The Journal of Systems and Software 239 (2026) 112918

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Multi-agent deep reinforcement learning for penetration testing of IoT devices through their mobile companion app☆

Francesco Pagano [a,*], Mariano Ceccato [a], Alessio Merlo [b], Paolo Tonella [c]

[a] University of Verona, Verona, Italy
[b] CASD – University School of Advanced Defense Studies, Rome, Italy
[c] Università della Svizzera italiana, Lugano, Switzerland

ARTICLE INFO

Dataset link: https://github.com/X3no21/Mithras

Keywords:
Penetration testing
IoT systems
Test generation

ABSTRACT

The increasing integration of IoT devices into critical infrastructures has made them prime targets for cyber-attacks. Many of these devices rely on outdated or legacy software, which introduces inherent vulnerabilities and complicates firmware updates, making the identification and testing of these weaknesses essential. Existing methods, such as Diane and IoTFuzzer, typically employ black-box approaches, mutating network requests generated during device operation to craft potential attack vectors. However, black-box methods recognize successful exploits based on external feedback signals, such as errors or crashes, which makes them ineffective when the effect of an exploit is not an error and the feedback signal requires execution trace analysis. To address these limitations, we introduce MITHRAS, the first gray-box approach that uses mobile companion apps to deliver maliciously mutated requests directly to IoT devices, under the guidance of the distance between the execution trace and a potential vulnerability sink. MITHRAS uses Deep Reinforcement Learning to efficiently navigate the communication code within companion apps, dynamically mutating request payloads before transmission. Adapting to past attack outcomes, MITHRAS dynamically refines its strategy, mimicking human decision-making to improve exploit generation effectiveness.

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 (explodingtopics.com, 2023). Kumar et al. (2019) studied IoT devices in people's homes, showing how they spread worldwide and which companies produce the most of them.

The rapid growth of IoT has raised security concerns, as these devices often expose numerous vulnerabilities, as evidenced 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 (cve.mitre.org, 2023f), underscoring the need for reliable methodologies for discovering and fixing vulnerabilities. Recent studies (Alrawi et al., 2019; threatpost.com, 2023) 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 have focused on identifying security issues in IoT devices but suffer from several limitations. Static analysis (Redini et al., 2020; David et al., 2018) often results in false positives, while black-box dynamic analysis (Feng et al., 2021; Liu et al., 2024) struggles with encrypted or proprietary protocols. Moreover, many existing approaches rely on directly probing the device endpoints, which is, of course, simpler but might be infeasible as it ignores the device state. In fact, many devices employ end-to-end encryption and, more importantly, require complex state-dependent interactions. Many API requests (e.g., to read or modify settings) can only be executed after the device is in a specific state. Bringing the device into such a state manually or via random fuzzing is highly inefficient. However, the companion app already implements the precise logic to do so. A prototypical example is authentication, which is a prerequisite for most sensitive requests. This process often uses a challenge–response mechanism: the IoT device contains the "verifier" code, but the companion app contains the essential "prover" code. This forces security testers to use the companion app, as it is the only component that can successfully pass the authentication challenge and generate the necessary access tokens.

To address the challenge of encryption and authentication, state-of-the-art solutions like IoTFuzzer (Chen et al., 2018) and Diane (Redini

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail address: francesco.pagano@univr.it (F. Pagano).

https://doi.org/10.1016/j.jss.2026.112918
Received 25 November 2025; Received in revised form 23 March 2026; Accepted 23 April 2026
Available online 28 April 2026

et al., 2021) successfully leverage mobile companion apps. Operating in a strict black-box setting against physical hardware, they modify the request content to trigger memory-corruption crashes on the IoT device. These tools are highly effective for their specific operational goal: discovering memory safety issues on locked commercial hardware without requiring firmware extraction. However, relying primarily on network-level anomalies (e.g., timeouts or connection resets) limits their applicability to logical vulnerabilities, such as Remote Code Execution (RCE). A successful RCE exploit often leaves the device fully operational and returns a valid '200 OK' HTTP response, which a crash-fuzzer legitimately interprets as a benign interaction.

To tackle the generation of logical exploits, we propose MITHRAS, a multi-agent Deep Reinforcement Learning (Deep RL) approach designed for a different, yet equally critical, operational setting: a white-box or gray-box scenario. In enterprise environments, such as vendor testing labs, certification authorities, or red-teaming exercises, security analysts typically have legitimate access to the device's compiled firmware. By assuming firmware availability, MITHRAS operates within a full emulation framework (Kim et al., 2020; Chen et al., 2016), enabling static instrumentation of the firmware alongside dynamic monitoring of the companion app. This setup allows the multi-agent paradigm to decouple two complex tasks: (1) stateful navigation, managed by the `Layout` agent to reach app procedures communicating with the IoT device; and (2) payload generation, handled by the `Payload` agent, which maliciously mutates parameters to exploit the device.

While black-box tools prioritize broad deployment on physical devices, MITHRAS leverages its gray-box setting to replace coarse network-level responses with fine-grained runtime execution traces. This rich feedback serves as a precise reward signal to guide smart payload mutations and automatically pinpoint vulnerable code paths. Consequently, MITHRAS is explicitly driven by successful code execution rather than system crashes, automating the generation of functional Proofs of Concept (PoCs) for logical vulnerabilities.

MITHRAS is the first solution to apply Deep RL techniques to generate functional exploits against IoT devices by actively bridging the mobile companion app and the instrumented firmware. By dynamically crafting and refining malicious payloads based on precise execution feedback, MITHRAS demonstrates how full firmware access can be leveraged to automatically trigger and verify complex security vulnerabilities.

In summary, we make the following contributions:

- We propose a novel multi-agent Deep RL architecture that relies 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 pairs of companion software applications to evaluate their efficacy in triggering vulnerabilities.

2. Related work

IoT devices are vulnerable 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 Costin et al. (2014), Davidson et al. (2013), Shoshitaishvili et al. (2015), David et al. (2018), and Redini et al. (2020). However, these solutions face the challenge of identifying vulnerabilities that often result in false positives, leading to unreliable analyses, and cannot confirm whether a vulnerability is effectively exploitable.

To address limitations in static analysis, some solutions use dynamic analysis. Feng et al. (2021) trigger exposed APIs to infer request structures and mutate values; however, this black-box approach struggles with custom encodings as it lacks knowledge of the code configuration. Zheng et al. (2019) improved whole-system emulation performance by combining system and user emulation, using Afl (Fioraldi et al., 2020) to mutate system call parameters. Nevertheless, this requires extensive environment customization and focuses solely on system calls, rather than vulnerabilities in the executed code.

Gray-box solutions aim for higher precision. Du et al. (2022) introduced AflIot, which runs Linux-based IoT binaries in an emulated environment without specialized hardware peripherals. Its instrumentation provides feedback to Afl to discover crash-inducing inputs, though shared-memory constraints limit it to one binary at a time. Ronin (Romdhana et al., 2023) uses Deep RL to navigate mobile app GUIs and mutate Intent (developer.android.com, 2023b) parameters to exploit Inter-Component Communication (ICC) vulnerabilities. Ronin also statically instruments the app's code to inject logic that collects real-time app coverage. Similarly, Romdhana et al. (2022) developed ARES, which uses Deep RL to maximize code coverage by mimicking user interactions and executing action sequences designed to crash the app.

The most relevant approaches are app-based fuzzers, such as IoTFuzzer (Chen et al., 2018) and Diane (Redini et al., 2021). As discussed, these tools pioneered the use of companion apps to bypass encryption. However, their design objective is to serve as crash fuzzers targeting memory corruption. Their feedback is limited to network-level responses (e.g., timeouts), which is sufficient for detecting crashes but not for logical exploits. This makes them unsuited for the RCE vulnerabilities MITHRAS targets, where a successful exploit often returns a valid '200 OK'. Furthermore, their stateless approach cannot model the multi-step sequences needed for complex stateful interactions, such as authorization.

Regarding multi-agent architecture, MITHRAS differs from the cooperative paradigms classified by Oroojlooy and Hajinezhad (2023). In *Independent Q-Learning*, agents learn separate Q-functions entirely independently, whereas MITHRAS agents are interdependent. In the *Fully Observable Critic* paradigm, agents share global observations to address non-stationarity, differing from MITHRAS, where agents rely on individual observations. The *Consensus* paradigm requires agents to communicate to reach agreement on solutions, whereas MITHRAS's strict hierarchical dependency model does not. The *Value Function Factorization* paradigm combines Q-functions for global rewards; in contrast, MITHRAS agents learn separate Q-functions with conditional dependencies. Finally, the *Learn to Communicate* paradigm enables agents to dynamically determine what to share, whereas MITHRAS agents interact through a fixed structure in which the master's decisions directly influence the slave.

3. Background

3.1. Deep RL

The purpose 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 state of the environment 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 determined by a policy function π, which specifies the actions to take in the current state s_t. A policy can be:

- **Deterministic**: $\pi(s_t)$, that is, 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 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 (Haarnoja et al., 2018) (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 policy's entropy, which 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 combines the expected return and an 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 (Schulman, 2015) (TRPO) is another prominent Deep RL algorithm, known for its stability and efficiency in policy updates. Unlike some other Deep RL algorithms, which 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, helping maintain stability and preventing 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 enable 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, controlled policy updates, thereby avoiding shifts that could lead to divergence or suboptimal performance. This is crucial due to the complexity of the environment, which involves both a smartphone companion app and an IoT device. Given these constraints, we prioritized highly stable algorithms and excluded Q-Learning (Watkins, 1989) from consideration. We selected SAC for its diverse actions that improve exploration and coverage, and TRPO for reusing similar actions to maximize the reachability of promising paths. SAC, with its off-policy approach, outperforms A2C (Mnih, 2016), DDPG (Silver et al., 2014), and TD3 (Fujimoto et al., 2018) in terms of stability and exploration. As demonstrated by Romdhana et al. (2022), SAC achieves higher coverage than DDPG due to its entropy regularization, which prevents premature convergence. TRPO, compared to PPO (Schulman et al., 2017), offers stricter policy control, making it more stable in sensitive environments. TRPO also ensures more stable on-policy updates than TD3, which can be unstable due to its off-policy nature.

3.2. RCE vulnerabilities

RCE vulnerabilities are among the most critical security issues in software systems, allowing attackers to execute arbitrary code on a remote machine. Their importance is underscored by their classification in cwe.mitre.org (2023), which highlights the inherent risk of executing 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 input is used directly to construct system commands or scripts without rigorous validation, it 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 properly sanitized, allowing arbitrary commands to be executed on the system without any control.

4. Methodology

This section describes MITHRAS, our approach to automated penetration testing of IoT devices using 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.

4.1. Setup phase

Fig. 1 provides an overview of the setup phase of MITHRAS, which is divided into two parts: (i) static analysis of the companion app of the IoT device to identify methods that handle communication between the smartphone and the IoT device, and (ii) static analysis of the firmware of the IoT device to locate command execution functions and instrument the code to generate runtime feedback from actions taken by the Deep RL agent.

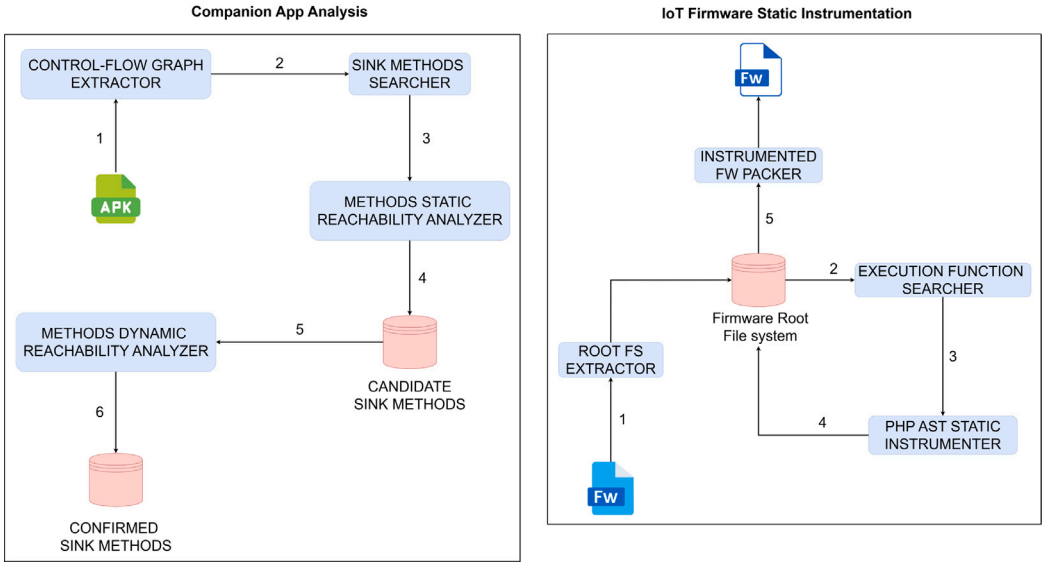

Fig. 1. Setup Phase.

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 (CFG) (Step 1). Once the graph is established, the `Sink Methods Searcher` module identifies all methods that handle 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). 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 further filters methods, 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 after executing a candidate sink method, the method is deemed valid. After both static and dynamic reachability analyses are completed, 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 firmware's software (Step 1). After extracting the root file system, the `Execution Function Searcher` module scans the codebase. This module searches for standard execution functions inherent to the scripting language (e.g., `system()`, `exec()`, `passthru()` in PHP) that could potentially lead to RCE if provided with unsanitized input (Step 3). The goal here is not to definitively flag a vulnerability, but to identify all possible execution sinks.

Once these target functions are mapped, the `Php AST Static Instru-menter` module traverses each `.php` file to automatically instrument the code (Step 4). This module modifies the original Abstract Syntax Tree (AST) by inserting logging mechanisms just before and after the identified execution functions. This instrumentation captures critical runtime feedback for the Deep RL agent during the dynamic phase: specifically, it logs every function or method call made during the script's execution (code coverage) and captures the standard output of any commands issued through the execution functions. This data is then transmitted via a socket to the Deep RL agent. Finally, the `Instrumented FW Packer` module repacks the modified file system and the original firmware files into a new `.bin` file (Step 5), producing the final emulatable package.

4.2. Dynamic analysis phase

Fig. 2 shows an overview of the' dynamic analysis phase of MITHRAS. This phase involves two Deep RL agents that interact with the companion app and the IoT device. In the following, we will discuss each step of the methodology and detail the two Deep RL agents.

4.2.1. Overview

The `Layout Agent` begins by setting its state to the companion app's current state 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 in 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 on 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:

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

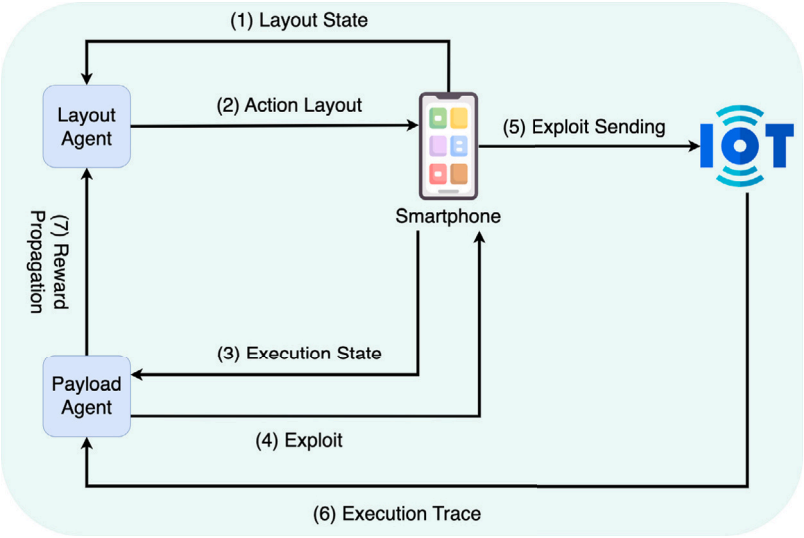

Fig. 2. Dynamic Analysis Phase.

Where r_l represents the reward obtained by the `Layout Agent` based on its interaction with the companion app's Graphical User Interface (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 achieving sink methods in the mobile companion app, we need to translate the problem into the standard RL framework: an MDP, defined by the 5-tuple $\langle S, A, R, P, \rho_0 \rangle$. Here, S represents the set of possible GUI states, A is the set of interactions (clicks, scrolls), R is the reward function guiding the agent towards sinks, P is the transition probability function $P(s_{t+1}|s_t, a_t)$ describing the app's response to inputs, and ρ_0 is the initial state distribution (launching the app). Furthermore, the testing problem must be structured as an RL task with 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 represents the *Activity Context*: $(a_0, \ldots, a_n)$ uses a one-hot encoding where a_i is set to 1 only if the currently active activity is the ith one (mapped from the static analysis), and 0 otherwise. The second component represents the *Widget Availability*: w_j is set to 1 if the jth widget (identified by its unique resource ID or XPath) is available and interactive 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 MDP's action set A. MITHRAS identifies executable events by analyzing the attributes of dumped widgets (clickable, long-clickable, scrollable). To accommodate the high-dimensional nature of UI interactions, we formulate the action space as a multi-discrete space. Each action a has three parts: the first identifies the target widget index or system action; the second provides a string input index (pointing to a predefined fuzzing dictionary); and the third is context-dependent, determining either the action type (click, 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 $\Gamma_1 = 100$, $\Gamma_2 = 70$, $\Gamma_3 = 50$, $\Gamma_4 = 80$).

We define the distance metric $d_t(s^*)$ based on the *Activity Transition Graph* (ATG) constructed during the Setup Phase. Let $G_{app} = (V, E)$ be the graph where vertices V represent activities and edges E represent valid transitions (e.g., button clicks). The term $d_t(s^*)$ is defined as the length of the shortest path (number of edges) from the node representing the current activity v_{curr} to the node v_{target} containing the target sink s^*. If $v_{curr} = v_{target}$, the distance is refined to the depth of the widget triggering the sink within the current View Hierarchy. This hierarchical distance calculation allows the agent to receive dense rewards even before reaching the exact sink.

4.4. Payload agent

4.4.1. State representation

The state representation of the IoT device is entirely based on the software running on it. The state $s_t \in S$ is represented as $(p_0, \ldots, p_n)$, a bit vector encoding indicating the `.php` files traversed during the last execution. Clearly, p_i is set to 1 if the corresponding `.php` file was executed and 0 if it was not. This sparse representation allows the agent to correlate specific input mutations with the activation of new code paths on the backend.

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

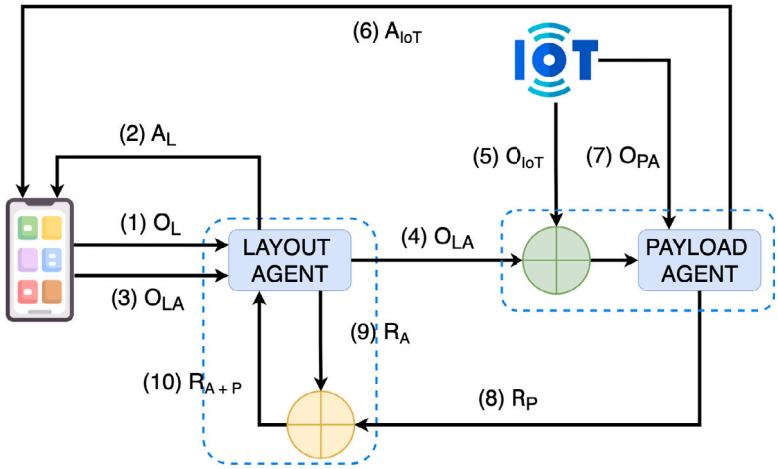

Fig. 3. Deep RL Agent interaction.

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)`); 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 them in a format acceptable to the IoT device can increase the likelihood that the modified payload is accepted 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:

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

where $d_t(v^*)$ indicates the distance to the closest execution function v^* achieved at time t (resp. $t-1$), with $\Gamma_1 > \Gamma_3 > \Gamma_2$ (in our implementation $\Gamma_1 = 100$, $\Gamma_2 = 50$, $\Gamma_3 = 40$).

The distance metric $d_t(v^*)$ is computed on the CFG of the target PHP script, which is extracted during the Setup Phase. Let $G_{php} = (B, T)$ where B represents the set of basic blocks and T represents control flow transitions. The term $d_t(v^*)$ represents the shortest path distance (minimum number of transitions) from the basic block b_{curr} currently executed by the payload to the basic block b_{vuln} containing the identified execution function (e.g., a call to `system()` or `exec()`). This distance is computed dynamically by mapping the runtime execution trace received from the instrumented firmware back to the static CFG.

4.5. Multi agent

Fig. 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 depend in part 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, the Payload Agent is triggered. The observation the Payload Agent receives is the combination of the IoT device's state, the last sink function triggered in the companion app, and the sink function's parameter values. This last observation depends on the action the Layout Agent executes.

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 the Payload Agent maliciously mutates. 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 to produce 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 Algorithm 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 runtime, as well as the code to execute before the original method. 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 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

Algorithm 1 RL Agents

Input: *companion_app_apk*, *device_firmware*, *agent_file*, *episodes*, *steps*, *sink_methods*
Output:

```
1: payload_manager ← loadPayloadManager(agent_file)
2: for episodes do
3:     companion_app.reset()
4:     iot_device.reset()
5:     for steps do
6:         app_state ← companion_app.getState()
7:         w_idx, i_idx, o_idx ← agent_layout.getAction(app_state)
8:         agent_layout.performAction(w_idx, i_idx, o_idx)
9:
10:        while no Timeout do
11:            wait for a send message from the companion app instrumenter
12:        end while
13:        if Timeout then
14:            reward ← agent_layout.computeReward()
15:            agent_layout.updatePolicy(reward)
16:            continue
17:        end if
18:        agent_layout.sendSinkParameterValuesToPayloadAgent(sink_method_hit)
19:        iot_device_state ← iot_device.getState()
20:        param_idx, operator_idx, operation_idx, additional_operator_idx ← agent_payload.getAction(iot_device_state)
21:        parameter_value ← agent_payload.performAction(seedQueue, param_idx, seed_idx, operation_idx, additional_operator_idx)
22:
23:        send parameter_value to the payload_manager
24:        while no Timeout do
25:            wait for the reply from the IoT device
26:        end while
27:        if Timeout then
28:            reward ← agent_payload.computeReward()
29:            agent_payload.updatePolicy(reward)
30:            continue
31:        end if
32:
33:        iot_execution_trace ← getIoTProgramExecutionTrace()
34:        reward_payload ← agent_payload.computeReward(iot_execution_trace)
35:        agent_payload.updatePolicy(reward_payload)
36:        reward_layout ← agent_layout.computeReward(sink_methods)
37:        reward ← reward_layout + α * reward_payload
38:        agent_layout.updatePolicy(reward)
39:    end for
40: end for
```

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 invoked during the app's execution, the `Layout Agent` wakes the `Payload Agent` and sends it the sink function's parameter values (line 18). Upon activation, the `Payload Agent` retrieves the IoT device's current state (line 19), then updates the sink function parameter based on both the device's state 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 for its internal timeout to expire (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 the IoT device responds. In that case, the `Payload Agent` retrieves the device's execution trace (line 32), computes a reward based on it, 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).

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 (Kim et al., 2020) to emulate IoT devices from their firmware files. For the smartphone emulator, MITHRAS uses the Android emulator framework (developer.android.com, 2023a) to set up a fully functional emulator, enabling the installation of companion apps to interact with the emulated IoT devices. To implement the Layout and Payload agents, MITHRAS relies on Stable-Baselines3 (github.com, 2023c), a robust reinforcement learning library. During the evaluation phase, we leveraged Emba (github.com, 2023a) solely as an offline oracle to check for known CVEs and establish a reliable ground truth. To actually locate the execution functions and perform the static instrumentation, MITHRAS navigates the Abstract Syntax Tree (AST) of the IoT firmware's `php` files using the Php-Parser library (github.com, 2023b). This module automatically identifies and inserts logging logic around all discovered execution functions. The logging code captures data at these sink functions when invoked by mutated inputs, allowing the RL agent to discover which execution paths are actively vulnerable to RCEs during runtime exploration. Interaction with the companion app's user interface to reach sink functions is supported by the Appium library (appium.io, 2023). The Frida library (frida.re, 2023) is used to monitor the execution of the companion app and to log any sink functions that are triggered. When a sink function is triggered, MITHRAS captures its parameter values and forwards them to the Layout Agent as a `JSON` object.

5. Evaluation

In this section, we present a proof of concept evaluation of the proposed approach, considering the firmware available from the D-Link vendor for IoT routers. We seek to address five research questions, divided into two distinct studies, respectively, on the reachability of the companion app to the sink (Sections 5.2 and 5.3) and on the vulnerability exploitation of the IoT device (Sections 5.4 and 5.5).

5.1. The need for companion app mediation

Before detailing the performance of the individual agents in the following studies, we address the fundamental necessity of the multi-agent architecture via a validation experiment. Specifically, we investigate whether the Companion App (navigated by the Layout Agent) is strictly required for payload delivery, or whether the Payload Agent could operate independently by sending requests directly to the device. To test this, we isolated the exact malicious payloads generated by the Payload Agent, which successfully triggered RCEs when delivered through the app, and attempted to inject them directly into the IoT device's network interface using standard scripts (e.g., cURL), thereby bypassing the companion app and the Layout Agent. The results indicated a 100% failure rate for direct injection. In every instance, the IoT device returned either "Permission Denied" or a connection error. This failure occurs because the analyzed IoT firmwares (see Section 5.5) employ state-dependent security mechanisms, such as challenge–response authentication, session tokens, and timestamp-based replay protection. The companion app implements the specific "Prover" logic required to dynamically satisfy these challenges. By bypassing the app, the payloads lacked the correct context (valid session tokens, fresh nonces), causing the device to reject them immediately. These results empirically confirm that the Layout Agent is functionally essential to maintain a valid connection state (authentication, session liveness) that allows the Payload Agent's mutations to be accepted.

5.2. 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 companion Android apps and the corresponding IoT device programs for the study, specifically focusing on apps that interact with smartwatch devices (see Table 1). To address RQ1, we used the Frida library (frida.re, 2023) for dynamic instrumentation, collecting data on sink functions 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 is more effective at reaching sink functions and generating diverse inputs.

We conducted the experimental campaign across 10 episodes for each mobile companion app, each lasting 20 min, using Deep RL and black-box fuzzer algorithms. We used the Area Under the Curve (AUC) metric to evaluate efficiency, based on sink coverage over time. 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, classifying them as Negligible (N), Small (S), Medium (M), and Large (L) according to the standard thresholds of 0.2, 0.5, and 0.8. Otherwise, we applied the Wilcoxon non-parametric test with the Vargha–Delaney effect size, whose standard thresholds (applied to $2 \cdot |eff\_size - 0.5|$) are: 0.147, 0.33, 0.474.

Table 1
Companion app used in the testing campaign.

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

5.3. Companion app's sink reachability results

The right-hand diagram of Fig. 4 shows the sink function coverage of the companion apps' per algorithm, averaged over 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 at reaching sink functions, achieving over 80% coverage in most cases.

The' left-hand diagram of Fig. 4 shows the cumulative number of unique sink functions reached across episodes, averaged over the selected companion apps. SAC achieved the highest number of unique sink functions across episodes, outperforming TRPO and the black-box fuzzer. 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 in episodes for SAC, TRPO, and the black-box fuzzer algorithm. We calculated the mean number of unique sink functions per episode for each app and algorithm, averaging across repetitions. To assess overall performance, we combined the episode-averaged results and calculated the mean for each algorithm. We applied a one-way ANOVA test for unique sink functions and calculated Cohen's effect size d, 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 the 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 throughout the episodes for SAC, TRPO, and the black-box fuzzer algorithm. We calculated the mean number of unique inputs per episode for each app and algorithm, averaging 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 effect size d was significant; for AUC, we used the Wilcoxon test with the 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).

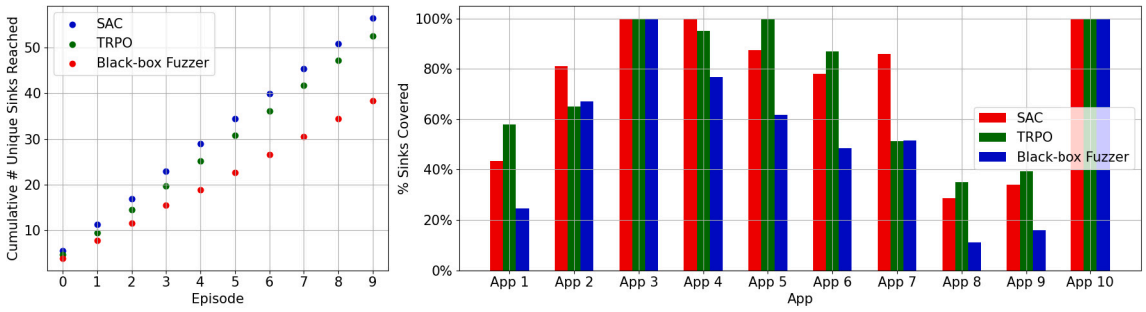

Fig. 4. (i) The mean cumulative number of unique sinks across episodes is averaged across all apps. (ii) The mean percentage of sink functions reached across all apps is averaged over all repetitions.

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).

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 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).

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

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.

5.4. IoT device's vulnerability exploitation

RQ4 [Vulnerability Exploitation]: *How many unique vulnerabilities are successfully discovered and exploited by the Deep RL agent during its dynamic interaction?*

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 accessible through the app and the number of unique maliciously mutated inputs used to exploit RCE vulnerabilities in 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. Instead of learning a policy, it selects the next action to incrementally approach an execution function identified during the static instrumentation phase. Specifically, the algorithm stores payloads that demonstrate progress towards an execution sink and iteratively reapplies actions to these payloads, attempting to reduce the distance to the target function.

We selected firmware from 10 D-Link IoT routers (dlink.com, 2023), as they are publicly available and primarily run `php` code. We focus on routers, as they represent a high-impact target within any IoT ecosystem (Alrawi et al., 2019; threatpost.com, 2023). As a router is the primary network entry point, compromising it not only allows an attacker to control or monitor all network traffic for other connected devices but also serves as a pivot point for launching lateral attacks against other IoT devices on the network. We use the D-Link Wi-Fi app (version 1.4.8) to interact with the firmware during testing. Table 4 lists the firmware versions and the corresponding models.

While our evaluation focuses on D-Link routers, this choice was primarily dictated by the practical constraints of firmware acquisition in the current IoT landscape. Obtaining decryptable and emulatable firmware images from diverse vendors is a significant challenge, as many manufacturers encrypt their firmware or do not provide public downloads. We selected the D-Link dataset specifically because the vendor maintains a comprehensive public repository of unencrypted firmware images, which enabled us to build a reliable, reproducible ground truth for our experiments. Despite this dataset limitation, the MITHRAS methodology is designed to be vendor-agnostic. The underlying techniques, static analysis of standard Linux file systems and emulation via FirmAE/QEMU, are compatible with the wide range of Linux-based embedded architectures (e.g., MIPS, ARM) and web interfaces (CGI/PHP) commonly found across the IoT ecosystem, suggesting that the approach would be applicable beyond the specific devices tested.

Table 4
Firmware used in the testing campaign.

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

We ran the selected firmware on a Raspberry Pi 5 (raspberrypi.com, 2023) to simulate a router, enabling a connection to the companion mobile app on a smartphone. To minimize overhead, we used a lightweight Linux distribution built with the Yocto build system (yoctoproject.org, 2023). The Raspberry Pi ARM (arm.com, 2023) CPU allowed us to run the ARM-based firmware natively. For non-ARM architectures, like MIPS, we used the Firmadyne (Chen et al., 2016) emulation tool. To address RQ4, we instrumented the PHP files within the IoT firmware using the PHP-Parser library (github.com, 2023b), as described in the Methodology section, to intercept the output of any executed command. For RQ5, we compared Deep RL and black-box fuzzer algorithms to evaluate their effectiveness in generating successful exploits over time. The experiments consisted of 10 episodes per mobile companion app-IoT firmware pair, each lasting 20 min 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 5.2.

5.5. IoT device's vulnerability exploitation results

Fig. 5 shows the mean number of unique vulnerabilities exploited per firmware, averaged across the experiment repetitions. To evaluate the effectiveness of our dynamic exploration, we needed a verified ground truth. Therefore, the figure compares the number of unique exploits dynamically discovered by MITHRAS against a set of known vulnerabilities (indicated as *# Real Vulnerabilities* in the figure). To reliably construct this ground truth, we used the Emba tool (github.com, 2023a) solely as an offline oracle to verify whether the firmware contained previously disclosed CVEs (cve.mitre.org, 2023a,b,c,d,e). We then manually verified that the execution paths successfully exploited by our Payload Agent actually mapped to those specific CVEs, confirming their validity as true positives. 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 typically exploits only one unique vulnerability per firmware, the Deep RL agent exploits multiple 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 per episode for each firmware and algorithm, averaging across repetitions. We used

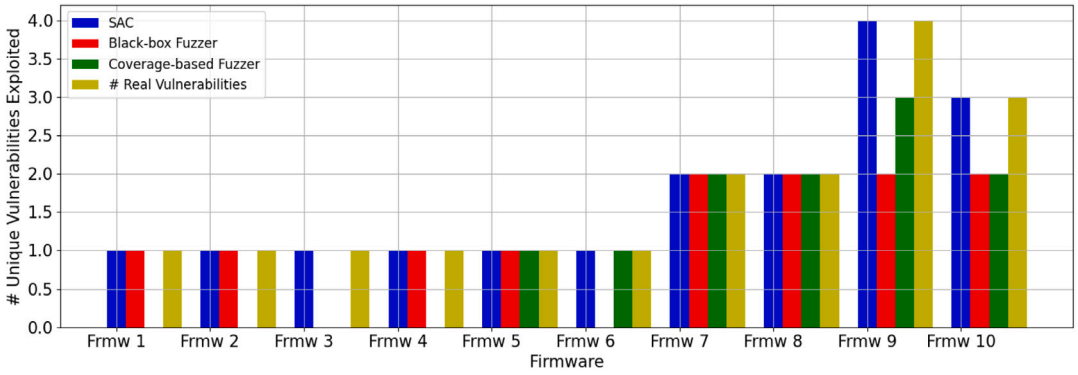

Fig. 5. Mean number of unique vulnerabilities exploited per firmware, averaged across the repetitions.

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).

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

a Wilcoxon test to compare the number of unique APIs reached and calculated the paired Vargha–Delaney effect size. We applied a one-way analysis of variance (ANOVA) and calculated Cohen's effect size (d) for AUC and the number of successful exploits. SAC consistently outperforms the black-box fuzzer and the coverage-based algorithm across all firmwares in terms of unique APIs discovered, AUC, and successful exploits. In general, 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 the statistical and practical superiority of SAC's over both the black-box and coverage-based algorithms.

Table 6 shows the mean number of unique inputs that reach RCE vulnerabilities of the IoT device and the AUC in episodes for the SAC and black-box fuzzer algorithms. We calculated each firmware's average number of unique inputs per episode, averaging 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 used a one-way analysis of variance (ANOVA) and calculated Cohen's d. 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

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).

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)

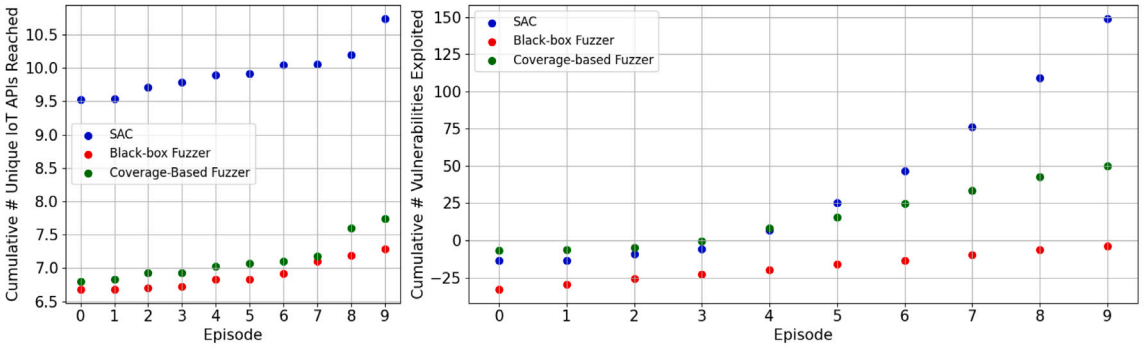

Fig. 6. (i) Mean cumulative number of unique IoT APIs reached (ii) Mean number of vulnerabilities exploited over the episodes.

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 65.44 for the coverage-based algorithm, with large effect sizes (L) further supporting the practical significance of SAC's superiority.

Fig. 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.

RQ4: The results show the SAC algorithm achieves high coverage in successfully exploiting vulnerable execution paths, outperforming the black-box fuzzer in 4 cases and the coverage-based fuzzer in 6 cases.

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.

5.6. MITHRAS's Execution Statistics

A complete test generation session lasts 100 min. 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 s over 10 min), 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 rose 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 s. This delay occurs because the Layout Agent pauses upon reaching a sink function and waits for the Payload Agent to compute the reward. After injecting a maliciously

mutated payload into the smartphone, the Payload Agent waits 10 s before processing the execution trace received from the IoT device.

6. Case study

To demonstrate MITHRAS's capability to handle complex, state-dependent vulnerabilities, we analyze an RCE exploit found in the D-Link DIR-818L firmware (version FW105b01) interacting with the D-Link Wi-Fi companion app (version 1.4.8). This case study highlights why the companion app's mediation is strictly necessary and how the Layout and Payload agents cooperate.

6.1. Authentication and state acquisition

The targeted vulnerability resides in the `SetQoSSettings` API exposed by the IoT device. However, this endpoint is protected and requires a valid administrative session. Direct fuzzing attempts against this endpoint fail because the device enforces a challenge–response authentication mechanism (HNAP1) involving dynamic nonces and timestamps.

In MITHRAS, the Layout Agent navigates the app. When the app detects a missing session token, it triggers the internal authentication routine (identified as function `a` in the obfuscated code: `public static a(...)`). This function executes the multi-step handshake: (1) The app sends a SOAP `Login` request with the `Action` parameter set to "request"; (2) The device responds with a `LoginResult` of "OK", a unique session `Cookie`, and a cryptographic `Challenge` string (e.g., a random nonce); (3) The app computes the MD5 hash of the credentials combined with the received challenge and sends a second `Login` request with the `Action` set to "login" and the computed hash in the `LoginPassword` field. Only after this sequence returns "success" does the device establish a valid session. MITHRAS leverages the app's native implementation of function `a` to transparently bypass this complex authentication, a task that would be computationally infeasible for a stateless black-box fuzzer to reverse-engineer and replicate dynamically.

6.2. Payload injection and exploitation

Once authenticated, the Layout Agent navigates to the QoS configuration activity. This triggers the app's network handler (function `Q` in the code: `public static Q(...)`) to prepare the `SetQoSSettings` request. Here, the Payload Agent intervenes. It intercepts the arguments passed to function `Q` and identifies the target API and its parameters. The agent targets the `uplink` XML tag within the SOAP body.

Instead of the legitimate integer bandwidth value (e.g., "512"), the Payload Agent injects a command substitution payload `$(id>rce4) #`, composed as follows: (1) `$(...)`: Forces the underlying shell to execute the enclosed command; (2) `id>rce4`: Executes the `id` command (to reveal user privileges) and redirects the output to a file named `rce4` in the web server's public directory; (3) `#`: A comment character used to neutralize the remainder of the original shell command on the device, preventing syntax errors.

6.3. Verification

The exploitation is verified via a side-channel check. After sending the mutated request, MITHRAS attempts to download the created file (`http://<router_ip>/HNAP1/rce4`). The file's contents, `uid=0(root)`, confirm that the command was executed successfully with root privileges, validating the RCE.

7. Discussion & limitations

The experimental results demonstrate that MITHRAS outperforms black-box random strategies in companion app communication coverage, IoT APIs reached, and successful vulnerability triggering. Through manual code inspection and payloads generated by the `Payload Agent`, we verified the true positives detected by Emba and mapped the identified vulnerabilities to public CVEs (cve.mitre.org, 2023a,b,c,d,e), confirming their validity.

We excluded direct comparisons against crash-fuzzers like IoTFuzzer (Chen et al., 2018) or Diane (Redini et al., 2021) because their goal is finding memory corruption via *crashes*, whereas MITHRAS targets logical RCEs via *successful code execution*. These disparate objectives make head-to-head comparisons impractical.

Furthermore, it is important to contextualize MITHRAS's preconditions compared to purely black-box tools. A primary limitation is the need for an unencrypted firmware binary. MITHRAS cannot operate against completely opaque targets with locked hardware and encrypted firmware, because the Deep RL reward mechanism requires emulation and static instrumentation to capture execution traces. In white-box or gray-box settings (e.g., vendor labs, red-teaming), firmware access is a standard prerequisite, ensuring analysts possess the explicit legal authorization and intellectual property rights to extract, reverse-engineer, and modify proprietary code. This trades the ease of deployment of black-box fuzzers for the depth required to generate complex logical exploits.

Given this operational dependency, our current evaluation serves primarily as a proof-of-concept validating the multi-agent approach. We focused on a prevalent IoT stack: Linux embedded systems utilizing PHP/CGI web handlers. While the core theoretical framework is technology-agnostic, the instrumentation module is currently tailored to PHP, and the action space targets RCE vulnerabilities (e.g., by monitoring execution functions such as `system()` or `exec()` to reward actions). Extending MITHRAS to other logical vulnerabilities (e.g., SQL injection, path traversal) or ecosystems (e.g., binary-only daemons, custom protocols, alternative languages) is conceptually possible, but it would require dedicated engineering effort. Consequently, our future work will address the current constraints by integrating multi-language instrumentation, dynamic analysis of compiled binaries, and expanded payload dictionaries, extending MITHRAS's generalizability across diverse IoT architectures and vulnerability classes.

When considering the choice between black-box fuzzers and white/gray-box exploit generators like MITHRAS, we believe that executing *both* types of tools in a mixed security testing environment would provide significant value. A tester could employ IoTFuzzer for initial crash detection on a physical device, and subsequently use MITHRAS on the emulated firmware to investigate specific logical flaws or verify the exploitability of statically reported vulnerabilities that do not inherently cause a crash.

For the app's dynamic analysis, we used Frida to monitor sink functions. If the app employs anti-Frida mechanisms (preemptive.com, 2023), neither RL nor black-box algorithms can inject mutations. To overcome this, we could modify the app to bypass these checks or use solutions such as VirtualApp (github.com, 2023d) to run the app in a container, thereby enabling instrumentation with lower detection risk. Regarding resource intensity, despite its complex components, MITHRAS has a manageable footprint. We successfully executed the entire infrastructure (smartphone emulator, firmware emulation, and Deep RL agents) on a 4 GB RAM Raspberry Pi 5, confirming it runs on low-cost edge devices without requiring high-performance computing clusters. Furthermore, to mitigate setup effort for diverse dependencies, we developed a Dockerized framework version that encapsulates the entire pipeline, enabling a streamlined, OS-agnostic deployment process.

8. 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 that communicates 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 in IoT API coverage and the number of vulnerabilities successfully exploited.

CRediT authorship contribution statement

Francesco Pagano: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Formal analysis, Data curation, Conceptualization. **Mariano Ceccato:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Alessio Merlo:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Paolo Tonella:** Writing – review & editing, Validation, Supervision, Methodology, Formal analysis, Conceptualization.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author(s) used Gemini in order to correct the grammar of the written text. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the published article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data and code supporting the findings of this study are publicly available in the GitHub repository: https://github.com/X3no21/Mithras. The repository includes the experimental test data as well as the tool developed and evaluated in this work.

References

Alrawi, Omar, Lever, Chaz, Antonakakis, Manos, Monrose, Fabian, 2019. Sok: Security evaluation of home-based iot deployments. In: 2019 IEEE Symposium on Security and Privacy (Sp). IEEE, pp. 1362–1380.

appium.io, 2023. Appium. https://appium.io/docs/en/latest/. (Accessed 23 March 2026).

arm.com, 2023. Arm. https://www.arm.com/. (Accessed 23 March 2026).

Chen, Jiongyi, Diao, Wenrui, Zhao, Qingchuan, Zuo, Chaoshun, Lin, Zhiqiang, Wang, XiaoFeng, Lau, Wing Cheong, Sun, Menghan, Yang, Ronghai, Zhang, Kehuan, 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing.. In: NDSS.

Chen, Daming D, Woo, Maverick, Brumley, David, Egele, Manuel, 2016. Towards automated dynamic analysis for linux-based embedded firmware.. In: NDSS, vol. 1, p. 1.

Costin, Andrei, Zaddach, Jonas, Francillon, Aurélien, Balzarotti, Davide, 2014. A {large-scale} analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 95–110.

cve.mitre.org, 2023a. CVE-2018–19986. https://www.cve.org/CVERecord?id=CVE-2018-19986. (Accessed 23 March 2026).

cve.mitre.org, 2023b. CVE-2018–19987. https://www.cve.org/CVERecord?id=CVE-2018-19987. (Accessed 23 March 2026).

cve.mitre.org, 2023c. CVE-2018–19988. https://www.cve.org/CVERecord?id=CVE-2018-19988. (Accessed 23 March 2026).

cve.mitre.org, 2023d. CVE-2018–19989. https://www.cve.org/CVERecord?id=CVE-2018-19989. (Accessed 23 March 2026).

cve.mitre.org, 2023e. CVE-2018–19990. https://www.cve.org/CVERecord?id=CVE-2018-19990. (Accessed 23 March 2026).

cve.mitre.org, 2023f. CVE routers. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=router. (Accessed 23 March 2026).

cwe.mitre.org, 2023. CWE-94: Improper control of generation of code ('code injection'). https://cwe.mitre.org/data/definitions/94.html. (Accessed 23 March 2026).

David, Yaniv, Partush, Nimrod, Yahav, Eran, 2018. Firmup: Precise static detection of common vulnerabilities in firmware. SIGPLAN Not. (ISSN: 0362-1340) 53 (2), 392–404. http://dx.doi.org/10.1145/3296957.3177157, https://doi.org/10.1145/3296957.3177157.

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

developer.android.com, 2023a. Android developer. https://developer.android.com/studio/run/emulator?hl=en. (Accessed 23 March 2026).

developer.android.com, 2023b. Intent. https://developer.android.com/reference/android/content/Intent. (Accessed 23 March 2026).

dlink.com, 2023. Dlink. https://www.dlink.com/uk/en. (Accessed 23 March 2026).

Du, Xuechao, Chen, Andong, He, Boyuan, Chen, Hao, Zhang, Fan, Chen, Yan, 2022. AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation. Comput. Secur. 122, 102889.

explodingtopics.com, 2023. Number of IoT devices (2024). https://explodingtopics.com/blog/number-of-iot-devices. (Accessed 23 March 2026).

Feng, Xiaotao, Sun, Ruoxi, Zhu, Xiaogang, Xue, Minhui, Wen, Sheng, Liu, Dongxi, Nepal, Surya, Xiang, Yang, 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. pp. 337–350.

Fioraldi, Andrea, Maier, Dominik, Eißfeldt, Heiko, Heuse, Marc, 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.

frida.re, 2023. Frida. https://frida.re. (Accessed 23 March 2026).

Fujimoto, Scott, Hoof, Herke, Meger, David, 2018. Addressing function approximation error in actor-critic methods. In: International Conference on Machine Learning. PMLR, pp. 1587–1596.

github.com, 2023a. Emba. https://github.com/e-m-b-a/emba. (Accessed 23 March 2026).

github.com, 2023b. PHP-parser. https://github.com/nikic/PHP-Parser. (Accessed 23 March 2026).

github.com, 2023c. Stable baseline. https://github.com/DLR-RM/stable-baselines3. (Accessed 23 March 2026).

github.com, 2023d. VirtualApp. https://github.com/asLody/VirtualApp/tree/master. (Accessed 23 March 2026).

Haarnoja, Tuomas, Zhou, Aurick, Abbeel, Pieter, Levine, Sergey, 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International Conference on Machine Learning. PMLR, pp. 1861–1870.

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

Kumar, Deepak, Shen, Kelly, Case, Benton, Garg, Deepali, Alperovich, Galina, Kuznetsov, Dmitry, Gupta, Rajarshi, Durumeric, Zakir, 2019. All things considered: An analysis of {iot} devices on home networks. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1169–1185.

Liu, H., Gan, S., Zhang, C., Gao, Z., Zhang, H., Wang, X., Gao, G., 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, (ISSN: 2375-1207) p. 130. http://dx.doi.org/10.1109/SP54263.2024.00127, https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00127.

Mnih, Volodymyr, 2016. Asynchronous methods for deep reinforcement learning. arXiv preprint arXiv:1602.01783.

Oroojlooy, Afshin, Hajinezhad, Davood, 2023. A review of cooperative multi-agent deep reinforcement learning. Appl. Intell. 53 (11), 13677–13722.

preemptive.com, 2023. Detect frida for android. https://darvincitech.wordpress.com/2019/12/23/detect-frida-for-android/. (Accessed 23 March 2026).

raspberrypi.com, 2023. Raspberry pi 5. https://www.raspberrypi.com/products/raspberry-pi-5/. (Accessed 23 March 2026).

Redini, Nilo, Continella, Andrea, Das, Dipanjan, De Pasquale, Giulio, Spahn, Noah, Machiry, Aravind, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, 2021. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In: 2021 IEEE Symposium on Security and Privacy. SP, pp. 484–500. http://dx.doi.org/10.1109/SP40001.2021.00066.

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

Romdhana, Andrea, Merlo, Alessio, Ceccato, Mariano, Tonella, Paolo, 2022. Deep reinforcement learning for black-box testing of android apps. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4), 1–29.

Romdhana, Andrea, Merlo, Alessio, Ceccato, Mariano, Tonella, Paolo, 2023. Assessing the security of inter-app communications in android through reinforcement learning. Comput. Secur. (ISSN: 0167-4048) 131, 103311. http://dx.doi.org/10.1016/j.cose.2023.103311, https://www.sciencedirect.com/science/article/pii/S0167404823002213.

Schulman, John, 2015. Trust region policy optimization. arXiv preprint arXiv:1502.05477.

Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, Klimov, Oleg, 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Shoshitaishvili, Yan, Wang, Ruoyu, Hauser, Christophe, Kruegel, Christopher, Vigna, Giovanni, 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS, vol. 1, p. 1.

Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, Riedmiller, Martin, 2014. Deterministic policy gradient algorithms. In: International Conference on Machine Learning. Pmlr, pp. 387–395.

threatpost.com, 2023. Travel routers, NAS devices among easily hacked IoT devices. https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/. (Accessed 23 March 23 2026).

Watkins, Christopher John Cornish Hellaby, 1989. Learning from delayed rewards.

yoctoproject.org, 2023. Yocto. https://www.yoctoproject.org/. (Accessed 23 March 2026).

Zheng, Yaowen, Davanian, Ali, Yin, Heng, Song, Chengyu, Zhu, Hongsong, Sun, Limin, 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, pp. 1099–1114, https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.