# MODE: A Customizable Open-Source Testing Framework for IoT Systems and Methodologies

Rareș Cristea[1], Ciprian Paduraru[1] and Alin Stefanescu[1,2]
[1]*Department of Computer Science, University of Bucharest, Romania*
[2]*Institute for Logic and Data Science, Romania*

Keywords: IoT, Fuzzing, Vulnerabilities, Application, Deployment, Guided.

Abstract: With the growing integration of software and hardware, IoT security solutions must become more efficient to maintain user trust, boost enterprise revenue, and support developers. While fuzzing is a common testing method, few solutions exist for fuzzing an entire IoT application stack. The absence of an open-source application set limits accurate methodology comparisons. This paper addresses these gaps by providing an open-source application set with real and artificially injected issues and proposing a framework for guided fuzzing. The solutions are language-agnostic and compatible with various hardware. Finally, we evaluate these methods to assess their impact on vulnerability discovery.

## 1 INTRODUCTION

The rapid growth of Internet of Things (IoT) applications has outpaced testing methodologies. IoT spans smart car systems, healthcare, transportation, vendor applications, and smart cities. IoT systems typically involve software for interconnected sensors, actuators, apps, gateways, and servers. The diversity of manufacturers complicates ensuring reliability and security. Problems arise at all levels, from isolated apps to protocols and interactivity over time. These vulnerabilities expose systems to attacks such as Distributed Denial-of-Service (DDoS) (Al-Hadhrami and Hussain, 2021)and identity management issues (Sadique et al., 2020).

M. Bures et al. (Bures et al., 2020) highlight the challenges of interoperability and integration testing in IoT systems, stressing the need for IoT-specific approaches to handle the combinatorial complexity of diverse devices. Limited standardization further hinders platform-agnostic testing tools (Dias et al., 2018).

In Fig. 1 we represent the four *challenges* in defining a comprehensive IoT testing setup. These challenges are further split between artifact and virtual challenges.

1. **Testing Devices -** There is no common basis for evaluating testing approaches using an application set with known, identifiable software issues. Such a set would enable comparative evaluation and rapid experimentation with various test methods, directly enhancing the testability of interoperability vulnerabilities.
2. **Testing Orchestrator -** A unique aspect of IoT systems is the ”hub” or device orchestrator, which compensates for the limited computing power of simple sensors and manages connections and communications. Most systems have a local edge device, while others rely on cloud-based orchestration.
3. **Testing Methodologies -** Many testing methods exist, but no clear framework compares them. Functional tests are the most common, while newer methods, like guided fuzzing, extend classical approaches to IoT. These face challenges such as interactivity and persistence at the application layer, requiring efficiency comparable to other solutions.
4. **Testing Context -** IoT-specific issues can be affected by factors external to the system (Seeger et al., 2020) (Kühn et al., 2018) or third-party systems (El-hajj et al., 2019).

This article builds upon our previous work (Păduraru et al., 2021), where we first introduced our abstraction of the communications in an IoT system and continued in (Cristea et al., 2022) to introduce the application set and the functional framework, which



Cristea, R., Paduraru, C. and Stefanescu, A.
MODE: A Customizable Open-Source Testing Framework for IoT Systems and Methodologies.
DOI: 10.5220/0013267500003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 441-448
ISBN: 978-989-758-742-9; ISSN: 2184-4895

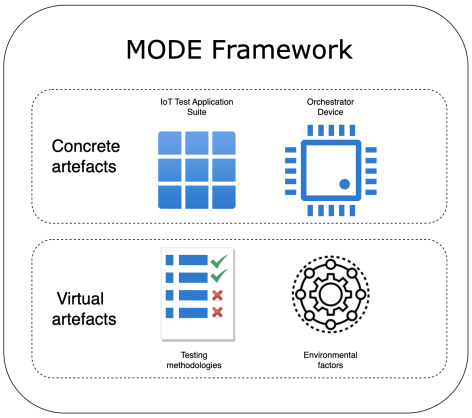


Figure 1: Graphical representation of the customizable components of the testing framework.

were necessary stepping stones to provide this article's fuzzing methodology. The previously discovered and introduced bugs are described and used in the fuzzing methodology.

In our previous work (Păduraru et al., 2021) we explored a theoretical framework that would enable more complete testing of IoT systems, by designing a communications system that simulates real-life IoT networks and allows developers or testers to leverage their preferred testing methodologies over the system. We further developed in (Cristea et al., 2022) an application set that works as a proof of concept of the theoretically described framework and a proof-of-concept version of the framework.

The contributions of this article are threefold:

- We define a testing framework offering variability across all four vertices of diversification. This was achieved by adding a fuzz testing methodology for end-to-end IoT scenarios. Our solution also leverages the developer's assumptions and prior knowledge about possible data flows at runtime.
- We extend the existing testing methodology with a distributed systems testing application called RESTler (Atlidakis et al., 2019).
- We provide an open-source application set that allows the users to apply the testing methods that we defined and compare them with new methodologies. Other developers can include their applications, thus extending the existing application set.

The paper is structured as follows: Section 2 reviews IoT testing efforts. Section 3 formalizes the IoT software stack using graphs. Section 4 details guided fuzzing methods. Section 6 evaluates these methods and their complementarity with functional testing. Section 7 concludes with future work.

## 2 RELATED WORK

Software testing is a vital part of the software development lifecycle. In IoT systems, this is more challenging due to the broad range of attack surfaces. Most IoT-specific tools are vendor-specific, supporting only limited devices and protocols (Dias et al., 2018).

Various organizations have issued standards for IoT system design. The W3C consortium proposed taxonomies for IoT vocabulary [1], including a JSON-formatted "Things Description" to abstract interactions in IoT systems. While thorough, this proposal requires extensions for real-world applications and existing communication protocols. Few ISO standards address IoT, such as ISO/IEC 21823-1:2019, which outlines a framework for IoT interoperability [2], though these are rarely adopted in industry (Gaborović et al., 2022).

Communication interfaces for IoT were explored in (Păduraru et al., 2021) and applied in (Cristea et al., 2022), with OpenAPI identified as a strong candidate for RESTful APIs. AsyncAPI, derived from OpenAPI, extends support to protocols like MQTT (Tzavaras et al., 2023).In IoT, low-power devices require an orchestrator to handle data processing. Typically, an IoT hub serves this role locally, but it can be offloaded to the cloud or a hybrid edge-cloud setup (Wu, 2021).

Bures et al. (Bures et al., 2020) suggest that "cross-over techniques between path-based testing and combinatorial interaction testing for close APIs in IoT systems" can be beneficial. The RESTler (Atlidakis et al., 2019) tool suite supports this approach, effectively analyzing cloud services using REST APIs. Architecturally, it identifies producer-consumer relationships from OpenAPI specifications. Our framework builds on RESTler by incorporating system-defining graphs, input/output variable dictionaries, and user-supplied communication flows to enhance effectiveness. (Bures et al., 2020) also review interoperability and integration testing literature, concluding that IoT-specific test configurations require tailored tactics beyond RESTler's capabilities, a challenge further explored by (Lin et al., 2022).

In our previous work (Cristea et al., 2022), we proposed fuzzing to deeply explore application vulnerabilities. Advanced fuzzers like AFL and its improvement AFL++ (Fioraldi et al., 2020) include implementations tailored for IoT. FIRM-AFL (Zheng et al., 2019) combines user and system mode emulation for optimal performance. (Eceiza et al., 2021) outlines

[1] https://www.w3.org/2023/10/wot-wg-2023.html
[2] https://www.iso.org/standard/71885.html

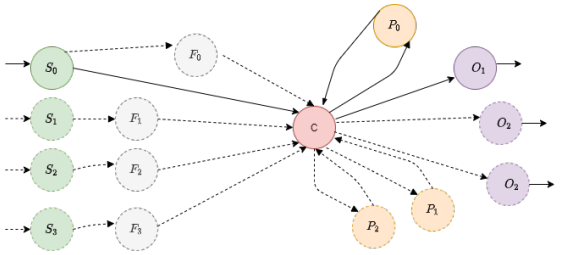


Figure 2: An example of a compatibility graph for an IoT software stack includes sensors $(S_0, S_1, S_2, S_3)$ collecting video inputs $(V_{init})$, image filtering nodes $(F_0, F_1, F_2, F_3)$, a central hub $(C)$ connecting all nodes, and processing nodes $(P_0, P_1, P_2)$ for tasks like detecting abnormal events.

fuzzing challenges in embedded systems, with test case generation being a key focus of tools like "Building Fast Fuzzers" (Gopinath and Zeller, 2019) and Skyfire (Wang et al., 2017), which uses grammar and example data to create cases precise enough for pretesting yet broad enough to uncover errors.

RESTful HTTP APIs have also been studied. (Martino et al., 2016) developed a framework to analyze Java or C/C++ code of IoT applications, generating a common semantic interface suited for open-source projects. In contrast, our approach assumes developer-provided interfaces, enabling custom integration with closed-source applications exposing APIs.

# 3 ABSTRACTING THE IoT TESTING ENVIRONMENT

In this section, we first formally define the abstractions we propose for the case of testing an IoT software stack based on graph theory. We then discuss some technical aspects required to implement graph mapping in a practical implementation. Finally, most of this section is devoted to presenting our proposed methods for end-to-end hierarchical fuzz testing of the implemented software stack in an IoT environment.

## 3.1 Graph Based Mapping

Continuing the work in (Cristea et al., 2022), we formalize the specification of connected IoT components using graph terminology. Fig. 2 describes an example of compatibility graph *compatibility graph*, $G_{compat}$.

We describe the producer/consumer relationship between devices using an oriented graph with the following rule:

1. $V$ - a set of nodes representing processes .
2. $E$ - a set of oriented edges describing the possible connections between the processes (nodes in $V$). An edge $e(source, destination) \in E$, represents that the output produced by the *source* node will connect to the input in the consumer, *destination* node. Further, for each node $v \in V$, we consider a set of incoming nodes, $V_{in}(v) = \{v_{in} \in V \,\|\, (v_{in}, v) \in E\}$, and outgoing nodes, $V_{out}(v) = \{v_{out} \in V \,\|\, (v, v_{out}) \in E\}$.
3. Developers can provide a knowledge base for each node $v \in V$, specifying input and output interfaces $If_{in}(v)$ and $If_{out}(v)$.
4. Developers can define hard requirements for deployment by specifying non-removable nodes $V_r$ and edges $E_r$ in $G_{compat}$.
5. Developers can specify probabilities $Prob_V$ and $Prob_E$ for nodes and edges, reflecting realistic usage scenarios where some processes and connections are more common.

At runtime, a subset $G \subseteq G_{compat}$ of the compatibility graph executes the required tasks. At time $t$, user requests or system events determine a subset of the graph, initiating communication flows between nodes. These flows represent sequences of ordered nodes processing inputs and outputs. Pure input nodes in $G$ are $V_{init(G)} = \{N_1^{(1)}, N_1^{(2)}, \ldots, N_1^{(R)}\}$, while output-only nodes are $V_{out(G)} = \{N_{Num_1}^{(1)}, N_{Num_2}^{(2)}, \ldots, N_{Num_R}^{(R)}\}$. All nodes and communication links belong to $G_{compat}$.

Communication between applications is centrally managed at the top level, with decentralization at lower levels. The orchestrator, a central hub application node (marked as $C$ in Fig. 2), is implemented in our framework. Its role is to trigger requests to applications, collect data, and forward it within the communication flow. Hierarchically, each node may have its own central node, as discussed in Section 6. A top-level central node, commonly described in IoT literature, simplifies process management, aids message observation, and improves fuzzing process control.

## 3.2 Communication and Endpoint Specifications

By using the OpenAPI specification, applications can automatically identify (through smart code agents) the set of all endpoints for inter-application communication in the IoT software stack defined by the compatibility graph $G_{compat}$. This also allows our framework to automatically identify the set of initial nodes (without input dependency), i.e., $V_{init}(G_{compat})$, and the format of input-output buffers, $Buffer_{out}$,

$Buffer_{in}$, for each node. Then, RESTler helps generate the source code needed to send requests and process responses between applications based on the given compatibility graph. The code generated in this step is a textitRESTler grammar. The resulting component can then be used by two other components: (a) *RESTler Test* to check the availability of each endpoint, and (b) *RESTler Fuzz* to generate and run guided tests and systematically explore the state space of the graph.

# 4 FUZZING METHODS

We assume that the developer is generally willing to specify a set of functional tests, as described in (Păduraru et al., 2021), (Cristea et al., 2022), representing various *communication flows* in the deployed application as a whole as part of any common software development process.

$$FuncTests_{S_{App}} = \{Test_1 = (G_1, Bf_{in}(G_1), Bf_{out}(G_1)), \ldots \quad (1)$$
$$Test_N = (G_N, Bf_{in}(G_N), Bf_{out}(G_{Nd}))\}$$

For a given IoT software stack of applications $S_{App}$, we denote this set as $FuncTests_{S_{App}}$, Eq. 1. Thus, a functional test in our abstract definition is an instance of $G$ as well as specifications for inputs and the corresponding expected outputs, i.e. $Bf_{in}(G) = \cup_{Bf_{in}(v) \| v \in V_{init}(G)}$ and $Bf_{out}(G) = \cup_{Bf_{in}(v) \| v \in V(G) and \nexists e \in E(G), s.t. source(e) = v}$

Our proposed fuzzing methods operate at two hierarchical levels:

- Level 1: The graph level where different instances of $G \subseteq G_{compat}$ are fuzzed. This simulates the use of different nodes and communication flows from the original compatibility graph. The purpose of fuzzing at this level is to detect as many potential problems associated with the different flows at runtime.
- Level 2: The buffers of the deployed processes. After applying Level 1 and obtaining a graph $G$, our methods can continue fuzzing on the input nodes in $G$, i.e., the set $V_{init}(G)$. An important feature of our framework is the support for persistence testing at this level.

A key challenge in fuzzing is efficiently managing resources to identify critical issues, especially in IoT systems with complex communication and persistence needs. Expanding on prior work (Cristea et al., 2022) using BDD for functional testing, the current strategy automates the analysis of developer-defined patterns and input-output hints to guide fuzzing effectively.

## 4.1 Fuzzing at the Graph Level

The algorithm for fuzzing a subgraph $G \subseteq G_{compat}$ involves three main steps:

1. **Initialization:** An initial graph $G = (V_r, E_r)$ is created, containing only the required nodes and edges to define a valid starting point.

2. **Sampling Input Nodes:** A random subset of input nodes from $V_{init}(G)$ is added to $G$. The number of nodes is sampled from a user-defined range $[MinInitNodes, MaxInitNodes]$, allowing the graph's initial size to be tailored to the test scenario.

3. **Dynamic Edge Addition:** For a random number of steps (from $[MinSteps, MaxSteps]$), edges are added to $G$ from $G_{compat}$, provided their source nodes are already in $G$ and the edges are not yet included. Edge selection follows user-defined probabilities ($Prob_E$).

Parameterization of node and step ranges ensures the algorithm adapts to different $G_{compat}$ sizes. The resulting graph $G$ provides a flexible runtime instance for fuzzing, combining scalability and randomness for effective IoT system testing.

## 4.2 Fuzzing at the Processes' Buffers Level

Starting from a fixed graph $G$, this fuzzing plane generates diverse values and parameters for application endpoints by modifying the input buffers of input nodes, $V_{init}(G)$, and their associated buffers, $Bf_{in}(G)$. The fuzzing process has two main objectives:

(a) Ensure output nodes $v$, which lack connections to other nodes in $G$, produce in-range output values for each parameter in $Bf_{out}(v)$.

(b) Test the individual processes (nodes) involved in the flows of $G$ for common problems such as crashes, non-determinism, etc.

### 4.2.1 Guiding the Fuzzing Process

The initial set of functional tests defined by the developer for the IoT software stack, i.e., $FuncTest_{S_{App}}$, can serve as the first level to suggest how to prioritize testing efforts. Thus, we propose a three-level fuzzing methodology. We define with $Test_k \in FuncTest_{S_{App}}$ defining the $k$-th functional test.

Each of these test specifications, which represent a workflow in a graph $G$, has a set of input variables/-parameters $Inputs(Test_k) \subseteq Bfs_{in}(G)$ that maps the variable names $P_{name_i}$ to their corresponding outputs

$P_{value_i}$, with $1 \leq i \leq card(Inputs(Test_k))$. In addition, we extract the possible ranges of values given for each variable name of a given application $Inputs(Test_k)$ from all values given by users in the entire test set and the optional hints in the data dictionaries.

These ranges of values are further aggregated and generative models are built for each parameter $P$. For example, if $P$ is a string type variable, this generative model becomes a regular string pattern expression (grammar). For numeric values, the learning process creates a value set $D(P)$ consisting of all the values given by the user in the functional tests for $P$, i.e., $D(P) = \{Value(P)_1, Value(P)_2, ....\}$, where the minimum and maximum of these values are determined, i.e., $min(D(P)$ and $max(D(P)$. Thus, $Range(P) = [min(D(P), max(D(P)]$.

If node $v$ and its application $A = App(v)$ have a set of input parameters $\{P_1, ..., P_{numInputs(A)}\}$, then the range of inputs generated for it by the fuzzing mechanism is denoted by $InputSpan(A)^i$, where $i$ is the index of the method used for the spanning process (more details later in this section). The span of the input of the fixed graph instance $G$ used by the fuzzing process at this level can be written as: $InputSpan(G)^i = \cup\{InputSpan(App(v))^i | v \in V_{init}(G)\}$

The sampling of the value for each input parameter $P$ of an application is controlled by one of the following three functor methods, referred to as *input span levels* in the text that follows:

1. **Input Span Layer 1 -** The first sampling method selects one of the discrete values in $D(P)$. Thus, this method generates permutations between the known values/clues given for each parameter.
2. **Input Span Layer 2 -** In the second layer, a sample is drawn from the range of values for each parameter $R(P)$. Permutations of values between the minimum and maximum values of each variable are determined if the variable $P$ is numeric, or a string corresponding to the regular grammar if $P$ is a string type. At this level, the first tests are generated with values that have not been used before.
3. **Input Span Layer 3 -** The third layer samples the value even over a wider range, taking into account the entire set of possible values that the parameter's value type $P$ can take, i.e., $R_{type(P)}$. This is the most general form of fuzzing and does not require any prior knowledge of the input parameter, i.e., it can be applied without hints, dictionaries of possible values for data types, or functional tests.

The algorithm in Listing 1 performs end-to-end fuzzing of a workflow within the graph $G$, using one of three sampling methods to set parameter values. It includes persistence checking, retaining the previous application state in $G$ at each input generation step. Setting the persistence parameter to 1 makes the algorithm equivalent to classical fuzzing, which clears memory state on each pass. The algorithm supports distributed execution due to the independence of sampling processes.

Listing 1: Fuzzing at input buffers level and checking results.

```
fuzzProcess(spanLayerIndex):
  Initialize applications in G
  For each A=App(v) with v∈V_init(G):
    Instantiate A
    Sample tests for persistency testing
    NumTests =
        UniformSample(MinPersistTests, MaxPersistTests)
    For testIter in 1...NumTests:
      Set new values
      For each input parameter P of A:
        SetValue(A,P) = SampleValue(P, spanLayerIndex)
      Simulate execution of G
      Evaluate results
```

#### 4.2.2 Scheduling Efforts

Technically, $InputSpan(G)^1 \subseteq InputSpan(G)^2 \subseteq InputSpan(G)^3$. While $InputSpan(G)^1$ is a finite set, the others are potentially infinite, so resource prioritization must be applied.

To make sense of the computational overhead, our method proposes the following time partitioning among the three levels defined above. We consider as the user's input parameter the total time allowed for the fuzzing process, denoted by $TimeAllowed$. In addition, the user also specifies the percentage of this total time that should be spent approximately performing the fuzzing on each of the three layers.

**Results Checking.** The evaluation of results comes from the call *Evaluate results* in Listing 1, line 13. The first level checks if fuzzing values produce outputs within the range defined by disconnected output nodes in $G$, referred to as *Cond* in Section 4.2. This harmlessness testing, valued in industry, enables basic checks and security testing. The second level identifies common fuzzing issues, such as segmentation errors and boundary crossings.

Fuzzing methods in software testing cannot ensure completeness, but user hints from functional tests or dictionaries can reduce time and effort. For example, if a camera sensor app produces only $360 \times 240$ images, not knowing this fixed resolution could lead the algorithm to search an infinite range, missing the cor-

rect one. Our evaluation ensures all injected errors in layer 1 (and possibly layer 2) are caught, while new issues are identified in layers 2 and 3.

# 5 RESULTS

The main parts of the resulting framework are the ten applications, the backend support to facilitate their deployment, testing, and an overview of how users can extend or replace existing applications without sacrificing background infrastructure. The implemented back-end infrastructure can be reused with minimal developer effort. All the artifacts presented in the framework are available open source and documented including the process of adding and removing existing applications.

The applications were originally developed as part of an undergraduate course in Software Engineering at the University of Bucharest in the 2020-2022 academic years. The goal of the course was to teach students software engineering methods and practices in the areas of IoT and security. The students were free to choose what type of IoT device they would create the software for. We selected from the final projects those that could be best reused for security testing. Our experiment aligns with (Liu, 2005), showing that real use cases motivate students while also helping identify issues in their source code. The pedagogical process that enabled student-led contribution to the application set is detailed in (Cristea and Păduraru, 2023).

The application set is open-sourced, available, and documented on GitHub[3]. The current set includes various smart home applications. These are built using three different programming languages: Rust, C++, and Python. The variety in programming languages used for development follows the diversity found in the Smart Home market.

Communication between devices is mainly handled through HTTP and complementary functions through the MQTT protocol using Mosquitto [4]. MQTT's advantage is that it is lightweight and compatible with many operating systems and hardware, from lower-powered devices to complete servers. The solution implies the use of a publisher/subscriber model for processing messages through the *application orchestrator* node in our graph-based deployment.

As for deployment, our backend infrastructure provides immediate support with available scripts and documentation, using two methods:

1. Docker deployment, where each application runs in a Docker container on a specific *IP* and *PORT*. This has the advantage of being convenient, consuming fewer resources, and can be used without preparation or special hardware. It works on various popular operating systems, on users' local PC, or in cloud environments.
2. *RaspberryPi* devices, where any application can be used on a real embedded *RaspberryPi* device (we did our tests on the Raspbian ARM v7l OS ). Communication is handled over the available Wi-Fi connections.

# 6 EVALUATION

## 6.1 Vulnerability Issues and Artificial Injection

Assessing the literature (Zhu et al., 2022), (Păduraru et al., 2021), (Cristea et al., 2022), we divide the problems into three different categories:

1. Application-level problems: most commonly occur at the application level and result in invalid responses or crashes.
2. Communication flow problems: The expected behavior is not met or undefined behavior occurs after a runtime flow that connects one or more applications.
3. Persistence level problems: The expected behavior is not met or undefined behavior occurs after multiple inputs are applied in sequence, either at a single application level or using a connected flow of applications.

Our application suite, developed by independent teams, reflects real-world IoT scenarios where software and devices from different vendors form complex, often unpredictable systems. A list of known bugs is available in our repository. Currently, there are 15 bugs, including 7 source code issues such as segmentation errors, data range and buffer index checks, and concurrency problems. These application-level issues were missed during student testing sessions and the required functional testing for the final project.

The remaining 8 problems were manually introduced to assess our method's ability to detect subtle persistence issues and errors in multi-application communication. For instance, one issue involved the lack of cleanup in the communication flow between a *SmartTV* application (adjusting screen brightness based on lighting) and a *WindowWow* application (serving a smart window). A bug was introduced

[3]https://github.com/unibuc-cs/IoT-application-set
[4]https://mosquitto.org/

where *WindowWow* sends excessively high values outside the TV's range, preventing the device from updating brightness or responding effectively.

An example of a persistence error was injected into the same *WindowWow* application that uses a light sensor to automatically control the opening or closing of curtains connected to the smart window device.

The following is a list of examples of issues discovered in our application set. Each issue was triggered by one or multiple applications communicating. Some issues were identified as part of an automated rule defined in our Hub Application:

- FlowerPower - Application - Flowerpower: does not check for optional key existence in JSON object on PUT /settings.
- SmartTV - Application - TV brightness should be set to a maximum of 10, but the value is not validated by the app.
- FlowerPower, WindWow - Communication - Rule 2 reduces the window's luminosity if the temperature is over 30 degrees, but Rule 3 unnecessarily turns on the lamp due to low luminosity.
- WindWow - Persistence - WindWow crashes when trying to set luminosity to 25 and curtains are closed on GET /settings/settingName/settingValue (artificial bug).
- SmartKettle, WindWow - Communication - SmartKettle's temperature decreases for WindWow's temperatures under 0 degrees Celsius instead of increasing.
- SmartTeeth - Application - "localhost" is set as the hostname of the listening server, refusing outside connections.
- FlowerPower - Application - In FlowerPower: activateSolarLamp does not change luminosity.

## 6.2 Test Methods Evaluation

The effectiveness of our proposed fuzzing methods is evaluated as follows: Each application in the set was required to have functional tests based on the BDD methodology (Cristea et al., 2022). However, source code coverage does not guarantee state coverage. For example, a line accessing an array index may be marked as covered even if only one index value is tested, leaving many cases untested (Hemmati, 2015).

To evaluate the proposed fuzzing method in Listing 1 and the layers defined in Section 4.2.1, we fuzz-tested each application (node) on a separate physical process and the *central hub* node. Next, we applied system-level fuzz testing, and finally, we assessed the performance of these methods in detecting both known and new issues.

The results indicate that functional tests, despite achieving near-full code coverage, failed to detect all 15 known issues. The $layer, 1, method$ found 6 issues, $layer, 2, method$ detected 11, and only the

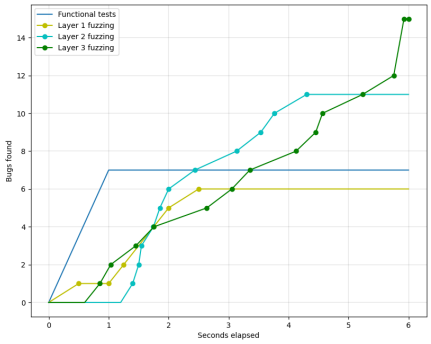


Figure 3: Visual representation of the effectiveness between the three fuzzing layers and functional testing. While functional tests are run almost instantly, they are limited as they are individually crafted. The fuzzing method provides better results for issue identification.

$layer, 3, method$ uncovered all. Layer 3 relies on fuzz testing and consumes its time budget, with 6 seconds (96 requests) being sufficient to detect all bugs. However, blind testing raises resource costs, emphasizing the need for effective time management. Functional tests can address known issues first, followed by sequential application of the three methods, with $layer, 3, method$ extending beyond regression windows for thoroughness (Do et al., 2008).

An hour-long fuzz testing session for the current applications generated 61,860 requests and flagged 1,165 issues. Depending on system configurations, malformed requests might or might not qualify as bugs. Testers can adjust fuzzer configurations to better define bugs and improve relevance.

This study proposes a framework for evaluating IoT testing solutions against state-of-the-art methods. Our approach enhances RESTler (Atlidakis et al., 2019) by integrating graph terminology, dependency checks, user-provided hints, and test suites, expanding the *testing surface*. However, it requires additional setup effort from developers.

# 7 CONCLUSIONS AND FUTURE WORK

This paper presented a framework for testing an IoT software stack using guided fuzzing and introduced the first open-source application set offering backend-level reusability for further experimentation. Artificial errors were inserted to evaluate the effectiveness of our fuzzing methods, which work in IoT environments regardless of programming language or hardware. Results were compared based on efficiency in detecting problems and computation time. Preliminary findings show that fuzzing effectively identifies

issues in deployed IoT stacks, particularly when developers contribute with hints, data dictionaries, and parameter specifications. These inputs can be manually added or automatically extracted from existing functional tests within our framework. Future plans include expanding the application set and source code issues for better method comparisons and investing in persistence testing with symbolic and concolic execution to provide faster feedback by identifying linked parameters.

## ACKNOWLEDGEMENTS

This research was supported by European Union's Horizon Europe research and innovation programme under grant agreement no. 101070455, project DYNABIC.

## REFERENCES

Al-Hadhrami, Y. and Hussain, F. K. (2021). DDoS attacks in IoT networks: a comprehensive systematic literature review. *World Wide Web*, 24(3):971–1001.

Atlidakis, V., Godefroid, P., and Polishchuk, M. (2019). RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st ICSE*, pages 748–758. ISSN: 1558-1225.

Bures, M., Klima, M., Rechtberger, V., Bellekens, X., Tachtatzis, C., Atkinson, R., and Ahmed, B. S. (2020). Interoperability and Integration Testing Methods for IoT Systems: A Systematic Mapping Study. In *Software Engineering and Formal Methods*, pages 93–112.

Cristea, R., Feraru, M., and Paduraru, C. (2022). Building blocks for IoT testing - a benchmark of IoT apps and a functional testing framework. In *2022 IEEE/ACM 4th International Workshop (SERP4IoT)*, pages 25–32.

Cristea, R. and Păduraru, C. (2023). An experiment to build an open source application for the Internet of Things as part of a software engineering course. In *2023 IEEE/ACM 5th International Workshop (SERP4IoT)*.

Dias, J. P., Couto, F., Paiva, A. C., and Ferreira, H. S. (2018). A Brief Overview of Existing Tools for Testing the Internet-of-Things. In *2018 IEEE ICST*, pages 104–109.

Do, H., Mirarab, S., Tahvildari, L., and Rothermel, G. (2008). An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the 16th ACM SIGSOFT International Symposium on FSE*.

Eceiza, M., Flores, J. L., and Iturbe, M. (2021). Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems. *IEEE Internet of Things Journal*. Conference Name: IEEE Internet of Things Journal.

El-hajj, M., Fadlallah, A., Chamoun, M., and Serhrouchni, A. (2019). A Survey of Internet of Things (IoT) Authentication Schemes. *Sensors*, 19(5):1141. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++ : Combining Incremental Steps of Fuzzing Research.

Gaborović, A., Karić, K., Blagojević, M., and Plašić, J. (2022). *Comparative analysis of ISO/IEC and IEEE standards in the field of Internet of Things*.

Gopinath, R. and Zeller, A. (2019). Building Fast Fuzzers. arXiv:1911.07707 [cs].

Hemmati, H. (2015). How Effective Are Code Coverage Criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156.

Sadique, K., Rahmani, R., and Johannesson, P. (2020). "IMSC-EIoTD: Identity Management and Secure Communication for Edge IoT Devices". en. In: *Sensors* 20.22 (). (Visited on 03/07/2022).

Kühn, F., Hellbrück, H., and Fischer, S. (2018). A Model-based Approach for Self-healing IoT Systems:. In *Proceedings of the 7th International Conference on Sensor Networks*, pages 135–140.

Lin, J., Li, T., Chen, Y., Wei, G., Lin, J., Zhang, S., and Xu, H. (2022). foREST: A Tree-based Approach for Fuzzing RESTful APIs. arXiv:2203.02906 [cs].

Liu, C. (2005). Enriching software engineering courses with service-learning projects and the open-source approach. In *Proceedings of the 27th ICSE*, pages 613–614. ACM.

Martino, B. D., Esposito, A., and Cretella, G. (2016). Towards a IoT Framework for the Matchmaking of Sensors' Interfaces. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 888–894.

Păduraru, C., Cristea, R., and Stăniloiu, E. (2021). RiverIoT - a Framework Proposal for Fuzzing IoT Applications. In *2021 IEEE/ACM 3rd International Workshop (SERP4IoT)*, pages 52–58.

Seeger, J., Bröring, A., and Carle, G. (2020). Optimally Self-Healing IoT Choreographies. *ACM Transactions on Internet Technology*, 20(3):27:1–27:20.

Tzavaras, A., Mainas, N., and Petrakis, E. G. M. (2023). OpenAPI framework for the Web of Things. *Internet of Things*, 21:100675.

Wang, J., Chen, B., Wei, L., and Liu, Y. (2017). Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. ISSN: 2375-1207.

Wu, Y. (2021). Cloud-Edge Orchestration for the Internet of Things: Architecture and AI-Powered Data Processing. *IEEE Internet of Things Journal*, 8(16):12792–12805. Conference Name: IEEE Internet of Things Journal.

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., and Sun, L. (2019). FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the 28th USENIX Conference on Security Symposium*.

Zhu, X., Wen, S., Camtepe, S., and Xiang, Y. (2022). Fuzzing: A Survey for Roadmap. *ACM Computing Surveys*, pages 230:1–230:36.