Cyber-Physical Deception Through Coordinated IoT Honeypots

Chongqi Guan
The Pennsylvania State University

Guohong Cao
The Pennsylvania State University

Abstract

As Internet of Things (IoT) devices become widely deployed, they face numerous threats due to the inherent vulnerabilities and interconnected nature of these devices. One effective approach to enhancing IoT security is the deployment of honeypot systems, which can attract, engage, and deceive potential attackers, thereby exposing their attack methodologies and strategies. However, traditional honeypots often fail to effectively deceive attackers due to their inability to emulate the physical and network dependencies present in real-world IoT environments. Consequently, attackers can easily detect inconsistencies among the honeypots after launching attacks from multiple sources, spanning both cyber and physical domains, to verify device status. To address this challenge, we propose a Cyber-Physical Deception System (CPDS) capable of mimicking the intricate cyber-physical connections among IoT devices by coordinating various IoT honeypots. Specifically, we model the vulnerabilities of individual IoT devices by collecting and analyzing attack traces. We analyze the physical and network dependencies among IoT devices and formulate them as Prolog rules. Then, we coordinate the honeypots based on the attacker's actions and the dependency rules, ensuring cross-layer consistency among the honeypots. We implemented our deception system by leveraging software-defined networking, enhancing existing IoT honeypots, and configuring them to work in concert. Through online deployment, human evaluation on real attack scenario and extensive simulation experiments, we have demonstrated the effectiveness of CPDS in terms of fidelity and scalability.

1 Introduction

The past two decades have witnessed the widespread deployment of IoT systems. Popular commodity IoT platforms, such as Samsung SmartThings [1], Apple Homekit [2], and Huawei AI Life [3], connect and manage enormous IoT devices with diverse physical functionalities. However, these devices are subject to numerous threats due to their inherent vulnerabilities [4, 5, 6, 7, 8], stemming from outdated or broken security modules, improper patch management, insufficient access control, and inadequate physical security.

To address these issues, we turn to honeypots which are valuable security tools widely used by practitioners to gain insight into the dynamic threat landscape [9]. These decoy systems are designed to attract, engage, and deceive potential attackers by emulating services and devices in a controlled environment. A successful honeypot efficiently attracts attackers through various vulnerabilities, evades detection by reconnaissance and honeypot identification tools, provides convincing responses, deceives attackers into believing they are interacting with real devices, and collects any attack traces left by the attackers for future analysis. Thus, honeypots have the potential to enhance the overall security of IoT systems.

Consider a smart home scenario: a thief aiming to steal valuable assets gathers information about the target house. Attackers might exploit software vulnerabilities in IoT cameras [10] to access video streams and control the camera (e.g., pan, tilt, zoom) to surveil the property. They may also compromise other IoT devices, such as smart speakers [11], smart thermostats, smart bulbs, or smart plugs, to check their device statuses and sensor readings. In this context, IoT honeypots can be deployed to deceive the attacker and mislead their decision-making process. For example, a parked car visible in the live video stream from a camera honeypot, an active air conditioner shown in a smart thermostat honeypot, and a switched-on smart bulb honeypot together might indicate that the residents are at home, thereby deterring the thief from further intrusion. In a military scenario, compromised soldier mounted cameras and other sensors can help attackers obtain tactical and battlefield information. Then, honeypots (e.g., cameras and fake videos) can be employed to deceive attackers with false battlefield information, such as fake troop movement, leading them to make incorrect decisions.

However, deploying IoT honeypots for deception presents a unique challenge. Unlike traditional cyberattacks that primarily target individual servers, the threat landscape of the IoT ecosystem extends to interconnected IoT devices [12, 13, 14]. Attackers can leverage information from multiple sources,

spanning both the cyber and physical domains, to verify device statuses, craft attack strategies, and ultimately achieve their objectives. Any inconsistency among IoT honeypots, such as a smart bulb indicating 'on' while a video feed shows a dark room, could alert the attacker to the presence of honeypots and potential deception. The intricate connections among IoT devices underscore the urgent need for a comprehensive defense strategy that extends beyond isolated honeypots designed for conventional servers or devices.

To address this challenge, we propose a Cyber-Physical Deception System (CPDS) which can coordinate IoT honeypots, thereby mimicking the multifaceted nature of the IoT ecosystem and creating a cohesive deception environment. Specifically, we first model the vulnerabilities of individual IoT devices by collecting and analyzing attack traces. Next, we generate dependency Prolog rules based on the network topology and device locations of the target IoT system we aim to emulate. We then propose a coordinated algorithm to ensure consistency among different honeypots at runtime.

The main contributions of this paper are as follows:

- We model the vulnerability of individual device by analyzing attack traces and formulate physical and network dependencies among IoT devices as Prolog rules.
- We propose a honeypot coordination algorithm based on the exploited vulnerabilities and dependency prolog rules to ensure consistency among honeypots.
- We enhance existing IoT camera honeypots to improve their adaptability to environmental changes, and implement CPDS using various IoT honeypots.
- Through a combinations of online deployments, real experiments, and simulations, we validate the performance and deception capability of CPDS.

2 Background and Related Work

2.1 Threat Model

In this paper, we focus on individual attackers aiming to infiltrate IoT systems such as smart home. These IoT systems consist of inter-connected IoT devices (e.g., cameras, smart plug, smart bulb, etc.) that leverage short-range, low-power protocols (e.g., WiFi) to communicate with each other. The IoT devices may have software vulnerabilities, which are rooted in the firmware [4, 15], as well as hardware vulnerabilities that may be exploited via physical channels [7, 11, 16, 17].

We assume that attackers can locate IoT devices using open-source or custom reconnaissance tools such as Nmap [18] and Masscan [19]. Additionally, they may sniff network traffic to extract IoT device characteristics, such as brand, model, or firmware version. Once identified, attackers might exploit device vulnerabilities to compromise these IoT devices, potentially stealing sensitive data (e.g., video streams, device logs), disabling the devices, or installing malwares for subsequent attacks.

Given the general absence of firewalls or MAC address filtering in most smart home or smart factory networks, we assume that if attackers gain access to the IoT network, they can transmit spoofed, handcrafted packets to any other device on the same network to exploit its vulnerability. Additionally, many off-the-shelf IoT products expose unprotected network services to the home network, potentially enabling attackers to utilize these services after compromising victim devices. To achieve their ultimate goals, attackers may repeat the aforementioned attacks across multiple devices, enabling them to devise more sophisticated attacks.

2.2 Dependency

We identify and define two types of dependencies in a IoT system: **physical dependency**, and **network dependency**.

Two devices are physically dependent if one can impact the other through a physical medium. There are two types of physical dependencies: *direct* and *indirect*. Direct physical dependency occurs when one device is physically attached to another, and changing the status of one device directly alters the status of the other. The most common example of direct physical dependency is electrical dependency, such as the relationship between a smart plug and a smart bulb. The smart bulb is directly physically dependent on the smart plug, as the plug can turn off the bulb by cutting its power supply.

In contrast, indirect physical dependency arises when one IoT device can change the surrounding environment, thereby impacting other IoT devices. For instance, the video stream of a camera may be affected if a smart bulb is turned on or off. Similarly, a smart thermostat's reading may gradually increase if a smart heater is turned on. Indirect physical dependencies typically requires two devices to be physically located in close proximity. For example, the camera and the smart bulb in a smart home need to be in the same room (e.g. both at the entrance) to demonstrate an indirect physical dependency.

Network dependency, on the other hand, occurs when a IoT devices impact other devices by sending packets through the network. These physical and network dependencies can be exploited by attackers to compromise target devices, verify device status, or launch attacks.

2.3 Cross-layer Consistency

IoT devices within an IoT system often interact and influence each other based on their physical and network dependencies, as well as their functionalities. Attackers can exploit these causal relationships among IoT devices for validation, and any inconsistencies among honeypots can significantly reduce the effectiveness of the deception. This presents a unique challenge in deceiving attackers within the IoT ecosystem.

For example, Figure 1 illustrates a typical deception scenario in a smart home environment. In the physical layer, we emulate various IoT devices located in different rooms. Some

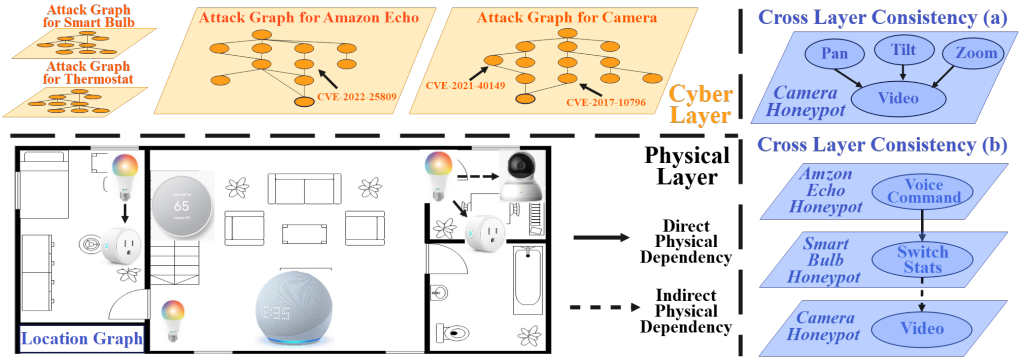


Figure 1: Cyber physical deception in a smart home scenario

devices are directly connected, as indicated by the solid arrows (e.g., in the bedroom, the smart bulb is plugged into the smart plug). The indirect physical dependencies among devices are shown by the dotted arrows (e.g., at the entrance, the smart bulb can affect the video content of the camera). In the cyber layer, we utilize IoT honeypots to mimic the software vulnerabilities and functionalities of various IoT devices, as depicted in the attack graphs.

To effectively deceive attackers, it is crucial to maintain consistency across all honeypots in both the cyber and physical layers. Specifically, the "cross-layer consistency (a)" in Figure 1 represents a set of consistencies within a camera honeypot. Attackers may exploit specific vulnerabilities to manipulate the camera (e.g., pan, tilt, zoom) and verify their attack effects by checking the video stream. To maintain cross-layer consistency, a camera honeypot must not only emulate vulnerabilities in the cyber layer and provide authentic video streams in the physical layer but also support interactive camera controls that realistically mimic a real camera.

On the other hand, "cross-layer consistency (b)" represents a set of consistencies across three honeypots emulating a smart speaker, a smart bulb, and an IoT camera. Certain functionalities of an Amazon Echo honeypot and a smart bulb honeypot may be interconnected if they are paired. For instance, When an attacker interacts with the Amazon Echo and sends a "turn on the light" voice command, the switch status of the smart bulb should change to "on". Additionally, if the smart bulb and the camera are placed in the same room, altering the switch status of the smart bulb should logically impact the video content generated by the camera honeypot. If the smart bulb is turned on, the camera should display brighter video footage, as if it were illuminated by the bulb.

To formalize this concept, we define "cross-layer consistency" among a set of honeypots $H = \{h_1, h_2, ..., h_n\}$ as follows: $\forall (h_i, h_j) \in H, \forall t, \mathcal{R}(S^t_{h_i}, S^t_{h_j}) = True$, where t denotes timestamp, S represents the state of a honeypot in either the cyber or physical layer (e.g., switch on or off for a smart bulb) and $\mathcal{R}$ is the set of Prolog rules that represent the dependencies among different emulated devices. In other words, maintaining "cross-layer consistency" among honeypots means that, at any given time, the states of any two honeypots should not conflict. More details will be introduced in Section 3.3.

2.4 Related Work

There has been considerable research [20, 21, 22, 23, 24, 25, 26, 27] in recent years on using honeypots to emulate various types of IoT devices. Below, we provide details on these honeypots, focusing on whether they can emulate the corresponding IoT devices' physical functionality, and whether they have the potential to emulate the network and physical dependencies.

IoTPOT [20] is the first honeypot specifically designed for IoT devices. It is a low-interaction honeypot focusing on emulating telnet services commonly used by IoT devices. Due to its fixed response logic, IoTPOT can not emulate physical functionalities or dependencies among IoT devices.

HoneyCloud [21] and Honeware[22] both leverage the firmware image of IoT devices, QEMU and some hardware like RaspberryPis to construct IoT honeypots. Through careful customization, these hardware honeypots can emulate physical functionality of IoT devices. However, deploying and maintaining hardware based IoT honeypots are costly and not scalable.

VPNhoneypot [28], IoTCMal [29] and Siphon [30] all construct a hybird honeypot structure leveraging real IoT devices at backend to handle attacker request. These type of honeypots can emulate physical functionality due to the real device at backend but suffers from scalability and cost issues.

Hakim *et al.* [31] introduced U-POT, an IoT honeypot framework specifically designed for the UPnP (Universal Plug and Play) protocol which is widely used in smart home devices. It uses device description files to automate honeypots and provide fake responses. U-POT is able to emulate the physical functionality of simple IoT devices such as smart plug or smart bulb by crafting fake responses indicating their device status has changed (i.e., switch on).

HoneyIoT [32] is an adaptive IoT honeypot that employs reinforcement learning to analyze the interaction history between attackers and IoT devices. This allows it to select the most appropriate response at run time. Additionally, Hon-

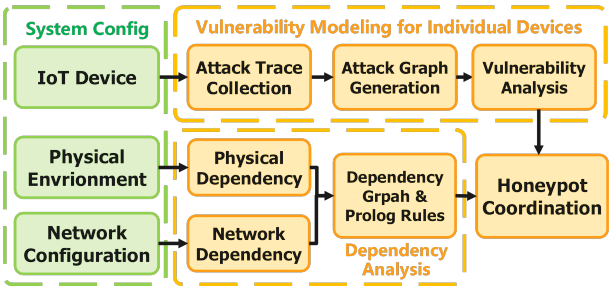


Figure 2: Our cyber-physical deception system (CPDS)

eyIoT utilizes a differential analysis-based method to mutate responses, thus enhancing its authenticity.

HoneyCam [33] and SweetCam [34] are camera honeypots capable of providing interactive live video streams by leveraging pre-recorded 360° video. They design methods to map the 360° video to different fields of view according to the attacker's camera control commands, allowing them to fully emulate the physical functionality of IoT cameras. However, these camera honeypots can not emulate the physical dependencies between IoT cameras and other devices, such as smart bulbs, as they are unable to respond to environment changes. To address this issue, we further enhance Honeycam in Section 4.2.

Existing research on honeypots has predominantly focused on emulating individual IoT devices, often overlooking the physical and network dependencies within IoT ecosystems. This oversight significantly impacts the fidelity and effectiveness of IoT honeypots in real-world deployments. In this paper, we address this gap by coordinating diverse IoT honeypots to ensure cross-layer consistency, thereby enhancing the fidelity of the overall deception system.

3 System Design

3.1 System Overview

Figure 2 gives an overview of the proposed CPDS which consists of four main modules: *system configuration, vulnerability modeling for individual devices, dependency analysis*, and *honeypot coordination.*

The system configuration serves as the input to CPDS and is typically derived from the real IoT system intended for emulation. For example, to emulate a smart home as illustrated in Figure 1, the IoT devices may include surveillance cameras, smart plugs, smart speakers, and smart thermostats, etc. The physical environment component specifies the layout of these IoT devices within the real system, such as the specific rooms where they are located. The network configuration component defines the communication links between these devices, indicating whether they can interact with one another over the network.

The Vulnerability Modeling for Individual Devices module aims to conduct systematic and realistic analysis on the emulated devices, and will be explained in detail in Section 3.2.

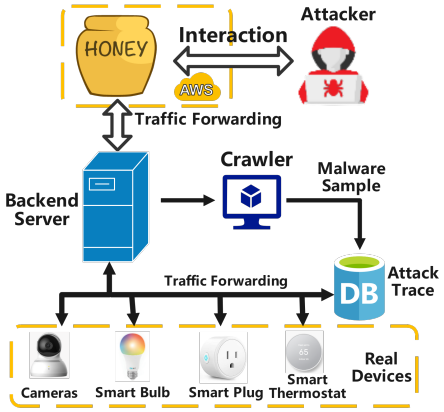


Figure 3: The testbed used for attack trace collection

The Dependency Analysis module takes the system configuration as inputs to model the interactions among IoT devices. It first generates a dependency graph based on the network topology and device location, and then derives Prolog rules representing various dependencies. The dependency analysis module will be presented in Section 3.3. Subsequently, leveraging the results of vulnerability modeling and dependency Prolog rules, we design a coordination algorithm to ensure cross-layer consistency at runtime. The specifics of honeypot coordination are detailed in Section 3.4.

3.2 Vulnerability Modeling for Individual Devices

Modeling IoT devices for security purposes is a complex and widely studied challenge. Prior research has explored methods for intrusion detection, anomaly detection, and attack simulation, with significant efforts focused on constructing behavioral models that emulate device functionality and vulnerabilities [35, 36, 37]. However, the requirements of CPDS differ significantly from these traditional IoT behavior modeling tasks.

CPDS requires honeypots to precisely replicate not only the functionality of IoT devices but also their vulnerabilities. This is particularly challenging given the heterogeneity of IoT devices, where vulnerabilities vary significantly based on factors such as brand, model, and firmware version. Accurate modeling of these vulnerabilities is critical to predict the effects of potential exploits and to deceive attackers effectively. Moreover, attackers often perform reconnaissance attacks [18] to gather detailed information, such as firmware versions or serial numbers, about target devices before initiating actual attacks. Any inconsistency detected during reconnaissance can reveal the honeypot, causing attackers to abandon further actions. To address this, CPDS requires comprehensive behavioral modeling of target IoT devices, ensuring fidelity throughout the entire attack period, from reconnaissance to subsequent interactions. These unique requirements set our approach apart from conventional IoT modeling techniques.

Device Model	Manufacture	Device Type	Vulnerability ID
NC220	TP-Link	Camera	CVE-2020-12109, etc.
RLC-410W	Reolink	Camera	CVE-2021-44402, etc.
E1 Zoom	Reolink	Camera	CVE-2021-40149
Home	YI	Camera	CVE-2018-3928, etc.
DS-2CD2183G	Hikvision	Camera	CVE-2021-36260
Insight	Wemo	Smart Plug	CVE-2018-6692
Mini	Wemo	Smart Plug	CVE-2018-6692
HS103-P4	TPlink	Smart Plug	CVE-2019-15745
VMB3000	Netgear	Router	CVE-2019-3949, etc.
DGN2220	Netgear	Router	CVE-2020-35577, etc.
TL-WR840N	TP-Link	Router	CVE-2018-14336, etc.
DIR-3040	D-Link	Router	CVE-2021-21819, etc.
WS5200	Huawei	Router	CVE-2019-5268, etc.
WS7200	Huawei	Router	N/A
Hue Wifi	Philips	Smart Bulb	CVE-2019-18980, etc.
Ring	Amazon	Doorbell	CVE-2019-9483
Sonos Speakers	Sonos	Smart Speakers	CVE-2018-11316
Echo Dot	Amazon	Smart Speakers	CVE-2022-25809

Table 1: IoT devices used for trace collection

We will elaborate on our methodology in the following subsections.

3.2.1 Attack Trace Collection

In order to learn how the attackers interact with IoT devices and model their vulnerabilities, we first build a system to collect attack traces. As shown in Figure 3, the system consists of a frontend virtual machine running in the public internet, a backend server for traffic forwarding and preliminary traffic analysis, and a few IoT devices including various models of IoT cameras, routers and smart plugs (The detailed list of IoT devices is shown in Table 1).

The system interacts with the attacker by forwarding the received packet to one of the IoT devices. Based on the attacker's request, the corresponding IoT device sends the necessary files or responses so that the attacker can continue to interact with the corresponding IoT device. This process continues until the attacker exploits certain vulnerability of the target device or stops interacting with the IoT device. We also filter out any commands containing malware download instructions such as Wget or Curl, and forward them to a crawler in a sandbox to automatically collect malware from the attacker's control and command server. Our system maintains the log traces and may have to be rebooted in some cases to recover from the attacks. Then, a new cycle starts which may select a different IoT device for a different attacker. By doing this, our system can obtain different attacker traces, targeting different kinds of IoT devices. These attack traces will then be further analyzed and used for vulnerability modeling.

3.2.2 Attack Graph Generation

We have implemented our attack trace collection system and deployed on the public Internet for a duration of two months starting from Sep. 2023, where the backend infrastructure comprises a server and various IoT devices, as listed in Table 1. Each IoT device within the backend ecosystem has been profiled, and a dedicated database has been established to store traces from each individual attack session. These traces contain details such as attacker IP addresses, timestamps of attacks, the protocols used, targeted services, and exploited vulnerabilities.

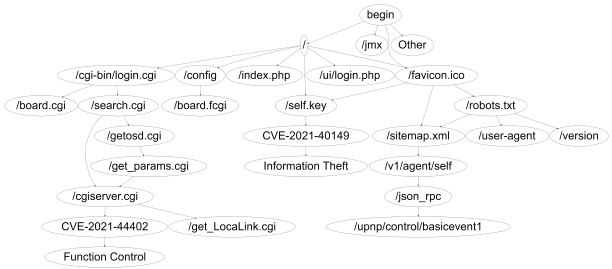


Figure 4: A partial attack graph generated based on the collected attack trace against Reolink cameras

An initial analysis of the trace data reveals the presence of remotely exploitable CVEs associated with these IoT devices. To gain deeper insights into attacker behaviors against IoT devices and model their vulnerabilities more comprehensively, we have generated attack graphs for IoT devices based on the interactions between attackers and the IoT devices.

Figure 4 shows an attack graph against Reolink camera over the HTTP port. As the whole graph is too big, we only show a partial attack graph emphasizing specific vulnerabilities exploited and the attack behavior. In the attack graph, a node represents the attacker's action such as probing a directory, accessing a resource or exploiting a vulnerability. For example, the node with '/' indicates that the attacker probes the root directory. The node with '/favicon.ico' indicates that the attacker tries to access the favicon file which is usually a small icon indicating the device type or manufacturer. The node with 'CVE-2021-40149' means that the attacker exploits a specific vulnerability with Common Vulnerabilities and Exposures (CVE) ID 2021-40149, i.e., a publicly disclosed IoT security flaw on Reolink cameras where no security check is enforced when the attacker acquires SSL private key through the HTTP server. The edges connecting two nodes indicate that some attackers have taken another action after receiving the previous response from the IoT device.

From the attack graph, we can see that the attacker conducts various pre-attack checks to gather information from the remote host before launching attacks. For example, as shown in Figure 4, some attackers first access the favicon file to identify that this is a Reolink camera, by matching the MD5 hash of favicon or by analyzing its image. Then, they decide to exploit the 'CVE-2021-40149' vulnerability by sending various requests. On the other hand, if the attacker notices that the remote host is not a Reolink camera (the MD5 hash does not match), he may not proceed with follow up attacks.

To better model the potential impacts after an attacker exploits vulnerabilities, we categorize the attack impacts into six categories as shown in Table 2. We also attach these at-

Attack Impacts	Explanation
Root	Attackers have root privilege of target device, enabling them to execute all following attacks. They can also leverage victim device to send spoofed commands to other IoT device on the same network
Function Control	Attackers can manipulate specific functions of the target device (i.e., camera movement, light switches, etc.). However, the device itself is not fully compromised.
Event Access	Attackers can intercept events or status information from IoT devices (i.e., device power on or off, etc.).
Information Theft	Attackers can obtain credential information (i.e., Wifi credentials, keys, etc.) from victim device.
Denial of Service (Dos)	Attackers can disrupt the normal operation of the target IoT device, rendering it temporarily or permanently paralyzed.

Table 2: Type of Attack Impacts on IoT devices

tack impacts to the attack graph right after the vulnerability nodes. For instance, since 'CVE-2021-40149' discloses the SSL private key to the attacker, its attack impact is categorized as Information Theft. On the other hand, since the attacker can turn off the camera by exploiting 'CVE-2021-44402', we categorize its attack impact as Function Control. These attack actions may propagate through both network dependency and physical dependency, enabling attackers to perform cross-layer validation. The modelled vulnerability for each device is then translated to Prolog facts for dependency analysis and honeypot coordination. For example, the fact presented in Listing 1 means that vulnerability 'CVE-2021-40149' exists on Reolink E1 Zoom cameras.

```
vulExists(reolink_E1_Zoom, 'CVE-2021-40149').
vulProperty('CVE-2021-40149', informationTheft
    ).
```

Listing 1: Prolog fact for CVEs found on specific devices

3.3 Dependency Analysis

The dependency analysis module models the interactions among IoT devices. It generates dependency graphs based on the system configuration and then derive Prolog rules for each dependency to further guide coordination among honeypots. Algorithm 1 outlines the pseudocode for the dependency graph generation process, and Figure 5 shows an example dependency graph generated from the configuration shown in Figure 1. Specifically, we represent direct physical dependencies, indirect physical dependencies, and network dependencies using Prolog rules.

Direct physical dependency occurs when one device is physically connected to another, and changing the status of one device directly affects the status of the other. Therefore,

Algorithm 1: Dependency Graph Generation

Input: (1) Device Set $\mathcal{D}$
(2) Device Physical Connection Set $\mathcal{C}$
(3) Network Topology $\mathcal{T}$
(4) Device Location Set $\mathcal{L}$
Output: Dependency Graph $\mathcal{G}$ for all IoT devices

Algorithm dependency_graph($\mathcal{D}$, $\mathcal{L}$, $\mathcal{C}$, $\mathcal{T}$)
  $\mathcal{G}$ = DiGraph ($\mathcal{D}$) // Initialize G as a directed graph
  foreach *physical connection* $c(d_a, d_b)$ *in* $\mathcal{C}$ **do**
    $\mathcal{G}$.add_edge (d_a, d_b, label = "Direct Physical")
  foreach *device* d_a, d_b *in* $\mathcal{D}$ **do**
    if $\mathcal{L}(d_a) = \mathcal{L}(d_b)$ **then**
      if d_a *can affect* d_b **then**
        G.add_edge (d_a, d_b, label = "Indirect Physical")
      if d_b *can affect* d_a **then**
        G.add_edge (d_b, d_a, label = "Indirect Physical")
  $\mathcal{C}$ = Union Find ($\mathcal{T}$) // Find connected components. Each is a subgraph where any two nodes are connected by a path.
  foreach c *in* $\mathcal{C}$ **do**
    foreach *device* d_a, d_b *in* c **do**
      G.add_edge (d_a, d_b, label = "Network")
  return G

direct physical dependency is predetermined by the defender based on the floor plan in the system configuration. For example, consider the bedroom in Figure 1. The smart bulb is directly connected to the smart plug, and turning off the smart plug will cut off the power supply to the smart bulb. Thus, there exists a direct physical dependency between the bedroom smart bulb and the bedroom smart plug. Since direct physical dependencies remain static in the IoT system regardless of network topology and environment, we can hard code Prolog rules to represent them, as shown in Listing 2.

```
off(Device) :-
    pluginto(Device,Smartplug),
    smartplug(Smartplug),
    off(Smartplug).
```

Listing 2: Direct Physical Dependency

Indirect physical dependency occurs when an IoT device changes the surrounding environment (e.g. light) , thereby affecting other IoT devices. For instance, the video stream of a surveillance camera may be affected if a smart bulb located in the same room is turned on or off. The Prolog rules for indirect physical dependencies are generated based on the device type, device location, and environmental factors. Listing 3 provides an example of Prolog rules for indirect physical dependencies.

```
on(Light) :-
```

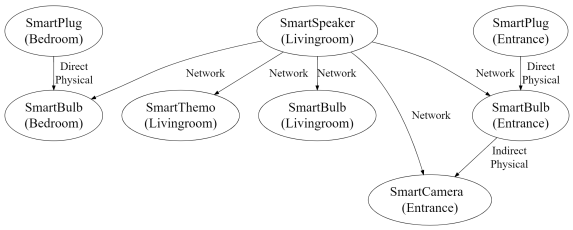


Figure 5: Dependency Graph Example

```
    on(Smartbulb),
    smartbulb(Smartbulb).

switchVideo (Camera, Light, Room) :-
    on(Light),
    camera(Camera),
    inRoom(Light, Room),
    inRoom(Camera, Room).
```

Listing 3: Indirect Physical Dependency

Network dependency occurs when IoT devices in the system interact with each other by sending packets through the network. For example, if a smart speaker is paired with a smart bulb, it can remotely turn the light on or off by sending corresponding request packets to the smart bulb. The network dependency is directly generated based on the network topology in the network configuration file. Typically, most IoT systems lack firewalls or MAC address filtering. Therefore, if the network topology indicates that two devices are connected, we assume they can exchange packets with each other. Listing 4 provides an example of Prolog rules for network dependencies.

```
connected(Device1, Device2) :-
    network(Device1, Network),
    network(Device2, Network).
```

Listing 4: Network Dependency

3.4 Honeypot Coordination

Coordination among multiple honeypots is essential to accurately emulate the interactions between devices in an IoT ecosystem. This coordination ensures that the honeypots can faithfully mimic the interconnected functionalities of real-world IoT devices, thereby maintaining cross-layer consistency across CPDS.

To develop appropriate honeypot coordination policies for a given IoT system, we begin by defining individual honeypots that emulate different IoT devices in the system. For instance, a defender may establish a Prolog fact such as *camera (Honeypot1)* to represent a camera honeypot in CPDS, and then utilize the *VulExists* predicate from Listing 1 to denote the vulnerabilities it emulates. Defenders can leverage the vulnerability analysis results (Figure 4) to incorporate additional vulnerabilities into the camera honeypots, aligning

Algorithm 2: Honeypot Coordination Algorithm

Input: (1) Prolog Rules Set $\mathcal{R}$
(2) Honeypot Set $\mathcal{H}$
(3) Attacker Action A
Output: Ensure Consistency in $\mathcal{H}$

```
1  Algorithm coordination(R, H, A)
2      finished = {}
3      ActionStack = [A]
4      while ActionStack ≠ ∅ do
5          CurAction = ActionStack.pop()
6          if CurAction not in finished then
7              H.handle(CurAction)
8              foreach NextAction in R(A) do
9                  if NextAction not in finished then
10                     ActionStack.push(NextAction)
```

with their deception goals. To streamline the coordination process, we abstract the core functionalities of each honeypot into APIs. For example, camera honeypots may have APIs for switching video, panning, tilting, and zooming, while smart plug honeypots might have APIs for turning on and off. This abstraction enables seamless interaction between honeypots.

At runtime, when an attack action triggers specific device functionalities emulated by honeypots, our system executes the coordination algorithm presented in Algorithm 2. This algorithm employs a stack-based approach to handle attack actions and their effects across our system. It takes three inputs: the set of Prolog rules extracted from device dependencies ($\mathcal{R}$), the set of honeypots in the system ($\mathcal{H}$), and the attacker action (A). The algorithm processes the attack action by invoking relevant APIs of the affected honeypot and then explores subsequent actions based on the Prolog rules. For instance, if an attacker has turned on a smart bulb, the coordination algorithm will first inform the smart bulb honeypot to change its status through corresponding APIs. It will then check the Prolog rules (i.e., Listing 3) and inform the camera honeypot to switch its videos to align with the smart bulb honeypot's new state. This coordination process ensures that all direct and indirect effects of an attack action are properly simulated across the entire IoT system, providing a more realistic and interconnected emulation of an IoT environment. We will describe the implementation details in the next section.

4 Implementation

Our cyber physical deception system is mainly implemented with Python. We also use Prolog to generate honeypot coordination policies, MySQL for database operations, Mininet for SDN simulation, and Graphviz to visualize attack graphs and dependency graphs.

As shown in Figure 7, the implementation has three components: deception system configuration and Prolog rule generation (green box), honeypot selection for device emulation

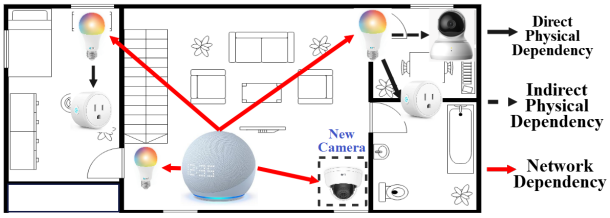


Figure 6: Deception system deployment based on a configuration template

(yellow box), and deception network setup (blue box). We will give their details in the following subsections.

4.1 Deception System Configuration

The goal of CPDS is to provide a comprehensive emulation of an IoT system, which requires a system configuration input about the target IoT system to be emulated. This configuration should contain information about the IoT devices in the target system, their physical layout (e.g., the rooms where they are located, which will be used to compute indirect physical dependencies), and the direct physical and network dependencies among the devices. All this information is encapsulated in a JSON-formatted configuration file.

For a fine-grained emulation, users must conduct a detailed analysis of the target IoT system to extract this information. For example, consider the smart home environment shown in Figure 1, which consists of eight interconnected IoT devices, including smart bulbs, cameras, smart plugs, and smart speakers, distributed across three rooms (bedroom, living room, and entrance). Users then model each IoT device using the vulnerability analysis module (Section 3.2) to identify exploitable functionalities and analyze their interdependencies (Section 3.3) to generate appropriate Prolog rules. Although this process involves considerable manual effort, it ensures the highest emulation quality by accurately modeling each device, emulating the physical layout of the IoT system, and capturing all physical and network dependencies between devices.

In scenarios where efficiency and scalability are prioritized, users can utilize pre-defined configuration templates to quickly deploy a CPDS without starting from scratch. These templates include pre-modeled off-the-shelf devices, predefined device layouts, and established physical and network dependencies. Users can then modify the configuration file by editing devices and dependencies as needed to align with the specific IoT system they aim to emulate.

Figure 6 illustrates how a pre-defined configuration template for the smart home in Figure 1 can be leveraged. For example, when a camera is added in the living room to monitor the backyard, the configuration template is updated by including the corresponding device, as shown in Listing 5. If the new camera is paired with the smart speaker in the living room, users can add network dependencies between them, and the system will automatically generate the corresponding

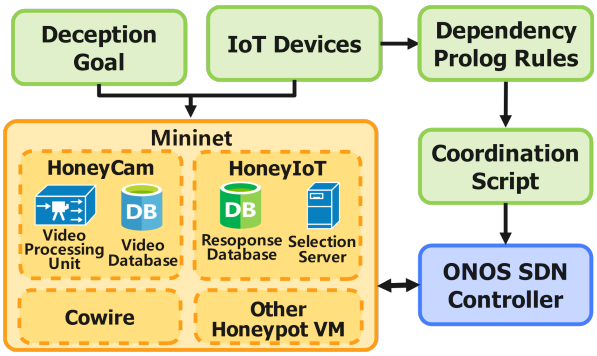


Figure 7: Major components in system implementation

Prolog rules. Similarly, users can remove specific devices or dependencies from the CPDS by editing the configuration file. This approach enables a more agile deployment of CPDS, significantly reducing the amount of manual efforts required for device modeling and Prolog rule computation, while maintaining flexibility to customize the setup to specific requirements.

```
"device": [
 ...
  {"id": 8, "name": "LivingroomSmartCamera", "location
 ": "Livingroom", "type": "SmartCamera"}
]
...
```

Listing 5: IoT Device Configuration

4.2 Honeypot Selection for Device Emulation

Based on the deception system configuration, we identify the devices to be emulated with honeypots. The selection, customization, and enhancement of IoT honeypots to emulate these devices is a crucial step, as it directly impacts the quality of deception and the associated system deployment costs. In our implementation, the honeypot selection process considers several key factors:

1) Emulated Device Types: Unlike traditional IoT honeypots such as IoTPot [20], which only emulate the Telnet protocol, most state-of-the-art IoT honeypots specialize in emulating specific device types. For example, HoneyCam [33] and Siphon [30] can emulate IoT cameras by providing live video streams. Therefore, choosing honeypots based on their ability to accurately emulate the desired IoT device types is essential.

2) Fidelity: Honeypots must accurately mimic real IoT devices. High-fidelity honeypots significantly enhance the overall deception quality by making it more difficult for attackers to differentiate them from genuine devices. In CPDS, we prioritize honeypots capable of accurately emulating the physical functionality of devices. Moreover, to ensure cross-layer consistency, we require honeypots to emulate certain functionalities even if attackers may not directly perceive them. For example, our camera honeypots not only need to provide

interactive video streams to the attacker, the video content must also comply with the emulated devices' surrounding physical environment, including factors such as time of day and weather conditions.

3) Scalability: To effectively emulate large-scale IoT systems, our deception system must strike a balance between fidelity and resource requirements. We prioritize lightweight, virtualized honeypot solutions over resource-intensive approaches, as long as they meet our deception goals. Thus, we do not use real device-based honeypots, such as Siphon [30] and VPNHoneypot [28], due to their high hardware requirements and significant deployment costs. Instead, we opt for virtualized honeypot solutions like HoneyCam [33] and Cowrie [38]. These can be easily deployed and scaled without relying on physical IoT devices, offering a more flexible and cost-effective approach to large-scale IoT system emulation while maintaining sufficient fidelity for our deception goals.

Based on the selection criteria outlined above, our implementation uses the following honeypots to emulate each type of devices shown in Figure 1, enhancing or customizing them for better deception capability.

Camera Honeypot: This honeypot is primarily achieved through an enhanced version of HoneyCam [33]. HoneyCam is a scalable high-interaction IoT camera honeypot based on pre-recorded 360°video. It provides interactive video streams by mapping the video to different fields of view based on the attacker's camera control commands. However, in CPDS, the camera honeypot must not only handle camera control commands from attackers but also respond to environmental changes. For example, outdoor surveillance cameras need to provide video content that reflects current weather conditions and time of day (details in Figure 11). Similarly, indoor cameras must adjust their video content based on the status of other IoT devices in the environment (details in Figure 12).

To satisfy these requirements, as shown in Figure 8, we enhance HoneyCam through the following improvements: 1) We prepare multiple pre-recorded 360°videos captured under various environmental conditions (e.g., different times of day, smart bulb on/off states, etc.). 2) We implement an environment module that dynamically switches between videos based on status changes of dependent IoT devices and the surrounding environment. The environment module retrieves real-time weather information from the internet based on the emulated camera's location as defined by the defender. 3) We utilize a stream overlay technique [39] to emulate weather conditions that are challenging to pre-record (e.g., fog, rain). We create overlay videos with corresponding weather effects and transparent backgrounds, and then leverage FFmpeg to incorporate these overlays on top of the generated interactive video stream in real-time while pushing to the frontend.

These enhancements significantly improve the fidelity and adaptability of our camera honeypots, making them more responsive to both attacker interactions and environmental changes.

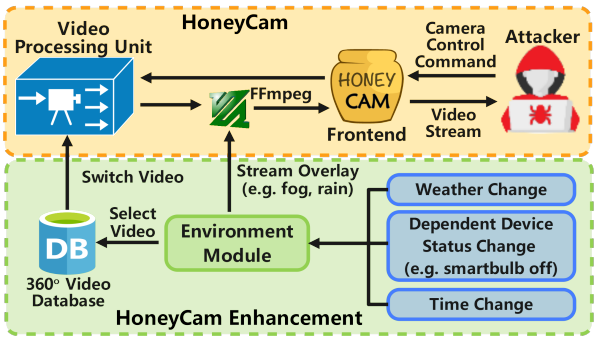


Figure 8: HoneyCam Enhancement

Smart Plug Honeypot: This honeypot is primarily implemented through a combination of HoneyIoT [32] and U-PoT [31]. HoneyIoT is an adaptive IoT honeypot that employs reinforcement learning to analyze the interaction history between attackers and IoT devices. This allows it to select the most appropriate response from a database at runtime. Additionally, HoneyIoT utilizes a differential analysis-based method to mutate responses, enhancing their authenticity. In our implementation, we leverage the attack traces collected in Section 3.2 to train the reinforcement learning agent and construct the response database. U-PoT [31] specializes in emulating the UPnP (Universal Plug and Play) protocol. Given that most smart plugs implement UPnP to provide remote control functions, we utilize U-PoT to accurately emulate these UPnP services. This is particularly important for maintaining the illusion of genuine smart plugs that can be controlled even when users are away from home.

Router Honeypots: This honeypot is primarily achieved through a combination of HoneyIoT and Cowrie [38]. Cowrie is a widely adopted, open-source, medium-interaction SSH honeypot. Its lightweight design makes it ideal for emulating IoT devices that support Telnet or SSH services. By configuring Cowrie, we can efficiently simulate the SSH services provided by the router. Concurrently, we leverage HoneyIoT to emulate other ports opened by the router.

Other Device Honeypots: Similar to the aforementioned honeypots, other devices in smart home environments (e.g., smart bulbs, smart thermostats, etc) are emulated using a combination of existing honeypots, depending on the services they provide and the ports they open. For instance, for smart bulbs that support UPnP services, we leverage U-PoT to construct the corresponding smart bulb honeypot.

4.3 Deception Network Setup

Software-Defined Networking (SDN) has gained popularity for addressing traditional network challenges by decoupling the control plane from the data plane and centralizing network control within a logically centralized controller. SDN is well-suited for our system because it enables easy and flexible network management. Specifically, SDN allows network administrators to configure, manage and optimize network

Table 3: Online Deployment Basic Statistics

Emulated Device	Honeypot Used	Total Attack Session	Unique Attacker IP	C&C Server detected	Malware Collected
IoT Camera	Enhanced HoneyCam	48532	4632	779	523
IoT Camera	-> Real-camera [30]	53762	4473	0	0
Smart Plug	HoneyIoT, UPot	17658	1438	237	163
Smart Plug	-> Dionaea [40]	13790	836	26	19
Router	HoneyIoT, Cowire	97894	6270	963	870
Router	-> Kippo [41]	68591	4680	89	69
Smart Bulb	HoneyIoT, UPoT	6850	531	67	52
Smart Bulb	-> Glutton [42]	8274	610	18	14
Other	HoneyIoT, etc	28642	2457	238	148

resources through software applications, streamlining the process of honeypot deployment, management and revocation. Moreover, the centralized control architecture provided by SDN offers an ideal platform for executing our honeypot coordination algorithm. When facing attacks, we can effectively coordinate the behavior and status of deployed honeypots, ensuring a consistent deception scenario.

Given these benefits, we have chosen to build our deception system on top of SDN. Specifically, we use mininet [43] for network emulation as it allows us to easily simulate SDN networks running honeypot VMs and openflow switches. We use Open Network Operating System (ONOS), an open-source SDN controller that provides control plane for managing honeypots. After analyzing the IoT system to be emulated and generating the corresponding configuration file, we use python scripts to automatically create and connect honeypot VMs based on the network topology configuration file.

To coordinate the honeypots and ensure consistency across our deception system, we have devised a streamlined approach. We abstract the core functionalities of each honeypot (e.g., Switch Video, Pan, Tilt, Zoom for camera honeypots, Turn on/off for smart plug honeypots) into APIs. When an attack action triggers specific honeypot functionalities at runtime, CPDS utilizes the ONOS controller to execute the coordination algorithm as shown in Algorithm 2. This algorithm invokes the relevant APIs of the affected honeypots, propagating the effects of the attack action across different IoT honeypots. In this way, we can accurately emulate the cyber and physical impacts of the attacks in the emulated IoT system, ensuring cross-layer consistency.

5 Evaluations

In this section, we evaluate the performance of CPDS from three perspectives. First, we implement and deploy the aforementioned IoT honeypots that will be used in CPDS, as detailed in Section 4.2, on the public internet to evaluate and compare their performance in emulating various devices. Second, to evaluate CPDS's deception capability against human attackers, we construct two real-life attack scenario demos. These scenarios are powered by CPDS, a baseline version of the system with no coordination module, or actual IoT devices. We then recruit participants to interact with these scenarios and identify whether they are engaging with real devices or honeypots. The experiments have been approved by our Institutional Review Board (IRB). Finally, we evaluate the scalability, response time, and resource consumption of CPDS through extensive simulations.

5.1 Online Deployment

The accurate emulation of IoT ecosystem hinges on the performance of individual IoT honeypots emulating different IoT devices within the system. As outlined in Section 4.2, we implement and enhance existing IoT honeypots to emulate various IoT devices. To evaluate the overall performance, we deploy these honeypots on the public internet and exposing them to real-world attack patterns.

5.1.1 Experiment Setup

As discussed in Section 4.2, we have enhanced HoneyCam in order to emulate the IoT cameras in CPDS. Specifically, we use Insta360 One X2 [44] to pre-record 360-degree video at different time of the day. We then preprocess the pre-recorded video using Insta studio to ensure no sensitive information remains. To emulate different weather conditions for outdoor camera, we prepare overlay videos with various weather effects such as fog and rain.

In our deployment, the frontend of our enhanced HoneyCam consists of virtual servers placed on the public internet. Similar to real IoT cameras (as listed in Table 1), we open some ports to provide video streams and control services. Specifically, we open port 554 to allow attackers to access streams through RTSP protocols. We also open port 80 and provide web pages that are exact replicas of the TPlink NC220 camera, which provides live video stream and camera control services. The backend server runs on a desktop with AMD Ryzen 7 5800, 32GB RAM and an NVIDIA RTX 3080 GPU, primarily serving live stream videos to the frontend honeypots.

For comparison purpose, we also deploy three other camera Honeypots: 1) the original HoneyCam without enhancement, 2) a low-interaction recorded-video based camera honeypot

[45] that does not provide camera control functionality, 3) a real-camera based honeypot similar to Siphon [30] using a Reolink camera as the video source.

For other IoT devices (e.g., smart plugs, routers, etc.), we configure and integrate various existing IoT honeypots to emulate these devices, as detailed in Section 4.2. For comparison, we select several open-source honeypots to emulate these devices, utilizing their capability to mimic the same open ports and protocols. For example, Dionaea [40], which supports emulation of the UPnP protocol, is customized to emulate smart plugs. For each emulated device, we deploy virtual servers running the designated honeypots on the public internet across three geographic locations: the United States (North Virginia), France (Paris), and Japan (Tokyo).

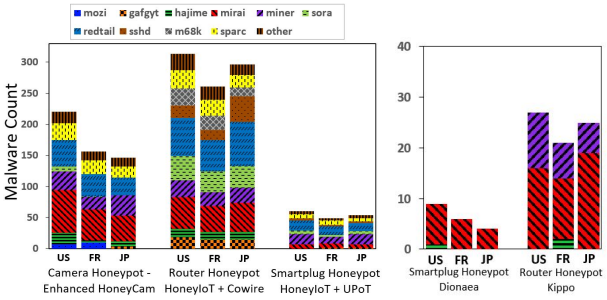
5.1.2 Basic Statistics

Table 3 summarizes the basic statistics of our online deployment. Different IoT devices support various services, operate on different ports, and are emulated by different Honeypots, resulting in varying volumes and types of attack traffic. For example, the smart plug emulated by HoneyIoT and UPoT received 17,658 attack sessions from 1,438 unique attacker IPs, whereas the router honeypot emulated by HoneyIoT and Cowire received 97,894 attack sessions from 6,270 unique attacker IPs. This discrepancy is due to the different ports and services supported by each device. For example, the router honeypot opens port 80 for web services and port 22 for SSH services, which are commonly targeted by attackers, while smart plugs typically lack these services.

The performance of devices emulated by different honeypots varies significantly. For example, router honeypots emulated by the open-source honeypot Kippo [41] cannot engage attackers effectively due to their fixed response logic and identifiable fingerprints [46, 47]. As a result, it can not effectively detect attacker's command-and-control server or collect malware. In contrast, Cowire and HoneyIoT, which are medium to high interaction honeypots equipped with built-in crawlers, can actively collect malware for further analysis.

We have also crawled malware from attacker's malware distribution server and utilized VirusTotal [48] for malware classification and analysis. Figure 9 illustrates the malware collected by different honeypots across various geographic locations based on their signatures. Taking the malware collected by the Enhanced HoneyCam in US as an example, about 58.8% of the samples were identified as botnet malware. The majority of these are variants and successors of the well-known Mirai botnet [49], of which we collected 69 samples. We also gathered 7 *Mozi* malware samples, which partially reuse Mirai's source code. Unlike Mirai, which typically brute-forces open ports with password cracking, *Mozi* exploits well-known vulnerabilities in the web servers of IoT devices. Other samples include 8 *Sora* and 27 *Sparc* malware variants, both of which belong to the Mirai botnet family.

The router honeypot offering SSH services captures some



(a) (b)

Figure 9: Malware collected by honeypots emulating different IoT devices in the United States, France, and Japan. (a) Malware collected by honeypots customized or utilized by CPDS. (b) Malware collected by other open-source honeypots capable of emulating these IoT devices.

different botnet malware such as *m68k*. VirusTotal analysis of m68k reveals signatures characteristic of both *Gafgyt* [50] and *Mirai* malware, suggesting that attackers have merged these two well-known malware families to create a new, hybrid threat. Furthermore, we have collected a substantial amount of cryptocurrency mining malware, such as *Redtail* [51], which targets vulnerabilities in off-the-shelf devices (e.g., TP-Link routers) to exploit their computational resources for mining cryptocurrency. A small portion of the samples remains unidentified by VirusTotal and are categorized as "other."

Smart plug honeypots collect fewer attack traces and malware samples than other honeypots, likely due to their limited exposure to widely targeted services like Telnet, SSH, or HTTP. For example, smart plug honeypots emulated by HoneyIoT and UPoT in the US collected 23 botnet malware samples (including Mirai, Sora, Sshd, and Sparc), 32 cryptocurrency mining malware samples (including Redtail), and 5 "other" samples.

As shown in Figure 9 (b), we also analyzed malware samples collected by other open-source honeypots capable of emulating IoT devices over similar protocols. For example, we customized the Dionaea honeypot [40] to open the same ports as a smart plug. When deployed in the United States, the Dionaea honeypot collected only eight botnet malware samples, primarily outdated Mirai variants. This suggests that more sophisticated attackers may fingerprint [46, 47] and bypass Dionaea honeypots during their scanning and attack processes. To address this, we rely on more adaptive and high-interaction honeypots, such as Cowrie and HoneyIoT, to support CPDS effectively.

5.1.3 Enhanced HoneyCam

Figure 10 illustrates the cumulative distribution function (CDF) of attackers' video access times for four camera honeypots: *recorded-video, real-camera, original HoneyCam*, and *our enhanced HoneyCam*. The results show that most attackers quickly lost interest in the recorded-video based camera

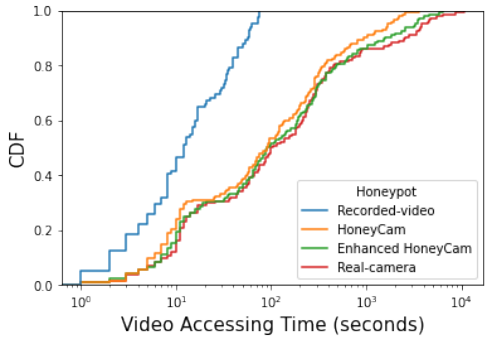


Figure 10: CDF of attacker video accessing time

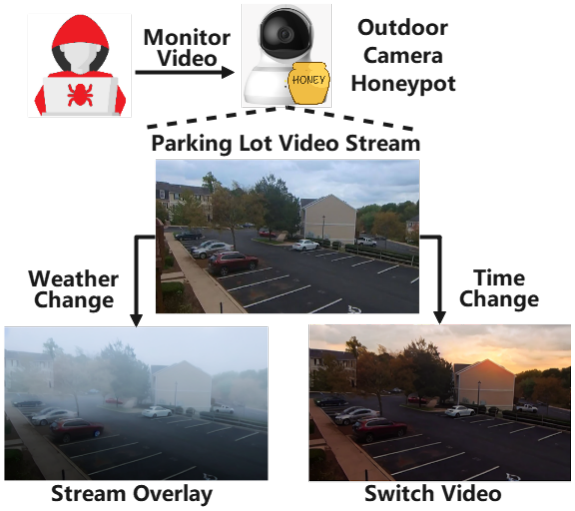


Figure 11: Real demo attack scenario (a). Our enhanced HoneyCam, which emulates the outdoor camera, switches to appropriate video as time changes and overlays corresponding weather effects in response to weather changes.

honeypot. In contrast, HoneyCam, our enhanced HoneyCam, and the real camera all engaged attackers for significantly longer periods. Notably, we observed more frequent repeated visits to our enhanced HoneyCam and the real-camera based honeypot, resulting in longer overall video access times compared to the original HoneyCam in some instances. Some attackers even spent a cumulative time of over an hour interacting with our enhanced HoneyCam. These findings demonstrate that our enhanced HoneyCam exhibits a high deception capability comparable to that of a real camera-based honeypot. The extended engagement times and repeated visits suggest that our enhancements have significantly improved HoneyCam's ability to mimic genuine camera behavior, thereby increasing its effectiveness in deception.

5.2 Real Demo

To evaluate the deception quality of CPDS against human attackers, we construct two typical scenarios:

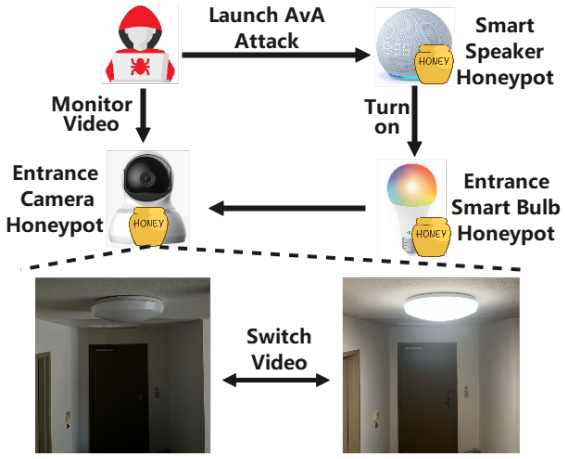


Figure 12: Real demo attack scenario (b). Attackers leverage the Alexa vs Alexa (AvA) Attack to remotely control the smart bulb's on/off state. Accordingly, our enhanced HoneyCam adjusts its video stream based on the smart bulb's status.

- Scenario (a): The attacker has compromised a surveillance camera. The attacker can obtain the live video stream, and send camera control commands (e.g., pan, tilt and zoom) to change the viewport. Scenario (a) aims to evaluate the effectiveness of maintaining "Cross-Layer Consistency (a)" shown in Figure 1.
- Scenario (b): The attacker has launched voice command injection attack (i.e., Alexa vs Alexa Attack [11], Barrier-Bypass [17], etc.) and can control the smart bulb through injected commands. The attacker has also compromised a smart camera and is able to see the smart bulb through the video stream. This scenario aims to evaluate the effectiveness of maintaining "Cross-Layer Consistency (b)" shown in Figure 1.

For comparison, we emulate these scenarios with three different system set ups: **our CPDS**, **Real Device**, and **Baseline**. In the **Real Device** setup, actual IoT devices are deployed to implement these scenarios, while the **Baseline** setup uses a simplified version of our deception system where each honeypot operates independently without coordination. To simulate real-world interactions, we recruit human participants to act as attackers. They send attack commands and monitor device statuses across all setups.

In scenario (a), CPDS utilize a modified version of HoneyCam [33] to serve as the video source. Given the specific requirements of this scenario, such as emulating an outdoor surveillance camera, we need to simulate potential environmental changes (e.g., time of day, weather changes, etc.). To achieve this, we enhance the video processing unit module of HoneyCam by integrating a stream overlay function. This allows us to superimpose additional overlay videos, representing fog or rain, onto the generated video stream based on the current weather conditions, as shown in Figure 11. To

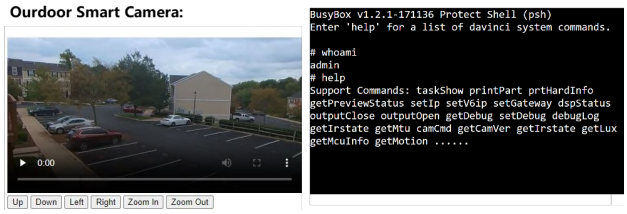


Figure 13: Survey webpage for scenario (a). Users may view the video stream and control the camera through the buttons below, or through the ssh session on the right.

simulate time changes, we prepare multiple video streams recorded during different time periods of the day and select the appropriate stream based on the current time. For the real device setup, the video stream is sourced directly from a real IoT camera, similar to Siphon [30]. For baseline, we simply utilize a traditional recorded-video based camera honeypot [45] which repeatedly play a one-hour long video footage. To ensure privacy, we review all prepared video streams in HoneyCam and set the resolution of the live video stream to 360p.

For scenario (b), as shown in Figure 12, we assume that the attacker launches voice command injection attack and sends turn on or off command to a connected smart bulb. Meanwhile, the attacker has compromised a smart camera and is able to see the smart bulb through the video stream. To simplify the demo, we aggregate the attack actions into a single attack button on the monitoring web page. We also provide a live video stream window, camera control buttons, and smart bulb status on the monitoring web page. For CPDS, we model the vulnerability, perform dependency analysis, and set up corresponding coordination policies. Similar to scenario (a), we use HoneyCam to emulate the smart camera. We use a combination of HoneyIoT and UPoT to emulate the smart bulb and a combination of HoneyIoT and Cowire to emulate the smart speaker. For the real device-based demo, the video stream is sourced from an IoT camera, and the attack button directly controls the smart speaker to change the switch status of the smart bulb. For the baseline setup, we simply remove the coordination module where the smart bulb honeypot, smart camera honeypot and smart speaker honeypot work independently.

Figure 13 shows an example of the survey webpage for the outdoor smart camera, which includes a live video stream, camera control buttons, and a reverse shell session. A similar web interface exists for attack scenario (b). We provide users with comprehensive information about the available shell commands for the smart camera, including their usage syntax and sample outputs. Users can view the video stream and control the camera using the buttons below or via the SSH session on the right for as long as they want.

Users are informed in advance that there is a 50% chance that the provided device status and video stream are from real devices, and a 50% chance that they are generated by

		User Identify System As	
Scenario	**System Type**	Real System	Fake System
Scenario (a)	Real Device	23	6
	CPDS	21	5
	Baseline	10	14
Scenario (b)	Real Device	19	3
	CPDS	18	5
	Baseline	3	21

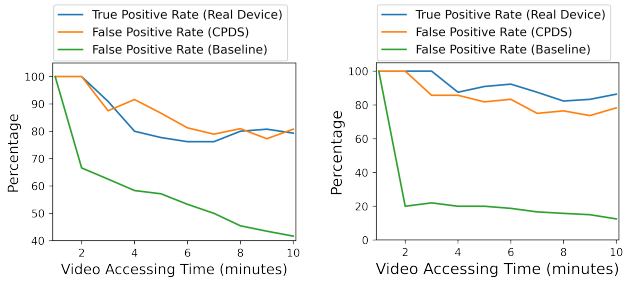
Table 4: Survey Results

honeypots and therefore fake. Upon completion, participants fill out a brief survey assessing whether they believe they are interacting with genuine IoT devices or honeypots. The survey results fall into four categories:

- True Positive: The scenario is set up by real IoT devices, and human users correctly identify it as such.
- True Negative: The scenario is set up by honeypots (e.g, CPDS or baseline system), and human users correctly identify it as such.
- False Positive: The scenario is set up by honeypots (e.g, CPDS or baseline system), and human users falsely identify it as being from real IoT devices.
- False Negative: The scenario is set up by real IoT devices, and human users falsely identify it as being from honeypots (e.g, CPDS or baseline system).

We collected all survey responses from users who remained on the monitoring webpage for more than one minute. For scenario (a), 79 valid surveys were collected, comprising 29 surveys from setups using real IoT devices, 26 from CPDS setups, and 28 from the baseline. As shown in Table 4, the real IoT device setup resulted in 23 true positive cases and 6 false negative cases, resulting in an overall True Positive Rate (TPR) of 79.3%. The TPR reflects the rate of correct user identification when the scenario was established using real devices. For the CPDS setup, there were 21 false positive cases and 5 true negative cases, resulting in an overall False Positive Rate (FPR) of 80.7%. This FPR represents the rate at which users incorrectly identified CPDS as real, signifying the success rate of CPDS in deceiving users. For the baseline setup, there were 10 false positive cases and 14 true negative cases, leading to an overall FPR of 41.7%.

To further analyze the deception capabilities of CPDS compared to real IoT devices and the baseline, we classify users based on their video access time (i.e., the time spent on the monitoring webpage). Figure 14 shows the accumulated TPR and FPR over time, categorized at one-minute intervals. The trends for the blue line (TPR for scenario setup by real device) and the orange line (FPR for scenario setup by CPDS) are highly similar, indicating that users behave similarly regardless of whether the scenario is set up with real IoT devices or CPDS. In contrast, the green line (FPR for scenario setup



(a) Scenario (a) (b) Scenario (b)

Figure 14: True (and False) Positive Rate over video accessing time, which shows that users cannot differentiate between scenarios created by real IoT devices and CPDS, but can easily identify scenarios set up by baseline.

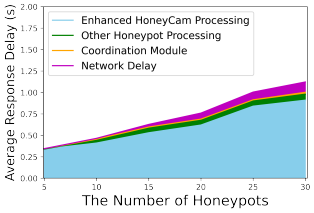


Figure 15: Response delay against one attack action

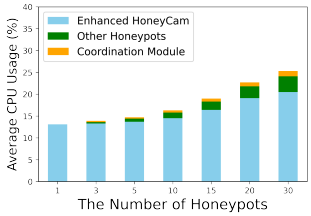


Figure 16: Resource consumption distribution

by baseline) is way lower, indicating that users notice that they interact with fake IoT cameras especially when watching video for a longer time.

For scenario (b), there are 69 valid surveys, with 21 set up by real IoT devices, 23 by CPDS, and 24 by baseline. As shown in Table 4, we observe a total of 19 true positive cases and 3 false negative cases, resulting in an overall TPR of 86.3% with real device. For comparison, when the scenario is setup by CPDS, there are 18 false positive cases and 5 true negative cases, leading to an overall FPR of 78.2%. When the scenario is setup by baseline, there are 3 false positive cases and 21 true negative cases, leading to an overall FPR of 12.5%. The trends of FPR and TPR over video accessing time are shown in Figure 14 (b). The FPR in the baseline setup (the green line) clearly shows that most users can quickly identify that they are interacting with a fake IoT system, likely because the video remain unchanged when they turn off the light.

The similarity between the TPR in the real device setup and the FPR in CPDS across both scenarios suggests that users cannot differentiate whether they are interacting with real IoT devices or CPDS. This indicates that CPDS effectively emulates real-world IoT environments, achieving a high level of fidelity comparable to genuine IoT devices in both scenarios.

5.3 Scalability

To evaluate the scalability of our deception system, we design a simulation experiment. We prepare 20 distinct IoT system setups (similar to Figure 1) for our deception system to emulate. These setups vary in the number and types of IoT devices, physical environments (e.g., floor plans, device locations), and network typologies.

For each IoT system, we follow the implementation steps discussed in Section 4. The simulation was conducted on our backend server described in Section 5.1.1. We also prepared 25GB of video files for our enhanced HoneyCam.

Figure 15 illustrates the average response delay with respect to honeypot density, defined as the average number of emulated devices located in the same room. Here, the response delay for CPDS includes the honeypot processing time (covering our enhanced HoneyCam, HoneyIoT, UPoT, Cowrie, etc.), the coordination module processing time, and the simulated network delay during coordination. As honeypot density increases, attack actions are likely to affect more honeypots, leading to an increase in average response delay. Breaking down the delay further, a significant portion is attributed to the generation of interactive video streams to emulate the physical functionalities of IoT cameras. In contrast, the delay introduced by our coordination module is at the millisecond level and thus negligible.

Figure 16 presents the estimated CPU usage of CPDS consisting of different honeypots. For all simulation cases, we have one enhanced HoneyCam instance emulating an IoT camera and multiple HoneyIoT, Cowrie, and UPoT instances emulating other IoT devices. We observe that the majority of CPU usage is consumed while generating interactive videos, and our coordination module consumes few resources even when coordinating 30 honeypots.

These results show that our coordination module introduces negligible cost while emulating physical dependencies and coordinating individual IoT honeypots. This finding underscores the feasibility and scalability of our deception system.

6 Conclusions

In this paper, we addressed the pressing need for a comprehensive defense strategy that extends beyond isolated honeypots, given the interconnected nature of IoT devices. To meet this challenge, we designed, implemented, and evaluated a cyber-physical deception system (CPDS) capable of mimicking the intricate cyber-physical connections among IoT devices by coordinating various IoT honeypots. Our approach involved modeling the vulnerabilities of individual IoT devices through collection and analysis of attack traces, analyzing physical and network dependencies among IoT devices and formulating them as Prolog rules, and coordinating honeypots based on attacker actions and dependency rules to ensure cross-layer consistency. We implemented CPDS by leveraging software-defined networking, enhancing existing IoT honeypots, and configuring them to work in concert. Extensive online deployments, human evaluations, and simulation results have validated the effectiveness of CPDS in terms of both scalability and fidelity.

7 Acknowledgments

This research was partially supported by the National Science Foundation under grant #2125208 and by the Army Research Laboratory under Cooperative Agreement Number W911NF-24-2-0132. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Army Research Laboratory, or the U.S. Government.

8 Ethics Considerations

Since the real demo experiment in Section 5.2 involves user participation based on surveys, it has been approved by our Institutional Review Board (IRB). Participants are fully informed about the purpose of the experiments. Apart from this aspect, this research does not involve any personal data, sensitive information, or vulnerability discovery and therefore does not raise specific ethical concerns. The study focuses on the design of a cyber-physical deception system through honeypot coordination, which is purely technical in nature. As such, no ethical approvals or considerations were required.

9 Compliance with the Open Science Policy

This work complies with the newly introduced Open Science Policy by USENIX. All code will be made publicly available, and demos (i.e., similar to Section 5.2) will be provided to illustrate the deception scenarios discussed. The URL of the Zenodo repository is https://doi.org/10.5281/zenodo.1472979

References

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

[2] Apple Homekit. https://www.apple.com/shop/homekit.

[3] Huawei AI Life. https://consumer.huawei.com/ai-life/.

[4] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying IoT security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Commun. Surveys & Tutorials*, April 2019.

[5] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. *IEEE Symp. on Security and Privacy*, May 2019.

[6] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, and M. Antonakakis. The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. *USENIX Security Symp.*, 2021.

[7] M. Ozmen, X. Li, A. Chu, Z. Celik, and X. Zhang B. Hoxha. Discovering IoT Physical Channel Vulnerabilities. *ACM CCS*, 2022.

[8] W. Ding and H. Hu. On the Safety of IoT Device Physical Interaction Control. *ACM CCS*, 2018.

[9] P. Vervier and Y. Shen. Before Toasters Rise Up: A View into the Emerging IoT Threat Landscape. *RAID*, 2018.

[10] CVE-2013-1599: Command Injection Vulnerability in D-Link Cameras. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1599.

[11] S. Esposito, D. Sgandurra, and G. Bella. Alexa versus alexa: Controlling smart speakers by self-issuing voice commands. *ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022.

[12] G. Xue, Y. Wan, X. Lin, K. Xu, and F. Wang. An Effective Machine Learning Based Algorithm for Inferring User Activities From IoT Device Events. *IEEE Journal on Selected Areas in Commun. (JSAC)*, July 2022.

[13] Z. Fang, H. Fu, T. Gu, P. Hu, J. Song, T. Jaeger, and P. Mohapatra. IOTA: A framework for analyzing system-level security of IoTs. *ACM/IEEE Conference on Internet of Things Design and Implementation (IoTDI)*, 2022.

[14] W. Ding, H. Hu, and L Cheng. IoTSafe: Enforcing Safety and Security Policy with Real IoT Physical Interaction Discovery. *NDSS*, 2022.

[15] A. Costin, A. Zarras, and A. Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. *ACM CCS*, 2016.

[16] T. Sugawara, B. Cyr, S. Rampazzi, D. Genkin, and K. Fu. Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems. *USENIX Security Symp.*, 2020.

[17] P. Walker, T. Zhang, C. Shi, N. Saxena, and Y. Chen. Barrier-Bypass: Out-of-Sight Clean Voice Command Injection Attacks through Physical Barriers. *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2023.

[18] Nmap: Open-source network scanner. https://nmap.org/.

[19] Masscan. https://github.com/robertdavidgrah/masscan.

[20] Y. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: Analysing the Rise of IoT Compromises. *USENIX Workshop on Offensive Technol.*, 2015.

[21] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. Chen, T. Xu, Y. Chen, and J. Yang. Understanding Fileless Attacks on Linux-Based IoT Devices with HoneyCloud. *ACM MobiSys*, 2019.

[22] A. Vetterl, and R. Clayton. Honware: A Virtual Honeypot Framework for Capturing CPE and IoT Zero Days. *APWG Symp. on Electronic Crime Research*, 2019.

[23] T. Luo, Z. Xu, X. Jin, Y. Jia, and X. Ouyang. IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Devices. *Black Hat*, 2017.

[24] J. Daubert, D. Boopalan, M. Mühlhäuser, and E. Vasilomanolakis. HoneyDrone: A medium-interaction unmanned aerial vehicle honeypot. *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2018.

[25] E. López-Morales, C. Rubio-Medrano, A. Doupé, Y. Shoshitaishvili, R. Wang, T. Bao, and G. Ahn. HoneyPLC: A Next-Generation Honeypot for Industrial Control Systems. In *ACM CCS*, 2020.

[26] C. Guan, G. Cao, and S. Zhu. HoneyLLM: Enabling Shell Honeypots with Large Language Models. *IEEE Conf. on Commun. and Network Security (CNS)*, 2024.

[27] T. Yu, Y. Xin, and C. Zhang. HoneyFactory: Container-Based Comprehensive Cyber Deception Honeynet Architecture. *Electronics)*, January 2024.

[28] A. Tambe, Y. Aung, R. Sridharan, M. Ochoa, N. Tippenhauer, A. Shabtai, and Y. Elovici. Detection of Threats to IoT Devices Using Scalable VPN-Forwarded Honeypots. *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2019.

[29] B. Wang, Y. Dou, Y. Sang, Y. Zhang, and J. Huang. IoTCMal: Towards a Hybrid IoT Honeypot for Capturing and Analyzing Malware. In *IEEE Int'l Conf. on Commun. (ICC)*, 2020.

[30] J. Guarnizo, A. Tambe, S Bhunia, M. Ochoa, N. Tippenhauer, A. Shabtai, and Y. Elovici. Siphon: Towards Scalable High-Interaction Physical Honeypots. *ACM Workshop on Cyber-Physical System Security (CPSS)*, 2017.

[31] M. Hakim, H. Aksu, A. Uluagac, and K. Akkaya. U-PoT: A Honeypot Framework for UPnP-Based IoT Devices. *IEEE Int'l Performance Computing and Commun. Conf. (IPCCC)*, 2018.

[32] C. Guan, H. Liu, G. Cao, S. Zhu, and T. La Porta. HoneyIoT:Adaptive High-Interaction Honeypot for IoT Devices Through Reinforcement Learning. *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2023.

[33] C. Guan, X. Chen, G. Cao, S. Zhu, and T. La Porta. HoneyCam: Scalable High-Interaction Honeypot for IoT Cameras Based on 360-Degree Video. *IEEE Conf. on Commun. and Network Security (CNS)*, 2022.

[34] Z. Zhao, S. Srinivasa, and E. Vasilomanolakis. SweetCam: an IP Camera Honeypot. *Workshop on CPS&IoT Security and Privacy (CPSIoTSec)*, 2023.

[35] H. Lin, C. Li, J. Yang, Z. Wang, L. Fan, and C. Duan. CP-IoT: A Cross-Platform Monitoring System for Smart Home. *NDSS*, 2024.

[36] T. Hu, D. Dubois, and D. Choffnes. BehavIoT: Measuring Smart Home IoT Behavior Using Network-Inferred Behavior Models. *ACM on Internet Measurement Conference (IMC)*, 2023.

[37] J. Ortiz, C. Crawford, and F. Le. DeviceMien: network device behavior modeling for identifying unknown IoT devices. *International Conference on Internet of Things Design and Implementation (IoTDI)*, 2019.

[38] Cowrie SSH/Telnet Honeypot. https://github.com/cowrie/cowrie.

[39] FFmpeg: A Complete, Cross-Platform Solution to Record, Convert and Stream Audio and Video. https://ffmpeg.org/.

[40] Dionaea honeypot. https://github.com/DinoTools/dionaea.

[41] Kippo: Lightweight SSH Honeypot. https://github.com/desaster/kippo.

[42] Glutton: Generic Low Interaction Honeypot. https://github.com/mushorg/glutton.

[43] Mininet. https://mininet.org/.

[44] Insta360 ONE X2: The pocket camera crew. https://www.insta360.com/product/insta360-onex2.

[45] A. Ziaie Tabari and X. Ou. A Multi-Phased Multi-Faceted IoT Honeypot Ecosystem. *ACM CCS*, 2020.

[46] S. Morishita, T. Hoizumi, W. Ueno, R. Tanabe, C. Gañán, M. van Eeten, K. Yoshioka, and T. Matsumoto. Detect Me If You... Oh Wait. An Internet-Wide View of Self-Revealing Honeypots. *IFIP/IEEE Symp. on Integrated Network and Service Management*, 2019.

[47] A. Vetterl, and R. Clayton. Bitter Harvest: Systematically Fingerprinting Low- and Medium-interaction Honeypots at Internet Scale. *USENIX Workshop on Offensive Technol.*, 2018.

[48] VirusTotal. https://www.virustotal.com/.

[49] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the Mirai Botnet. *USENIX Security Symp.*, 2017.

[50] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. Chaves, Í. Cunha, D. Guedes, and W. Meira. The Evolution of Bashlite and Mirai IoT Botnets. In *IEEE Symp. on Computers and Commun. (ISCC)*, 2018.

[51] E. Tekiner, A. Acar, and S Uluagac. A Lightweight IoT Cryptojacking Detection Mechanism in Heterogeneous Smart Home Networks. *NDSS*, 2023.