System-Level Security Analysis of IoTs

By

ZHENG FANG

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Prof. Prasant Mohapatra, Chair

Prof. Matt Bishop

Assoc. Prof. Zubair Shafiq

Committee in Charge

2022

Dedications

`40d3963c636464e6f530955b60a14625`

`61d04851ce0844b14257a0288340977b`

CONTENTS

LIST OF FIGURES

List of Tables

ABSTRACT

System-Level Security Analysis of IoTs

The global Internet of Things (IoT) market is growing rapidly. It is predicted that the total global worth of IoT technology could be as much as 6.2 trillion US dollars by 2025. Most IoT systems involve IoT devices, communication protocols, remote cloud, IoT applications, mobile apps, and the physical environment. However, existing IoT security analyses only focus on a subset of all the essential components, such as device firmware or communication protocols, and ignore IoT systems' interactive nature, resulting in limited attack detection capabilities.

In this dissertation, we introduce frameworks to evaluate the security of IoTs at the system level. We design and prototype FORESEE, a cross-layer vulnerability analysis framework for IoT systems. It generates a multi-layer *IoT hypothesis graph* which is amenable to existing model checking tools. Even though we come up with a state space compression algorithm to reduce the size of the hypothesis graph, in the worst case scenario, the size of the generated hypothesis graph would still be exponential. To tackle the state explosion problem, we propose IOTA, a logic programming-based framework which generates exploit-dependency attack graphs and computes metrics (*shortest attack trace*, *blast radius*, and *severity score*) to help IoT system administrators to evaluate attack complexity and the impact of each vulnerability.

We then explore the problem of monitoring IoT app execution and uncovering user privacy leakage via wireless traffic analysis. We propose and implement a novel IoT security enforcement framework called IOTGAZE that can detect potential anomalies and vulnerabilities in the IoT system by comparing the sniffed wireless events sequence and the sequence extracted from IoT apps' descriptions. Moreover, in IOTSPY, we show that it is possible to infer the user's privacy, like living habits, routines, and even installed IoT applications by just sniffing the encrypted wireless traffic.

Our research confirms that the interactive nature of IoT components requires us to consider these components simultaneously so as to uncover more vulnerabilities and evaluate their impacts. We hope our works will stimulate more research on IoT system-level security.

ACKNOWLEDGMENTS

First, I would like to express deepest gratitude to my advisor Professor Prasant Mohapatra for his guidance and support. Professor Mohapatra always gives me great advice, motivation, and an outstanding degree of freedom for my research work. I feel so fortunate to work with him and will never forget his guidance and encouragement.

I also thank Professor Matt Bishop and Professor Zubair Shafiq for being my dissertation committee member. Special thanks go to Professor Trent Jaeger and Professor Zhiyun Qian, for their generous mentoring and collaboration. I would also like to thank all my colleagues at UC Davis Networks Lab. Especially I thank Hao Fu, Tianbo Gu, and Pengfei Hu for tremendous help and great joy they brought to me.

I am forever indebted to my parents for their constant love, understanding, and support. The sacrifices they have made for me are beyond words.

Chapter 1

Introduction

Internet of Things (IoT) is one of the most booming trends in recent years. We are seeing an increasing number of smart home platforms, such as Samsung SmartThings [1], Google Home [2], Philips Hue [3], and so on. IoT systems have also being applied to many other scenarios, such as healthcare [4, 5], industrial manufacturing [6, 7], and battlefield [8, 9], etc. It is predicted that IoT market size will reach $1.6 trillion by 2025 [10]. The number of IoT devices connected to the internet is projected to be 30.9 billion by 2025 [11].

Most of the existing IoT platforms, e.g., [1, 2, 12–14] etc., share similar architecture. Figure 1.1 illustrates a typical IoT system, which consists of 6 components: devices, communication protocols, IoT cloud, IoT apps, physical environment, and mobile apps. IoT *devices* include sensor, actuator, and infrastructure devices such as Wi-Fi router and various gateways. The devices communicate with each other and the corresponding gateways using wireless, low-power, and short-range *communication protocols*.

IoT *applications* are programmed using trigger-action paradigm, where triggers are some cyber/physical events or device status change and actions are devices' actions. The *mobile apps* are used for device setup, remote monitoring and control. IoT *cloud* hosts IoT apps and provides many other features like device management and analytics. The *physical environment* is a necessary component of IoT systems because they sense and modify physical environment features. For example, if a humidifier is turned on, the smart home's humidify will increase, which will be captured by the humidity sensor.

However, researchers have found numerous vulnerabilities on all of these components of IoT systems, as discussed in the next section. What's worse, IoT devices can interact with each other via IoT apps or physical channels, resulting in multiple attack traces to the same system resource. Therefore, to harden IoT systems, we should consider all of the system components simultaneously so that we can find out and fix all

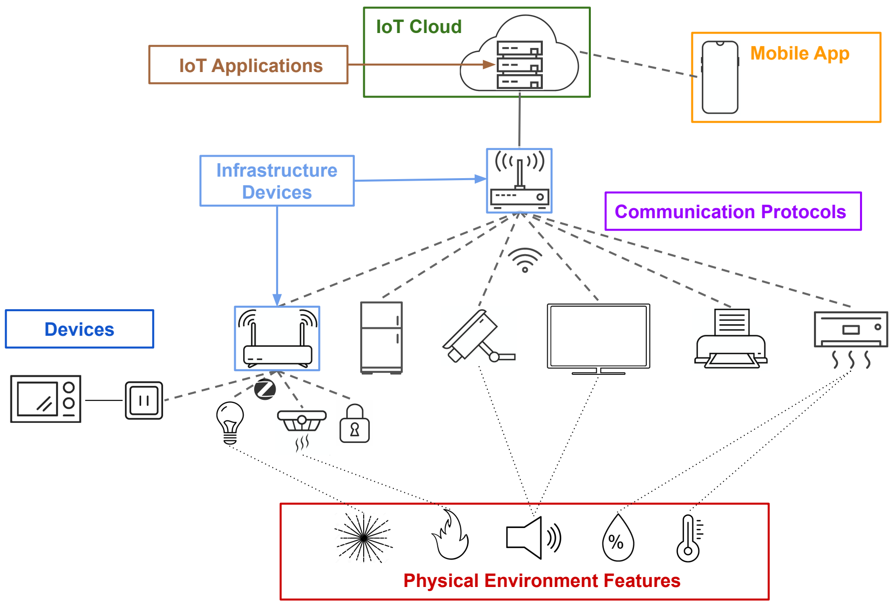


Figure 1.1: Typical IoT system and its components. IoT devices can sense environment events or change environment features, causing a chain of effects.

the potential attack traces.

Attack graphs [15–17] provide us an elegant approach to the problem by enumerating all of the attack traces. There are two types of attack graphs: *state-based attack graph* [16,18] and *exploit-dependency attack graph* [15, 17]. State-based attack graphs utilize model checking as the reasoning engine. But they suffer scalability issues in that the size of the graph grows exponentially with the number of system state variables (The number of system state variables is a linear function of the system device count). In comparison, it takes polynomial time to construct exploit-dependency attack graphs, and the generated attack graph size is a quadratic function of the system device count [15].

1.1 Vulnerabilities and Threats

Due to the rapid expanding of IoT market and a lack of security consciousness, vulnerabilities in IoT systems are ubiquitous. And interactive nature of IoT components can escalate the impact of a security breach in unanticipated ways. In this section, we explain the functionality of each IoT component, outline the potential vulnerabilities and threats discovered on it, and discuss the potential impacts on other components.

1.1.1 Devices

We use the term "IoT devices" to refer to both end devices connected to the IoT network and network infrastructure devices such as routers and gateways. Most of the device vulnerabilities are rooted in the firmware [19–22]. However, some vulnerabilities are found in the device's physical components such as gyroscopic sensors [23] and microphone [24].

A critical but often neglected threat to IoT devices relates to vulnerabilities of companion mobile apps, such as hard-coded keys. For example, the issue with August lock mobile app enables the attacker to decrypt communication between the lock base station and the mobile phone, resulting in Wi-Fi credential leakage if the attacker is physically adjacent to the smart home [25]. Moreover, the hard-coded key of the Eques plug mobile app makes it possible for the attacker to send spoofed commands to the remote server, allowing him to control the end devices through the internet [26].

Once a device is compromised, it can be used to attack other components of the IoT system in three different ways. First of all, if the attacker gets root privilege to a fully-fledged device such as a router or camera, he can send malicious commands to other devices on the same network. Second, the attacker can utilize the compromised device to inject physical events, such as changing the room's illuminance or injecting audio/video. Moreover, the attacker can take advantage of the devices' physical dependency to compromise other devices. For example, he can disable air conditioner by simply launching a DoS attack to the smart outlet it plugs in, or he can stop the smart sprinkler by compromising the water valve.

1.1.2 Communication Protocols

IoT systems utilize short-range, low-power protocols such as Zigbee and Bluetooth to communicate with the end devices. As a result, these end devices are first connected to a gateway (also called base station, bridge or hub) in order to communicate with the remote cloud. Notice that in an IoT system each protocol can have *multiple network instances*. For example, a smart home user can have both Smartthings [1] and Xiaomi [27] devices; even if they both use Zigbee protocol, there are two different Zigbee networks, one for the Smartthings platform and the other for Xiaomi. For system security analysis, We should consider multiple wireless network instances individually, because the compromise of the first network instance does not necessarily lead to the compromise of the second one.

Some devices use plaintext or hard-coded key [28] to transmit Wi-Fi credentials during the device setup phase. As a result, the attacker can easily steal the credentials and access the Wi-Fi network. Since there is no firewall or MAC address filtering in most of the smart home networks, if the attacker gains access to the network, he can freely send packets to other devices on the same network. To make things worse, many IoT devices, such as router, camera, or thermostat, expose unprotected network services to the home network, making it possible for the attacker to further compromise these devices after he joins the home network.

1.1.3 IoT Applications

Typically, IoT applications can be designed using *if-this-then-that* paradigm, where *this* represents a trigger and *that* means some action. A trigger can be either an individual IoT event, such as device state change, or sensor input. An action is some IoT device behavior, such as turning on the AC.

IoT apps automate the device behavior and result in device dependency, which exposes extra attack surface to the adversary. Consider "If smoke is detected, sound the alarm and open the window." as an example. To make the victim's window open, the attacker does not have to directly attack the window opener; instead, he can just compromise the smoke detector, and the IoT app will do the rest of the attack for him. Even though the attacker has to know the existence of such app before he launches the attack, people have shown this can be done via wireless sniffing [29, 30].

Moreover, for an IoT system, when multiple IoT apps are installed, they can issue conflicting commands, or create infinite loops. Therefore, it is necessary to analyze all of the installed IoT apps together in order to uncover such vulnerabilities.

1.1.4 Physical Medium

One of the unique features of IoT systems compared with other networked systems is that IoT devices can interact with each other via the physical medium. For example, if the door lock is locked, then the door opener cannot make the door open. Different from IoT apps-based device dependency which only exist if the app is installed by the user, physical device dependencies always exist in an IoT system. An attacker can utilize various physical dependencies to compromise IoT devices. For instance, he can turn off the AC or light bulb plugged into a smart outlet by launching a denial-of-service (DoS) attack to the outlet.

1.1.5 Mobile Apps

IoT mobile apps are used to setup the corresponding IoT devices. Besides, both IoT devices and mobile apps have connections to the remote cloud server so that users can utilize mobile apps to remotely control IoT devices. The most common type of vulnerability in mobile apps is cryptographic misuse. In particular, there are several reports [25, 26] about hard-coded keys in different apps, resulting in Wi-Fi credential leak

Table 1.1: IoT components and typical vulnerabilities/threats.

IoT System Component	Vulnerabilities / Threats
IoT App	Conflicting commands Repeated commands Sniffing
Mobile App	Malware Cryptographic misuses
Device Firmware	Unprotected network service Unprotected firmware updating process Buffer overflow
Device Hardware	Spoofing DoS Unprotected buttons/pins
Communication Protocol	Sniffing Spoofing Jamming
Physical Medium	Event injection Device chaining

or remote device control.

Table 1.1 summarizes common vulnerabilities/threats in all of the core IoT components.

1.2 Motivation

Given that there can be multiple vulnerabilities in different IoT components and that IoT devices interact with each other in various ways, we must uncover and fix all the attack traces in order to harden a certain system resource. Figure 1.2 shows a simple smart home IoT system consisting of 8 devices. We identified 5 attack traces (highlighted in different colors) leading to the compromise of different devices. Especially, there can be multiple attack traces to the same device — both the red and purple attack trace lead to the compromise of the doorlock.

Therefore, in order to truly defend some resource in an IoT system, we need to find out and fix all

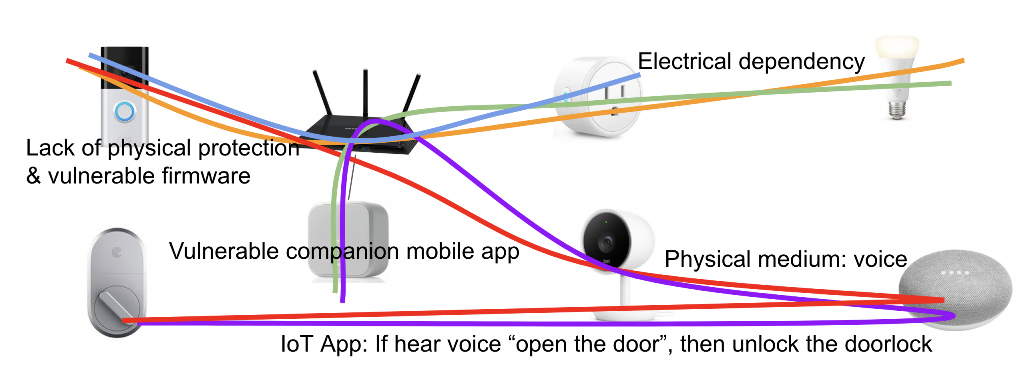

Electrical dependency
Lack of physical protection
& vulnerable firmware
Vulnerable companion mobile app
Physical medium: voice
IoT App: If hear voice “open the door”, then unlock the doorlock

Figure 1.2: A smart home IoT system example with the attack traces highlighted.

the attack traces to that resource. Attack graphs have been proven useful in system defense because they enumerate all of the system resources that can be compromised and the corresponding attack traces.

Figure 1.3 is the attack graph generated by our framework showing the two attack traces to the doorlock in Figure 1.2. From the figure, it is clear that to defend the doorlock, the system administrator should fix at least two CVEs shown in the attack graph.

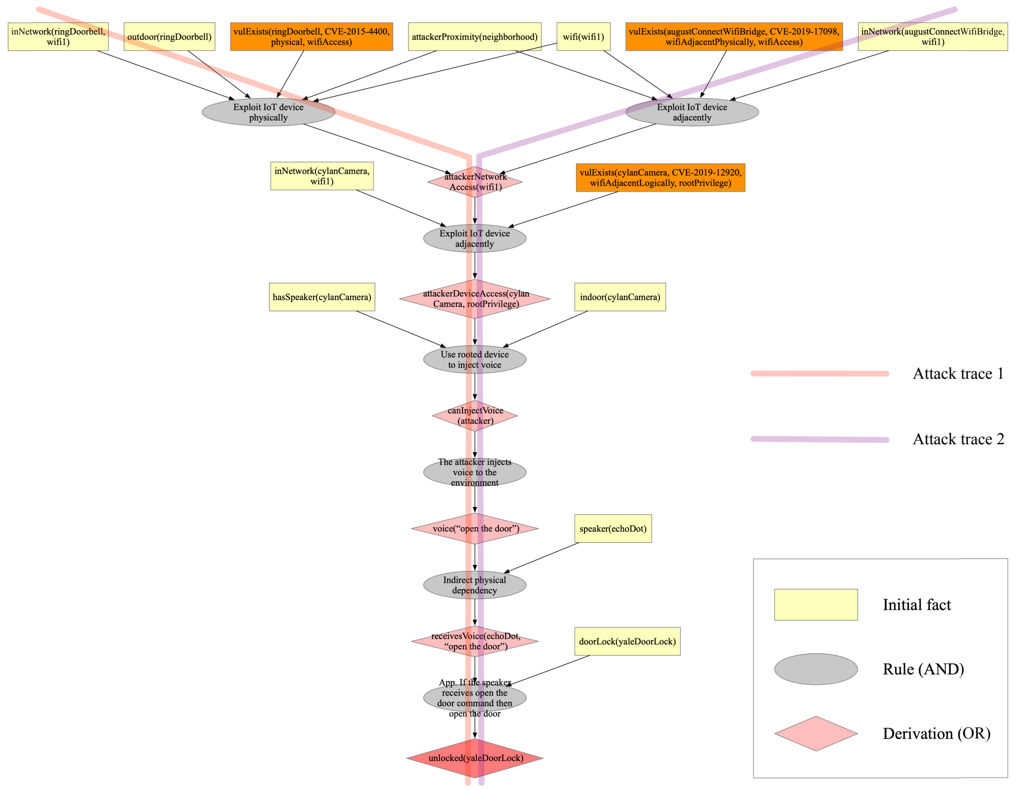

inNetwork(ringDoorbell, wifi1)
outdoor(ringDoorbell)
vulExists(ringDoorbell, CVE-2015-4400, physical, wifiAccess)
attackerProximity(neighborhood)
wifi(wifi1)
vulExists(augustConnectWifiBridge, CVE-2019-17098, wifiAdjacentPhysically, wifiAccess)
inNetwork(augustConnectWifiBridge, wifi1)
Exploit IoT device physically
Exploit IoT device adjacently
inNetwork(cylanCamera, wifi1)
attackerNetwork Access(wifi1)
vulExists(cylanCamera, CVE-2019-12920, wifiAdjacentLogically, rootPrivilege)
Exploit IoT device adjacently
hasSpeaker(cylanCamera)
attackerDeviceAccess(cylan Camera, rootPrivilege)
indoor(cylanCamera)
Use rooted device to inject voice
canInjectVoice (attacker)
The attacker injects voice to the environment
voice(“open the door”)
speaker(echoDot)
Indirect physical dependency
receivesVoice(echoDot, “open the door”)
doorLock(yaleDoorLock)
App. If the speaker receives open the door command then open the door
unlocked(yaleDoorLock)
Attack trace 1
Attack trace 2
Initial fact
Rule (AND)
Derivation (OR)

Figure 1.3: IoT attack graph example.

However, existing attack graph frameworks cannot be readily applied to IoT systems due to multiple design limitations. First, existing frameworks were designed for conventional computer networks and do not model essential IoT components such as IoT apps and devices' physical dependencies. Second, many IoT devices communicate using low-power protocols such as Zigbee or ZWave, which most of the existing vulnerability scanners cannot scan. For example, all of the vulnerability scanners listed on [31] only support IP-based devices. Moreover, they do not model exploits on low-power, short-range protocols which are ubiquitous in IoT systems. Finally, there are no quantitative criteria for administrators to harden the system in such a way that vulnerabilities with the largest impacts get patched first. As today's IoT systems may contain hundreds of vulnerabilities, it is necessary to patch vulnerabilities efficiently.

1.3 Related Work

1.3.1 IoT System Security

There are some existing works which attempt to analyze IoT system security by considering multiple IoT components. [32, 33] focused on uncovering application-level vulnerabilities using model checking techniques. Ding *et al.* [34] presented a framework that discovers potential physical interactions across applications using natural language processing (NLP) techniques and evaluated the risk score of each inter-app interaction chain. Mohsin *et al.* [35] proposed a formal framework for IoT security analysis based on satisfiability modulo theories (SMT). A probabilistic model checking based framework to analyze the risks quantitatively in [36].

Though some of these works claim to perform system-level analysis, they still just consider only a subset of the core IoT components identified by our work. For example, [32, 33, 35] consider devices and IoT apps, and [34] considers devices, IoT apps, and physical channels. Besides, they do not model individual vulnerabilities and exploits.

1.3.2 Attack Graphs

Attack graph. Automatic attack graph construction techniques are critical for system security analysis of networked systems. There has been extensive study on attack graphs for conventional computer networks. Sheyner *et al.* [16] proposed automated generation of attack graphs based on symbolic model checking. But their framework suffers from the state space explosion issue, making it difficult to model systems with hundreds of hosts. [15,17] utilized the monotonicity assumption to design attack graphs that can be generated in polynomial time. Attack graphs are also applied to intrusion detection systems [37].

Attack graph analyses. Ou *et al.* [38] introduce hypothetical analysis which answers the question of "what if there are some vulnerabilities in the system?". Sheyner *et al.* [16] propose two analyses, i.e., the minimal set of exploits to prevent so that the attackers fail to achieve their goals and the likelihood that the attacker will succeed. Ingols *et al.* [39] present automatic recommendations to improve system security by identifying a bottleneck device and patching vulnerabilities to prevent attackers from accessing the bottleneck. [40, 41] presented methods to harden computer networks using attack graphs. Nguyen *et al.* [42] propose a method to attribute safety violations to either bad apps or misconfigurations.

1.3.3 Wireless Traffic Analysis

The device fingerprinting technique is developed in [43, 44] to distinguish between legitimate devices and attacker devices. By analyzing the encrypted network traffic, [45] builds app fingerprints and [46, 47] can build the fingerprints for identifying the types of devices. Graph-based detection approaches [48, 49] can also be applied to IoT to detect anomalies. HoMonit [30] compares the IoT activities inferred from the encrypted traffic with their expected behaviors dictated in their source code. [50] discovers that public Wi-Fi networks can leak multiple categories of user privacy, such as identity privacy, financial privacy, social privacy, etc. The sensitive information, like the personal PIN and personal data, frequently typed by the user on mobile devices can be inferred through the motion sensors of a wearable device on user's wrist [51, 52]. The social relationships and demographics can be leaked by leveraging simple signal information from APs without examining the Wi-Fi traffic [53].

Chapter 2

FORESEE: A Cross-Layer Vulnerability Detection Framework for the Internet of Things

Existing IoT security research focuses on subsets of all of the core components in IoT [54–57]. However, the heterogeneity and interaction between different IoT components require a cross-layer framework which considers IoT system in a holistic view. For example, researchers at Pen Test Parterners have found that attackers can exploit communication protocol vulnerabilities to change the physical state of an IoT system, such as unlocking the doorlock [58] and increasing the temperature of a hair straightener [59]. Making the matters worse, the physical state change can be sensed by sensors, further triggering other actuator behavior. Even though recent IoT security analysis frameworks [60, 61] claim to be cross-layer, they still focus on a subset of IoT system components, such as communication protocol stack, failing to consider other essential components such as users' behavior, physical environment, and IoT applications. However, an increasing number of IoT attacks and threats [24, 62, 63] shows that cyber attack can lead to physical breach, and vice versa.

To solve these challenges, we present FORESEE, a cross-layer security analysis framework, to treat IoT system security from a holistic perspective. We first decouple and model an IoT system as a multi-layer graph, including *physical environment layer*, *device layer*, *communication layer* and *application layer*. The multi-layer graph models both intra-layer and inter-layer interaction between different components. Furthermore, FORESEE decomposes real-world IoT attacks into individual exploits and integrates them into the multi-layer graph, generating attack traces to show how they interact with system components.

The benefits of our approach are threefold. First, by considering all of the core components simultaneously, we can discover more vulnerabilities than existing frameworks do. For example, suppose an incompetent user is surfing the internet at home, and the indoor camera is running a vulnerable network service. If the user clicks the phishing site created by the attacker, the camera will be exploited. The attacker can use the compromised camera to further spoof a “intruder detected” event to trigger the alarm or other unwanted device behavior. Existing works fail to model user state or behavior and thus will not be able to detect such an attack path.

Second, we identify and model various device interactions. For instance, if an air conditioner is plugged into a smart outlet and the outlet has a denial-of-service (DoS) vulnerability, then the attacker can disable the air conditioner by launching a DoS attack on the outlet. As a result, the AC will also be off, failing to cool the room. This may even trigger other potential actions such as opening the window, etc. Frameworks focusing solely on software applications cannot discover such vulnerabilities because they involve electrical dependence between devices. Lastly, we can examine how seemingly unimportant vulnerabilities escalate and cause disastrous results due to the interactive nature of the IoT devices. This helps us better evaluate the vulnerabilities’ impact on system security and prioritize the protection against them.

We create a multi-layer graph by defining system states at different layers using boolean variables and formulating state transitions within each layer and between adjacent layers. After generating the multi-layer IoT system graph, we explore all the existing and potential attacks and incorporate them into the graph to form a final *hypothesis graph*. Then we apply model checking technique to detect various vulnerabilities and attacks. To alleviate the size explosion problem of the hypothesis graph, we design a state compression algorithm to intelligently generate independent sub-graphs without compromising the vulnerability detection ability.

2.1 System overview

Figure 2.1 depicts the structure of the framework. First of all, we gather all of the components of the target IoT system, including all the physical features (*Env*), user states and behaviors (*Usr*), devices installed (*Dev*), communication events (*Com*), and software applications installed (*App*). Then we construct the multi-layer IoT system transition graph. Thereafter, we decompose real-world IoT attacks into atomic attacks [64]. From the atomic attacks and the multi-layer system transition graph, we build the hypothesis graph and perform vulnerability detection with respect to the specified correctness properties. Finally, if there is a violation of the specified property, an error trace is returned to help us identify the cause. In Section 2.3, we present a state compression algorithm that selects applications and user states relevant to the given

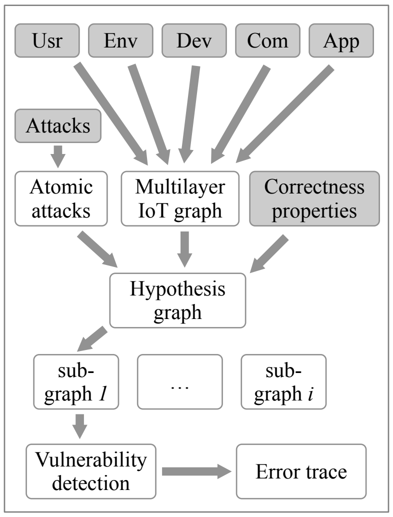


Figure 2.1: System overview.

correctness property. With the help of this approach, we can collect the relevant components and atomic attacks and directly generate the subgraph of the hypothesis graph for verification.

Before constructing the IoT hypothesis graph, we should determine the input of the framework shown as gray boxes in Figure 2.1. For a given IoT system, *App* and *Dev* are already known. Then, we can determine *Com* and *Env* based on *App* and *Dev*, since the communication events subscribed or issued by the apps and the mapping between devices and physical features are known. The *Attacks* are derived from the vulnerabilities, which are determined by searching the Common Vulnerabilities and Exposures (CVE) [65] entries for each device and protocol of the system. Once we know the vulnerabilities, we can establish the set of potential attacks on the IoT system. *Correctness properties* are system-specific. Soteria [33] proposed dozens of properties specific to smart home applications and five general properties such as no conflicting control commands or repeated commands in one code branch, etc. However, to the best of our knowledge, currently there is no work that automatically generates correctness properties or comprehensively deals with user states and behaviors.

2.2 Multi-Layer State Transition Graph

2.2.1 Multi-Layer Graph Construction

The heterogeneous and dynamic nature of IoT systems brings huge challenges for system security analysis. First of all, IoT systems directly interact with the physical environment, and can potentially interact with an infinite number of user states and behavior. Second, devices may be added to or removed from the IoT

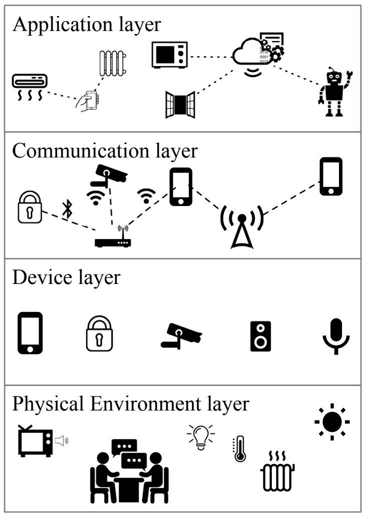


Figure 2.2: IoT hierarchy.

system frequently. Moreover, the framework should consider both system security and user safety as they are an essential part for most of IoT systems. To deal with the above challenges, we propose a novel formal framework that abstracts a complicated IoT system into a clear, layered structure. Our approach effectively decouples the processing logic of one layer from another so that the vulnerabilities within one layer would not be mixed with others. Moreover, multi-layer graph also enables us to detect violations which involve multiple layers, such as inconsistency between physical and device layer. Figure 2.2 gives an overview of the IoT hierarchy, which consists of four layers — *physical environment layer, device layer, communication layer*, and *application layer*.

We abstract the internal behavior of each layer as a directed, unweighted state transition graph $L = (V, E)$. In the graph, the node $v \in V$ represents a certain system state of the entire layer. A set of atomic propositions (AP) [66] and their values constitute distinct system states. Each atomic proposition is a boolean variable and describes the smallest unit of the system state that has the characteristic properties of an IoT element. By representing a system state at one layer using a collection of atomic propositions, we make our multi-layer state transition graph amenable to model checking algorithms. Sensor measurements of continuous values are discretized into boolean values and represented by atomic propositions as well. For instance, an AP at the device layer describes the value of one temperature sensor. The value of AP is $True$ if the temperature exceeds the threshold 80°F; otherwise, the value is $False$. We can use the set $v = \{b_{AP_1}, ..., b_{AP_i}, ..., b_{AP_K}\}$ to represent the nodes at a certain layer, where K is the number of APs in a certain layer. The value of b_{AP_i} is either $True$ or $False$. For a layer that has K atomic propositions, there will be 2^K nodes at that layer, representing 2^K system states. Then one edge $e \in E$ describes system state transition. The formal definition of multi-layer graph is as follows:

Definition. (Multi-Layer IoT System Transition Graph) A multi-layer IoT system transition graph is a tuple

$$G = (L^{(1)}, \dots, L^{(4)}, M),$$

where $L^{(i)} = (V^{(i)}, E^{(i)}),\ i = 1, \dots, 4$, denotes the system state transition sub-graph at physical environment, device, communication or application layer. M is the set of cross-layer edges which indicate the relationships between the adjacent layers and is formally defined as:

$$M = \bigcup_{i \in \{1,2,3\}} (M^{(i,i+1)} \cup M^{(i+1,i)}),$$

where $M^{(i,j)}$ is the set of edges from layer i to layer j.

In the rest of this section, we give detailed definitions of system transition for each layer, along with the node mappings (i.e., cross-layer edges).

Physical environment layer describes facts about physical surroundings and the user states in the IoT system, such as room temperature, humidity, the user being asleep, etc. Here we put the user states in this layer because they also describe the objective fact. The node $v \in V^{(1)}$ represents one specific state of the environment. The edges between nodes denote the system state transition, which may be caused by environmental change or the user's state change.

Suppose v_i and v_j are two nodes at physical environment layer. One atomic proposition AP_l at this layer describes the environmental temperature. If the temperature is larger than threshold θ, the value of b_{AP_l} is $True$, otherwise it is $False$. If the value of AP_l in node v_i is different from the one in node v_j while all of the other atomic propositions are of the same value, then there will be an edge from v_i to v_j and an edge from v_j to v_i, implying environmental temperature change.

Let us re-consider the "user and TV" example mentioned in the Introduction section. That the user leaves home without turning off the TV can be represented as an edge between a physical environment-layer node containing $env.user.watch\_TV$ and a device-layer node v_s, where $dev.TV.on$, $dev.presence.false$ and $dev.door.closed$ hold true. Furthermore, the "open the door" voice from TV causes the system state transition from v_s (through communication layer and application layer) to another device-layer node v_t

where $dev.TV.on$, $dev.presence.false$ and $dev.door.open$ hold true. In v_t, $dev.presence.false$ and $dev.door.open$ indicate a violation which can be detected by our framework.

Device layer focuses on IoT device status, which is determined by the variable values in the embedded OS of the device. The set of atomic propositions at this layer describes the measurements of environment features and actuator configurations. Some devices can sense the environment, such as the pressure sensor, while the other devices can be configured and operated directly by the user or controlled remotely by software applications, such as an air conditioner or a light bulb. The node $v \in V^{(2)}$ conveys the status of all IoT devices, in terms of atomic propositions and their values. Suppose an IoT device can detect the window state "open or closed", and the value of corresponding atomic proposition AP_k reflects the window state. If two nodes v_i and v_j have distinct $True$ and $False$ values of atomic proposition AP_k, and the other atomic propositions in the two nodes have the same value, then there is an edge to connect these two nodes, indicating a window state change event, such as "opening the window" or "closing the window". If the IoT system functions normally, every edge at application layer corresponds to an edge at device layer, because application commands are delivered to the devices and devices' configuration change are transmitted to the decision maker. The additional edges at device layer indicate some device is compromised, and thus the device status is no longer reported to the decision maker.

There exist cross-layer edges between physical environment layer and device layer, which reflect the route of state transmission. For instance, an edge from physical environment layer to device layer reflects how devices perceive the ground-truth physical state. The nodes $v_i \in V^{(1)}$ and $v_j \in V^{(2)}$ in the two layers have edges if and only if for each atomic proposition $AP_i \in v_i$, all of the associated atomic propositions in v_j have the same value as AP_i. There may be multiple edges connected to one node at physical environment layer, because one environment feature can be measured by multiple devices. For example, humidity can be measured by both thermostat and water leakage sensor. It should be pointed out that environment measurement by IoT devices is not necessarily equal to ground truth at physical environment layer, as devices could be malfunctioning or compromised.

Communication layer models the events transmitted between devices and the decision makers. Since we consider the most common case in which decision makers reside in the remote cloud which is proprietary and closed-source, we do not model the communication between different decision makers. The events can be categorized into data transmitted from sensors to decision makers, and commands from decision makers to executive devices. The set of the atomic propositions in this layer indicates these events.

The change of information to be transmitted due to sensor measurement is represented as edges in this layer. Suppose v_i is a node where an atomic proposition $humidity \geq 80\%$ holds true, and v_j is a node where the atomic proposition $humidity < 80\%$ holds true. Then the edge between v_i and v_j represents the communication event of information change to be sent by the humidity sensor, due to the humidity decrease.

An upgoing edge from device layer to communication layer indicates that a sensor detects an environmental change and delivers the information to decision makers via transmitting data packets, while a downgoing edge from communication layer to device layer implies a command is delivered to an actuator, causing its configuration change. Due to communication protocol defects or attacks, the communication event may be tampered, thus generating additional edges which lead to some system states that violate the correctness properties.

Application layer formalizes the state of decision makers, which is determined by the set of variable values of software proxies running on the decision making infrastructure. These software proxies act as conduits for physical devices. Hence, the set of atomic propositions in this layer characterizes decision maker's knowledge about the IoT system.

Every node in this layer denotes one particular decision maker state, and an edge represents decision maker state transition due to application rules, or environmental change and actuator configuration change reflected in decision maker's states. Consider room temperature increase causes window open as an example. Suppose the atomic proposition $app.win.closed$ holds true in v_i, while $app.win.open$ hold true in v_j. In particular, $app.temp >$ 80°F holds true in both v_i and v_j. Then the edge between the two nodes stands for the application rule to open the window when room temperature is higher than 80°F.

The edge from communication to application layer signifies that the event packets sent by the sensor are faithfully delivered to the decision maker, triggering the update of variable value in decision maker. Similarly, an edge from application layer to communication layer indicates that the decision maker' state is updated due to the application rules, and it also generates command packets to be sent to the actuator(s).

Only verifying that a system does not satisfy the property is not sufficient; we should also visit back to identify the root causes of attacks. In our framework, the interconnection among the layers is explicitly captured by their node mappings, which helps trace the influences from one layer to another and finally identify the propagation path a venerability.

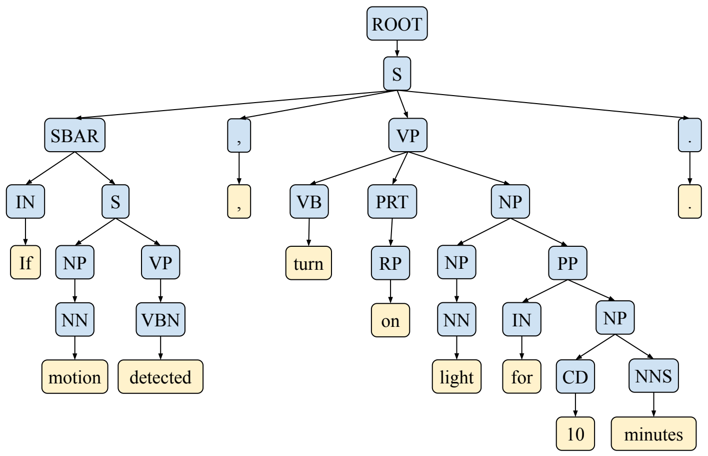


Figure 2.3: The constituency parse tree of the app description “If motion detected, turn on light for 10 minutes.”

2.2.2 IoT App Description Analysis

Since IoT applications decide the functionality of an IoT system and they are dependent on the users’ configuration, we need to design an approach to automatically extract the app logic based on app description or the app source code. Our method extracts apps’ semantic information from their descriptions using NLP techniques. A typical IoT app description is in trigger-action format where the trigger is some IoT event and the action means some device behavior.

First of all, we use an NLP parser to construct the parse tree and split the sentence into the conditional clause and the main clause by doing a breadth-first search (BFS) on the parse tree to find the tree node with label `SBAR`, which is the root of the subtree for the conditional clause. Then the conditional clause is obtained by concatenating the leaf nodes of this subtree. The main clause is constructed by removing the conditional clause from the original description. After that, we extract the noun and verb phrases from each clause using regular expression chunker and match the noun and verb phrases with device name and device actions, respectively. The matching is based on Word2Vec embedding [67]. Because the embedding is only for individual words, we split every phrase into words and choose the highest word pair similarity as the match result.

As an example, the parse tree of an IoT app description is shown in Figure 2.3. The conditional and main clauses after splitting are “motion detected” and “turn on light for 10 minutes.” The regular expression patterns for chunking is shown in Listing 2.1. The final extracted app logic can be represented as a Python dictionary shown in Listing 2.2.

```
NP: {<DT>?<JJ>*<NN.*>+}
VP: {<VB.*><IN|RP>?}
```

Listing 2.1: Regular expression patterns for chunking.

```
{'conditional': (['motion sensor'], ['motion']), 'main': (['bulb'], ['on'])}
```

Listing 2.2: Internal representation of an IoT app logic.

Based on the extracted IoT app logic and IoT system configuration information, we can get the state-transition graph for the application layer. Since the mapping of atomic propositions from one layer to another are straightforward, we can decide the cross-layer edges and the set of atomic propositions to be considered for the other three layers. Last, we need to decide the edges within the device and physical layer. The physical environment change are consistent in all of the IoT systems (For example, for all the IoT systems which consider environment temperature, there is an edge labeled with *temperature increase* from the node which contains *env.temperature.low* to the one which contains *env.temperature.high*) and thus can be pre-defined. The app logic will also be used by the dynamic selection algorithm (explained in Section 2.3.4).

2.3 Hypothesis Graph

Due to the interactive nature of IoT components, attacks may trigger unexpected security issues. Thus, it is necessary to model the attacker's behavior and integrate it into the IoT system model to construct a novel, more realistic state transition graph amenable to existing formal verification tools. We name our multi-layer IoT state transition graph with attacker behaviors as *hypothesis graph*. The formal definition of hypothesis graph is given below.

Definition. (IoT Hypothesis Graph) An IoT hypothesis graph is a multi-layer graph $G = (L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, M)$, where $L^{(l)} = (V^{(l)}, E^{(l)}),\ l \in \{1, 2, 3, 4\}$ is state transition graph at layer l and M is the node mapping. Each node $v \in V^{(1)} \cup \cdots \cup V^{(4)}$ denotes the system and the *attacker's state*. Each edge $e \in E^{(1)} \cup \cdots \cup E^{(4)} \cup M$ denotes environmental change, user's behavior, information flow, or *attacker's behavior*.

Compared with the multi-layer IoT system transition graph, the hypothesis graph contains extra atomic propositions for attacker's states, which can appear at all of the layers except the environment layer, and

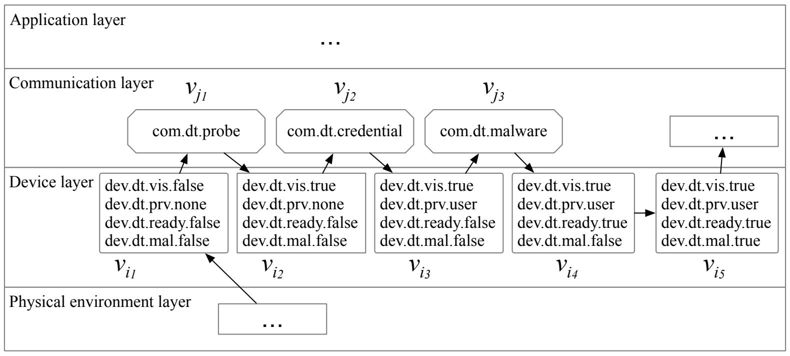

Application layer
...
Communication layer
v_{j1}
v_{j2}
v_{j3}
com.dt.probe
com.dt.credential
com.dt.malware
...
Device layer
dev.dt.vis.false dev.dt.prv.none dev.dt.ready.false dev.dt.mal.false
dev.dt.vis.true dev.dt.prv.none dev.dt.ready.false dev.dt.mal.false
dev.dt.vis.true dev.dt.prv.user dev.dt.ready.false dev.dt.mal.false
dev.dt.vis.true dev.dt.prv.user dev.dt.ready.true dev.dt.mal.false
dev.dt.vis.true dev.dt.prv.user dev.dt.ready.true dev.dt.mal.true
v_{i1}
v_{i2}
v_{i3}
v_{i4}
v_{i5}
Physical environment layer
...

Figure 2.4: The illustration of Mirai cross-layer attack.

additional edges for attacker's behaviors, which are across the device and communication layer, across communication and application layer, or within the device layer or physical environment layer.

2.3.1 Modeling Attacker Behavior

A real, complete attack may involve multiple atomic attacks. To model passive attacks (which do not change system configuration), we introduce atomic propositions associated with devices' or events' **visibility** to the attacker. To model active attacks (which change the system configuration), we introduce atomic propositions associated with the services running on a device and attacker's **privilege** on a device. We assume an attacker may have one of the three privileges on a device: $none$, $user$, and $root$. To model attacks via the network, we add atomic propositions to represent malware or other packets generated by the attacker such as username-password pair.

Here we use Mirai attack as an example and show how to decompose it into atomic attacks and represent each atomic attack. In Mirai attack, the device's infection mechanism can be decomposed into the following four steps — scanning the potential victim, brute-force login, malware dispatch, and malware execution. The first three steps happen at the communication layer, while the last one is at the device layer. Assuming the victim device is d_t, Figure 2.4 illustrates the cross-layer attack. Node $v_{i_1}{\sim}v_{i_5}$ are device-layer nodes representing system and attacker's states before or after the atomic attack. $v_{j_1}{\sim}v_{j_3}$ are communication-layer nodes representing probing packets, credential, or malware image, respectively. For clarity, we only list APs that are relevant to the Mirai attack. The values of all the other APs in $v_{i_1}{\sim}v_{i_5}$ are the same.

Device scanning: In this step, the attacker sends TCP SYN probes to pseudorandom IPv4 addresses on Telnet TCP ports. If the device d_t responds, then the attacker knows the existence of d_t, i.e., the device becomes visible to the attacker. This is reflected as the AP value change from $dev.d_t.vis.false$ to

$dev.d_t.vis.true$. The atomic attack is represented as two added edges (v_{i_1}, v_{j_1}) and (v_{j_1}, v_{i_2}).

Brute-force login: The attacker attempts to log in to the device by trying 10 different credentials. A successful login give the attacker $user$ privilege, which is reflected as AP value change in v_{i_2} and v_{i_3}. The attack behavior is also represented as two added edges (v_{i_2}, v_{j_2}) and (v_{j_2}, v_{i_3}).

Malware dispatch: After logging in to the victim device, the attacker checks the system environment, including OS version and CPU architecture, etc., and then download the malware binary image. Similar to the previous two steps, we use two APs to represent the system state before and after the attack. More specifically, $dev.d_t.ready.false$ holds true in v_{i_3} (meaning the device is not ready for the launch of the malware image), while $dev.d_t.ready.true$ holds true in v_{i_4}. Edge (v_{i_3}, v_{j_3}) and (v_{j_3}, v_{i_4}) model this atomic attack.

Malware execution: This step is the loading and execution of malware binary image. The AP $dev.d_t.mal.true$ in v_{i_5} indicates the malware process is running. The atomic attack is represented as the edge (v_{i_4}, v_{i_5}). Once executed, the malware performs a sequence of sabotage such as obfuscating its process name, killing other processes, or privilege escalation, etc. All these malicious behaviors are represented as additional edges that follow node v_{i_5}.

Many real-world IoT attacks can be decomposed as atomic attacks mentioned above. Our added atomic propositions make sure the correct sequence of atomic attacks which should be followed by the attacker. For example, the attacker should first sniff the existence of a device; only then can he launch the remote-to-user attack. To formally define an atomic attack, we need to identify the system and attacker states before and after the attack. Then the attack behavior is represented as the added edge between these two nodes.

2.3.2 Constructing Hypothesis Graph

As is shown in Section 2.3.1, for some attack, we need to introduce new atomic propositions (e.g., $dev.dt.vis.false$ and $dev.dt.prv.none$, etc.) to represent the attack. In this subsection, we define a basic operation named *state expansion* to show how to accommodate the newly inserted atomic propositions. Then we represent the attack behavior as edges and construct the final hypothesis graph.

State expansion. Suppose we are trying to insert an atomic proposition ap to a certain layer and the original graph of this layer has $|V|$ nodes and $|E|$ edges. After state expansion, the new graph for this layer has $|V'|$ nodes and $|E'|$ edges. If ap is independent of all the existing atomic propositions, then we have $|V'| = 2 \times |V|$ and $|E'| = 2 \times |E|$. Formally, when we try to insert an atomic proposition ap to layer l, first duplicate the original graph of layer l (The cross-layer edges are also duplicated.), then make all of the

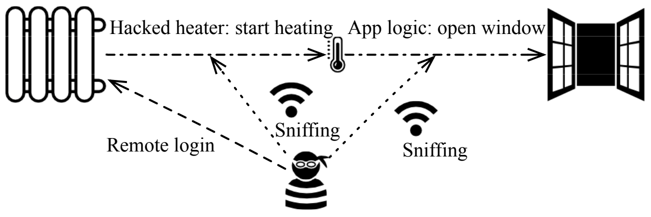


Figure 2.5: Example of smart home attack.

nodes of one copy have ap being *True*, while the other copy have ap being *False*.

After state expansion for all of the attacks which require additional atomic propositions, we can safely add edges to represent attack behaviors. The resulting graph is an IoT hypothesis graph whose nodes depicts system states including the attacker's state at certain layer and whose edges represent state transition due to environmental change, user's behavior, information flow, or attacker's behavior.

2.3.3 Vulnerability Detection

Our framework is built on top of model checking, which takes system graph and correctness properties as input, and outputs a counterexample if the system does not satisfy a certain correctness property.

Specifying correctness property. Correctness properties for a system can be classified as safety property (that something "bad" will never happen) and liveness property (that something "good" will eventually happen), which are expressed as Linear Temporal Logic (LTL) formulas [66].

Model checking. Though there are many model checking algorithms, their inputs all originate from Kripke structure [66]. Our multi-layer hypothesis graph conforms to the definition of Kripke structure, and thus is applicable to existing model checkers.

2.3.4 State Space Compression

A major challenge of model checking is the state explosion problem. Though introducing user and attack can make the system model more realistic, the number of nodes of the model gets 2^k (k is the number of newly introduced atomic propositions to represent user and attacker's states) times bigger, thus worsening the state explosion problem. Therefore, we propose a dynamic selection algorithm that selects relevant applications and user states, given the correctness property. Because the algorithm is executed before constructing the hypothesis graph, it can be used regardless of the model checking algorithm chosen.

Our algorithm is based on the observation that every correctness property or IoT application involves environment features and/or actuator configurations. Moreover, each user state and associated behavior can also be seen as a *virtual* application. Hence, formally any given application i can be represented as

$$App^{(i)} = (E_{in}^{(i)}, A_{in}^{(i)}, E_{out}^{(i)}, A_{out}^{(i)}),$$

where $E_{in}^{(i)}$ is the set of input environment features (including user states), $A_{in}^{(i)}$ is the set of input actuator configuration, and $E_{out}^{(i)}$ and $A_{out}^{(i)}$ are the output counterparts. For a virtual app of user state and behaviors, E_{in} is the set of the current user state, $A_{in} = \varnothing$, E_{out} is the set of all of the possible next state from current state, and A_{out} is the set of all the possible next actuator configuration due to user's behavior in current user state. Then we can determine whether any two given apps are related or not using the following rule: If one's output environment feature/actuator configuration is used by the other as input, or if the two apps have common output environment feature/actuator configuration, then they are related.

The algorithm is shown in Algorithm 1. Given the pool of applications and user data, along with the specified correctness property, the algorithm starts from environment features and actuator configurations used by the property as seeds, then iteratively marks applications (including virtual apps) until the set of marked apps does not change. The subroutine `is_related`(x, y) determines whether app x and y are related by checking whether one's output environment feature or actuator behavior is the other's input. If one of the set intersection is non-empty (meaning components interact with each other), there is a dependency. In Line 3, it puts $App^{(0)}$ as the seed. After we put all the related apps into S, we can remove $App^{(0)}$ (Line 16).

2.4 Case Study

We illustrate our framework's wide applicability by designing hypothesis graphs for two IoT systems: smart home and smart healthcare.

2.4.1 Smart Home

In this subsection, we present a proof-of-concept attack inspired by [34, 54, 57] and the corresponding IoT hypothesis graph. The attacks cross device, communication, and application layer. The attacker's goal is to break into a smart house. The smart house is equipped with a heater and an automatic window. Among the software applications installed, there is one particular app — If the temperature is greater than the threshold, then turn on the heater. The IoT setting and the attacker are illustrated in Figure 2.5. The safety properties is expressed in LTL syntax as

$$\boldsymbol{G}(dev.window.open \rightarrow \neg dev.user.state.u_0).$$

Algorithm 1: Dynamic Selection Algorithm

```
Input: $S_{App} = \{App^{(1)} \ldots, App^{(n)}\}$: the set of all the apps installed and the virtual apps
        representing user states and behaviors.
        $p$: the correctness property specified.
Output: $S \subseteq S_{App}$: the set of all the apps that should be considered together for the specified
        correctness property.
1  Algorithm dynamic_selection($S_{App}, p$)
2      Construct the virtual app $App^{(0)}$ by determining the set $E_{in}, E_{out}, A_{in},$ and $A_{out}$ from the
        given correctness property $p$.
3      $S = \{App^{(0)}\}$
       /* Iteratively add related apps to $S$.                                     */
4      $old\_size = |S|$
5      do
6          $T = S_{App} \backslash S$
7          for $x \in T$ do
8              for $y \in S$ do
9                  if is_related($x, y$) then
10                     $S = S \cup \{x\}$
11                 end
12             end
13         end
14         $new\_size = |S|$
15     while $new\_size \neq old\_size$
16     $S = S \backslash \{App^{(0)}\}$
17     return $S$
1  Procedure is_related($x, y$)
       /* Determine if app $x$ and $y$ are related.                                 */
2      return $x[E_{in}] \cap y[E_{out}] \neq \varnothing$ or $x[E_{out}] \cap y[E_{in}] \neq \varnothing$ or $x[E_{out}] \cap y[E_{out}] \neq \varnothing$ or
        $x[A_{in}] \cap y[A_{out}] \neq \varnothing$ or $x[A_{out}] \cap y[A_{in}] \neq \varnothing$ or $x[A_{out}] \cap y[A_{out}] \neq \varnothing$
```

Figure 2.6 shows the corresponding hypothesis graph for the scenario. For clarity, we only label each node with atomic propositions whose value get changed from the preceding node. The final red node denotes the violation of the property, i.e., the attacker's goal is achieved, and the label for each edge shows the cause of the state transition. Notice that there could be multiple paths connecting the same starting and ending node and here we are only showing one path for illustration.

The atomic proposition in the bottom left initial state tells us that the room temperature is less than the threshold. The increase of room temperature is sensed by the temperature sensor, and the sensor generates a wireless event (represented by the communication layer node with the atomic proposition *c_temp>80*). This wireless event is sniffed by the attacker, whose sniffing behavior is denoted as edge ①. The decision

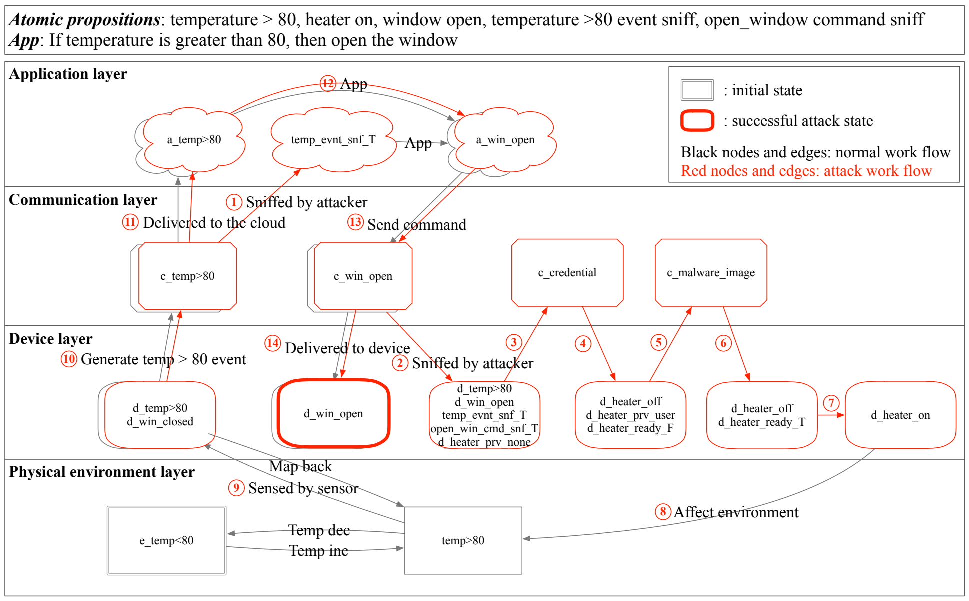


Figure 2.6: The hypothesis graph for a smart home with attack trace.

maker receives the event and updates the variable values in the software proxy. The App logic controls the window to open by sending the window open command (denoted by the node labeled with *c_win_open*) to the window. This control signal is also sniffed by the attacker (labeled by ②), and thus cause the atomic proposition *temp_evnt_snf* and *open_win_cmd_snf* to hold true. Edge ③∼⑦ represent the brute-force login, malware dispatch, and malware execution, as explained in Section 2.3.1. Edge ⑧∼⑭ denote the attacker's exploitation of the system's vulnerability to force the window open.

The model checking algorithm first determines that the node with a thick red border is a state which violates the specified correctness property. Then it traces back and marks all the preceding nodes until it reaches the initial node. Since from the initial system state we can finally reach the violating state, the hypothesis graph does not satisfy the specified correctness property, and the error trace is the red path in Figure 2.6.

2.4.2 Smart Healthcare

Our framework can be easily applied to other IoT scenarios, such as smart healthcare, or smart factories. Here we show how to create IoT hypothesis graphs for an automatic blood glucose control system, where the insulin pump automatically injects insulin when the blood glucose gets higher the normal range. The

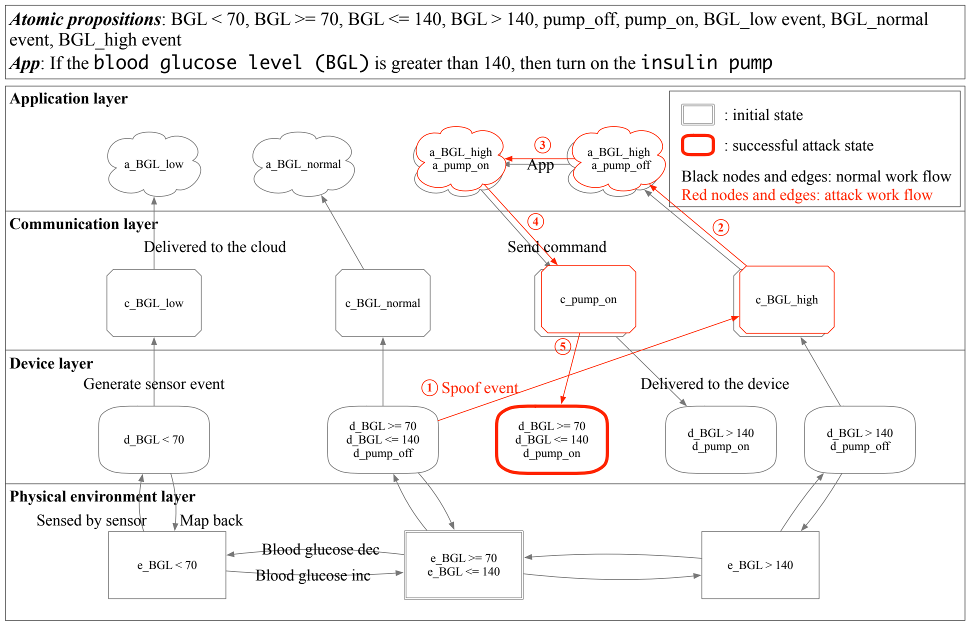


Figure 2.7: The hypothesis graph for an automatic blood glucose control system. The red shapes and lines illustrate the attack trace.

correctness property is represented as the following LTL formula

$$\boldsymbol{G}(dev.pump.on \rightarrow dev.BGL.high).$$

The attacker's goal is to make the pump inject insulin when the user's glucose level is within the normal range, which may cause severe medical issues. Therefore, if we can validate the above correctness property, we will be able to decide whether the attacker can achieve his goal.

Different from smart home scenarios where the physical environment features are physical quantities such as temperature, illuminance, smoke, etc., in smart healthcare, the environment features are the patient's biological features such as blood glucose level, or blood oxygen saturation, etc. The devices are also medical sensors and actuators such as blood glucose sensor and insulin pump. The communication and application layer are similar to those of the smart homes, representing communication events and decision maker states. The hypothesis graph of the above smart healthcare example is shown in Figure 2.7.

To differentiate atomic propositions at different layers, we add prefixes according to the layers they belong to. The atomic propositions are prefixed with `e`, `d`, `c`, `a`, from the lowest layer to the highest layer. Similar to Figure 2.6, to reduce clutter, we label each node with only the atomic propositions which hold true in that node.

The normal system transitions are represented in gray color, while the nodes and edges marked in red represent attack traces. Initially, the patient's blood glucose level (BGL) is within the normal range so the insulin pump is off. The attacker launches attack by first spoofing a BGL high event (as denoted by edge ①). Then the spoofed packet is delivered to the decision maker (②), triggering a command to turn on the pump (③). As a result, the pump is turned on even though the patient's actual BGL is normal (④, ⑤). Due to the initial attack edge ①, the resulting successful attack state can be reached from the initial state. Hence, the model checker detects the violation to the specified correctness property and returns the attack trace.

The above two examples show that since the four layers identified are essential for different IoT systems, and IoT cyber attacks also have similar mechanisms, our framework can be easily applied to different IoT scenarios.

2.5 Evaluation

2.5.1 Implementation

We implement FORESEE based on the Spin model checker [68]. Spin takes a system modeling language called Promela (Process Meta Language) [68] as its input language, and accepts correctness properties specified as Linear Temporal Logic (LTL) formulas. We use IoTSan to convert SmartApps to Promela language, then modify the Promela code by inserting variables for physical environment, device, and communication layer, as well as atomic attacks. After that, we perform verification by running the compiled program. If the system does not satisfy the given correctness property, the verifier will return an execution trace that caused the violation.

2.5.2 State Compression Ratio

To evaluate the effectiveness of our dynamic selection algorithm, we created 6 smart home scenarios whose IoT apps are chosen from the Samsung SmartApps [69]. On average, each scenario consists of 7.4 apps. We count the number of states of the generated hypothesis graph before and after applying the dynamic selection algorithm. Compression ratios are computed by dividing the number of states of the hypothesis graph generated from the largest subsets of related apps by the one generated from the original app set. The results are shown in Figure 2.8, where the state compression ratios are shown above the black dots in the figure. We can see from the figure that the compression ratio ranges from 1.5% to 0.057%, and for the larger set of apps our algorithm tends to achieve better ratio.

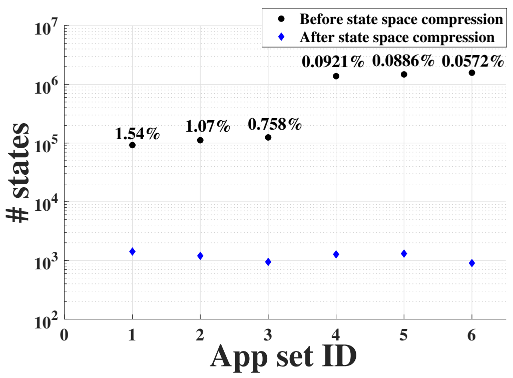


Figure 2.8: Number of states before and after compression and the compression ratio.

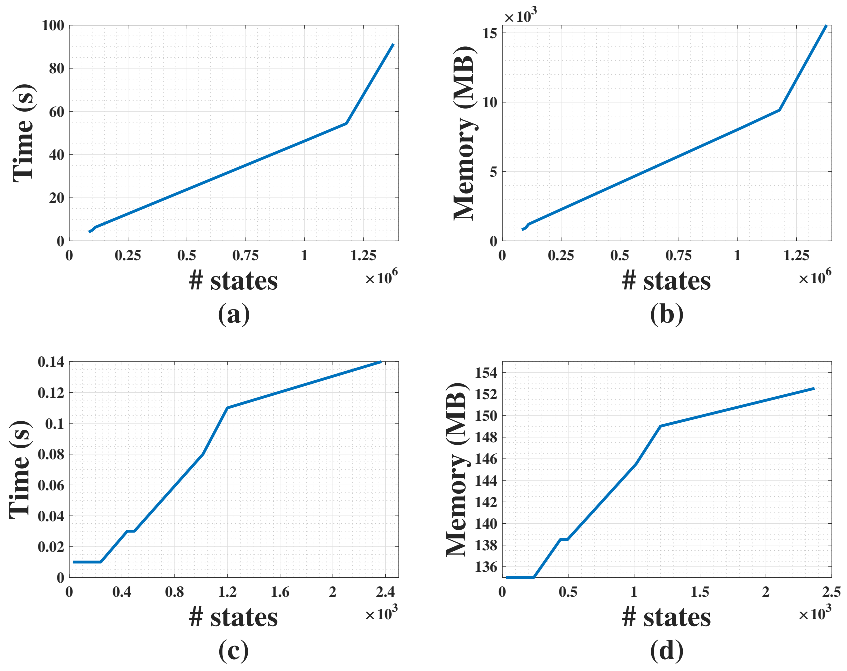


Figure 2.9: Impact of hypothesis graph size (measured by number of states) on verification time and memory usage. 2.9 (a) and 2.9 (b) show the scenarios where the models pass the verifier; 2.9 (c) and 2.9 (d) show the scenarios where there are violations to the property.

2.5.3 Performance Analysis

For a given LTL property, we test the scalability of our framework under two different settings: 1) when our system passes the verifier; 2) when there is a violation of the property. The time and space complexity of hypothesis graphs that pass the verifier are shown in Figure 2.9(a) and 2.9(b), while the ones of the hypothesis graph that fail to pass are shown in Figure 2.9(c) and 2.9(d). The x-axis variable "# states" denotes the number of unique states of the hypothesis graph traversed by Spin model checker. This is used as a measure of the size of the hypothesis graph. The y-axis variable "Memory (MB)" in Figure 2.9 denotes the sum of memory used to store all these states, hash table, depth-first search stack, and other overhead. Due to the server's memory limit, the maximum number of states we can run is around 1.4×10^6. Since Spin will immediately return after detecting a violation (i.e., an acceptance cycle), the verification process takes much less time and space if there is a violation. The scalability of our framework when there exists a violation is shown in Figure 2.9. From the figures, we can see the time and space cost scale linearly with the graph size, and it takes much less time and memory when there is a violation of the property.

2.6 Discussion

Our hypothesis graph uses model checking to detect IoT vulnerabilities. Researchers have also proposed other graph-based approaches for network security analysis [15, 17, 38]. In this section, we compare these methods and discuss benefits and limitations of our framework.

Cycles in the graph. Attack graphs created by [15, 17] utilize a key assumption of monotonicity, meaning once the attacker has gained some privilege, he will always have that privilege in the following attacks. However, this is not true for IoT systems because there could be some negative feedback loops due to automatic control. For example, when the attacker compromises the heater to increase room temperature, an IoT app will sense this physical event and automatically turn on the air conditioner to lower the temperature. Since attack graphs based on monotonicity assumption cannot deal with cycles, they cannot model and detect such attacks. In comparison, we do not make this assumption and our hypothesis graphs can uncover violations of correctness properties involving cycles.

State explosion. All of the model checked-based analyses encounter the state explosion issue. Even though our framework utilizes a dynamic selection algorithm, state explosion may still occur if the set of atomic propositions we need to consider gets larger. This can happen due to new IoT apps installed or devices enhanced with new capabilities, both of which introduce additional device interaction, connecting previously independent app sets. To further mitigate this issue, we select the bit-state hashing option during the verification. Instead of using the original state vectors, this approximation technique [70] uses the hash value of state vectors to index the state array, reducing memory usage to 1 percent while achieving the approximation ratio close to 100 percent [71].

2.7 Summary

In this paper, we design and prototype FORESEE, a cross-layer vulnerability analysis framework for IoT systems. We propose a formal approach to construct the IoT hypothesis graph which includes all of the core components of IoT systems, including user states and behavior that are largely ignored in existing works, and the potential attacks. We also present an approach to extract the IoT application logic from app descriptions using NLP techniques, and show the experimental results. Besides, we design a state compression algorithm to reduce the size of the generated hypothesis graph. The framework can detect vulnerabilities and threats caused by any interaction between IoT core components. Our evaluation shows that the prototype scales well and works for hypothesis graphs with millions of nodes. The compression algorithm is able to reduce the number of states by three orders of magnitude.

Chapter 3

IOTA: A Framework for Analyzing System-Level Security of IoTs

Existing research on IoT security only focuses on a single or a subset of the IoT components. For instance, [19,72,73] analyze IoT device firmware, [63,74] investigates IoT wireless protocols, and [42,75,76] sanitize IoT applications. However, for interconnected systems, hardening individual components cannot guarantee security because there are multiple paths to compromise system resources. For example, attackers can unlock a smart doorlock by exploiting vulnerabilities on the lock [77], but they may also compromise an indoor camera [78] and use it to inject voice, triggering a smart speaker to launch the door-open command [79]. In this paper, we try to address the following research problem — How to verify IoT systems security and uncover threats in a systematic way?

Attack graphs [15–17] provide us an elegant approach to the problem by enumerating all of the paths to potential *attack goals*, i.e., system resources which can be compromised by the attacker. There are two types of attack graphs: *state-based attack graph* [16,18] and *exploit-dependency attack graph* [15,17]. State-based attack graphs utilize model checking as the reasoning engine. But they suffer scalability issues in that the size of the graph grows exponentially with the number of system state variables (The number of system state variables is a linear function of the system device count). In comparison, it takes polynomial time to construct exploit-dependency attack graphs, and the generated attack graph size is a quadratic function of the system device count [15].

However, existing exploit-dependency attack graph frameworks cannot be readily applied to IoT systems due to multiple design limitations. First, existing exploit-dependency attack graphs were designed for conventional computer networks and do not model essential IoT components such as IoT apps and devices' physical dependencies. Second, many IoT devices communicate using low-power protocols such as Zigbee or ZWave, which most of the existing vulnerability scanners cannot scan. For example, all of the vulnerability scanners listed on [31] only support IP-based devices. Moreover, existing frameworks do not model exploits on low-power, short-range protocols which are ubiquitous in IoT systems. Finally, there are no quantitative criteria for administrators to harden the system in such a way that vulnerabilities with the largest impacts get patched first. As today's IoT systems may contain hundreds of vulnerabilities, it is necessary to patch vulnerabilities efficiently.

Goals. In this paper, our goal is to build a system-level security analysis framework for IoTs which, given the IoT system configurations (i.e., device, network information, and the IoT apps installed), (a) constructs exploit-dependency attack graphs to uncover resources that can be compromised and reveal potential attack traces; and, (b) computes a suite of metrics to interpret the generated attack graph and provide recommendations for system hardening.

As exploits and devices' dependencies are the key building blocks of attack graphs, to achieve (a), we extract exploit models and device dependencies from IoT system configurations and represent them as Prolog clauses [38]. More specifically, IOTA scans IoT system configurations for individual vulnerabilities and builds *exploit models* (consisting of precondition and effect) based on scanned CVEs. We identify three types of device dependencies: *app-based dependency*, *indirect physical dependency*, and *direct physical dependency*. The app-based dependencies are specified by IoT app semantics (i.e., trigger-action rules). Since IoT apps' source code can be unavailable in some platforms, such as IFTTT [80], we utilize natural language processing (NLP) techniques to extract app semantics from app descriptions. The direct and indirect physical dependencies are universal in IoT systems and thus are hard-coded as Prolog rules. Finally, Prolog clauses are sent to MulVAL [38] to generate attack graphs.

With regards to (b), we propose three novel metrics: *shortest attack trace* to an attack goal, *blast radius* of a vulnerability, and the *severity score* of a vulnerability. The shortest attack trace to an attack goal node provides the lower bound of the attack complexity in terms of the number of exploits to launch. A vulnerability's blast radius tells us the potential capabilities the attacker can get on the system by exploiting *only* that vulnerability. Severity score helps us identify the most critical vulnerability to patch when the importance of each resource we want to protect is known. In addition, the concept of *attack evidence* (defined to help us compute blast radius) can also be used to compute the *minimal set of vulnerabilities to patch to thwart an attack goal* [16]. These metrics help administrators interpret the attack graphs, sort the vulnerabilities based on their impacts on the system, and make informed choices about system hardening.

To evaluate IOTA, we generate 37 synthetic smart home IoT systems based on 532 real-world IoT apps and a list of 59 smart home devices of 26 types. We scan the CVE database since 2010 and find 127 IoT CVEs on those 59 smart home devices. Our vulnerability analysis module achieves 80.56% accuracy for exploit precondition identification and 88.19% accuracy for exploit effect. We manually check 27 shortest attack traces whose depths are at least 9 and find out 62.8% of them are beyond anticipation. In particular, the graph analyzer module reveals a shortest trace of depth 18 for an IoT system consisting of only 7 devices, implying that attack traces can be very deep for even a small IoT system. The case study illustrates the effectiveness of using the proposed metrics to estimate attack complexity and their impacts on the system. IOTA is highly scalable. In practice, it only takes around 1.2s and 120MB memory to evaluate IoT systems of 50 devices.

3.1 Threat Model

In this work, we consider individual attackers whose goal is to break into the system. They can be physically adjacent to the IoT system, enabling them to be within the radio range of the wireless local area networks, such as WiFi or Zigbee networks. Besides, the attacker can physically access outside, unprotected IoT devices, such as a doorbell or outdoor surveillance cameras. We also assume the attacker is able to extract IoT app semantics because he can install sniffers and infer event type from the sniffed packets [81]. We treat the remote IoT cloud as trustworthy and do not consider the compromise of the cloud itself. However, if there exist vulnerabilities on the companion mobile app, the attackers can spoof commands to the remote cloud.

3.2 System Overview

IOTA is a framework for automatic IoT attack graph generation and analysis. The generated attack graphs show all of the attack traces to IoT system resources that attackers can compromise. Based on the attack graphs, we further propose two metrics to provide recommendations for system hardening. Prior research on IoT system security [42, 62, 82] only focuses a subset of the essential IoT components, which neither uncover all of the attack traces nor provide guidance on how to prioritize patching vulnerabilities when there are dozens or hundreds of CVEs in an IoT system.

Figure 3.1 illustrates the pipeline of IOTA, which consists of four stages. **Vulnerability Scanning and**

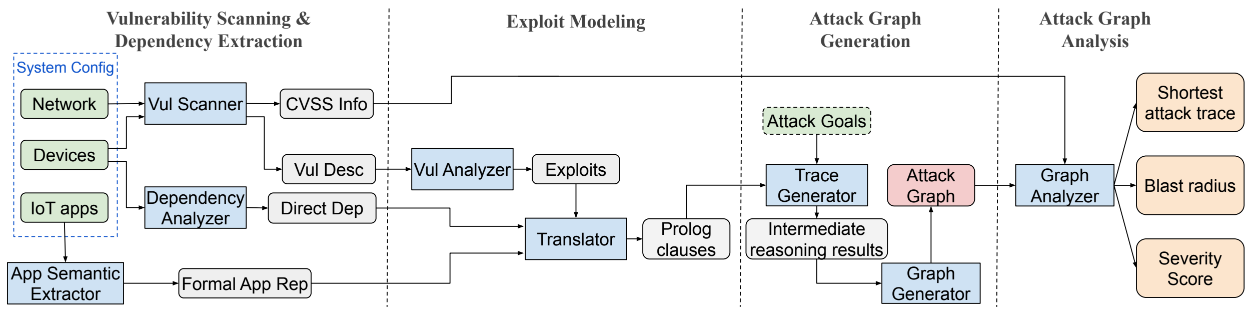


Figure 3.1: IOTA pipeline. The blue boxes are IOTA modules. The green, red, and gray boxes represent input, output, and intermediate results, respectively. The *Attack Goals* can be set by the system administrator and is optional.

Dependency Extraction stage scans the devices and network protocols for vulnerabilities and extracts IoT app semantics. It also extracts direct device dependencies from the system configuration file. **Exploit Modeling** stage maps vulnerabilities to exploits based on the vulnerability description and Common Vulnerability Scoring System (CVSS) [83] scores such as Attack Vector and Confidentiality, Integrity, Availability (CIA) triad. Exploits, direct dependencies, and app-based device dependencies are then translated to Prolog clauses. **Attack Graph Generation** stage reads attack goals (optional) and the translated Prolog clauses and generates IoT attack graphs. If the user does not provide attack goals, we enumerate all system resources (i.e., privileges on devices or tamper of the physical features) as potential attack goals. Then we modify the intermediate reasoning results and send them to MulVAL [38] to generate attack graphs. **Attack Graph Analysis** stage takes the generated attack graph as input and computes three metrics: the shortest attack traces to attack goal nodes and the blast radius and severity score of each vulnerability.

3.3 Iota Design

In this section, we explain the design of the IOTA modules as shown in Figure 3.1. The implementation details are explained in Section 3.4.

3.3.1 Vulnerability Scanner

To the best of our knowledge, there are no existing vulnerability scanners readily available for low power communication protocols such as Zigbee or Bluetooth Low Energy (BLE). Therefore, we design a vulnerability scanning approach based on CVE database searching. Our approach is practical because of some device vendors' ignorance of vulnerability report [84, 85] and the slow firmware upgrade rate [73].

Given the IoT devices installed and the communication protocols used, the Vulnerability Scanner module

Table 3.1: IoT device direct dependencies and examples.

Direct Dependency	Example
Electrical	Outlet → AC; Switch → Light bulb
Mechanical	Door lock → Door opener
Utility	Water valve → Sprinkler; Gas valve → Stove

searches the CVE database [65] for vulnerabilities. We fetch the CVE JSON feeds since 2010 from the National Vulnerability Database (NVD) [86] and parse the JSON files to get information relevant to our exploit modeling, including impact score, exploitability score, exploit range, exploit result (CIA triad), and the vulnerability description. The parsing results are stored in a local MySQL database. In total, there are 121,210 CVEs from 2010 to April 2021. After discarding CVEs without CVSS information, our database contains 113,180 records. For each device instance listed in the system configuration file, we query the database for the device name using full-text search in boolean mode to make sure it only returns CVE records when all of the query keywords appear in the CVE description.

The scanned vulnerability on each device is then translated to Prolog facts. For example, the following fact shown in Listing 3.1 means vulnerability `CVE-2020-8864` exists on `dLinkRouter`.

```
vulExists(dLinkRouter, 'CVE-2020-8864').
```

Listing 3.1: Prolog fact for a CVE found on a device.

3.3.2 Dependency Analyzer

The Dependency Analyzer module models how IoT devices interact with each other via physical channels. We identify and define two types of physical dependencies: direct dependency and indirect dependency. Two devices are *directly dependent* if they are both actuators. There are three types of direct dependencies as listed in Table 3.1. The most common direct dependency is **electrical dependency**, such as the one between smart outlet and air conditioner. The second type is **mechanical dependency**. For example, the door opener cannot open the door if the door lock is locked. We define the third type as **utility dependency**. For example, gas valve and smart stove are dependent via gas. Even though direct dependencies can have a huge impact on IoT system security, they are overlooked by existing IoT security analysis frameworks.

Two devices are *indirectly dependent* if one of them is an actuator and the other is a sensor. We consider and model six physical channels: temperature, humidity, illuminance, voice, smoke, and water. We include "voice" as a physical channel because many devices like cameras and TVs can play human voice in the smart home, and some devices can recognize human voice and execute the corresponding instructions.

For indirect physical dependencies, we identify the physical channels a device can sense and modify using NLP techniques. Google Assistant [87] lists all the *traits* a device can have for each device type. A trait consists of attributes (i.e., sensor readings) and commands (i.e., capabilities). We use Stanford CoreNLP framework [88] and Natural Language Toolkit (NLTK) [89] to parse the natural language description for each attribute and command description to extract nouns. After that, we use the Word2Vec model [67] to get the embedding of each noun and the six physical channels, and then compute cosine similarity between them. If the similarity is lager than the threshold, then we add the corresponding physical channel to the set of channels the device senses or modifies. The NLP results are sent to the Translator module to generate Prolog rules.

Direct physical dependencies are hard-coded as Prolog rules because they are universal in all IoT systems, regardless of the installation of certain IoT apps. Besides, since there are limited kinds of direct dependencies, hard-coding them guarantees accuracy. During the execution of a Prolog program, a certain dependency rule will be activated only when the corresponding device is installed. For example, if AC is on, then the room temperature will be low. But if there is no temperature sensor installed, the predicate of sensor reporting low temperature will not hold true. Listing 3.2 and Listing 3.3 are example of Prolog rules for indirect and direct dependencies, respectively.

```
high(temperature) :-
    on(Heater),
    heater(Heater).

reportsHigh(TemperatureSensor, temperature) :-
    high(temperature),
    temperatureSensor(TemperatureSensor).
```

Listing 3.2: Indirect physical dependency.

```
off(Device) :-
    plugInto(Device, Outlet),
    outlet(Outlet),
    off(Outlet).
```

Listing 3.3: Direct physical dependency.

3.3.3 App Semantic Extractor

The App Semantic Extractor module extracts semantic information from IoT app descriptions using NLP techniques. Compared with program analysis, analyzing IoT app descriptions in NLP is more applicable in that app descriptions are publicly available while IoT apps' code may be proprietary on some platforms. In smart home platforms, developers write a short description to explain the functionality of their IoT apps to smart home users. Typically, these app descriptions are written in "If this, then that" form, which makes it suitable for NLP techniques.

We use Stanford CoreNLP framework and Natural Language Toolkit (NLTK) for app description analysis. Given an app description, we use CoreNLP parser to construct the constituency parse tree and split the sentence into the conditional clause and the main clause based on tree node labeled `SBAR` (subordinate clause). We do a breadth-first search on the parse tree to find the tree node with label `SBAR`, which is the root of the conditional clause. Then the conditional clause is obtained by concatenating the leaf nodes of the subtree whose root node has label `SBAR`. We construct the main clause by removing the conditional clause string from the whole sentence.

Because the conditional clause and the main clause may contain multiple conditions or actions, we further split each clause into simple sentences based on tree node labeled `CC` (coordinating conjunction). The coordinating conjunction represents either logic AND or logic OR relationship between the two simple sentences. For example, the split of SmartApp *Hall Light: Welcome Home*'s description "Turn on hall light if someone comes home and the door opens." is shown in Listing 3.4. The conditional clause is split into two simple sentences with logic AND relationship. Since the main clause contains just one simple sentence, the relationship is set to `'NONE'`.

```
conditional:  ('AND', ['someone comes home', 'the door opens'])
main:  ('NONE', ['Turn on the hall light'])
```

Listing 3.4: Splitting clauses into simple sentences for the SmartApp *Hall Light: Welcome Home*.

After splitting each clause into simple sentences, we extract noun and verb phrases from each simple sentence and match them to IoT device names and device action using Word2Vec similarity. We use a regular expression chunker to extract noun phrases and verb phrases. The regular expression patterns we use for chunking and the extracted phrases for the SmartApp description are shown in Listing 3.5 and Listing 3.6, respectively.

```
NP: {<DT>?<JJ>*<NN.*>+}
VP: {<VB.*><IN|RP>?}
```

Listing 3.5: Regular expression patterns.

```
conditional clause: [(['someone'], ['comes']), (['the door'], ['opens'])]
main clause: [(['the hall light'], ['Turn on'])]
```

Listing 3.6: Noun and verb phrases extracted for the SmartApp *Hall Light: Welcome Home*'s description.

Finally, we use Word2Vec model to match the extracted noun phrases and verb phrases with our predefined list of devices and device actions. Since Word2Vec only computes similarities between individual words, we compare each word in a noun phrase against each word in a device name. The app semantic extraction result is represented as a Python tuple shown in Listing 3.7. This internal representation is used together with app configuration information in the Translator module to generate Prolog rules.

```
('AND', ['motion sensor', 'door contact sensor'], ['motion', 'open'], 'NONE', ['
    bulb'], ['on'])
```

Listing 3.7: Internal representation of the IoT app semantic.

3.3.4 Vulnerability Analyzer

The Vulnerability Analyzer module maps vulnerabilities to exploit models. Exploit modeling is essential for attack graph construction because attack traces are composed of individual exploits. To the best of our knowledge, our work is the first to attempt to *automatically* generate exploit models based on CVEs' natural language description and CVSS scores. Though MulVAL [38] formally represents exploits as Prolog rules, it only considers privilege-escalation in computer networks. Our exploit models are designed for generic IoT systems and consist of exploit precondition and effect. A *precondition* is the privilege the attacker should have in order to launch an exploit. An *effect* is the direct result of an exploit.

Precondition. For IoT systems, we define five types of preconditions listed in Table 3.2. Because IoT systems typically involve low-power, short-range, wireless protocols such as Wifi or ZigBee, the physically or logically adjacent precondition should be defined for each network type specifically, such as `Wifi adjacent logically`, `Zigbee adjacent physically`, etc. We cannot just use the "attack vec-

Table 3.2: Types of exploit preconditions on IoT devices.

Precondition	Explanation
Network	An attacker can exploit the vulnerability from the internet. There's no prior privilege required on the IoT system.
Adjacent physically	The attacker needs to be within the radio range of a wireless network, but he/she does not need to be on the network.
Adjacent logically	The attacker should be both within the radio range and on the wireless network, in order to launch the exploit.
Local	The exploit requires access to the device with at least user privilege, such as establishing Telnet or SSH connection to the device.
Physical	The attacker needs physical access to the vulnerable device to commit exploits.

tor" value as the precondition of each CVE in the NVD database because that field can be ambiguous or sometimes incorrect: According to [90], it assigns "network" as the precondition whenever there is a lack of information to decide the exploit range. Besides, the value does not differentiate "physically adjacent" and "logically adjacent".

We predict the exploit precondition based on protocol type, CVE description, and the CVSS attack vector. If an exploit's attack vector is `local` or `physical`, we keep its value. If the attack vector is `adjacent`, we check its CVE description. If the description contains keywords such as "sniff", "decrypt" or their synonyms, we will assign the precondition as `adjacent physically`, otherwise `adjacent logically`. If the original attack vector is `network`, we will first check the protocol. If the protocol is a low-power protocol, then we invoke the approach of determining `adjacent`; otherwise, we set the precondition to `network`.

Effect. From an attacker's perspective, exploit effects include gaining privileges on IoT devices, accessing wireless traffic, or making devices denial-of-service. We categorize exploit effects into six types as listed in Table 3.3. If attackers get `root` privilege on a device, they can use it to attack other devices by sniffing or spoofing wireless traffic. The `device control` privilege implies both `command injection` and `event access` privilege, but not the capability of attacking other devices on the same system.

For each vulnerability, we decide its exploit effect based on the CVE description and confidentiality, integrity, and availability (CIA) subscores of CVSS. We first seek to extract the effect from the CVE description by matching the keywords for each effect type. If the description does not contain any keywords,

Table 3.3: Types of exploit effects on IoT devices.

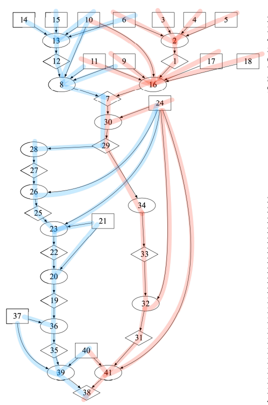
Effect	Explanation
Root	Attackers have root privilege on a device, which can be used to send spoofed commands to other IoT devices on the same network.
Device control	The attacker can run any command the device supports, and sniff and inject any device event. But the device cannot be used to send spoofed messages to other devices.
Command injection	The attacker can inject any commands to the device, but does not have access to the IoT events on that device.
Event access	The attacker is able to sniff and spoof events on an IoT device, but he/she cannot inject commands to that device.
Wifi access	The attacker obtains the Wifi credentials and is able to join the Wifi network.
DoS	The IoT device becomes denial-of-service.

we try to identify the *exploit mechanism* defined in [91] and communication protocol from the description using the same keyword matching approach. In combination with the CVSS' CIA subscores, we can infer the exploit effect. For example, suppose the exploit mechanism is buffer overflow. Then we check the CIA subscores to set effect to `denial of service` (if only the availability score is greater than the threshold), or `root privilege` (if confidentiality, integrity, and availability are all greater than the threshold).

Probability. We compute probability based on the *exploitability* subscore of the CVSS score for each CVE, which considers key factors such as attack vector, attack complexity, privileges required by the attacker, user interactions (for example, the DNS rebinding exploit requires the user to click a bait website), and attack timing (e.g., an exploit is effective only when the device is unpaired). We convert the exploitability subscore to exploit probability by rescaling the exploitability subscore to the range of 0 to 1. With exploit probabilities, we can compute the probability of each node on the attack graph using the formula given in [92]. Our graph analyzer implements probability computation, which is used as one of the building blocks of the *severity* metric defined in Section 3.3.6.

The exploit models are also translated to Prolog facts. For example, the `vulProperty` fact in Listing 3.8 is the exploit model for `CVE-2020-8864`. The precondition is the attacker being on the same Wifi network as `dLinkRouter` and the effect is that the attacker gets root privilege on this device. The exploit probability is 0.7.

```
vulExists(dLinkRouter, 'CVE-2020-8864').
```



```
1,"attackerDeviceAccess(hueBridge,rootPrivilege)","OR"
2,"RULE 54 (Exploit. The attacker physically adjacent to the zigbee network launches exploit)","AND"
3,"vulExists(hueBridge,'CVE-2020-6007',zigbeeAdjacentPhysically,rootPrivilege)","LEAF"
4,"inNetwork(hueBridge,zigbee1)","LEAF"
5,"zigbee(zigbee1)","LEAF"
6,"attackerProximity(neighborhood)","LEAF"
7,"attackerDeviceAccess(sonosSpeaker,deviceControl)","OR"
8,"RULE 51 (Exploit. The attacker on the wifi network launches exploit)","AND"
9,"vulExists(sonosSpeaker,'CVE-2018-11316',wifiAdjacentLogically,deviceControl)","LEAF"
10,"wifi(wifi1)","LEAF"
11,"inNetwork(sonosSpeaker,wifi1)","LEAF"
12,"attackerNetworkAccess(wifi1)","OR"
13,"RULE 49 (Exploit. The attacker launches exploit through the internet)","AND"
14,"vulExists(augustConnectWifiBridge,'CVE-2019-17098',wifiAdjacentPhysically,wifiAccess)","LEAF"
15,"inNetwork(augustConnectWifiBridge,wifi1)","LEAF"
16,"RULE 61 (Exploit. Use rooted device to inject wireless commands to devices on the same network)","AND"
17,"notEqual(hueBridge,sonosSpeaker)","LEAF"
18,"inNetwork(hueBridge,wifi1)","LEAF"
19,"exists(smoke)","OR"
20,"RULE 19 (Actuator-Physical. Oven triggers smoke)","AND"
21,"oven(samsungSmartOven)","LEAF"
22,"on(samsungSmartOven)","OR"
23,"RULE 32 (App. IFTTT Preheat your oven.)","AND"
24,"speaker(sonosSpeaker)","LEAF"
25,"receivesVoice(sonosSpeaker,'Preheat the oven')","OR"
26,"RULE 21 (Physical-Sensor. Speaker receives a voice command)","AND"
27,"voice('Preheat the oven')","OR"
28,"RULE 59 (Exploit. The attacker can inject arbitrary voice to the environment)","AND"
29,"canInjectVoice(attacker)","OR"
30,"RULE 78 (Exploit. Use controlled speaker to inject voice to the environment)","AND"
31,"receivesVoice(sonosSpeaker,'Alexa unlock the front door.')","OR"
32,"RULE 21 (Physical-Sensor. Speaker receives a voice command)","AND"
33,"voice('Alexa unlock the front door.')","OR"
34,"RULE 58 (Exploit. The attacker can inject arbitrary voice to the environment)","AND"
35,"reportsSmoke(nestProtect)","OR"
36,"RULE 20 (Physical-Sensor. Smoke detector reports smoke when there is smoke)","AND"
37,"smokeDetector(nestProtect)","LEAF"
38,"unlocked(augustDoorLock)","OR"
39,"RULE 33 (App. IoTCOM B4 Open door when smoke detected.)","AND"
40,"doorLock(augustDoorLock)","LEAF"
41,"RULE 34 (App. Alexia skill August Smart Home.)","AND"
```

Figure 3.2: IoT attack graph example. The meaning and type of each node is shown on the right.

```
vulProperty('CVE-2020-8864', wifiAdjacentLogically, rootPrivilege).
```

Listing 3.8: Prolog fact for an exploit model.

3.3.5 Attack Graph Generator

The Attack Graph Generator module takes the Prolog rule and fact file as input and verifies whether the attack goals (either provided by the administrator or automatically generated by IOTA) can be achieved. If a goal can be achieved, it will generate the attack graph showing all the attack traces; otherwise, the IoT system is protected from that attack goal. When the administrator knows his security objectives, he can set the attack goal by taking the logic NOT of the objectives. For example, if the objective is to protect the camera from being rooted, then the attack goal is the attacker's gaining root privilege on the camera. When there are no security objectives specified, we enumerate all the potential privileges attackers may get on all the devices as attack goals.

Figure 3.2 shows a small attack graph, where node 38 represents the attack goal — to unlock the doorlock. The meaning of each node is annotated on the right of the figure. In total, there are four *attack traces* (formally defined in Section 3.3.6) in the attack graph, and two of them are highlighted in red and blue. The attacker can reach node 7 (i.e., controlling Sonos speaker) by exploiting Hue bridge or August Wifi bridge. And from node 7, there are two ways to get to node 38: via the Alexa skill [93] (node 41) or by starting the oven to trigger smoke and using IoT app "IoTCOM B4" [94] (node 39).

Essentially, there are three kinds of nodes. The rectangle nodes represent *primitive facts* about the system state or the attacker state that are true before the exploit happens. The ellipse nodes represent Prolog *rules*, such as exploits or apps' execution. The diamond nodes stand for *derivation*, viz., new states about the system or the attacker after launching an exploit or executing an app. A derivation node can also be a precondition of another rule node. The logic meaning of each node is also annotated on the right of the figure. A detailed explanation of attack graph structure can be found in [15].

3.3.6 Attack Graph Analyzer

Because the generated attack graph can be gigantic, containing thousands of nodes, it is impractical to visualize the graph. Therefore, we propose two metrics to extract critical attack traces and quantify the impact of vulnerabilities.

Shortest Attack Trace. Among all of the attack traces to a specific attack goal, the *shortest attack trace* takes the minimum number of exploits and provides a lower bound of the attack complexity to that goal node. For instance, the shortest attack trace to the goal node (node 5) in Figure 3.2 is highlighted in red whose depth is 12. Below we formally define the shortest attack trace and relevant concepts.

Definition. (Attack Trace) Given an attack graph G, an attack trace to a derivation node n is a subgraph G' satisfying the following conditions: (1) Any OR node of G' has only one incoming edge; (2) Any AND nodes of G' has incoming edges from all its parent nodes; (3) All the source nodes of G' are primitive fact nodes; and (4) The sink node of G' is node n.

Definition. (Depth of an Attack Trace) The depth of an attack trace is the *longest path* from any primitive fact node to the sink node of the attack trace.

Definition. (Shortest Attack Trace) For a given attack graph and a derivation node n, the shortest attack trace is the attack trace to n with the smallest depth.

We cannot apply Dijkstra's algorithm to the shortest attack trace problem because our definition of shortest attack trace is different from the shortest path in graph theory: (1) There can be multiple source nodes; (2) The attack trace is a subgraph, not a path. Hence, we design a recursive algorithm, i.e., Algorithm 2, to compute the shortest attack trace to a specified attack goal node. The depth of a leaf node is defined as 0.

Blast Radius. The *blast radius* measures each vulnerability's impacts on the IoT system and can be used for system hardening. For example, if vulnerability A's blast radius is a superset of that of vulnerability B, we conclude that A's impact is bigger than B's, and therefore we should fix A first.

Algorithm 2: Shortest Attack Trace Algorithm

```
Input: (1) Attack graph $G$, (2) Attack goal node $n$
Output: Shortest attack trace to $n$ on $G$
Algorithm shortest_trace($G, n$)
    $res\_node$ = TraceNode($n$)
    if $n$ is leaf node then
        return $(0, res\_node)$
    end
    /* If current node is OR node, take the minimum of the parent nodes */
    if $n$ is OR node then
        Let $l$ be the list of parent nodes of $n$
        $min\_depth = \infty$
        for each node $m$ in $l$ do
            $(cur\_len, cur\_pred)$ = shortest_trace($G, m$)
            if $min\_depth > cur\_len$ then
                Set $res\_node.preds$ to $m$
                Update $min\_depth$
            end
        end
        return $(min\_depth + 1, res\_node)$
    end
    /* If current node is AND node, take the maximum of the parent nodes */
    if $n$ is AND node then
        Let $l$ be the list of parent nodes of $n$
        $min\_depth = -\infty$
        Set $res\_node.preds$ to $l$
        for each node $m$ in $l$ do
            $(cur\_len, cur\_pred)$ = shortest_trace($G, m$)
            if $min\_depth < cur\_len$ then
                Update $min\_depth$
            end
        end
        return $(min\_depth + 1, res\_node)$
    end
```

Definition. (Blast Radius (BR)) Given an attack graph, the blast radius of vulnerability v is the set of all of the privileges (represented as derivation nodes) the attacker gets after exploiting *only* v.

As there can be more than one trace to a certain node, and a vulnerability can be used in multiple attacks, we must keep track of vulnerabilities involved for each trace to a certain node in the attack graph. We come up with the following concepts to help us compute the blast radius of each vulnerability.

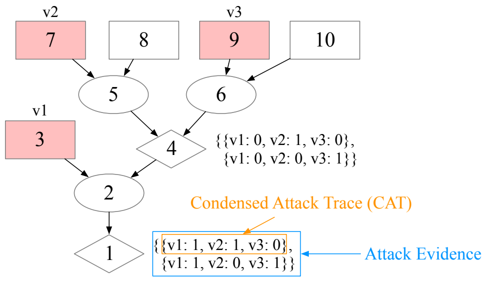


Figure 3.3: Example attack graph and the corresponding attack evidence for node 1 and node 4. Node 3, 7, and 9 are primitive fact nodes describing different vulnerabilities represented as $v1$, $v2$, and $v3$.

Definition. (Condensed Attack Trace (CAT)) Given an attack graph G, the condensed attack trace of a node n is the map from all of the vulnerabilities on G to 0 (when the vulnerability is not used) or 1 (when used) along some attack trace to n.

Definition. (Attack Evidence) The attack evidence of a node n is the set of its condensed attack traces.

Figure 3.3 illustrates an example attack graph and the corresponding attack evidences for node 1 and node 4. Since there are two attack traces to node 4 involving different vulnerabilities, the attack evidence for node 4 contains two elements, so is node 1. We compute the vulnerability evidence for each node in a forward fashion from leaf nodes to the goal nodes. Our merging algorithms are explained in Algorithm 3 and Algorithm 4 for OR and AND nodes, respectively. After getting the vulnerability evidence for each node, we can determine whether a derivation node should be included in some vulnerability's blast radius using Algorithm 5. The complete blast radius algorithm is given in Algorithm 6.

Algorithm 3: Attack Evidence Merge — OR

```
Input: Attack Evidence of two nodes: $a$, $b$
Output: Attack Evidence of the child node $c$, an OR node
Algorithm merge_ae_OR($a$, $b$)
    Let $merged\_ae$ be a copy of $a.ae$
    for $cat$ in $b.attack\_evidence$ do
        if $cat$ not in $a.attack\_evidence$ then
            $merged\_ae$.append($cat$)
        end
    end
    return $merged\_ae$
```

Algorithm 4: Attack Evidence Merge — AND

Input: (1) Attack evidence of two nodes: a, b, (2) $Vuls$: The set of all the CVEs on the attack graph

Output: Attack evidence of the child node c, an AND node

```
Algorithm merge_ae_AND($a, b, Vuls$)
    $merged\_ae$ = []
    for $cat1$ in $a.attack\_evidence$ do
        for $cat2$ in $b.attack\_evidence$ do
            /* Initialize $merged$, suppose $|Vuls| = p$ */
            $merged = \{v_1 : 0, \ldots, v_p : 0\}$
            for $vul$ in $Vuls$ do $merged[vul] = \max(cat1[vul], cat2[vul])$
            if $merged$ not in $merged\_ae$ then
                $merged\_ae$.append($merged$)
            end
        end
    end
    return $merged\_ae$
```

Algorithm 5: Determine Blast Radius

Input: (1) att_ev: attack evidences for all of the attack graph nodes, (2) $Vuls$: the map from $node\_id$ to the node's vulnerability evidence for all of the nodes

Output: The BR of each vulnerability in the attack graph

```
Algorithm determine_br($att\_ev$, $Vuls$)
    /* initialize $br$ */
    Let $br$ be an empty map
    for $vul$ in $Vuls$ do
        $br[vul] = \emptyset$
    end
    for $n$ in $att\_ev$ do
        if $n$.type is OR then
            for $cat$ in $att\_ev[n]$ do
                if sum($cat$.values()) == 1 then
                    Find the $key$ s.t. $cat[key]$ == 1
                    $br[key] = br[key] \cup \{n\}$
                end
            end
        end
    end
    return $br$
```

```
Algorithm 6: Blast Radius Algorithm
  Input: Attack graph $G$
  Output: Blast radius of each vulnerability in $G$
  Algorithm blast_radius($G$)
      /* Generate the list of unique vulnerabilities */
      $Vuls$ = []
      for $node$ in $G$ do
          if $node$ is a primitive fact node and $node$ describes a vulnerability $v$ then
              $Vuls$.append($v$)
          end
      end
      /* Initialize attack evidence for primitive fact nodes, suppose $|Vuls| = p$ */
      $queue$ = []
      for $node$ in $G$ do
          $node.cat$ = [$\{v_1 : 0, \ldots, v_p : 0\}$]
          if $node$ is a primitive fact node and $node$ describes a vulnerability $v$ then
              find the index $i$ such that $Vuls[i]$ = $v$
              $node.cat$ = [$\{v_1 : 0, \ldots, v_i : 1, \ldots, v_p : 0\}$]
              for $child$ in $node$.children do
                  if $child$ not in $queue$ then
                      $queue$.enqueue($child$)
                  end
              end
          end
      end
      /* Iteratively build attack evidence for all the nodes */
      while $queue$.length != 0 do
          $node$ = $queue$.dequeue()
          $cur\_ae$ = $node$.parents[0].cat
          for $i$ in 1 to length($node$.parents) do
              $next\_ae$ = $node$.parents[i].cat
              if $node$.type is AND then
                  $cur\_ae$ = merge_ae_AND($cur\_ae$, $next\_ae$)
              end
              else
                  $cur\_ae$ = merge_ae_OR($cur\_ae$, $next\_ae$)
              end
          end
          $node$.cat = $cur\_ae$
      end
      Let $att\_ev$ be the attack evidences for all nodes
      return determine_br($att\_ev$, $Vuls$)
```

Attack evidence provides a summary of vulnerabilities involved along each attack trace to a certain node and is useful for other important problems. For example, we can use it to compute the *minimal set of vulnerabilities to patch to thwart an attack goal* defined in [16]. We can count the occurrence of each vulnerability in the attack evidence and iteratively choose the vulnerabilities in descent order of occurrence; if the current vulnerability is in the same condensed attack trace of some chosen vulnerability, then we consider the next one. The iteration stops when all of the condensed attack traces contain at least one vulnerability chosen. For another application, if administrators assign numerical values as the *complexity* of each exploit, they can calculate the complexity of each attack trace by summing the exploit complexity for each condensed attack trace.

Severity. Even though a vulnerability's blast radius shows all of the privileges the attacker can get by exploiting only that vulnerability, it treats these privileges equally. However, this is usually not the case in reality. For example, the attacker's being able to unlock a doorlock is much more severe than turning on a smart light bulb. Hence, we come up with the metric of severity (also called severity score) for system administrator to sort the vulnerabilities if all the attack goals in an attack graph is known. The severity score of a node is a non-negative number representing its contribution to all of the attack goal nodes. The larger a node's severity is, the more it contributes. Our definition of severity is different from CVSS' impact score in that ours consider the impact of a vulnerability on all of the compromised system resources (represented as attack goals), whereas the latter considers the *immediate* impact. For example, if a CVE enables an attacker to get the root privilege on a device, then the CVSS' impact score represents this step, not considering any implied results, such as attacker using the rooted device to compromise another device and ultimately reaching his goal.

Computing the severity score for each node requires the following as input: (1) the probability of each node of the attack graph, and (2) the severity of attack goal nodes. The severity of a goal node can be converted from the importance of the resources it represents and is set by the administrator. For example, we set the severity of controlling a smart lock to 100, due to its significance for the smart home; in comparison, we set the severity of turning off the coffee machine to 5, since it is not important as an attack goal. The probability of each node of an attack graph is computed using [92].

The severity for each node is computed *backwards*, from attack goal nodes to the leaf nodes. Our formulas to back-propagate severity are shown in Figure 3.4. If node 0 is an OR node with severity S_0 (as the subfigure on the left), then the larger the probability of a parent node is, the more likely will that parent node lead to the OR node. Hence, we should assign a larger portion of S_0 to that parent node. In contrast, if

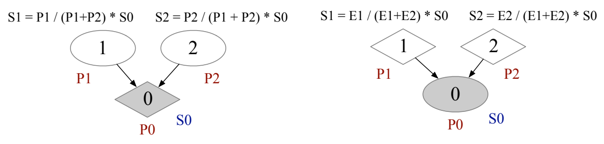


Figure 3.4: The formulas to back-propagate severity scores for AND and OR nodes. P_i and E_i ($E_i = -\log P_i$) mean the probability and entropy of node i, respectively.

node 0 is an AND node (as the subfigure on the right), all of its parent nodes must be true simultaneously. Thus the parent node having the smallest probability becomes the most critical condition and should get the largest portion of S_0. The formula can be generalized to the case where there are more than 2 parent nodes. Suppose node 0 has k parent nodes with probabilities P_1 to P_k, then the severity of parent node i is computed as $S_i = P_i/(P_1 + \cdots + P_k) \times S_0$. The formula for AND node can be generalized similarly. Our backward computation algorithm is summarized in Algorithm 7.

Algorithm 7: Severity Score Algorithm

```
Input: (1) Attack graph $G$, (2) A list of severity scores of all of the attack goal nodes
       $initial\_severity$
Output: The severity of each attack graph node
Algorithm severity($G, initial\_severity$)
    Let $set = \emptyset$ // it stores nodes to be processed next
    Add $initial\_severity$ nodes to $set$
    Let $num\_children$ as an empty dictionary
    Initialize $num\_children$ with keys being node ID and values being the number of children for
      each node in $G$
    /* Compute the severity for each non-goal node backward */
    while length($set$) != 0 do
        $node$ = $set$.pop()
        if $num\_children[node]$ != 0 then
            // current node is not ready for back-propagation
            continue
        end
        for $parent$ in $node.parents$ do
            $num\_children[parent]$ -= 1
        end
        if $node$.type is OR then
            Let $prob\_sum$ be the probabilities of nodes in $node.parents$
            for $parent$ in $node.parents$ do
                $parent$.severity += $node$.severity$\times$ $parent$.probability / $prob\_sum$
                if $num\_children[parent]$ == 0 then
                    // the parent node is ready for back-propagation
                    $set$.enqueue($parent$)
                end
            end
        end
        if $node$.type is AND then
            if a parent node is a primitive fact node representing individual vulnerability then
                Modify the parent node's probability to 0.9 // to avoid numerical issue
            end
            Let $parents\_entropy$ be the entropy of nodes in $node.parents$
            for $parent$ in $node.parents$ do
                $parent$.severity += $node$.severity$\times$ $parent$.entropy / sum($parents\_entropy$)
                $set$.enqueue($parent$)
                if $num\_children[parent]$ == 0 then
                    $set$.enqueue($parent$)
                end
            end
        end
        $procesed\_nodes$.add($node$)
    end
```

3.4 Implementation

The IOTA modules are implemented in Python using 2475 LoC. Physical dependencies and exploit rules are implemented in Prolog using 1179 LoC. The framework utilizes MySQL Connector Python library [1] for database operations and MulVAL [38] for attack graph generation.

Translator. The Translator module converts IoT system configuration and vulnerabilities to Prolog clauses. The initial **system configuration** (specified in JSON format) is sent to the Translator module to generate Prolog facts. The example system configuration and the translated results are shown in Listing 3.9 and 3.10, respectively.

```
{
    "devices": [
        {"name": "D-Link Router",
         "type": "router",
         "network": ["wifi1"]
        },
        {"name": "Smartthings Hub",
         "type": "gateway",
         "network": ["wifi1", "zigbee1"]
        }
    ],
    "networks": [
        {"name": "wifi1",
         "type": "Wifi"
        },
        {"name": "zigbee1",
         "type": "Zigbee"
        }
    ]
}
```

Listing 3.9: Example IoT system configuration JSON file.

The above JSON file lists the device and network settings of an IoT system. The device and network names are specified by the user, while device and network types use standard names predefined.

[1]https://github.com/mysql/mysql-connector-python

```
router(dLinkRouter).
inNetwork(dLinkRouter, wifi1).

gateway(smartthingsHub).
inNetwork(smartthingsHub, wifi1).
inNetwork(smartthingsHub, zigbee1).

wifi(wifi1).
zigbee(zigbee1).
```

Listing 3.10: Translated Prolog facts on system configuration.

IoT apps are first sent to the App Semantic Extractor and then translated to Prolog rules. Listing 3.11 is an example configuration of the SmartApp *Hall Light* explained in Section 3.3.3. The Translator combines parsed app semantic tuple (Listing 3.7) and app configuration (Listing 3.11) to generate Prolog rules shown in Listing 3.12.

```
{
  "apps": [
    {"App name": "Light on when I come home",
     "description": "Turn on the hall light if           there is motion and the
     door opens.",
     "device map": {
        "bulb": "Hue Wifi Bulb",
        "contact sensor": "Ring Contact Sensor",
        "motion sensor": "Mijia Motion Sensor"
      }
    }
  ]
}
```

Listing 3.11: Example IoT app configuration JSON file.

```
on(Bulb) :-
    bulb(Bulb),
    reportsMotion(MotionSensor),
    motionSensor(MotionSensor),
    open(DoorContactSensor),
    doorContactSensor(DoorContactSensor).
```

Listing 3.12: Prolog rule for IoT app *Hall Light: Welcome Home*.

Attack Graph Generator. The Attack Graph Generator concatenates exploit rules, indirect physical dependency rules, and the translated IoT app rules into a Prolog rule file. It also combines all the translated Prolog facts (including facts about device and network configuration and direct physical dependencies) and vulnerabilities (i.e., vulnerability existence facts and exploit model facts) into a Prolog fact file. The attack goals are also inserted into the Prolog fact file. After that, the rule and fact file are then sent to MulVAL [38] library to generate the Prolog reasoning log file and the attack graph.

3.5 Evaluation

3.5.1 Dataset

To generate attack graphs and conduct analysis, we need to obtain IoT system configurations, including device instances and IoT apps installed. Though thousands of IoT apps are available, how users choose apps and device instances to install is still unknown. To the best of our knowledge, currently, there is no public dataset of IoT systems configured by different users. Such information gap has long been a challenge to IoT system security research [42,94]. To evaluate IOTA, we generate synthetic IoT systems based on real-world IoT apps and device instances. We use a top-down approach to generate IoT systems by choosing the IoT app bundles first, as they determine the whole system's functionality. Once we have determined the IoT app bundle for the system, we create system instances by selecting device instances. To emulate the scenario where a user installs IoT devices but does not connect them to any IoT apps, we add individual IoT devices in one-third of the system instances created.

We consider SmartApps in the SmartThings Repository [69], and IFTTT applets [95] for SmartThings platform and create a pool of 532 IoT apps. We build a list of 59 smart home devices of 26 types, covering all of the device types listed on SmartThings Products List [96], from motion sensors, outlets to home appliances like TV, smart oven, etc. The devices are from 16 different platforms, all of which, except Roku, HP, and Aqara, are listed on Smartthing Partners [97]. In total, we create 37 IoT system instances. The first 18 instances are created based on the 6 app bundles used in [94] (which contains malicious apps), while the next 12 instances are generated based on 4 app bundles chosen from our app pool (which are treated as benign apps). The last 7 systems are of bigger size, with at most 50 devices, to further evaluate the scalability of our framework.

3.5.2 Results

Dependency Analyzer. For indirect physical dependency between different devices, our dependency analyzer module automatically detects the physical channels each device senses or modifies. We evaluate the effectiveness of the module using Google Home device descriptions [87], which specify the traits for each device type. We crawl the natural language descriptions of the traits and capabilities associated with different device types and run our module to identify the physical channels they interact with. The results are listed in Table 3.4. From the table, we can see that 7 out of the 11 physical channels are correctly predicted for different devices, except AC, heater, and light bulb. The AC and heater missed the humidity because there is no such keywords related to humidity in the trait descriptions. The light bulb has temperature as one of the channels it modifies. This is because the description of the `ColorSetting` trait uses the word "temperature" to represent color temperature.

Vulnerability Scanning. The vulnerability scanner queries CVE database with the full name of a given IoT device instance. The scanning result is shown in Figure 3.5, where the devices with the largest number of CVEs are illustrated. On average, there are 7.2 CVEs per device. Figure 3.5 shows that device types with the largest number of detected vulnerabilities are routers, cameras, and gateways. The reason could be that due to their pivotal position in IoT systems, security researchers tend to analyze these types of devices. We manually checked all CVE records and found out that 94.2% of them are relevant to the queried device. The typical devices and their CVEs identified are listed in Table 3.5.

We further verified the scanned CVEs with 12 real-world IoT devices and found out 5 of them still contain vulnerabilities: we obtained the exploit scripts for Philips Wifi Bulb, D-Link DCS-5009L Camera, and Eques Elf Smart Plug and successfully launched attacks against these devices. For Wemo Insight Smart Plug and Radio Thermostat, we confirmed the existence of the vulnerabilities by matching the firmware version of devices with the one in the vulnerability reports.

Vulnerability Analysis. We ran our vulnerability analyzer on 127 CVE records of smart home IoT devices collected by the Vulnerability Scanner module and manually checked the accuracy of the predicted exploit precondition and effects. The results are shown in Figure 3.6. Overall, our Vulnerability Analyzer achieves 80% and 88% prediction accuracy for precondition and effect, respectively. From Figure 3.6(a), we can see that the class `local` and `physical` have the highest accuracy, because the CVSS attack vectors for `physical` type is almost 100% accurate. And for low-power protocols, most of the time, the exploit range is local; hence, we can decide the `local` type with the help of protocol type. The precondition types

Table 3.4: Physical channels identified for different device types. **T**: temperature, **H**: humidity, **I**: illuminance, **V**: voice, **S**: smoke, **W**: water. ○ represents channels identified by our module, while ✓ represents channels confirmed by manual examination.

Device Type	T	H	I	V	S	W
AC	○					
	✓	✓				
Camera				○		
				✓		
Heater	○					
	✓	✓				
Humidifier		○				
		✓				
Light Bulb	○		○			
			✓			
Smoke Detector					○	
					✓	
Sprinkler						○
						✓
Speaker				○		
				✓		
TV				○		
				✓		

with the lowest prediction accuracy are `Adjacent physically` and `Adjacent logically`. This is because some CVEs' descriptions provide vague information for these two types.

According to Figure 3.6(b), the most accurate class is `root`. This is because there are multiple effective indicators, such as keywords like "root", "arbitrary", etc., the CVSS subscores (confidentiality, integrity, and availability subscore all being high), and the exploit mechanism like buffer overflow, or integer overflow, etc. With these indicators combined, our prediction is accurate. The high accuracy for both the precondition and effect prediction shows our module is highly effective.

Attack Graph Generation and Analysis. Table 3.6 illustrates analysis results for 10 IoT system instances from the 37 instances we built. The first column is the ID of the system. The first four rows are the analysis results for systems built based on app bundles used in [94], and the rest of the rows are results for

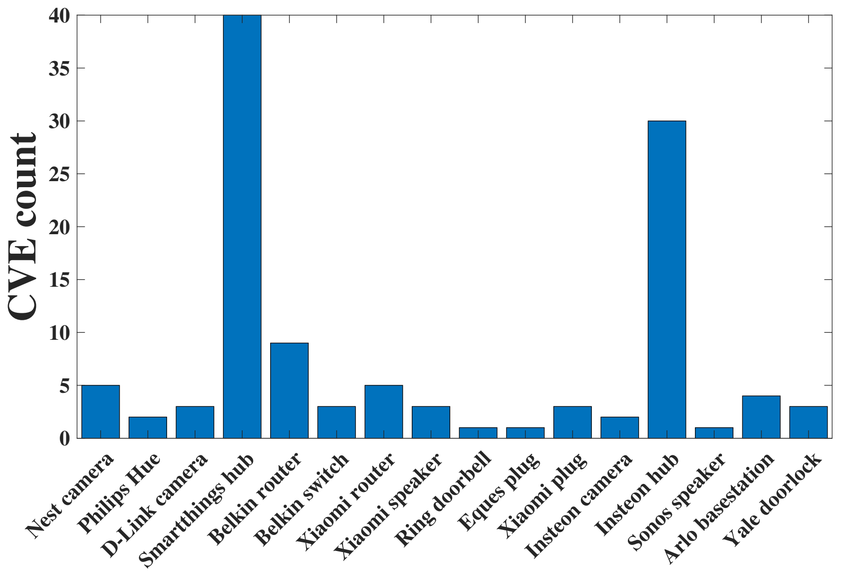


Figure 3.5: The number of CVEs scanned on IoT devices.

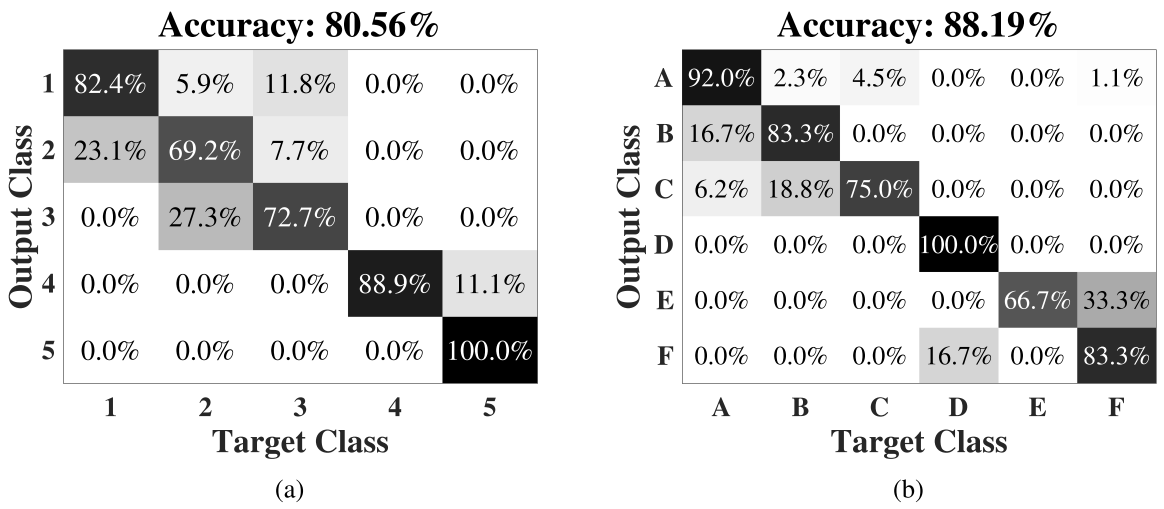


Figure 3.6: (a) Confusion matrix for exploit precondition identification. Label **1** to **5** denote preconditions listed in Table 3.2. (b) Confusion matrix for exploit effects. Label **A** to **F** represent exploit effects listed in Table 3.3.

systems built from our own app bundles. The **# CVEs** column is the number of vulnerabilities found on the given system. We enumerate all of the system resource compromises as potential attack goals, and the **# Goals** column denotes the number of attack goals shown on the attack graph, which can be achieved by the attacker for a given system.

Table 3.5: CVEs on typical IoT devices.

Device Instance	Typical CVE(s) Scanned
Hue Wifi Bulb	CVE-2019-18980
Hue Bridge	CVE-2020-6007
Nest Cam IQ Indoor	CVE-2019-5035, CVE-2019-5036, CVE-2019-5037
D-Link DCS Camera	CVE-2019-10999
Ring Doorbell	CVE-2019-9483
Yale Lock	CVE-2019-17627
August Bridge	CVE-2019-17098
Smartthings Hub	CVE-2018-3904, CVE-2018-3917, CVE-2018-3919, CVE-2018-3925
Xiaomi Gateway	CVE-2019-15913, CVE-2019-15914
Hue Bridge	CVE-2020-6007
Arlo Basestation	CVE-2019-3949, CVE-2019-3950
Sonos Speaker	CVE-2018-11316
Xiaomi Motion Sensor	CVE-2019-15913

Table 3.7 shows the distribution of the shortest depths of the attack traces to different goal nodes for 10 attack graphs in Table 3.6. From the figure, the largest portion (43.9%) of the attack traces have the shortest depths among 5 $\sim$ 8. To evaluate the effectiveness of the attack graphs, we manually check 27 shortest attack traces whose depths are at least 9. As a result, 62.8% of the attack traces revealed by IOTA are not anticipated by the system designers.

3.5.3 Case Study

Shortest Attack Trace. The shortest attack trace to the attack goal node "opening the window" in System 28 has a depth of 18. In this example, a physically adjacent attacker first exploits `CVE-2019-17098` on the smart lock gateway to sniff the home Wifi credentials. Then he exploits `CVE-2019-3949` on the camera basestation to control the indoor camera. After that, he utilizes the rooted camera to inject the voice command "preheat the oven" into the smart home, which is sensed by the smart speaker. The speaker triggers the IoT app to start the oven. The oven may trigger smoke, which is sensed by a smoke detector. Finally, another IoT app opens the window when smoke is detected.

Blast Radius. System 37 consists of 50 different devices, including all of the device types in Figure 3.5, and Wifi printer, smart TV, humidifier and toaster, etc. The vulnerability `CVE-2018-11314` identified on

Table 3.6: Attack graph analysis results on IoT system instances.

Sys ID	# Devices	# CVEs	# Nodes	# Edges	# Goals	Shortest depth*	CVE (|BR|)†	CVE (Severity)‡
1	4	3	12	15	1	6	CVE-2019-18980 (4)	CVE-2019-18980 (20)
4	6	4	37	39	5	(2, 8)	CVE-2020-6007 (5)	CVE-2020-6007 (27.2)
8	7	4	35	44	4	(4, 10)	CVE-2019-18980 (4)	CVE-2019-17098 (51.2)
11	7	3	25	26	9	(6, 16)	CVE-2018-3904 (9)	CVE-2018-3904 (115)
19	10	6	36	35	4	(2, 8)	CVE-2020-6007 (4)	CVE-2019-17627 (100)
26	12	7	117	173	19	(4, 10)	CVE-2020-6007 (5)	CVE-2019-17098 (166.9)
28	15	9	130	182	20	(4, 18)	CVE-2019-3949 (29)	CVE-2019-3949 (136.5)
32	23	11	131	186	23	(2, 8)	CVE-2018-3904 (12)	CVE-2018-11314 (136.8)
33	31	16	209	310	23	(2, 10)	CVE-2018-3904 (19)	CVE-2018-11314 (139.2)
37	50	28	338	577	43	(2, 14)	CVE-2018-11314 (32)	CVE-2019-5035 (100.6)

*: When there are multiple attack goals in attack graph, (x, y) means the min and max depth of the shortest attack traces to different goals.

†: |BR| is the cardinality of blast radius of the CVE. This column shows the CVE with the largest blast radius cardinality.

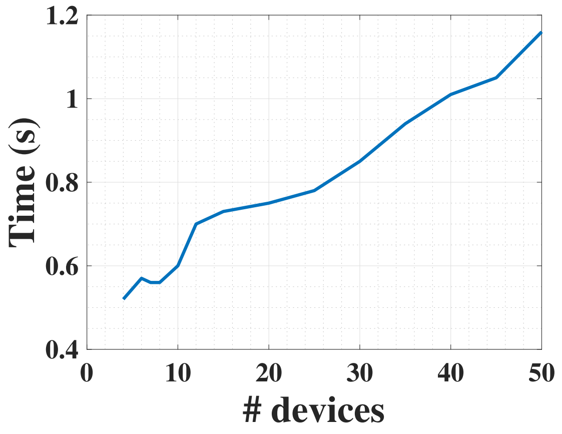
‡: CVE (Severity) is the CVE with the highest severity score. We specify initial severity scores for different attack goals ourselves.

Table 3.7: Distribution of the shortest depths for different attack goals. $\boldsymbol{d}$ means the depth of the shortest attack trace to an attack goal node.

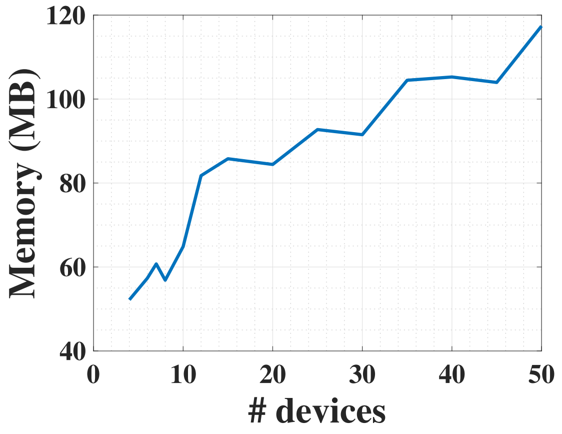
Shortest Trace Depth	Trace Count	Percentage
$d \leq 4$	51	36.7%
$5 \leq d \leq 8$	61	43.9%
$9 \leq d \leq 12$	24	17.3%
$13 \leq d \leq 16$	2	1.4%
$d \geq 17$	1	0.7%

the Roku TV has the largest blast radius, whose cardinality is 32. By exploiting `CVE-2018-11314`, the attacker on the internet can control the smart TV and play arbitrary video. System 37 has multiple voice-related IoT apps installed, such as turning on/off the light, turning on/off the humidifier, opening the window, and locking/unlocking the door if the smart speaker receives the corresponding voice command. As a result, after compromising the TV, the attacker can control those end devices by playing videos containing the voice commands. The attacker can further compromise physical environment features such as illuminance and humidity. The blast radius of `CVE-2018-11314` directly tells system administrators about all these compromises caused by this vulnerability.

Severity. System 19 consists of 10 devices, including contact sensor, temperature sensor, doorlock,



(a)



(b)

Figure 3.7: (a): CPU time vs IoT system size. (b): Memory usage vs IoT system size.

smart bulb, smart speaker, and gateways. According to Table 3.6, even though `CVE-2020-6007` has the largest blast radius, which contains 4 privileges which can be obtained by the attacker, these privileges are all about controlling the bulb and the room's illuminance. Because the system administrator assigns small severity to these compromises, the severity of `CVE-2020-6007` is 60.98. In comparison, `CVE-2019-17627`'s blast radius only contains attacker's controlling the doorlock. However, since this compromise is treated as severe, the administrator assigns 100 as its severity score. And the severity score gets back-propagated to `CVE-2019-17627`. We can see that severity scores can help the administrator evaluate the system-level impact of each vulnerability, thus prioritizing the fix.

3.5.4 Scalability

The time and memory complexity of our framework are shown in Figure 3.7. From the figure we can see that, in reality, it only takes around 1.2 seconds and 120MB of memory to generate the attack graph and perform attack graph analysis for an IoT system with 50 devices. The CPU time and memory consumption grow almost linearly with the number of devices. Our graph analysis algorithms will not asymptotically increase time complexity on top of the attack graph generation algorithm because the shortest attack trace and severity score algorithm only traverse the graph once. Though the time complexity of the blast radius algorithm is bounded by the sum of the number of traces to each node, in practice, this number is at the scale of $O(n^2)$ where n is the number of devices.

3.6 Summary

In this work, we design and prototype a novel framework, IOTA, for automatic, system-level IoT system security analysis. IOTA takes system configuration and CVE database as input and generates attack graphs showing all of the potential attack traces. Our framework further analyzes the attack graph by computing metrics, viz. the shortest attack trace, blast radius, and severity score, to help system administrators evaluate vulnerabilities' impacts. We create 37 synthetic smart home IoT systems based on 532 real-world IoT apps and 59 smart home devices and utilize our framework to analyze their security. Evaluation results show that IOTA is both effective (62.8% of the attack traces revealed are beyond system designers' anticipation) and highly efficient (it takes less than 1.2 seconds to analyze IoT systems of 50 devices).

Chapter 4

IoTGaze: IoT Security Enforcement via Wireless Context Analysis

4.1 Introduction

The rapid development of the Internet of Things (IoT) has an increasingly bigger impact on how we live and work. IoT technology enables interconnection, service automation, and other convenience in a variety of application scenarios, such as smart home, smart factory, and smart city, etc. By 2022, the number of connected IoT devices will reach to 29 billion [98]. The market value of IoT will reach to \$1.2 trillion in 2022 with a compound annual growth rate of 13.6% starting from 2017 according to the IDC prediction [99]. To increase their market share, different companies develop their IoT platforms for third-party developers to build apps to realize service provision automation. The popular IoT program platforms include Samsung's SmartThings [100], Apples' HomeKit [101] and Google Home [102].

Despite the exploding devices and fast growth of platforms of IoT, the security and privacy solution is not keeping the pace. Emerging vulnerabilities and attacks in IoT have brought tremendous loss. Within 20 hours, 65,000 IoT devices were rapidly infected and utilized to launch Mira attacks leading to internet outage [103]. By exploiting a major bug in the implementation of the *ZigBee* light link protocol, the attacker can use one single malicious bulb to turn off all the city lights [63]. Most critical security and privacy threats come from the IoT platforms and their affiliated apps. For instance, despite the Samsung SmartThings platform has a capability separation model, the apps can still request the capabilities that they do not need. The platform lacks effective means to audit the requests. The authors [62] found that 55% of SmartApp did not use all the rights to device operations that their requested capabilities implied, and 42% of SmartApps were granted capabilities that were not explicitly requested or used. Once gaining access to the capabilities, the malicious apps may not follow the user expectation and their app descriptions, resulting in serious security issues.

To relieve the security and privacy threats, the researchers propose solutions from different perspectives. By embedding extra code, FlowFence [104] and Soteria [105] can monitor the data flows and related control flows to prevent all the implicit flows from IoT apps via static program analysis. ContextIoT [56] uses the runtime logging to extract the essential context for building a context-based permission system. SmarthAuth [106] collects the security-relevant context information from analyzing IoT apps' source code, annotations, and descriptions. IoTGuard [107] dynamically collects the apps' information to enforce safety and security policies. However, these approaches require good knowledge about the program framework and app code. They have to modify the apps' source code or patch the apps and platforms to realize the discovery and prevention of threats. As can be seen, most existing solutions focus on the program analysis for platforms and apps. Then we come up with a question: *Can we open a new path to enhance the defense of IoT security and privacy?*

In this work, we look outside the IoT platforms and apps, and rethink the IoT security and privacy problems from the wireless perspective, and propose a new concept `Wireless Context` in IoT. Distinct from the program-based context, the IoT *Wireless Context* is inferred from the wireless communication traffic. We propose and implement a novel IoT security enforcement framework called IoTGaze that can detect potential anomalies and vulnerabilities in the IoT system. First, IoTGaze extracts the wireless packet features to correlate the communication traffic with the interaction of events between apps and devices. IoTGaze constantly sniffs the encrypted wireless traffic and generates the interaction event sequence. Second, IoTGaze discovers the temporal event dependencies and builds the *Wireless Context* for IoT system. Third, IoTGaze extracts the actual user expected *IoT Context* from IoT apps' descriptions and user interfaces (UI). By comparing the detected *Wireless Context* with *IoT Context*, IoTGaze can discover the anomaly in current IoT system. Lastly, by exploring the wireless event dependencies, IoTGaze is able to discover the unknown vulnerabilities that are caused by the inter-app interaction chain via hidden channels, such as light, temperature, humidity, etc., and can be exploited by the attacker to launch attacks against IoT system.

4.2 Threat Model

In this paper, we consider the security and privacy problems on the typical IoT interaction chain: devices, apps, and IoT platform. Based on the program framework, the developers write apps that request the capabilities access privilege from devices, and then control the devices to implement service automation. For instance, the description of one app is *"Turn on the indoor surveillance if the householder leaves home, otherwise turn off the indoor surveillance"*. The related IoT devices are presence sensor and surveillance camera. The security and privacy issues for the IoT system we want to detect are: **(a)** `App misbehavior`. For instance, when the household is at home, the app should turn off the surveillance to prevent privacy leakage. But the app may not turn off the surveillance and still monitor the activities of the household and uploads the data to somewhere else. **(b)** `Event spoofing`. The attacker may spoof a *"presence.not_present"* command to the IoT hub and the hub turns off the surveillance. Then the intruder could break into the house. **(c)** `Over-privilege`. The app may request irrelevant capabilities from the platform, such as the lock control privilege. Then the app may unlock the door when the household leaves home, which triggers serious security issues. **(d)** `Device failure`. The hardware flaws and software bugs may cause device failure. The attacker can also launch an attack to make the device (e.g., surveillance camera) unresponsive. **(e)** `Hidden vulnerabilities`. This type of vulnerabilities is caused by some hidden channels that multiple apps interact with simultaneously. Consider the scenario where one heater control app can automatically turn on the heater in winter after 8:00 PM, and another app opens the window automatically if the room temperature is higher than 90°. The indoor temperature is the physical channel, and the attacker can spoof a command event to let the heater keep working, leading to an increase of the temperature, and finally open the window and break in. We design a novel anomaly and vulnerability detection framework called IoTGaze that addresses the above security and privacy threats in IoT system.

4.3 System overview

In this section, we provide an overview of IoTGaze, and describe key components and workflow, as shown in Fig. 4.1.

Wireless Context Generation. The challenge here is how to use the sniffed raw wireless packets to generate the IoT `Wireless Context`. We decompose this problem into three subtasks: (1) *How to correlate the wireless traffic with the IoT interaction events and generate the fingerprints for the events?* We utilize limited features extracted from the encrypted wireless traffic and generate effective fingerprints

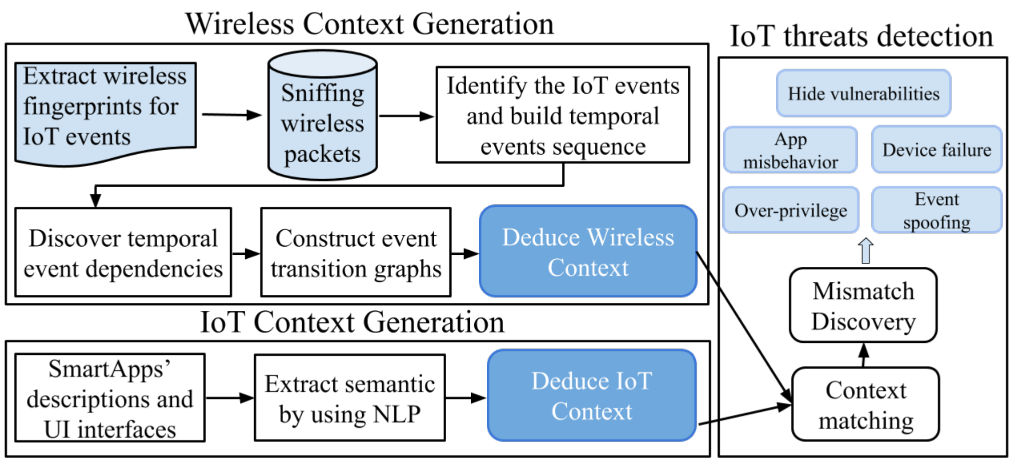

Wireless Context Generation
Extract wireless fingerprints for IoT events
Sniffing wireless packets
Identify the IoT events and build temporal events sequence
Discover temporal event dependencies
Construct event transition graphs
Deduce Wireless Context
IoT Context Generation
SmartApps' descriptions and UI interfaces
Extract semantic by using NLP
Deduce IoT Context
IoT threats detection
Hide vulnerabilities
App misbehavior
Device failure
Over-privilege
Event spoofing
Mismatch Discovery
Context matching

Figure 4.1: Overview of IoTGaze

to detect IoT events. (2) *How to use the sequential packets to generate the sequential IoT events?* What we sniff is the wireless packet sequence, but for vulnerability detection we should work on events. Thus, we design an approach to automatically segment the packet sequence and generate the temporal event sequence. (3) *How to discover the temporal event dependencies and generate the wireless context?* We design an event dependency discovery method that accurately extracts the event dependencies and their causal relationship for building the wireless context graph.

IoT Context Generation. The `IoT Context` we define here is the event interaction chain between smart apps (which run in the cloud and interact with devices via the IoT gateway such as SmartThings Hub) and devices. The IoT context is the user expected app behaviors, which may not be the real execution behaviors of apps. The malicious apps may deceitfully inform users about their functionalities but surreptitiously execute some malicious activities. To accurately extract the IoT context, we analyze the apps' descriptions and UIs that are directly exposed to users and usually cannot deceit users compared with program code. We extract the IoT context from app description and UI using NLP techniques and build the corresponding event transition graph that represents the app work logic expected by the user.

Anomaly and Vulnerability Detection. The IoT context represents the IoT automation services expected by the user, while the wireless context reveals what practical automation services are happening. Each context is expressed by a set of event transition graphs. We propose an approach to discover the mismatch and anomaly by comparing the event transition graphs under different contexts. By further analysis, we can discover the hidden vulnerabilities that are caused by the inter-app interaction and can be used by an attacker to launch attacks. Then we can prevent the attacks before they happen. Next, we describe these components of IoTGaze in detail in the following section.

4.4 Wireless Context Generation

We design and deploy a third-party *guardian* who *gazes* at the wireless communication traffic and detect potential anomalies and vulnerabilities. The *guardian* sniffs the encrypted wireless packets generated by the IoT activities and record the packets sequence $P = \{p_1, p_2, ..., p_i, ..., p_n\}$. Our goal here is to analyze the packet sequence and generate the wireless context. The wireless context is represented by a set of event transition graphs. In this section, we explain the procedures of wireless context generation in detail.

4.4.1 IoT Event Fingerprinting

Before generating the event sequence, we need to correlate the wireless traffic with the IoT events. We design the fingerprints to identify the IoT events. To realize service automation, event-driven smart apps receive data from various sensors (such as motion sensor, temperature sensor, and contact sensor) and issue commands to one or more actuators (e.g., smart bulb, smart power outlet, and smart lock, etc.) via the local IoT hub as the intermediary. We define IoT events as the activities that IoT hub interacts with sensors and actuators via wireless communication.

We use a packet sequence $P_{e_i} = \{p_1, p_2, ..., p_i, ..., p_{N_{e_i}}\}$ to represent the sniffed traffic for a unique event e_i. We extract the features from the packets' attributes except the encrypted data content to fingerprint the event. We list the features as following:(1) *Packet size*. A packet size could vary depending on what it transmits for which event. (2) *Packet direction*. A packet could be sent from the hub to a device or the opposite way. (3) *Packet interval*. The shorter packet interval signifies the higher transmission rate. Due to the difference of software and hardware, IoT devices may have distinct transmission rate, burst rate, response latency, and throughput, leading to varying packet interval. (4) *Packet layer*. Packets may be transmitted in different layers for a specific protocol. The above are common features across various wireless communication protocols, such as WiFi, Zigbee, Z-Wave, and Bluetooth lower Energy (BLE). Each protocol may have additional features. For example, The IP-based communication protocol may have features like IP source/destination address and source/destination port. By using the features set, we can generate the following fingerprint for a unique IoT event:

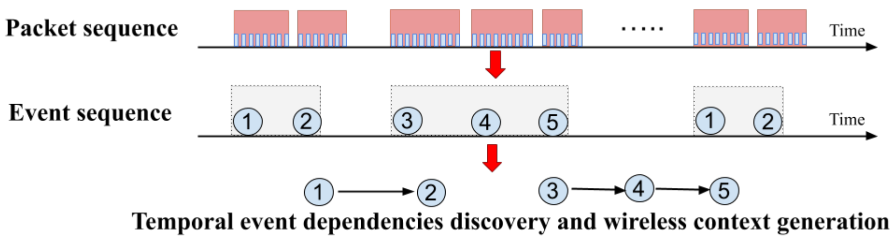

Packet sequence
Time
Event sequence
1
2
3
4
5
1
2
Time
1
2
3
4
5
Temporal event dependencies discovery and wireless context generation

Figure 4.2: Procedures of wireless context generation.

$$F_{e_i} = \begin{array}{c} \begin{array}{cccc} p_1 & \cdots & \cdots & p_{N_{e_i}} \end{array} \\ \begin{pmatrix} f_{1,1} & f_{2,1} & \cdots & f_{N_{e_i},1} \\ f_{1,2} & f_{2,2} & \cdots & f_{N_{e_i},2} \\ \vdots & \vdots & \ddots & \vdots \\ f_{1,\mathcal{M}} & f_{2,\mathcal{M}} & \cdots & f_{N_{e_i},\mathcal{M}} \end{pmatrix} \end{array}$$

where N_{e_i} denotes the number of packets transmitted for event e_i, and $\mathcal{M}$ denotes the number of features we extract for a specific communication protocol.

We collect and create the fingerprints data set for each event, and use the Random Forest supervised machine learning model as the classifier $\mathcal{C}$. The value of N_{e_i} varies depending on the event e_i. In order to feed the fingerprints matrix F_{e_i} into the same machine learning model, we fix the number of packets to $\mathcal{N}$ and pad the matrix with zero if N_{e_i} is less than $\mathcal{N}$. The optimal value of $\mathcal{N}$ will be discussed and selected in the later evaluation section. Then we use the classifier $\mathcal{C}$ to classify the new, unlabeled fingerprints and identify the events.

4.4.2 Sequential Events Generation

We analyze the sniffed packets sequence:

$$P = \{(p_1, t_1), (p_2, t_2), ..., (p_i, t_i), ..., (p_n, t_n)\}, \tag{4.1}$$

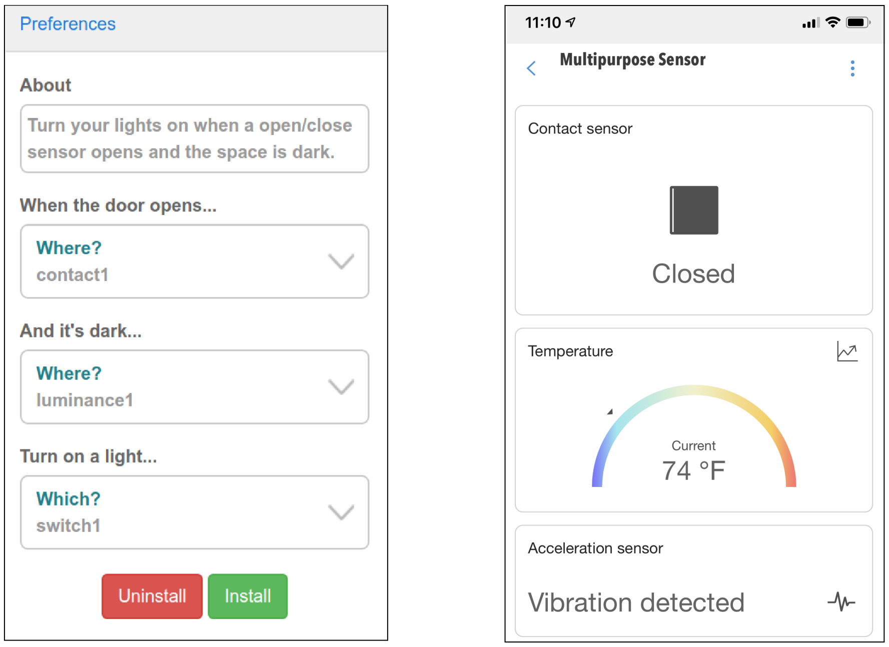
and identify the events using the generated fingerprints. Considering the packets for an event are sent within a short time. We use a sliding window with a maximum $\mathcal{N}$ packets within a fixed time interval of $\mathcal{T}$, and produce the matrix F. Then we feed the matrix F to the classifier $\mathcal{C}$ and output the probabilities for each event. If the maximum probability value predicted from event e_j is larger than the predefined classification threshold θ, then we think the packet sequence is created by event e_j. Otherwise, we will go to the next sliding window and continue to make the identification. The default step size is one, and the step size changes to the number of packets from event e_j once event e_j is detected.

4.4.3 Temporal Event Dependencies Detection

After identifying individual wireless events, we can construct the event stream $E = \langle (e_1, t_1), (e_2, t_2), ..., (e_n, t_n) \rangle$, where e_i ($i = 1, \ldots, n$) is the wireless event happening at time t_i. Notice that e_i and e_j ($i \neq j$) can be the same event happening at different time. A temporal event dependency means a set of events occur together with a chronological pattern. If event type a and b have a temporal dependency, then the time interval between them should follow a normal distribution $\mathcal{N}(\mu(a, b), \sigma^2)$ with σ being approximately equal to the standard deviation of the network delay. Thus, even though the expectation μ depends on the particular event type, the standard deviation is independent of event types. To determine if a and b are temporally dependent, we can collect all the samples of time interval between a and b from the event stream, and compute the sample standard deviation $\sigma(a, b)$ and compare it with the threshold τ (τ is a predefined parameter which is slightly larger than the standard deviation of network delay). If $\sigma(a, b) < \tau$, then we conclude a and b are temporally dependent, and vice versa.

Once we have identified all the pairs of events that are temporally dependent, we reconstruct the dependency sequence by concatenating these pairs. Formally, if $[a, b]$ and $[b, c]$ are dependent pairs, and $\mu(a, c) = \mu(a, b) + \mu(b, c)$, then we can get a dependency sequence $[a, b, c]$. Following such procedure, we iteratively check and concatenate sequences. In addition, we find that even if there is a dependency sequence $[a, b, c, d]$, $[c, d]$ itself could be a dependency sequence. We further discover these "subsequences of dependency sequences" using the number of occurrence in the input event stream. For example, if $[a, b, c, d]$ occurs 100 times, $[b, c, d]$ occurs 100 times, but $[c, d]$ occurs 150 times, then we know that $[b, c, d]$ is not a dependency sequence (since it is just a part of the dependency sequence $[a, b, c, d]$), but $[c, d]$ is a dependency sequence by itself and it occurs 50 times.

Generating Wireless Context. After discovering the event dependencies, we can build the event transition graph for each event dependency. As shown in Fig. 4.2, the event transition graph ① $\rightarrow$ ② represents a certain wireless context, such as *"If detecting a human presence, open the surveillance camera."*. The wireless context is extracted from the wireless traffic and can reflect the real activities of the currently installed apps. However, the wireless may not the expected IoT context from the user. We introduce the approach to extract the IoT context in the following section. If the wireless context violates the IoT context expected by the user, then it indicates potential anomalies in the current IoT system.


Preferences

About

Turn your lights on when a open/close sensor opens and the space is dark.

When the door opens...

Where? contact1

And it's dark...

Where? luminance1

Turn on a light...

Which? switch1

11:10

Multipurpose Sensor

Contact sensor

Closed

Temperature

Current 74 °F

Acceleration sensor

Vibration detected

Figure 4.3: (a) The installation interface of the SmartApp *Brighten-Dark-Places* in Samsung SmartThings platform. (b) The capabilities that one multipurpose sensor has for the Samsung SmartThings platform.

4.5 IoT context generation

In this section, we explain how to collect the IoT context that is the expected automation services from users. Due to the existing of malicious apps, the activities of smart apps cannot represent the actual IoT context. Some existing work conducts the static and dynamical analysis of the apps' code and checks if the actions of the program match what the apps describe. Instead of analyzing the apps' source code, we exploit the apps' description and UI that the apps usually do not tamper or spoof. Fig. 4.3(a) shows the installation interface of one SmartApp *Brighten-Dark-Places*. Based on the app description, the user chooses to install the app or not. Meanwhile, the needed capabilities are requested by the app, and the user needs to select the devices that can provide the capabilities. As we can see, the content in the installation interface is directly exposed to the user and tells the user what the app plans to do and is not usually tampered. If the user chooses to install the app, that means the app's description can reflect the user truly expected app service. If we know all the smart apps installed by the user, we can build the IoT context based on these apps' descriptions and UI. Now, we introduce the IoT context generation approach and implement it on Samsung SmartThings platform [100].

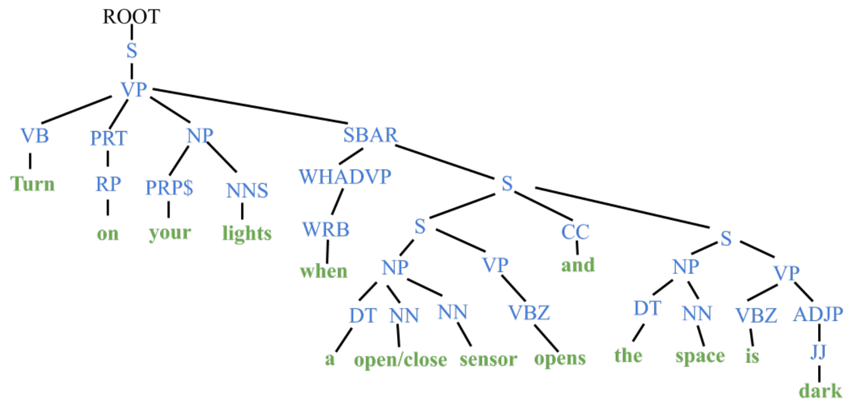


Figure 4.4: Stanford constituency tree representation of the description from the Brighten-Dark-Places SmartApp.

4.5.1 App Description Analysis

The research work [108, 109] has revealed that most IoT applications following the “If-This-Then-That" (IFTTT) programming paradigm, which can also be reflected by their apps’ description. The first step for analyzing apps’ behaviors is to obtain the causal relationship from the description. One effective method is to identify the conditional and main clause from the description. The conditional clause involves some sensors’ state change (e.g., the camera recognizes someone’s face), and the main clause involves some devices’ actions (e.g., unlock the door). Then, we can extract the related devices and their actions from the noun phrase and the verb phrase, respectively.

We use Stanford parser [110] to analyze the sentence structure of the app descriptions. To segment the description sentence into clauses, we build the constituency parse tree and split the sentence by label **S** (*Simple declarative clause*) or **SBAR** (*Clause introduced by a subordinating conjunction*). As shown in Fig. 4.4, the extracted three clauses are: “Turn on your lights", “a open/close sensor opens", and “the space is dark". The subordinating conjunction “when" signifies the causal relationship of the clauses via identifying the trigger and action. We analyze the dependency parse tree in Fig. 4.5 and extract the noun phrase and verb phrase from each clause. Considering the first clause as an example, “lights" is the accusative object of the verb “Turn" and this dependency is represented as *dobj*(“Turn", “lights"). But the extracted semantic may be human-readable and not machine-readable. We continue to do the capability matching process.

4.5.2 Capability Matching

The SmartApps interact with devices based on their capabilities. The capabilities have to be well decomposed in order to prevent over-privilege. The Samsung SmartThings platform maintains a capability

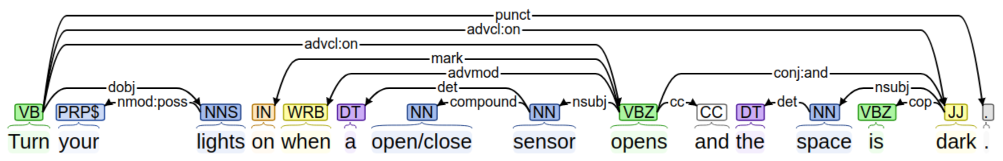


Figure 4.5: Stanford dependency tree representation of the description from the Brighten-Dark-Places SmartApp.

list [111] that SmartApps can request. Fig. 4.3(b) shows the multipurpose sensor has three capabilities that can provide to apps: *capability.contactSensor*, *capability.temperatureMeasurement*, and *capability.accelerationSensor*. Although we have extracted the app behavior from the description, there could still be a semantic gap between the wording of the description and the capabilities. Hence, we need to establish the relationship between the non-phrases in the description and the capabilities. The verb phrase in the same clause may also provide useful information for the matching and could also be considered during the matching. For example, “is dark" is more related to “illuminance" than “the space".

We match noun phrases and verb phrases to the capabilities based on the similarity score computed by the Word2Vec [112] model trained on the part of Google News dataset (about 100 billion words). Because the Word2Vec only gives embedding for words, we split every phrase into a tuple of individual words. This operation is also performed for capability names. We take the highest score of all the possible word pairs between a phrase tuple and a capability tuple as the similarity of these two tuples. Once we have the similarity score for each phrase and capability pair, we match the clause to the most similar capability. For each clause, if the most similar capability is already taken by some other capabilities, the second most similar one is chosen. Taking BRIGHTEN-DARK-PLACES SmartApp as an example, the matching result is “Turn on your lights" $\leftrightarrow$ *capability.switch*, “a open/close sensor opens" $\leftrightarrow$ *capability.contactSensor*, and “the space is dark" $\leftrightarrow$ *capability.illuminanceMeasurement*.

4.5.3 Event Transition Graph Generation

After extracting the app logic and matching the verb and noun phrases to the actual capabilities, we discover the commands from the verb phrases. For example, “Turn on" clearly indicates the capability command *capability.switch.on()*. We construct the SmartApp's behavior as an event transition graph where each node represents the capability command. The complete workflow for our example SmartApp is shown in Fig. 4.6. The final event transition logic is: *contactSensor.open*$\rightarrow$*illuminanceMeasurement* < *threshold*$\rightarrow$*switch.on()*, which shows the app work logic expected by user. We build the event transition graphs for all the SmartApps

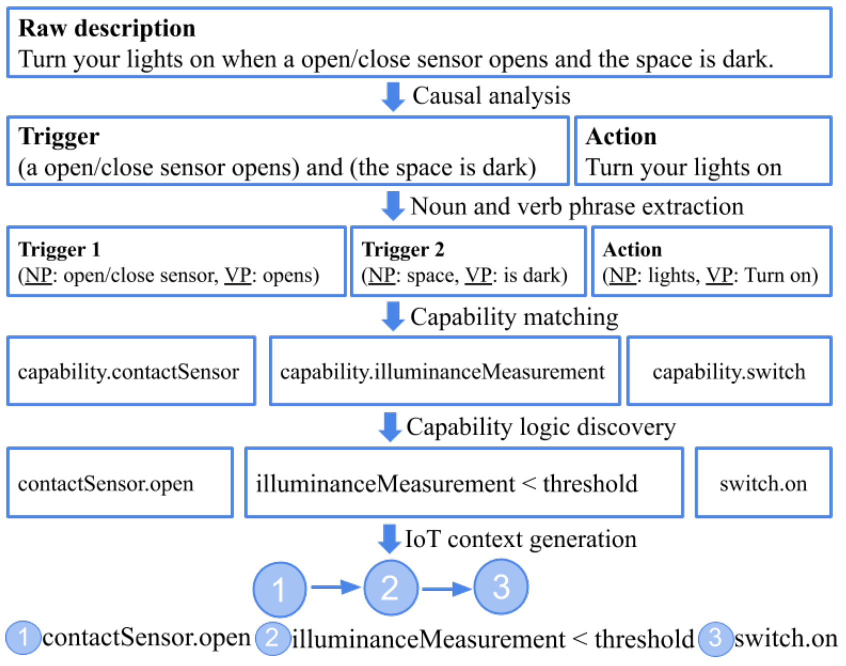

Raw description
Turn your lights on when a open/close sensor opens and the space is dark.
Causal analysis
Trigger
(a open/close sensor opens) and (the space is dark)
Action
Turn your lights on
Noun and verb phrase extraction
Trigger 1
(NP: open/close sensor, VP: opens)
Trigger 2
(NP: space, VP: is dark)
Action
(NP: lights, VP: Turn on)
Capability matching
capability.contactSensor
capability.illuminanceMeasurement
capability.switch
Capability logic discovery
contactSensor.open
illuminanceMeasurement < threshold
switch.on
IoT context generation
1
2
3
1 contactSensor.open
2 illuminanceMeasurement < threshold
3 switch.on

Figure 4.6: IoT context generation from the *Brighten-Dark-Places* SmartApp.

installed by a user, which represent the `IoT context` in the current system.

4.6 Anomaly and vulnerability detection

In this section, we introduce how to use the generated wireless context and IoT context to discover the anomalies and potential vulnerabilities in the IoT system. Each context is represented by a set of event transition graphs. We use $G = \{g_1, g_2, ..., g_i, ...\}$ and $G' = \{d_1, d_2, ..., d_j, ...\}$ to represent two sets of event transition graphs for IoT context and wireless context respectively. The nodes in the graphs g_i and d_j describe the corresponding IoT interaction events, which are represented by the unified capability commands. Meanwhile, all the events are numbered and given a global ID. The IoT context is the user expected app behaviors, and the wireless context is the actual app behaviors detected via the wireless communication traffic. If the wireless context violates the IoT context, that means potential anomalies and vulnerabilities. For each detected event transition graph d_j in the wireless context G', we check if we can find exactly the matched event transition graph g_i in the IoT context G. The match means the d_j and g_i should have identical event IDs and identical event dependencies. If there is no such match, we think there is a potential anomaly in the system.

The Fig. 4.7 provide the examples of the typical anomalies we can detect via our approach. The first IoT

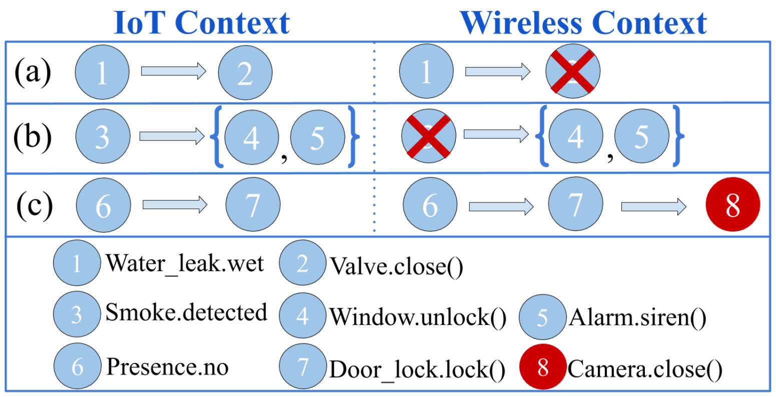


Figure 4.7: Discovery of various anomalies.

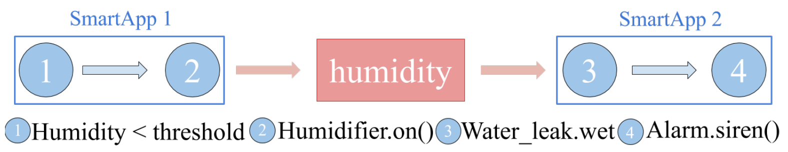


Figure 4.8: Discovery of hidden vulnerability.

context is "*If the water leak sensor detects the wet, close the valve*, which is represented by event transition *Water_leak.wet→Valve.close()*. For the wireless context, we only detect the first event and miss the second event. This anomaly could be caused by `device failure` or `app misbehavior`. The valve may not work due to its hardware flaws or software bugs. Also, the anomaly could be due to the app misbehavior. Once the app receives the wet alarm from the water leak sensor, it should send the command to close the valve. But from the wireless side, the app does not execute the second step. The second example is caused by `event spoofing`. Only when detecting smoke, the window is opened, and the alarm is triggered. But the attacker may spoof a fake smoke detected event and trigger subsequent actions. For the third example, we find an additional action *Camera.close()* is detected due to `over-privilege`. The actual IoT context is *If no presence is detected, lock the door.* The malicious app requests the non-necessary privilege for the camera and closes the camera after people leave the room. Then the attacker gets a chance to break in without the camera monitoring.

Furthermore, our approach can detect the potential vulnerabilities that are caused by the inter-app interaction chain. Most research work focuses on the local behaviors for one single app. They ignore the potential event interaction chain that crosses multiple apps. The chain is formed via some hidden channels, such as temperature, humidity, light, etc. The formed interaction chain could be leveraged by the attacker

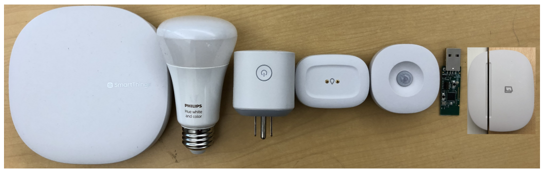


Figure 4.9: Smart devices and ZigBee packet sniffer used in our testbed for evaluating IoTGaze.

to launch attacks. Fig. 4.8 shows an example of how to discover the vulnerabilities via our approach. For wireless context, we detect a event chain ① → ② → ③ → ④. But we can only find the ① → ② and ③ → ④ in the IoT context. The first app opens the humidifier once the humidity is less than a threshold. Meanwhile, the humidity could influence the input of the second app. One malicious app can change the threshold and let the humidifier keeps working until triggering the water leak alarm. IoTGaze can detect such hidden vulnerabilities in advance and propose solutions to prevent such attacks.

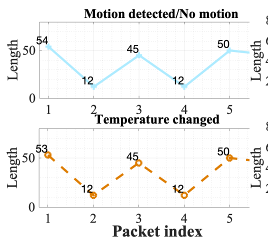
4.7 Evaluation

In order to demonstrate the feasibility and effectiveness of IoTGaze, we implement our framework on the Samsung SmartThings platform. Fig. 4.9 exhibits the IoT devices we use in our testbed. All the devices are connected to a SmartThings hub with ZigBee wireless communication protocol. We use TI CC2531 USB Dongle [113] and install the Zigbee protocol sniffer [114] to sniff the wireless communication traffic between hub and devices. A set of SmartApps is installed to SmartThings to enable the provision of automation services. We design extensive experiments from various aspects to thoroughly evaluate our approach.

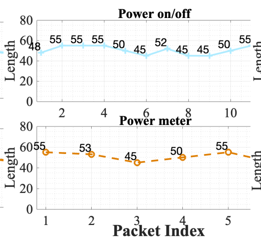
4.7.1 Anomaly Detection Evaluation

To verify the accuracy of our event fingerprinting approach, we use the five most commonly used IoT devices for our experiments: Motion sensor, Outlet, Water leak sensor, Philips Hue A19, and Multipurpose sensor. These devices can generate 19 types of events that can be found via the SmartThings iOS app. Each event corresponds to a SmartThings capability command. For example, Fig. 4.10(d) shows that the Philips Hue A19 can generate the following IoT events: *power on/off, color control, dimmer control,* and *color temperature control.* The corresponding capability commands are *switch.on(), switch.off(), colorControl.setColor(), switchLevel.setLevel(),* and *colorTemperature.setColorTemperature().* Our goal is to identify these events via sniffing the wireless packets.

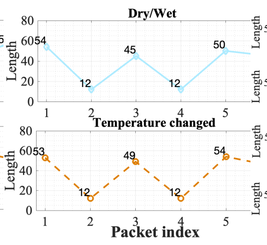
Event Fingerprinting Analysis. We collect the sniffed Zigbee wireless traffic and correlate them with



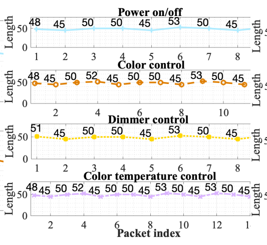
(a) Motion sensor



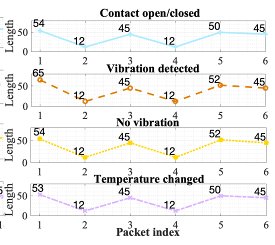
(b) Outlet



(c) Water leak sensor

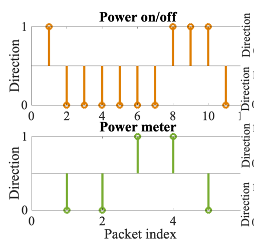


(d) Philips Hue A19

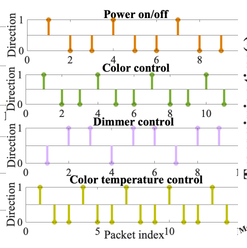


(e) Multipurpose sensor

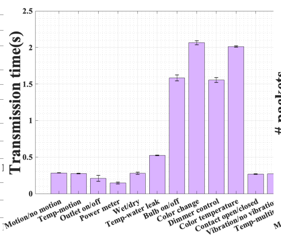
Figure 4.10: Packet size sequence from transmitted packets for different type events.



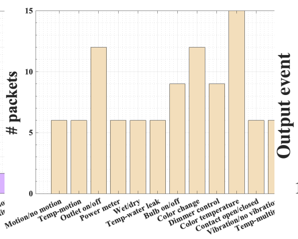
(a)



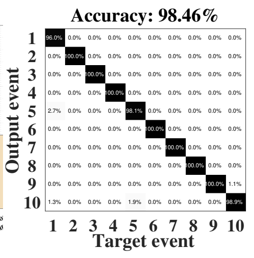
(b)



(c)



(d)



(e)

Figure 4.11: (a) Packet direction for outlet related events. (b) Packet direction for Philips Hue A19 related events. (c) # packets transmission for different events. (d) Transmission time for different events. (e) Confusion matrix for event identification.

the downloaded event history from the SmartThings app. Here are observations from the experiments: (1) For most events, the packet size sequences are distinct, as shown in Fig. 4.10. Although the event pair *motion detected/no motion*, *power on/off*, and *dry/wet* have the same packet length with different data fields, events in each pair are contrary to each other, which implies that we can use one variable to record the device's status to distinguish these events. (2) Although some sensors use identical capabilities for the same purpose, their packet sequence and size are still distinct. For instance, the motion sensor, water leak sensor, and multipurpose sensor can all detect temperature change. Once the temperate change is detected, they all send the same event via capability command *temperature.value*. However, their packet size sequences are distinguishable and can be used for fingerprinting, as shown in Fig. 4.10(a)(c)(e). (3) The direction of the packet in the sequence can also be used to distinguish some events. We use **0** to denote the direction from the device to the hub and use **1** for the reverse direction. Fig. 4.11(a)(b) show the packet directions for different events from outlet and Philips Hue A19, and verify the effectiveness of using packet direction as a fingerprint feature. (4) The inter-packet time interval and the transmission time can also be used to distinguish different events. The transmission time for each event is shown in Fig. 4.11(c), from which we can see that the shortest average transmission time is 0.1477s (for power meter event), while the longest average transmission time is 2.0656s (for color change event).

Event Collection and Model Training. To train the event classifier $\mathcal{C}$, we deploy the testbed in a typical office environment and continuously sniff and collect three weeks' wireless packet sequence and build the fingerprint matrix for each event. Fig. 4.11(d) shows that the maximum number of packets transmitted, for all kinds of events, is 15. So we set the parameter of $\mathcal{N}$ defined in section 4.4.1 to be 15. We label the data by matching the recorded event history in the SmartThings app. For devices that are triggered very infrequently in a real office environment, such as water leak sensor, we manually wet and dry it to generate sufficient event samples for the training. We train a random forest classifier using the event samples.

Event Detection Analysis. Based on the devices' capabilities in our testbed, we select the existing apps in the SmartThings Public Github Repository [115] and also develop our customized apps to build a total of 35 apps library and install them in the SmartThings platform. We constantly sniff the wireless traffic for another one week and identify the event in a real-time manner and generate the event sequence. We set the value of parameter $\mathcal{T}$ and θ in Section 4.4.2 to be 2.1s and 0.7 respectively. Before the identification, we remove the unrelated packets, including beacon packets, link-maintain packets, and acknowledgment packets. We still compare the detected events with the recorded events in the SmartThings hub and compute the detection accuracy. We select the ten most frequently occurred events and show their confusion matrix for classification in Fig. 4.11(e). The overall identification accuracy is 98.46%. The detection failure is mainly due to packet loss — the sniffer may miss some packets due to its limitation or other signal interference.

Wireless and IoT Context Discovery. After generating the sequence of the events, we mine the event dependencies using our algorithm and discover the wireless context. We successfully detect wireless context consisting of 35 event dependencies. The first eight items in Table 4.1 show part of the detected wireless context. We also use the proposed NLP approach to extract the IoT context from 35 apps installed, which exactly matches the detected wireless context. To further evaluate the applicability of our approach, we generate some complicated event dependencies, shown in the last four items in Table 4.1. We insert these events to the sequence of the already existing events and verify that we can still discover these complicated wireless context. The experimental results demonstrate the effectiveness of our IoT context and wireless context discovery.

Anomaly Generation and Detection. For the installed 35 apps, we design and insert malicious code to

Table 4.1: Anomaly detection via the discovery and comparison of IoT and wireless context.

No.	Event dependencies discovered in IoT context	Detected wireless context
1	$\boxed{\text{motion sensor}} \xrightarrow{\text{motion.active}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{Philips Hue}}$	~~①~~ → ②
2	$\boxed{\text{multipurpose sensor}} \xrightarrow{\text{temperature.value}} \boxed{\text{hub}} \xrightarrow{\text{colorControl.setColor()}} \boxed{\text{Philips Hue}}$	~~③~~ → ④
3	$\boxed{\text{outlet}} \xrightarrow{\text{power.value}} \boxed{\text{hub}} \xrightarrow{\text{switch.off()}} \boxed{\text{outlet}}$	~~⑤~~ → ⑥
4	$\boxed{\text{water leak sensor}} \xrightarrow{\text{water.wet}} \boxed{\text{hub}} \xrightarrow{\text{switch.off()}} \boxed{\text{outlet}}$	⑦ → ~~⑥~~
5	$\boxed{\text{multipurpose sensor}} \xrightarrow{\text{acceleration.active}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{Philips Hue}}$	⑧ → ~~②~~
6	$\boxed{\text{multipurpose sensor}} \xrightarrow{\text{contact.open}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{Philips Hue}}$	⑨ → ~~②~~
7	$\boxed{\text{multipurpose sensor}} \xrightarrow{\text{contact.close}} \boxed{\text{hub}} \xrightarrow{\text{switch.off()}} \boxed{\text{outlet}}$	⑨ → ⑥ → ②
8	$\boxed{\text{motion sensor}} \xrightarrow{\text{motion.inactive}} \boxed{\text{hub}} \xrightarrow{\text{colorControl.setHue()}} \boxed{\text{Philips Hue}}$	① → ⑩ → ②
9	$\boxed{\text{hub}} \xrightarrow{\text{switch.off()}} \boxed{\text{bulb}}, \boxed{\text{hub}} \xrightarrow{\text{lock.lock()}} \boxed{\text{lock}}, \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{camera}}$	⑪ → ⑫ → ~~⑬~~
10	$\{ \boxed{\text{multipurpose}} \xrightarrow{\text{contact.open}}, \boxed{\text{illuminance sensor}} \xrightarrow{\text{illuminance.value}} \} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{bulb}}$	{ ~~⑨~~, ⑭ } → ⑪
11	$\boxed{\text{multipurpose}} \xrightarrow{\text{temperature.value}} \boxed{\text{hub}} \{ \xrightarrow{\text{colorControl.setColor()}} \boxed{\text{Hue}}, \xrightarrow{\text{switch.on()}} \boxed{\text{heater}} \}$	③ → { ~~④~~, ~~⑮~~ }
12	$\{ \boxed{\text{thermostat}} \xrightarrow{\text{presence.not\_present}}, \boxed{\text{multipurpose}} \xrightarrow{\text{contact.closed}}, \boxed{\text{lock}} \xrightarrow{\text{lock.unlocked}} \} \boxed{\text{hub}} \xrightarrow{\text{lock.lock()}} \boxed{\text{lock}}$	{ ⑯, ⑰, ⑱ } → ~~⑫~~

Table 4.2: Anomaly detection results for three types of anomaly.

	Spoofing	**Overprivilege**	**Misbehavior**
Precision	97.22%	98.55%	98.29%
Recall	94.82%	98.36%	95.20%

the apps to generate anomalies. For each app, we modify its code to generate the following three types of anomalies: (a) Event Spoofing. We add the code in the app to spoof some events for triggering purpose. The first three items in Table 4.1 show the examples, such as event sequence changing from ① → ② to ~~①~~→ ②. (b) App Misbehavior. For the item 4-6 in Table 4.1, we modify the apps' code and make the app *not* execute the triggered actions, leading to the event sequence change from ⑦ → ⑥ to ⑦ → ~~⑥~~. (c) Over-privilege. The item 7-8 show the anomaly samples we generate. We modify the code to request the non-necessary capabilities and execute the addition actions, such as changing from ⑨ → ⑥ to ⑨ → ⑥ → ②. For each app, we generate 100 anomalies for each threat type and try to detect them via our approach.

The results of anomaly detection are shown in Table 4.2. We can see that the detection for overprivilege has both high precision and recall. This is because the overprivileged event sequence is different from all of the normal sequences, so the precision and recall are just limited by the success rate of event detection. Spoofing and misbehavior can occasionally generate event sequence which match the normal app behavior. Therefore, their precision is high but recall is low.

4.7.2 Hidden Vulnerabilities Discovery

The current research mostly focuses on security vulnerability detection per SmartApp. The local behaviors of one single app may explicitly or implicitly affect the whole IoT system. The potential interactions between apps, devices, and the environment may produce vulnerabilities that cannot be discovered by per-app analysis. We propose to use the wireless context to discover the hidden vulnerabilities that also can be exploited by attackers.

In wireless context, we can find some wireless event dependencies spanning multiple applications. This is because these applications are somehow *correlated* together via some hidden channels. We thoroughly investigate the 183 apps in the SmartThings Public GitHub Repository [115] and analyze their interactions with other apps, devices, and the environment. We find that there are three kinds of channels that can cause potential vulnerabilities: (1) **Capability**. Two applications can interact if the first app's output is the trigger of the second app. For example, the SmartApp "NFC Tag Toggle" in the official SmartThings GitHub allows toggling of a switch, lock, or garage door. And another SmartApp "Door State to Color Light (Hue Bulb)" changes the color of Hue bulbs based on the door status. In this example, the two apps are directly chained via the capability *doorControl*. (2) **Physical channel**. The environment elements can be changed due to the input or output of some apps and cause potential interaction chains. We take one physical channel *smoke* as an example. The toaster may cause smoke, which makes alarm siren. (3) **System channel**. Some global variables in the IoT program framework may be shared by some SmartApps. The *location.mode* in SmartThings platform enables the devices to behave differently in different scenarios. For example, if the current location.mode is "Home" and the motion sensor detects motion, then turn on the light. But if the location.mode is "Away" and the motion is detected, then turn on the camera. All these three kinds of channels can generate unexpected application interaction, rendering system vulnerabilities.

We discover and provide the list of all the hidden vulnerabilities for each type of channel from the SmartThings platform. Based on the NLP approach in Section 4.5, the capabilities related to the apps' input and output are extracted. We list seven capabilities that are shared by apps in Table 4.4. The capability *switch(light)* generates 127 potential inter-app interaction chains and has the highest risk score. We use Word2Vec [112] to establish the mapping between physical channels and apps' input and output. In total, nine physical channels are discovered, as shown in Table 4.4. The illuminance, energy, and temperature are the channels that bring the most of inter-app interactions. When we program the malicious apps, we show that system variable *location.mode* from SmartThings program platform *location.mode* are frequently used

Table 4.3: Hidden vulnerabilities discovery via analyzing wireless context.

No.	Event dependencies discovered in wireless context
1	$\boxed{\text{time}} \rightarrow \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{heater}} \rightarrow \boxed{\textbf{temperature}} \rightarrow \boxed{\text{temperature sensor}} \xrightarrow{\text{temperature.value}} \boxed{\text{hub}} \xrightarrow{\text{window.open()}} \boxed{\text{window}}$
2	$\boxed{\text{temperature sensor}} \xrightarrow{\text{temperature.value}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{fan}} \rightarrow \boxed{\textbf{motion}} \rightarrow \boxed{\text{motion sensor}} \xrightarrow{\text{motion.active}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{light}}$
3	$\boxed{\text{water leak sensor}} \xrightarrow{\text{water.wet}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{light}} \rightarrow \boxed{\textbf{illuminance}} \rightarrow \boxed{\text{illum sensor}} \xrightarrow{\text{illuminance.value}} \boxed{\text{hub}} \xrightarrow{\text{windowShade.close()}} \boxed{\text{window shade}}$
4	$\boxed{\text{presence sensor}} \xrightarrow{\text{presence.not\_present}} \boxed{\text{hub}} \xrightarrow{\text{lock.lock()}} \boxed{\textbf{lock}} \xrightarrow{\text{lock.lock}} \boxed{\text{hub}} \xrightarrow{\text{colorControl.setColor()}} \boxed{\text{Hue}}$
5	$\boxed{\text{presence sensor}} \xrightarrow{\text{presence.present}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\textbf{bulb}} \xrightarrow{\text{switch.on}} \boxed{\text{hub}} \xrightarrow{\text{camera.take()}} \boxed{\text{camera}}$
6	$\boxed{\text{multipurpose sensor}} \xrightarrow{\text{temperature.value}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{AC}} \xrightarrow{\text{switch.on}} \boxed{\text{hub}} \xrightarrow{\text{switch.on()}} \boxed{\text{bulb}}$
7	$\boxed{\text{presence.not\_present}} \xrightarrow{\text{presence.not\_present}} \boxed{\text{hub}} \rightarrow \boxed{\textbf{location mode}} \rightarrow \boxed{\text{hub}} \xrightarrow{\text{switch.off()}} \boxed{\text{light}}$

Table 4.4: Statistics of hidden channels identified from official SmartApps.

Channel Type	Channel	# Apps related	# Interaction chains
Capability	swtich(light)	28	127
	doorControl	4	4
	lock	8	22
	switch(heater)	10	27
	switch(AC)	9	23
	colorControl	7	6
	thermostat	9	20
Physical	leakage	4	5
	illuminance	29	132
	energy	36	134
	contact	20	37
	acceleration	9	18
	smoke	10	17
	temperature	18	127
	motion	14	13
	humidity	4	3
System	location.mode	9	16

and modified by some apps, which can also cause security-relevant issues. We list the vulnerabilities found for each type of channel in Table 4.3 and show the statistics about vulnerabilities in Table 4.4.

4.8 Summary

In this paper, we propose a novel IoT anomaly detection framework called IoTGaze. Instead of exploring the threats inside platform and apps, we deploy a third-party monitor IoTGaze, who gazes at the wireless traffic and detects the potential threats in the IoT system via analyzing the encrypted wireless packets. We propose a new concept called `wireless context` in IoT that represents the observed app logic from wireless sniffing. We design a fingerprinting based event detection approach and use it to generate the event sequence via sniffed wireless packets. We design an algorithm to discover the temporal event dependencies and build the wireless context. We also extract the IoT context that reflects user expected app behaviors via analyzing apps' descriptions via natural language processing techniques. By matching the wireless and IoT context, we can detect the anomalies that are happening in the IoT system. Furthermore, the event dependencies discovered by IoTGaze can reveal some potential vulnerabilities that are caused by the inter-app interaction via some hidden channels. We prototype our approach on the Samsung SmartThings platform and demonstrate the feasibility and effectiveness of IoTGaze.

Chapter 5

IoTSpy: Uncovering Human Privacy Leakage in IoT Networks via Mining Wireless Context

5.1 Introduction

With the advent of IoT techniques, the way we live has changed radically. Numerous IoT devices and apps can be deployed in our house to implement any automation services you expect. Smart Home is driven by massive amounts of data collected by devices installed in the house. These devices may collect a user's state data and environment data and upload them to the cloud for further processing, potentially compromising the user's privacy. A user does not know if the privacy data is uploaded to the correct location on the Internet. Although the data is uploaded to the authenticated Internet location, a user cannot control how the data is used and prevents data abuse.

For the current IoT system, user privacy can also be leaked in other aspects. In BLE network, IoT devices periodically broadcast advertisement packets to announce their presence. Randomized MAC address is introduced to prevent device tracking. However, the work [116] defeats address randomization and exploits the asynchronous nature of payload and address changes from advertising packets to achieve tracking. Apple uses BLE as the communication protocol between iOS and macOS to support interoperability. But, the researchers [117] leverage the unencrypted data fields to reverse engineer the message types for continuity protocol and behaviorally profile users.

Although IoT communication traffic is encrypted, it can still leak user privacy by correlating the traffic patterns with user behaviors. The researchers [118] investigate four IoT devices - a Sense sleep monitor, a Nest Cam Indoor security camera, a WeMo switch, and an Amazon Echo, and infer user's activities via analyzing the networking traffic. For instance, By observing the traffic from Amazon Echo, we can detect if a user is interacting with an intelligent personal assistant. But, simply mapping the traffic with user activities can only disclose some essential user privacy.

In this paper, we explore human privacy leakage issues in IoT communication. Our work explores the more deeply human privacy from encrypted wireless traffic, instead of simply mapping the wireless traffic patterns with user activities. We thoroughly investigate all the user activities that can be leaked and divide them into four groups by difficulty. The existing work can only reveal the user activities with the lowest difficulty. Our approach can infer more crucial user privacy that has not yet been discovered by current research work [119, 120] and seriously do threaten human security.

In this work, we investigate to mine the `wireless context`. The wireless context represents the regular IoT activities and is reflected by the temporal dependencies of IoT events. The wireless context can be used to infer more complex user activities and disclose some critical user privacy. For instance, by deploying a wireless sniffer outside a user's house, our approach can infer the user's daily routine and living habits. We can also infer the applications' behavior installed by the user, which can be exploited by the attacker to breach the IoT system. We present all the possible user privacy that can be leaked via encrypted traffic sniffing and analyze the potential threats to the user's security and privacy.

5.2 User Privacy Leakage Model

In this paper, we consider the user privacy leakage problem in a typical smart home scenario. We can deduce and get to know different kinds of user privacy by sniffing the wireless packets transmitted from the interaction between IoT devices and gateway. The typical IoT wireless communication protocols encrypt the data payload fields. So we cannot easily fetch the sensor value or commands that IoT devices transmit and receive. Our goal in this paper is not to hack the data payload fields but to utilize the accessible packet fields that are not encrypted, such as packet header, to fingerprint IoT events. For instance, when a thermometer reports the change of temperature value, we do know the temperature is changed but do not know what the changed value is. The main challenges of our work can be divided into two steps: **(a)** *How can we use the limited packet information to fingerprint and detect the IoT events?* **(b)** *How can we use the detected IoT events to deduce the user privacy?*

In this section, we mainly discuss all kinds of user privacy that can be potentially leaked from the encrypted traffic. The technical detail is provided in the next section. Based on the difficulty of privacy inference, we divide the user privacy leakage into the four levels.

(1) Direct User Activity Leakage. We utilize the IoT wireless event to infer human activity. Some IoT events can directly reveal what a user is doing without further deduce. For instance, suppose the user installs one app *Coffee After Shower* that implements the automation service *"Turn on your coffee machine while you are taking a shower"*. When the humidity and thermometer sensors installed in bathroom detect the value changed, they transmit the sensed value to the IoT gateway. Then the sensed data is uploaded to the app in the cloud. Once the moisture and temperature value exceed the threshold, the app turns on the coffee machine via issuing command transmitted from the IoT gateway to coffee machine. So there are two `IoT events`: **(a)** humidity and thermometer sensors report *sensed data* to IoT gateway. **(b)** IoT gateway sends *turn on* command to coffee machine. These two IoT events are executed via the transmission of wireless packets. The second event can directly reflect the user activity "*turn on coffee machine*". The first event cannot directly reveal user activity. Actually, most IoT events triggered by the interaction between actuators and IoT gateway can directly infer user activity.

(2) Indirect User Activity Leakage. Some IoT events cannot directly reveal the user activity. For instance, most IoT sensors report their sensed data value to the cloud apps via the gateway. As mentioned before, we do not aim to hack the encrypted data payload field. So we can only know the occurrence of event *the sensor reports a value"*, but cannot obtain the sensed data value. However, we can still deduce user activity via limited event information. We still use the app *Coffee After shower* as an example. The first IoT event cannot disclose the real temperature and moisture value. But we can still deduce that the user is taking a shower in the bathroom. Most sensors use the on-change reporting mode. Events are generated only if the measured values have changed. For this example, if the humidity and thermometer sensors frequently report the measured values changed within a short time period, we can deduce that the user is taking a shower in the bathroom. We even do not need to know if the measured value is increased or decreased.

Furthermore, we can even obtain user's other privacy information based on the detected IoT events, such as **user's mood**. There is an app named *"Hue Mood Lighting"* in the official list of SmartApps from Samsung SmartThings. The app implements the functionality of *"Sets the colors and brightness level of your Philips Hue lights to match your mood"*. From the wireless perspective, we can detect the IoT event that *"IoT gateway sends color and brightness change commands to Philips Hue lights"*. But we cannot know the color and brightness level that the user sets. However, we can still deduce the user's mood by analyzing the frequency of the user changing the color and brightness. When people have stable emotions, they usually

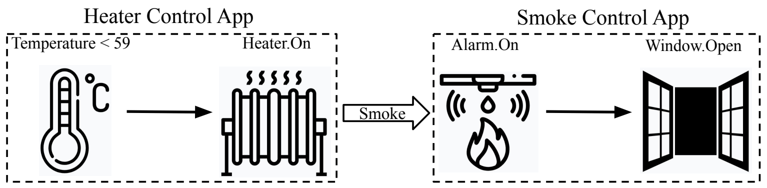


Figure 5.1: An Example of Attack Chain from Inter-app Physical Interaction

set the color and brightness they prefer and will not change the setting for a long time. But if a person is suffering from insomnia or irritability, they may frequently change the color and brightness of light in a short time. Therefore, based on the properties of some IoT events, such as the frequency of occurrence, we can deduce users' privacy information, such as "user has stable emotion or not."

(3) User Living Habits and Routine Leakage. The aforementioned user privacy can be inferred from one single IoT event or multiple IoT events from the same event type. From the time dimension of view, if we continuously sniff the wireless packets and detect the IoT events, we can build the IoT event sequence. Based on the sequential IoT events, we can mine the temporal event dependency and obtain corresponding user activity dependency. The user activity dependency can reflect the user's living habits or routine. For example, people usually have a routine when they wake up, such as *take a shower, play music, turn the coffee machine, turn on the toaster, lock the door, and leave home for work*. For these user activities, if their corresponding IoT events frequently appear together in the chronological order, we can infer that the user has a wake-up routine. If all the IoT event dependencies are discovered, the user's living habits can be completely exposed to the eavesdropper. The eavesdropper can even predict the user's future activities, which may cause disastrous results.

(4) User Installed IoT App Logic Leakage. The smart home system is actuated by IoT apps. The install apps also reveal the user's privacy. By mining the dependency of the IoT events, we can infer the corresponding installed app logic. As mentioned above, there are two IoT events triggered by the *Coffee After Shower* app. These two IoT events frequently appear together in chronological order. We can detect the dependency from time dimension and deduce the corresponding app logic.

The inferred app logic can not only leak a lof of user privacy but also cause serious security problems. When the attackers know the installed app logics, they can launch some attacks to breach the system. Suppose one app implements the automation service *"unlock the door if smoke is detected"*. If the attackers know the app logic, they can launch an event spoofing attack and fake a smoke detected event. Then the

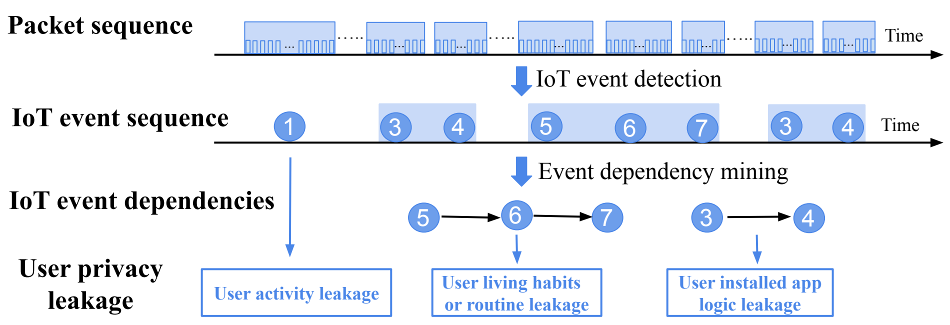


Figure 5.2: Overview of user privacy leakage.

door is automatically unlocked, and the attack can break in. Furthermore, if the attackers can infer the app logics from multiple apps, they can launch more advanced attacks through the inter-app physical interaction chains [82]. Fig. 5.1 shows an example of inter-app physical interaction, where a heater control application turns on a heater when the temperature is lower than a threshold, and a smoke control application opens the windows when smoke is detected. In this example, the *smoke* physical channel can connect the heater and the smoke sensor to create an inter-app interaction chain. If the attackers know the two apps' logics, they can utilize the temperature sensor's firmware vulnerabilities to manipulate the sensor to continuously report low temperature and keep the heater always working until smoke is produced, which leads to an unexpected act of window opening and a break-in.

5.3 Methodology

In this section, we describe the technical details about how user privacy can be leaked. Our goal is to analyze the encrypted wireless packets and eavesdrop user privacy. The procedures of eavesdropping are illustrated in Fig. 5.2.

5.3.1 Fine-grained IoT Event Stream Detection

The interactions between IoT devices and gateway are implemented by using some RF protocol. For a specific protocol, we use a corresponding packet sniffer tool to capture wireless network packets. A sniffer captures a raw wireless packet, and it tries to decode the packet. The packet fields for communication purpose is not usually encrypted, but the payload of the packet containing the user data is encrypted to avoid data leakage. Therefore, we have to utilize the unencrypted packet fields to capture the wireless communication pattern for the detection of some IoT events.

After encoding the raw wireless packets, we can get an unordered wireless packet sequence. We can regroup the wireless packets based on the detected device address. After regroup, we can get the new sniffed packet sequence $P_{d_i} = \langle p_1, p_2, \ldots, p_n \rangle$ for a device d_i. The packet sequence is generated from the interactions between the IoT device and gateway. We have to segment the packet sequence to some subsequence, and each subsequence represents a unique IoT interaction event.

For Zigbee and Z-wave, we find that most events only consist of tens of wireless packets and transmit packets within a few seconds. Then, an effective way to segment the packets is to set an appropriate time interval. For instance, if two packets have a time interval of more than 3s, we assume that they belong to two events. After we generate the subsequence, we can extract the features to build the vector for detection. In [121], We investigate some features from wireless packets to fingerprint the IoT devices, such as packet size, packet direction, and packet inter-arrival time. We can also utilize the features to build the fingerprint vector for IoT events. Then we can compute the calculated similarity between the detected vector with the authenticated vectors of events in the database. If the similarity with a specific authenticated vector exceeds a threshold, that means one IoT event is detected.

5.3.2 Mining Patterns From IoT Event Stream

After identifying individual IoT events, we can construct the event stream

$$E = \langle (e_1, t_1), (e_2, t_2), ..., (e_n, t_n) \rangle$$

where e_i ($i = 1, \ldots, n$) is the IoT event happening at time t_i. We discuss different types of potential human privacy leakage in Section II. The direct user privacy can be inferred from one individual IoT event. For instance, when a user opens its house door, the smart lock automatically sends a notification to the user. The IoT interaction event between smart lock and gateway can be detected. Then an attacker can infer that the door status is changed and the user may enter into the house. In this paper, we focus on a deeper user privacy leakage disclosed by mining patterns from IoT even stream. The mined IoT event patterns represent the **wireless context** on the fly. We mine the wireless context generated by IoT apps in [122]. In this work, we propose to mine more types of wireless context.

5.3.2.1 Frequent continuity mining

We define the *frequent continuity* as one or more IoT events frequently appear during fixed time duration, and the time intervals between events follow an induced pattern. For instance, if a humidity sensor is installed in the bathroom, we can infer if a user is taking a bath. From the event sequence E. if we find an event subsequence

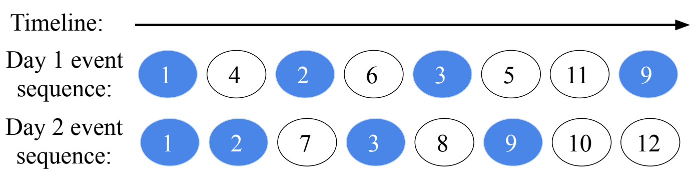


Figure 5.3: Daily routine mining example.

$$E_k = \langle (e_i, t_i), (e_i, t_{i+1}), (e_i, t_{i+2}), ..., (e_i, t_{i+L}) \rangle$$

where IoT event e_i represents humidity change event, and the time intervals between events

$$\mathbb{T}_k = (t_{(i,i+1)}, t_{(i+1,i+2)}, ..., t_{(i+L-1,i+L)})$$

follow some pattern. Then we can determine when a user is in the bathroom and taking a shower. Also, a event subsequence including many bulb color change events can be used to infer user's mood.

5.3.2.2 Period pattern mining

We define *period pattern* as a set of events frequently occur in order during a fixed period, such as one day or one week. Other events may appear between two events, and the time interval between two events may not be fixed. For human privacy, we can infer a user's daily routine by mining period pattern. Fig. 5.3 shows one daily routine mining example. By observing an event sequence from a user for two days, we find that the user will repeat four events: 1, 2, 3, and 9. The user may have other events between two adjacent events. The four events are not executed in fixed time points with a strict time interval but definitely follow an order. One example of such repeat event sequence for a user is like: *brush teeth*, *close door*, *open door*, and *turn on light*. This describes a user's daily routine: on the morning, the user gets up and brushes teeth, and leaves home for work, at nightfall, the user returns home, opens the door, and turns on the indoor light.

For a given time period P, we can segment the event stream E to M sequences:

$$E_{s_1} = \langle e_{(s_1,1)}, e_{(s_1,2)}, e_{(s_1,3)}, ..., e_{(s_1,N_1)} \rangle$$
$$......$$
$$E_{s_j} = \langle e_{(s_j,1)}, e_{(s_j,2)}, e_{(s_j,3)}, ..., e_{(s_j,N_j)} \rangle$$

Our goal is to extract the longest common subsequence of most of the M sequences. Notice that we do not compute the longest common subsequence for all the M sequences because a user may not strictly repeat their daily routine every day. First, for any two sequences E_{s_i} and E_{s_j}, we use the classical dynamic programming method to compute the longest common subsequence and build the common subsequence set. We select the subsequence that is repeated most often in the set as the period pattern for the whole even stream with a given period P.

5.3.2.3 Frequent episode mining

We define frequent episode as a frequent co-occurrence and ordered events. Notice that these events have no period pattern, and they can occur together at any time. For example, for an event sequence

$$E = (e_1, 1), (e_2, 2), (e_7, 3), (e_8, 4), (e_1, 5), (e_2, 6), (e_3, 7),$$
$$(e_9, 8), (e_{10}, 9), (e_1, 10), (e_2, 11), (e_7, 12)$$

we can get two frequent episode patterns $P_1 = \langle e_1, e_2 \rangle$ and $P_2 = \langle e_1, e_2, e_7 \rangle$. The first pattern has three occurrences and the second pattern has two occurrences. If each event represents a type of a user activity, the first pattern may represent one of living habits from the user with a greater probability. For example, the user turns on the coffee machine, and then plays music. The second pattern can also occur in the user's life but not frequently than the first pattern. Sometimes, the user may close the window while playing music.

We propose to use a *scan-and-join* algorithm to mine the frequent episode. We scan the event stream E and count the frequency of occurrence for each pair event $\langle (e_i, t_i), (e_j, t_{i+1}) \rangle$. Event e_i and e_j are two adjacent events from the time dimension. Then we can sort the event pair based on the frequency of occurrence. We can set up a *threshold* and select the event pairs with the number of occurrences exceeding the threshold to proceed. Some of the event pair may be the subsequence of other event sequences of 3-length or more. Suppose event pairs (e_i, e_j) and (e_j, e_k) have the approximate number of occurrence, we can determine whether to join the two event pairs to form the 3-length event sequence (e_i, e_j, e_k). We scan the event stream E again and count the number of event sequence (e_i, e_j, e_k). If the number also approximates with the numbers from event pairs (e_i, e_j) and (e_j, e_k), we can confidently join them. Otherwise, the two event pairs represent two of the living habits separately. Notice that we do not compute the frequency of 3-length in advance, considering the huge memory cost. We repeat the procedures to continue the join and check if we can build the frequent episode with a length of more than 3. Ultimately, we can build the frequent episode patterns that can infer a user's living habits or other privacy.

If events $\langle (e_i, t_i), (e_j, t_j), ..., (e_m, t_m), (e_n, t_n) \rangle$ frequently occur together and the event intervals $\langle (t_j - t_i), ..., (t_n - t_m) \rangle$ are fixed, we think these events form a strictly sequential pattern. Notice that these events may not be adjacent to each other, and may appear at discrete time points. They may not occur together for a short time. Nevertheless, from the whole time domain, we may find the hidden dependency. To mine the pattern, we enumerate all the event pair (e_i, e_j) from the event stream E. There may be other events between them in chronological order. We also need to consider the event pair (e_j, e_i). Then we compute the time intervals for all the event pair (e_i, e_j). If the variance of time intervals approach to zero or falls below a specific threshold, we think event e_i and e_j have a strictly sequential pattern. Next, we need to check if we can join different event pairs and build a larger sequential pattern. If events pair (e_i, e_j) and (e_j, e_k) both follow sequential patterns, we can further check if event pair (e_i, e_k) also follows sequential pattern and the time interval between event e_i and e_j are the sum of the interval between event e_i and e_j and the interval between event e_j and e_k. If they are equal, we can induce (e_i, e_j, e_k) can build a strictly sequential pattern. We iterate this procedure and concatenate the event pairs. Finally, we may discover some strictly sequential patterns.

Human activities usually do not have strictly sequential patterns. However, such information can leak another kind of human privacy - *what smart apps a user installs for smart home*. Smart home is driven by a set of smart apps that a user chooses to install. Smart apps execute some activities automatically to enable automation services. These activities may follow strictly sequential patterns. For instance, an app implements a functionality *"detect a human presence, then turn on the light"*. We have two IoT events: *Presence sensor sends a "status change" notification to the app* and *the app issues a "turn on" command to smart bulb*. These two events execute sequentially and often occur together, and the time interval between them, including network transmission delay and software progress, is almost fixed and has no big variance. So we can detect such app logic by mining the sequential pattern from the two events. For another smart app *"If detect the door is closed, after 5 minutes, lock the door."*, we can see the two IoT events still have a temporal dependency but spanning a longer time. Our approach can still detect such a sequential pattern with 5 minutes interval. The mined app logics can imply what a user prefers. The app logics can even be used by an attacker to launch attacks and create more critical security threats to users.

5.3.3 User Privacy Leakage from Mined Event Patterns

Now, we can infer different kinds of user privacy described based on the mined patterns from the sniffed and detected IoT event stream. The IoT events generated by actuators can directly expose user activities, such as *"user changes bulb color, or user turns on the air conditioner."* In the previous section, we propose a series of methods to mine various event patterns. These patterns may reflect some sensors' status and do not directly disclose user's activities. We have to correlate these event patterns with specific user activities. For example, although we can mine the frequent continuity pattern for a humidity sensor, the machine cannot learn that such a pattern indicates a user is taking a shower. We need to correlate the detected patterns with user activities automatically.

Most of the correlations come from human-readable content from commercial IoT devices' introduction, IoT reports, magazines, newspapers, and others. We use Natural Language Processing (NLP) techniques to analyze the human-readable content and extract useful information. We can calculate the semantical similarity between the words describing the sensors and the user's behavior. Specifically, we use word embeddings from a pre-trained Word2Vec model [123] and compute the cosine distance between the words. We can use public SmartApp descriptions to collect words describing sensor events and user's activities. Then we can induce practical user activities by analyzing the detected IoT event patterns.

5.4 Evaluation

In order to demonstrate the feasibility of our privacy eavesdropping approach, we implement our idea and conduct some experiments on the Samsung SmartThings platform. Several commercial IoT devices are deployed in our testbed. All the devices are connected to a SmartThings hub with Zigbee wireless communication protocol. A set of SmartApps is installed to SmartThings to enable the provision of automation services.

5.4.1 Strictly sequential pattern mining

Encrypted traffic sniffing. SmartThings hub can support wireless transmission with Zigbee protocol. We use the five most commonly used IoT devices that adopt Zigbee for our evaluation: Motion sensor, Outlet, Water leak sensor, Philips Hue A19, and Multipurpose sensor. In order to sniff the Zigbee traffic, we use TI CC2531 USB Dongle [113] and the Zigbee protocol sniffing tool [114] to capture the wireless communication packets between hub and devices. We can sniff all the Zigbee channels one by one to check the channel state and configure the sniffer to the channel that SmartThings is using. After capturing the packet sequence, we can use the Device Network ID from the packet to differentiate the packets for each device. For example, the Philips Hue A19 is identified by the Device Network ID 0x5882.

IoT event collection. Before event detection, we have to collect events to build the fingerprint matrix or authenticated vectors. We deploy the testbed in a typical office environment. Based on our testbed device capabilities, we select the existing apps in the SmartThings Public Github Repository [115] and develop our customized apps to build a total of 35 SmartApps library and install them in the SmartThings platform.

Table 5.1: Packet vector for IoT event detection.

IoT Event Name	Packet Size with Direction Sequence
MotionSensor.Active/InActive	-54, -12, 45, -12, 50, -45
MotionSensor.TemperatureChange	-53, -12, 45, -12, 50, -45
Outlet.PowerOn/Off	48, -55, -55, -55, -50, -45, -52, 45, 45, 50, -55, -45
Outlet.PowerMeter	-55, -53, 45, 50, -55, -45
WaterLeak.Dry/Wet	-54, -12, 45, -12, 50, -45
WaterLeak.TemperatureChange	-53, -12, 49, -12, 54, -45
PhilipsHue.PowerOn/Off	48, -45, -50, 50, -45, -53, 50, -45, -53
PhilipsHue.ColorControl	48, -45, -50, 52, -45, -50, 50, -45, -53, 50, -45, -53
PhilipsHue.DimmerControl	-51, 45, 50, -50, 45, 53, -50, 45, 53
PhilipsHue.ColorTemperature	48, -45, -50, 52, -45, -50, 50, -45, -53, 50, -45, -53, 50, -45, -54
MultipurposeSensor.Open/Close	-54, -12, 45, -12, 50, -45
MultipurposeSensor.Vibration	-65, -12, 45, -12, 52, -45
MultipurposeSensor.NoVibration	-45, -12, 45, -12, 52, -45
MultipurposeSensor.TemperatureChange	-53, -12, 45, -12, 50, -45

These SmartApps provide the customized IoT services, such as *If motion is detected, turn on the Philips Hue A19*, and automatically generate the corresponding IoT events. We continuously sniff and collect three weeks' wireless packet sequence for all IoT events. For some events that are triggered very infrequently in a practical environment, such as *WaterLeak.Dry/Wet*, we manually wet and dry the water leak sensor to generate sufficient event samples for the event fingerprinting. We label the packet sequence by matching the event history downloaded from the SmartThings iOS app. **Packet feature analysis for event detection**. Before identifying the IoT events, we have to generate the fingerprint vectors for each event. The five IoT devices can generate a total of 19 types of events, which can be found via the SmartThings iOS app. Each event corresponds to a SmartThings capability command. For example, the Philips Hue A19 can generate the following IoT events: *power on/off, color control, dimmer control,* and *color temperature control.* The corresponding capability commands are *switch.on(), switch.off(), colorControl.setColor(), switchLevel.setLevel(),* and *colorTemperature.setColorTemperature().* We collect the sniffed Zigbee wireless packets and match them with the downloaded event history from the SmartThings iOS app.

We have the following observations for the event detection features from the experiments:

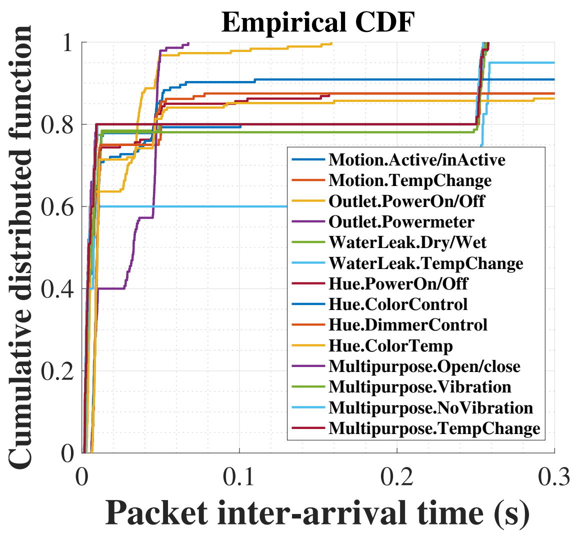


Figure 5.4: CDF of packet inter-arrival time.

(1) For most events, the packet size sequences are distinct, as shown in Table 5.1. Although the event pair *motion.Active / InActive*, *Outlet.PowerOn / off*, and *WaterLeak.Dry / Wet* have the same packet size sequence, the events in each pair are contrary to each other, which implies that we can use one variable to record the device's status to distinguish two contrary events.

(2) Although some sensors use identical capabilities for the same purpose, their packet sequence and size are still distinct. For instance, the motion sensor, water leak sensor, and multipurpose sensor can all detect temperature change. Once the temperate change is detected, they all send the same event via capability command *temperature.value* in SmartThings API. However, their packet size sequences are distinguishable and can be used for fingerprinting, as shown in Table 5.1.

(3) The direction of the packet in the sequence can also be used to distinguish some events. We use positive packet size to denote the direction from the hub to the device and negative packet size to show the opposite direction. The table shows the packet directions for different events from outlet and Philips Hue A19, and verify the effectiveness of using packet direction as a fingerprint feature.

(4) The time interval between adjacent packets of the same event can also be used to distinguish different events. The cumulative distribution function (CDF) of the time interval of all the events is shown in Fig. 5.4.

Based on the observation, we find that the feature vector extracted from the packet sequence is sufficiently used to identify IoT event. The same type of events has an identical packet sequence. By comparing

Table 5.2: Leakage of user installed IoT app logic.

No.	Discovered SmartApps Logic
1	$\boxed{\text{Motion}} \xrightarrow{\text{Active}} \boxed{\text{Hub}} \xrightarrow{\text{PowerOn}} \boxed{\text{Hue}}$
2	$\boxed{\text{Multipurpose}} \xrightarrow{\text{TempChange}} \boxed{\text{Hub}} \xrightarrow{\text{ColorControl}} \boxed{\text{Hue}}$
3	$\boxed{\text{Outlet}} \xrightarrow{\text{PowerMeter}} \boxed{\text{Hub}} \xrightarrow{\text{PowerOff}} \boxed{\text{Outlet}}$
4	$\boxed{\text{Water leak}} \xrightarrow{\text{Wet}} \boxed{\text{Hub}} \xrightarrow{\text{PowerOff}} \boxed{\text{Outlet}}$

the feature vector, we can distinguish different types of events.

IoT event detection performance analysis. We constantly sniff the wireless traffic for another one week and attempt to identify the unlabeled events in a real-time manner and generate the event sequence. By observing all kinds of events, the maximum number of packets transmitted is 15, and the maximum transmission time is nearly 2.06s. Therefore, we use a time gap of 2.1s to segment the wireless packets for events. Before event detection, we remove the unrelated packets, including beacon packets, link-maintain packets, and acknowledgment packets. We still compare the detected events with the recorded events in the SmartThings iOS app and compute the detection accuracy. The overall identification accuracy is more than 98%. The detection failure is mainly due to packet loss — the sniffer may miss some packets due to its limitation or other signal interference.

User Privacy Leakage Analysis. After detecting the IoT events and their dependencies, we infer user's privacy. The detected IoT event *Multipurpose.ContactOpen/Close* directly shows that people enter into the office or leave the office. Sometimes, the events *Motion.Active* are frequently detected within a short time, which implies that people paces back and forth in the room and may be thinking about some questions. 35 event dependencies are detected, which exactly match the app logic from 35 installed SmartApps. Table 5.2 shows part of the deduced app logic via event dependency discover.

Future work: First, in this work, we work on the Samsung SmartThings platform, which uses ZigBee protocol. We may evaluate our approach on other platforms based on WiFi and BLE protocol. Second, to mine the practical user privacy in real-world scenarios, we plan to deploy our system in users' homes and collect a larger dataset that may include a variety of practical IoT event patterns, which may be better utilized to evaluate the performance of our event pattern mining algorithms and human privacy eavesdropping approach.

5.5 Summary

In this paper, we explore user privacy leakage from a new angle – wireless context analysis. Most existing research can only discover some superficial user privacy via traffic pattern matching. In this paper, we define the wireless context as the wireless communication logics triggered by user activities. We find that the wireless context can leak more types of user privacy that existing works have ignored. Wireless context can be mined via discovering a variety of IoT event dependencies. The temporal IoT event dependencies can disclose human living habits and routines, even the app logics installed by the user. We implement our wiretaps of privacy on the Samsung SmartThings platform and demonstrate the feasibility of our approach.

Chapter 6

Conclusion

The interactive nature of IoT system components brings tremendous challenges in IoT system security. Hence, it is important to consider all of the essential components simultaneously when analyzing IoT system security. In this dissertation, we propose frameworks on generating attack graphs for IoT systems and the wireless traffic analysis for IoT app anomaly detection. The generated attack graphs enumerate all of the potential attack traces, thus providing useful guidance for system hardening. Moreover, we combine wireless traffic analysis with NLP-based IoT app analysis to detect potential threats in IoT systems, such as malicious IoT apps, command injection, or user privacy leakage.

FORESEE is a cross-layer vulnerability analysis framework for IoT systems. We design a novel IoT hypothesis graph which considers all of the core components of IoT systems, including user states and behavior and the potential attacks, which are largely ignored by existing research. Our hypothesis graph is amenable to existing model checking tools. We have applied our framework to smart home and smart healthcare systems. To further tackle the state explosion problem of hypothesis graphs, we propose IOTA, which takes system configuration and CVE database as input and generates exploit-dependency attack graphs. Based on the attack graphs, IOTA computes three metrics, viz. the shortest attack trace, blast radius, and severity score, to help system administrators evaluate vulnerabilities' impacts. Evaluation results show that IOTA is both effective and highly efficient.

Though attack graphs show all of the potential attack traces to IoT system resources, they do not dynamically monitor IoT system execution at runtime. Therefore, we design IOTGAZE, which sniffs the wireless traffic and compares the actual execution with the expected behavior specified by IoT apps to detect potential threats. In IOTGAZE, we design a fingerprinting-based event detection approach and use it to generate the event sequence via sniffed wireless packets. Moreover, we extract the IoT context which represents user expected app behaviors via analyzing apps' descriptions via natural language processing techniques. We apply our framework to the Samsung SmartThings platform and demonstrate the feasibility and effectiveness of IoTGaze. Finally, we explore the user privacy leakage issue in IoT systems. IoTGaze shows that it is possible to infer user's privacy, like living habits, routines, and even installed IoT applications by just sniffing the encrypted wireless traffic.

We hope the frameworks proposed in this dissertation will be helpful for system-level IoT security analysis and IoT system execution monitoring in the presence of an adversary.

Bibliography

[1] SmartThings, https://smartthings.developer.samsung.com.

[2] Google Nest, https://developers.google.com/home?hl=en.

[3] Philips Hue, https://developers.meethue.com/develop.

[4] L. Catarinucci, D. De Donno, L. Mainetti, L. Palano, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An iot-aware architecture for smart healthcare systems," *IEEE internet of things journal*, vol. 2, no. 6, pp. 515–526, 2015.

[5] S. B. Baker, W. Xiang, and I. Atkinson, "Internet of things for smart healthcare: Technologies, challenges, and opportunities," *IEEE Access*, vol. 5, pp. 26 521–26 544, 2017.

[6] R. Sanchez-Iborra and M.-D. Cano, "State of the art in lp-wan solutions for industrial iot services," *Sensors*, vol. 16, no. 5, p. 708, 2016.

[7] B. Chen, J. Wan, L. Shu, P. Li, M. Mukherjee, and B. Yin, "Smart factory of industry 4.0: Key technologies, application case, and challenges," *Ieee Access*, vol. 6, pp. 6505–6519, 2017.

[8] T. Abdelzaher, N. Ayanian, T. Basar, S. Diggavi, J. Diesner, D. Ganesan, R. Govindan, S. Jha, T. Lepoint, B. Marlin *et al.*, "Will distributed computing revolutionize peace? the emergence of battlefield iot," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1129–1138.

[9] M. J. Farooq and Q. Zhu, "On the secure and reconfigurable multi-layer network design for critical information dissemination in the internet of battlefield things (iobt)," *IEEE Transactions on Wireless Communications*, vol. 17, no. 4, pp. 2618–2632, 2018.

[10] "Forecast end-user spending on IoT solutions worldwide from 2017 to 2025," https://www.statista.com/statistics/976313/global-iot-market-size/.

[11] "Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025," https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[12] Apple HomeKit, https://developer.apple.com/homekit.

[13] OpenHab, "Openhab," https://www.openhab.org.

[14] Alexa, https://developer.amazon.com.

[15] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *CCS*, 2006.

[16] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *IEEE Symposium on Security and Privacy*, 2002.

[17] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *CCS*, 2002.

[18] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000.* IEEE, 2000, pp. 156–165.

[19] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security*, 2014.

[20] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Asia CCS*, 2016.

[21] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *CCS*, 2017.

[22] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.

[23] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking drones with intentional sound noise on gyroscopic sensors," in *USENIX Security*, 2015.

[24] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu, "Light commands: laser-based audio injection attacks on voice-controllable systems," in *USENIX Security*, 2020.

[25] M. Price, "August smart locks could be giving hackers your wi-fi credentials," https://www.cnet.com/home/security/august-smart-locks-could-be-giving-hackers-your-wi-fi-credentials/.

[26] Eques Elf Smart Plug, https://nvd.nist.gov/vuln/detail/CVE-2019-15745.

[27] Xiaomi, https://global.developer.mi.com/.

[28] NVD, "August Lock," https://nvd.nist.gov/vuln/detail/CVE-2019-17098.

[29] 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.

[30] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in *CCS*, 2018.

[31] Vulnerability Scanning Tools, https://owasp.org/www-community/Vulnerability_Scanning_Tools.

[32] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel, "IotSan: Fortifying the safety of iot systems," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, 2018, pp. 191–203. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281440

[33] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated IoT safety and security analysis," in *2018 USENIX Annual Technical Conference*, 2018, pp. 147–158. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/celik

[34] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018, pp. 832–846. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243865

[35] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman, "IoTSAT: A formal framework for security analysis of the internet of things (IoT)," in *2016 IEEE Conference on Communications and Network Security (CNS)*, 2016, pp. 180–188.

[36] M. Mohsin, M. U. Sardar, O. Hasan, and Z. Anwar, "IoTRiskAnalyzer: A probabilistic model checking based framework for formal risk analytics of the internet of things," *IEEE Access*, vol. 5, pp. 5494–5505, 2017.

[37] F. Capobianco, R. George, K. Huang, T. Jaeger, S. Krishnamurthy, Z. Qian, M. Payer, and P. Yu, "Employing attack graphs for intrusion detection," in *Proceedings of the New Security Paradigms Workshop*, 2019, pp. 16–30.

[38] X. Ou, S. Govindavajhala, and A. W. Appel, "Mulval: A logic-based network security analyzer." in *USENIX Security*, 2005.

[39] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 121–130.

[40] M. Albanese, S. Jajodia, and S. Noel, "Time-efficient and cost-effective network hardening using attack graphs," in *IEEE/IFIP DSN)*, 2012.

[41] K. Durkota, V. Lisỳ, B. Bošanskỳ, and C. Kiekintveld, "Optimal network security hardening using attack graph games," in *International Joint Conference on Artificial Intelligence*, 2015.

[42] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel, "Iotsan: Fortifying the safety of iot systems," in *CoNext*, 2018.

[43] J. Han, A. J. Chung, M. K. Sinha, M. Harishankar, S. Pan, H. Y. Noh, P. Zhang, and P. Tague, "Do you feel what i hear? enabling autonomous iot device pairing using different sensor types," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 836–852.

[44] D. Formby, P. Srinivasan, A. Leonard, J. Rogers, and R. A. Beyah, "Who's in control of your control system? device fingerprinting for cyber-physical systems." in *NDSS*, 2016.

[45] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 439–454.

[46] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2177–2184.

[47] T. Gu and P. Mohapatra, "Bf-iot: Securing the iot networks via fingerprinting-based device authentication," in *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2018, pp. 254–262.

[48] J. Jiang, C. Jiuming, T. Gu, K.-K. Raymond Choo, C. Liu, M. Yu, W. Huang, and P. Mohapatra, "Anomaly detection with graph convolutional networks for insider threat and fraud detection," in *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019.

[49] ——, "Warder: Online insider threat detection system using multi-feature modeling and graph-based correlation," in *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, 2019.

[50] N. Cheng, X. Oscar Wang, W. Cheng, P. Mohapatra, and A. Seneviratne, "Characterizing privacy leakage of public wifi networks for users on travel," in *2013 Proceedings IEEE INFOCOM*, April 2013, pp. 2769–2777.

[51] Y. Liu and Z. Li, "aleak: Privacy leakage through context - free wearable side-channel," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, April 2018, pp. 1232–1240.

[52] C. Wang, X. Guo, Y. Chen, Y. Wang, and B. Liu, "Personal pin leakage from wearable devices," *IEEE Transactions on Mobile Computing*, vol. 17, no. 3, pp. 646–660, March 2018.

[53] C. Wang, C. Wang, Y. Chen, L. Xie, and S. Lu, "Smartphone privacy leakage of social relationships and demographics from surrounding access points," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 678–688.

[54] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium*, 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[55] 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 *26th USENIX Security Symposium*, 2017, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian

[56] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. Unviersity, "Contexlot: Towards providing contextual integrity to appified iot platforms." in *NDSS*, vol. 2, 2017, pp. 2–2.

[57] 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*, ser. CCS '18, 2018, pp. 1074–1088. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243820

[58] P. T. P. LLP, "Z-shave. exploiting z-wave downgrade attacks," https://www.pentestpartners.com/security-blog/z-shave-exploiting-z-wave-downgrade-attacks/.

[59] ——, "Burning down the house with iot," https://www.pentestpartners.com/security-blog/burning-down-the-house-with-iot/.

[60] Y. Wan, K. Xu, G. Xue, and F. Wang, "Iotargos: A multi-layer security monitoring system for internet-of-things in smart homes," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 874–883.

[61] A. Wang, A. Mohaisen, and S. Chen, "Xlf: A cross-layer framework to secure the internet of things (iot)," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1830–1839.

[62] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 636–654.

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

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

[65] Common Vulnerabilities and Exposures (CVE), https://cve.mitre.org/, 2021.

[66] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[67] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013.

[68] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.

[69] SmartThings Public GitHub Repo, https://github.com/SmartThingsComm\unity/SmartThingsPublic.

[70] G. J. Holzmann, "An improved protocol reachability analysis technique," *Software: Practice and Experience*, vol. 18, no. 2, pp. 137–161, 1988.

[71] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[72] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *USENIX Security*, 2019.

[73] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[74] E. Y. Vasserman and N. Hopper, "Vampire attacks: draining life from wireless ad hoc sensor networks," *IEEE transactions on mobile computing*, vol. 12, no. 2, pp. 318–332, 2011.

[75] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," in *USENIX ATC 18*, 2018.

[76] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu, “Understanding and automatically detecting conflicting interactions between smart home iot applications,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1215–1227.

[77] https://nvd.nist.gov/vuln/detail/CVE-2019-17627.

[78] https://nvd.nist.gov/vuln/detail/CVE-2019-5035.

[79] Alexa Skill, https://www.amazon.com/Amazon-Key/dp/B075LY9H6H.

[80] IFTTT SmartThings Applets, https://ifttt.com/smartthings, 2021.

[81] 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*. IEEE, 2020, pp. 884–893.

[82] W. Ding and H. Hu, “On the safety of iot device physical interaction control,” in *CCS*, 2018.

[83] Common Vulnerability Scoring System (CVSS), https://www.first.org/\cvss/, 2021.

[84] E. Pendergrass, “Cheap, hackable iot light bulbs (or, philips bulbs have no security),” https://blog.dammitly.net/2019/10/cheap-hackable-wifi-light-bulbs-or-iot.html.

[85] No Authentication Vulnerability in Radio Thermostat, https://www.trust\wave.com/en-us/resources/security-resources/security-advisories/?fid=1\8874.

[86] National Vulnerability Database (NVD), https://nvd.nist.gov/.

[87] Google Assistant, https://developers.google.com/assistant/smarthome/overview.

[88] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, “The Stanford CoreNLP natural language processing toolkit,” in *ACL*, 2014.

[89] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O’Reilly Media, Inc.", 2009.

[90] CVSS Scoring Rubrics, https://www.first.org/cvss/user-guide#Scoring-Rubrics.

[91] Common Weakness Enumeration (CWE), https://cwe.mitre.org/, 2021.

[92] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, “An attack graph-based probabilistic security metric,” in *IFIP Annual Conference on Data and Applications Security and Privacy*, 2008.

[93] August Smart Home, https://www.amazon.com/August-Home-Smart/dp/B06WW2XQ68.

[94] 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*, 2020, pp. 272–285.

[95] IFTTT Smart Home Applets, https://ifttt.com/search/query/smart%20ho\me?tab=applets, 2021.

[96] SmartThings Products List, https://www.smartthings.com/products-list.

[97] SmartThings Partners, https://www.smartthings.com/partners.

[98] Ericsson, "Ericsson mobility report: Internet of things forecast," https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2018.

[99] IDC, "Worldwide semiannual internet of things spending guide," https://www.idc.com/getdoc.jsp?containerId=IDC_P29475, 2018.

[100] Samsung, "Smartthings," https://www.smartthings.com, Accessed: 2015.

[101] Apple, "Homekit," https://developer.apple.com/homekit/, Accessed: 2015.

[102] Google, "Google home," https://developers.google.com/iot/, Accessed: 2015.

[103] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[104] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 531–548.

[105] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1687–1704.

[106] 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 *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 361–378.

[107] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot." in *NDSS*, 2019.

[108] C. Nandi and M. D. Ernst, "Automatic trigger generation for rule-based smart homes," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 2016, pp. 97–102.

[109] I. Bastys, M. Balliu, and A. Sabelfeld, "If this then what?: Controlling flows in iot apps," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security(ACM CCS'18)*. ACM, 2018, pp. 1102–1119.

[110] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[111] Samsung, “Capabilities reference for smartthings iot platform,” https://docs.smartthings.com/en/latest/capabilities-reference.html, 2018.

[112] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[113] T. INSTRUMENTS, “Zigbee protocol packet sniffer,” http://www.ti.com/tool/PACKET-SNIFFER, 2019.

[114] ——, “Cc2531 usb evaluation module kit,” http://www.ti.com/tool/cc2531emk, 2019.

[115] S. developer, “Smartthings public github repository,” https://github.com/SmartThingsCommunity/SmartThingsPublic, 2019.

[116] J. K. Becker, D. Li, and D. Starobinski, “Tracking anonymized bluetooth devices,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 50–65, 2019.

[117] J. Martin, D. Alpuche, K. Bodeman, L. Brown, E. Fenske, L. Foppe, T. Mayberry, E. Rye, B. Sipes, and S. Teplov, “Handoff all your privacy–a review of apple’s bluetooth low energy continuity protocol,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 4, pp. 34–53, 2019.

[118] N. Apthorpe, D. Reisman, and N. Feamster, “A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic,” *arXiv preprint arXiv:1705.06805*, 2017.

[119] T. OConnor, W. Enck, and B. Reaves, “Blinded and confused: uncovering systemic flaws in device telemetry for smart-home internet of things,” in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, 2019, pp. 140–150.

[120] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster, “Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic,” *arXiv preprint arXiv:1708.05044*, 2017.

[121] T. Gu and P. Mohapatra, “BF-IoT: Securing the iot networks via fingerprinting-based device authentication,” in *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, 2018, pp. 254–262.

[122] 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*, 2020.

[123] Y. Goldberg and O. Levy, “word2vec explained: deriving mikolov et al.’s negative-sampling word-embedding method,” *arXiv preprint arXiv:1402.3722*, 2014.