This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

CommandFence: A Novel Digital-Twin-Based Preventive Framework for Securing Smart Home Systems

Yinhao Xiao*, Yizhen Jia†, Qin Hu‡, Xiuzhen Cheng§†, Bei Gong¶, Jiguo Yu‖
*School of Information Science, Guangdong University of Finance and Economics, Guangzhou, China.
20191081@gdufe.edu.cn
†Department of Computer Science, The George Washington University, Washington, D. C., USA.
{chen2015,cheng}@gwu.edu
‡Department of Computer and Information Science, Indiana University - Purdue University Indianapolis, IN, USA.
qinhu@iu.edu (Corresponding Author)
§School of Computer Science and Technology, Shandong University, Qingdao , P. R. China.
¶School of Information Technology, Beijing University of Technology, Beijing 100124, P. R. China.
gongbei@bjut.edu.cn
‖Qilu University of Technology (Shandong Academy of Sciences)
Shandong Computer Science Center (National Supercomputer Center in Jinan), Jinan, Shandong, 250014, P. R. China.
jiguoyu@sina.com.

***Abstract*—Smart home systems are both technologically and economically advancing rapidly. As people become gradually inalienable to smart home infrastructures, their security conditions are getting more and more closely tied to everyone's privacy and safety. In this paper, we consider smart apps, either malicious ones with evil intentions or benign ones with logic errors, that can cause property loss or even physical sufferings to the user when being executed in a smart home environment and interacting with human activities and environmental changes. Unfortunately, current preventive measures rely on permission-based access control, failing to provide ideal protections against such threats due to the nature of their rigid designs. In this paper, we propose CommandFence, a novel digital-twin-based security framework that adopts a fundamentally new concept of protecting the smart home system by letting any sequence of app commands to be executed in a virtual smart home system, in which a deep-q network (DQN) is used to predict if the sequence could lead to a risky consequence. CommandFence is composed of an Interposition Layer to interpose app commands and an Emulation Layer to figure out whether they can cause any risky smart home state if correlating with possible human activities and environmental changes. We fully implemented our CommandFence implementation and tested against 553 official SmartApps on the Samsung SmartThings platform and successfully identified 34 potentially dangerous ones, with 31 of them reported to be problematic the first time to our best knowledge. Moreover, We tested our CommandFence on the 10 malicious SmartApps created by [1] and successfully identified 7 of them as risky, with the missed ones actually only causing smartphone information leak (not harmful to the smart home system). We also tested CommandFence against the 17 benign SmartApps with logic errors developed by [2] and achieved a 100% accuracy. Our experimental studies indicate that adopting CommandFence incurs a neglectable overhead of 0.1675 seconds.**

I. Introduction

Smart home systems have been advancing rapidly in recent years. Technologically, the advent of numerous smart home devices and platforms such as the Samsung SmartThings platform, gradually encourages the realization of completely automated home environments [3]. Economically, the total revenue of the global smart home market has reached 90,968 million USD by the year of 2020 [4]. More importantly, as machine learning techniques have been widely adopted by various smart home applications [5] to bring intelligence, smart home systems are getting truely smarter and smarter.

Regardless of the swift development of current smart home systems, their security conditions are far less than satisfactory. As smart devices in a smart home environment are typically controlled by the corresponding apps hosted in a smartphone, securing smart apps plays a crucial rule for the normal operations of smart home systems. Nevertheless, this is a non-trivial task. Sophisticated malicious smart home apps have been developed and they are gradually taking control of multiple sensitive devices such as smart home speakers [6]. In this paper, we consider two types of popular threats: (1) malicious apps developed with deliberate evil intentions such as those proposed by Jia *et al.* [1] and Fernandes *et al.* [7], and (2) benign apps with logic implementation errors such as those discovered by Celik *et al* [2]. These two threat models are detailed with examples in Section II. Our intention is to provide a unified framework based on digital-twin to counter the threats thus protecting the whole smart home system.

Nowadays, mainstream protective methods [1], [8], [9] against the threats mentioned above in smart home systems are mainly based on access control policies, which usually let

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

users or service providers set up a list of *static* rules, e.g., *who* is allowed to do *what*, to regulate the executions of the apps without considering the *interactions* of app commands, human activities, and environmental changes when being operated in a *dynamic* smart home system. Nevertheless, such interactions could threaten the security and safety of smart home environments if being exploited by attackers. On the other hand, permission-based mechanisms also suffer from issues such as coarse-grained policies and over-privileged access rights, which are unavoidable due to the nature of their rigid designs. Thus permission policies in access control are not effective in mitigating the aforementioned two security threats. There also exist non-access-control-based defensive mechanisms [10]–[12] but they can barely mitigate the security threats mentioned above as they target either memory corruptions [10]–[13] or function level vulnerabilities [13]. Such approaches again completely ignore the interactions of smart apps, human activities, and environmental changes, as they focus on an app itself without considering its interplay with the smart home environments when being executed.

Witnessing the above urgent situations, in this paper, we propose CommandFence, a novel digital-twin-based security framework to thwart the two types of security threats aiming at protecting smart home systems. CommandFence is composed of an Interposition Layer and an Emulation Layer. In the Interposition Layer, app commands are first interposed and then directed to the Emulation Layer. In the Emulation Layer, a virtual smart home environment is configured where the directed commands are executed and a learning model is used to predict whether these commands combining with human activities and environmental changes would lead to a harmful situation. If not, the commands would be passed to the real smart home environment; otherwise, they would be dropped. We fully implemented our CommandFence framework based on Android hooking for the Interposition Layer and the Deep-Q Network (DQN) for the Emulation Layer. The major reason we leveraged DQN in our implementation is because DQN is a typical and efficient reinforcement learning technique that follows an action-reward feedback structure, which perfectly fits the definition and context of our smart home security problem [14], as the states of a smart home system change along with a feedback every time a smart home user performs an action therein.

Note that our approach holds a fundamentally different philosophy which states that if the commands from smart home apps can potentially lead to a risky consequence, they should be treated as dangerous regardless of whether they are malicious or not and whether they follow their access policies or not. This novel design helps us identify threatening apps that are overlooked in the past, as witnessed by our extensive experimental studies. Also note that CommandFence is completely orthogonal to permission-based access control, and does not require any hardware upgrade if adopted, which means that existing smart home systems can implement CommandFence as plug-in software without changing their built-in access-control-based mechanisms.

Our Contributions. The major contributions are listed as follows:

- We proposed CommandFence, a novel digital-twin-based preventive framework to defend against the threats caused by malicious apps with evil intentions as well as benign ones with design flaws or logical errors that may harm a smart home environment when being executed and interacting with human activities and environmental changes. CommandFence is built on a fundamentally new idea, i.e., digital-twin, and is orthogonal to the well-received permission-based access control mechanisms, but it can capture the app commands that may lead the smart home environment to a risky state when being executed and drop them before any insecure situation can appear.
- We implemented our proposed CommandFence framework and trained the DQN with 232,315 pieces of sensing data collected from two real smart home environments. We tested our implementation on 553 official SmartApps[1] on the Samsung SmartThings platform and successfully identified 34 erroneous ones, among which 31 were reported to be problematic the first time to our best knowledge, which may cause a risky situation due to careless implementations[2]. We then tested our implementation against the 10 pure evil SmartApps created by [1], and CommandFence labeled 7 of them as risky, with the rest 3 being missed because they are not harmful to the smart home system but only cause smartphone information leaks. We also tested our implementation against the 17 benign SmartApps with logic errors [2] and achieved 100% success rates of flagging them as risky. Finally, we measured the latency of CommandFence and found that adopting our implementation incurred a neglectable delay of 0.1675 seconds.

Paper Organization. The rest of the paper is organized as follows. Section II presents the background knowledge and formulates our problem. Section III details the design of our CommandFence framework. Section IV demonstrates our implementation of CommandFence. Section V reports our evaluation results on CommandFence. Section VI outlines the most related work. Section VII concludes the paper with a future research discussion.

II. Background

In this section, we first present the structure of a general smart home system. Then we show how a typical permission-based access control framework functions and demonstrate its limitations. Finally we present our threat models.

A. A General Smart Home System

A general smart home system normally contains the following components: smart home apps running on a smartphone,

[1]A SmartApp is a tiny Groovy-based smart home app running on the SmartThings platform.

[2]Note that since the security of SmartApps have been extensively studied [2], [7], [8], [15], most problematic SmartApps have been patched or deleted from the official community.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

a network router, smart home devices (e.g., smart light bulbs, smart locks, and so on), device-associated smart home servers, and a central hub (optional) serving as a manager that may connect to other components mentioned above.

The lifecycle of a regular smart home control command begins with it being sent out from the corresponding smart home app and ends after its execution is finished at the target smart home device. There are three approaches to deliver a command to the target device. The first approach makes use of local communications where the smartphone app, the target device, and the router get involved. When a command comes out from the smartphone, it is routed either to the router which later delivers it to the target device if Wi-Fi is used (in the latter context, we omit the use of the router), or directly to the device if Bluetooth is used. The second approach involves the smartphone app, the router, the device, and the app server. In this scenario, a command coming from the smartphone is first sent to the app server, which then forwards it to the target device. The third approach differs from the second one by employing a central hub, in which a command is first sent to the hub server, which then forwards it to the central hub; next the hub directs the command to the device. Most of the popular industrial smart home systems such as Wemo, August, and TP-LINK adopt the first and the second approaches [16]–[18]; while several larger integrated smart home systems such as the Samsung SmartThings System tend to adopt the third approach [3]. The overall configuration of a general smart home system is illustrated in Fig. 1.

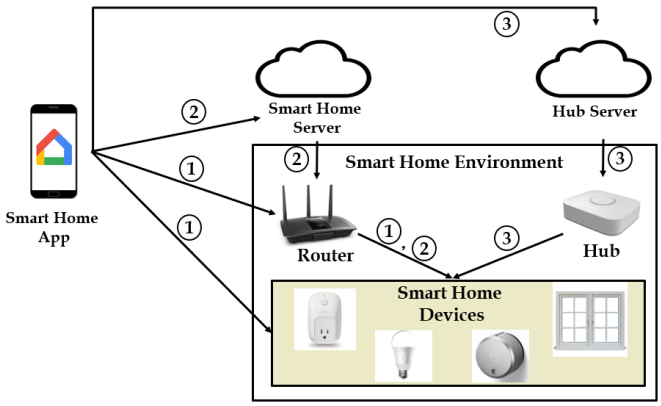

Fig. 1: A typical smart home system. Control flows ①, ②, and ③ correspond to the first, second, and third approach, respectively, to illustrate how a command is delivered to the target device.

B. Typical Permission-based Access Control Frameworks

As mentioned earlier, smart home systems are mainly protected via permission-based access control policies. Typical permission-based access control frameworks rely on permission management, i.e., *who* is allowed to do *what* according to the permission policy. In order to realize this philosophy, a "guard" is needed to check permission policy and filter out unauthorized commands in real-time. Both academia and industry have made tremendous efforts in designing and implementing the "guard". One of the designs is to implement the "guard" as the Android/iOS permission-based mechanism, where each app is assigned a set of permissions granted by the user [19]. Another common design is to implement the "guard" on the app servers or the central hub as a controller. Upon receiving a command, the servers or the hub verify if the pair of the identity and the control command matches with the permission policy. If no match is found, the command is deemed unauthorized and then blocked. Existing research such as SmartAuth [20], ContexIoT [1] and DTAP [9] adopt such an approach. In industry, this design was named the *trigger-action platform*. In fact, several off-the-shelf platforms were designed following this principle, e.g., the If-This-Then-That (IFTTT) platform [21], Microsoft Flow, and Zapier [22] [23]. The last type of design is to implement the "guard" on each smart home device as a decentralized permission-based access control framework. Examples such as IoT smart contract-based access control [24] and FairAccess [25] leverage this design.

Even though permission-based access control has been widely used for smart home systems, it has some unresolvable issues. First, it is not trivial to define permissions in a fine-grained manner for a smart home system. Unlike in a smartphone where permissions are limited and enumerable, e.g., audio, camera, location, since they are hardcoded in the OS kernels [26], the permissions in a smart home system are not only greatly outnumbered, but also more complicated concerning the contexts than those in a smartphone. For instance, according to a user's living habits, a smart light app turns on a light into normal mode between 6 PM and 7 PM when the user arrives in home from work and begins to cook. It then switches the light to the reading mode between 7 PM and 8 PM when the user is reading. Afterwards, it switches the light to the relaxation mode between 8 PM and 10 PM when the user watches TV. Finally, it turns the light off when the user goes to bed. This example represents only one living habit. Nevertheless, people may have different habits, significantly complicating the permission specifications in smart home scenarios. The second issue lies in that such a permission-based design is vulnerable to the overprivilege issue, which means that an app is granted more permissions than it needs. The advent of the overprivilege problem is usually due to a crude permission-granting design. Yet, it is difficult to avoid since a strict permission-granting design which can somehow relieve the overprivilege issue may harm the user experience due to tedious operations. According to the newest research published in 2019, it is estimated that 48% of the smartphone apps are overprivileged [27]. The third issue is the inherent limitations of current access control policies, which only specify *who* is allowed to do *what*, without considering the time-variant interactions of app commands, human activities, and environmental changes in a working smart home system. Nevertheless, their interplay can be exploited by attackers if not properly handled. For example, a smart-window app may be programmed to open windows when inside temperature is high but this activity is deemed dangerous if nobody is at

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

home. Last, since most of the smart home devices operate on different low-level OSes and hardware, and hence adopt heterogeneous protocols and functions, it is hard to come up with a unified permission management scheme that can take advantage of the hidden correlations among the app commands to better protect the smart home system. In short, due to these unresolvable challenges, applying permission-based access control to smart home systems is facing great challenges that may not be overcome in a foreseeable future.

C. Threat Models

We consider harmful smart home apps that run on a user's smartphone, send commands to the user's smart home devices, and cause dangerous and vulnerable consequences to the user's smart home environment. There exist mainly two types of such apps identified in practice: (1) malicious apps developed with deliberate evil intentions such as those proposed by Jia *et al.* [1] and Fernandes *et al.* [7], and (2) benign apps with logic errors introduced by implementation procedures such as those discovered by Celik *et al.* [2].

- **Malicious Apps with Pure Evil Intentions.** Such apps are created with only one purpose, i.e., to cause damage to a user's home property or even to harm a user's health. Jia *et al.* [1] demonstrated that by creating a malicious smart light app that constantly strobes a light at a high rate, attackers can easily trigger a user's seizure, hence causing a significantly dangerous situation, if the user has the health problem of seizure.
- **Benign Apps with Logic Errors.** Benign apps can also lead to a harmful situation if the implementation logic is not properly handled. This usually happens when multiple apps are running under one smart home environment such that an implementation flaw within a benign app may be triggered by other benign apps. Celik *et al* [2] demonstrated an example in which a smart home environment is deployed with a smoke-alarm app and a water-leak-detector app. The smoke-alarm app would open the water valve and activate the fire sprinklers if smoke is detected and the indoor temperature is over-heated (i.e., an indoor fire happens). The water-leak-detector app shuts off the main water supply valve if it detects a water leak through a moisture sensor. However, if these two apps are deployed together, a dangerous situation may happen: if a fire accident goes on, the smoke-alarm app would activate the fire sprinklers to clear the fire and smoke; then the water-leak-detector app shuts off the main water supply since it detects "a water leak"; as a consequence, the user's home may be burned down.

III. CommandFence: A Digital-Twin Based Protection Framework

In this section, we describe the basic idea of CommandFence, our digital-twin based protection system that can prevent the executions of app commands who might make the smart home insecure. We first present our problem formulation, which is abstracted from the threat models presented in Section II-C. Then we propose the overall structure of CommandFence, which aims to solve the formulated problem. Note that in this section, we present CommandFence as a high-level framework without implementation details since there might exist different implementations due to various considerations (e.g., different OSes). Our own implementation of CommandFence is detailed in Section IV.

A. Problem Formulation

In this subsection, we formally define our problem, which can be regarded as a unified abstraction of our threat models. To proceed, we need the following definitions.

Definition 1. *A state of a smart home environment is a vector of the states of the smart home devices $S = (s_1, s_2, \cdots, s_n)$, where n is the total number of devices, s_k is the state of the k-th device, e.g., "on" for a smart light bulb, or a temperature figure for a thermostat.*

A smart home state S has a score r indicating its "risk" level. Note that r is scored manually based on common beliefs, where the higher the r, the riskier the smart home state. For example, if S_1 indicates that a user is sleeping while the window is unlocked and S_2 indicates that the user is not at home while the door is unlocked, then obviously, S_2 is riskier than S_1 and should have a higher risk score, i.e., $r_1 < r_2$. Also note that the value of r does not have to be fine-grained since it is mainly used to characterize whether or not a smart home state is risky.

Generally speaking, in a typical smart home environment, human activities, environmental variations, or user instructions from the smartphone apps, can all cause smart home state changes, which then trigger the issues of control commands from one or more smart home apps. One can draw two implications from this observation. First, depending on the granularity of the state values, not all state changes can result in the issues of app commands. For example, environmental variations can cause the transitions from one normal state to another without triggering the issue of any app command. Second, at the smartphone a sequence of commands from one or more smart apps may be issued in order at a fairly short period of time for real-time services to jointly complete one smart home task. Here are a few examples:

- A fire-alarm app is described to raise a fire alarm and open windows once a fire is detected. Thus this app issues two commands in a sequence: opening windows then raising an alarm.
- An auto-unlock app unlocks the door if it detects the fingerprints of a user who is trying to get into the house matches one of the records in the database; then an auto-turn-on-light app turns on the light at the door way if it detects the door is unlocked. In this case, the door-unlocking command sent from the auto-unlock app triggers the light-opening command from the auto-turn-on-light app.
- When the window is open but the outside temperature is high, the smart home state triggers the smart-window app

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

to close the window and the smart-HVAC app to turn on the air conditioning system.

The first example describes a command sequence with two commands coming from the same app while the last two present sequences of commands issued by different apps that are triggered at a short period of time by either human activities or environmental changes. One can see that commands from different apps may be correlated, and they can interact with human activities or environmental variations to trigger the smart home state transitions. As smart home apps are usually written by different third-party developers, such an interplay can give chances to evil apps or make benign apps vulnerable (see the examples in Section II-C). In this paper, we intend to investigate the correlations among the smart apps, human activities, and environmental variations, and employ the concept of digital twin to figure out whether the execution of a command sequence can lead to a risky smart home state. Our problem is formally defined as follows:

Problem. *Given a normal smart home state S_N, i.e., its risk score r_N is less than a threshold δ, does the execution of a command sequence lead to a risky state if executed in order?*

To tackle this problem, we present CommandFence, a digital-twin based framework that can execute an app command sequence in an emulated smart home environment before it is sent to the real smart home system and predict whether the operations of these commands can result in a risky state, no matter what human activities may occur and what environmental changes may appear. Based on the prediction result, the app commands are either dropped if they are deemed harmful, or otherwise are allowed to be operated in the real smart home system. This implies that CommandFence can prevent a smart home from reaching a harmful state caused by the interactions of smart apps, human activities, and environmental changes.

Note that in sequel we employ the word "action" to represent a smart app, a human activity, or an environmental change. Also note that a command sequence refers to the commands coming in order from one or more smart apps that are interrelated in some way (e.g., triggered by the same event). As smart home systems are designed to provide real time services, a command sequence should arrive at a short period of time. Yet, one may not rule out the possibility that some undiscovered malicious apps are particularly cunning to deliberately send commands in a sequence with a long-time gap. In this case, CommandFence treats the commands as individual ones and handle them independently – the command that may trigger a risky state from the current normal state will be blocked when it is discerned. Having said so, we haven't seen any malicious or problematic app with this type of intention, and in a smart home environment, if commands are sent separately with a long-time gap, it is likely that they may end up not threatening at all. For example, a malicious app strobing a light by turning it on and off at a long time gap has no risk to lead to a victim's seizure.

B. Overall Structure of CommandFence

As analyzed in Section II, one can see that the current permission-based access control frameworks are not capable of properly protecting smart home systems. Driven by this observation, we propose a novel unified framework termed CommandFence based on the concept of digital-twin. Note that CommandFence is orthogonal to the existing permission-based access control systems and is applicable to all smart home settings presented in Section II-A. The main process of CommandFence contains three steps. First, smart home app commands are interposed and isolated regardless of their sources. Second, the interposed commands are directed to an emulated smart home environment where they are executed and an evaluation predicting whether or not the outcome is dangerous would be given. Third, based on the prediction result, the commands are permitted at the real smart home environment if they are predicted to be Not Risky, or dropped otherwise. To accomplish these steps, we design a two-layer architecture containing an Interposition Layer and an Emulation Layer. After the commands are generated from the deployed smart apps, they are interposed and isolated in the Interposition Layer, then directed to the Emulation Layer. In the Emulation Layer where a virtual smart home environment is configured, these commands are executed. We leverage a method based on DQN to predict if the combination of human activities/environmental changes with the forwarded commands, altogether called *actions*, could result in a dangerous situation. If not, the commands would be eventually forwarded to the real smart home environment; otherwise, they would be blocked. Compared with the traditional permission-based access control frameworks, CommandFence does not maintain *static* permission policies; instead, we let all commands to be (virtually) executed regardless of their sources and natures in an emulated environment, and use a learning model to *dynamically* predict possible hazardous situations evolutionarily. The overall structure of our approach is demonstrated in Fig. 2.

C. Interposition Layer

The main purpose of the Interposition Layer is to direct the commands that are supposed to route to the real smart home system to the Emulation Layer for further risk analysis. To realize this goal, we separate the main function of this layer into two processes: command identification and command direction.

- **Command Identification.** Command identification is a process of recognizing, locating and reassembling smart home commands by analyzing raw traffics or codes. Recognizing a command is to figure out what a command is; locating a command is to know where a command is; and reassembling a command is to put together the pieces (i.e., network packets) of a command. There is more than one solution to implement this process. From the perspective of analyzing raw traffics, one can extract the commands directly from the traffic contents if they

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

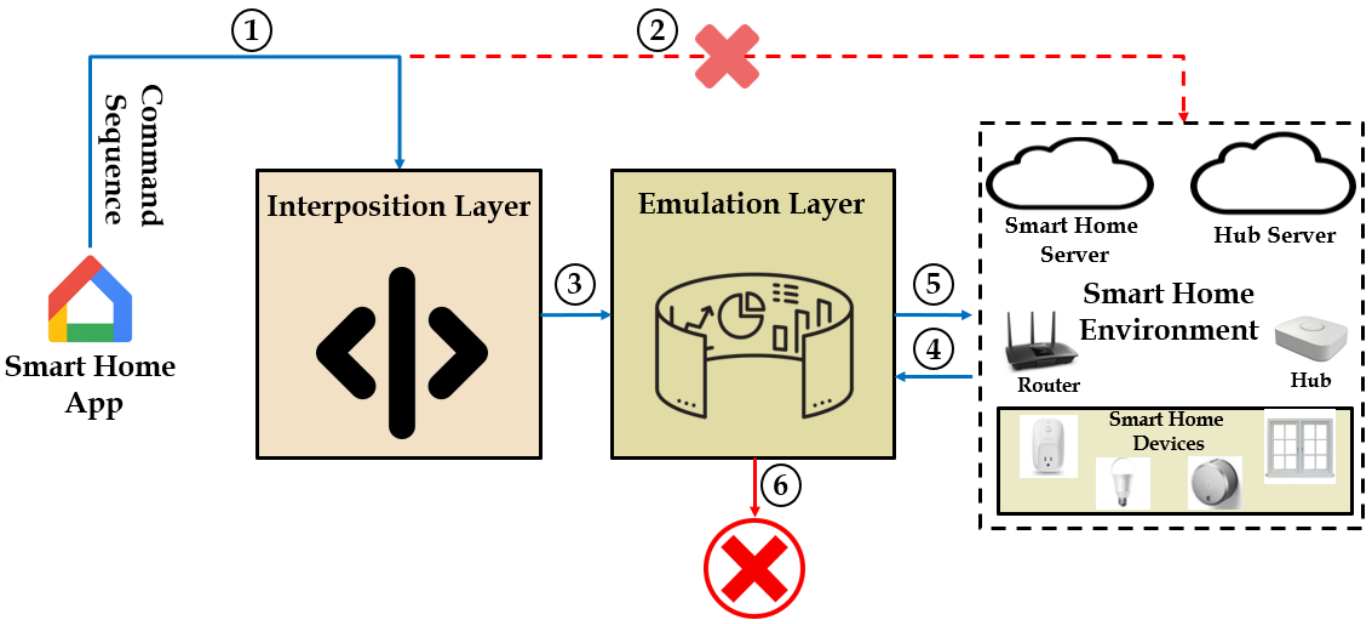

Fig. 2: The high-level structure of our digital-twin based approach. A command sequence, generated by a target smart home app, is intercepted by the Interposition Layer (marked with ①) instead of being directed to the real smart home environment as is usually done (marked with ②). The Interposition Layer then forwards the intercepted command sequence to the Emulation Layer for further analysis (marked with ③). At the same time, the Emulation Layer requests the real smart home to provide the current smart home state (marked with ④). Upon retrieving the smart home state, the Emulation Layer determines whether the command sequence would lead to a harmful situation if being executed based on the smart home state, and subsequently, determines whether to forward it to the real smart home system (marked with ⑤), or to block it (marked with ⑥).

are not encrypted [28]. On the other hand, if the traffics are encrypted, one can adopt the middlebox technique to decrypt the traffics if either of the communication parties does not conduct certificate checking [29]. If certificate checking is enforced, however, there may not exist an ideal solution so far. To overcome this issue, we propose to employ the code analysis technique for command identification. Code analysis intends to interpret the commands from the code level by employing hooking techniques (existing in all OSes) so that one can intercept and interpret the commands before they are sent, and thus, is a method of analyzing commands from the source other than collecting the traffics from the middle as aforementioned. We detail our implementation for the Android architecture in Section IV.

- **Command Direction.** Directing a command is a process of routing the command to the designated place, which, in our context, is the Emulation Layer. This process can be done by manipulating traffics such as altering the IP header of the packets or by manipulating codes. In Section IV, we detail our implementation of command direction to the Emulation Layer using Android hooking mechanisms.

We use a real-world example of interposing a "Turn On" command to a Wemo smart light for illustrating the above two processes. First, for command identification, we reverse-engineer the firmware of the device and find out the key term of the command, which is `SetBinaryState` in this case. We then set up a middlebox between the Wemo smart light and its corresponding app to monitor the traffics, and pay attention to any traffic that contains the term `SetBinaryState` to locate the command. Yet, the overall "Turn On" command is not just a `SetBinaryState` word, but a fairly long XML data block sent in three separated packets. Hence, we need to identify and reassemble all the packets related to this command. In the end, during the command direction process, we leverage the middlebox to redirect the command to the designated destination, i.e., the Emulation Layer, for further analysis.

D. Emulation Layer

After the commands are directed from the Interposition Layer, our next step is to evaluate whether the set of actions executed in a sequence could pose a threat to the current smart home system. We leverage a method based on DQN to complete this step. The reasons to employ DQN are threefold. First, it is non-trivial to match a smart home state with a risk score since such a process may be highly complicated and non-linear. In this case, the deep fully-connected network in DQN can help resolve the issue. Second, Q-learning, as one of the main reinforcement algorithms, fits our problem properly since it requires reward feedback (r in our case) from the environment of each state which helps reduce redundant calculations, while other machine learning and deep learning

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

algorithms such as SVM, Decision Tree, Bayesian Inference, CNN, LSTM, and so on, do not have such a structure, and hence, can hardly be applicable to our approach. Third, DQN can realize an end-to-end prediction by not having to enumerate all possible states, reducing tons of workloads from human efforts.

Based on the work done by Mnih *et al.* [14], we formulate the goal of DQN for our problem to maximize an action-value function (also known as the Q function) shown as follows. Here an action refers to an app command, or a human activity, or an environmental change that can cause a smart home state transition.

$$Q(\phi(S), A; \theta) = \max \mathbb{E}_\theta[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | S_t = S, A_t = A] \quad (1)$$

where t is the iteration step (each step corresponds to a chosen action), S_t is the state at step t, A_t is the chosen action by DQN at t, r_t is the risk level at t, γ is the risk discount used to determine the importance of future risks, ϕ is the function representing the output of the fully-connected feedforward network in the DQN, and θ is the model parameter for $Q(\phi(S), A; \theta)$ obtained from our trained neural network. Generally speaking, the Q function represents the cumulative future risks.

With a large number of training data, our goal is to minimize the loss function for the Q-learning network at each iteration i as follows:

$$L_i(\theta_i) = \mathbb{E}[(r + \gamma \max_{A'} \hat{Q}(\phi(S'), A'; \theta_i^-) - Q(\phi(S), A; \theta_i))^2] \quad (2)$$

where i is the iteration index for the episodes (each episode begins with a new normal state), $\hat{Q}(\phi(S'), A'; \theta_i^-)$ is the Q function for the target network with S' being the current state and $Q(\phi(S), A; \theta_i)$ is the Q function for the approximating network (the target network is used to compute the loss for each iteration and the approximating network is an estimation of the optimized situation at each iteration). The purpose of this loss function is to approximate our goal Q function $Q(\phi(S), A; \theta_i)$ to the maximum possible value of $\gamma \hat{Q}(\phi(S'), A'; \theta_i^-)$ given all choices of A'.

Motivated by the algorithm proposed in [14], we design a training algorithm for the DQN for CommandFence in Algorithm 1. The inputs to this algorithm include all possible normal smart home states $\{S_N\}$ and all possible actions $\mathscr{A}$, the risk threshold δ, and a greedy factor ϵ. We first initialize the approximating network Q and the target network $\hat{Q}$ to be identical with random weights (Lines 1-2). Then we set the total training episodes to M (Line 3), a sufficiently large number that is related to the total number of normal states but is much much larger than that to guarantee that no normal state can be missed for all the possible combinations of actions. At each time we begin a new training episode, we initialize the state to a randomly chosen normal smart home state (Lines 4). Next we set the total transition steps to T, a sufficiently large number that is related to the total number of actions but is much much larger than that to guarantee that all possible action combinations can be selected for each episode. At each step, we apply an ϵ-greedy policy algorithm to choose a proper action, and execute the action in our emulated smart home environment to retrieve a risk score (Lines 6-11) [30]. Afterwards, we set S_{t+1} to S_t and store the four-tuple (S_t, A_t, r_t, S_{t+1}) in memory so that one can sample from it later (Lines 12-14). At Line 15, we set a temporal value y_j to represent the Q value, which equals r_j if the risk score is greater than the risk threshold δ, and equals the discounted risk score otherwise. We then perform gradient descent on $(y_j - Q(\phi(S_j), A_j; \theta))^2$ with respect to θ (i.e., updating θ), and set $\hat{Q}$ to be equivalent to Q (Lines 16-17). In the end, we return the trained model parameter θ (Line 18).

Algorithm 1 Training Algorithm for our DQN

Input: The set of all normal smart home states $\{S_N\}$, the set of all choosable actions (commands and activities) $\mathscr{A}$, risk threshold δ, and greedy factor $\epsilon \in (0, 1)$.

Output: Model parameter θ

1: Initialize a Q function Q for the approximating network with random parameter θ
2: Initialize a Q function $\hat{Q}$ for the target network with parameter $\theta^- = \theta$
3: **for** episode$= 1 \cdots M$ **do**
4: Initialize a smart home state to a normal one, i.e., randomly choose S_1 from $\{S_N\}$
5: **for** $t = 1$ to T **do**
6: Generate a random number $\sigma \in (0, 1)$
7: **if** $\sigma \leq \epsilon$ **then**
8: Let $A_t = \arg\max_{A \in \mathscr{A}} Q(\phi(S_t), A; \theta)$
9: **else**
10: Choose A_t randomly from $\mathscr{A}$
11: Execute command A_t and retrieve a risk score r_t
12: Set $S_{t+1} = S_t$
13: Store the four-tuple (S_t, A_t, r_t, S_{t+1}) in memory
14: Sample (S_j, A_j, r_j, S_{j+1}) from memory
15: Set $y_j = \begin{cases} r_j & r_j \geq \delta \\ r_j + \gamma \max_{A'} \hat{Q}(\phi(S_{j+1}), A'; \theta^-) & o/w \end{cases}$
16: Perform gradient descent to minimize the loss $L(\theta)$ and update θ in the mean time, where the gradient is

$$\nabla_\theta (y_j - Q(\phi(S_j), A_j; \theta))^2$$

17: Set $\hat{Q} = Q$
18: **return** θ

Once a DQN is trained, given an initial smart home state (a normal state) and a command sequence, the DQN is able to predict whether the execution of the commands can lead to a risky state. The risk prediction algorithm is shown in Algorithm 2, which takes as inputs the model parameter θ trained by Algorithm 1, a command sequence $\mathcal{C}$ interposed from the smart home apps (from the Interposition Layer), an initial smart home state S_N obtained from the real smart home environment, a risk threshold δ, and the set of choosable actions $\mathscr{H}$ that include all possible human activities and environmental variations (i.e., excluding app commands), and

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

Algorithm 2 Risk Prediction

Input: A command sequence $\mathcal{C}$ passed from the Interposition Layer, an initial smart home state S_N, trained model parameter θ, a risk threshold δ, and the set of all choosable human activities and environmental variations $\mathscr{H}$.

Output: A binary outcome indicating if $\mathcal{C}$ can lead to a risky smart home state.

1: Execute the first command $\mathcal{C}[0]$ in S_N, then set S to be the new smart home state and retrieve r for S
2: **if** $r \geq \delta$ **then**
3: **return** "Risky"
4: Initialize a Q function Q with the trained model parameter θ and set $i = 1$
5: Set $A = \arg\max_{H \in \mathscr{H}} Q(\phi(S), H; \theta)$
6: **while** True **do**
7: Execute A in S, then update S to be the new smart home state and retrieve r for S
8: **if** $r \geq \delta$ **then**
9: **return** "Risky"
10: **if** $i = len(\mathcal{C})$ **then**
11: Exit loop
12: Set $A = \mathcal{C}[i]$
13: Execute A in S, then set S to be the new smart home state and retrieve r for S
14: **if** $r \geq \delta$ **then**
15: **return** "Risky"
16: Set $A = \arg\max_{H \in \mathscr{H}} Q(\phi(S), H; \theta)$
17: Set $i = i + 1$
18: **return** "Not Risky"

outputs a binary result indicating whether the execution of the command sequence starting from the smart home state S_N can lead to a risky state. Specifically, the algorithm first checks whether the first command in the sequence can lead to a risky state by executing it in the virtual environment (Emulation Layer); and if yes, the command sequence should be blocked (Lines 1-3). If the first command is not malicious, we next check whether any human activity or environmental change in the new smart home environment can incur a risky state (Lines 5 and 7-9). If not, we check next command (Line 12). This process is continued until all commands and the corresponding new states generated when they are executed are checked (Lines 6-17), and the algorithm returns "Not Risky" when the loop exits (Line 18). The rationale behinds our risk prediction lies in that a benign command can produce a risky smart home state when combined with human activities or environmental changes (e.g., opening window if no one is at home). Note that we only need to check the human activity or environmental variation that can maximize the expected future risk value for each command (Lines 5 and 16) as other activities are less risky.

The complexity of our overall approach can be determined based on that of Algorithm 1 and that of Algorithm 2. One can see that the complexity of Algorithm 1 is $O(M \cdot T)$ as its body involves two nested for loops of lengths M and T, respectively, where M is the total number of training episodes and T is the number of total iterations, while the complexity of Algorithm 1 is $O(len(\mathcal{C}))$ as the while loop exits when $i = len(\mathcal{C})$ (Lines 10-11) and i is updated at each round (Line 17), where $len(\mathcal{C})$ is the number of commands in the given command sequence. Hence, the complexity of our overall approach is $O(M \cdot T) + O(len(\mathcal{C}))$.

IV. Implementation of CommandFence

In this section, we detail our implementation of CommandFence, the digital-twin based smart home protection framework proposed in Section III. The overall implementation architecture is shown in Fig. 3.

A. Implementation of the Interposition Layer

As mentioned in Section III, the Interposition Layer involves two consecutive processes: command identification and command direction. Our implementation is carried out under the Android ecosystem.

1) Implementation of Command Identification: To identify and interpret a command via code-level analysis, we need to conduct reverse engineering on the related Android smart home apps. As we stated in Section III, the reason we leverage code-level analysis to interpret traffics lies in that man-in-the-middle methods cannot deal with encrypted traffics that are protected by verified certificates. Note that when performing code-level analysis, we do not need to reverse-engineer all the apps we tested since the traffic transmission methods are extended from several fundamental functions, which are to be unfolded below.

When an Android app is compiled into an executable, it is represented as a binary named Android package (`apk`) that can be directly executed by the Android system. Reversing an `apk` binary into a human-readable source requires three steps: transforming the `apk` to a Smali source, transforming the Smali source to a DEX, and transforming the DEX to a JAR. In our implementation, we utilized Apktool, Smali, and Dex2Jar to respectively handle these three steps [31]. Upon retrieving the human-readable source, i.e., Java in this case, the next step is to recognize the function calls that are related to the invocation of the smart home commands. In our consideration, whenever a command is generated from a smart home app, it needs to be sent out upon a smartphone. Current smartphones support only two ways for wireless data transmissions, namely Wi-Fi and Bluetooth. Therefore, a command is sent out either through Wi-Fi or Bluetooth. Thus we mainly investigated the Wi-Fi and Bluetooth related functions.

Normally, a developer of smart home apps tends to use the standard APIs recommended in the Android documentation for implementing Wi-Fi and Bluetooth transmissions [32]. To implement the Wi-Fi transmissions, the developer can use either high level APIs, i.e., `HttpURLConnection` and `HttpsURLConnection` (encrypted), or a low level API, i.e., `java.net.Socket`. To implement Bluetooth transmissions, the developer can

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

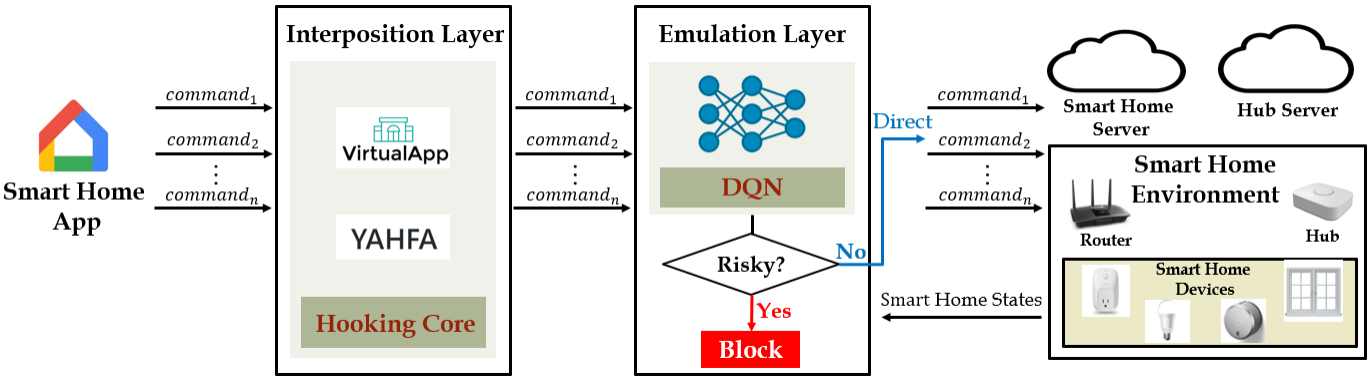

Fig. 3: The overall structure of our implementation.

only use `BluetoothSocket` for connection and communications. After a careful scrutiny, we found that all the APIs mentioned above, regardless of Wi-Fi or Bluetooth, construct commands by calling a member method called `getOutputStream`, which returns a referenced body of a command in a `java.io.OutputStream` object. Afterwards, the reference is passed to a `java.io.Writer` object, which calls the `write` function to write the main body of the command. Hence, we hooked the `write` function in the `Writer` class, as well as the `write` functions in the related subclasses extended from `Writer`, e.g., `BufferedWriter`, `PrintWriter`, `StringWriter`, `OutputStreamWriter`, etc. Finally, we examined if the string resource passed into these functions contains the keyword of a targeted command. If it does, the string itself is the command. To figure out the keyword, we simply sent the targeted commands multiple times, manually analyzed the string contents, and extracted the most semantically related text as the keyword for the command.

It is also not uncommon for a smart home app developer not to use the standard APIs for security purposes – the developer can implement some key codes at the native C/C++ level (Android NDK) or adopt other third-party frameworks [33]. If this is the case, one needs to reverse-engineer the binary of the target smart home app, i.e., the `apk` file, to specifically look for the functions used for communications. In this paper, we mainly used the Samsung SmartThings platform for our performance evaluation; therefore, we reverse-engineered its corresponding Android app, the SmartThings app (the full package is `com.smartthings.android`), and found out that SmartThings app actually leverages a third-party framework, `Retrofit`, which was developed and maintained by Square, Inc [34]. A command from a SmartApp running on the SmartThings app is sent using the construction function of `retrofit2.RequestBuilder` with the format shown as follows:

```
api/devices/{deviceId}/commands/action/{tileAction}
```

where `deviceId` is the id of the smart home device this command targets, and `tileAction` is the command body.

Note that we do not need to reassemble packets of a command since code analysis does not interpret commands at the traffic level.

2) Implementation of Command Direction: As mentioned in Section III, manipulating traffics for command direction is tedious and error-prone. Therefore, after a careful scrutiny, we decided to leverage the Android API hooking to handle this implementation because the hooking framework is the only known technique that is capable of rewriting a function call without modifying the original codes of a target app.

In this implementation, we chose VirtualApp and Yet Another Hook Framework for ART (YAHFA) to accomplish the hooking process [35] [36]. We intended not to use the traditional Android hooking frameworks Xposed and Frida that were frequently adopted by other researchers because the Xposed framework requires rooting the target devices and has severe drawbacks when being migrated to Android versions above 5.0 [37], and Frida requires a master host sending commands via cable, making it less mobile and flexible [38]. The main hooking process is shown as follows.

- First, VirtualApp substitutes the original `BinderProxy` with `BinderProvider`, which contains all core Android system components.
- Second, every time a hooked Android app launches a function call, the request would be replaced by `StubActivity` proxied at VirtualApp, which then forwards the request to `BinderProvider` and registers a corresponding callback function.
- Third, YAHFA is attached to the VirtualApp process, and checks if the function is a target function to be hooked (called `originMethod` in YAHFA) based on the hook plugin. Then it copies the target function codes for execution.
- Fourth, if YAHFA finds that an `originMethod` should be hooked, it makes a backup for this function (called `backupMethod` in YAHFA), then changes the entry point `entry_point_from_jni_` of `originMethod` to the one of the hook function, i.e., the injected function (called `hookMethod` in YAHFA).
- Last, `YAHFA` changes the entry of the assembly codes

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

`entry_point_from_quick_compiled_code_` of the `originMethod` to the one of the `hookMethod`.

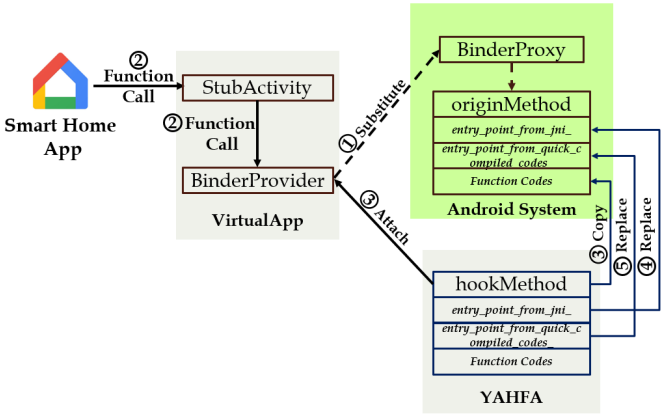

Fig. 4: The process of the hooking procedure.

The above process is illustrated in Fig. 4. The overall hooking procedure was implemented with a total of 51,338 lines of codes in Java and native C/C++. Upon completing the procedure, one can basically hook any function calls in any Android app. We leverage this procedure to hook the function calls mentioned above to re-direct the commands to our Emulation Layer.

B. Implementation of the Emulation Layer

To construct a virtual smart home environment, we wrote 1,230 lines of python codes to define the states, parse the real smart home sensing data, describe actions, and simulate state transitions. We also wrote 395 lines of python codes to build the DQN model. In the following we employ the parameters defining a real smart home environment presented in Section V, to concretely demonstrate our implementation of the Emulation Layer.

After multiple testings with the data collected from the real smart home system (see Section V), we decided to implement a three-layer fully-connected network trained by DQN, with the first layer having 38 neurons, where 38 is the total number of devices in our smart home testbed, since we wish the network to capture just as much information as a smart home state has to ensure the lowest possibility of suffering from an overfitting problem. The number of neurons in the second layer is a constant 8 representing a bottleneck layer. The number of neurons in the last layer is 26, which is the total number of actions we considered since we hope that our network can capture just as much information as all actions have for the purpose of ensuring the lowest possibility to suffer from an overfitting problem. The first and second layers adopt the rectified linear unit (ReLU) as the activation function [39] while the last layer adopts a simple matrix multiplication without activation function. Finally, we trained this three-layer network based on Algorithm 1, and implemented the prediction phase based on Algorithm 2, to complete the functions of DQN. All the codes are written with Python and the Google deep-learning framework Tensorflow [40].

V. Performance Evaluation

In this section, we demonstrate the performance of our CommandFence implementation. We first present the evaluation results of the Interposition Layer and the Emulation Layer; then we evaluate the overall performance of CommandFence; finally we report the latency of CommandFence.

Note that for convenience in our performance evaluation studies we considered the interactions of app commands and human activities only, deliberately ignoring the impact of environmental changes, as human activities and environmental changes basically play the same rule in CommandFence: they cause the smart home state changes, which then trigger the issues of app commands. On the other hand, it is not practical for us to establish a smart home testbed that can control a physical home to capture the impact of environmental variations on smart home states, making it impossible for us to collect the corresponding training data.

A. Performance of the Interposition Layer

To evaluate the performance of the Interposition Layer, we mainly considered the success rate of hooking the commands generated by the target smart home apps. For this purpose, we downloaded the top 15 standalone smart home apps with the highest ratings in the Google Play Store. We also downloaded the SmartThings App which serves as the integrated runtime environment for all Samsung smart home apps (SmartApps) we evaluated. Therefore, the SmartThings App can be treated as a collection of a large number of SmartApps. Then we started these 15 smart home apps and the SmartThings App in our hooking framework running on two real Android devices, Google Nexus 7 and Amazon BLU R2, and manually launched the commands and see if our framework manages to hook the related API calls. As a result, the Interposition Layer successfully hooked all the API calls related to the command generations without interfering with the irrelevant functions. We detailed the specifications of our hooking method against the top 8 smart home apps in Table I.

B. Performance of the Emulation Layer

To evaluate the performance of the Emulation Layer, we first validated our DQN using the training and testing data obtained from a real smart home testbed we established. Then we evaluated the Emulation Layer as a whole using real SmartApps on the SmartThings platform with our trained DQN model.

1) Data Collection from a Real Smart Home System: In order to build an emulated smart home environment, we need to have data from a real smart home system for training purpose. Therefore, we assembled four sets of 7 passive devices with Arduino (sensors that can passively record ambient sensing data). The specifications of the selected sensors are listed in Table II. Additionally, we had an Apple Watch Series 3 as our sleep sensor which can record a user's heartbeat and tell if the user is sleeping or awake: if the user's heartbeat is lower than 50 beats per minute (BPM), it means this user is sleeping. Hence, we had a total of 29 passive devices. Finally,

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

TABLE I: Specifications of hooking the top 8 smart home apps of the Google Play Store.

Package Name	Hooking Method	Command Keyword
com.att.shm	HttpWriter->writeString	content
com.remotefairy4	WriterBasedGenerator->write	net_cmds
com.tuya.smartlife	PrintWriter->write	switch
com.belkin.wemoandroid	PrintWriter->write	SetBinaryState
mobile.alfred.com.alfredmobile	JsonWriter->write	action
com.google.android.apps.chromecast.app	BufferedWriter->write	scan_wifi
com.helloinspire.glasshouse	PrintWriter->write	CameraLauncher
com.tplink.kasa_android	Writer->write	transition_light_state

TABLE II: Specifications of the 7 sensors.

Sensor Type	Chip/Module ID	Physical Meaning and Data Structure	Value Range and Precision	Output Protocol
Smoke Sensor	MQ-2	Gas Concentration (Integer)	1-1000 PPM	Analog Voltage
Temperature Sensor	MLX9061	Centigrade (Float)	0-100 °C (2-digit decimals)	I2C
Humidity Sensor	DHT11	Humidity Level (Integer)	5-95 RH	Serial
Light Sensor	BH1750	Luminance (Integer)	1-65535 LX	I2C
Sound Sensor	LM386	Sound Strength (Integer)	20-200 DB	Analog Voltage
Flame Sensor	LM393	Flame Occurence (Binary Digit)	0 or 1	Digital Voltage
Motion Sensor	HC-SR501	Motion Occurence (Binary Digit)	0 or 1	Digital Voltage

we adopted 9 active smart devices, i.e., 4 smart lightbulbs (1 in the living room, 1 in the bedroom, 1 in the kitchen and 1 in the bathroom), 2 switches for opening/closing the windows (1 in the living room and 1 in the bedroom), 1 smart door lock, 1 smart alarm, and 1 smart surveillance camera. Therefore, we had in total 38 devices, i.e., the size of an input smart home state to the DQN is 38. We deployed these 38 devices at two different apartments (the same 38 devices were moved from one apartment to another), and asked three volunteers to live in each apartment for 7 days to collect data. The volunteers randomly performed 21 smart home commands and 5 daily activities to simulate a real routine home environment, and hence, we had 26 actions in total as shown in Table IV. This means that our DQN can choose one in 26 actions at each iteration. As a result, we recorded 232,315 smart home states, among which 2,887 were collected while the volunteers were showering, 33,954 were collected while they were sleeping, 3,909 were collected while they were cooking, 32,468 were collected while they were entering the home, and 128,030 were collecting while nobody is at home. Additionally, we defined four risk levels with $r = 0$ signals "not risky". The risk level $r = 1$ reflects a least vulnerable state where a user may suffer from privacy leaks but no actual property loss could happen, e.g., lights are turned on when nobody is at home. The risk level $r = 2$ reflects a moderately risky situation where a user may suffer from property loss, e.g., the camera is disabled when the user is sleeping. The risk level $r = 3$ reflects a dangerous situation where a user can experience physical damage with little additional attack efforts, e.g., strobing lights causing possible seizures.

To better assess the performance of CommandFence, we program a Python script with 116 lines of codes to automatically label all the ground-truth data, i.e., to label the collected 232,315 smart home states, even though DQN does not need these labels. This labeling is based on common senses according to the aforementioned definitions of risk levels. More specifically, we considered 19 risky scenarios in our experiment and the criteria for labeling the risk level of a state by the script is reported in Table III. These labels are treated as ground-truth results, which will be compared with the predicted results of DQN to assess the accuracy of DQN prediction.

TABLE III: The 19 cases based on which our automated program labels the risk levels of the states.

Case ID	State Description		Risk Level
	User Status	Home Status	
1	Out of Home	Lights On	1
2	Sleeping	Lights On	1
3	Out of Home	Windows Opened & Camera Off	2
4	Sleeping	Windows Opened & Camera Off	2
5	In Bathroom	Windows Opened & Camera Off	2
6	Out of Home	Door Unlock	3
7	Sleeping	Door Unlock	3
8	In Bathroom	Door Unlock	3
9	Out of Home	Fire Detected & Fire Alarm Off	3
10	Sleeping	Fire Detected & Fire Alarm Off	3
11	In Bathroom	Fire Detected & Fire Alarm Off	3
12	Out of Home	Fire Detected & Windows Closed	3
13	Sleeping	Fire Detected & Windows Closed	3
14	In Bathroom	Fire Detected & Windows Closed	3
15	Any Status	Fire Detected (Except Kitchen)	3
16	Any Status	Smoke Detected & Smoke Alarm Off	3
17	Any Status	Toxic Gas Detected & Windows Closed	3
18	Any Status	Lights Strobing	3
19	Any Status	All Other Status	0

2) Evaluation of the DQN Model: We set M to be 10,000,000 since such a large value helps DQN to better recognize all states; we set T to be 300 since 300 is big enough to cover 26 actions (described in Section V-B1); we set the risk discount γ to be 0.8 which is a commonly used value for DQN; and we set ϵ to be 0.8 since we wish to have a high chance of choosing an action with the highest Q value while leaving a small chance to choose an action with other Q values. Then for each risk level $r = 0, 1, 2, 3$, where $r = 0$ corresponds to the scenarios without any risk and other r values are defined in Section V-B1, we randomly sampled 500 pieces from the 232,315 collected smart home states as the test data and used the leftover as the training data. Then we

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

TABLE IV: Details of the 26 actions, including 21 smart home commands and 5 human activities.

Type of Actions	Descriptions	Notation	Type of Actions	Descriptions	Notation
Command	Turn on Lights	c_1	Command	Shutdown Camera	c_{14}
Command	Turn off Lights	c_2	Command	Turn on Camera	c_{15}
Command	Unlock Door	c_3	Command	Take Photos with Camera	c_{16}
Command	Lock Door	c_4	Command	Dim Lights	c_{17}
Command	Raise Fire Alarm	c_5	Command	Brighten Lights	c_{18}
Command	Disable Fire Alarm	c_6	Command	Shutdown Water Supply	c_{19}
Command	Send Message Smart Speaker	c_7	Command	Turn Heater Up	c_{20}
Command	Open Windows	c_8	Command	Turn Heater Down	c_{21}
Command	Close Windows	c_9	Human Activity	Cook	h_1
Command	Raise Smoke Alarm	c_{10}	Human Activity	Shower	h_2
Command	Disable Smoke Alarm	c_{11}	Human Activity	Sleep	h_3
Command	Get Door Lock Battery Status	c_{12}	Human Activity	Not At Home	h_4
Command	Change Door Lock Pin	c_{13}	Human Activity	Entering Home	h_5

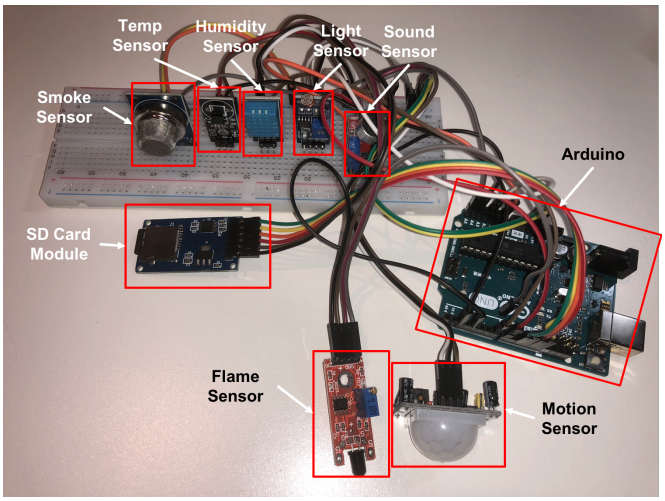

Fig. 5: A set of 7 passive devices configured in a real home environment to collect data for training and test purposes.

employed Algorithm 1 to train the DQN based on the training data and manually checked if the outputs of Algorithm 2 on the test data were reasonable based on common sense. For each risk level we repeated the above process twice. The results indicate that the overall prediction accuracy of our DQN model is 86.3%, which means that among the 4000 test states, 3452 were correctly interpreted by the DQN while the other 548 states (13.7%) were misinterpreted. More specifically, for the 1000 risk-free states, 763 were correctly predicted (prediction accuracy is 76.3%), while for $r = 1, 2, 3$, the numbers of states that were correctly predicted were respectively 840, 866, and 983 (the corresponding accuracies were respectively 84%, 86.6%, and 98.3%). One can see that the accuracy is lower for smaller r values. Such a phenomenon may be justified as follows. When r is smaller, the loss function drops more quickly and the gradient descent function ends sooner. On the other hand, from the definition of the Q function, a smaller r value implies a smaller feedback value. In both cases, the training process may stop too early, making the model under-trained and less accurate.

To better demonstrate the advantage brought by DQN, we conducted experiments by training the aforementioned three-layer network in a traditional stochastic gradient descent way, instead of applying Algorithm 1, resulting in an instance of multi-layer perceptron. Then we repeated the above process (i.e., randomly sampling 4000 states for testing and using the leftover for training) for training and predicting, and obtained an overall accuracy of 72.8%, which is much less than that of the case with DQN (86.3%).

C. Performance of the Overall CommandFence System

To evaluate our implementation of the CommandFence system, we need to consider the two threat models introduced in Section II, i.e, malicious apps with pure evil intentions and benign ones with logic errors. In order to do so, we leverage the Samsung SmartThings platform as the testbed since it is the largest and most influential platform for smart home systems [7]. We carried out two sets of experiments for the two threat models. For each set, we underwent a similar two-step procedure: (1) we wrote a generic python program to automatically extract the sequence of smart home commands and their corresponding human activities (i.e. actions) in a SmartApp; (2) we then employed the DQN model trained with the collected 232,315 smart home states, ran Algorithm 2 for the command sequences extracted from these SmartApps, and checked if they can lead to a risky smart home state.

1) Effectiveness Against Evil SmartApps: To evaluate the effectiveness of CommandFence against the SmartApps with pure evil intentions, we used the 10 SmartApps created by Jia *et al.* that were designed with only evil intentions [1]. CommandFence successfully identified 7 of them as dangerous to the smart home environment such as the one that turns on a fake alarm when CO density is normal. The other three SmartApps were deemed as riskless since they do not lead to any risky smart home state as they only cause information leakage (such as leaking contacts and smartphone side-channel

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

data) on the smartphone side. While ContexIoT, proposed by Jia *et al.*, managed to identify 8 of these malicious SmartApps [1]. However, ContexIoT requires a fairly long time for profiling the behavioral contexts of a user before it can identify the apps as malicious.

2) Effectiveness Against Benign SmartApps with Logic Errors: To evaluate the effectiveness of CommandFence against benign SmartApps with logic errors, we manually downloaded 553 SmartApps from the official SmartApp distribution community [41] for testing. CommandFence was able to identify 44 SmartApps that may be risky. After a careful manual analysis, we identified that 34 (77.27%) of them could indeed result in a risky state due to careless implementations such as turning on a light when nobody is at home, resulting in possible privacy leakage. Note that since the security of SmartApps has been extensively studied in multiple works [2], [7], [8], [15], most problematic SmartApps have already been patched or deleted from the community. To the best of our knowledge, among the 34 SmartApps tagged risky by CommandFence, 31 of them were newly identified and were never reported before. In addition, we utilized the 17 SmartApps in `MALIOT` written by Celik *et al.* [2] for testing. These SmartApps have logic errors such as "the speaker is turned on while the user is sleeping" and "the fire alarm sounds when there is no fire detected". As a result, CommandFence successfully tagged all of these problematic SmartApps, yielding a 100% success rate. Table V summaries the performance results of CommandFence.

TABLE V: Performance results of CommandFence.

Threat Model Type	Source of Test Samples	Success Rates
Benign SmartApps with Logic Errors	`MALIOT` Written by [2]	17 out of 17 (100%)
	Official SmartApps Community	34 out of 44 (77.27%)
Evil Malicious SmartApps	Created by [1]	7 out of 10 (70%)

D. Latency Analysis

To evaluate the latency of CommandFence, we downloaded the official SmartThings Android app and installed 20 SmartApps on the SmartThings app (10 of them were normal apps and the other 10 were dangerous ones created by Jia *et al.* [1]). Then we set up our own `device handler` which serves as a mandatory part to define how the devices should react to the functions in a SmartApp [42]. The detailed method for hooking the SmartThings app is illustrated in Section IV. For these 20 SmartApps, we measured the elapsed times for their operations with and without CommandFence. Then we took the difference between these two elapsed times as the latency of CommandFence. Our results showed that CommandFence imposes an averaged latency of 0.1675 seconds. The distribution of the latencies of these 20 SmartApps is shown in Fig. 6.

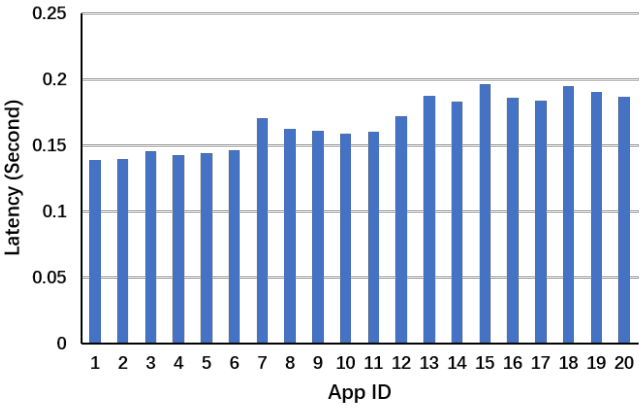

Fig. 6: The distribution of the latency of CommandFence tested over 20 SmartApps, with the App IDs from 1 to 10 being normal SmartApps while the APP IDs from 11 to 20 being dangerous ones created by [1].

VI. Related Works

With the increasing number and type of IoT devices deployed in our daily life, more and more studies were carried out to investigate the security issues of IoT. Based on the survey conducted by Xiao *et al.*, one can classify the major related works into two categories, i.e., *attack-based* and *defense-based* [43].

A. Attack-Based Studies

Existing attack-based studies investigated either app-based attacks or non-app-based attacks. App-based attacks use malicious apps to launch attacks on a system. Zhang *et al.* discovered that the platforms for home-use virtual personal assistants (VPA), i.e., home speakers, have employed coarse-grained access control and managed to create malicious apps that can launch two new attacks: voice squatting and voice masquerading [44]. Ronen *et al.* developed a worm that exploits the vulnerabilities in the Zigbee light link protocol in Philips Hue [45]. Xiao *et al.* identified multiple vulnerabilities in the current home-use electroencephalography (EEG) system allowing an attacker to carry out remote attacks with malicious apps and proximate attacks with a radio receiver to steal a user's cerebral information [46]. Jia *et al.* designed an algorithm to automatically identify vulnerabilities in smart home traffics and implemented six attacks based on malicious apps [15]. Zhou *et al.* invented a composite method to analyze IoT firmware, mobile apps, and clouds all together to locate vulnerabilities among their communications. Then they proposed two types of attacks using "phantom devices", i.e., malicious apps to breach a user's privacy [47]. App-based attacks are under our consideration in this paper but our intention is to provide a preventive measure such that the app commands, whether from malicious apps or benign ones, that may lead to risky states are dropped before they can be executed in the smart home environment.

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2022.3184185

Non-app-based attacks are those using all means other than apps to launch attacks. Zhang *et al.* managed to create inaudible sounds through ultrasonic carriers against speech recognition systems and successfully took full control over a few most well-known speech recognition systems, such as Google Now, Samsung S Voice, Huawei HiVoice, Cortana, and Alexa [48]. Sugawara *et al.* took a step forward by successfully encoding commands into laser lights. If the speakers or voice devices are hit by the lights, they execute whatever commands encoded in the lights, making this attack a highly alerting threat in real world [49]. Eberz *et al.* formulated a signal injection attack against electrocardiography (ECG) biometrics through a wristband named Nymi Band and achieved a success rate of 81% [50]. Li *et al.* utilized the encrypted traffic information generated through video surveillance systems to launch side-channel attacks for analyzing the users' activities [51].

B. Defense-Based Studies

Defense-based studies focus on either access control or non access control. Access-control-based approaches are mainly centered on the developments of new access control mechanisms to mitigate current threats brought by malicious or logically flawed apps. Jia *et al.* proposed a context-based permission system named *ContextIoT*, to perform access control based on a user's behavioral history [1]. Tian *et al.* presented *SmartAuth* in inhibiting the potential overprivileged SmartApps by matching the code-level implementations and the descriptions of the SmartApps [8]. Fernandes *et al.* put forward *FlowFence*, a permission control system that achieves security goals in IoT by asking the intended data flow patterns for sensitive data use [52]. Birnbach *et al.* proposed a sensor-fusion-based mechanism to automatically identify a user's activities based on the sensor statuses for verification [53]. Fernandes *et al.* made use of the concept of decentralization and proposed a shared rule-specific token called *XToken* to prevent permissions from being misused [9]. Liu *et al.* leveraged the Trusted Execution Environment (TEE) and blockchain technologies to realize fine-grained and accountable access control and overcome the over-privilege issues in IoT systems [54]. Yet, all these methods are permission-based, which may be insufficient to provide ideal protections as we mentioned in Section II.

Non-access-control-based studies refer to those that intend to improve the IoT security without using access control mechanisms. Chen *et al.* proposed an automatic fuzzing framework called *IoTFuzzer*, to discover the memory corruptions in IoT devices without accessing the firmware images [10]. Xu *et al.* presented a system named *CIDER*, to quickly recover an already compromised IoT device even if an attacker took root control of it [11]. Zheng *et al.* introduced a lightweight fuzzing framework termed *Firm-AFL*, for IoT firmware vulnerability identification using user-mode QEMU [12]. Feng *et al.* developed a black-box fuzzing technique, namely Snipuzz, for identifying vulnerabilities within IoT firmware [55]. Aafer *et al.* described a log-based dynamic fuzzing approach to discover vulnerabilities hidden in Android SmartTVs [56]. The five works mentioned above mainly enhance security from the device side, which may not be effective in defending an IoT system as a whole. Qi *et al.* proposed a novel protective framework termed *iRuler* using information flow graph and natural language processing, to identify function-level vulnerabilities such as looping, conflict, duplication, etc [13]. We also carry out non-access-control-based studies in this paper, but our goals are different from those mentioned above. We intend to find out the app commands whose executions may be harmful to a smart home environment when interacting with human behaviors and environmental variations, while existing works aim to find out vulnerabilities in apps without considering the dynamism of their operating environment.

VII. Conclusions and Future Research

In this paper, we proposed CommandFence, a novel preventive framework to secure smart home systems from reaching risky states when app commands combined human activities and environmental variations are executed. CommandFence is based on digital-twin, a new concept that is fundamentally different than those employed by the existing well-received access control frameworks. It is composed of an Interposition Layer and an Emulation Layer to first interpose the app commands and then execute them in a virtual smart home environment that can predict whether the operations of the commands combining with human activities and environmental variations may transition a normal smart home environment to a dangerous one, and if yes, the commands are dropped before they are sent to the real smart home system. CommandFence does not require any hardware update, it can be adopted as a software plugin, and it is orthogonal to the existing protection frameworks, providing another layer of security for smart home systems. Our extensive experimental studies indicated that CommandFence is effective and efficient. It successfully identified 31 brand-new problematic SmartApps out of the 553 commercial SmartApps, tagged 7 out of 10 malicious SmartApps as risky, and recognized all the 17 logically-erroneous SmartApps. CommandFence incurs a neglectable overhead of 0.1675 seconds, thus having little negative impact on the normal operations of smart home systems.

This is an exploratory work of employing the novel concept of digital-twin for preventive security in smart home systems. Even though CommandFence can effectively mitigate the security problems brought by the two threat models we considered in this paper, its current design can only deal with threats from the smart home apps installed in a user's smartphone, while in practice threats may come from other sources including remote attackers from the external environments. Therefore, we will consider the design of defensive mechanisms embedded into smart devices, endowing them with the capabilities to hinder attacks from all sources. On the other hand, the detection accuracy and false-alarm rate of CommandFence could be further improved if we can collect more training data from a broader range of scenarios and employ the recently developed advanced reinforcement learning techniques [57]–[59] to cope

with the more complicated variations including those that are not human-related.

A more complicated smart home scenario, which can be extended to a smart office setting, is one with a large number of sensing devices deployed. Since our design of CommandFence heavily relies on the environment (a property of DQN), it is nontrivial to directly extend its implementation presented in this paper to handle such cases. At the minimum we need to change the current structure of the Emulation Layer and re-train the entire neural network since the number of neurons in the first layer is equal to the number of devices and the number of neurons in the third layer is equal to the number of actions. To avoid a prohibitively large neural network, we intend to proceed from three directions in our future research to avoid completely retrain the whole network (i.e., via fine-tuning the existing one). First, if the sensors are densely deployed, it is likely that the sensing data of nearby devices are correlated, meaning that the dimension of the input data to the first layer can be greatly reduced [60] [61] [62]. Second, if the devices are sparsely located in a large area, the sensing data may not be correlated, but one can ameliorate the hardness by first employing anomaly detection algorithms to distinguish the anomaly data from the normal one [63] then feeding the anomaly data to the neural network. Third, considering the scenario with a large number of devices densely deployed in a large area, the methods mentioned above can be combined, and other reinforcement learning techniques can also be explored to handle the challenges.

Another line of future research we intend to consider is to extend the concept of digital-twin based preventive security to more general IoT settings such as industrial IoT (IIoT), smart health, smart grid, and even smart transportation systems, as it has a high potential of revealing vulnerabilities that could not be identified via traditional protection mechanisms. However, migrating the basic idea of CommandFence to these systems is hard as they are mechanically and technologically different than the smart home environments under our consideration. Intuitively, application-specific and complex modifications to our design and implementation are unavoidable. For example, if we intend to migrate our approach to an IIoT scenario, we have to implement the IIoT-version of Interposition Layer and Emulation Layer. To implement the Interposition Layer, we need to intercept the voltage signals emitted by programmable logic controllers (PLCs), which are routed through industrial routers and then directed to the designated actuators such as relays, transistors, servos and motors [64]. Yet, in most IIoT systems, the PLCs and routers are not open for re-program, meaning that we cannot inject the codes for intercepting the voltage signals. Obviously, designing new PLCs and routers with interception codes embedded inside to replace the old ones is unlikely to be adopted since developing a PLC or a router is exceedingly difficult and is usually accomplished by highly experienced corporations such as Siemens, Beckhoff and Schneider. A feasible approach is to implement (hardware and software) a standalone voltage interceptor and place it between the PLCs and the routers. Then whenever a PLC emits a voltage signal, the interceptor can intercept the signal and direct it to the Emulation Layer for further analysis. To implement the Emulation Layer, we need to first develop a program reversing the voltage back to some human-readable languages such as ladder diagram, instruction language, function block diagram, or sequential function chart, which would be interpreted as commands. We then need to implement a digital-twin version of the IIoT system which should include as many digital actuators as possible, e.g., sensors, relays, motors, servos, CNC machines. Lastly, we can execute the commands in the digital-twin environment and employ learning algorithms to complete different tasks such as active defense, fault diagnosis [64], vulnerability identification and accident avoidance. One can see that the overall migration process to IIoT requires tremendous efforts to accomplish. Nevertheless, the potential of exploiting digital twin to develop defense mechanisms for more complicated systems such as IIoT is definitely worthy of exploration.

Acknowledgment

This study was partially supported by the National Key R&D Program of China (2019YFB2102600), the National Natural Science Foundation of China (62002067, 61832012, 61971014 and 11675199), and the US National Science Foundation (IIS-1741279, CNS-2105004 and CNS-1704397).

References

[1] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. J. Unviersity, "Contexiot: Towards providing contextual integrity to appified iot platforms," in *Proceedings of The Network and Distributed System Security Symposium*, vol. 2017, 2017.

[2] Z. B. Celik, P. D. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," *CoRR*, vol. abs/1805.08876, 2018. [Online]. Available: http://arxiv.org/abs/1805.08876

[3] "Samsung smartthings," 2018. [Online]. Available: https://www.smartthings.com/

[4] "Smart home market," 2018. [Online]. Available: https://www.statista.com/outlook/279/100/smart-home/worldwide

[5] D. Djenouri, R. Laidi, Y. Djenouri, and I. Balasingham, "Machine learning for smart building applications: Review and taxonomy," *ACM Comput. Surv.*, vol. 52, no. 2, mar 2019. [Online]. Available: https://doi.org/10.1145/3311950

[6] "Malicious app attacks," https://venturebeat.com/2018/05/30/smart-home-speakers-are-vulnerable-to-a-variety-of-attacks/, 2018.

[7] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 636–654.

[8] 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)*. Vancouver, BC: USENIX Association, 2017, pp. 361–378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian

[9] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash, "Decentralized action integrity for trigger-action iot platforms," in *Proceedings of The Network and Distributed System Security Symposium*, 2018, pp. 1–16.

[10] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *Proceedings of The Network and Distributed System Security Symposium*, 2018, pp. 1–15.

[11] M. Xu, M. Huber, Z. Sun, P. England, M. Peinado, S. Lee, A. Marochko, D. Mattoon, R. Spiger, and S. Thom, "Dominance as a new trusted computing primitive for the internet of things," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1415–1430.

[12] 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 *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[13] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. A. Gunter, "Charting the attack surface of trigger-action iot platforms," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 14391453. [Online]. Available: https://doi.org/10.1145/3319535.3345662

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, no. 7540, pp. 518–529, 2015.

[15] Y. Jia, Y. X. Xiao, J. Yu, X. Cheng, Z. Liang, and Z. W. Wan, "A novel graph-based mechanism for identifying traffic vulnerabilities in smart home iot," in *Computer Communications, IEEE INFOCOM 2018-The 37th Annual IEEE International Conference on*. IEEE, 2018, pp. 1–9.

[16] "August smart home," 2018. [Online]. Available: https://august.com/

[17] "Tp-link smart home," 2018. [Online]. Available: https://www.tp-link.com/us/home-networking/smart-home/

[18] "Wemo smart home," 2018. [Online]. Available: http://www.belkin.com/us/Products/home-automation/c/wemo-home-automation/

[19] Z. Aung and W. Zaw, "Permission-based android malware detection," *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[20] D. Preuveneers and W. Joosen, "Smartauth: dynamic context fingerprinting for continuous user authentication," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 2185–2191.

[21] "Ifttt," 2018. [Online]. Available: https://ifttt.com/

[22] "Zapier," 2018. [Online]. Available: https://zapier.com/

[23] "Microsoft flow," 2018. [Online]. Available: https://flow.microsoft.com/en-us/

[24] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the internet of things," *arXiv preprint arXiv:1802.04410*, 2018.

[25] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, "Fairaccess: a new blockchain-based access control framework for the internet of things," *Security and Communication Networks*, vol. 9, no. 18, pp. 5943–5964, 2016.

[26] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779

[27] S. Wu and J. Liu, "Overprivileged permission detection for android applications," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, May 2019, pp. 1–6.

[28] Y. Jia, Y. Xiao, J. Yu, X. Cheng, Z. Liang, and Z. Wan, "A novel graph-based mechanism for identifying traffic vulnerabilities in smart home iot," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, April 2018, pp. 1493–1501.

[29] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787502

[30] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," in *Annual Conference on Artificial Intelligence*. Springer, 2011, pp. 335–346.

[31] P. O. Fora, "Beginners guide to reverse engineering android apps," in *RSA Conference*, 2014, pp. 21–22.

[32] "Documentation for app developers," 2018. [Online]. Available: https://developer.android.com/docs/

[33] "Android ndk," 2018. [Online]. Available: https://developer.android.com/ndk/

[34] "Retrofit," 2018. [Online]. Available: http://square.github.io/retrofit/

[35] "Virtualapp," 2018. [Online]. Available: https://github.com/asLody/VirtualApp

[36] "Yet another hook framework for art," 2018. [Online]. Available: https://github.com/rk700/YAHFA

[37] "Android xposed," 2018. [Online]. Available: http://repo.xposed.info/

[38] "Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." 2018. [Online]. Available: https://www.frida.re/docs/home/

[39] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[41] Samsung, "Smartthings community," https://community.smartthings.com/, 2018.

[42] ——, "Device handlers," https://docs.smartthings.com/en/latest/device-type-developers-guide/, 2018.

[43] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, Aug 2019.

[44] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian, "Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1381–1396.

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

[46] Y. Xiao, Y. Jia, X. Cheng, J. Yu, Z. Liang, and Z. Tian, "I can see your brain: Investigating home-use electroencephalography system security," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6681–6691, Aug 2019.

[47] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1133–1150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zhou

[48] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu, "Dolphinattack: Inaudible voice commands," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 103–117.

[49] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu, "Light commands: Laser-based audio injection attacks on voice-controllable systems," in *Proceedings of the 29th USENIX Security Symposium*, August 2020, pp. 2631–2648.

[50] S. Eberz, N. Paoletti, M. Roeschlin, M. Kwiatkowska, I. Martinovic, and A. Patané, "Broken hearted: How to attack ecg biometrics," in *Proceedings of The Network and Distributed System Security Symposium*, 2017, pp. 1–15.

[51] H. Li, Y. He, L. Sun, X. Cheng, and J. Yu, "Side-channel information leakage of encrypted video stream in video surveillance systems," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[52] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks." in *USENIX Security Symposium*. USENIX Association, 2016, pp. 531–548.

[53] S. Birnbach, S. Eberz, and I. Martinovic, "Peeves: Physical event verification in smart homes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS 19. New York, NY, USA: Association for Computing Machinery, 2019, p. 14551467. [Online]. Available: https://doi.org/10.1145/3319535.3354254

[54] C. Liu, M. Xu, H. Guo, X. Cheng, Y. Xiao, D. Yu, B. Gong, A. Yerukhimovich, S. Wang, and W. Lv, "Tokoin: a coin-based accountable access control scheme for internet of things," *arXiv preprint arXiv:2011.04919*, 2020.

[55] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 337350. [Online]. Available: https://doi.org/10.1145/3460120.3484543

[56] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin, "Android SmartTVs vulnerability discovery via Log-Guided fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX

Association, Aug. 2021, pp. 2759–2776. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/aafer

[57] D. Zha, J. Xie, W. Ma, S. Zhang, X. Lian, X. Hu, and J. Liu, "Douzero: Mastering doudizhu with self-play deep reinforcement learning," *ICML*, pp. 12 333–12 344, 2021.

[58] D. Garg, S. Chakraborty, C. Cundy, J. Song, and S. Ermon, "Iq-learn: Inverse soft-q learning for imitation," *ArXiv*, vol. abs/2106.12142, 2021.

[59] M. Laskin, A. Srinivas, and P. Abbeel, "Curl: Contrastive unsupervised representations for reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5639–5650.

[60] H. Gao, Y. Yang, X. Zhang, C. Li, Q. Yang, and Y. Wang, "Dimension reduction for hyperspectral remote sensor data based on multi-objective particle swarm optimization algorithm and game theory," *Sensors*, vol. 19, no. 6, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/6/1327

[61] I. Schizas, G. Giannakis, and Z.-Q. Luo, "Optimal dimensionality reduction for multi-sensor fusion in the presence of fading and noise," in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 4, 2006, pp. IV–IV.

[62] A. Schclar, "Multi-sensor fusion via reduction of dimensionality," *arXiv preprint arXiv:1211.2863*, 2012.

[63] L. Erhan, M. Ndubuaku, M. Di Mauro, W. Song, M. Chen, G. Fortino, O. Bagdasar, and A. Liotta, "Smart anomaly detection in sensor systems: A multi-perspective review," *Information Fusion*, vol. 67, pp. 64–79, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1566253520303717

[64] Y. Djenouri, A. Belhadi, G. Srivastava, U. Ghosh, P. Chatterjee, and J. C.-W. Lin, "Fast and accurate deep learning framework for secure fault diagnosis in the industrial internet of things," *IEEE Internet of Things Journal*, pp. 1–1, 2021.