Multi-Agent Deep Reinforcement Learning for Penetration Testing of IoT Devices through their Mobile Companion App

Francesco Pagano[1], Mariano Ceccato[1], Alessio Merlo[2], and Paolo Tonella[3]

[1]University of Verona
[2]CASD - School of Advanced Defense Studies
[3]Università della Svizzera italiana

March 05, 2025

Abstract

The increasing integration of IoT devices into critical infrastructure has made them prime targets for cyberattacks. Many of these devices rely on outdated or legacy software, which introduces inherent vulnerabilities and complicates firmware updates, making identifying and testing these weaknesses essential. Traditional methods typically employ black-box approaches, mutating network requests generated during device operation to craft potential attack vectors. However, these methods face limitations when dealing with encrypted or proprietary protocols. Recent tools, such as Diane and IoTFuzzer, interact with IoT devices through their mobile companion apps and use fuzzing techniques to modify request content, causing crashes in IoT device software. Although these approaches can effectively trigger software crashes, they do not generate actual exploits, as they do not precisely target or exploit specific vulnerabilities. To address these limitations, we introduce MITHRAS, the first approach that uses mobile companion apps to deliver maliciously mutated requests directly to IoT devices, explicitly targeting Remote Code Execution (RCE) vulnerabilities. MITHRAS uses Deep Reinforcement Learning to efficiently navigate the communication code within companion apps, dynamically mutating request payloads before transmission. Adapting to previous attack outcomes, MITHRAS refines its strategy, mimicking human decision-making to improve the effectiveness of exploit generation.

Multi-Agent Deep Reinforcement Learning for Penetration Testing of IoT Devices through their Mobile Companion App

FRANCESCO PAGANO, University of Verona, Italy
MARIANO CECCATO, University of Verona, Italy
ALESSIO MERLO, CASD – School of Advanced Defense Studies, Italy
PAOLO TONELLA, Università della Svizzera italiana, Switzerland

The increasing integration of IoT devices into critical infrastructure has made them prime targets for cyberattacks. Many of these devices rely on outdated or legacy software, which introduces inherent vulnerabilities and complicates firmware updates, making identifying and testing these weaknesses essential. Traditional methods typically employ black-box approaches, mutating network requests generated during device operation to craft potential attack vectors. However, these methods face limitations when dealing with encrypted or proprietary protocols. Recent tools, such as Diane and IoTFuzzer, interact with IoT devices through their mobile companion apps and use fuzzing techniques to modify request content, causing crashes in IoT device software. Although these approaches can effectively trigger software crashes, they do not generate actual exploits, as they do not precisely target or exploit specific vulnerabilities. To address these limitations, we introduce MITHRAS, the first approach that uses mobile companion apps to deliver maliciously mutated requests directly to IoT devices, explicitly targeting Remote Code Execution (RCE) vulnerabilities. MITHRAS uses Deep Reinforcement Learning to efficiently navigate the communication code within companion apps, dynamically mutating request payloads before transmission. Adapting to previous attack outcomes, MITHRAS refines its strategy, mimicking human decision-making to improve the effectiveness of exploit generation.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: penetration testing, IoT systems, test generation

1 INTRODUCTION

Internet of Things (IoT) devices have been growing in number over the years. The number of active devices has increased from 3.4 billion to 5.55 billion, according to Exploding Topics [20]. Kumar et al. [31] studied IoT devices in people's homes, showing how these devices spread worldwide and which companies make the most of them.

The rapid growth of IoT has raised security concerns as such devices often expose several vulnerabilities, as shown by the increasing number of reported Common Vulnerabilities and Exposures (CVEs). For example, the number of CVEs for routers increased from 134 in 2019 to 435 in 2023 [12], demanding reliable methodologies to discover and fix vulnerabilities. Recent studies [1, 46] demonstrated that most of these vulnerabilities are in the firmware. The firmware includes the operating system, the startup software, and the files that control the device's operation.

Several research projects focus on identifying security issues in IoT devices, such as Karonte [39] and FirmUp [14]. These solutions primarily use static analysis techniques to detect vulnerabilities within the firmware code, often resulting in many false positives. They do not identify specific inputs that could exploit these vulnerabilities. Other research works, such as Snipuzz [21] and Labrador [32], aim to automatically identify firmware vulnerabilities at runtime by executing software within the firmware. These solutions work in a black-box manner, crafting malicious

Authors' addresses: Francesco Pagano, francesco.pagano@univr.it, University of Verona, Verona, Italy; Mariano Ceccato, mariano.ceccato@univr.it, University of Verona, Verona, Italy; Alessio Merlo, alessio.merlo@unicasd.it, CASD – School of Advanced Defense Studies, Rome, Italy; Paolo Tonella, paolo.tonella@usi.ch, Università della Svizzera italiana, Lugano, Switzerland.

requests to the IoT device by collecting and modifying a set of requests generated by the IoT device itself during regular operation. However, these methods are ineffective if the network requests are encrypted or use proprietary protocols.

To address the challenge of making effective malicious requests to IoT devices, solutions such as those proposed in [5] and [38] take advantage of existing client applications, like companion mobile apps, that interact with IoT devices. These clients already implement all the necessary communication functions, allowing users to interact with IoT devices through smartphones. Previous research used these apps to modify the content of network requests directed to the companion app's connected IoT device before they are packaged into network requests. This approach aims to identify inputs that may cause the IoT device's program to crash. These solutions have demonstrated greater effectiveness in generating valid requests accepted by the connected IoT device than approaches that attempt to craft requests from scratch. While they additionally leverage static analysis techniques on the companion app to guide it towards the communication functions responsible for interacting with the IoT device, these solutions still operate in a complete black-box manner concerning the IoT device, which introduces several limitations:

- **Limited feedback on the attack's success.** This limitation forces these solutions to rely on random modifications of the request's content, with feedback depending on either a network response or its absence. This approach can result in false positives, as a negative response code may indicate a poorly structured request rather than a successful attack. Additionally, the absence of a response could be due to network issues, such as a lost packet, rather than the outcome of the attack.
- **Restricted to simple attacks.** Due to the lack of guidance in modifying request parameters, these approaches struggle to execute complex multistage attacks. For instance, many IoT device services require valid authorization tokens, which can only be obtained by calling the appropriate login API before accessing other target APIs. Additionally, these target APIs often require supplementary data that must be retrieved from another API exposed by the IoT device beforehand.
- **Inability to pinpoint the location of the vulnerability.** The absence of detailed firmware information prevents these approaches from effectively locating the vulnerable code within the firmware of the IoT device. This limitation requires users to manually inspect the firmware code to identify the vulnerability, relying on requests that successfully conducted the attack.
- **Lack of coverage of potentially vulnerable services.** A complete black-box approach prevents these solutions from focusing on and exploiting potentially vulnerable services that are accidentally missed when performing black-box attacks.

To address these limitations, we have developed MITHRAS. This tool uses a combination of static and dynamic analysis to exploit Remote Code Execution (RCE) vulnerabilities inside the `php` code of the IoT's firmware device through its companion app. MITHRAS performs static instrumentation on the firmware code to insert a logging code that supports the dynamic analysis phase. It also analyzes the companion app's compiled code to identify the methods responsible for communication between the smartphone and the IoT device.

During the dynamic analysis phase, MITHRAS leverages multi-agent Deep Reinforcement Learning (Deep RL) techniques. Although existing black-box fuzzing techniques are efficient in quickly determining actions to perform, they are not well suited for complex problems, like multistage attacks, that require a sequence of actions to be solved. Additionally, black-box fuzzers do not learn from past experiences to select future actions, making them less adaptable to complex scenarios. These limitations are addressed by Deep RL, which learns a policy over time to tackle complex problems by mimicking human behavior.

MITHRAS introduces a novel multi-agent Deep RL paradigm for efficient interaction with Mobile-IoT environments. It employs two specialized agents: the `Layout` agent, which explores the companion app, and the `Payload` agent, which crafts malicious payloads to exploit IoT device vulnerabilities. Unlike conventional architectures assuming agent independence, MITHRAS adopts a hierarchical, sequential structure where the Layout Agent's actions guide the Payload Agent. The Payload Agent's observation space depends on the app state achieved by the Layout Agent. With separate Q-functions and inter-agent coordination, this paradigm enables a unique master-slave relationship optimized for the complexities of Mobile-IoT ecosystems.

Using these methods, MITHRAS can effectively exploit vulnerabilities in IoT devices, overcoming the limitations of previous approaches. MITHRAS is the first solution that uses Deep RL techniques to reach the vulnerable code inside the IoT device's firmware, crafting and refining the malicious payloads based on the feedback received from the instrumented IoT device's firmware. This approach allows MITHRAS to effectively identify and exploit security vulnerabilities, making it different from previous methods that lacked precise feedback mechanisms and comprehensive interaction with the firmware.

In summary, we make the following contributions:

- We propose a static analysis approach to instrument the firmware of an IoT device, injecting the required code to get runtime feedback;
- We propose a novel multi-agent Deep RL architecture that rely on a master-slave relationship between two different agents that work on the same environment;
- We implemented the MITHRAS methodology in a tool available at: https://github.com/X3no21/Mithras;
- We tested MITHRAS on 10 firmware-companion app pairs to evaluate its efficacy in exploiting vulnerabilities.

2 RELATED WORK

IoT devices present many vulnerabilities because their code often remains legacy and is neither maintained nor updated. Over the years, several research works have emerged that focus on discovering vulnerabilities in the firmware of IoT devices without executing them, such as [6], [15], [44], and [39]. However, these solutions face the challenge of identifying vulnerabilities that can result in false positives, leading to unreliable analyzes. Moreover, these tools cannot confirm whether a vulnerability can be effectively exploited.

To address the limitations of static analysis, some solutions use dynamic analysis techniques to discover and exploit vulnerabilities in the IoT device's firmware. For example, Feng et al. [21] proposed a method that triggers the exposed APIs of an IoT device to infer the structure of request messages and employs fuzzing techniques to mutate the values within these requests. This approach operates in a black-box manner, meaning it does not know the IoT device's code configuration. Consequently, it faces challenges when mutating values in custom encoding requests. Zheng et al.[49] proposed a solution to improve the performance of whole-system emulation for IoT device programs by combining system and user emulation. They use the Afl fuzzer[22] to mutate system call parameters of the emulated binaries. However, this approach requires extensive environment customization and focuses solely on system call parameters, not vulnerabilities within the executed code.

Some solutions exploit gray-box analyses, making the vulnerability discovery process more accurate and precise than the latter research works. Du et al. [19] introduced AflIot, which instruments binaries of Linux-based IoT devices to run in an emulated environment without specific hardware peripherals. The instrumentation inserts code to provide feedback for the Afl fuzzer, aiming to discover inputs that cause crashes. However, AflIot targets one binary at a time,

using shared memory between the instrumented program and the fuzzer. Consequently, the fuzzer can handle only one program per session and must be installed on the same system as the program. Ronin [41] uses gray-box analysis techniques to discover vulnerabilities in Inter-Component Communication (ICC) within a mobile app. Using Deep RL techniques, it can automatically navigate the app's GUI and maliciously mutate the parameters of Intent [17] objects to exploit vulnerabilities at runtime. Ronin also statically instruments the app's code to inject logic that collects real-time app coverage. This allows it to calculate the distance to vulnerabilities at each step and update its internal policy accordingly. Regarding the use of Deep RL techniques for efficient navigation of the code of a companion app, Romdhana et al. [40] developed ARES. This tool uses Deep RL to maximize code coverage by mimicking user interactions with the mobile app. ARES also aims to execute sequences of actions designed to crash the app under test.

Some solutions, like IoTFuzzer [5], use companion mobile apps to mutate parameters and generate network requests to crash IoT device programs. However, IoTFuzzer cannot pinpoint specific vulnerabilities in the device code. It identifies crashes based on network response codes or the absence of responses, which can lead to false positives. For example, a bad response code might indicate a malformed request rather than a crash, and the lack of a response could be due to network issues rather than a crash. This reliance on external responses also limits the ability to inspect the device's internal state to confirm if a vulnerability was triggered.

Redini et al. [39] proposed Diane, which improves method selection for mutation compared to IoTFuzzer. Diane identifies methods in the companion app that contribute to the final request and mutates their parameters. Despite this improvement, Diane still faces the same limitations as IoTFuzzer.

To address the limitations of current state-of-the-art solutions, MITHRAS leverages Deep RL techniques to train agents that automatically exploit real-world vulnerabilities in IoT devices through their companion apps. MITHRAS introduces a novel multi-agent paradigm employing two agents in a master-slave relationship, where one agent's execution depends on and is directed by the other. This approach challenges the traditional independence of agents in multi-agent paradigms. Oroojlooy et al. [34] classify multi-agent paradigms into five categories: Independent Q-Learning, Fully Observable Critics, Value Function Factorization, Consensus, and Learn to Communicate. In the *Independent Q-Learning* model, agents learn separate Q-functions and receive distinct observations from the shared environment, operating entirely independently—contrasting with MITHRAS, where agents are interdependent. In the *Fully Observable Critic* paradigm, agents maintain separate Q-functions and action spaces but share global observations to address non-stationarity, differing from MITHRAS, where agents rely on individual observations without sharing them. The *Consensus* paradigm allows agents to communicate within sub-groups and reach agreement on solutions to update a shared policy, which contrasts with MITHRAS, as its agents do not achieve consensus but instead follow a hierarchical dependency. The *Value Function Factorization* paradigm combines agents' Q-functions into a shared Q-function, enabling agents to optimize local and global rewards. In contrast, MITHRAS's agents learn separate Q-functions with conditional dependencies. Finally, the *Learn to Communicate* paradigm enables agents to dynamically determine what information to share and with whom, while MITHRAS's agents do not explicitly learn to communicate but interact through a hierarchical relationship where one agent's decisions directly influence the other.

3 BACKGROUND

3.1 Deep Reinforcement Learning (Deep RL)

The aim of Reinforcement Learning (RL) is to train an *agent* that interacts with an *environment* by executing *actions* to achieve a specific *goal*. The agent is guided towards its goal through feedback called *rewards*, which are consequences

of its actions. At each time step t, the agent receives an observation x_t, a partial or complete view of the current environment state s_t, and selects an action a_t. After executing the action a_t, the environment transitions from state s_t to s_{t+1}. The agent receives a reward $R(x_t, a_t, x_{t+1})$, rewarding or penalizing the action taken based on its outcome. The agent's behavior is entirely defined by a policy function π, which determines which actions to take in the current state s_t. A policy can be:

- **Deterministic**: $\pi(s_t)$, meaning that in state s_t, the agent will always choose a specific action a_t.
- **Stochastic**: $\pi(a_t|s_t)$, meaning that in state s_t, the agent selects action a_t with a certain probability. In this case, there is no direct mapping between a specific state s_t and a unique action a_t.

The environment is typically modeled as a Markov Decision Process (MDP), which is represented by the following 5-tuple: $\langle S, A, R, P, \rho_0 \rangle$, where:

- S is the set of all possible states.
- A is the set of all possible actions.
- $R : S \times A \to \mathbb{R}$ is the reward function, where $r_t = R(s_t, a_t, s_{t+1})$.
- $P : S \times A \to P(s)$ is the transition probability function, where $P(s_{t+1}|s_t, a_t)$ defines the probability of transitioning from state s_t to state s_{t+1} after taking action a_t.
- $\rho_0(s)$ is the distribution of the initial state of the environment.

Deep Neural Networks (DNNs) can represent the agent's policies. DNNs are powerful function approximators capable of representing complex functions, such as the policy functions of agents operating in environments with high-dimensional state or action spaces. When DNNs represent policy functions in RL, the field is called Deep Reinforcement Learning (Deep RL). We describe two of the most well-known and widely used Deep RL algorithms in the following.

3.1.1 Soft Actor-Critic (SAC). Soft Actor-Critic [29] (SAC) is an off-policy Deep RL algorithm that has gained attention for its balance between exploration and exploitation. SAC achieves this balance by maximizing a trade-off between the expected reward and the entropy of the policy, where entropy represents the randomness in the agent's action selection. This approach encourages the agent to explore a wide range of actions, preventing it from becoming too deterministic and helping it discover better strategies.

The SAC algorithm involves learning two key elements: the policy (actor) and the value function (critic). The policy determines the probability distribution over actions for each state, while the value function estimates the expected return. SAC updates the policy by minimizing a loss function that includes both the expected return and the entropy term, leading to more exploratory and robust policies. This makes SAC particularly effective in environments with continuous action spaces, where exploration is crucial for discovering optimal strategies.

3.1.2 Trust Region Policy Optimization (TRPO). Trust Region Policy Optimization [42] (TRPO) is another prominent Deep RL algorithm, known for its stability and efficiency in policy updates. Unlike some other Deep RL algorithms that can suffer from instability during learning, TRPO uses a trust region approach to limit the size of policy updates. This constraint ensures that the new policy does not deviate too far from the previous one, which helps maintain stability and prevent the agent from making drastic changes that could lead to suboptimal performance.

TRPO optimizes the policy by maximizing a surrogate objective function while keeping the policy updates within a predefined trust region. This is typically achieved through conjugate gradient methods and line search techniques, which allow TRPO to find the optimal policy update while respecting the trust region constraint. The result is a more

stable learning process, making TRPO particularly useful in environments with high-dimensional or continuous action spaces where significant policy changes can be detrimental.

3.1.3 Algorithm's choices motivations. This research focused on Deep RL algorithms that ensure consistent and controlled policy updates, avoiding shifts that could lead to divergence or suboptimal performance. This is crucial due to the complexity of the environment, which involves both a companion app on the smartphone and an IoT device. Given these constraints, we prioritized highly stable algorithms and excluded Q-Learning [47] from consideration. We selected SAC for its diverse actions, enhancing exploration and coverage, and TRPO for reusing similar actions to maximize the reachability of promising paths. SAC, with its off-policy approach, outperforms A2C [33], DDPG [45], and TD3 [24] in terms of stability and exploration. As demonstrated by Romdhana et al. [40], SAC achieves higher coverage than DDPG due to its entropy regularization, which prevents premature convergence. TRPO, compared to PPO [43], offers stricter policy control, making it more stable in sensitive environments. TRPO also ensures more stable on-policy updates than TD3, which may suffer from instability due to its off-policy nature.

3.2 Remote Code Execution Vulnerabilities

Remote Code Execution (RCE) vulnerabilities are among software systems' most critical security issues, allowing attackers to execute arbitrary code on a remote machine. Their importance is remarked by their classification under [13], which highlights the inherent risk associated with the execution of unintended commands or scripts. RCE vulnerabilities typically arise due to several potential issues in software development, including:

- **Improper Input Validation and Sanitization**: One of the primary causes of RCE is the inadequate validation or sanitization of user inputs. When user inputs are directly used to construct system commands or scripts without rigorous validation, they can be exploited to inject and execute malicious code.
- **Deserialization of Untrusted Data**: Deserialization flaws occur when an application processes serialized data from untrusted sources without proper validation, potentially leading to executing arbitrary code embedded in the serialized data.
- **Insecure Software Configuration**: Misconfigurations, such as overly permissive execution environments or improper access controls, can expose an application to RCE vulnerabilities by allowing untrusted code execution.

```php
<?php
if (isset($_GET['cmd'])) {
    $command = $_GET['cmd'];
    system($command);
}
?>
```

Listing 1. Example of an RCE vulnerability inside a PHP code snippet

Listing 1 shows an example of a PHP script that directly passes user input from the cmd parameter of a GET request to the `system()` function. The input to the `system()` function is not correctly sanitized, leading to the execution of arbitrary commands on the system without any control.

4 METHODOLOGY

This section describes MITHRAS, our approach to automatic penetration testing of IoT devices based on Deep RL. The MITHRAS's approach is composed of two phases: i) an analysis phase on the mobile companion app associated with

the IoT device and the static instrumentation of the IoT device's firmware; ii) a dynamic exploration phase based on Deep RL techniques to execute penetration tests on the IoT device through its companion app.

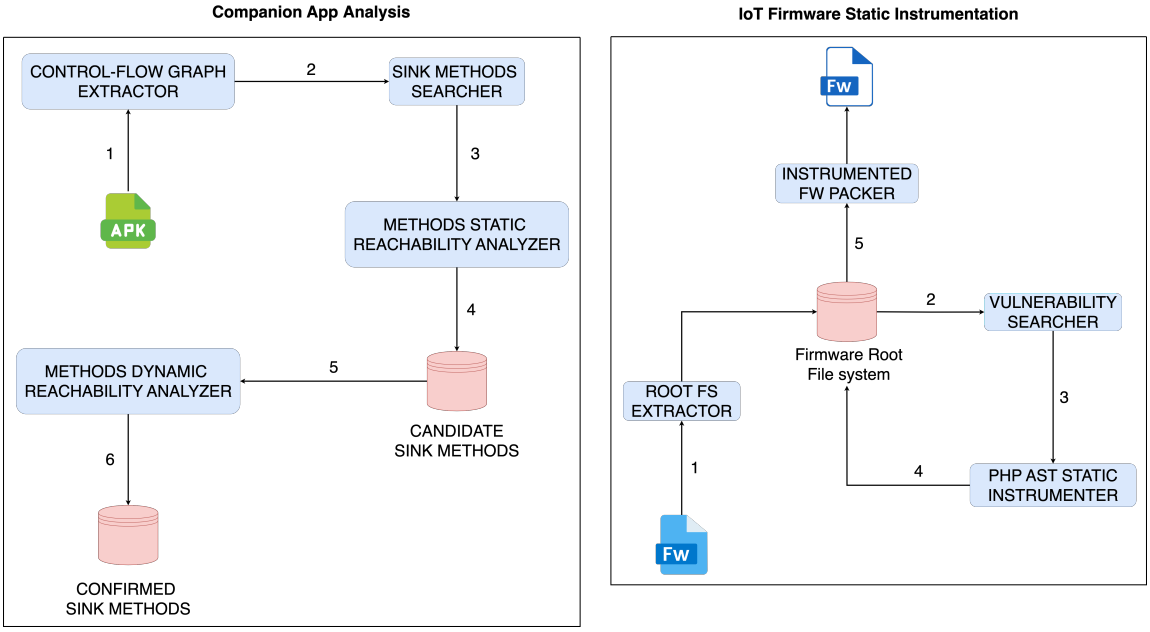

Fig. 1. Setup Phase

4.1 Setup Phase

Figure 1 provides an overview of MITHRAS's setup phase, which is divided into two parts: (i) static analysis of the IoT device's companion app to identify methods that handle communication between the smartphone and the IoT device, and (ii) static analysis of the IoT device's firmware to search for RCE vulnerabilities and instrument the firmware with code that generates runtime feedback from the actions taken by the Deep RL agent.

4.1.1 Companion App Analysis. The `Control-Flow Graph Extractor` module begins by extracting the app's `.smali` code from its compiled artifact and constructing a comprehensive control flow graph (Step 1). Once the graph is established, the `Sink Methods Searcher` module identifies all methods responsible for handling network requests to the IoT device, referred to as candidate `sink` methods (Step 2). These sink methods manage to send requests to the IoT device and process its responses. Next, the `Methods Static Reachability Analyzer` module checks whether the identified candidate sink methods are reachable from the app's layout classes by statically analyzing all possible paths originating from the layout classes' methods (Step 3). Any unreachable methods are excluded from the list, and the remaining reachable methods are stored in a buffer (Step 4). The `Methods Dynamic Reachability Analyzer` then filters the methods further, distinguishing those that initiate network requests to the IoT device based on real-time network monitoring during app execution on a smartphone (Step 5). This analysis is guided by the `Layout Agent`, described in Section 4.2, which interacts with the app at runtime to trigger the execution of candidate sink methods. If a network request to the IoT device is detected following the execution of a candidate sink method, the

method is confirmed as valid. After completing both static and dynamic reachability analyses, the confirmed sink methods are stored in a buffer (Step 6) and passed to the next dynamic analysis phase.

4.1.2 IoT Firmware Static Instrumentation. The `Root FS Extractor` module starts by unpacking the firmware's `.bin` file to extract the embedded root file system, which contains the software executed by the firmware (Step 1). After extracting the root file system, the `Vulnerability Search` module scans it for vulnerabilities, focusing specifically on remote code execution (RCE) issues within `.php` files (Step 3). This scanning process uses existing tools designed for detecting such vulnerabilities. If any RCE vulnerabilities are found, the `Php AST Static Instrumenter` module traverses each `.php` file to instrument the code (Step 4). This module modifies the original code by inserting logging mechanisms that generate feedback for the Deep RL agent during the further dynamic analysis phase. Specifically, it logs every function or method call made when a PHP script is executed and the output of any commands issued through the vulnerable execution functions identified earlier. This information is transmitted via socket to the Deep RL agent. Once the instrumentation code is generated and inserted, the module updates the `.php` files with these modifications. Finally, the `Instrumented FW Packer` module repacks the modified file system, along with the original firmware files, into a new `.bin` file (Step 5), producing the final instrumented firmware package.

4.2 Dynamic Analysis Phase

Figure 2 shows an overview of MITHRAS's dynamic analysis phase. This phase is characterized by two Deep RL agents interacting with the companion app and the IoT device. In the following, we will discuss each methodology step and detail the two Deep RL agents.

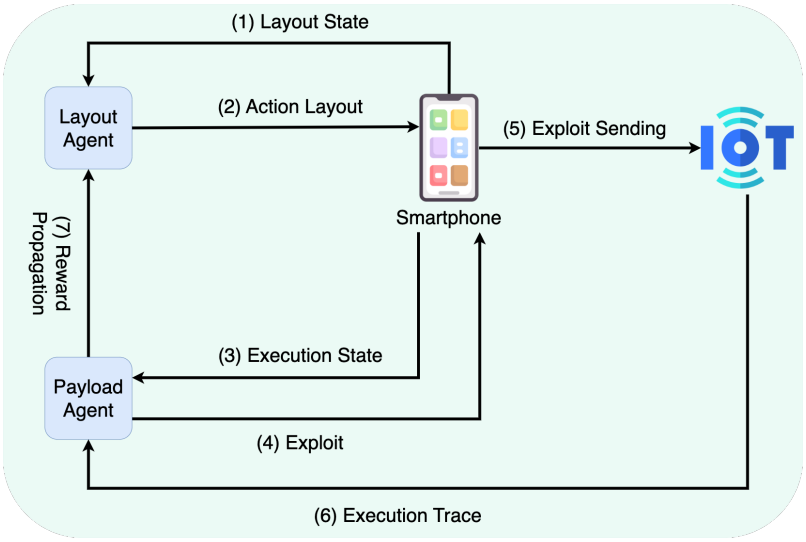

Fig. 2. Dynamic Analysis Phase

4.2.1 Overview. The `Layout Agent` begins by setting its state based on the current state of the companion app on the device (Step 1). Using this state, the `Layout Agent` selects an action from its action space and executes it on the device where the companion app is installed (Step 2). If the app's execution triggers a sink method, the `Layout`

`Agent` pauses, and the `Payload Agent` takes over. The `Payload Agent` then selects an action to execute based on the triggered sink method on the companion app (Step 3). This action involves modifying an original parameter of the request payload intended for the IoT device, potentially introducing a malicious modification. After performing the parameter modification, the `Payload Agent` injects the new modified parameter's value into the companion app's memory and resumes the app's execution (Step 4). The companion app then executes the modified payload to the IoT device (Step 5). The IoT device, in turn, sends the execution trace—detailing the methods and functions called within the `.php` files and the output of the vulnerable execution functions—to the `Payload Agent` (Step 6). The `Payload Agent` updates its state based on this execution trace and calculates the reward, which is then passed to the `Layout Agent`. The `Layout Agent` computes its internal reward based on the successful triggering of a sink method within the companion app and the impact of the malicious payload's execution (Step 7). The final reward R_l of the `Layout Agent` is calculated as follows:

$$R_l = r_l + \alpha \cdot r_p$$

Where r_l represents the reward obtained by the `Layout Agent` based on its interaction with the companion app's GUI, r_p represents the reward obtained by the `Payload Agent` after executing the attack; and α is a discount factor applied to r_p (in our implementation, set to 0.4).

4.3 Layout Agent

To apply RL to the challenge of reaching sink methods within the mobile companion app, we need to translate the problem into the standard mathematical framework of RL: a Markov Decision Process (MDP), defined by the 5-tuple $\langle S, A, R, P, \rho 0 \rangle$. Furthermore, the testing problem must be structured as an RL task, broken down into multiple finite-length episodes.

4.3.1 State Representation. The state representation of the companion app is entirely based on its GUI. The state $s_t \in S$ is expressed as a composite state $(a_0, \ldots, a_n, w_0, \ldots, w_m)$. The first component, $(a_0, \ldots, a_n)$, uses a one-hot encoding to indicate the current activity, where a_i is set to 1 only if the currently active activity is the ith one, and 0 for all other activities. The second component, w_j is set to 1 if the jth widget is available in the current activity; otherwise, it is set to 0.

4.3.2 Action Space. User interaction events in the companion app are mapped to the action set A of the MDP. MITHRAS identifies executable events by analyzing dumped widgets' attributes (clickable, long-clickable, scrollable). Each action a has three parts: the first identifies the target widget or system action; the second provides a string input (using an index from a predefined dictionary), and the third is context-dependent, determining either the action type (click or long-click) or the scrolling direction for scrollable widgets.

4.3.3 Reward Function. The RL algorithm used by MITHRAS receives a reward r_l every time it executes an action. We define the following reward function:

$$r_l = \begin{cases} \Gamma_1 & \text{if } d_t(s^*) = 0 \wedge s^* \notin \text{PrevReachedSinks}, \\ \Gamma_2 & \text{if } d_t(s^*) = 0 \wedge s^* \in \text{PrevReachedSinks}, \\ \Gamma_3 & \text{if } d_t(s^*) - d_{t-1}(s^*) \leq 0 \\ -\Gamma_3 & d_t(s^*) - d_{t-1}(s^*) > 0, \\ -\Gamma_4 & \text{companion app crashes} \end{cases}$$

where $d_t(s^*)$ indicates the distance to the closest sink s^* achieved at time t (resp. $t-1$), with $\Gamma_1 > \Gamma_4 > \Gamma_2 > \Gamma_3$ (in our implementation $\Gamma_1 = 100$, $\Gamma_2 = 70$, $\Gamma_3 = 50$, $\Gamma_4 = 80$)

At time t, the agent receives the maximum positive reward (Γ_1) if it reaches a sink function identified during the static analysis phase. It gets a smaller positive reward (Γ_2) if it revisits an already visited sink. When the agent executes an action that brings it closer to a sink compared to the previous time step, it is awarded a positive reward (Γ_3). Conversely, if the executed action increases the distance to the sink, the agent incurs a negative reward ($-\Gamma_3$). The agent also receives the maximum negative reward ($-\Gamma_4$) if the executed action causes the companion app to crash.

4.4 Payload Agent

4.4.1 State Representation. The state representation of the IoT device is based entirely on the software running on the device. The state $s_t \in S$ is represented as $(p_0, \ldots, p_n)$, a bit vector encoding to indicate the `.php` files traversed during the last execution. Expressly, p_i is set to 1 if the corresponding `.php` file was executed and 0 if it was not.

4.4.2 Action Space. Sink method parameter modification operations are mapped to the action set A of the MDP. MITHRAS extracts the sink method's parameter values from the last sink call and performs actions on them. Each action a consists of four elements: the first element identifies the sink parameter to be modified; the second element specifies the modification to be performed, which may involve adding a string (using an index from a predefined dictionary) as a prefix or suffix to the current parameter value or replacing the parameter value with a command that directly executes code on the command line (e.g., `$(id>rce)`, to execute the `id` shell command and redirect its output into a file named `rce`); the third element specifies the index of a string from a dictionary to be used as a prefix or suffix for the current sink parameter's value, while the fourth element defines an additional modification to apply to the altered payload, such as encoding special characters. This further modification is necessary because IoT devices might escape or filter special characters, such as spaces, and encoding these characters in a format acceptable to the IoT device can enhance the acceptance of the modified payload as valid.

4.4.3 Reward Function. The RL algorithm used by MITHRAS receives a reward r_p every time it executes an action. We define the following reward function:

$$r_p = \begin{cases} \Gamma_1 & \text{if vulnerability on IoT device is reached and successfully exploited,} \\ -\Gamma_3 & \text{if the timeout occurs or vulnerability on IoT device is reached and not exploited} \\ \Gamma_2 - \Gamma_3 & \text{if } d_t(v^*) - d_{t-1}(v^*) \leq 0 \\ -\Gamma_2 - \Gamma_3 & d_t(v^*) - d_{t-1}(v^*) > 0, \end{cases}$$

where $d_t(v^*)$ indicates the distance to the closest vulnerability v^* achieved at time t (resp. $t-1$), with $\Gamma_1 > \Gamma_3 > \Gamma_2$ (in our implementation $\Gamma_1 = 100$, $\Gamma_2 = 50$, $\Gamma_3 = 40$) At time t, the agent receives the maximum positive reward Γ_1 if the sink method's parameter modification successfully exploits a vulnerability within the IoT device and a negative reward $-\Gamma_3$ if it fails to do so. When the agent executes an action closer to exploiting a vulnerability than the previous step, it is awarded a positive reward $\Gamma_2 - \Gamma_3$. Conversely, if the executed action increases the distance to a vulnerability without reaching it, the agent incurs a negative reward $-\Gamma_2 - \Gamma_3$.

4.5 Multi Agent

Figure 3 illustrates the interaction between the Layout and Payload agents in our multi-agent Deep RL system. Each agent learns its Q-function, enabling independent decision-making. Instead of interacting with the environment in parallel, the agents operate sequentially. Specifically, the Payload agent's observations partially depend on the sink function executed on the IoT device and its parameter values. First, the Layout agent interacts with the companion app to reach a sink function. If the mobile app's execution reaches a sink function, it triggers the Payload agent's execution. The observation the Payload agent receives is the combination of the IoT device's state and the last sink function triggered in the companion app alongside its parameter values. This last observation depends on the Layout agent's executed action.

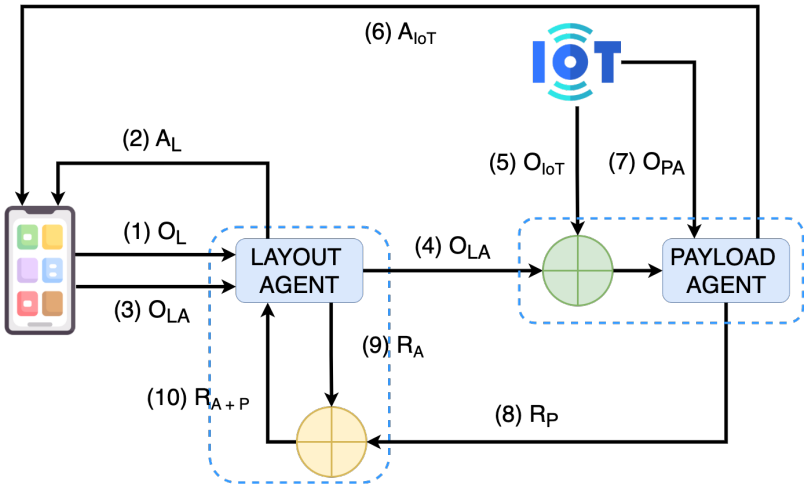

Fig. 3. Deep RL Agent interaction

Specifically, the interactions between the two agents can be described as follows: First, the Layout agent receives the observation O_L from the mobile companion app on the smartphone (Step 1). Based on O_L, it selects an action A_L to perform on the smartphone's layout (Step 2). After executing the action A_L on the smartphone, the Layout agent retrieves the observation O_{LA} from the smartphone (Step 3). This observation contains the last sink method invoked and its parameter values, which are to be maliciously mutated by the Payload agent. Next, the Payload agent receives the observation O_{LA} from the Layout agent (Step 4). It also reads the execution state of the previously executed action, O_{IoT}, from the IoT device (Step 5), combining these observations to produce the complete information needed to select the following action. The Payload agent then chooses an action, A_{IoT}, to modify the parameter values received from

the Layout agent and applies these modifications to the mobile companion app (Step 6). After executing the action A_{IoT}, the Payload agent retrieves the execution state of the IoT device, O_{PA} (Step 7). Based on this state, it computes its reward, R_P, and sends it to the Layout agent (Step 8). Meanwhile, the Layout agent computes its reward, R_A, based on the reachability of a sink method within the companion app (Step 9). The Layout agent combines its reward, R_A, with the reward R_P received from the Payload agent, producing the final reward, R_{A+P}. This final reward is used to update the Layout agent's internal policy.

4.6 Pseudocode

The pseudocode in Listing 1 outlines the core operations of the MITHRAS's dynamic analysis. Initially, the `Layout Agent` connects to the smartphone to inject the `payload manager` binary (Line 1). This binary enables communication with the companion app. The agent configures the payload manager using the `agent_file` configuration file, which specifies the methods in the companion app that need to be hooked at run-time, along with the code to be executed before the original method execution. The `Layout Agent` executes its logic for each episode, with the number of episodes provided as an input parameter (lines 3-30). At the start of each episode, the `Layout Agent` resets the state of the companion app and the IoT device (lines 3-4). During each step, the `Layout Agent` retrieves the current state of the companion app (Line 6), selects and performs actions based on the state and its internal policy (Lines 7-8), and waits until either a sink method is triggered or an internal timeout occurs (Lines 10-12). If no sink function is reached before the timeout, the `Layout Agent` computes a penalty in the reward function to reflect the failure to reach a sink function (lines 13-17). However, if a sink function is hit during the app's execution, the `Layout Agent` wakes up the `Payload Agent` and sends it the sink function parameter values (line 18). Upon activation, the `Payload Agent` retrieves the current state of the IoT device (line 19), then mutates the values of the parameter of the sink function based on both the state of the device and its internal policy (lines 20-21). After performing this mutation, the `Payload Agent` sends the modified parameter values back to the payload manager running on the smartphone (line 23) and waits until its internal timeout expires (Lines 24-26). If the `Payload Agent`'s internal timeout expires without receiving feedback, it calculates a negative reward, indicating that the exploit was unsuccessful, and updates its internal policy accordingly (lines 27-31). However, suppose a response from the IoT device is received. In that case, the `Payload Agent` retrieves the device's execution trace (line 32), computes the reward based on this trace, and updates its internal policy (lines 33-34). Finally, the `Payload Agent` sends its computed reward back to the `Layout Agent`, which also computes its reward and updates its policy accordingly (lines 35-37).

Algorithm 1 RL Agents

Input: $companion\_app\_apk, device\_firmware, agent\_file, episodes, steps, sink\_methods$
Output:

$payload\_manager \leftarrow$ loadPayloadManager($agent\_file$)
for episodes **do**
  companion_app.reset()
  iot_device.reset()
  for steps **do**
    $app\_state \leftarrow$ companion_app.getState()
    $w\_idx, i\_idx, o\_idx \leftarrow$ agent_layout.getAction(app_state)
    $agent\_layout$.performAction(w_idx, i_idx, o_idx)

    while no Timeout **do**
      wait for a **send** message from the *companion app instrumenter*
    end while
    if Timeout **then**
      $reward \leftarrow$ agent_layout.computeReward()
      agent_layout.updatePolicy($reward$)
      continue
    end if
    $agent\_layout$.sendSinkParameterValuesToPayloadAgent($sink\_method\_hit$)
    $iot\_device\_state \leftarrow$ iot_device.getState()
    $param\_idx, operator\_idx, operation\_idx, additional\_operator\_idx \leftarrow$ agent_payload.getAction(iot_device_state)
    $parameter\_value \leftarrow agent\_payload$.performAction($seedQueue$, $param\_idx$, $seed\_idx$, $operation\_idx$, $additional\_operator\_idx$)

    send $parameter\_value$ to the $payload\_manager$
    while no Timeout **do**
      wait for the reply from the *IoT device*
    end while
    if Timeout **then**
      $reward \leftarrow$ agent_payload.computeReward()
      agent_payload.updatePolicy($reward$)
      continue
    end if
    $iot\_execution\_trace \leftarrow$ getIoTProgramExecutionTrace()
    $reward\_payload \leftarrow$ agent_payload.computeReward($iot\_execution\_trace$)
    agent_payload.updatePolicy($reward\_payload$)
    $reward\_layout \leftarrow$ agent_layout.computeReward($sink\_methods$)
    $reward \leftarrow reward\_layout + \alpha * reward\_payload$
    agent_layout.updatePolicy($reward$)
  end for
end for

4.7 Implementation

The MITHRAS methodology is implemented within a framework that enables users to fully emulate a functional Mobile-IoT environment and interact with it through a multi-agent Deep RL system. MITHRAS utilizes the FirmAE framework [30] to emulate IoT devices from their firmware files. For the smartphone emulator, MITHRAS employs the Android emulation framework [16] to set up a fully functional emulator, allowing the installation of companion apps to interact with the emulated IoT devices. To implement the Layout and Payload agents, MITHRAS relies on Stable-Baselines3 [27], a robust library for reinforcement learning. For the vulnerability identification task, we leveraged Emba [25]. This tool performs various static security analyses in firmware to identify vulnerabilities, such as RCE. Emba also uses a dataset of public CVEs to check for known and certified vulnerabilities in the analyzed firmware. Based on the results of the static analysis, MITHRAS modifies the IoT firmware's `php` files using the Php-Parser library [26] to insert the logging logic. The logging code captures data at invoked sink functions, allowing us to assess which mutated inputs successfully exploited which RCE vulnerabilities. Interaction with the companion app's user interface

to reach sink functions is supported by the Appium library [2]. The Frida library [23] is used to monitor the execution of the companion app and log any triggered sink functions. When a sink function is triggered, MITHRAS captures its parameter values and forwards them to the Layout agent as a `JSON` object.

5 EVALUATION

We seek to address five research questions, divided into two distinct studies, respectively, on the sink reachability of the companion app (Sections 5.1 and 5.2) and on the vulnerability exploitation of the IoT device (Section 5.3 and 5.4).

5.1 Companion App's Sink Reachability

RQ1 [Sink Reachability]: *Which sinks identified by static analysis are reached by the Layout Agent?*
RQ2 [Efficiency]: *How does the number of sinks reached by the Layout Agent grow across episodes?*
RQ3 [RL algorithm]: *Which RL algorithm between SAC and TRPO performs better when implementing the Layout Agent? How does the sink coverage curve compare to that of a black-box fuzzer algorithm?*

This study investigates the ability of a Deep RL-based agent to automatically interact with mobile companion apps to reach sink functions identified during static analysis. We focus on coverage of unique sinks reached and the diversity of inputs that enable the agent to reach them. The goal is to determine whether a Deep RL algorithm results in more sink functions being reached within a predefined period than the black-box fuzzer algorithm. A black-box fuzzer is an algorithm that operates within the same action space as Deep RL algorithms but generates the action vector randomly at each step, without relying on a learned policy. We selected 10 pairs of Android companion apps and corresponding IoT device programs for the study, specifically focusing on apps interacting with smartwatch devices (see Table 1). To address RQ1, we used the Frida library [23] for dynamic instrumentation, collecting data on which sink functions were invoked during execution to compute coverage. For RQ2 and RQ3, we compared two Deep RL algorithms, SAC and TRPO, against the black-box fuzzer to determine which algorithm is more effective in reaching sink functions and generating diverse inputs.

Id	App	Version
App 1	com.habitrpg.android.habitica	4.2.2
App 2	com.radioplayer.mobile	7.4.1
App 3	de.komoot.android	2023.26.7
App 4	com.funnmedia.waterminder	5.1.5
App 5	com.pallo.passiontimerscoped	705.1.4
App 6	org.iggymedia.periodtracker	1.096
App 7	com.google.android.wearable.app	2.63.0.532400257
App 8	au.com.auspost.android	8.9.9-4790
App 9	com.calm.android	6.26.1
App 10	com.outdooractive.Outdooractive	3.13.4

Table 1. Companion app used in the testing campaign

We conducted the experimental campaign over 10 episodes for each mobile companion app, running each episode for 20 minutes using Deep RL and black-box fuzzer algorithms. We used the Area Under the Curve (AUC) metric to

evaluate efficiency, considering the sink coverage over a time curve. To account for the algorithms' non-determinism, each experiment was repeated 10 times. We applied the Shapiro-Wilk test to assess whether the distributions were Gaussian. If the distributions were Gaussian, we used the ANOVA test and Cohen's effect size, classified into Negligible (N), Small (S), Medium (M) and Large (L) according to the standard thresholds 0.2, 0.5, 0.8. Otherwise, we applied the Wilcoxon non-parametric test with Vargha-Delaney effect size, whose standard thresholds (applied to $2 \cdot |eff\_size - 0.5|$) are: 0.147, 0.33, 0.474.

5.2 Companion App's Sink Reachability Results

Figure 4's right-hand diagram shows the companion apps sink function coverage per algorithm, averaged across repetitions for each app. SAC and TRPO consistently outperform the black-box fuzzer algorithm, except in *App 3* and *App 10*, where coverage converges to 100% with all three techniques. Deep RL algorithms are more effective in reaching sink functions, covering over 80% in most cases.

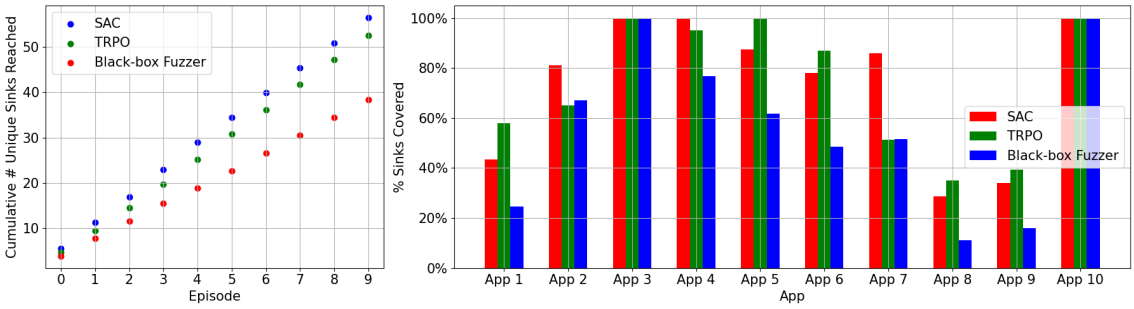

Fig. 4. i) The mean cumulative number of unique sinks reached over the episodes is averaged over all apps. ii) The mean percentage of sink functions reached across all apps is averaged over all repetitions.

Figure 4's left-hand diagram shows the cumulative number of unique sink functions reached across episodes, averaged over the selected companion apps. SAC reached the highest number of unique sink functions across episodes, outperforming TRPO and the black-box fuzzer algorithm. In particular, SAC and TRPO exhibit a similar upward trend in discovering unique sink functions over time, whereas the black-box fuzzer algorithm lags significantly behind.

Table 2 shows the average number of unique sink functions and AUC across episodes for SAC, TRPO, and the black-box fuzzer algorithm. We calculated the mean number of unique sink functions episode by episode for each app and algorithm, averaging the results between repetitions. To assess overall performance, we combined the results averaged by episode and calculated the mean for each algorithm. We applied a one-way ANOVA test for unique sink functions and calculated Cohen's *d* effect size, which was significant. For AUC, we used the paired Wilcoxon test and paired Vargha-Delaney effect size. In *App 7*, SAC discovered 25.12 unique sink functions, outperforming TRPO (14.49) and the black-box fuzzer algorithm (14.02), with a medium effect size (M). SAC also achieved a higher AUC (227.35) compared to TRPO (130.75) and black-box fuzzer (125.75). Across all apps, SAC averaged 31.27 unique sink functions, compared to TRPO's 28.20 and the black-box fuzzer's 20.99, with a large effect size (L). SAC also led in AUC (281.69 vs. 253.33 for TRPO and 188.81 for black-box fuzzer), with a large effect size (L).

Table 3 presents the average number of unique inputs that reach the sink functions and AUC across episodes for SAC, TRPO, and the black-box fuzzer algorithm. We calculated the mean number of unique inputs episode by episode

App	Average unique sinks across episodes (SAC)	Average unique sinks across episodes (TRPO)	Average unique sinks across episodes (Fuzzer)	Mean AUC across episodes (SAC)	Mean AUC across episodes (TRPO)	Mean AUC across episodes (Fuzzer)
App 1	7.66 M(Fuzzer)	9.23 M(Fuzzer)	3.95	69.40 M(Fuzzer)	82.95	35.45
App 2	8.79	6.89	7.23	79.10	61.80	65.00
App 3	150.93	134.45	103.47	1355.70	1208.90	932.45
App 4	12.70	11.54	9.16	114.55	103.90	82.30
App 5	18.65	22.77	13.26	167.40	204.70	119.10
App 6	35.71 M(Fuzzer)	38.49 M(Fuzzer)	20.42	322.70	346.85	183.00
App 7	25.12 M(Fuzzer,TRPO)	14.49	14.02	227.35	130.75	125.75
App 8	5.19 L(Fuzzer)	6.01 L(Fuzzer)	2.03	47.15	54.25	18.40
App 9	5.36 M(Fuzzer)	6.12 M(Fuzzer)	2.53	48.20	54.90	22.70
App 10	42.61	31.98	33.80	385.40	284.30	303.90
Overall	31.27 L(Fuzzer)	28.20 L(Fuzzer)	20.99	281.69 L(Fuzzer)	253.33 L(Fuzzer)	188.81

Table 2. Mean number of unique sink functions and AUC across episodes for each algorithm. Effect sizes between the best-performing algorithm (highlighted) and other ones are reported only when the p-value is statistically significant (S = Small; M = Medium; L = Large)

for each app and algorithm, averaging the results across repetitions. We also combined the results across all apps to calculate the mean for each algorithm. For unique inputs, we used a one-way ANOVA, and Cohen's *d* effect size was significant, while for AUC, we applied the Wilcoxon test with paired Vargha-Delaney effect size. In *App 7*, SAC generated 90.06 unique inputs, outperforming TRPO (36.36) and black-box fuzzer (64.64), with a medium effect size (M) compared to TRPO. Similarly, SAC achieved a higher AUC (810.65) compared to TRPO (330.05) and the black-box fuzzer (580.75), again with large effect sizes (L). Across all apps, SAC averaged 118.76 unique inputs, compared to TRPO's 48.32 and the black-box fuzzer's 87.22, with large effect sizes (L) confirmed by ANOVA. SAC also led in AUC (1069.22 vs 433.29 for TRPO and 785.96 for black-box fuzzer), showing a large effect size (L).

App	Average sinks reached with unique inputs across episodes (SAC)	Average sinks reached with unique inputs across episodes (TRPO)	Average sinks reached with unique inputs across episodes (Fuzzer)	Mean AUC across episodes (SAC)	Mean AUC across episodes (TRPO)	Mean AUC across episodes (Fuzzer)
App 1	13.18	11.95	9.35	119.45	107.55	84.80
App 2	35.82	24.32	35.91	321.70 L(TRPO)	217.50	323.40
App 3	372.88 M(TRPO)	165.41	240.72	3359.65 L(TRPO)	1484.30	2175.30
App 4	14.30	12.09	9.93	129.05 L(Fuzzer)	108.75	89.05
App 5	347.60 L(TRPO)	70.63	347.55 L(TRPO)	3112.35 L(TRPO)	633.20	3130.70 M(TRPO)
App 6	203.52 M(TRPO),L(Fuzzer)	87.37	77.73	1839.40 L(Fuzzer,TRPO)	785.20	696.70
App 7	90.06 M(TRPO)	36.36	64.64 L(TRPO)	810.65 L(TRPO)	330.05	580.75
App 8	14.23 M(TRPO),L(Fuzzer)	8.32	5.50	129.40 L(Fuzzer)	75.15	49.75
App 9	8.94	9.32	7.09	80.30	83.80	64.20
App 10	87.11	57.40	73.81	790.30	507.40	665.00
Overall	118.76 L(Fuzzer,TRPO)	48.32	87.22	1069.22 L(Fuzzer,TRPO)	433.29	785.96

Table 3. Mean number of unique inputs reaching sink functions and AUC across episodes for each algorithm. Effect sizes between the best-performing algorithm (highlighted) and the other ones are reported only when the p-value is statistically significant (S = Small; M = Medium; L = Large)

RQ1: Deep RL algorithms effectively reach sink functions, outperforming the black-box fuzzer algorithm in almost every tested app.

RQ2: SAC and TRPO outperform the black-box fuzzer algorithm in reaching unique sinks across episodes, with SAC achieving the fastest coverage of sink functions identified during static analysis.

RQ3: The results indicate that the SAC algorithm is the most effective, providing higher coverage of sink functions in the companion app and more diverse inputs. This improves code coverage and the chance of reaching APIs with potential RCE vulnerabilities. Therefore, we selected SAC as our Deep RL algorithm for the second study.

5.3 IoT Device's Vulnerability Exploitation

RQ4 [Vulnerability Reachability and Exploitation]: *Among the vulnerability instances identified by static analysis in the code, which are reached/exploited?*
RQ5 [Efficiency]: *How does the Payload Agent's number of vulnerabilities reached/triggered grow across episodes? How does the curve compare to that of a black-box fuzzer and coverage-based algorithms?*

This study investigates the ability of a Deep RL-based agent to automatically exploit vulnerabilities in the firmware of IoT devices through its companion mobile app. We focus on the number of unique IoT APIs that can be reached through the app and the number of unique maliciously mutated inputs used to exploit the RCE vulnerabilities inside the IoT API code. The goal is to determine whether a Deep RL agent, mutating input through the companion app, can exploit more vulnerabilities within a set time frame compared to a black-box fuzzer and a coverage-based fuzzer. A black-box fuzzer is an algorithm that operates within the same action space as Deep RL algorithms but generates the action vector randomly at each step, without relying on a learned policy. A coverage-based fuzzer is an algorithm that operates within the same action space as Deep RL algorithms. Instead of learning a policy, it selects the next action to perform on the IoT device to approach a vulnerability identified during the static analysis phase incrementally. Specifically, the algorithm stores payloads demonstrating progress toward the vulnerability and reapplies actions to these payloads, iteratively reducing the distance to the target vulnerability in the IoT device.

We selected firmware from 10 D-Link IoT routers [18], as they are publicly available and primarily run `php` code. We used the *D-Link Wi-Fi* app [35] (version *1.4.8*) to interact with the firmware during testing. Table 4 lists the firmware versions and the corresponding models.

Id	Model Name	Firmware Version
Frmw 1	DIR-645	FW105B01
Frmw 2	DIR-818L	FW105b01
Frmw 3	DIR-822 B1	FW202KRb06
Frmw 4	DIR-822 C1	FW303WWb04_i4sa_middle
Frmw 5	DIR-846	enFW100A53DLA-Retail
Frmw 6	DIR-860L B1	FW203b03
Frmw 7	DIR-868L B1	FW203b01
Frmw 8	DIR-880L A1	FW107WWb08
Frmw 9	DIR-890 A1	FW111b04
Frmw 10	DIR-890L A1	FW100b25

Table 4. Firmware used in the testing campaign

We ran the selected firmware on a Raspberry Pi 5 device [37] to simulate a router, enabling the connection to the mobile companion app installed on a smartphone. To minimize overhead, we used a lightweight Linux distribution built with the Yocto [48] build system. The Raspberry Pi ARM [3] CPU allowed us to run the ARM-based firmware natively. For non-ARM architectures, like MIPS, we used the Firmadyne [4] emulation tool. To address RQ4, we first detected RCE vulnerabilities in the `php` code using Emba [25]. Based on the static analysis results, we modified the IoT firmware's `php` files using the Php-Parser library [26]. For RQ5, we compared Deep RL and black-box fuzzer algorithms to evaluate their effectiveness in exploiting vulnerabilities over time. The experiments consisted of 10 episodes per mobile companion app-IoT firmware pair, each episode running for 20 minutes for each algorithm. Performance was evaluated using the AUC metric, as in previous studies. Each experiment was repeated ten times to account for non-determinism. We applied the Shapiro-Wilk test to assess normality, followed by ANOVA for Gaussian distributions or the Wilcoxon non-parametric test otherwise. We adopted the same effect sizes, classification notations, and rules as described in Section 5.1.

5.4 IoT Device's Vulnerability Exploitation Results

Figure 5 shows the mean number of unique vulnerabilities exploited per firmware, averaged across the experiment repetitions. The figure compares the number of unique exploited vulnerabilities with the vulnerabilities detected during the static analysis phase per firmware (indicated as *# Real Vulnerabilities* in the figure). We verified that these vulnerabilities are true positives by manually analyzing the firmware's code and mapping them to existing CVEs [7–11], which confirm their validity. The Deep RL agent successfully reaches and exploits more vulnerabilities on average than the black-box fuzzer in four of ten firmware samples. Although the black-box fuzzer algorithm typically exploits only one unique vulnerability per firmware, the Deep RL agent manages to exploit different unique vulnerabilities identified during the static analysis phase.

Table 5 shows the mean number of unique IoT APIs reached, AUC values, and the number of successful exploits for SAC and the black-box fuzzer algorithm in different IoT firmware. We calculated the average number of unique APIs and vulnerabilities exploited episode by episode for each firmware and algorithm, averaging the results across repetitions. We used a Wilcoxon test to compare the number of unique APIs reached and calculated the paired Vargha-Delaney effect size. We applied a one-way ANOVA test and calculated Cohen's d effect size for AUC and the number of successful exploits. SAC consistently outperforms the black-box fuzzer and the coverage-based algorithm in all firmware

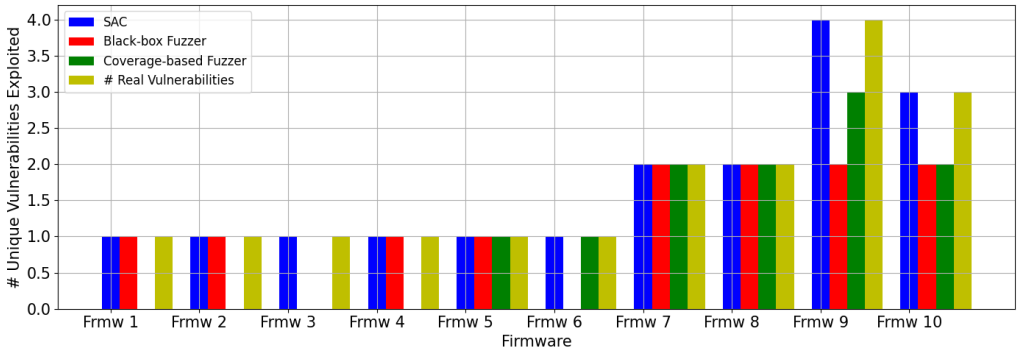

Fig. 5. Mean number of unique vulnerabilities exploited per firmware, averaged across the repetitions

concerning unique APIs discovered, AUC, and successful exploits. Overall, SAC averaged 9.97 unique APIs compared to 6.95 for the black-box fuzzer algorithm and 7.06 for the coverage-based fuzzer. Similarly, SAC successfully exploited an average of 14.95 vulnerabilities, compared to 5.85 for the black-box fuzzer and 7.49 for the coverage-based fuzzer. The AUC for SAC was also significantly higher, with an overall value of 134.39 compared to 52.83 for the black-box fuzzer algorithm and 69.66 for the coverage-based fuzzer. In all cases, large effect sizes (L) confirm SAC's statistical and practical superiority over both the black-box and coverage-based algorithms.

Firmware	Average unique IoT APIs across episodes (SAC)	Average unique IoT APIs across episodes (Rand)	Average unique IoT APIs across episodes (Cov)	Mean Area across episodes (SAC)	Mean Area across episodes (Rand)	Mean Area across episodes (Cov)	Average successful exploit number across episodes (SAC)	Average successful exploit number across episodes (Rand)	Average successful exploit number across episodes (Cov)
Frmw1	9.85 L(Cov,Rand)	6.89	7.05	136.60	49.83	68.40	15.33 M(Rand),L(Cov)	5.46	7.36
Frmw2	9.98 L(Cov,Rand)	7.15	7.08	136.15 L(Cov,Rand)	57.78	73.10	14.93 M(Rand),L(Cov)	6.34	8.03
Frmw3	10.00 L(Cov,Rand)	6.88	7.07	137.33 L(Cov,Rand)	55.23	67.53	15.39 L(Cov,Rand)	6.19	7.30
Frmw4	9.86 L(Cov,Rand)	7.15	7.02	129.38 L(Cov,Rand)	52.83	64.95	14.53 M(Rand),L(Cov)	5.84	7.06
Frmw5	9.96 L(Cov,Rand)	6.86	7.22	143.83 L(Cov,Rand)	55.38	73.40	15.86 L(Cov,Rand)	6.29	7.88
Frmw6	10.36 L(Cov,Rand)	6.83	7.25 S(Rand)	148.83 L(Cov,Rand)	48.73	70.30	16.39 L(Cov,Rand)	5.35	7.42
Frmw7	9.98 L(Cov,Rand)	6.92	6.98	134.58 L(Cov,Rand)	57.58	66.03	14.68 L(Cov,Rand)	6.40	7.26
Frmw8	9.89 L(Cov,Rand)	7.21	7.02	117.88 L(Cov,Rand)	48.90	78.80	13.25 M(Rand),L(Cov)	5.46	8.21
Frmw9	9.99 L(Cov,Rand)	6.72	6.93	129.33 L(Cov,Rand)	52.30	73.70	14.67 M(Rand)	5.68	7.88 L(Rand)
Frmw10	9.86 L(Cov,Rand)	6.88	6.99	129.98 L(Cov,Rand)	49.73	60.38	14.47 L(Cov,Rand)	5.52	6.50
Overall	9.97 L(Cov,Rand)	6.95	7.06	134.39 L(Cov,Rand)	52.83	69.66	14.95 L(Cov,Rand)	5.85	7.49

Table 5. Mean number of unique sink APIs on IoT devices and AUC across episodes for each algorithm. Effect sizes between the SAC algorithm and the best fuzzer (the type of fuzzer is in brackets) are reported only when the p-value is statistically significant (S = Small; M = Medium; L = Large)

Table 6 shows the mean number of unique inputs that reach RCE vulnerabilities of the IoT device and the AUC across episodes for the SAC and the black-box fuzzer algorithms. We calculated each firmware's average number of unique inputs episode by episode, averaging the results across experiment repetitions. We used a paired Wilcoxon test with the paired Vargha-Delaney effect size to compare the number of unique inputs generated by each algorithm. For AUC, we applied a one-way analysis of variance (ANOVA) test and calculated Cohen's d effect size. SAC averages 8.03 unique inputs, compared to 6.49 for the black-box fuzzer algorithm and 5.81 for the coverage-based algorithm, with large effect sizes (L) observed across all firmware. This confirms that SAC consistently discovers more unique inputs capable

of triggering RCE vulnerabilities. In terms of AUC, SAC also shows a clear advantage. For example, SAC achieves an overall AUC of 116.92 compared to 50.10 for the black-box fuzzer algorithm and to 65.44 for the coverage-based algorithm, with large effect sizes (L) further supporting the practical significance of the superiority of SAC.

Firmware	Average unique inputs across episodes (SAC)	Average unique inputs across episodes (Cov)	Average unique inputs across episodes (Rand)	Mean Area across episodes (SAC)	Mean Area across episodes (Cov)	Mean Area across episodes (Rand)
Frmw 1	6.69 L(Rand)	4.98	6.17 L(Cov)	111.53 L(Rand, Cov)	46.02	63.52
Frmw 2	6.81 M(Rand),S(Cov)	5.20	6.08 M(Cov)	110.50 M(Cov)	53.40	66.25
Frmw 3	6.81 L(Rand),M(Cov)	5.17	6.06 M(Cov)	114.10 M(Rand),L(Cov)	52.08	61.33
Frmw 4	6.88 L(Rand),S(Cov)	5.17	6.10 M(Cov)	109.60 M(Rand),L(Cov)	49.10	59.90
Frmw 5	6.95 L(Rand),S(Cov)	5.04	6.26 M(Cov)	120.30 M(Rand),L(Cov)	51.75	66.65
Frmw 6	9.54 L(Rand,Cov)	6.58	7.08 S(Cov)	125.25 L(Rand,Cov)	47.27	65.90
Frmw 7	8.85 L(Rand,Cov)	6.30	6.77 S(Cov)	121.92 M(Rand,Cov)	54.48	64.80
Frmw 8	8.85 L(Rand,Cov)	6.44	6.70	109.70 M(Cov)	45.83	75.28
Frmw 9	9.73 L(Rand,Cov)	6.56	6.87	125.92 M(Rand),L(Cov)	51.90	71.95
Frmw 10	9.25 L(Rand,Cov)	6.60	6.83	120.38 L(Rand,Cov)	49.17	58.85
Overall	8.03 L(Rand,Cov)	5.81	6.49	116.92 L(Rand,Cov)	50.10	65.44 L(Cov)

Table 6. Mean number of unique inputs reaching IoT device's RCE vulnerabilities and AUC across episodes for each algorithm. Effect sizes between the SAC algorithm and the best fuzzer (the type of fuzzer is in brackets) are reported only when the p-value is statistically significant (S = Small; M = Medium; L = Large)

Figure 6 presents the mean cumulative number of unique IoT APIs reached and the successfully exploited vulnerabilities across episodes averaged over the selected firmware samples. The left subfigure shows the cumulative number of unique IoT APIs triggered, with SAC consistently reaching more APIs than the black-box fuzzer and the coverage-based algorithms. Despite this difference, the growth trend remains similar for all the algorithms. The right subfigure illustrates the mean cumulative number of exploited vulnerabilities. SAC demonstrates a significantly higher and faster exploitation rate than the black-box fuzzer and the coverage-based algorithms.

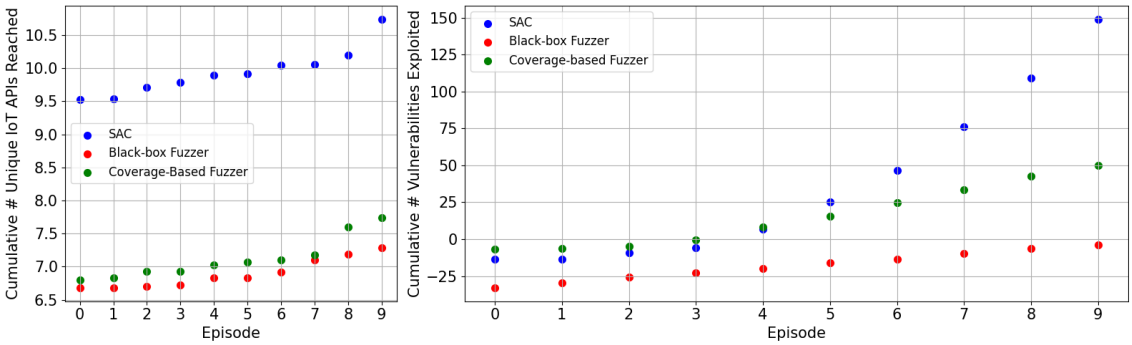

Fig. 6. i) Mean cumulative number of unique IoT APIs reached ii) Mean number of vulnerabilities exploited over the episodes

RQ4: The results show that the SAC algorithm achieves high coverage in exploiting vulnerabilities identified during static analysis, outperforming the black-box fuzzer in 4 cases and the coverage-based algorithm in 6 cases exploiting the same vulnerabilities in the others.

RQ5: The SAC algorithm consistently identifies more unique IoT APIs and achieves higher coverage of the API set than the black-box fuzzer and the coverage-based algorithm. Additionally, SAC exploits more vulnerabilities over episodes, proving more effective overall.

5.5 MITHRAS's Execution Statistics

A complete test generation session lasts 100 minutes. Each episode increases CPU utilization by 12% and consumes 3.83 W of energy. Execution time and energy consumption metrics were recorded for each 10-minute episode. At idle, the Raspberry Pi 5 exhibited an average CPU utilization of 0.18% (sampled every 5 seconds over 10 minutes), an energy consumption of 3.01 W, and a RAM usage of 0.24 GB. When running the original IoT device software, average CPU utilization raised to 12%, energy consumption to 3.83 W, and RAM usage to 0.28 GB. With the instrumented IoT device software, CPU utilization increased to an average of 15%, energy consumption to 3.96 W, and RAM usage to 0.29 GB. Regarding execution timing, the Layout agent triggers a new action on the app's layout approximately every 25 seconds. This delay occurs because the Layout agent pauses upon reaching a sink function, awaiting the reward computation from the Payload agent. After injecting a maliciously mutated payload into the smartphone, the Payload agent waits 10 seconds before processing the execution trace received from the IoT device.

6 DISCUSSION & LIMITATIONS

The experimental results demonstrate that MITHRAS outperforms black-box random strategies concerning the coverage of the communication function within the companion app, the IoT APIs reached, and successful exploit attempts. We manually inspected the firmware's code to verify which vulnerabilities Emba detected were true positives. We used payloads generated by the `Payload Agent` to confirm that the agent exploited the vulnerabilities discovered by Emba. Based on our manual code inspection and analysis of Emba's results, we mapped the identified vulnerabilities to public CVEs [7–11] that affect the firmware selected for the testing campaign, confirming the validity of these vulnerabilities. MITHRAS's methodology can be extended to support additional vulnerabilities. While the current implementation focuses solely on exploiting RCE vulnerabilities, it can be adapted for other types. To extend support for a new vulnerability, the user must modify the `Php AST Static Instrumenter` to inject the necessary code for obtaining runtime feedback during exploitation. If the vulnerability involves function calls within the PHP code, adaptation is straightforward, as MITHRAS already handles function calls that execute system commands on the IoT device. However, additional modifications may be required for other types of vulnerabilities, both to identify the vulnerable code and inject instrumentation able to log its output. Finally, the Payload agent's action space must be modified to support actions that trigger the new vulnerability, requiring updates to its logic accordingly. Our approach has shown promise, but it also presents limitations. For the companion app on the smartphone, we use the Frida tool to instrument the sink functions identified during static analysis and trigger the `Payload Agent`. However, if the app employs anti-Frida mechanisms, as described in [36], neither the RL nor the black-box algorithms can inject payload mutations. To overcome this, we could modify the app to bypass these checks or explore alternative solutions such as VirtualApp [28], which allows the app to run in a container app installed on the smartphone, enabling the instrumentation of sink functions with a lower risk of detection. MITHRAS's static analysis on the IoT device is currently limited to `php` code, which means it does not collect coverage for other programming languages or binaries. This limits the runtime feedback and the range of vulnerabilities (such as RCE) the tool can target. A future enhancement would be to extend the analysis to cover other languages and binary files.

7 CONCLUSIONS

This paper presents MITHRAS, a novel security testing approach that maliciously uses companion mobile apps to mutate legitimate data sent to IoT devices. By combining static and dynamic analysis, MITHRAS automatically identifies the app's code for communication with the IoT device. It employs Deep RL techniques to automate the app's navigation and mutate legitimate data delivered to the IoT device. Additionally, it injects code into the IoT device's programs to collect runtime feedback, enabling the Deep RL agents to compute rewards at each step. Experimental results show that the Deep RL approach significantly outperforms black-box random exploration regarding IoT API coverage and the number of successfully exploited vulnerabilities.

8 DATA AVAILABILITY

We make our replication package publicly available to support the reproducibility of our findings: https://github.com/X3no21/Mithras.

REFERENCES

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)*. IEEE, 1362–1380.

[2] appium.io. Accessed in February 26, 2025. Appium. https://appium.io/docs/en/latest/.

[3] arm.com. Accessed in February 26, 2025. Arm. https://www.arm.com/.

[4] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware.. In *NDSS*, Vol. 1. 1–1.

[5] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[6] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A {Large-scale} analysis of the security of embedded firmwares. In *23rd USENIX security symposium (USENIX Security 14)*. 95–110.

[7] cve.mitre.org. Accessed in February 26, 2025. CVE-2018-19986. https://www.cve.org/CVERecord?id=CVE-2018-19986.

[8] cve.mitre.org. Accessed in February 26, 2025. CVE-2018-19987. https://www.cve.org/CVERecord?id=CVE-2018-19987.

[9] cve.mitre.org. Accessed in February 26, 2025. CVE-2018-19988. https://www.cve.org/CVERecord?id=CVE-2018-19988.

[10] cve.mitre.org. Accessed in February 26, 2025. CVE-2018-19989. https://www.cve.org/CVERecord?id=CVE-2018-19989.

[11] cve.mitre.org. Accessed in February 26, 2025. CVE-2018-19990. https://www.cve.org/CVERecord?id=CVE-2018-19990.

[12] cve.mitre.org. Accessed in February 26, 2025. CVE Routers. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=router.

[13] cwe.mitre.org. Accessed in February 26, 2025. CWE-94: Improper Control of Generation of Code ('Code Injection'). https://cwe.mitre.org/data/definitions/94.html.

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. *SIGPLAN Not.* 53, 2 (mar 2018), 392–404. https://doi.org/10.1145/3296957.3177157

[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. 463–478.

[16] developer.android.com. Accessed in February 26, 2025. Android Developer. https://developer.android.com/studio/run/emulator?hl=en.

[17] developer.android.com. Accessed in February 26, 2025. Intent. https://developer.android.com/reference/android/content/Intent.

[18] dlink.com. Accessed in February 26, 2025. Dlink. https://www.dlink.com/uk/en.

[19] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. 2022. AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation. *Computers & Security* 122 (2022), 102889.

[20] explodingtopics.com. Accessed in February 26, 2025. Number of IoT Devices (2024). https://explodingtopics.com/blog/number-of-iot-devices.

[21] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*. 337–350.

[22] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[23] frida.re. Accessed in February 26, 2025. Frida. https://frida.re.

[24] Scott Fujimoto, Herke Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*. PMLR, 1587–1596.

[25] github.com. Accessed in February 26, 2025. Emba. https://github.com/e-m-b-a/emba.

[26] github.com. Accessed in February 26, 2025. PHP-Parser. https://github.com/nikic/PHP-Parser.

[27] github.com. Accessed in February 26, 2025. Stable Baseline. https://github.com/DLR-RM/stable-baselines3.

[28] github.com. Accessed in February 26, 2025. VirtualApp. https://github.com/asLody/VirtualApp/tree/master.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[30] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 733–745.

[31] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. 2019. All things considered: An analysis of {IoT} devices on home networks. In *28th USENIX security symposium (USENIX Security 19)*. 1169–1185.

[32] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang, X. Wang, and G. Gao. 2024. LABRADOR: Response Guided Directed Fuzzing for Black-box IoT Devices. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 130–130. https://doi.org/10.1109/SP54263.2024.00127

[33] Volodymyr Mnih. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv:1602.01783* (2016).

[34] Afshin Oroojlooy and Davood Hajinezhad. 2023. A review of cooperative multi-agent deep reinforcement learning. *Applied Intelligence* 53, 11 (2023), 13677–13722.

[35] play.google.com. Accessed in February 26, 2025. Dlink Wifi App. https://play.google.com/store/apps/details?id=com.dlink.dlinkwifi&hl=en.

[36] preemptive.com. Accessed in February 26, 2025. Detect Frida for Android. https://darvincitech.wordpress.com/2019/12/23/detect-frida-for-android/.

[37] raspberrypi.com. Accessed in February 26, 2025. Raspberry Pi 5. https://www.raspberrypi.com/products/raspberry-pi-5/.

[38] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In *2021 IEEE Symposium on Security and Privacy (SP)*. 484–500. https://doi.org/10.1109/SP40001.2021.00066

[39] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1544–1561.

[40] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–29.

[41] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2023. Assessing the security of inter-app communications in android through reinforcement learning. *Computers & Security* 131 (2023), 103311. https://doi.org/10.1016/j.cose.2023.103311

[42] John Schulman. 2015. Trust Region Policy Optimization. *arXiv preprint arXiv:1502.05477* (2015).

[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[44] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*, Vol. 1. 1–1.

[45] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. In *International conference on machine learning*. Pmlr, 387–395.

[46] threatpost.com. Accessed in February 26, 2025. Travel Routers, NAS Devices Among Easily Hacked IoT Devices. https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/.

[47] Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards. (1989).

[48] yoctoproject.org. Accessed in February 26, 2025. Yocto. https://www.yoctoproject.org/.

[49] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng