IoT Fuzzing using AGAPIA and the River Framework

Eduard Stăniloiu[1], Rareș Cristea[2] and Bogdan Ghimiș[2]
[1]*Department of Computer Science, University Politehnica of Bucharest, Bucharest, Romania*
[2]*Department of Computer Science, University of Bucharest, Bucharest, Romania*

Keywords: Internet-of-Things, Fuzzing, Specification Graph, Testing, Software Engineering Tools.

Abstract: As the number of Internet of Things (IoT) systems continues to grow, so does the security risk imposed by interconnecting heterogeneous devices from different vendors. Testing and validating the security of IoT systems is difficult, especially due to the fact that most of the software is proprietary (closed-source) and the system's embedded nature makes it hard to collect data, such as memory corruptions. This paper proposes to extend the novel AGAPIA language to enable IoT developers to write safer programs that can be tested and validated with state of the art fuzzers, such as RiverIoT. We present how simple additions can enable AGAPIA modules to be integrated with the RiverIoT architecture, thus facilitating better device testing. The proposed approach also enables users, not just developers, to perform system wide, black-box, testing, increasing the reliability of the system. We show how the abstractions provided by the AGAPIA language enable the fast development of an Air Quality Monitoring application and how small additions to existing programming languages can improve the testing and validation of IoT systems.

1 INTRODUCTION

The Internet of Things (IoT) is a system of interconnected devices that can collect data from their surrounding environment and act upon it. IoT has grown significantly in recent years, helping people to work smarter and improving their quality of life. We can now find smart devices everywhere, from our homes (smart light bulbs, IP cameras), to our cars (smart sensors, cameras) and cities (weather sensors, pollution sensors). With the continuous development of network technologies, such as 5G, the number of IoT devices will continue to grow (Li et al., 2018), bringing even more technology into our lives in a bid to make everything faster, smarter and more comfortable.

The growing number of IoT devices (and their vendors) rises privacy and security concerns. In a fast paced world, where the vendors compete with eachother to achieve the best time to market for device development (Wurm et al., 2016), there is a lot of room for programming errors that can lead to vulnerabilities and exploits (English et al., 2019) (Hernandez et al., 2014). The typical architecture for an IoT system is composed of multiple heterogeneous devices, not necessarily from the same vendor, connected through different protocols over the Internet. Most of the devices run proprietary firmware, with in-house implementations of protocol standards, developed in a memory unsafe language (such as the C programming language), making them a lucrative target for attackers.

Given the closed-source nature of the IoT devices code, it is difficult for a 3rd party (user, another vendor, etc) to audit and validate the correctness of the implementation. Because it can not access the code, the 3rd party must rely on black-box testing, combining fuzzing techniques with functional testing. RiverIoT (Paduraru et al., 2021) is an open-source framework that enables the fuzzing of end-to-end IoT applications. In order to provide efficient fuzzing, RiverIoT relies on a JSON specification of the Input/Output (I/O) of the devices fuzzed. We'll discuss this in greater detail in Section 2.2.

As previously stated, the programming language used to develop the devices' firmware plays a significant role in the security and robustness of the system. IoT systems and their applications are, in fact, a highly dynamic and modular distributed system. Distributed programming and synchronization is generally regarded as a non-trivial task. There are multiple frameworks and high-level languages that address the issue of distributed programming. The AGAPIA language (Paduraru, 2014) attempts to increase the developer productivity and the language expressiveness by using a transparent communication model and simple high level statements. AGAPIA expresses dis-

Stăniloiu, E., Cristea, R. and Ghimis, B.
IoT Fuzzing using AGAPIA and the River Framework.
DOI: 10.5220/0010604503240332
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 324-332
ISBN: 978-989-758-523-4
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

tributed applications as *modules* that expose a clear, simple and structured I/O interface.

AGAPIA is a Domain Specific Language (DSL) built on top of the C programming language. Because of this, (we believe that) it can easily be used with existing firmware code to model the IoT system's interaction and expose the device's I/O interface.

In this paper, we propose to express IoT devices as AGAPIA modules, and represent the IoT system's interactions as relationships between AGAPIA modules. By doing so, we have a clear, structured, specification of the I/O of a device. We can easily extend AGAPIA to convert the structured I/O spec to (and from) JSON, so we can easily use RiverIoT to test both the device and the IoT system.

The rest of the paper is structured as follows. Section 2 details the AGAPIA language and RiverIoT. Section 3 presents related work. Section 4 presents the necessary AGAPIA extensions and how to leverage AGAPIA's I/O models with RiverIoT. Section 5 presents an example of how AGAPIA can be used with an IoT system. Section 6 concludes.

2 BACKGROUND

2.1 AGAPIA

There is a real need for developing a unifying programming language that allows developers to quickly prototype and test their software before deploying it to an IoT device. The current trend is to write the code in a low-level programming language, like C, that will be compiled with specialized compilers (like the Arduino IDE) for each IoT device since the code can be run on different architectures (like PIC, AVR, ARM).

The problem with writing in a low-level programming language is that it is a tedious and error prone process, since one must carefully implement the basic data structures and communication channels. While this offers a lot of flexibility and control in terms of instructions granularity, it hinders development. Even when one has access to a powerful IoT device (like the Raspberry PI) and can write code in a higher-level programming language, like Python, a problem still persists: it is difficult and time consuming to write correct, distributed programs.

There are papers that investigate a way to program using visual objects like (Leonardo, 2013) and (Boshernitsan and Downes, 2004), but they fall short on developing for the multiprocessor devices.

AGAPIA (Paduraru, 2015) is a Domain Specific Language (DSL) that was designed to simplify the development of parallel software using a simpler syntax for spawning new processes via forks and integrating with Open-MPI (Gabriel et al., 2004) to deliver a friendlier distributed and parallel programming experience.

The main component of an AGAPIA program is the module that can be thought of a 2 dimensional block (a square) that can receive information from both the north and west sides and can pass information through both east and south parts (Figure 1). The inputs and the outputs are optional, so one can define modules that generates data (no inputs, only outputs) or that produce side effects when given certain inputs (only inputs, no outputs).

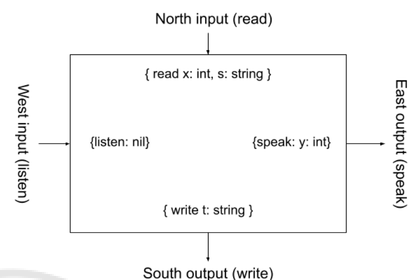

Figure 1: An AGAPIA module.

The main job of a module is to transform the given input(s) into the output(s). The basic way of defining a module is to define its interface (the inputs and the outputs) and based on them one can compose modules together.

By defining a module in this way, one can compose modules in three ways: vertically, horizontally, and diagonally (Figures 2, 3, and 4). One can think of an AGAPIA program as a two dimensional structure (Banu-Demergian et al., 2013), composed of modules that talk to one another in order to solve a problem more efficiently. The composition model used by AGAPIA is multiplexed both in space (Figure 2) and time (Figure 3) (Paduraru and Stefanescu, 2020).

After defining the interface for each module, one can write the code for that module in either C or C++ language. The user code for each module can contain either pure C/C++ code (called here atomic) or can contain AGAPIA instructions or compositions (Stefanescu and Paduraru, 2016). The difference between the two modes is that the atomic code must have access to all the variables before it is scheduled, whereas when using AGAPIA instructions one can run code in parallel.

The AGAPIA team is currently working on producing a graphical user interface where one can easily define the topology of an AGAPIA program and define the interface between the modules and after that can write the module's code.

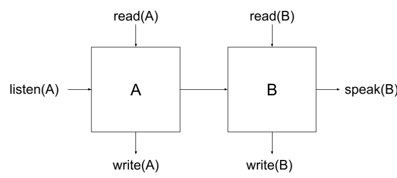

Figure 2: Horizontal module composition.

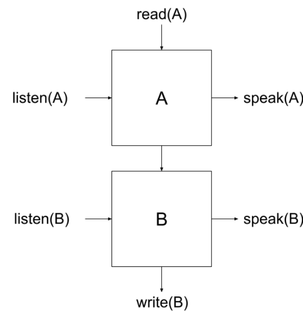

Figure 3: Vertical module composition.

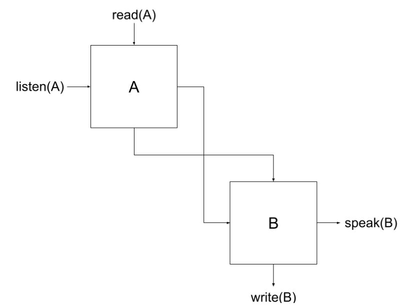

Figure 4: Diagonal module composition.

2.2 RiverIoT

RiverIoT is an integrated testing framework that enables end-to-end fuzzing for IoT systems. The framework employs guided fuzzing through state of the art methods, like concolic execution (Ghimis et al., 2020) and AI techniques (Paduraru et al., 2019), (Paduraru. and Melemciuc., 2018), (Paduraru. and Melemciuc., 2018), (Paduraru et al., 2020).

There are two major challenges when fuzzing embedded devices: 1) the hardware dependence of the firmware and 2) the closed-source nature of the firmware. Inspired by (Feng et al., 2020) and (Clements et al., 2020), RiverIoT addresses both of those issues with emulation. Using an emulator allows RiverIoT to run the firmware on a general purpose system, without requiring 1) the physical hardware of the IoT device. When a binary is emulated, the emulator translates each binary instruction into an intermediate representation before translating it again into instructions for the host machine architecture. Doing so, RiverIoT can dynamically instrument the binary instructions without needing the 2) source code.

Since an IoT system can be logically represented as a graph of connected components, RiverIoT can perform fuzz testing at three different levels:

- graph level - changing the nodes and edges inside the deployed IoT system
- node level - fuzzing each device individually
- interconnection of input/output nodes between different nodes and their communication (i.e., how a node can be influenced by others producing its input connection, or the communication channel between them).

In order to use RiverIoT, one must provide a description of the graph (nodes and edges) and the I/O structure that the applications expect, in JSON format. With this JSON specification, the framework can understand the communication between the different components and it can start mutating the graph and generate inputs for the applications.

RiverIoT expects the graph to be able to handle dynamic configurations at runtime, like changing nodes and edges. The reason for this is that the framework attempts to simulate issues like device failures, restarts, repositioning thus testing the resilience of the system under test.

As it can be seen in the graph specification presented in Listing 1, the model is quite simple and safe-explanatory. One must define the IoT applications (”iot-nodes”) and their communication channels (”io-edges”). Each application is assigned an ID and a description of it's input and output buffers, as exemplified in Listing 2. Each edge, also identified by an ID, describes the input and output node (based on the node's ID) and the buffer specification used for that node (based on the node's buffer ID).

```
{
  "configuration-name": "Generic
      Network",
  "iot-nodes": {
    "1": { // Buffer description },
    "2": { ... }
  },
  "io-edges": {
    "1": {
      "vout": 3, // Output node
      "vin": 1, // Input node
      "vout-buffer": 1, // Buffer id
          sending the data
      "vin-buffer": 2, // Buffer id
          receiving the data
    },
    "2": { ... }
  }
}
```

Listing 1: Compatibility graph specification example.

The example in Listing 2 probably contains some more fields than one might be expecting, but most of them are actually there to provide more context to the

reader. RiverIoT is mostly interested in the ”token-type” and ”byte-size” fields. With those two, it can start generating inputs for the application under test.

```
{
  "device-name": "An IP Camera",
  "optional": "false",
  "class": "camera",
  "buffers": {
    "1": {
      "token-delimitators": " ",
      "protocol": "HTTP",
      "protocol-setting": "http://192.1
         68.0.112:8080/",
      "buffer-tokens": [{
        "name": "Camera command",
        "description": "Input that
            selects command",
        "token-type": "string",
        "byte-size": 256,
        "regex-rule": "[a-zA-Z]+=[a-zA-
            Z0-9]+", // Optional
            parameter to guide fuzzer
            generator
        "optional": false
      },
      {
        "name": "Camera ISO Value",
        "description": "Sensitivity to
            light",
        "token-type": "int",
        "byte-size": 4,
        "optional": true
      }]
    },
  }
}
```

Listing 2: Single device buffers’ specification.

With the JSON specification ready, the framework can start the fuzzing process. For each node, the framework will perform guided fuzzing in isolation, testing each binary program that can be deployed on a physical device. RiverIoT employs a combination of symbolic execution (Ghimis et al., 2020) and genetic algorithms (Paduraru et al., 2017) to fuzz the targets in a bid to achieve good code coverage while keeping the runtime performance and overhead acceptable.

3 RELATED WORK

The number of IoT devices has long passed the number of people on Earth, and with the continuous development of network technologies, computing power and applications, the number of devices is set to keep growing. As IoT devices become omnipresent in our lives and homes, it is only natural to become concerned about the privacy and security risks that they bring into our lives. To address those, we need to discover the tools that enable us to test and validate complex IoT systems.

Fuzz testing has proven time and time again that it can detect real-life vulnerabilities in existing systems. With the growing popularity of IoT, researchers are starting to focus on fuzz testing IoT devices.

FIRM-AFL (Zheng et al., 2019) is an IoT specific, grey-box fuzzer that runs on the AFL[1] fuzzer. Because most of the IoT applications firmware is proprietary, the traditional approach of re-compiling the source code with fuzzer specific instrumentation is not feasible. Instead, FIRM-AFL uses both user-mode and system-mode emulation to dynamically instrument the binary. Combining the two emulation types enables FIRM-AFL to achieve better performance than only using system-mode emulation such as AFL and AFLplusplus (Fioraldi et al., 2020). However, the firmware must run a POSIX compatible operating system for it to work.

Others, like AFLNet (Pham et al., 2020) and IoT-Fuzzer (Chen et al., 2018) capture valid messages exchanged by the applications to build a meaningful initial seed corpus and then replay and mutate those messages. AFLNet also builds a graph of dependencies between messages; doing so it is able to determine that certain messages are valid only after others: i.e. when using the FTP protocol, an application must first authenticate before it is able to use any other command.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Skyfire (Wang et al., 2017) is a data-driven seed generator for fuzzer’s inputs. It consumes an input corpus and a grammar of highly-structured inputs and generates a probabilistic context-sensitive grammar (PCSG) that encapsulates syntax and semantic rules. Providing AFL with the generated PCSG they have increased the code line coverage and funcion coverage by 20% and 15% respectively. Doing so, their experiment confirms that providing an input description can lead generic fuzzers to perform big-step mutations based on the grammar. Grammar-based testing is also the focus of Zeller and Gopinath’s “Building Fast Fuzzers” (Gopinath and Zeller, 2019), which focus on improving the throughput of fuzzed inputs.

One of the main challenges of fuzzing IoT devices is the hardware-dependence of the firmware. P2IM (Feng et al., 2020) focuses on providing a framework that eliminates this challenge and provides a common interface for off-the-shelf fuzzers to continuously execute a firmware binary. P2IM is generating a peripheral model based on the classification of the registers accessed by the firmware, by watching the read / write

[1] https://github.com/google/AFL

requests on registers. It accesses memory-mapped registers through the interfaces exposed by the QEMU [2] processor emulator. The reliance on QEMU restricts the number of possible fuzzers that can be used with the framework to the ones that have QEMU integration.

HALucinator (Clements et al., 2020) challenges P2IM's tightly coupled model, and proposes a decoupling model between the hardware and firmware of embedded systems. Based on the fact that firmware developers regularly develop code using abstractions, such as Hardware Abstraction Layers (HALs), HALucinator proposes a framework that implements High-Level Emulation (HLE), by implementing hooks for the HALs' exposed functions.

Industry efforts have led to the development of IoT I/O standards, like the MIPI general-purpose set of standards, (MIPIAlliance, 2020), or the ONVIF protocol for IP Camera communications (Organization, 2020). Having accepted I/O standards and protocols in place enables the interconnectivity of different IoT devices from different vendors. Standard protocols also enable the use of protocol-based fuzzers like AFLNet.

RiverIoT proposes a framework that enables greybox fuzzing of IoT binaries. Like others, it uses emulation to overcome the lack of source-code and performs dynamic binary instrumentation to guide the fuzzing process. It requires a description of the IoT system, modeled using JSON, that describes the system as a graph of nodes and edges. The JSON specification also describes the expected structure of the Input and Output of the applications running on the nodes. With this specification, RiverIoT is capable to fuzz not only individual nodes, but also fuzz the entire system by performing graph mutations.

AGAPIA (Paduraru, 2014) was designed to simplify the development of parallel software and to increase the developer productivity by using a transparent communication model and simple high level statements. The language aims to deliver a friendly distributed and parallel programming experience. AGAPIA expresses distributed applications as *modules* that expose a clear, simple and structured I/O interface. AGAPIA is a Domain Specific Language (DSL) built on top of the C programming language. Because of this, we believe that existing firmware code can easily be integrated in AGAPIA models, exposing in a structured way the IoT system's interaction and the devices' I/O interfaces.

[2] https://www.qemu.org/

4 IMPLEMENTATION

AGAPIA represents distributed applications as a graph of connected modules with structured I/O interfaces. The structured I/O interfaces and the graph representation used by AGAPIA fits perfectly with RiverIoT's architecture. We propose that we leverage this structure and build upon it to provide a simple interface that can interact with RiverIoT's JSON I/O description model. We plan on doing so by adding AGAPIA extensions that convert AGAPIA I/O Simple Interfaces to and from JSON objects.

4.1 AGAPIA Extentions

4.1.1 Generating Interface Descriptions

The AGAPIA language uses module abstractions to represent running processes. A module performs I/O operations by defining at most four interfaces: Input is received on the North and West interfaces, and Output is sent on the South and East interfaces. A module's interface could be represented as a tuple of interfaces: (west; north; east; south).

Listing 5 depicts the composition scheme for Simple Interfaces (SI) and the Module's Interface (MI), as defined in the language grammar.

```
Interfaces
SI ::= nil | int | bool | float |
       string | buffer |
       (SI, SI) | (SI [] )
MI ::= (SI) | (SI;SI) | (SI;)*
```

Listing 3: AGAPIA Simple Interface and Module Interface Grammar.

As it can be observed in Listing 5, a SI can be:

- a basic type, such as `nil`, `int`, `bool`, `float`
- a stream of characters, `string`, or of bytes `buffer`
- a structure, by combining simple data types, `(SI, SI)`
- an array of the above, `SI[]`

On closer inspection, the structure of a SI is very similar to that of a JSON object.

Based on this observation, we add a new AGAPIA macro: `@JSONIFY`. The novel deployment macro will generate a JSON description, compatible with RiverIoT, from the SI, for each module in the system. Given a module with the following MI: `(int; (bool, float); buffer; nil)`, a JSON similar to the one in Listing 4 will be generated.

```
{
  "buffers": {
    "1": {
      "buffer-tokens": [{
        "token-type": "int",
        "byte-size": 4,
      }]
    },
    "2": {
      "buffer-tokens": [{
        "token-type": "bool",
        "byte-size": 4,
      },
      {
        "token-type": "float",
        "byte-size": 4,
      }]
    },
    "3": {
      "buffer-tokens": [{
        "token-type": "string",
        "byte-size": 1024,
      }]
    },
    "4": {
      "buffer-tokens": [{}]
    }
  }
}
```

Listing 4: @JSONIFY - Macro Generated JSON Example.

With the generated JSON description of the system in place, we can feed it to RiverIoT to start the fuzzing process.

4.1.2 Device Hardware Requirements

As device firmware is dependant on the hardware specifications where it should be run, we need provide a way of reporting these requirements to RiverIoT. This is required so RiverIoT will know how to emulate the correct architecture for the firmware.

To do so, we propose adding another macro, `@RESOURCES`, that uses JSON notation to define the target architecture and required resources. An example is presented in Listing 5.

```
@RESOURCES{
  [{
    "name": "ARCH",
    "type": "text",
    "text": "ARM"
  }]
}
@JSONIFY
module main
{ listen a: int } // North
{ read b: bool, f: float } // West
{
  // ..source code for program..
}
{ speak buf: buffer } // South
{ write nil } // East
```

Listing 5: @RESOURCES - Defining HW Requirements Example.

4.2 RiverIoT to AGAPIA

As previously states, RiverIoT makes use of an I/O specification that states how a node, from an IoT system, conveys it's communication protocol and how it relates to other IoT nodes.

By extending the AGAPIA language with the `@JSONIFY` and `@RESOURCES` deployment macros, we'll have a readily available JSON description of the IoT system under test.

As depicted in Figure 2, a horizontal composition of the modules **A** and **B** is noted as `A # B`.

A system modeled as `A # B # C`, based on the added extensions, will generate a graph representation similar to the one in Listing 6

```
{
  "iot-nodes": {
    "A": {
      "buffers": { // Buffer
          description },
      "resources": { // @Resources }
    },
    "B": { ... },
    "C": { ... }
  },
  "io-edges": {
    "1": {
      "vout": B, // Output node
      "vin": A, // Input node
    },
    "2": {
      "vout": C, // Output node
      "vin": B, // Input node
    },
  }
}
```

Listing 6: Example of Generated System Graph Description.

RiverIoT can now create a high level view of the communication that takes place inside the IoT system so it can provide guided, meaningful inputs to the devices. Having an overview of the connected nodes and edges, the framework can also mutate the graph to test the system's fault tolerance and dynamism.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

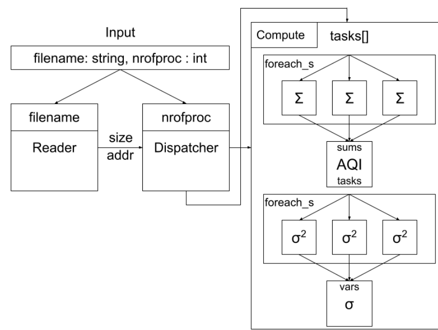

Figure 5: The AGAPIA IoT application architecture. In this figure we have the architecture of a distributed IoT application using the AGAPIA programming language. The input for this application consists of a filename that contains air pollution related information as well as the number of devices. After the Reader modules convert the data into a useful structure, it passes it to the Dispatcher module. Here the tasks are distributed to the IoT devices and after the sums (Σ), the variation (σ^2) and the standard deviation (σ) is calculated it will be printed on the screen.

5 EVALUATION: AGAPIA IoT EXAMPLE

As a real-life IoT experiment we have an example shared on our Github page. The architecture can be seen in Figure 6. In this experiment, the Air Quality Index (AQI) in Seoul was considered. We used the public Kaggle dataset[3]. This dataset contains values of 6 pollutants that were measured and averaged every hour between 2017 and 2019.

We can consider that in an IoT scenario these data are collected by IoT devices scattered around a town and they periodically communicate with a server, relaying their average measurements. We would also like to have a real-time visualization of this data so that we can announce the people or institutions in case of a hazardous event.

Even if this data set contains information spanning two years, we can consider that the IoT devices pass the information on much faster (like every minute or second). In this case, one must have an efficient way to process this information.

Using the formulas detailed in (Kanchan et al., 2015) a comparison was made between using pure MPI code and using the AGAPIA programming language. Since AGAPIA is a high-level programming language we considered that it will run slower than using a pure C/MPI approach. While we consider that the performance hit is negligible, we think that the prototyping ability of AGAPIA programming language outweighs the cons.

[3]https://www.kaggle.com/bappekim/air-pollution-in-seoul

File size(MB)	Processes	Serial(s)	MPI(s)	AGAPIA(s)
700	2	1.43	0.92	1.00
	5		0.62	1.00
1024	2	2.1	1.35	1.50
	5		0.9	1.50

Figure 6: Performance comparison between MPI and AGAPIA based on the (Kanchan et al., 2015) metrics.

For the MPI approach, a basic Scatter/Gather algorithm was used where the master (the server that has received the readings from the IoT devices) sent the averaged values of the pollutants back to the IoT devices in order to calculate the variance and standard deviation faster. In this case, each device had two responsibilities. Firstly they must gather the data and send it every minute, and secondly, when master requests, they must receive data, calculate the mean and AQI and report back to master.

When using the AGAPIA programming language, these two responsibilities were merged. Each IoT device in our cluster is represented by a module, that can read and process the data as it comes into the IoT device. In this way, it is easier to think of each IoT device as a composable module and when needed, the master can change the module that runs on a range of devices such that they can help with other computations.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes an extension of the AGAPIA programming language with IoT-aware deployment macros that enables a fast and simple integration with a testing framework, such as RiverIoT. The added extensions generate a JSON specification of the entire system under test (SUT), describing both the graph of connected IoT devices, as well as their I/O communication interfaces. By doing so, it provides the testing framework with a high level view of the entire system, as well as the expected input and output of each node. The generated specification is read by the orchestrating component of RiverIoT which decides if it's going to perform input generation or graph mutations for the SUT. Overall, the paper explores how small additions to existing programming languages can improve the testing and validation of IoT systems.

ACKNOWLEDGMENT

This work was supported by a grant of Romanian Ministry of Research, Innovation and Digitization

UEFISCDI no. 401PED/2020.

REFERENCES

Banu-Demergian, I. T., Paduraru, C., and Stefanescu, G. (2013). A new representation of two-dimensional patterns and applications to interactive programming. In *International Conference on Fundamentals of Software Engineering*, pages 183–198. Springer.

Boshernitsan, M. and Downes, M. S. (2004). *Visual programming languages: A survey*. Citeseer.

Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., and Zhang, K. (2018). Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*.

Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. (2020). Halucinator: Firmware rehosting through abstraction layer emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1201–1218.

English, K. V., Obaidat, I., and Sridhar, M. (2019). Exploiting memory corruption vulnerabilities in connman for iot devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 247–255. IEEE.

Feng, B., Mera, A., and Lu, L. (2020). P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1237–1254.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Ghimis, B., Paduraru, M., and Stefanescu, A. (2020). River 2.0: an open-source testing framework using ai techniques. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next-Generation Testing*, pages 13–18.

Gopinath, R. and Zeller, A. (2019). Building fast fuzzers. *arXiv preprint arXiv:1911.07707*.

Hernandez, G., Arias, O., Buentello, D., and Jin, Y. (2014). Smart nest thermostat: A smart spy in your home. *Black Hat USA*, (2015).

Kanchan, K., Gorai, A., and Goyal, P. (2015). A review on air quality indexing system. *Asian Journal of Atmospheric Environment*, 9:101–113.

Leonardo, P. (2013). Child programming: an adequate domain specific language for programming specific robots.

Li, S., Da Xu, L., and Zhao, S. (2018). 5g internet of things: A survey. *Journal of Industrial Information Integration*, 10:1–9.

MIPIAlliance (2020). Mipi white paper: Enabling the iot opportunity. Technical report, MIPIAlliance.

Organization, O. (2020). Profile m – release candidate. Technical report, ONVIF Organization.

Paduraru, C. (2015). Research on agapia language, compiler and applications. *Ph. D. dissertation*.

Paduraru., C. and Melemciuc., M. (2018). An automatic test data generation tool using machine learning. In *Proceedings of the 13th International Conference on Software Technologies - Volume 1: ICSOFT,*, pages 472–481. INSTICC, SciTePress.

Paduraru, C., Melemciuc, M.-C., and Paduraru, M. (2019). Automatic test data generation for a given set of applications using recurrent neural networks. In van Sinderen, M. and Maciaszek, L. A., editors, *Software Technologies*, pages 307–326, Cham. Springer International Publishing.

Paduraru, C., Melemciuc, M.-C., and Stefanescu, A. (2017). A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1857–1863.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2020). Optimizing decision making in concolic execution using reinforcement learning. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 52–61.

Paduraru, C. I. (2014). Dataflow programming using agapia. In *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, pages 87–94. IEEE.

Paduraru, C. I., Cristea, R., and Staniloiu, E. (2021). Riveriot - a framework proposal for fuzzing iot applications. *International Conference on Software Engineering ICSE 2021, Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, page to appear.

Paduraru, C. I. and Stefanescu, G. (2020). Adaptive virtual organisms: A compositional model for complex hardware-software binding. *Fundamenta Informaticae*, 173(2-3):139–176.

Pham, V.-T., Böhme, M., and Roychoudhury, A. (2020). Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE.

Stefanescu, G. and Paduraru, C. I. (2016). Self-assembling heterogeneous interactive systems. In *Proceedings of the International Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture*, pages 1–7.

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. IEEE.

Wurm, J., Hoang, K., Arias, O., Sadeghi, A.-R., and Jin, Y. (2016). Security analysis on consumer and industrial iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524. IEEE.

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 *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114.