# MITHRAS: A Dynamic Analysis Framework for the Mobile-IoT Ecosystem

Francesco Pagano
francesco.pagano@dibris.unige.it
University of Genova
Genova, 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

## ABSTRACT

Firmware re-hosting is crucial when developing methodologies to simulate and execute device-specific firmware, including techniques for firmware testing and security assessments. Although state-of-the-art solutions such as Firmadyne and FirmAE emulate IoT firmware, they cannot simulate communication with external clients and provide limited real-time and security testing support. In this demonstration, we introduce MITHRAS. This novel simulation framework enables static instrumentation and full emulation of IoT device firmware, allowing communication with external clients such as the companion app installed on a smartphone. MITHRAS also supports dynamic instrumentation of the code of the mobile companion app, allowing users to gather detailed information on the app's execution. MITHRAS supports seamless communication between the emulated IoT device and its companion smartphone app, providing a fully integrated emulation environment. Moreover, it offers real-time tracing of `php` script executions. [1]

**ACM Reference Format:**
Francesco Pagano, Mariano Ceccato, Alessio Merlo, and Paolo Tonella. 2025. MITHRAS: A Dynamic Analysis Framework for the Mobile-IoT Ecosystem. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Over the years, IoT devices have become increasingly ubiquitous and play a critical role in various sectors of our daily lives. The software on these devices is embedded within their firmware, a self-contained file that includes the necessary software and kernel for the device's operation.

Usually, developers test the firmware of an IoT device by directly uploading it to the physical device. i) Modifying and re-uploading the firmware can potentially corrupt the device's software, causing it to become completely unresponsive and unusable, effectively rendering the device inoperable; ii) Physical devices often lack support for saving or restoring states, making it difficult to inspect the device's execution after a failure or to revert to a previous state; iii) Acquiring the specific device for analysis can be challenging, as it may be unavailable or discontinued; iv) Scaling the analysis across multiple devices is complex, as users must obtain multiple types of hardware.

Solutions such as Firmadyne [1] and FirmAE [8] enable developers to execute the firmware of IoT devices in an emulated environment. These tools emulate peripheral devices to allow device drivers to function correctly. They can start services on the emulated IoT device and expose them through web interfaces on dedicated private networks. However, they cannot establish full communication between the emulated device and external clients. Current state-of-the-art solutions are based on QEMU [9], which supports user-mode emulation of specific executables and full-system emulation of real devices. However, full-system emulation struggles to expose certain services, such as Multicast DNS (mDNS) [12], essential to make specific services available to external clients for a query. This limitation arises because QEMU employs SLIRP [10]. This model simulates a physical network but does not allow the emulated IoT device to connect to the Internet or handle ICMP packets. Evaluating communication between the IoT device and external clients is critical to a comprehensive device security assessment.

MITHRAS is the first emulation platform to extend the functionality of QEMU, enabling full-system emulation of the firmware of the IoT device while ensuring proper communication between the emulation IoT device and external clients. MITHRAS includes the following functionalities:

- **Firmware emulation and client communication**: Enables full emulation of IoT device firmware within QEMU, supporting communication between the emulated IoT device and external clients, such as companion apps running on emulated Android devices.
- **Security-testing support**: Supports security testing of IoT device-smartphone pairs by statically instrumenting the IoT device firmware and dynamically instrumenting the companion app's methods responsible for communication with the IoT device. Specifically, MITHRAS assists users in identifying and exploiting Remote Code Execution (RCE) vulnerabilities.

[1] Demonstration video available at: https://drive.google.com/file/d/1smSTfU9QK7RfLC_EHotcGAjEaJCMBf3M/view?usp=sharing


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

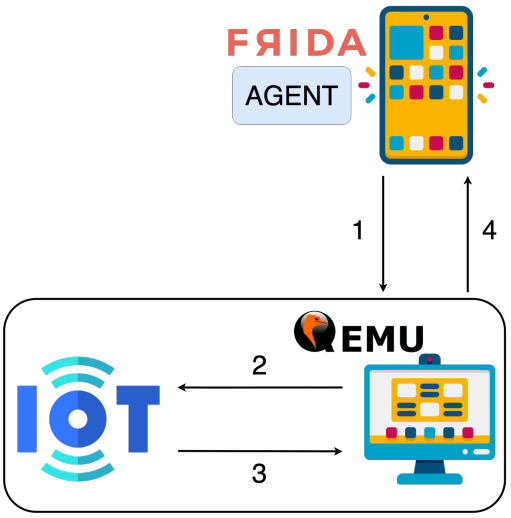


**Figure 1: MITHRAS's run-time workflow**

- **Firmware code instrumentation**: Uses the Php-Parser [7] library to insert logging code into `php` scripts, collecting code coverage information at the script, function, method, and branch levels. In addition, it monitors the execution of system commands by inserting logging code before critical PHP function calls, such as *system* and *passthru*, capturing both the executed commands and their output.

MITHRAS's implementation can be found in [2].

## 2 MITHRAS

This section outlines the workflow of MITHRAS. The workflow is divided into two main parts: the mobile component, which focuses on analyzing the target mobile companion app to be executed on the smartphone and setting up its dynamic instrumentation, and the firmware component, which involves the static instrumentation of the firmware's software and configuring the emulation environment to run the firmware.

### 2.1 MITHRAS's Run-Time Workflow

Figure 1 illustrates the execution workflow of the emulated environment of MITHRAS. First, MITHRAS launches the mobile companion app on the smartphone using a running instance of the Frida server, which is part of the Frida framework. The Frida server monitors the app's execution and allows the injection of user-defined code, referred to as a *hook*, replacing the original code of specific app's methods. The Frida agent monitors the execution of the app and, when a *sink method* (i.e., a method responsible for sending data directly to the connected IoT device) identified during the static analysis phase of the app (detailed in Section 2.2) is invoked, it runs the hook code from the uploaded agent script in place of the original method. The hook logs the method call information and sends it to the host connected to the Frida agent. Due to limitations in existing emulated environments, network requests from the emulated smartphone cannot reach the emulated IoT device, as both devices operate on distinct private networks. To overcome this issue, MITHRAS deploys a proxy server on the host machine that runs the emulated IoT device and the smartphone. The proxy server, built with the Flask [3] library, supports communication by redirecting incoming network requests to their intended target using the device's public address. The proxy server forwards network requests generated by the companion mobile app on the smartphone, which are directed to a host with a private IP address to the emulated IoT device. It is also responsible for relaying the responses received from the emulated IoT device back to the emulated smartphone. Most mobile companion apps discover the IoT device on the network through services exposed through the mDNS protocol. This protocol enables devices on the same local network to communicate directly by broadcasting service discovery requests to all devices on the network. When a device needs to access a service that another device provides, it sends a request to discover the device hosting that service. In the case of IoT device-smartphone communication, the mobile companion app must locate the IoT device on the same network. However, the IoT device and the smartphone reside on separate networks in the emulated environment. This network segmentation prevents ICMP packets, which the mDNS protocol relies on for device discovery, from traversing the emulated network. To address this limitation, MITHRAS installs a custom app on the smartphone that exposes an mDNS service within the local smartphone network, allowing the companion mobile app to establish a connection. The custom app uses the JmDNS [6] library to create and expose a simulated mDNS service that mimics the real one provided by the IoT device. From the perspective of the mobile companion app, the service exposed by the custom app appears to be the original service from the IoT device, allowing the companion app to proceed with the IoT device configuration.

### 2.2 Mobile Companion App Instrumentation Workflow

Figure 2 shows the workflow MITHRAS relative to the instrumentation of the companion mobile app. 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). The `Control-Flow Graph Extractor` module leverages the Androguard [5] library to compute the complete Control Flow Graph (CFG) of the app. Once the graph is calculated, 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), using the Androguard library. These sink methods 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 ones are stored in a repository (Step 4). The `Frida Agent Uploader` sends the Frida agent script and the list of methods to hook on the Frida server installed on the smartphone (Step 5). This script includes a list of sink methods

[2] https://anonymous.4open.science/r/Mithras-Tool-766F

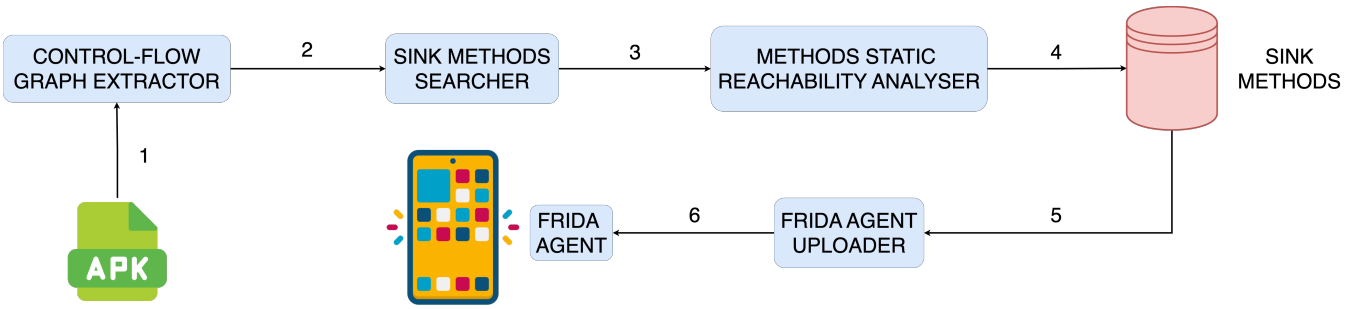


**Figure 2: Mobile workflow**

identified during the static analysis phase and the implementation of *hooks*. The agent script includes logic to log all sink method invocations and method calls along the execution path from a layout class to the sink function. It is a JavaScript file that the user can manually modify by adding JavaScript code to extend the functionality of sink methods, enabling customized behavior as needed. After generating the agent script as a single JavaScript file, the `Frida Agent Uploader` module uploads the file to the *Frida Agent* process, a component of the Frida library running on the smartphone (Step 6). This process is responsible for launching the app and loading the uploaded Frida agent script to instrument the app dynamically.

## 2.3 Firmware Instrumentation Workflow

Figure 3 illustrates the workflow of MITHRAS related to the instrumentation and emulation of the IoT device's firmware. First, the `Firmware Extractor` module extracts the firmware's content (Step 1), producing the *Extracted Firmware* (Step 2). Next, the `Emulation Initializer` module leverages the Firmadyne emulation library to set up the emulated environment for the extracted IoT device firmware (Step 3). Specifically, it automatically inspects the extracted file system of the firmware to identify the network interfaces exposed by the firmware. It also extracts other critical information necessary for emulation, such as the kernel version. The Firmadyne library uses kernel information to select the appropriate pre-built kernel for execution within the emulated environment.

The `Php AST Static Instrumenter` module uses the PhpParser library to modify the content of the `php` files extracted from the firmware (Step 4). These modifications insert logging code into the firmware's `php` files to track functions, method calls, and branch executions. Additionally, the instrumentation provides information on which `php` files are executed at runtime, thereby improving the coverage data collected during emulation. The code inserted by the `Php AST Static Instrumenter` module includes logic to send detailed information–such as which function, method, script, or branch was executed, along with the exact line number in the source file–as a string through a socket opened by a designated host that collects this data. The user must manually specify the host address during MITHRAS's configuration in a dedicated configuration file.

The `Php AST Static Instrumenter` module begins by computing the complete Abstract Syntax Tree (AST) of each `php` file found in the firmware's extracted file system. It then traverses each AST to perform the static instrumentation. The instrumentation process varies depending on the type of target statement: i) **Script Instrumentation**: The module inspects the root of the `php` file's AST and inserts logging code at the start of the AST to capture script-level execution. ii) **Function Instrumentation**: The module navigates the AST of each `php` file, searches for function definition statements, and inserts logging code at the beginning of each function's implementation block to track function calls. iii) **Method Instrumentation**: The module searches the AST of each `php` file for class definition statements, identifies methods within each class, and inserts logging code at the beginning of each method's implementation block to monitor method executions. iv) **Branch Instrumentation**: The module traverses the AST of each `php` file, looking for branch statements (e.g., if-else or switch statements), and inserts logging code at the start of the code block that executes if the branch condition is (resp. is not) met, enabling tracking of control flow paths. v) **System Command Tracking**: The module navigates the AST of each `php` file, searching for calls to system command execution functions (e.g., *exec*, *passthru*, *shell_exec*, *system*) and, when these function calls are found, it inserts logging code before the function call and modifies the call to log the execution trace of the command it executes.

After modifying the `php` file's ASTs, the `Php AST Static Instrumenter` module packs back the modified ASTs into `php` files and uploads them into the *Extracted Firmware* folder (Step 5). Finally, the `Firmware Emulator` module emulates the instrumented firmware (Step 6). This module is built entirely on the Firmadyne emulation library and uses QEMU for full-system emulation, enabling it to emulate firmware across a wide range of CPU architectures. The `Firmware Emulator` first prepares the file system to be pushed into the emulated virtual machine by packaging the statically instrumented files from the `Php AST Static Instrumenter` module into a QCOW2 [2] image file. It then performs an initial two-minute execution of the firmware to dynamically identify runtime information, such as exposed network interfaces, core services provided by the emulated firmware to external clients, and the programs that run at startup. After collecting this information, the `Firmware Emulator` automatically generates a `run.sh` script that configures the necessary tasks inferred from

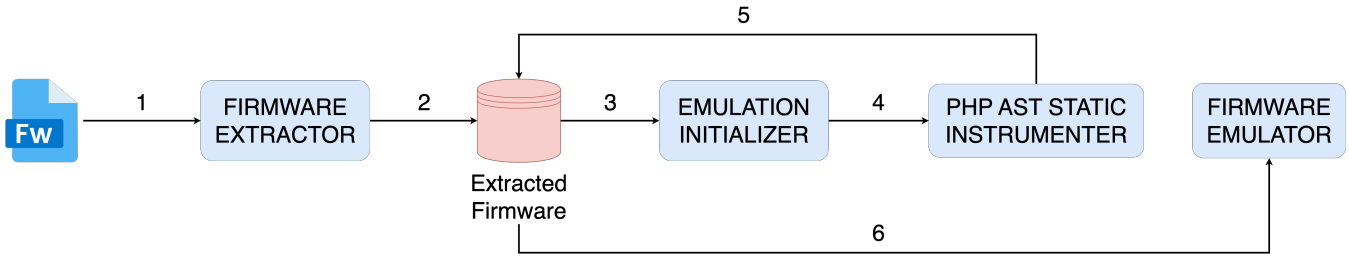


**Figure 3: Firmware workflow**

the initial firmware execution and proceeds with the full firmware emulation. The module also sets up a TAP network [4] between the emulated firmware and the host running QEMU, isolating all network traffic onto a dedicated, exclusive network to facilitate communication between the host and the emulated device.

## 3 RESULTS

Using ten D-Link firmware files, we evaluated the efficacy of MITHRAS's emulation capabilities. We selected firmware from this specific vendor because D-Link has publicly available firmware on its site, with known vulnerabilities documented in the Common Vulnerabilities and Exposures (CVE) database. We focused on router firmware, which typically connects with a dedicated mobile companion app. For the companion mobile app needed to interface with the IoT device, we used version 1.4.8 of the D-Link Wi-Fi app, available in the Google Play Store[3].

We evaluated the capabilities of Firmadyne and FirmAE to simulate a fully functional Mobile-IoT environment using these mobile companion apps and firmware. The evaluation demonstrated that Firmadyne and FirmAE fall short due to the limitations of their emulated network. MITHRAS, instead, successfully emulated all ten firmware files of the router, enabling proper communication with the companion app running on an Android smartphone emulator. To demonstrate MITHRAS's effectiveness in supporting security testing activities, we instantiated an existing app security testing framework that utilizes Deep Reinforcement Learning (DRL) [13]. Our tool automatically navigates the mobile app's interface to trigger communication with the IoT device, allowing for malicious modifications to the network requests directed at the emulated device to exploit vulnerabilities. We tested MITHRAS's capabilities through an experimental campaign, conducting ten episodes of 10 minutes each per firmware. The results demonstrated that our tool could exploit vulnerabilities on average 15 times per firmware.

## 4 CONCLUSIONS & FUTURE WORKS

This demonstration presents MITHRAS, the first simulation framework in which the firmware of an IoT device can be emulated using a system-level emulation approach, while also supporting communication with an emulated smartphone. Our framework supports the testing of scenarios where the IoT device interacts with its companion app installed on the smartphone, eliminating the need to physically set up real devices. MITHRAS also supports static instrumentation of the IoT device's firmware and dynamic instrumentation of the companion app, to collect comprehensive code coverage information from both platforms and to inject malicious payloads when the app communicates with the device.

MITHRAS currently supports only the static instrumentation of `php` files within the firmware. We plan to extend support to additional file types, such as binaries. Additionally, we aim to replace the Firmadyne library, currently used for emulating the IoT device's firmware, with Renode [11], an emulation platform that allows users to define low-level implementations of specific peripherals. This will enhance support for firmware that requires interaction with specialized hardware components.

[3] https://play.google.com/store/apps/details?id=com.dlink.dlinkwifi&hl=en

## REFERENCES

[1] 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.
[2] fileinfo.com. Accessed in January 14, 2025. QCOW2. https://fileinfo.com/extension/qcow2.
[3] flask.palletsprojects.com. Accessed in January 14, 2025. Flask. https://flask.palletsprojects.com/en/3.0.x/.
[4] gigamon.com. Accessed in January 14, 2025. Understanding Network TAPs – The First Step to Visibility. https://www.gigamon.com/resources/resource-library/white-paper/understanding-network-taps-first-step-to-visibility.html.
[5] github.com. Accessed in January 14, 2025. Androguard. https://github.com/androguard/androguard.
[6] github.com. Accessed in January 14, 2025. JmDNS. https://github.com/jmdns/jmdns.
[7] github.com. Accessed in January 14, 2025. PHP-Parser. https://github.com/nikic/PHP-Parser.
[8] 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.
[9] qemu.org. Accessed in January 14, 2025. QEMU. https://www.qemu.org.
[10] qemu.org. Accessed in January 14, 2025. QEMU Networking. https://wiki.qemu.org/Documentation/Networking.
[11] renode.io. Accessed in January 14, 2025. Renode. https://renode.io.
[12] rfc editor.org. Accessed in January 14, 2025. MDNS. https://www.rfc-editor.org/rfc/rfc6762.
[13] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep Reinforcement Learning for Black-box Testing of Android Apps. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 65:1–65:29.

## APPENDIX

### Demonstration Video

The demonstration video can be accessed at: https://drive.google.com/file/d/1smSTfU9QK7RfLC_EHotcGAjEaJCMBf3M/view?usp=sharing

## 4.1 Demonstration Walkthrough

Mithras is designed to run on a PC with Ubuntu. We recommend avoiding using virtual machines or Docker containers due to limitations in the QEMU process, which Mithras relies on. These limitations may cause issues when emulating specific IoT device firmware in virtualized environments.

### Tested Setup

- **Operating System**: Ubuntu 24.04 LTS
- **Processor**: Intel 9th Gen i9 or equivalent
- **Memory**: 32 GB RAM

Mithras has been tested and optimized on this hardware configuration to ensure stable performance during firmware emulation and testing. However, these configurations are only suggested ones

## SET UP ANDROID EMULATOR

### 1. Download and Install Android Studio

You can download Android Studio from the official website:
Download Android Studio.
Follow the instructions provided for your operating system to complete the installation.

### 2. Set Up Environment Variables

Export `ANDROID_HOME` environment variable into the .bashrc file

```
echo "export ANDROID_HOME=~/Android/Sdk" >> ~/.bashrc
source ~/.bashrc
```

### 3. Download and Configure Android Emulator

When setting up the Android emulator, select the **Google APIs** version instead of the **Google Play** version. We highly recommend installing the Android 11 emulator, which is the most stable and fully supports ARM translations. The Google APIs emulator grants root permissions, which are crucial for Mithras to interact with the companion app during testing.

We recommend configuring an x86_64 Android emulator, as the modified mobile companion app used to test Mithras' main functionalities supports only the x86_64 architecture.

## INSTALL MITHRAS

### 1. Prepare the Python Environment

**Install the Python Virtual Environment Package**
To manage dependencies effectively, install the `python3-venv` package:

```
sudo apt install python3-venv
```

**Create a Virtual Environment**
Use the `venv` Python module to create an isolated Python environment:

```
python3 -m venv venv
```

**Activate the Virtual Environment**
Before installing dependencies, activate the virtual environment:

```
source venv/bin/activate
```

**Install `wheel`**
Ensure that the `wheel` package is installed to handle precompiled binaries:

```
pip install wheel
```

**Install Required Packages**
Finally, install all dependencies listed in the `requirements.txt` file:

```
pip install -r requirements.txt
```

### 2. Prepare the PHP Environment

**Install PHP (CLI Version)**
Install PHP to run scripts from the command line:

```
sudo apt install php-cli
```

**Install Composer**
Install Composer for managing PHP dependencies:

```
sudo apt install composer
```

**Install the Php-Parser Library**
Install the `Php-Parser` library required for the project:

```
cd ./mithras/src/firmware-instrumenter
composer dump-autoload
composer require nikic/php-parser:^4.0
```

### 3. Set Up Emulator Engines

**Install Binwalk**
Binwalk is required for firmware extraction and analysis:

```
sudo apt install binwalk
```

**Install cURL**
cURL is required to ensure the firmware is functioning correctly.

```
sudo apt install curl
```

**(Suggested) Set Up FirmAE**
Navigate to the toolkit directory, and run the following scripts:

```
cd FirmAE
./download.sh
./install.sh
./init.sh
```

**Set Up Firmware Analysis Toolkit**
Navigate to the toolkit directory, copy the modified setup script, and run it:

```
cd firmware-analysis-toolkit
cp ../mithras/src/firmware-instrumenter/fat_setup.sh ./setup.sh
./setup.sh
```

**Update Root Password in `fat.conf`**
Add the current system root password to the `sudo_password` field.

# INSTRUMENT IOT DEVICE FIRMWARE

## 1. Download Firmware for Testing

Download a publicly available firmware file to be instrumented and emulated. The repository also contains a compatible companion app that can be used to communicate with the IoT device:

```
wget https://github.com/pr0v3rbs/FirmAE/releases/
    download/v1.0/DIR-868L_fw_revB_2-05
    b02_eu_multi_20161117.zip
```

## 2. Prepare Emulation Environment

We recommend using the FirmAE emulation engine, as it is the most stable and supports a broader range of firmware than the firmware analysis toolkit.

**Firmware Emulation with Firmware-Analysis-Toolkit**

Navigate to the Firmware Analysis Toolkit directory and start the firmware emulation:

```
cd ./firmware-analysis-toolkit
sudo ./fat.py <firmware-file>
```

**Firmware Emulation with FirmAE**

Navigate to the FirmAE directory and start the firmware emulation:

```
cd ./FirmAE
sudo ./run.sh -r dlink <firmware-file>
```

## 3. Firmware Static Instrumentation

**Set Up Instrumentation Configurations**

Modify the configuration file *./mithras/src/firmware-instrumenter/instrumentation_pipeline.json* with the necessary details for firmware instrumentation. The *emulation_engine* parameter must match the emulation engine chosen at the beginning of the demonstration:

```
{
    "firmware_name": "<name-of-the-firmware-to-
        ↪ emulate>",
    "root_password": "<system-root-password>",
    "emulation_engine": "firmae|firmadyne"
}
```

**Instrument IoT Device Firmware**

Activate the Python virtual environment and run the instrumentation pipeline:

```
source ./venv/bin/activate
cd ./mithras/src/firmware-instrumenter
python router_instrumenter_pipeline.py
```

## 4. Unpack Resources

**Unpack Frida-Server**

Extract the Frida server binary:

```
unzip ./mithras/bin/frida-server.zip
```

**Unpack Companion App APK**

Extract the companion app APK file:

```
unzip ./apps/companion_app.zip
```

## 5. Execute Mobile-IoT Emulated Environment

**Set Up Router Configurations**

Modify the configuration file *./mithras/router_mapping.json*. This configuration file contains a list of firmware details. For each firmware, provide a *name* (matching the firmware's instrumentation step) and its public *ip_address*:

```
{
  "firmware1": {"model": "firmware1_name", "
      ↪ ip_address": "firmware_1_ip_address"},
  "firmware2": {"model": "firmware2_name", "
      ↪ ip_address": "firmware_2_ip_address"}
}
```

## 6. Set Up Emulation Configurations

Modify the configuration files *./mithras/config.json* and *./mithras/router_mapping.json*. These files define the configurations for establishing the emulated Mobile-IoT ecosystem. Key options include:

- **firmware_name**: Match the name used during the instrumentation step.
- **android_sdk_platforms**: Define the path of the `platforms` folder in the Android SDK.
- **proc_name**: The package name of the mobile companion app.
- **device_id**: The Android emulator's identifier, used by `adb` to select the appropriate emulator when multiple instances are running.
- **device_name**: The Android emulator's name, matching the name given during creation.
- **root_password**: The password for the root user on the system host.
- **emulation_engine**: The emulation engine selected by the user (this must match the emulation engine chosen at the beginning of the demonstration). Valid options are: *firmae* or *firmadyne*.

## 7. Emulate Mobile-IoT Environment

Activate the virtual environment and execute the firmware test script:

```
source venv/bin/activate
cd ./mithras
export ANDROID_HOME="<path-to-Android-Sdk-Folder>"
python test_firmware.py --cf ./config.json
```