection of remote injection attacks mostly focus on the event detection with the help of an array of sensors.

In PEEVES [16] [75], physical event fingerprints are used to verify the occurrence of IoT events. A set of deployed sensors continuously measure the physical channels of the IoT devices, and whenever there is any change in the physical environment, the relevant sensors capture the change as a *physical evidence*. To verify the occurrence of any event, this type of physical evidence is used as the signature of the occurred event. A trigger-action IoT network usually contains IoT devices with different modalities and types. The event occurrence in those devices leaves unique identifying signatures in the physical environment, and therefore, the physical event fingerprints captured by sensors in PEEVES give an opportunity to design a rule-based or an ML-based anomaly detection system.

In [17], authors propose *HaWatcher*, a semantics-aware anomaly detection system that uses a *Shadow Execution Engine* to determine the benign behavior of the IoT devices and detect anomaly in rules based on the semantics of IoT mobile app, device type, configurations etc. HaWatcher discovers device relations and harnesses the correlation extracted by an NLP model to detect and further resolve rule conflicts.

In [76], authors propose a context-aware security framework for smart home systems, namely *Aegis*. Aegis models the behavioral change in IoT devices and sensors due to user activities and develops a context-aware model to determine whether IoT devices or sensors are acting maliciously. Additionally, it uses contextual information extracted from IoT mobile apps to understand the trigger-action scenario better. Aegis utilizes Markov Chain based approaches to detect anomalous behavior in network and inform users about the suspicion through alerts and notifications.

Since complex physical relations exist between IoT devices and sensors [77] [78] [32], the literature needs anomaly detection systems that can perform event verification considering the following requirements:

- An event occurrence is impacting multiple physical channels, and therefore, multiple sensors capture physical evidences related to the single event occurrence.

- A sensor captures a set of physical evidences generated due to the occurrence of multiple events.

In [15], authors propose an event verification system that considers the aforementioned requirements and extracts multiple fingerprints related to the occurrence of one IoT event. The event verification system is designed to be robust against evasion attacks where attackers may exploit the complex relations between devices and sensors to evade anomaly detection procedure. The authors also propose two patching mechanisms against the evasion attack: 1) event verification system sofware patching, and 2) sensor location patching.

CHAPTER 3: Modeling Trigger Action Based IoT Attacks

In trigger-action IoT platforms, the execution of user-defined rules representing functional dependencies between event conditions and actions create a chain of interactions. Attackers inject fake event conditions to this chain to perform remote injection attacks and activate invalid actions in target IoT devices. In this chapter, we mathematically model the remote injection attack in trigger-action IoT platforms and share some directions about the possible defense solutions. We also discuss a threat model that illustrates attacker's capabilities in implementing the attack. Let's start with a possible attack scenario.

3.1 A Trigger-action Attack Scenario

Assume that Alice has a number of trigger-action enabled IoT devices in her smart home, including a *smart lock*, a *motion detector*, a *smart light*, an *accelerometer*, a *smart coffee machine*, and a *smart window*. Alice controls the functionality of each device through a mobile application from her cell phone. The devices are wirelessly connected to a hub and communicate with each other through the hub.

Alice sets up the following trigger-action rules:

- **R1**: When the smart lock of the front door is unlocked and the motion detector senses motion of walking, the smart light turns on.
- **R2**: If the smart light is turned on, the smart window opens and the smart coffee machine starts grinding coffee beans.
- **R3**: When there is any change in vibration, the accelerometer located in her bedroom measures the new vibration.

- **R4**: When the accelerometer measures any change in vibration in the home, it asks the smart lock to lock the front door.

- **R5**: If the front door is closed and the smart lock is locked, the smart light turns off and the smart window closes.

Alice sets up these rules to automate the following tasks:

- **T1**: When she comes home from outside, unlocks the front door, and walks towards the living room, she wants the smart light turned on, the smart window opened, and the smart coffee machine grinding coffee beans.

- **T2**: When the coffee is ready, she wants to take the coffee mug and enters into her bedroom. When she is in her bedroom, she wants the front door closed and locked.

- **T3**: When she leaves home for work, she wants the smart light turned off and the smart window closed.

Since her IoT devices support trigger-action capabilities, the execution of **R1** and **R2** automates **T1**, the execution of **R3** and **R4** automates **T2**, and the execution of **R5** automates **T3**.

Now, Bob, an attacker, wants to manipulate the behavior of the smart window. He wants the smart window opened when Alice is not at home and the front door is securely closed. To achieve his attack goal, he aims to inject fake event conditions into the chain of interactions. As we see in **R2**, the *window opening* action is pre-conditioned upon the *turn on* event in the smart light. We also see in **R1** that the *turning on* action in the smart light depends on the *door unlock* event in the smart lock and the *sense motion* event in the motion detector. The goal of Bob is to inject fake *door unlock* and *sense motion* event conditions into the chain of interactions so

that **R1** and **R2** can be maliciously executed and the smart window can be opened invalidly.

3.2 Threat Model

We assume that the attacker injects fake event conditions into the hub and force the hub to invoke actions in target IoT devices. The attacker impersonates valid IoT devices using computer programs called *ghost devices* [9]. The ghost devices impersonate IoT devices just by mimicking their characteristics and functionalities. The attacker collects valid ID and credentials of IoT devices from manufacturer's websites or public repositories, including GitHub so that the ghost devices remain unsuspicious to the network [79]. We also assume that the attacker is aware of the network setting. He is capable of performing reconnaissance operations or filtering network traffic using side channel attacks [18]. He is also capable of extracting real-time event information from network traffic using different IoT network analysis tools [80] [18]. He may be able to profile the defense actions and figure out the impact of IoT event occurrences, but these capabilities depend on the strategy of the attacker. The attacker always wants to evade detection by the defense solutions, and therefore, we assume that the attacker aims to perform attack actions in a minimally invasive way, i.e., the attacker doesn't perform injection operation if it is not needed.

3.3 Our Assumptions

We consider our trigger-action IoT platform a Markovian system, where the action invoked in an IoT device at a time instance only depends on the event conditions occurred at the previous time instance and does not depend on all other past event conditions that already influenced actions in other IoT devices. We assume that a configured trigger-action sequence contains N distinguishable event conditions: $c_1, c_2, ..., c_N$. The attacker injects a single fake event condition c_i or a set of fake event conditions $\{c_i\}, 1 \leq i \leq N$ in the chain of interactions to compromise the

functionality of a device of interest. Note that the attacker is capable to manipulate an existing chain and maliciously inject a fake event condition somewhere in the chain to impact the future event occurrences and actions. So, it's not mandatory for the attacker to start the chain of interactions although the attacker may choose to do it if his attack strategy instructs him to do that.

In trigger-action IoT networks, whenever an event occurs, it causes some physical changes in the environment, which can be perceived as a set of corresponding observations $\{o_i\}, 1 \leq i \leq M$. The observations are actually *physical evidences* emitted due to the occurrence of IoT events in the network. From the defense point of view, an array of sensors can be placed in the environment to capture these evidences so that the actual event occurrences can be verified based on their impact on the physical environment. Note that some event occurrences may trigger non-observable evidences, but others may trigger more than one evidence.

3.4 Probabilistic Model of a Trigger Action Based Attack

Since our ultimate objective is to provide a defense solution, we model the trigger-action attack as a *Hidden Markov Model (HMM)* problem. Event conditions reported to the hub remain unobservable to the analysis and defense agents. In Chapter 4, we discuss how a security analysis agent discovers optimal attack paths just by observing a sequence of physical evidences generated during a trigger-action IoT attack. In Chapters 5 and 6, we show how a defense agent takes optimal actions based on the observations emitted as physical evidences. Since event conditions reflect the status changes of the IoT devices, it is reasonable to encode them into hidden states of an HMM. A security analysis agent or a defense agent cannot observe these hidden states, but they can try to infer the hidden states by continuously observing the stream of physical evidences generated in the network.

We assume that when event conditions $\{c_t\}$ are reported to the hub at time t, an agent only observes the corresponding physical evidences $\{o_t\}$ emitted in the network.

The tasks of the agents are to determine the probabilistic relationship between event conditions and evidences, employ it to figure out the optimal attack paths, diagnose the crucial nodes in those paths, and finally devise an appropriate defense mechanism to thwart any future attacks.

We model a trigger action based attack as a 5-tuple HMM problem $\langle X, Y, Q, E, \sigma \rangle$, where there are N true states $X = \{x_1, x_2, ..., x_N\}$, and M observation states $Y = \{y_1, y_2, ..., y_M\}$. We define each x_i and y_j as follows:

- true state, x_i : state of responding to the occurrence of c_i
- observation state, y_j : a subset of the set of physical evidences $\{o_1, o_2, ..., o_M\}$, which are emitted when the environment makes transition to a new state

Note: In Chapter 4, we discuss in detail how we generate *state machines* representing the dynamic of our IoT environment that consists of these true states $\{x_i\} \in X$ and observation states $\{y_j\} \in Y$.

The term $Q = \{q_{ij}\}$ in our HMM definition is called the *state transition probability distribution* which is defined as:

$$q_{ij} = Pr(X_{t+1} = x_j | X_t = x_i), \quad 1 \leq i, j \leq N \tag{3.1}$$

Hence, q_{ij} is the probability of transitioning from the true state $x_i \in X$ at time t to any $x_j \in X$ at time $t+1$. When the environment makes a transition to the true state x_j, the environment emits a new observation $y_k \in Y$ with an *emission probability* $\mu_j(y_k) \in E$ which can be defined as:

$$\begin{aligned} \mu_j(y_k) = Pr(Y_{t+1} = y_k | X_{t+1} = x_j), \quad & 1 \leq j \leq N \\ & 1 \leq k \leq M \end{aligned} \tag{3.2}$$

The quantity $E = \{\mu_j(y_k)\}$ in the equation (3.2) is termed as *emission probability distribution* that represents the probabilistic relationship between evidences and

actual IoT events.

The final parameter of the definition of HMM, $\sigma = \{\sigma_i\}$, is termed as *initial state distribution* which can be defined as:

$$\sigma_i = Pr(X_1 = x_i), \quad 1 \leq i \leq N \tag{3.3}$$

Hence, σ_i is the initial state distribution at time instance $t = i$. It should be noted that the environment can be in any state at any time instance and therefore, we always need to maintain this σ_i to represent the likelihood of the environment being at a certain state $x_i, \forall i \in [1, N]$ at a time instance $t = i$.

3.5 Solving the HMM Problem and Possible Research Directions

We can solve this HMM problem to address the following research questions:

- How can we discern an optimal attack path by just observing a stream of physical evidences in a trigger-action IoT environment?
- How can we identify the most vulnerable nodes in an optimal attack path?
- How can we design an efficient defense solution for a trigger-action IoT platform where the defense agent only sees a set of physical evidences?

CHAPTER 4: Determining Optimal Attack Path and Identifying Crucial Attack Nodes

4.1 Introduction

Based on our discussion in Chapter 1, it is pretty evident that trigger-action platforms in IoT domain are vulnerable towards malicious event injection attacks. Attackers exploit chain of interactions to inject fake event conditions in the network and invoke undesirable actions in target IoT devices. In this chapter, we discuss how security analysis can be performed in trigger-action platforms to obtain optimal attack paths and which nodes in the optimal attack paths are frequently compromised by attackers to perform remote injection attacks.

4.1.1 Research Motivation

Since IoT devices in trigger-action platforms create a chain of interactions maintaining functional dependencies between event conditions and actions [9] [10], it is possible for adversaries to remotely inject malicious events somewhere in the interaction chain using a ghost device and activate safety-critical actions through the exploitation of autonomous trigger-action scenarios. For instance, in a smart home, an adversary can inject a fake thermometer reading of 110°F into the chain to maliciously simulate a fire hazard situation and trick the hub to ask the smart lock to open the front door in the absence of the home owner.

The existing literature includes a number of research efforts that attempt to solve such vulnerabilities caused by the trigger-actions in an IoT network. Most of them focus on validating security properties by identifying the unsafe or insecure state transitions in the network [9] [12] [13] and proposing countermeasures to prevent those

unsafe and insecure transitions. There is another line of research attempts where policy violations are addressed by checking sensitive user actions that may violate security policies [13]. The literature also includes research attempts where physical fingerprints of the IoT devices are extracted using machine learning techniques and further utilized to verify whether or not a certain event actually occurs [16]. Unfortunately, none of these efforts deal with the possible attack paths the attackers may choose to perform trigger-action based attacks and propose defense mechanisms to secure the most vulnerable nodes in those paths.

4.1.2 Our Contributions

We choose to propose *IoTMonitor*, a security system that adopts a Hidden Markov Model (HMM) based approach to determine optimal attack paths an attacker may follow to implement a trigger-action based attack, thus providing suggestions for subsequent patching and security measures. We envision our system examining the physical changes happening in an IoT environment due to the event occurrences, discovering the probabilistic relationship between physical evidences and underlying events using the Baum-Welch algorithm [20] [81], and discerning the optimal attack paths using the Viterbi algorithm [21]. When an optimal attack path is determined, we deploy IoTMonitor to identify the crucial nodes in the path that the attacker must compromise to carry out the attack. Such information can be used for prioritizing security measures for IoT platforms.

The contributions of this research work can be summarized as follows:

- We propose IoTMonitor, a Hidden Markov model based security system that identifies optimal attack paths in a trigger-action IoT environment based on the probabilistic relationship between actual IoT events and corresponding physical evidences;

- We implement the Baum-Welch algorithm to estimate transition and emission

probabilities, and the Viterbi algorithm to discern optimal attack paths;

- We propose an algorithm to detect the crucial nodes in an extracted optimal attack path, thus providing guidelines for subsequent security measures;
- We thoroughly evaluate the performance of IoTMonitor in detecting optimal attack paths and achieve high accuracy scores.

4.2 Problem Statement

Formulating the trigger-action attack as an HMM problem $\langle X, Y, Q, E, \sigma \rangle$, our objective is to answer the following questions:

1. How to determine the optimal value of $\theta = (\sigma, Q, E)$ given a complete observation sequence $Y = \{Y_1, Y_2, ..., Y_T\}$ using the following equation:

$$\theta^* = \underset{\theta}{argmax}\ Pr(Y_1, Y_2, ..., Y_T|\theta) \tag{4.1}$$

where, $Y_t \in \{y_1, y_2, ..., y_M\}$, and each y_j represents a subset of the physical evidence $\{o_1, o_2, ..., o_M\}$ captured by the deployed sensors in the environment?

2. How to determine the quantity $\omega_t(i)$ given a particular observation sequence $\langle Y_1, Y_2, ..., Y_t \rangle$ at time instance t and $Y_t = y_k$ such that

$$\begin{aligned} \omega_t(i) = \underset{x_1,...,x_{i-1}}{max} \Big\{ Pr(X_1 = x_1, ..., X_t = x_i, Y_1, ..., Y_t = y_k|\theta) \Big\}, \\ 2 \leq t \leq T,\ 1 \leq i \leq N \end{aligned} \tag{4.2}$$

where, $\omega_t(i)$ is the maximum probability of the occurrence of a particular state sequence $\langle x_1, x_2, ..., x_i \rangle$ at time instance t that corresponds to the aforementioned observation sequence $\langle Y_1, Y_2, ..., Y_t \rangle$?

3. How to detect the most crucial nodes in the attack chain the attacker must compromise to make an attack successful?

4.3 The IoTMonitor System

Since the attacker exploits trigger-action functionality of IoT network to manipulate a chain of interactions by injecting fake events, we can thwart a trigger-action attack effectively if we can identify the optimal attack path the attacker may follow and perform security hardening on the crucial nodes in the attack path. In this research work, we propose *IoTMonitor*, a security system that discerns the optimal attack paths by analyzing physical evidences generated during the attack cycle, which are probabilistically correlated to the actual underlying events. IoTMonitor formulates the attack as a Hidden Markov Model (HMM) problem (discussed in detail in chapter 3) and solves it to determine the most likely sequence of events occured during an attack cycle and further identifies the crucial nodes in that sequence. Hence, in this chapter, a *node* represents an event occurring at a particular device.

4.3.1 IoTMonitor Architecture

The proposed IoTMonitor comprises three main components:

1. State machine generator
2. Sequence extractor
3. Crucial node detector

Fig 4.1 shows the architecture of IoTMonitor. We discuss the components below in detail.

4.3.1.1 State Machine Generator

When events occur in the environment and the deployed sensors capture corresponding evidences per event occurrence, this component will construct a *state machine* to represent how the envrionment state changes across a series of time instances $t = 1, 2, ..., T$. Hence, *states* delineate useful information regarding the occurrence of different event condition $\{c_i\}$ and corresponding evidences $\{o_i\}$.

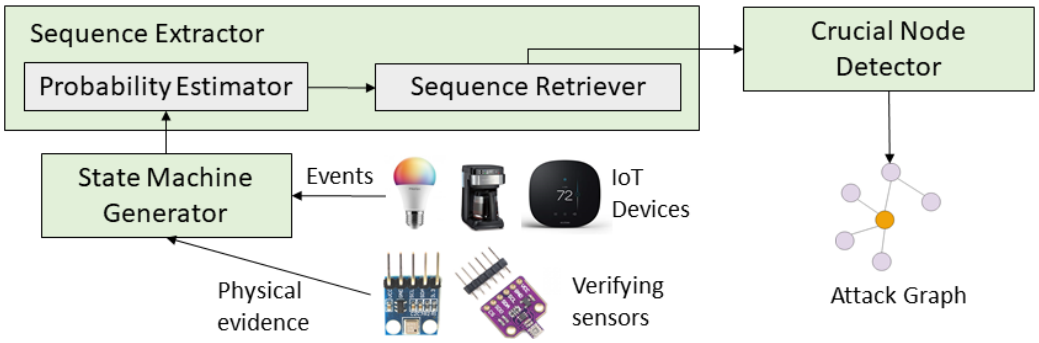

Figure 4.1: IoTMonitor System

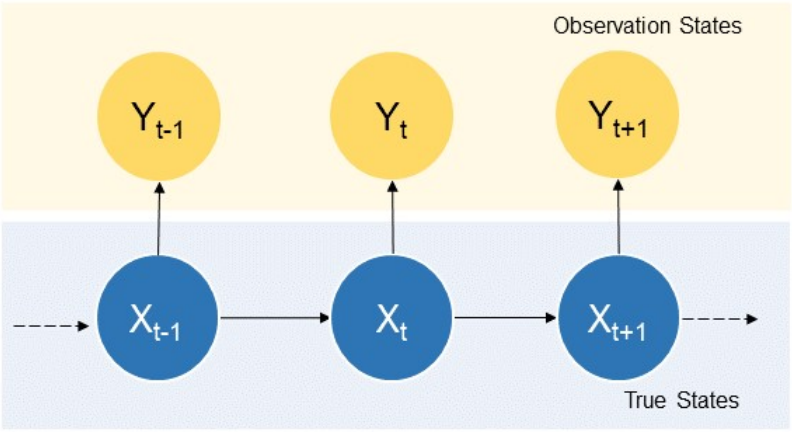

Figure 4.2: A Sample State Machine

The state machine accommodates two types of states:

1. **True states**: states that correspond to the actual event occurrences in the network, and

2. **Observation states**: states that represent physical evidences emitted due to the occurrence of event conditions in the network.

Hence, the true states remain hidden, but the observation states are leveraged to infer the hidden true state sequence. We formulate the HMM $\langle X, Y, Q, E, \sigma \rangle$ and define the state space using the same configurations shared in chapter 3. Figure 4.2 shows a sample state machine where *blue* circles represent the true states and *yellow* circles represent the observation states.

Hence, we assume that there are N true states $X = \{x_1, x_2, ..., x_N\}$, and T observation states $Y = \{y_1, y_2, ..., y_T\}$ in the state machine, where X_t and Y_t, respectively, denote the true state and observation state at time t. Here, $T \leq M$, and each y_j contains a subset of the physical evidences $\{o_1, o_2, ..., o_M\}$, where the total number of evidences is M. Note that each observation state Y_t in our experiment is determined with the help of a *sliding window* function, which is discussed in the later half of this chapter.

Note: For the rest of the chapter, we call *observation state* as only *observation* and use the terms *true state* and *state* interchangeably to mean the same thing. We also interchangeably use the terms *event condition* and *event* to mean the same aspect.

4.3.1.2 Sequence Extractor

Once the trigger action sequence is modeled as an HMM problem, IoTMonitor attempts to estimate the probability values and retrieves the optimal hidden state sequence from the observations. First, it starts with estimating the converged state distributions, transmission probabilities, and emission probabilities. Then, it seeks to figure out the underlying state sequence that maximizes the probability of getting a certain observation sequence. To accomplish both tasks, the *sequence extractor* employs the following two subcomponents: a) probability estimator, and b) sequence retriever. The details of both subcomponents are described below.

a) **Probability Estimator**: Given a complete observation sequence $\langle Y_1, Y_2, ..., Y_T \rangle$, the goal of this component is to determine the following:

$$\theta^* = \underset{\theta}{argmax}\ Pr(Y_1, Y_2, ..., Y_T|\theta) \tag{4.3}$$

We use the Baum-Welch algorithm [20] [81] to iteratively update the current model θ and solve equation (4.3). It uses a *forward-backward procedure* to find the maximum

likelihood estimate of θ given a certain set of observations. We assume that each observation Y_t is emitted by the environment at one discrete time instance $t = 1, 2, ..., T$.

Forward-backward Procedure: Let $\alpha_t(i)$ and $\beta_t(i)$ be the probabilities of getting the observation sequences $\langle Y_1, Y_2, ..., Y_t \rangle$ and $\langle Y_{t+1}, Y_{t+2}, ..., Y_T \rangle$, respectively, while the system is being in the true state x_i at time t. Note that the state x_i is not observable to the agent and therefore, it makes an inference over x_i from the given observation sequence. The probabilities can be formulated as:

$$\begin{aligned} \alpha_t(i) &= Pr(Y_1, Y_2, ..., Y_t, X_t = x_i|\theta) \\ \beta_t(i) &= Pr(Y_{t+1}, Y_{t+2}, ..., Y_T|X_t = x_i, \theta) \end{aligned} \tag{4.4}$$

We can compute $\alpha_t(i)$ and $\beta_t(i)$ by following the following steps:

1. **Initialization**:

$$\begin{aligned} \alpha_1(i) &= \sigma_i \mu_i(Y_1), \quad 1 \leq i \leq N \\ \beta_T(i) &= 1, \quad 1 \leq i \leq N \end{aligned} \tag{4.5}$$

2. **Induction**:

$$\begin{aligned} \alpha_{t+1}(j) &= \mu_j(Y_{t+1}) \sum_{i=1}^{N} \alpha_t(i) q_{ij}, \;\; 1 \leq t \leq T-1, \;\; 1 \leq j \leq N \\ \beta_t(i) &= \sum_{j=1}^{N} q_{ij} \mu_j(Y_{t+1}) \beta_{t+1}(j), \; t = T-1, ..., 2, 1, \;\; 1 \leq i \leq N \end{aligned} \tag{4.6}$$

These two steps combined is called the *forward-backward procedure*, and $\alpha_t(i)$ and $\beta_t(i)$ are termed as *forward variable* and *backward variable*, respectively.

Now, suppose $\delta_t(i)$ is the probability of the system being in the true state x_i at time instance t given the complete observation sequence $\langle Y_1, Y_2, ..., Y_T \rangle$ and the current model θ. We can define this probability in terms of the forward and backward variables $\alpha_t(i)$ and $\beta_t(i)$, i.e.,

$$\begin{aligned}\delta_t(i) &= Pr(X_t = x_i|Y_1, Y_2, ..., Y_T, \theta) \\ &= \frac{Pr(X_t = x_i, Y_1, Y_2, ..., Y_T|\theta)}{Pr(Y_1, Y_2, ..., Y_T|\theta)} \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}\end{aligned} \tag{4.7}$$

Again, given the complete observation sequence $\langle Y_1, Y_2, ..., Y_T \rangle$ and the current model θ, suppose, $\xi_t(i, j)$ is the probability of the system being in the true states x_i and x_j at time instances t and $t + 1$, respectively. So,

$$\begin{aligned}\xi_t(i, j) &= Pr(X_t = x_i, X_{t+1} = x_j|Y_1, Y_2, ..., Y_T, \theta) \\ &= \frac{Pr(X_t = x_i, X_{t+1} = x_j, Y_1, Y_2, ..., Y_T|\theta)}{Pr(Y_1, Y_2, ..., Y_T|\theta)} \\ &= \frac{\alpha_t(i)q_{ij}\beta_{t+1}(j)\mu_j(Y_{t+1})}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)q_{ij}\beta_{t+1}(j)\mu_j(Y_{t+1})}\end{aligned} \tag{4.8}$$

Now, we can update the initial state distribution $\bar{\sigma}_i$, transition probability $\bar{q}_{ij}$, and emission probability $\bar{\mu}_j(y_k)$ for $Y_{t+1} = y_k$ using these two parameters $\delta_t(i)$ and $\xi_t(i, j)$. The state distribution can be updated as:

$$\bar{\sigma}_i = \delta_1(i) \tag{4.9}$$

where, $\delta_1(i)$ is the probability of the system being in the true state x_i at time instance $t = 1$.

To update the transition probabilities, we have to compute the ratio of *the expected number of state transitions from x_i to only x_j* (the numerator of the equation (4.10)) and *the expected number of transitions from x_i to all other true states* (the denominator of the equation (4.10)).

$$\bar{q}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \delta_t(i)} \tag{4.10}$$

And to update the emission probabilities, we have to take the ratio of two other quantities: *the expected number of times being in state x_j and observing the observation y_k* (the numerator of the equation (4.11)), and *the expected number of times being in state x_j* (the denominator of the equation (4.11)).

$$\bar{\mu}_j(y_k) = \frac{\sum_{t=1}^{T} 1_{(Y_{t+1}=y_k)} \delta_t(j)}{\sum_{t=1}^{T} \delta_t(j)} \tag{4.11}$$

where,

$$1_{(Y_{t+1}=y_k)} = \begin{cases} 1, & \text{if } Y_{t+1} = y_k \\ 0, & \text{Otherwise} \end{cases} \tag{4.12}$$

The updated parameters $\bar{\sigma} = \{\bar{\sigma}_i\}$, $\bar{Q} = \{\bar{q}_{ij}\}$, and $\bar{E} = \{\bar{\mu}_j(y_k)\}$ now constitute the new model $\bar{\theta} = (\bar{\sigma}, \bar{Q}, \bar{E})$. We need to iterate the equations (4.9) (4.10), and (4.11) until we find $\bar{\theta} \approx \theta$. This convergence is guaranteed in [81] by Baum et al., where it is ensured that either 1) the initial model θ defines a critical point in the likelihood function where $\bar{\theta} = \theta$, or 2) $\bar{\theta}$ explains the observation sequence $\langle Y_1, Y_2, ..., Y_T \rangle$ more suitably than θ, i.e. $Pr(Y_1, Y_2, ..., Y_T|\bar{\theta}) > Pr(Y_1, Y_2, ..., Y_T|\theta)$ [82].

b) **Sequence Retriever**: Once the probability estimator determines the converged HMM model θ^*, now, it is job for the *Sequence Retriever* to extract the optimal sequence of hidden events using the sViterbi algorithm [21]. Given a particular observation sequence $\langle Y_1, Y_2, ..., Y_t \rangle$ at time instance t and $Y_t = y_k$, the goal here is to determine the following:

$$\begin{aligned}\omega_t(i) &= \underset{x_1,...,x_{i-1}}{max}\Big\{Pr(X_1 = x_1, ..., X_t = x_i, Y_1, ..., Y_t = y_k|\theta)\Big\} \\ &= \underset{x_1,x_2,...,x_{i-2}}{max}\Big\{\underset{x_{i-1}}{max}\Big\{\omega_{t-1}(i-1)q_{(i-1)(i)}\Big\}\mu_t(y_k)\Big\}, \\ &\qquad\qquad 2 \leq t \leq T,\ 1 \leq i \leq N\end{aligned} \tag{4.13}$$

Hence, $\omega_t(i)$ represents the maximum probability of the occurrence of a particular state sequence $\langle x_1, x_2, ..., x_i\rangle$ at time instance t that corresponds to the aforementioned observation sequence $\langle Y_1, Y_2, ..., Y_t\rangle$.

The equation (4.13) can be solved recursively to determine the highest probability of the occurrence of a complete state sequence $\langle x_1, x_2, ..., x_N\rangle$ for the time instance $2 \leq t \leq T$ given that $\omega_1(i) = \sigma_i\mu_i(Y_1)$. The recursion stops after computing $\omega_T(i)$ such as:

$$\omega_T^* = \underset{1\leq i\leq N}{max}\ \omega_T(i) \tag{4.14}$$

But to obtain the optimal hidden sequence, we must trace the arguments that maximize the equation (4.13) during each recursion. To achieve that, we introduce a variable χ to hold all the traces such as:

$$\chi_t(i) = \underset{1\leq i\leq N}{argmax}\Big\{\omega_{t-1}(i-1)q_{(i-1)(i)}\Big\}, 2 \leq t \leq T, 1 \leq i \leq N \tag{4.15}$$

Note that $\chi_1(i) = 0$ for $t = 1$ because we start tracing the states for the very first time at time instance $t = 2$ once we have at least one previous state.

Once we have $\chi_T(i)$, all we need is backtracking through the traces to discern the optimal hidden sequence such as:

$$\psi_t^* = \chi_{t+1}(\psi_{t+1}^*),\ t = T-1, ..., 2, 1 \tag{4.16}$$

Hence, $\psi_T^*(i) = \chi_T(i)$, and $\Upsilon = \{\psi_1^*, \psi_2^*, ..., \psi_T^*\}$ is the extracted optimal sequence. Note that each $\psi_t^* \in \Upsilon$ represents a true state in X.

4.3.1.3 Crucial Node Detector

After the *sequence retriever* extracts the optimal sequence $\Upsilon = \{\psi_1^*, \psi_2^*, ..., \psi_T^*\}$, the component *crucial node detector* applies Algorithm 1 to detect the crucial events in the attack chain the attacker must compromise to successfully implement the attack. Hence, we term the most frequently triggered events as *crucial events*.

Algorithm 1 takes p different extracted sequences $\Upsilon_1, \Upsilon_2, ..., \Upsilon_p$ and the original sequence $X = \{x_1, x_2, ..., x_N\}$ as inputs. The goal of the algorithm is to determine a pair of states which represent the event occurrences most likely to be triggered during a remote injection attack. Algorithm 1 first determines the *longest common subsequence* S_i between each Υ_i and X. Later, it computes the *SCORE* value for each pair of states in the subsequence using the following formula:

$$\begin{aligned} &SCORE\Big[S_i[j], S_i[j+1]\Big] = \text{number of times a pair} \\ &\big\{S_i[j], S_i[j+1]\big\} \text{ is present in the subsequence} \end{aligned} \tag{4.17}$$

Finally, the algorithm updates the *SCORE* values based on the presence of pairs in all subsequences and retrieves the pairs with the maximum *SCORE* value. It may output a number of pairs of states, such as $\{x_{c_i}, x_{c_j}\}$, where there is a crucial state transition in the state machine from x_{c_i} to x_{c_j}. Our goal is to identify the events (we call them *nodes*) associated with transitions exploited by the attackers to compromise the chain.

An Example

Suppose, there is a sequence of states (corresponding to some triggered events): {*door-opened*, *light-on*, *camera-on*, *fan-on*, *window-opened*}. And after making three

Algorithm 1 Crucial node detection algorithm

Input: $X, \Upsilon_1, \Upsilon_2, ..., \Upsilon_p$

Output: Pairs of true states responding to the most frequently triggered events

$i \leftarrow 1$
while $i \leq p$ **do**
  $S_i \leftarrow$ Longest Common Subsequence between X and Υ_i
  for $j \leftarrow 1$ to $(|S_i| - 1)$ **do**
    $E[i, j] \leftarrow \{S_i[j], S_i[j + 1]\}$
    if $E[i, j]$ not in $SCORE.Keys()$ **then**
      $SCORE[E[i, j]] \leftarrow 1$
    else
      $SCORE[E[i, j]] \leftarrow SCORE[E[i, j]] + 1$
    end if
  end for
end while
return $\underset{E[i,j]}{argmax}\ (SCORE[E[i, j]])$

separate attempts, the sequence retriever returns the following three sequences:

Sequence-1: {door-opened, light-on, light-on, camera-on, fan-on}

Sequence-2: {fan-on, light-on, camera-on, fan-on, window-opened}

Sequence-3: {door-opened, light-on, camera-on, window-opened, fan-on}

Now, if we apply Algorithm 1 on this scenario, we find that the pair {*light-on, camera-on*} obtains the highest score. Consequently, we can conclude that the transition from the state *light-on* to *camera-on* is the most vital one in the state machine, and the nodes associated with those states are the most crucial ones in the chain. IoTMonitor identifies these crucial nodes so that we can perform security hardening to minimize the attacker's chance of compromising an IoT network.

4.4 Evaluation and Results

To evaluate the performance of IoTMonitor, we utilize the PEEVES dataset [16] that records IoT event occurrences from 12 different IoT devices and sensor measure-

ments from 48 deployed sensors to verify those events. We use 24-hours data for our experiment, and our experiment executes on a 16 GB RAM and 4 CPU core system.

4.4.1 Dataset Processing

Our experiment mainly deals with three types of data:

1. Event data (encoded as true states)
2. Sensor measurements (encoded as observations)
3. Timestamps

. We concentrate only on those event occurrences which can be verified by the sensor measurements. Since sensor measurements here capture the physical changes that have happened in the environment due to the event occurrences, they can be used to crosscheck whether a certain event has occurred. We conceptualize the function *sliding window* to determine whether an event is verifiable. The function *sliding window* gives us a fixed time window (in milliseconds), w_i, to check all physical evidences collected by sensors to verify the actual occurrence of an IoT event. When an event occurrence is recorded at time t_i, the sliding window enables us to check whether the physical evidences collected between $t_i - w_i$ and $t_i + w_i$ are enough to verify the occurrence of that event. If the verification is not possible, we just discard the event occurrence from the sequence. In our experiment, we consider 20 such sliding windows with the window size between 105 milliseconds and 200 milliseconds with an increase of 5 milliseconds.

4.4.2 Experiment Setting

At the beginning of our experiment, we choose Gaussian distribution [83] to randomly assign the transition probabilities and initial state probabilities for each true state. On the other hand, we use Dirichlet distribution [84] to assign the emission

probabilities. Since assigning emission probabilities to observation states is both categorical and multinomial distribution tasks, we choose to utilize Dirichlet distribution to initially assign some probabilities against the observation states.

4.4.3 Probability Estimation Time

Probability estimation time represents the time required to estimate the converged transition probability distribution Q and the emission probability distribution E. Figure 4.3(a) presents the estimation time for four different sequences of events of variable lengths (5, 10, 15, and 20) against a range of sliding windows. In the figure we show the average estimation time after 10 executions.

As we can see from Figure 4.3(a), the longest estimation time is < 4 seconds for the sequence length of 20, while in most cases, it is < 0.5 seconds. We iterate our learning algorithm 1000 times and set the convergence threshold to 10^{-6}. When we see a change in probabilities $\leq 10^{-6}$, we assume that those probabilities are converged. We can see that when the window size increases, the estimation time starts to decrease and stabilize. There are a few exceptional cases where the estimation time increases sharply for an increase in window size. For example, when the window size increases from 105 to 110 for the sequence of length 20, we see a sudden spike. We examine the source data and find that this spike is caused by the appearance of two new events that were not present earlier. Since the number of unique events increases and repetition of same events decreases in the sequence, the initial state distribution and transition probabilities are needed to be adjusted which costs adversely to the total estimation time. However, this type of exception is transient, and the graph stabilizes eventually. We do not present the estimation time for the sequences of lengths > 20 in the Figure 4.3(a) since we observe very little change in pattern for those sequences.

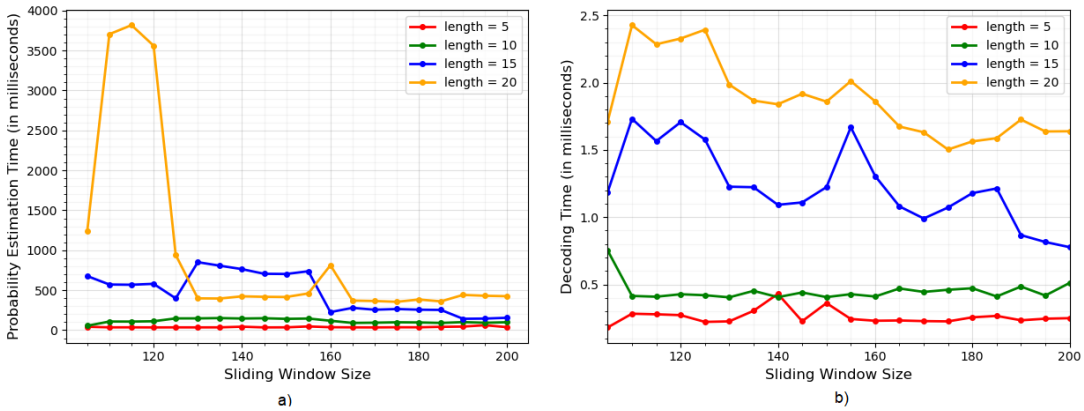

Figure 4.3: a) Probability estimation time with respect to sliding window size and length of the event sequence; b) Decoding time with respect to sliding window size and length of the event sequence

4.4.4 Decoding Time

Decoding time represents the time required to extract the hidden sequence when we have the converged θ^*. Similar to probability estimation time, we take average decoding time after 10 executions. The Figure 4.3(b) presents the decoding time for four different sequences of events with lengths 5, 10, 15, and 20 against a range of sliding windows.

If we look at the graph at Figure 4.3(b), we see that the decoding time decreases when the window size increases. The longest decoding time we get is < 2.5 milliseconds which is very fast for the retrieval of hidden event sequences. Although we see few little temporary spikes for the length 15 after sliding window 150, we still achieve < 2.0 milliseconds as the decoding time.

4.4.5 Computational Overhead

Since our experiment dedicates most of the computation time to estimate the probabilities, we measure *computational overhead* as the total number of iterations of the *forward-backward procedure* required to reach the convergence of transition probabilities and emission probabilities. In Figure 4.4, we present the required total number of iterations (in y-axis) with respect to the ratio between *the total number of unique*

observation states and *the total number of unique true states* (in x-axis). We can see that, the computational overhead increases roughly linearly with the ratio.

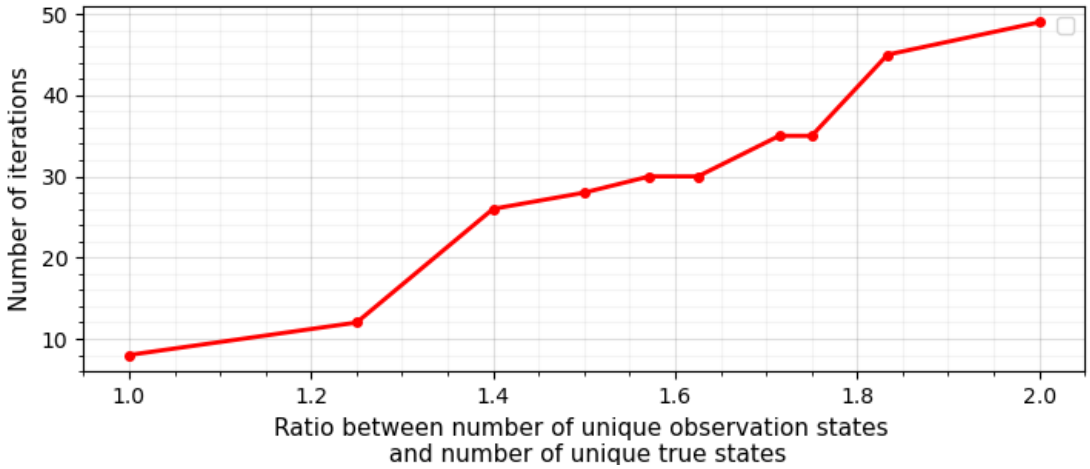

Figure 4.4: Number of iterations to estimate the converged transition probabilities and emission probabilities with respect to the ratio between number of observation states and number of true states

4.4.6 Accuracy Score

To determine how accurately the extracted hidden sequence of events represent the real events, we compute f-score for 29 different sequence of events starting with the length 2 and ending at length 30. We do not consider the sequence with length 1 because it does not offer any uncertainty in terms of transition and emission probability. To calculate the accuracy, we compare the extracted sequence with an original sequence and determine how similar they are in terms of nodes and functional dependencies. We present a heatmap to visually show the correlation among accuracy score, sliding window size and length of the event sequence. In Figure 4.5, the accuracy scores are presented as colors.

As we can see, when the length of event sequence is < 15, the increase in window size after 160 assures a very high accuracy score. We even get the accuracy score of 1.0 in some occasions. There is only one exception for the sequence of length 5. We see a decrease in accuracy score after the window size 105, and that's because we see a

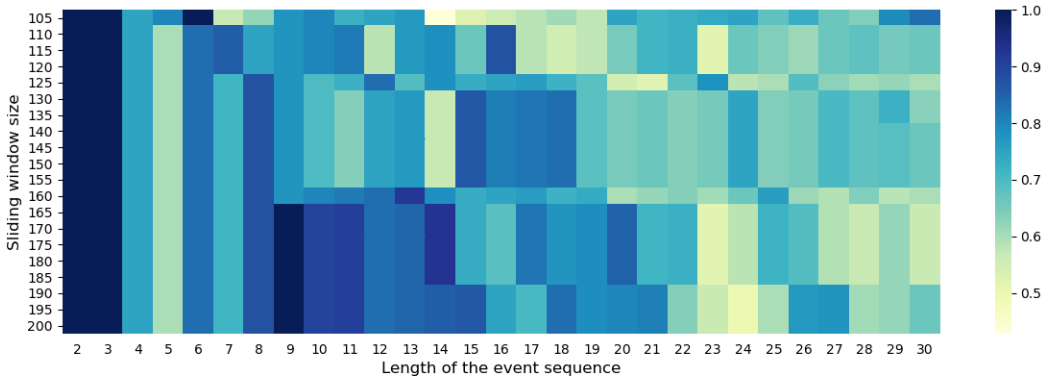

Figure 4.5: Accuracy score vs Sliding window size vs Length of the event sequence

completely new sequence for the window sizes 110 to 200. Similar pattern also arises, although to a less extent, for the sequence of length 7. But it is quite evident that the increase in window size for the smaller lengths ensures higher accuracy score (equals or close to 1.0). When the length increases to a considerable extent, we start to see the impact of sliding windows on the accuracy score diminishing slowly. Since our system emphasizes on the functional dependencies (in terms of transition probability) of the events to extract the hidden sequence, the longer the sequence becomes, the looser are the dependencies.

4.5 Conclusion

In this research work, we propose IoTMonitor that focuses on the extraction of the underlying event sequence using the HMM approach given a set of physical evidences emitted during a trigger-action based attack in an IoT environment. We use the Baum-Welch algorithm [20] to estimate transition and emission probabilities, and the Viterbi algorithm [21] to extract the underlying event sequence. Our experiments show that both probability estimation and sequence extraction operations converge reasonably fast. In terms of accuracy score, IoTMonitor achieves 100% in multiple cases and $\geq$ 90% in a number of cases. We draw a heatmap to visually show the correlation among accuracy score, sliding windows, and length of the event sequences. We also present an algorithm to identify the crucial events in the extracted sequence

which the attackers wish to compromise to implement a trigger-action attack.

CHAPTER 5: IoTWarden: A Deep Reinforcement Learning Based Real-time Defense System to Mitigate Trigger-action IoT Attacks

5.1 Introduction

We discuss in Chapter 1 that trigger-action IoT platforms enable IoT hubs instructing IoT devices to perform predefined actions (e.g., *turning on smart light*) based on specific event conditions reported by other IoT devices (e.g., *the door is unlocked*) [15]. Hence, event conditions act as *triggers*, which activate corresponding actions associated with user-defined rules that are set in a *rule engine*. These rules represent functional dependencies between various IoT event conditions and actions. In a smart home network, these functional dependencies create a chain of interactions among IoT devices to automate network tasks. IoT devices report event conditions to the hub and inform the cyber states of the devices. The hub then verifies those devices' physical states by leveraging physical evidences captured by deployed sensors in the network. The trigger-action functionality also allows the hub to communicate with users through important notifications and alerts about the physical developments (e.g., *when the fire alarm sounds*) in a smart home environment [85].

5.1.1 Research Motivation

We see in Chapters 1 and 2 that the chain of interactions creates security vulnerabilities in trigger-action enabled IoT environments [11]. Attackers can exploit this chain to inject fake event conditions into the network, trigger sensitive actions invalidly, and cause unsafe state transitions in the environment violating rule execution integrity [86]. Attackers can also collect seamless sensitive user data without raising suspicion to the defense system. By simulating the behavior of a valid IoT device,

an attacker can maliciously report a fake event condition to the smart hub to force it to invoke undesired actions in target IoT devices. It is possible for the attacker to perform this attack remotely using ghost devices. This attack is also called *event spoofing attack* since attackers deceive the hub with malicious reporting of spoofed event conditions [15]. By performing remote injection attack, it is possible for an attacker to maliciously invoke safety-critical actions (e.g., *opening the front door when the owner of the smart home is outside*) in target IoT devices in trigger-action IoT platforms [10].

In the literature, researchers mostly tried to address this vulnerability either by implementing offline IoT anomaly detection systems [16] [17] or developing static-analysis based security compliance enforcement systems [13] [9]. The literature lacks security solutions that can provide realtime defense against ongoing remote injection attacks having agile attack strategies.

5.1.2 Our Contributions

In this chapter, we describe *IoTWarden*, a Deep Reinforcement Learning (Deep RL) [23] based defense system that allows a defense agent to model attack behavior based on the impact of attack actions in the IoT environment and obstruct the progression of an ongoing attack in realtime.

We make the following contributions in this research work:

- We propose a Deep RL based real-time defense system, namely *IoTWarden*, that allows defenders to take optimal defense actions at runtime against ongoing trigger-action IoT attacks.
- We implement an LSTM-based [22] Recurrent Neural Network (RNN) to discern optimal attack sequences to help IoTWarden infer attack behavior at runtime.
- We implement Deep Q-Network [23] to obtain optimal defense policy and train the decision process of defenders.

- We conduct extensive experiments and simulation to evaluate the performance of IoTWarden, showing that the adoption of optimal defense policy yields improved security gain with very low computation overhead.

5.2 Problem Statement

Given that an attacker performs a remote injection attack in a trigger-action IoT platform, we aim to answer the following research questions:

1. How to discover attack behavior by observing an attacker's actions considering that the attacker may change the adopted attack strategy based on the defense actions taken in the network?

2. How to determine an optimal defense policy that instructs a defense agent to take effective defense decisions against a remote injection attack at runtime?

3. How to train the decision process of a defense agent ensuring that the overall security gain is maximized?

5.3 Remote Injection Attack

To perform a remote injection attack exploiting functional dependencies in a smart home network, an attacker injects necessary amount of event conditions into the hub, which act as *triggers* for the actions in other devices. We call the injection operations *exploits*, which trick the hub to activate actions in target devices. Note that we interchangeably use the terms *event condition* and *event* in this chapter because they signify the same aspect in our attack scenario.

5.3.1 Attack Definition

We assume that there are $\mathcal{N}$ possible event conditions in the network $C=\{c_i\}, 1\leq i\leq\mathcal{N}$, and the attacker conducts an optimal sequence of $\mathcal{M}$ exploits $\xi=\{e_j\}, 1\leq j\leq\mathcal{M}$ to report a subset of these conditions to the hub to perform a trigger-action attack.

We assume that the attacker performs an exploit $e_t \in \xi$ at timestep t to maliciously report an event condition $c_t \in C$ to the hub and expects the hub to invoke an event condition $c_{t'} \in C$ at another device, which is a target of the attacker. Since complex functional dependencies may exist in the chain of interactions, it is possible that the attacker needs to perform a set of exploits $\{e_t\} \in \xi$ at timestep t to report multiple event conditions $\{c_t\} \in C$ to achieve the attack goal. The attacker considers an IoT device compromised if the following definition 1 applies to it.

Definition 1 (IoT Device Compromise): An IoT device d_g is said to be *compromised* at timestep t if an event condition $c_g \in C$ is triggered at the device d_g due to the malicious reporting of a set of event conditions $\{c_t\} \in C$ to the IoT hub by an attacker through a sequence of *exploits* $\{e_t\} \in \xi$ adopting a strategy $z \in Z$ that dictates the attack operations.

5.3.2 Attack Characterization

To model complex interactions of the attacker with the network and characterize the attack progression over time, we assume that the defense agent embeds event conditions and corresponding exploits to construct a *directed acyclic attack dependency graph* $G = \{C, \xi\}$ [87]. Since attack dependency graph grows non-linearly across time horizon with the increase of the number of nodes, the defense agent considers the inclusion of *monotonicity* [88] property in attack behavior to prevent *state explosion problem* [89] and keep the attack dependency graph reasonably small to perform security analysis. This property enforces a constraint on the attack behavior by limiting the influence of past exploits on the future ones. We assume that the defender constructs this graph using state-of-the-art vulnerability analysis tools, such as TVA tool of [90].

5.3.3 Attack Strategy

Since exploits drive the remote injection attack, it is important for the attacker to follow an effective attack strategy in choosing the optimal exploits. We assume that the attacker crafts a set of attack strategies $Z = \{z_i\}, 1 \leq i \leq \kappa$, where κ is the total number of possible strategies for the attacker. Hence, each $z \in Z$ gives a unique set of exploits to perform during the attack. The attacker changes the adopted attack strategy once the defender blocks certain event conditions making some event reporting untenable. We assume that the attacker precomputes optimal attack paths from each IoT device to the ultimate target device using LSTM-based RNNs exploiting temporal dependencies available in the network. Based on these optimal attack paths, the attacker crafts several exploit sequences, which act as attack strategy $z \in Z$.

5.3.4 Threat Model

To define attacker's capabilities, we adopt the same threat model described in Section 3.2. Besides, we assume that the attacker is intelligent enough to profile the impact of IoT event occurrences over the physical environment and perform opportunistic attacks similar to the attacks explained in [16]. The attacker performs as few exploits as possible so that no noticeable suspicion is raised to the defense system. We assume that the attacker dynamically selects an attack strategy from a limited pool of strategies by observing defender's actions in real-time. The defense system uses physical evidences about the network collected by sensors to verify event occurrences. We assume that the attacker cannot compromise these sensors and is incapable of compromising the hub through which all sorts of communication among IoT devices occur.

5.4 IotWarden Defense System

We design *IoTWarden* as a real-time defense system that infers attack behavior by monitoring the impact of attacker's actions on the network and adopts an optimal

policy to select defense actions, maximizing the total security reward. IoTWarden continuously assesses the security status of the smart home network and takes necessary defense actions to make some exploits infeasible so that the attacker cannot complete a trigger-action IoT attack. We assume that IoTWarden is hosted in the hub, and it enables the hub to block the activation of actions in target IoT devices when it identifies the event injection in the network.

IoTWarden consists of three main components:

- State machine generator
- Policy determiner
- Policy enforcer

Figure 5.1 presents the system architecture of IoTWarden.

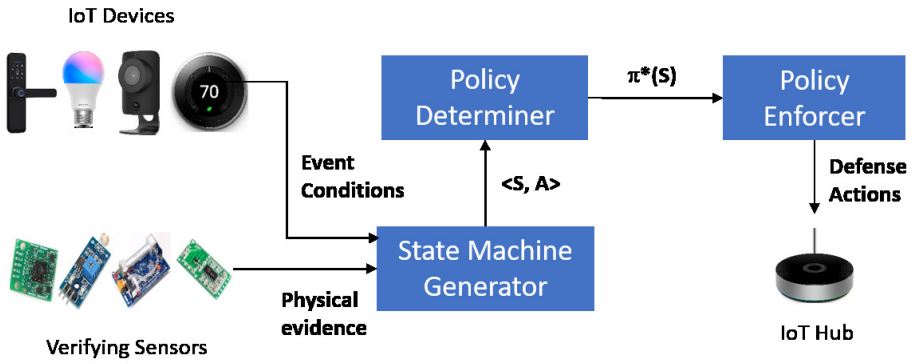

Figure 5.1: IoTWarden System Architecture

5.4.1 State Machine Generator

Considering the IoT network as system environment, this component of IoTWarden creates a *finite state machine* with $\mathcal{N}^{\mathcal{S}}$ unique system states, such as $S = \{s_i\}, 1 \leq i \leq \mathcal{N}^{\mathcal{S}}$ and $\mathcal{N}^{\mathcal{A}}$ unique actions, such as $A = \{a_k\}, 1 \leq k \leq \mathcal{N}^{\mathcal{A}}$. Hence, $s_t \in S$ represents the environment state inferred by the hub at timestep t due to the reporting of $\{c_t\} \in C$, and the action $a_t \in A$ makes the environment transition into another

state s_{t+1} at timestep $t+1$ with a probability $T(s_t, a_t, s_{t+1}) = Pr(s_{t+1}|s_t, a_t)$ yielding a security gain (reward) $R(s_t, a_t, s_{t+1}) = \{r_t\}$. We assume that the action space A contains the following four ($M = 4$) actions:

- a_1*: event injection*
- a_2*: checking device accessibility*
- a_3*: monitoring security status of the network*
- a_4*: blocking triggers*

Hence, a_1 represents an event injection action taken by an attacker to maliciously inject a fake event condition into the network, while a_2 represents the attacker's action to check whether a particular IoT device is accessible. a_3 represents the defense agent's action to monitor the security status of a network device, and a_4 represents the action the defense agent takes to block a certain trigger. Note that IoTWarden chooses $< a_1, a_2 >$ to mimic the behavior of the attacker during the training phase of the system. On the other hand, the defense agent takes the actions $< a_3, a_4 >$ during both the training and the deployment phase.

5.4.1.1 Reward Function

We design the defense agent to be reactive against attack actions. If *event injection* (a_1) actions are taken aggressively, we want the defense agent to take *blocking triggers* (a_4) actions more frequently. We define a parameter *attack proximity factor* p to indicate how close the attack is to the ultimate goal node. We compute p by taking ratio between the number of events already compromised and the total events in the attack chain. When p increases, we want the defense agent to take more *blocking triggers* (a_4) actions.

The ultimate objective here is to prevent the attacker from compromising the goal node. Therefore, we design the reward function to help the defense agent decide

when to allow the attacker to perform injection operations and when to start blocking triggers. The reward function is defined using the following equation (5.1):

$$R(.) = \begin{cases} n_{a_3} r_{a_3} - \frac{p n_{a_1} r_{a_1}}{n_{a_1}+n_{a_2}} - G_r & \text{if } \frac{n_{a_1} p}{n_{a_1}+n_{a_2}} < k \\ \frac{(n_{a_4} r_{a_4})(n_{a_3} r_{a_3})}{n_{a_4}+n_{a_3}} & \\ -max\big(n_{a_2} r_{a_2}, \frac{p n_{a_1} r_{a_1}}{n_{a_1}+n_{a_2}} + G_r\big) & \text{otherwise} \end{cases} \tag{5.1}$$

Hence, n_{a_1}, n_{a_2}, n_{a_3}, and n_{a_4} respectively represent the number of actions a_1, a_2, a_3, and a_4 taken in the environment. On the contrary, r_{a_1}, r_{a_2}, r_{a_3}, and r_{a_4} respectively represent the immediate reward given by the environment for taking the actions a_1, a_2, a_3, and a_4. The injection threshold k is a user defined parameter here. The value of k indicates the tolerance of the defense agent against the injection operations. The parameter G_r represents the reward for the attacker to compromise the ultimate goal node, and the defense agent always tries to make that impossible. Any node compromised beyond goal node yields $G_r = 0$.

As we see in the equation (5.1), the reward function instructs the defense agent to allow the attacker proceed with attack operations until k is reached, i.e., the attacker is allowed to take actions $a1$ and $a2$. However, the attacker needs to consider that n_{a_1} works as a multiplier for p even though the reward r_{a_1} is fixed, i.e., k is quickly reached if too much *event injection* (a_1) actions are taken. The defense agent continuously performs monitoring operations by taking action a_3 and checking whether the injection threshold k is met. Once k is met, the defense agent starts taking *blocking triggers* (a_4) actions aggressively and ensures that the overall reward is optimized considering the negative reward for the attacker due to the actions a_1 and a_2. We design the reward function in the equation (5.1) as a linear function because we want the defense agent to slowly learn the dynamic between attack and defense actions.

5.4.2 Policy Determiner

IoTWarden adopts an optimal policy that dictates how the defense agent chooses actions at runtime. To determine this policy, IoTWarden solves a Markov Decision Process $<S, A, T, R>$ and trains a policy that maximizes the future discounted reward, $R_t = \sum_{t=0}^{t=T} \gamma^t r_t$ at timestep t, where the simulation stops at $t = T$, and γ $(0 \leq \gamma \leq 1)$ is the *discount factor*. The discount factor γ tells the defense agent how much future rewards should be prioritized compared to the immediate reward received due to the just-taken action. If the value of γ is closer to 0, the immediate reward is prioritized. However, if it is closer to 1, the defense agent will try to focus more on the accumulated future rewards than the immediate reward given by the environment.

Given a state $s \in S$, the objective here is to discern a policy $\pi(s)$ that provides optimal state-action value pairs $<s, a>$ from a function $Q^*(s, a)$ (called *Q-function*) yielding the maximum security reward R_t, such as:

$$Q^{\pi^*}(s_t, a_t) = \max_{\pi} E\big[R_t | s_t = s, a_t = a, \pi\big] \tag{5.2}$$

The Q-function obeys the Bellman equation [91], and the equation (5.2) can be rewritten as:

$$Q^{\pi^*}(s_t, a_t) = E\big[r_t + \gamma \max_{a' \in \pi(s_t)} Q^*(s_{t+1}, a' | s_t, a)\big] \tag{5.3}$$

The policy determiner constructs a Deep Q-Network (DQN) [23] to estimate the Q-function and utilizes a neural network function approximator with weights θ_i at the iteration i to compute the temporal difference error δ. As shown in equation (5.5), it uses the *Huber loss* [92] function to minimize this error δ over a batch β of transitions $<s_t, a_t, s_{t+1}, r_t>$ from a replay memory. These transitions represent the defense agent's history of interactions with the environment. The policy determiner

samples these transitions from the replay memory using an *exploration-exploitation trade-off* approach similar to the ϵ-greedy approach discussed in [93]. The following equation (5.4) presents the formula used to compute δ and equation (5.5) shows how this quantity δ is minimized during the training period.

$$\delta = E\big[r_t + \gamma \max_{a' \in \pi(s_t)} Q(s_{t+1}, a'; \theta_{i+1}) - Q(s_t, a; \theta_i)\big] \tag{5.4}$$

$$\mathcal{L} = \frac{1}{|\beta|} \sum_{(s_t, a, s_{t+1}, r_t) \in \beta} \mathcal{L}(\delta) \tag{5.5}$$

where,

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases}$$

Hence, $\mathcal{L}(\delta)$ represents the *Huber loss* function [92], which is a hybrid loss function that takes the advantageous characteristics of both *Mean Squared Error (MSE)* and *Mean Absolute Error (MAE)* loss functions [94]. It is quadratic for small values of δ and linear for large values of δ. It is less sensitive to outliers, and therefore, it reduces outlier's impact on the squared error loss.

When we train the DQN, we ensure that the following action selection method is adopted to balance between exploration and exploitation.

- *Exploration*: an action a is randomly selected from A with the probability of ε,
- *Exploitation*: the optimal action $a_t \in A$ is chosen as a greedy option with the probability of $1 - \varepsilon$.

We formulate ε using the following equation (5.6):

$$\varepsilon = \varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end})\, e^{\frac{-t}{\varepsilon_{decay}}} \tag{5.6}$$

Hence, ε_{start} and ε_{end} respectively denotes the maximum and minimum value of a

range in which the value of ε lies. The parameter ϵ_{decay} denotes the decay factor that gradually diminishes the value of ε, and t denotes the current timestep. As t increases, the value of ε in the equation (5.6) decreases slowly. Based on the latest value of ε, either the *exploration* or the *exploitation* action selection method is adopted. The equation (5.6) is designed in such a way that when the defense agent just starts interacting with the network environment, the agent prefers the *exploration* method since initially, ε yields a value close to 1. But when the learning process is being continued for some time, the value of ε starts decreasing towards 0. In that case, the probability of selecting an action from the replay buffer adopting the *exploitation* method becomes higher. So, the agent chooses an action that has already been taken before so that the ultimate reward can be optimized.

5.4.3 Policy Enforcer

To counter a realtime attack, IoTWarden leverages the trained policy π^* to choose optimal defense actions. At timestep t, IoTWarden receives the state s_t from the environment and utilizes the function approximator to choose the best defense action $a_t \in A$ that maximizes the ultimate reward. Once the policy enforcer takes the chosen action, the environment returns a reward r_t to the defense agent based on the reward function given in equation (5.1). The complete policy enforcement process is illustrated in Figure 5.2.

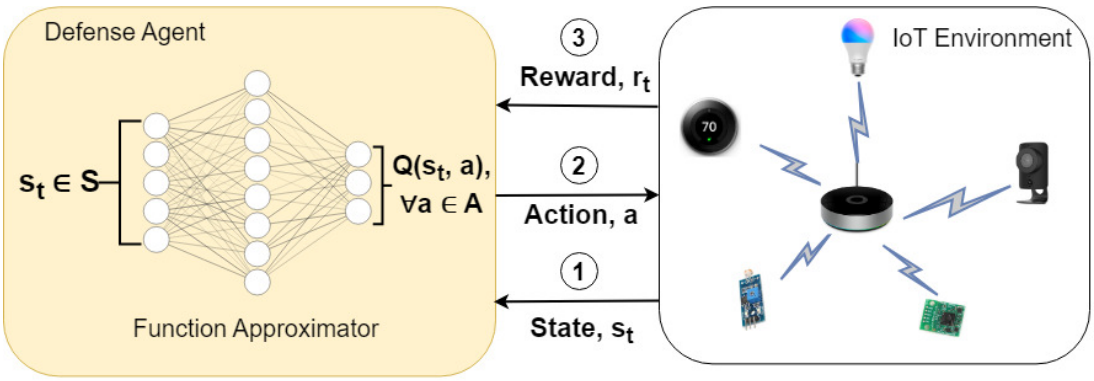

Figure 5.2: Illustration of IoTWarden Policy Enforcement

5.5 Experiment and Simulation

We implement IoTWarden using TensorFlow [95] and conduct experiments simulating a trigger-action attack in a smart home network to evaluate the performance of the system. We utilize the PEEVES [16] dataset to extract the state space needed to be encoded in the simulating environment. This dataset contains event traces collected from 12 different IoT devices and physical evidence measured by 48 different sensors. We utilize 24-hours data in our experiment. We run the experiment on an Apple M1 Pro machine with 16 GB RAM and 8-core GPU.

5.5.1 Determining Optimal Attack Sequences

We use a Hidden Markov Model based approach described in [11] to determine IoT events likely to occur in a trigger-action attack. We call them *crucial* nodes. Later, we implement an LSTM [22] based RNN to create optimal attack sequences exploiting temporal dependencies among IoT events captured in the PEEVES [16] dataset and considering each crucial node as the starting node of a possible attack chain. The RNN architecture contains one embedding layer to convert input event sequences into fixed-length vectors of size 128, two bidirectional LSTM layers with 64 and 32 units respectively, and one fully-connected output layer with 23 classes. Each event sequence given as input to the embedding layer records 227 different event occurrences. Table 5.1 lists the LSTM-based recurrent neural network settings we used in the experiment.

5.5.2 IoT Environment

From the optimal attack sequences, we extract events based on origin, convert them into states, and encode them into the system environment for simulation. We encode 12 unique states and 4 actions in the environment using OpenAI Gym [96]. We also integrate the reward function in the environment so that the defense agent is rewarded accordingly for the actions taken at runtime.

Table 5.1: LSTM-based recurrent neural network settings

Parameter	**Value/Quantity**
Number of Bidirectional LSTM layers	2
1st LSTM layer size	64 units
2nd LSTM layer size	32 units
Embedding layer size	128 units
Maximum sequence length	227
Number of classes in output layer	23 classes
Activation function for output layer	Softmax
Loss function	Categorical cross-entropy

5.5.3 Function Approximator

Given a state in the environment, we use a neural network *function approximator* to estimate reward for each available action and output the optimal action yielding the highest reward. The neural network we use has 2 hidden layers with 64 and 32 neurons, while the input and output layers include 128 and 4 units, respectively. We use the *ReLU* [97] activation function for the input and hidden layers. For the output layer, we use the *Linear* activation function. We use *Huber loss* [92] as the loss function in this neural network. The complete settings that we use for the function approximator are listed in Table 5.2.

Table 5.2: Function approximator settings

Parameter	**Quantity**
Number of Hidden layers	2
1st Hidden layer size	64 units
2nd Hidden layer size	32 units
Input Layer size	128 units
Output Layer size	4 units
Activation function for output layer	Linear
Activation function input and hidden layers	ReLu
Loss function	Huber loss

5.5.4 Deep Q-Network

The DQN we use to estimate Q-function has the hyperparameter settings listed in Table 5.3. We train the network for 250 episodes and use *Adam* [98] optimizer with the learning rate, $\alpha = 1e^{-4}$. During our training, we update a policy network at a constant rate, $\tau = 20$. To record the defense agent's interaction with the environment, we use a replay buffer of size $50,000$ and sample 16 interactions at a time when the optimal action is chosen.

Table 5.3: Hyperparameter settings for Deep Q-network

Parameter	**Quantity**
Total episodes	250
Number of epochs per episode	100
Optimizer	Adam
Minibatch size, β	16
Discount factor, γ	0.95
Learning rate, α	$1e^{-4}$
$(\epsilon_{start}, \epsilon_{end}, \epsilon_{decay})$	(1.0, 0.1, 0.99999)
Target network update frequency, τ	20 episodes
Replay buffer size	50,000

5.6 Performance Evaluation

5.6.1 Optimal Attack Sequence

Before training the LSTM-based RNN to extract optimal attack sequences, we split the PEEVES dataset into training and validation sets with 80-20 ratio. In Figure 5.3, we see the training and validation accuracy of the model over 100 epochs, while the training and validation loss are depicted in Figure 5.4. We see that both training and validation accuracy becomes > 0.99 after $epoch = 9$. Even though we see some unexpected decline in several occasions, for example, at $epoch = 44$, we still achieve > 0.90 accuracy all the time. On the other hand, both the training and validation loss becomes ≈ 0.02 most of the time after $epoch = 15$. Despite a few irregular loss at $epoch = 23, 44, 69$, we can conclude that after $epoch = 9$, the training and validation

loss always stays <0.10, and they become very close to 0 a number of times.

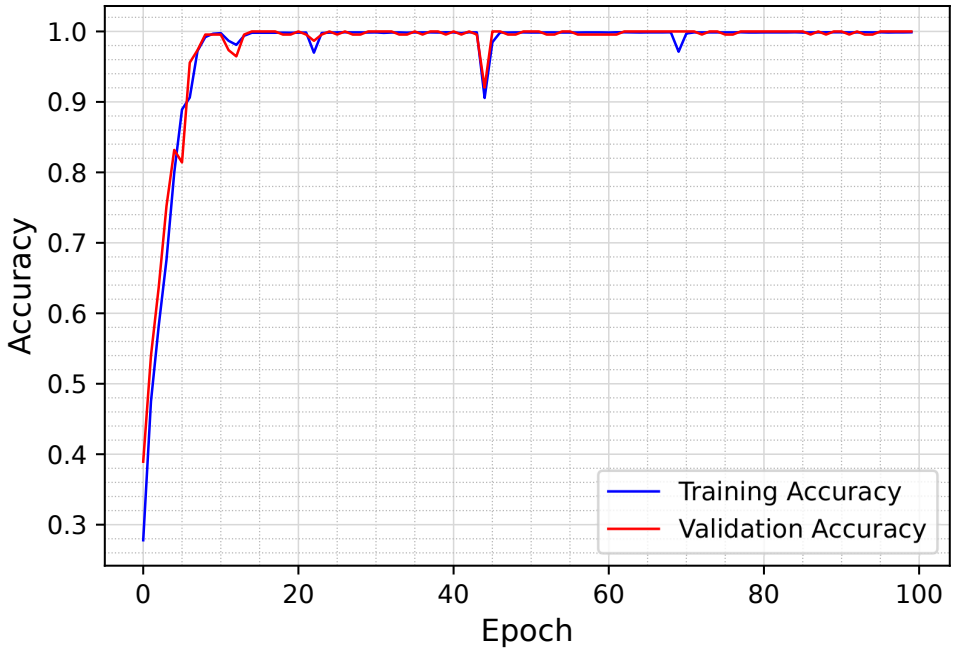

Figure 5.3: Training and validation accuracy over epochs

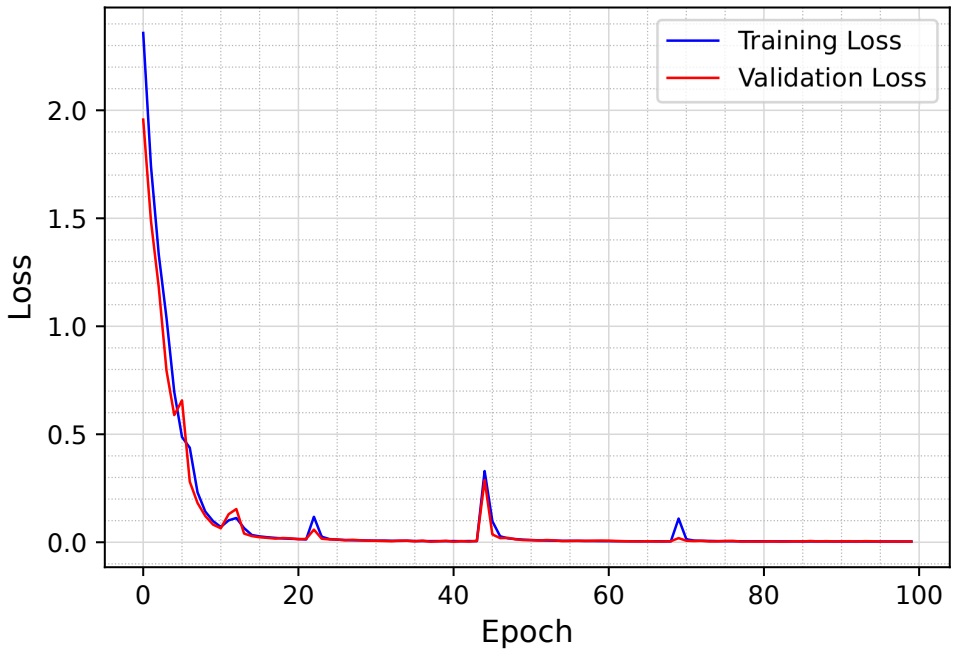

Figure 5.4: Training and validation loss over epochs

5.6.2 Rewards

In our simulation, the defense agent is trained for 250 episodes, interacting with the environment for 100 epochs in each episode. Each interaction yields a discrete reward for the defense agent, and the defense agent accumulates all the rewards received over 100 epochs together to compute the total reward it achieves in a single episode. Figure 5.5 shows the total reward earned in our simulation over the entire 250 episodes. We see that the total reward the defense agent receives for the first few episodes doesn't follow a stable pattern, and that's because instead of following a fixed optimal policy from the start, it explores all possible scenarios to determine the optimal policy that guarantees the maximization of the total reward at the end. Once the agent learns the optimal policy, the total reward it receives over the episodes follows a pretty stable pattern, as shown in Figure 5.5 after ≈ 70 episodes.

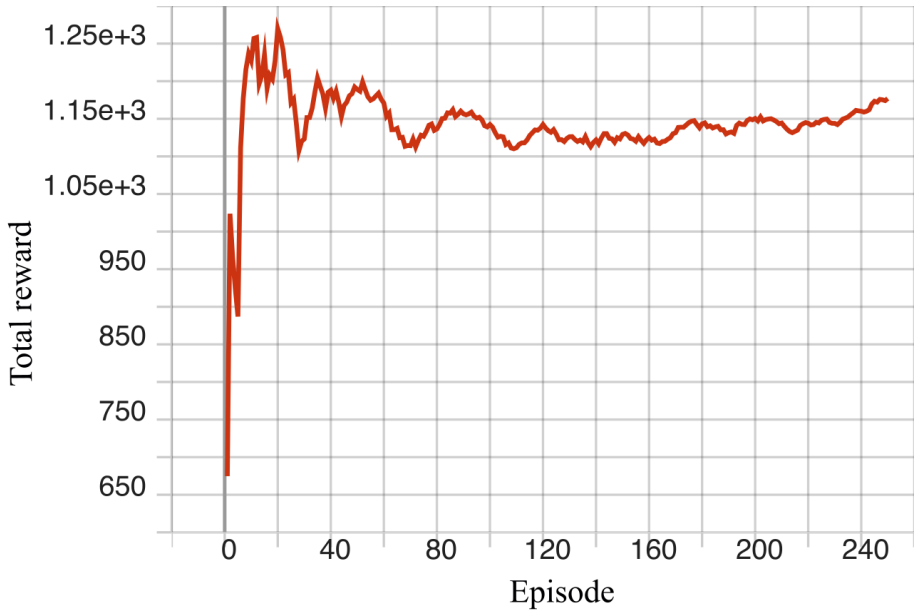

Figure 5.5: Reward over episodes

5.6.3 Computation Overhead

In our simulation, the defense agent dedicates most of its computation time to determine optimal state-action value pairs, sample experiences from the replay buffer, and train the policy network. Therefore, we compute the total time required in each episode to perform all these tasks and call it *time overhead*. Figure 5.6 shows how time overhead changes over the episodes. We can see that after 40 episodes, the overhead reaches stability, close to 2.85 seconds.

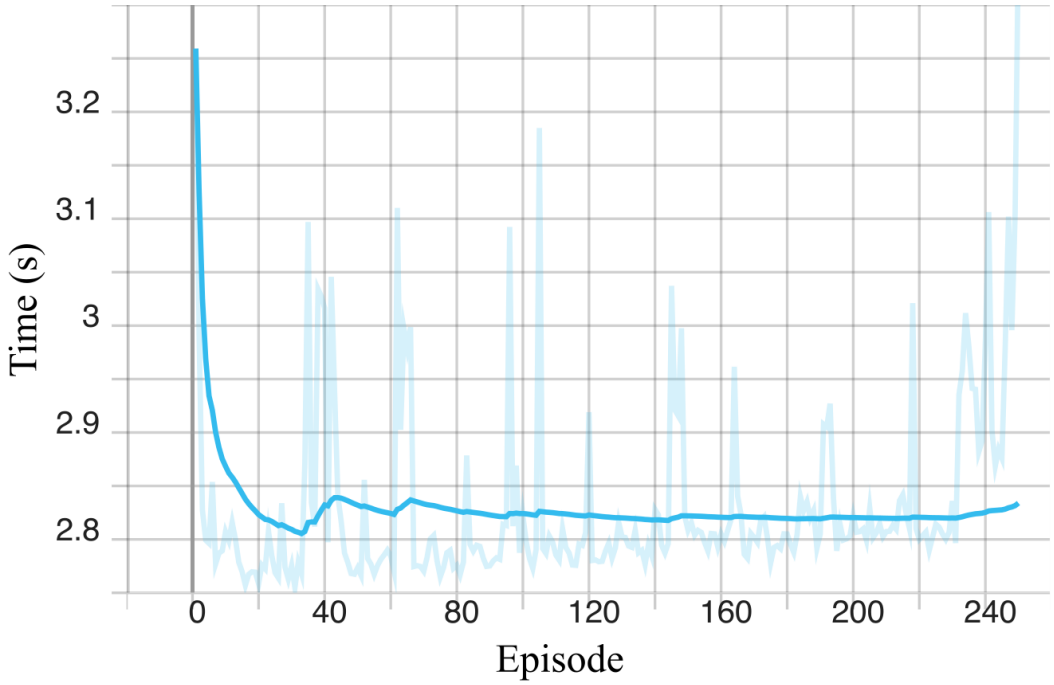

Figure 5.6: Time overhead over episodes

5.6.4 Attack-Defense Dynamic

We design the defense agent to be reactive against the attack actions. If *event injection* (a_1) actions are taken aggressively, the defense agent chooses to take *blocking triggers* (a_4) actions more often. As we see in Figure 5.7, the defense agent initially becomes very aggressive, but it quickly starts to learn the attack-defense dynamic. Since *blocking triggers* (a_4) action negatively impacts the availability of the network

devices, the defense agent avoids taking redundant *blocking triggers* (a_4) actions. Figure 5.7 shows that the number of *blocking triggers* (a_4) actions taken by the defense agent is always smaller than the number of *event injection* (a_1) actions taken after a certain number of episodes ($\approx$ 30 episodes). It is possible that the attacker needs to inject multiple fake events to compromise a certain device, especially if any trigger operation to that particular device has been blocked earlier by the defense agent. Therefore, the objective of a trained defense agent is to take less *blocking triggers* (a_4) actions compared to the *event injection* (a_1) actions taken, which is clearly evident from Figure 5.7.

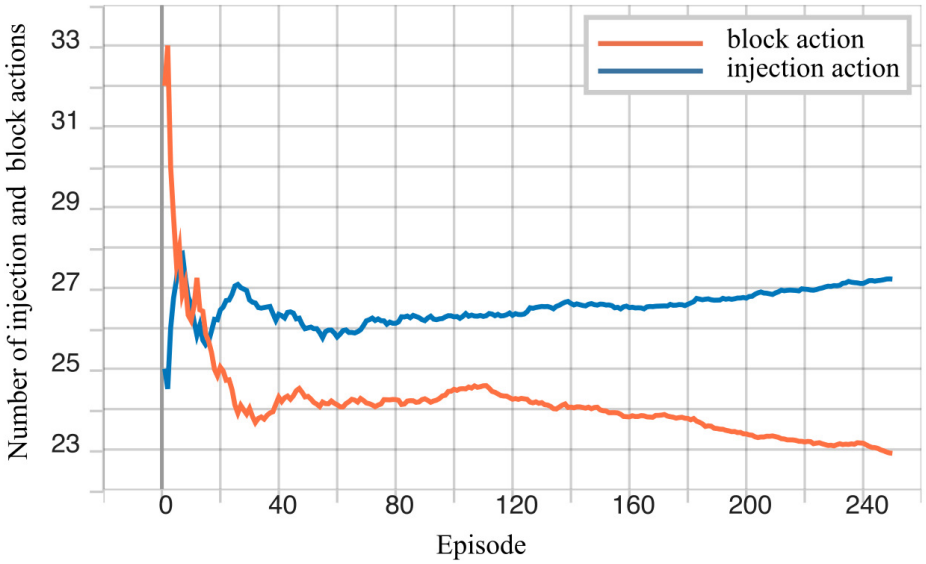

Figure 5.7: Number of injection and block actions over episodes

5.6.5 Impact of Injection Threshold

As we see in equation (5.1), the *injection threshold*, parameterized as k, dictates the selection of optimal defense actions. To show the impact of k on the reward function, we compute the *average episodic reward*, $\bar{R}_t = \frac{\sum_{t=1}^{N} R_t}{N}$, where $N = 100$ epochs, and R_t represents the total reward achieved in a single epoch. In Figure 5.8, we show how the injection threshold k ranging between $[0, 1]$ impacts the average episodic reward

achieved by the defense agent. We see that after $k = 0.25$, the impact of injection threshold on the reward function is quite constant, i.e., even though a greater k allows an attacker to perform more *event injection* operations, the defense agent still achieves a certain level of reward and effectively secure network nodes. In conclusion, once the defense agent determines the optimal defense policy, the increased aggressiveness in attack behavior barely changes the security status of the network.

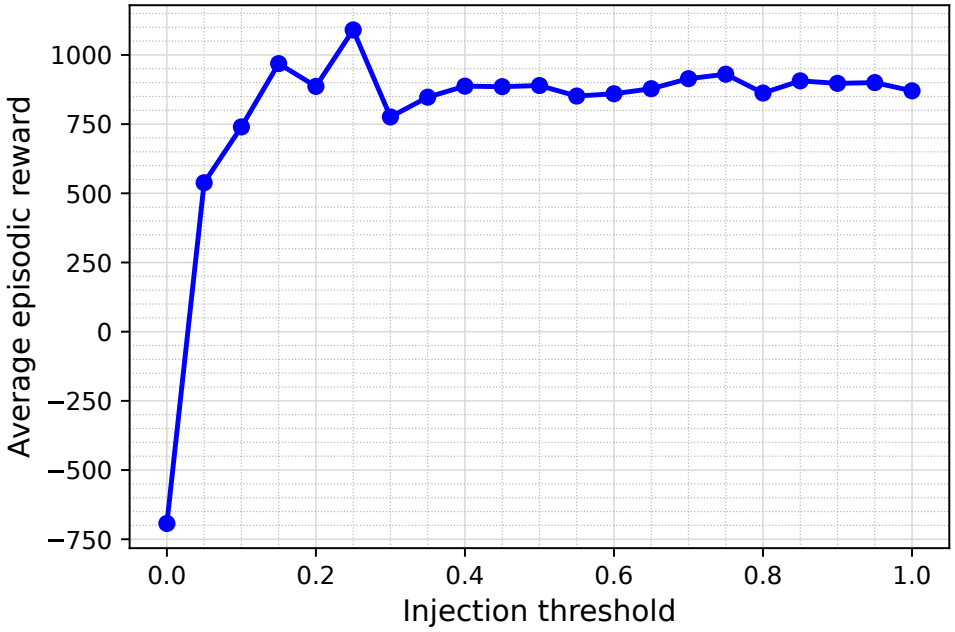

Figure 5.8: Average episodic return over injection thresholds

5.7 Conclusion

In this chapter, we propose a real-time defense system named *IoTWarden* that infers attack behaviors upon an IoT network and counters a trigger-action attack by following an optimal action policy. We implement a neural network function approximator to select optimal action at each environment state by maximizing the security gain and train the defense policy using a Deep Q-Network. We implement the system using TensorFlow and conduct extensive simulation to evaluate the performance of the system. The experiment results show that the defense agent is capable of

achieving stable security rewards with very low computation overhead by following an optimal defense policy under different aggressiveness levels of the injection operations conducted by an attacker. As our approach focuses on real time detection, it can work in parallel with the static analysis-based security measures.

CHAPTER 6: IoTHaven: An Online Defense System to Mitigate Remote Injection Attacks in Partially Observable Trigger-action IoT Platforms

6.1 Introduction

As we discuss in Section 5.3, an attacker can inject event conditions into a trigger-action IoT network through some exploits and implement a remote injection attack leveraging functional dependencies between event conditions and actions. It is possible for the attacker to force the IoT hub to invoke invalid actions in target IoT devices. IoTWarden, discussed in Section 5.4, provides real-time defense against such remote injection attacks, but IoTWarden must have complete visibility over network states to function properly. In this chapter, we discuss how online defense systems can be developed to ensure that the defense agent takes effective defense decisions to thwart attack progression without having full visibility over network states.

6.1.1 Research Motivation

As discussed earlier, the existing literature mostly offers static-analysis based security policy compliance enforcement systems [9] [13] or offline IoT anomaly detection systems [16] [17] to defend against remote injection attacks. We propose IoTWarden [19] as a real-time defense system against ongoing remote injection attacks, but it is not designed for partially observable platforms. Therefore, the literature needs online defense solutions that can function effectively with the partial view of the IoT network states.

6.1.2 Our Contributions

In this research work, we present a real-time defense system, namely *IoTHaven*, to counter remote injection attacks at runtime in partially observable IoT networks.

The main contributions of this chapter are listed below:

- We propose IoTHaven, which observes partial information about the network states yet discerns an optimal defense policy to obstruct the progression of a remote injection attack at runtime.
- We train an LSTM-based function approximator using Deep Reinforcement Learning to determine the optimal defense action given an observation at each timestep.
- We design a novel reward function for a defense agent that dictates the agent's decision process.
- We conduct extensive experiments to evaluate the performance of IoTHaven, showing that the optimal defense policy discerned under uncertainties of the network states yields improved security gain with a low computation overhead.

6.2 Problem Statement

Assuming that the defense agent only has partial visibility over network states, our goal is to answer the following questions:

1. How to model the decision process of a defense agent based on a stream of observations to take optimal defense decisions at runtime to counter ongoing remote injection attacks?
2. How to discern an optimal defense policy and train the decision process of the defense agent maximizing the overall security gain?

6.3 IoTHaven Defense System

We design *IoTHaven* as an online defense system that takes sensor data (we call *observations*) emitted due to the occurrence of IoT events as input and outputs an

optimal sequence of defense actions without observing the actual event occurrences in the network. At a certain timestep, it senses the latest observations and determines the optimal action considering the history of observations and already taken defense actions.

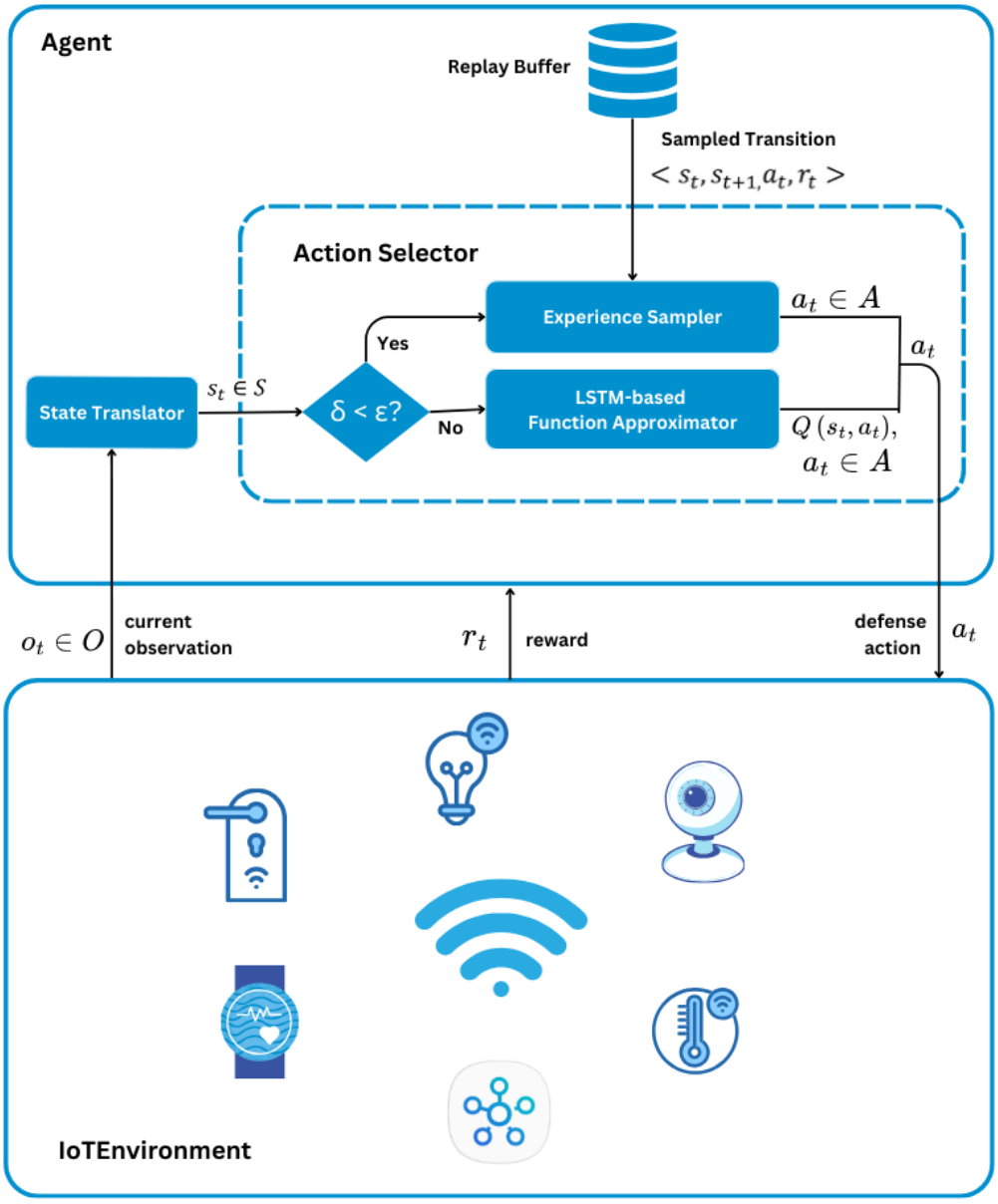

Figure 6.1: IoTHaven System Architecture

6.3.1 System Design

We consider the following assumptions during the system design:

- The attack starts from an arbitrary node in the chain of interactions, and
- Each block operation performed by the defense agent ends up blocking multiple exploits,
- The defense agent cannot see the actual event conditions, rather relies on the observations sensed from the environment.

We assume that the defense agent constructs a *directed acyclic attack dependency graph* $G = \{C, \xi\}$ to model the attacker's interactions with the network and quantify the attack progression over time. The defense agent enforces the *monotonicity* [88] property in the attack behavior to avoid *state explosion problem* [89] and keep the attack dependency graph reasonably small to perform security analysis using tools, such as TVA [90].

It is possible for the attacker to perform reconnaissance operations in the network and determine the most vulnerable nodes in the attack chain. The attacker can preselect the devices to mimic and strategically select the exploits to perform in the network. Since the event conditions are not fully visible to the defense agent, we assume that the defense agent cannot see what exploits are performed to make state transitions in the environment and what the true environment states are. Therefore, we choose *Partially Observable Markov Decision Process (POMDP)* as a mechanism to dictate the decision process of the defense agent.

6.3.2 System Architecture

Figure 6.1 presents the system architecture of IoTHaven. It contains two main components:

1. A system environment, namely *IoTEnvironment*, and

2. A defense agent

We discuss each component and its further sub-components below.

6.3.2.1 System Environment

Similar to IoTWarden, IoTHaven considers the IoT network as the system environment, where different environment states and attack/defense actions are encoded into a state machine. The state machine includes N unique environment states, such as $S = \{s_i\}, 1 \leq i \leq N$ and M unique actions, such as $A = \{a_j\}, 1 \leq j \leq M$. Hence, $s_t \in S$ represents the security state the environment transitions into due to the reporting of the event condition $c_t \in C$ at timestep t. Given that an action $a_t \in A$ is taken at timestep t, the environment further transitions into the state $s_{t+1} \in S$ with a transition probability $T(s_t, a_t, s_{t+1}) = Pr(s_{t+1}|s_t, a_t)$. When the environment is in the state s_{t+1}, we assume that the environment emits an observation $o_{t+1} \in O$ with an emission probability $\mu(s_{t+1}, a_t, o_{t+1}) = Pr(o_{t+1}|s_{t+1}, a_t)$, where the observation space is defined as $O = \{o_k\}, 1 \leq k \leq T$, and T is the total number timesteps. Since state transitions are conditioned to attack/defense actions, a reward function, such as $R(s_t, a_t, s_{t+1}) = \{r_t\}, 1 \leq t \leq T$, must be embedded into the state machine to incentivize the agent's actions.

6.3.2.2 Defense Agent

At a certain timestep t, the main task of the defense agent in IoTHaven is to sense the current observation $o_t \in O$ emitted from the environment and take the defense action $a_t \in A$ following an optimal defense policy. Given the configuration of the system environment, the agent tries to solve a POMDP $< S, A, O, T, \mu, R, \gamma >$ to discern the optimal defense policy. Hence, $\gamma \in [0, 1)$ is a discount factor that tells the agent how much future rewards are prioritized compared to the immediate reward given by the environment due to the just-taken action. If the value of γ is closer to 1, the defense agent prioritizes future rewards more than the immediate possible

rewards.

The ultimate objective of the defense agent is to determine an optimal policy π^* that generates a sequence of actions $a_t \in A$ by maximizing the expected discounted reward:

$$E\Big[\sum_{t=0}^{T} \gamma^t r_t\Big] \tag{6.1}$$

a) Reward Function: We assume that IoTHaven encodes the following equation (6.2) as the reward function for the defense agent in the system environment:

$$r_t = \omega_r r_s(o_t) + (1 - \omega_r)\Big[\frac{r_u(u_t)}{e^{-n_b}}\Big] - \rho_t \tag{6.2}$$

Hence, r_s represents the reward the agent receives for sensing the current observation o_t, r_u represents the reward for taking a block operation u_t, and ρ_t represents the cost the agent pays at timestep t. The quantity $\omega_r \in [0, 1]$ is a user-defined weighting term that allows the agent to specify which action between *sense observations* and *block triggers* is preferred. When $\omega_r = 1$, the agent only considers the sensing action, and when $\omega_r = 0$, the agent only considers the blocking action. The parameter n_b in equation (6.2) denotes the number of block actions taken so far by the defense agent.

The cost ρ_t in equation (6.2) is defined using the following equation (6.3):

$$\rho_t = \omega_\rho \rho_s(s_t) + (1 - \omega_\rho)\Big[\lambda^{n_b} \rho_u(u_t)\Big] + \rho_G \tag{6.3}$$

where, ρ_s represents the security cost for the environment being in the state s_t inferred by the defense agent from the observation o_t and ρ_u is the cost of taking the block action u_t. ρ_G is the cost the defense agent pays when the attacker is successful in compromising the goal node. Similar to ω_r, the quantity $\omega_\rho \in [0, 1]$ is a user-defined weighting term for the agent to specify which cost is more preferred between ρ_s and ρ_u.

We call the quantity $\lambda \in (0, 1]$ in equation (6.3) the *availability preference factor* that tells the defense agent the sensitivity of devices' availability. A larger λ makes block actions less attractive to the defense agent since a block action effectively makes an IoT device unavailable. Similar to equation (6.2), the parameter n_b in equation (6.3) denotes the number of block actions taken so far. The increase in the value of this parameter adversely impacts the defense agent's reward since we assume that the defense agent always tries to minimize the total cost.

b) State Translator: Since IoTHaven has partial visibility over environment states, it requires a *state translator* to estimate the *belief* in the environment state $s_t \in S$ every time it senses an observation $o_t \in O$. The belief is defined as the probability of the environment being in a particular state. The *state translator* computes $b(s_t)$ as the belief in the environment state s_t using the following formula:

$$\begin{aligned} b(s_t) &= Pr\Big(s_t | o_t, a_{t-1}, b(s_{t-1})\Big) \\ &= \frac{\mu(s_t, a_{t-1}, o_t) \sum_{s_{t-1} \in S} \Big[T(s_{t-1}, a_{t-1}, s_t) b(s_{t-1})\Big]}{\eta} \end{aligned} \tag{6.4}$$

Hence, η is a normalizing constant that can be defined as:

$$\begin{aligned} \eta &= Pr\Big(o_t | b(s_{t-1}), a_{t-1}\Big) \\ &= \sum_{s_t \in S} \left[\mu(s_t, a_{t-1}, o_t) \sum_{s_{t-1} \in S} \Big[T(s_{t-1}, a_{t-1}, s_t) b(s_{t-1})\Big]\right] \end{aligned} \tag{6.5}$$

IoTHaven leverages the belief point $b(s_t) = b_t$ to optimize a value function $V^{\pi}(.)$. The value function is defined as a *Bellman optimality equation* [91], such as:

$$V^{\pi}(b_t) = \left[p(b_t, a_t) + \gamma \sum_{b_{t+1}} \Big[\tau(b_t, a_t, b_{t+1}) V^{\pi}(b_{t+1})\Big]\right] \tag{6.6}$$

where,

$$p(b_t, a_t) = \sum_{s_t \in S} b_t R(s_t, a_t, s_{t+1}) = \sum_{s_t \in S} b_t r_t \tag{6.7}$$

and

$$\begin{aligned} &\tau\Big(b_t, a_t, b_{t+1}\Big) \\ &= \sum_{o_{t+1} \in O} Pr\Big(o_{t+1} | b_t, a\Big) \\ &= \sum_{o_{t+1} \in O} \sum_{s_{t+1} \in S} \Big[\mu(s_{t+1}, a_t, o_{t+1}) \sum_{s_t \in S} b_t * T(s_t, a_t, s_{t+1})\Big] \end{aligned} \tag{6.8}$$

To discern the optimal defense policy π^*, IoTHaven optimizes the value function utilizing the following equation (6.9):

$$\pi^* = argmax_{a_t}\Big[\ V^{\pi^*}(b_t)\Big] \tag{6.9}$$

c) Function Approximator: To determine optimal policy, IoTHaven optimizes the value function described in equation (6.9) using a Deep Recurrent Q-Network (DRQN) [25] function approximator. It utilizes deep reinforcement learning to train the function approximator. At each timestep t, the function approximator computes the temporal difference error Δ, as shown in equation (6.10).

$$\Delta = E\big[p(b_t, a_t) + \gamma \max_{a_t \in \pi(b_t)} V^{\pi}(b_{t+1}; \theta_t) - V^{\pi}(b_t; \theta_t)\big] \tag{6.10}$$

Hence, $V^{\pi}(.)$ is parameterized with the weights of the RNN used in the function approximator, θ_t.

The goal here is to minimize the loss function defined in the equation (6.11) over a batch β of transitions $< s_t, a_t, s_{t+1}, r_t >$ stored in a memory called *replay buffer*.

$$\mathcal{L} = \frac{1}{|\beta|} \sum_{(s_t, a_t, s_{t+1}, r_t) \in \beta} \mathcal{L}(\Delta) \tag{6.11}$$

where,

$$\mathcal{L}(\Delta) = \begin{cases} \frac{1}{2}\Delta^2 & \text{if } |\Delta| \leq 1 \\ |\Delta| - \frac{1}{2} & \text{otherwise} \end{cases}$$

d) Experience Sampler: To stabilize the learning procedure, IoTHaven periodically stores $< s_t, a_t, s_{t+1}, r_t >$ transitions in the replay buffer, which represent the agent's experiences with the environment. Later, following the *exploration-exploitation trade-off* approach discussed in [19], the *experience sampler* samples a fixed amount of transitions from the replay buffer into minibatch β. Based on a probability σ, IoTHaven either samples transitions to learn optimal action from experiences or utilizes the function approximator to estimate the optimal action, yielding the maximum reward. To decide how the action should be selected for a particular timestep t, IoTHaven computes ϵ using the equation (6.12) and compares its value with σ.

$$\varepsilon = \varepsilon_{end} + (\varepsilon_{start} - \varepsilon_{end})\, e^{\frac{-t}{\varepsilon_{decay}}} \tag{6.12}$$

Hence, ε_{start} and ε_{end} respectively denotes the maximum and minimum value of a range in which the value of ε lies. The parameter ε_{decay} denotes the decay factor that gradually diminishes the value of ε. When t increases, the value of ε in the equation (6.12) slowly starts decreasing.

6.4 Experiment and Simulation

We utilize TensorFlow [95] to implement IoTHaven and conduct experiments on a simulated smart home environment. We extract state space from the PEEVES [16] dataset that contains event traces collected from 12 distinct IoT devices and sensor measurements captured by 48 sensors. Later, we encode the state space in the simulating environment to create a state machine upon which all attack and defense actions are performed. We utilize OpenAI Gym [96] to encode 12 unique states and 4 actions in the environment. We consider a stream of sensor measurements as observations that maps to system states with some probabilities. We assume that each observation is emitted at each timestep due to the occurrence of an IoT event. To incentivize the defense action, we also incorporate the reward function in the environment. We run our experiments on an Apple M1 Pro machine with 16GB RAM and 8-core GPU.

6.4.1 Function Appproximator

We utilize an LSTM-based Deep Recurrent Q-Network (DRQN) as the *function approximator* to estimate action-specific rewards and devise the optimal defense policy, maximizing the security gain. At a given timestep, this function approximator takes a system state inferred from the latest observations as the input and provides the optimal defense action as the output. The settings we use to design this function approximator is listed in Table 6.1.

6.4.2 Training DRQN

We train the DRQN for 500 episodes and use Adam [98] optimizer with the learning rate, $\alpha = 1e^{-3}$. To ensure stability in the learning procedure, we periodically update a policy network at a constant rate, $\tau = 20$. We record the agent's experience of interacting with the environment in a replay buffer of size $60,000$ and sample 16 experiences at a time as a minibatch based on the exploration-exploitation constant

Table 6.1: Deep Recurrent Q-Network (DRQN) settings

Parameter	**Value/Quantity**
Number of LSTM layers	2
1st LSTM layer size	48 units
2nd LSTM layer size	32 units
Number of nodes in input layer	12
Number of nodes in output layer	4
Activation function for input and LSTM layers	Relu
Activation function for output layer	Softmax
Optimizer	Adam [98]
Loss function	Huber loss [92]

ϵ. The complete hyperparameter settings we use for the training procedure are listed in Table 6.2.

Table 6.2: Hyperparameter settings used for training

Parameter	**Quantity**
Total episodes	500
Number of epochs per episode	50
Minibatch size, β	16
Discount factor, γ	0.95
Learning rate, α	$1e^{-3}$
$(\varepsilon_{start}, \varepsilon_{end}, \varepsilon_{decay})$	(1.0, 0.1, 0.99999)
Target network update frequency, τ	20 episodes
Replay buffer size	60,000

6.5 Performance Evaluation

We run the simulation for 500 episodes with 50 epochs per episode and evaluate the performance of IoTHaven in terms of the following three metrics: 1) total reward over episodes, 2) time overhead over episodes, and 3) the number of injection & block actions over episodes.

6.5.1 Rewards

In our simulation, every time the defense agent interacts with the environment through an action, it receives a discrete reward from the environment. The defense agent accumulates all the rewards received over 50 epochs to compute the total reward

received in a single episode. Figure 6.2 shows the defense agent's total rewards over all 500 episodes. We can see a stable increasing pattern in reward after ≈ 60 episodes. Since the defense agent seeks to discern an optimal policy in the earlier episodes of the simulation, the agent explores all possible action scenarios just to learn the optimal policy and sometimes ends up making block operations aggressively that negatively impact the reward function. However, once the optimal defense policy is learnt, such as ≈ 60 episodes later in Figure 6.2, the agent takes actions at each timestep following a fixed policy that maximizes the total reward received at each episode.

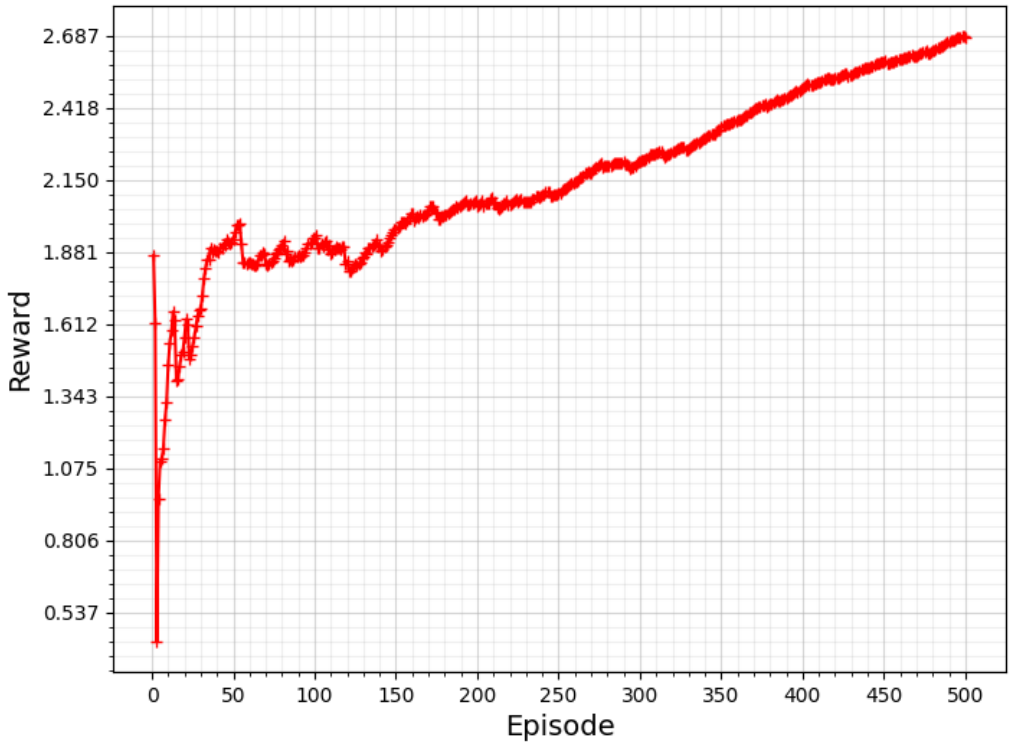

Figure 6.2: Reward over episodes

6.5.2 Time Overhead

In our simulation, the defense agent spends most of the computation time training the LSTM-based DRQN function approximator to determine the optimal action at

each timestep that maximizes the ultimate reward. It also spends some computation time storing interaction experiences in the replay buffer and sampling the stored experiences to help determine an action that stabilizes the learning procedure. We compute the time the agent takes (in seconds) to complete these tasks and call it *time overhead*. Since LSTM nodes of the DRQN utilize *cell states* [99] to retain a history of observations to a certain extent and leverage the past observations to accurately estimate the defense actions at each timestep, the time overhead is an important metric to consider to show the effectiveness of IoTHaven. The *cell states* in LSTM architecture are mainly used to capture and store long-term functional dependencies in sequential data. We see in Figure 6.3 that the maximum time required for IoTHaven to defend against a dynamic injection attack adopting an optimal defense policy is 12.41 seconds. This time overhead is greater than the time overhead we get in IoTWarden because DRQN now retains and process a history of observations before making a defense decision at each timestep. This overhead could be improved further if the agent was trained in a cloud.

6.5.3 Attack-Defense Dynamic

To show the attack-defense dynamic, we consider the reactiveness of the defense agent against the increased aggressiveness of the attacker. If the attacker aggressively makes injection operations in the network, we want the defense agent to increase the number of block operations decidedly. As we see in Figure 6.4, the defense agent matches the aggressiveness of the attacker and increases or decreases the number of block operations to take based on the aggressiveness of the attacker. The agent doesn't take unnecessary block operations since it negatively impacts the availability of the network devices.

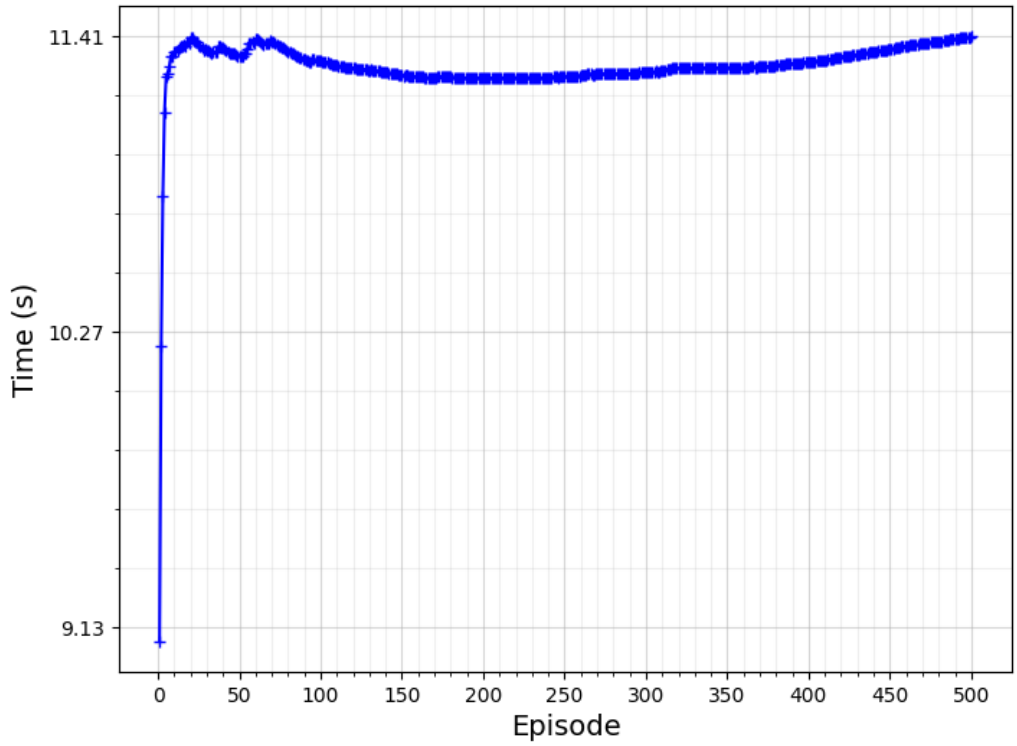

Figure 6.3: Time overhead over episodes

6.6 Conclusion

This chapter proposes IoTHaven, an online defense system to mitigate remote injection attacks in partially observable trigger-action IoT platforms. It discerns an optimal defense policy leveraging a sequence of observations representing physical evidence captured by sensors and takes optimal action at each timestep, maximizing the total security gain. We integrated a novel reward function in the system environment to dictate the decision process of the defense agent. IoTHaven obtains the optimal defense policy by training an LSTM-based Deep Recurrent Q-Network (DRQN) function approximator. We implemented IoTHaven using TensorFlow and conducted extensive experiments on a simulated smart home environment to evaluate the system's performance. The experimental results show that the defense system

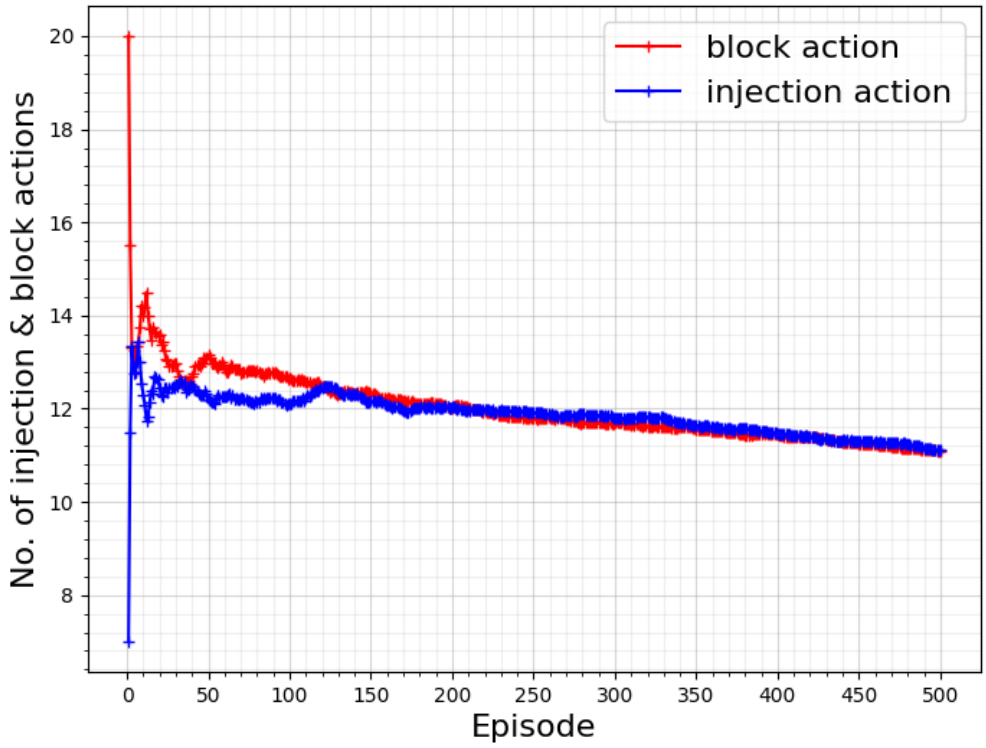

Figure 6.4: Number of injection & block actions over episodes

effectively takes optimal actions under the uncertainty of the actual system states, yielding stable security rewards with low computation overhead. Since IoTHaven doesn't require full visibility over network states, it is well-suited for heterogeneous and scalable trigger-action IoT platforms.

CHAPTER 7: Future Works

Since it is easier to automate and control network tasks, trigger-action IoT platforms will continue to become more popular in the coming days. Remote injection attacks or event spoofing attacks will also be more agile and complicated. The literature will require more research on the effective real-time defense solutions in the future. In this chapter, we discuss a few potential research works that will advance the state of the art approaches towards a more robust online defense system.

7.1 Competitive Multi-agent Reinforcement Learning (MARL) based Defense Solution

In *Competitive Multi-agent Reinforcement Learning (MARL)*, two or more agents engage in a competitive game, where each agent tries to maximize its own reward while considering the strategies and actions of other competing agents [100]. Each agent has its own reward function that dictates how the agent should optimally choose actions to maximize the overall gain. Each agent interacts with the environment by making an assumption that the competing agents try their best to impact the environment in their benefit. The agents can leverage the *minimax-Q* [101] [102] algorithm to infer the strategies and actions adopted by the competing agents. *Minimax-Q* algorithm is an opponent-independent Q-Learning [103] algorithm, and the agents use it when the opponents' strategies and actions cannot be optimally modeled. However, to better understand how opponents act in the environment, the agents can always perform opponent modeling and discover the opponents' behavior [104]. However, since all agents compete in the same environment in a competitive MARL scenario, the agents should always adopt their policies considering the possible non-stationary behavior

of the environment.

We see in Chapter 5 and 6 that the defense agent in a trigger-action platform always tries to infer the behavior of the attacker before taking an action at runtime. To discern an effective real-time defense policy, the defense agent must pay attention to the actions taken by the attacker and the consequences of the attacker's actions in the environment. In a trigger-action IoT platform, if there is an attacker who can perform opportunistic attack [16] and change the attack strategies based on the defense agent's actions at runtime, it is imperative for the defense agent to continuously perform behavioral analysis of the attacker before taking any defense action.

We think an effective online defense system for a trigger-action IoT platform can be well-designed if the the dynamic between the defense agent and the attacker can be modeled as a competitive MARL. Based on our previous discussion, both the defense agent and the attacker need to maintain their own reward function and choose appropriate learning algorithm to eventually yield optimal reward. The defense agent can conduct opponent modeling to learn the behavior of the attacker so that the defense actions produce maximum reward for the defense agent. We see in IoTWarden and IoTHaven that the defense agent takes defense actions as a reaction of the attacker's actions. If the defense system solves a competitive MARL to determine the attack-defense dynamic, the defense agent can effectively match the reactiveness of the attacker without incurring too much computation overhead.

7.2 Offloading Defense Policy Determination in Cloud

As we see in Chapter 6, the online defense against progressing remote injection attack demands defense actions to be taken under the uncertainty of the actual network states. The defense agent must translate observations into effective defense actions and collect optimized rewards for its actions. Since the optimal defense policy determination problem incurs computation overhead in the defense system, offloading the computation in cloud significantly reduces the computation overhead. As the

hub lacks computation power, hosting the defense agent in the hub and making effective defense actions at runtime negatively impact the overall performance of the defense system. When the discovery of defense policies and the further determination of defense actions are offloaded to the cloud, the hub can only be tasked to effectively enforce defense actions discerned by the defense agent improving the overall performance of the system.

To discern defense policies, the defense agent must maintain a function approximator in the cloud to estimate action-specific rewards. Similar to IoTHaven, the defense agent can use DRQN to obtain the optimal defense policy and adopt an *exploration-exploitation trade-off* approach to decide whether the stored experiences (i.e., *transitions*) should be prioritized over function approximator to get the current optimal defense action.

If the defense agent is hosted in cloud, the hub must communicate with the defense agent through a secure communication channel and vice versa. The agent should offer a defense decision to the hub whenever a new observation is emitted in the IoT environment. The hub should also be able to request for a defense action whenever there is a configuration change (e.g., *a new device is added/removed*) in the network. If there is a request from the hub, the defense agent should come up with an action suggestion considering the new configuration. To help the defense agent determine a defense action for the changed environment, the hub must send the configuration change information to the agent using the secure communication channel. However, since the hub is the ultimate entity to enforce the defense action suggested by the agent, the hub should also be able to reject a suggestion if needed. Therefore, the communication channel should also be used by the hub to send proper notification about the status (e.g., *enforced* or *rejected*) of suggested actions to the defense agent.

As a defense action enforcer, the hub must maintain a security policy for the IoT network. Whenever a new defense action is suggested to the hub, the hub first needs

to check whether the enforcement of that action complies with the defined security policy of the network. If yes, the action is enforced, and the defense agent is notified about the enforcement. However, if the suggested action doesn't comply with the security policy of the network, the hub must reject the action and send the rejection notification to the defense agent. The defense agent can further utilize this rejection information to refine the learning process and update the internal parameters of the neural network function approximator.

The defense agent, however, needs to maintain a *defense policy bank*, where a set of action sequences are stored as defense policies. Whenever the defense agent determines a defense action, the action must reflect the sequential pattern of one of these policies. If a defense action chosen by the agent is not enforced in the network, the defense agent knows how to update the chosen policy once it receives the rejection notification from the hub. It is imperative for the defense agent to update the chosen policy in such a way that the defense actions determined in the future following the updated policy are likely to be approved by the hub.

7.3 Designing Online Defense Systems with User-configurable Overhead

As we see in Chapters 5 and 6, the computation overhead of reinforcement learning (RL) based defense systems mostly occurs from the training procedure of the defense agent. In RL-based approaches, the agent chooses the optimal action at a certain state by exploring multiple state trajectories and estimating optimal reward along each trajectory. Since exploring state trajectories is a computation-heavy task, the hyperparameters chosen for the training procedure play a vital role in determining the computation overhead of the RL-based systems. If users can customize the hyperparameters considering the computation constraints of the network environment, managing the computation overhead incurred in the training procedure becomes more effortless. Users can also customize the function approximator settings based on the learning requirements and objectives. In online defense systems, the defense agent

must be reactive against attack actions. If users can efficiently manage system overhead by customizing hyperparameters and neural network settings, the defense agent can be trained to effectively obstruct the progress of a remote injection attack at runtime without sacrificing the overall performance of the system. We believe a well-designed online defense system with user-configurable overhead is a significant improvement to our system, particularly IoTHaven.

CHAPTER 8: Conclusion

Since the popularity of trigger-action IoT platforms is rising, the vulnerability of IoT devices due to the interaction with each other is also becoming prevalent. Adversaries exploit the chain of interactions among IoT devices to inject fake event conditions in the network to invalidly invoke actions in the target IoT devices. To perform a remote injection attack, an attacker performs reconnaissance over the IoT network, selects a few IoT devices to mimic, sends fake event conditions to the smart hub, and forces the hub to command actions in target IoT devices whose compromise benefits the attacker to achieve the ultimate attack goal. The attacker may have the capability to profile defense actions at runtime and change the attack strategy based on the impact of defense actions in the environment.

In this dissertation, we attempt to analyze the remote injection attack and discover attack behavior by observing the physical changes occur in the IoT environment due to the attack actions taken in the network. We also present two defense solutions to address the remote injection attack: one defense solution that works efficiently in fully observable IoT networks and one defense solution to provide realtime security in partially observable IoT networks. In this dissertation, we present 1) *IoTMonitor* for attack analysis and behavior discovery tasks, 2) *IoTWarden* for real-time defense in fully observable IoT networks, and 3) *IoTHaven* for online defense in partially observable IoT networks.

In IoTMonitor, we probabilistically map IoT event conditions with the corresponding physical evidence emitted during a remote injection attack using the Baum-Welch algorithm [20] and extract the hidden optimal sequence of event conditions using the Viterbi algorithm [21]. The optimal sequence of event conditions represent the opti-

mal attack path the attacker follows to successfully perform the injection attack. In IoTMonitor, we also determine the IoT devices mostly targeted by the attacker.

In IoTWarden, we present a Deep Reinforcement Learning (DRL) based defense approach that utilizes a neural network function approximator to select the optimal defense action at runtime. The function approximator is particularly designed to obtain the optimal defense policy given the actual states of the network. We also store the interactions of the defense agent with the environment in a replay buffer so that we can sometimes sample the optimal defense actions directly from the learned experiences avoiding the overfitting of the function approximator. We speculate attack strategies by training an LSTM-based RNN [22] with a number of attack sequences and train the function approximator using a Deep Q-Network (DQN) [23] so that it estimates the optimal state-action pairs maximizing the overall security gain.

In IoTHaven, we design and implement an online defense solution for partially observable IoT networks where the defense agent makes optimal defense actions under the uncertainties of the network states. The defense agent only observes the physical evidence generated in the IoT environment due to occurrence of events in different IoT devices. Hence, the defense agent solves a *Partially Observable Markov Decision Process (POMDP)* to discern the optimal defense policy that dictates how the defense actions should be taken to optimize the security reward just by observing physical changes in the network. We train a Deep Recurrent Q-Network (DRQN) [25] function approximator to help the defense agent choose the optimal state-action pairs at runtime.

REFERENCES

[1] "Internet of things and data placement." https://infohub.delltechnologies.com/en-us/l/edge-to-core-and-the-internet-of-things-2/internet-of-things-and-data-placement/. Accessed: 2024-06-15.

[2] "Ifttt: Every thing works better together." https://ifttt.com/. Accessed: 2024-06-15.

[3] "Samsung smartthings." https://www.smartthings.com/. Accessed: 2024-06-15.

[4] "Microsoft flow." https://powerautomate.microsoft.com/en-us/blog/welcome-to-microsoft-flow/. Accessed: 2024-06-15.

[5] "Home app: The foundation for a smarter home." https://www.apple.com/home-app/. Accessed: 2024-06-15.

[6] "openhab: Empowering the smart home." https://www.openhab.org/. Accessed: 2024-06-15.

[7] "Wink: The smarter app for the smart home." https://www.wink.com/. Accessed: 2024-06-15.

[8] "Zapier: Automation that moves you forward." https://zapier.com/. Accessed: 2024-06-15.

[9] Z. B. Celik, G. Tan, and P. Mcdaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[10] M. M. Alam and W. Wang, "A comprehensive survey on data provenance: State-of-the-art approaches and their deployments for iot security enforcement," *Journal of Computer Security*, vol. 29, pp. 423–446, 06 2021.

[11] M. M. Alam, M. S. I. Sajid, W. Wang, and J. Wei, "Iotmonitor: A hidden markov model-based security system to identify crucial attack nodes in trigger-action iot platforms," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1695–1700, IEEE, 2022.

[12] D. T. Nguyen, C. Song, Z. Qian, and S. V. Krishnamurthy, "IotSan: Fortifying the Safety of IoT Systems Dang," *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pp. 387–400, 2018.

[13] L. Babun, A. K. Sikder, A. Acar, and A. S. Uluagac, "Iotdots: A digital forensics framework for smart environments," *CoRR*, 2018.

[14] Q. Wang, W. Ul Hassan, A. Bates, and C. Gunter, "Fear and Logging in the Internet of Things," in *Network and Distributed Systems Security Symposium*, 2018.

[15] M. O. Ozmen, R. Song, H. Farrukh, and Z. B. Celik, "Evasion attacks and defenses on smart home physical event verification," in *30th Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2023.

[16] S. Birnbach, S. Eberz, and I. Martinovic, "Peeves: Physical event verification in smart homes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2019.

[17] C. Fu, Q. Zeng, and X. Du, "HAWatcher: Semantics-Aware anomaly detection for appified smart homes," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 4223–4240, Aug. 2021.

[18] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 1074–1088, Association for Computing Machinery, 2018.

[19] M. M. Alam, I. Jahan, and W. Wang, "Iotwarden: A deep reinforcement learning based real-time defense system to mitigate trigger-action iot attacks," in *2024 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6, IEEE, 2024.

[20] L. B. Baum and J. A. Eagon, "An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology," *Bulletin of the American Mathematical Society*, vol. 73, no. 3, pp. 360–363, 1967.

[21] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, p. 1735–1780, nov 1997.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *ArXiv*, vol. abs/1312.5602, 2013.

[24] M. M. Alam, A. B. M. M. Rahman, and W. Wang, "Iothaven: An online defense system to mitigate remote injection attacks in trigger-action iot platforms," in *2024 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, IEEE, 2024.

[25] M. J. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," *ArXiv*, vol. abs/1507.06527, 2015.

[26] S. Agarwal, P. Oser, and S. Lueders, "Detecting iot devices and how they put large heterogeneous networks at security risk," *Sensors*, vol. 19, no. 19, 2019.

[27] B. Huang, D. Chaki, A. Bouguettaya, and K.-Y. Lam, "A survey on conflict detection in iot-based smart homes," *ACM Comput. Surv.*, vol. 56, nov 2023.

[28] D. Chaki, A. Bouguettaya, and S. Mistry, "A conflict detection framework for iot services in multi-resident smart homes," in *2020 IEEE International Conference on Web Services (ICWS)*, pp. 224–231, 2020.

[29] D. Xiao, Q. Wang, M. Cai, Z. Zhu, and W. Zhao, "A3id: An automatic and interpretable implicit interference detection method for smart home via knowledge graph," *IEEE Internet of Things Journal*, vol. 7, no. 3, pp. 2197–2211, 2020.

[30] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, (New York, NY, USA), p. 1439–1453, Association for Computing Machinery, 2019.

[31] H. Chi, Q. Zeng, X. Du, and J. Yu, "Cross-app interference threats in smart homes: Categorization, detection and handling," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 411–423, 2020.

[32] W. Ding, H. Hu, and L. Cheng, "Iotsafe: Enforcing safety and security policy with real iot physical interaction discovery," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021.

[33] H. Chi, C. Fu, Q. Zeng, and X. Du, "Delay wreaks havoc on your smart home: Delay-based automation interference attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 285–302, 2022.

[34] "A comprehensive guide to smart home device compatibility." `https://www.adt.com/resources/smart-home-device-compatibility`. Accessed: 2024-07-18.

[35] H. Chi, Q. Zeng, and X. Du, "Detecting and handling IoT interaction threats in Multi-Platform Multi-Control-Channel smart homes," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 1559–1576, USENIX Association, Aug. 2023.

[36] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 433–451, 2016.

[37] E. Fernandes, J. Jung, and A. Prakash, "Security Analysis of Emerging Smart Home Applications," *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 636–654, 2016.

[38] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 539–552, USENIX Association, Aug. 2012.

[39] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 499–514, USENIX Association, Aug. 2015.

[40] "ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms," *Proceedings 2017 Network and Distributed System Security Symposium*, no. March, 2017.

[41] H. Nissenbaum, "Privacy as contextual integrity," *Wash. L. Rev.*, vol. 79, p. 119, 2004.

[42] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim, "Fact: Functionality-centric access control system for iot programming frameworks," in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '17 Abstracts, (New York, NY, USA), p. 43–54, Association for Computing Machinery, 2017.

[43] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, (USA), p. 361–378, USENIX Association, 2017.

[44] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu, and P. Mohapatra, "Iotgaze: Iot security enforcement via wireless context analysis," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pp. 884–893, 2020.

[45] Z. Alliance, "Zigbee specification faq." `https://web.archive.org/web/20130627172453/http://www.zigbee.org/Specifications/ZigBee/FAQ.aspx`. Accessed: 2024-06-15.

[46] NXP, "Zigbee cluster library user guide." https://www.nxp.com/docs/en/user-guide/JN-UG-3077.pdf. Accessed: 2024-06-15.

[47] Z. Alliance, "How z-wave works." https://z-wavealliance.org/learn-about-z-wave/. Accessed: 2024-06-15.

[48] "Mqtt: The standard for iot messaging." https://mqtt.org/. Accessed: 2024-06-15.

[49] "matter: The foundation for connected things." https://csa-iot.org/all-solutions/matter/. Accessed: 2024-06-15.

[50] T. Grouo, "What is thread?." https://threadgroup.org/What-is-Thread/Overview. Accessed: 2024-06-15.

[51] "Bluetooth technology overview." https://www.bluetooth.com/learn-about-bluetooth/tech-overview/. Accessed: 2024-06-15.

[52] B. Fouladi and S. Ghanoun, "Honey, i'm home!!, hacking zwave home automation systems," *Black Hat USA*, 2013.

[53] N. Lomas, "Critical flaw ided in zigbee smart home devices." https://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/, 2015. Accessed: 2024-06-15.

[54] "What is the mirai botnet?." https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/. Accessed: 2024-06-15.

[55] Y. Jia, F. Zhong, A. Alrawais, B. Gong, and X. Cheng, "Flowguard: An intelligent edge defense mechanism against iot ddos attacks," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9552–9562, 2020.

[56] P. Kumari and A. K. Jain, "A comprehensive study of ddos attacks over iot network and their countermeasures," *Computers Security*, vol. 127, p. 103096, 2023.

[57] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 195–212, 2017.

[58] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action iot platforms," in *Network and Distributed System Security Symposium, NDSS*, 01 2018.

[59] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang, "Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 465–481, 2020.

[60] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. A. Beyah, "Mpinspector: A systematic and automatic approach for evaluating the security of iot messaging protocols," in *USENIX Security Symposium*, 2022.

[61] C. Fu, Q. Zeng, H. Chi, X. Du, and S. L. Valluru, "Iot phantom-delay attacks: Demystifying and exploiting iot timeout behaviors," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 428–440, 2022.

[62] Z. B. Celik, P. Mcdaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *USENIX Annual Technical Conference*, 2018.

[63] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, (New York, NY, USA), p. 272–285, Association for Computing Machinery, 2020.

[64] M. Palekar, E. Fernandes, and F. Roesner, "Analysis of the susceptibility of smart home programming interfaces to end user error," in *2019 IEEE Security and Privacy Workshops (SPW)*, pp. 138–143, 2019.

[65] F. Corno, L. De Russis, and A. Monge Roffarello, "Empowering end users in debugging trigger-action rules," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, (New York, NY, USA), p. 1–13, Association for Computing Machinery, 2019.

[66] L. De Russis and A. Monge Roffarello, "A debugging approach for trigger-action programming," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI EA '18, (New York, NY, USA), p. 1–6, Association for Computing Machinery, 2018.

[67] B. Huang, H. Dong, and A. Bouguettaya, "Conflict detection in iot-based smart homes," in *2021 IEEE International Conference on Web Services (ICWS)*, pp. 303–313, 2021.

[68] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia, "Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes," in *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, (Republic and Canton of Geneva, CHE), p. 1501–1510, International World Wide Web Conferences Steering Committee, 2017.

[69] W. Ding and H. Hu, "On the safety of IoT device physical interaction control," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 832–846, 2018.

[70] X. Chen, X. Zhang, M. Elliot, X. Wang, and F. Wang, "Fix the leaking tap: A survey of trigger-action programming (tap) security issues, detection techniques and solutions," *Computers Security*, vol. 120, p. 102812, 2022.

[71] K. Yoshigoe, W. Dai, M. Abramson, and A. Jacobs, "Overcoming invasion of privacy in smart home environment with synthetic packet injection," in *2015 TRON Symposium (TRONSHOW)*, pp. 1–7, 2015.

[72] Y. Yu and J. Liu, "Tapinspector: Safety and liveness verification of concurrent trigger-action iot systems," *IEEE Transactions on Information Forensics and Security*, vol. 17, p. 3773–3788, 2022.

[73] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, "Static analysis of android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, 2017.

[74] K. Jiang, H. Zhang, W. Zhang, L. Fang, C. Ge, Y. Yuan, and Z. Liu, "Tapchain: A rule chain recognition model based on multiple features," *Security and Communication Networks*, vol. 2021, no. 1, 2021.

[75] S. Birnbach, S. Eberz, and I. Martinovic, "Haunted house: Physical smart home event verification in the presence of compromised sensors," *ACM Trans. Internet Things*, vol. 3, apr 2022.

[76] A. K. Sikder, L. Babun, H. Aksu, and A. S. Uluagac, "Aegis: A context-aware security framework for smart home systems," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, (New York, NY, USA), p. 28–41, Association for Computing Machinery, 2019.

[77] Z. B. Celik, P. McDaniel, G. Tan, L. Babun, and A. S. Uluagac, "Verifying internet of things safety and security in physical spaces," *IEEE Security Privacy*, vol. 17, no. 5, pp. 30–37, 2019.

[78] M. O. Ozmen, X. Li, A. Chu, Z. B. Celik, B. Hoxha, and X. Zhang, "Discovering iot physical channel vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, (New York, NY, USA), p. 2415–2428, Association for Computing Machinery, 2022.

[79] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, D. Zou, H. Jin, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation," in *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, (USA), USENIX Association, 2020.

[80] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky, "Packet-level signatures for smart home devices," in *Network and Distributed System Security Symposium*, 2020.

[81] L. E. Baum and G. R. Sell, "Growth transformations for functions on manifolds," *Pacific Journal of Mathematics*, vol. 27, no. 2, pp. 211–227, 1968.

[82] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[83] A. Grami, *The Gaussian Distribution*, pp. 201–238. 2019.

[84] "Dirichlet distribution." `https://users.ics.aalto.fi/ahonkela/dippa/node95.html`. Accessed: 2024-07-18.

[85] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel, "Program analysis of commodity iot applications for security and privacy: Challenges and opportunities," *ACM Comput. Surv.*, vol. 52, Aug. 2019.

[86] J. Fan, Y. He, B. Tang, Q. Li, and R. Sandhu, "Ruledger: Ensuring execution integrity in trigger-action iot platforms," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pp. 1–10, 2021.

[87] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, vol. 24, p. 9, dec 1999.

[88] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), p. 217–224, Association for Computing Machinery, 2002.

[89] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing, "Automated generation and analysis of attack graphs," in *Proceedings 2002 IEEE Symposium on Security and Privacy*, pp. 273–284, 2002.

[90] S. Jajodia, "Topological analysis of network attack vulnerability," PST '06, (New York, NY, USA), Association for Computing Machinery, 2006.

[91] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6, pp. 679–684, 1957.

[92] P. J. Huber, "Robust Estimation of a Location Parameter," *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73 – 101, 1964.

[93] M. Wunder, M. Littman, and M. Babes-Vroman, "Classes of multiagent q-learning dynamics with ϵ-greedy exploration," in *International Conference on Machine Learning*, pp. 1167–1174, 08 2010.

[94] R. Alake, "Loss functions in machine learning explained." `https://www.datacamp.com/tutorial/loss-function-in-machine-learning`. Accessed: 2024-07-18.

[95] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[96] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[97] K. Fukushima, "Visual feature extraction by a multilayered network of analog threshold elements," *IEEE Transactions on Systems Science and Cybernetics*, vol. 5, no. 4, pp. 322–333, 1969.

[98] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[99] "What is the purpose of the cell state in lstm?." `https://eitca.org/artificial-intelligence/eitc-ai-tff-tensorflow-fundamentals/natural-language-processing-with-tensorflow/long-short-term-memory-for-nlp/examination-review-long-short-term-memory-for-nlp/what-is-the-purpose-of-the-cell-state-in-lstm`. Accessed: 2024-07-18.

[100] L. Buşoniu, R. Babuška, and B. De Schutter, *Multi-agent Reinforcement Learning: An Overview*, pp. 183–221. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[101] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine Learning Proceedings 1994* (W. W. Cohen and H. Hirsh, eds.), pp. 157–163, San Francisco (CA): Morgan Kaufmann, 1994.

[102] M. L. Littman, "Value-function reinforcement learning in markov games," *Cognitive Systems Research*, vol. 2, no. 1, pp. 55–66, 2001.

[103] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 05 1992.

[104] D. Carmel and S. Markovitch, "Opponent modeling in multi-agent systems," in *Adaption and Learning in Multi-Agent Systems*, 1995.