



University ca' Foscari di Venezia  
Department of Environmental Science,  
Informatics and Statistics

Master's course in Software Dependability and Cyber Security

Thesis work

**Tainted flow analysis and propagation  
across interfaces of IoT ecosystem**

Supervisor:  
**prof. Agostino Cortesi**

Candidate:  
**Yuliy Khlyebnikov**

Academy year 2018-2019

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
<b>2</b>	<b>Related work</b>	<b>11</b>
2.1	State of Art in the IoT Security . . . . .	12
2.1.1	IoT security, general case study . . . . .	12
2.1.2	IoT security, authentication threats . . . . .	14
2.1.3	IoT security, communication threats . . . . .	16
2.1.4	IoT security, static analyses as possible solution	17
<b>3</b>	<b>Preliminaries</b>	<b>19</b>
3.1	Data-Flow analysis . . . . .	19
3.1.1	Tainted Flow analysis . . . . .	22
3.2	IoT ecosystem, case study scenario . . . . .	24
3.2.1	Illustrative scenario . . . . .	25
3.2.2	IoT back-end . . . . .	26
3.2.3	Servlet . . . . .	27
3.2.4	Front-end . . . . .	29
<b>4</b>	<b>Cross-interface taint analysis</b>	<b>32</b>
4.1	Formal model . . . . .	32
4.1.1	Program . . . . .	33
4.1.2	Communication medium . . . . .	34
4.1.3	IoT Ecosystem . . . . .	34
4.1.4	Data source and sink . . . . .	35
4.1.5	Julia functionalities and final algorithm . . . . .	35
4.2	Implementation . . . . .	37
4.3	Plant monitoring system analysis . . . . .	38
4.3.1	Tagging process . . . . .	39

4.3.2	Combining the analysis . . . . .	41
<b>5</b>	<b>Experimental results</b>	<b>43</b>
5.1	Communication Channel: Cloud (Firebase) . . . . .	44
5.1.1	Doorbell . . . . .	44
5.1.2	Electricity Monitor . . . . .	45
5.2	Communication Channel: NFC . . . . .	47
5.3	Communication Channel: Bluetooth . . . . .	48
5.4	Robocar . . . . .	51
<b>6</b>	<b>Conclusions</b>	<b>53</b>
6.1	Experimental work . . . . .	54
6.2	Future work . . . . .	55
6.2.1	Cross-program argument type control . . . . .	55
6.2.2	Cross-program constant value propagation . . . . .	57
6.3	Learned skills . . . . .	57

# Acknowledgments

Before starting this document, I would like to express my gratitude to all professors of Computer Science department of Ca' Foscari.

Thanks you, I have had a good opportunity to enrich my IT skills, participate in different projects and improve my teamwork. Heartfelt thanks *A. Cortesi, P. Ferrara and A. Mandal* for giving me an occasion to take part in this research work and gain insights into the Static Analysis field.

Thanks professor *A. Albarelli* for giving me opportunity to take part in PID group and meet and help a lot of Venice/Rovigo companies.

Heartfelt thanks my mum, stepfather, cousin for support me both, morally and materially during this 2 years of my master course.

Finally, thanks to all my classmates, for a great study together.

# Abstract

Internet of things is the network extension consisting of lots of physical objects which integrates various sensors and a software. A modern IoT ecosystem still comprises lots of security, privacy and data consistency threats. They are due to various reasons and in particular *Cross-program propagation of tainted data* which has been also listed in the OWASP IoT top 10 most critical security risks. When interactive programs run on distinct devices (like in IoT systems), they are possibly written in a different programming languages and communicate over different channels. Standard taint analyses detect if an un-sanitized value (e.g., properly escaped) coming from a source (e.g., methods retrieving some user input or sensitive data) flows into a sink (e.g., methods executing SQL queries or sending data through Internet) within a program. In this work we enhanced the existing static analysis mechanism for taint analysis to support the interactive multi-program system. The proposed framework has been implemented with a JuliaSoft static analyzer. Preliminary experimental on randomly chosen Github projects demonstrates the effectiveness of our approach by detecting serious vulnerabilities which are not getting discovered when analysis kept in isolation.

**Keywords** IoT Security, Static Analysis, Taint Analysis, Cross-Interface Taint Analysis

# Chapter 1

## Introduction

**The Internet of Things(IoT)** - is an extension of a network that consists of physical objects(*things*) that are integrate with various sensors and a software. It allows people to be up to date about the state of different data. These exchanged data can assume various types such as simple measurements( *temperature, humidity, wind direction*) that are reported by a variety of devices such as stationary, mobile and wearable sensors to more complex inputs such as multiple streams at time. This provides a huge benefit towards attaining industry 4.0 by facilitating a sophisticated mechanism for gathering and utilizing the generated information. Besides things, a typical IoT system is also comprised of gateways that communicate through a network to an enterprise back-end server. The figure 1.1, proposed by Eclipse, highlights a specific IoT scenario that contains three main parts: (1) *things*, (2) *gateways*, (3) *cloud platform*.

**Security, privacy and data consistency threats** are still a huge issue for a modern IoT ecosystem. With exponential expansion of new involved interconnected devices(*smart-locks, wearable, small and big appliances, car-keys*) the situation becomes more problematic. A report by the biggest cyber security software company, *Semantic* in [1], the number of attack on the IoT field grew from 6000 to 50000 between 2016 and 2017. Bellow are several examples of cyber security threats:

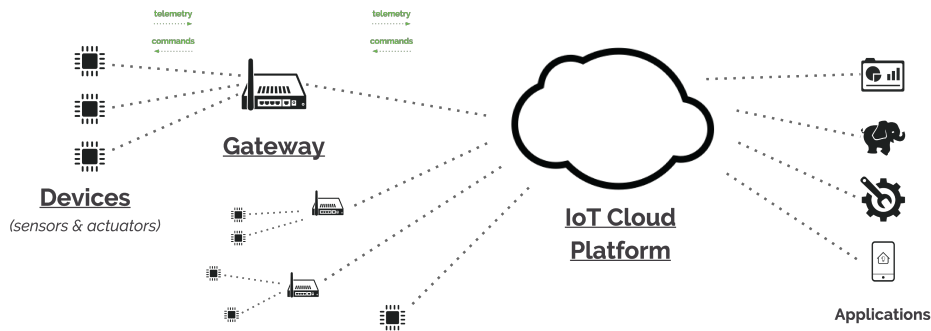


Figure 1.1: Generic IoT System

- **LocationSmart:** As reported by *Techrepublic* in [2], this threat compromised the user's smartphone location allowing everyone with no need of prior authentication to get the exact location of mobile phones in the US and Canada;
- **Jeep CAN bus hack:** is another IoT vulnerability in [14], discovered by a group of researchers on Jeep's CAN bus. It allowed an attacker to get control (*e.g. speed up or slow down, even veer of the road*) of the Jeep vehicle through so called *Sprint* cellular network;
- **Mirai botnet:** infected IoT components, as reported by *Csoonline* in [15], based on ARM processors, with a malware which turned them to the malicious network called botnet to be able to organize a large-scale DDoS attack;

**All these threats due to** the absence of security standards but also due to unstable software running over embedded devices. It actually becomes more involved process to debug and discover the code issues by the device's manufacturer. The major bottleneck to archive stable software is the lack of crucial tools and techniques for analyzing vulnerabilities. Another important and weak point of an IoT network are interconnections which allows sharing data among things. The security and integrity of an IoT system is challenged at multiple stages, as data flows through many devices, networks, and administrative boundaries. For this reason,

OWASP highlighted "insecure ecosystem interfaces" as a major security threat in OWASP IoT Top 10 2018 in [17]. Although there are a huge number of baselines in order to mitigate vulnerabilities most of them are not mandatory and thus not applied by manufacturer of an IoT device. Since the IoT is comprised of multiple heterogeneous components it is difficult to attain security with a single solution. Other at risk parts include insecure web, back-end API, cloud storage and mobile interface that all play a crucial role in comprising the device or it's related components. The optimal solution is a holistic framework that assess all security vulnerabilities.

Traditional testing and debug analysis often does not reveal possible security and privacy issues in IoT software. For this reason the usage of some automatized tools, such as static code analyzers becomes of vital importance. In particular a such analyzers are carried out within the implementation phase of Code Development Lifecycle. They allows a source code inspection against a range of properties to determine behavior of a program without executing it. Nowadays, on market there are a lot of both online an offline sound static analyzers like *Julia for Java*, *CodeSonar for C/C++*, *AlphaScan for JavaScript*, *Rails*, *dot NET*, etc. Although all of those tools are able to perform an efficient source code test, they are not suitable for IoT purposes. In fact, an IoT ecosystem, often consists of multiple components, executing independently and an isolated code analysis offered by all of the tools above is not sufficient. This makes it difficult to discover the vulnerabilities it experiences as it interacts with other programs.

**The main goal of this work** is to provide a technique to enhance the IoT taint analysis. In particular, devise a cross-interface taint checker which is capable of detecting security vulnerabilities of multiple related program having some common communication interfaces. In general, taint analysis allows to discover if an unsanitized value (*e.g., properly escaped*) coming from a source (*e.g., methods retrieving some user input*) flows into a sink (*e.g., methods executing SQL queries*) within a program. However this taint analysis on the cross-interface system represents the critical point and



has been also listed in *OWASP Top 10* at [17]. We addressed this issue by defining a cross-interface taint analysis mechanism based on the formal aspect of the data flow approach, presented in [5].

**In order to reach our goals** we first of all have studied the case by applying some of the data-flow analysis technique, introduced by professor *A.Cortesi* during the lectures. We have also analyzed already existing solutions, such as Julia static analyzer [18], which has been actually extended with new features. In particular, we used the Julia to construct a set of source and sinks for each program of the given IoT system. Then analyzed sets are used to detect taint paths for every program. Finally, the obtained result is parsed to determine the connectivity in the tainted paths among the interactive programs to construct the tainted graph. To apply and test the devised mechanism, we used an example plant monitoring IoT ecosystem developed by *Amit Mandal*. Where tainted path from IoT back-end to the front-end user application has been discovered.

Additionally, we performed a set of analysis of an IoT android based systems taken from GitHub repositories. We have grouped the tested projects on communication channel criteria and reported the detailed results on a Chapter 5. However, by providing programs of each single IoT component and annotated sources and sinks, we've got possible paths representing the taint data propagation. Again, such an approach appears to be a powerful technique to discover a lot of security issues *e.g* *SQLi*, *XSS*, *etc* during the system development phase.

The structure of the document is composed in 5 chapters as follows:

1. *Related work*
2. *Preliminaries*
3. *Cross.interface taint analysis*
4. *Experimental results*
5. *Conclusions*

In which I'll apply all of techniques learned during my master's Cyber security courses, such as Software correctness, security and reliability, Cloud computing, advanced data management and programming. Illustrate a detailed information to the problem definition, solution and discuss of obtained results. Finally conclude with the future use-cases and possible extensions of this work.

## Chapter 2

### Related work

IoT ecosystems have become, since last decade the most popular and comfortable technology in almost all the areas. As *IoT-Analytics resource* points out in [19], still nowadays there is a strong growth in sectors like: *connected industry* - 22%, *smart cities* - 20%, *connected cars* - 13%, *smart agriculture* - 6%, *connected buildings* - 5%, *connected health* - 5%, *smart retails* - 4%. This interconnects a different business giving us new opportunities but from technical point of view it's also a starting point for the diversity of devices which are capable to process and exchange data among them. The main reason for this diversified adaption of IoT is that it's closely related to other computing paradigms from wireless sensor network (*similar core concepts*) to edge computing and cloud computing technologies. Thus it allows to introduce some of the advanced techniques in order to collect information, manage physical resources (*e.g., inventory*) and logistics, remote work. It's necessary to mention that beside all of the facilities of IoT, it also inherits the security concerns of those technologies. Moreover, apart to the existing vulnerabilities, it also posses some unique security threat due to its own system architecture. As reports the "*Securelist*" resource in [20], one of the biggest antivirus software companies - "Kaspersky", the most popular attacks on IoT components are against Telnet passwords - 77.40%, and what concerns downloading malware preferred option is one of the *Mirai* family - 20%. But not only, there are different smaller cases where an intruder is able to introduce an attack and event completely impersonate of the

Thing. Recently, there were proposed lots of mitigation methods and literature to address these issues. In the following sections, I would like to gain insight into different IoT's security areas, discover weak points and analyze possible solution which are already available or not on the real world IoT scenario.

## **2.1 State of Art in the IoT Security**

In this section I will point out the actual state of the *Internet of Things's* technologies, underlining the security impact, basing my research on already available scientific literature.

### **2.1.1 IoT security, general case study**

*Frustaci et al.* [1], provided general definition of an IoT ecosystem based on three layers *perception, transportation, application*. Specified the related threats to each layer and highlighted the weakest one. Moreover, *KHATTAK et al.* [8] performed detailed analysis of the perception layer, by describing it's main components, identifying possible security attacks and available counter measurements. They further presented the security properties(*e.g. authentication, confidentiality, integrity, non-repudiation, etc*) and standards in order to archive the goals of modern IoT system. *GE, Mengmeng et al.* [2], proposed their formal definition of the framework for accessing the security of IoT, by showing how an intruder could be able to reach some vulnerable IoT device, through examination of potential attack paths. *ABDUL-GHANI et al.* [13], evaluated IoT reference model by extending it with a set of modern security attacks ( *e.g. Eavesdropping, Man-in-middle, Bluesnarfing, Hijacking, Dictionary Attack, UPD flood, etc* ). Moreover, provided a potential countermeasures for each IoT asset. *TWENEBOAH-KODUAH et al.* [6], evaluated the taxonomy of various system-inherent security issues responsible for exposing IoT systems to various cyber-threat vectors. They demonstrated attack scenarios on smart metering communication which shows that vulnerable IoT systems can be exploited by using crafted vectors. Beside the taxonomy based ap-

proach some literature focuses on security framework for IoT applications in general. *ASPLUND et al.* [3], presented an inter-view based study to identify the common IoT security requirements in sectors like, energy, water and health. Showing that, most of the respondents are not aware about the IoT security risks, prioritizing technology availability rather than robustness and data consistency. Similarly, *Trend Micro research* in [21], pointed out that a huge number (91%) of today's companies should improve their awareness against IoT attacks. *Das et al.* [4] performed a comparative study of different security protocols of IoT. For this purpose, they presented a taxonomy for security protocols used in IoT which includes device authentication, access control, privacy preservation etc. Whereas, *Mavropoulos et al.* [7] presented a class-based notation of the modeling language and a mechanism for transition between different models called Apparatus. However, majority of these systems are generic in nature and merely outlines the generic system vulnerabilities. It does not provide in depth analysis of major security issues such as device authentication and its exposure to network. *Hasan et al.* [9], applied Machine Learning approach to predict IoT attacks like, *Denial of Service, Malicious control, Spying*. They carried out a number of tests, which reported that, the most functional method was so called *Random Forest* in [23], with accuracy near 99,4%. However, such tests were performed on a virtual environment, neglecting some real-world factors. A study done by *LOHACHAB et al.* [10] reports the kinds of the most popular attack, executed on IoT systems - *DDoS*, underlying possible scenario, working mechanisms and eventual impact. Furthermore gave a vast panoramic of available tools(e.g. *data mining, deep learning, path identification, power spectrum analysis*) to mitigate the issue. However stressed on need of new more advanced technologies to deal with the novel *DDoS*. *Miloslavskaya, Tolstoy et al.* [11] provided a concept of *information security*, based on Big Data over the IoT field and highlighted its key features like *24×7 security coverage, holistic approach with defense-in-depth, advanced context-based analytics* in order to catch anomalies, threats with a real-time scenario.

### 2.1.2 IoT security, authentication threats

The biggest part of an IoT security issues deals with authentication mechanisms. Usually a such embedded devices includes reduced memory space, and computational capabilities. Thus making it impossible to carry out modern access control procedures or to store sufficient length protection passwords. In this regard, taking into account exploits discovered by [15], lots of threats, malware injection, data leakage were recently come out within IoT systems. *Sujatha et al.* [23] evaluated three factor authentication (*e.g. password, RFID, biometric*) and provided a testbet implementation to allow authentication with such factors. *GAMUNDANI et al.* [14] reports, most of them are due to weak password protection(80%), absence of encryption mechanisms on data transition(70%) or firmware updates(60%). However recent literature and lots of research work in IoT field has been done to address a such problem. *Hao et al.* [16] devised an end-to-end IoT device authentication which offers seamless integration of physical security with the asymmetric cryptography-based authentication mechanism. It estimates the device features by using type of intermediate nodes radio-frequency etc. Which then used in cryptographic key. The results showed better protection against various computation-based impersonation attacks. Similarly *SHAH in* [17] provided an secure authentication technique based on challenge-response mechanism and a concept of Vault (*e.g. finite set of equisized keys*). This gantee authorized access, by updating the vault over the time, even in case the key has been compromised by an intruder. They further showed relatively small energy consumption and short time response by testing the protocol on a real Arduino based, IoT system. Another result achieved in [18], with secure re-encryption mechanism. This allows to reduce computation process by sharing re-encrypted keys with proxy server which then will transform cipher text of a some IoT thing to allow it's decryption by others. *Challa et al.* [19] devised a signature-based authenticated mechanism for the IoT devices. It is simulated using NS2 and evaluated using *Burrows-Abadi-Needham* logic. Beside Taint Propagation Across IoT Ecosystem Interfaces authentication a secure network is also

essential to prevent IoT systems from cyber attacks. *Porambage et al.* [20] proposed secure two-phase authentication technique for sensor wireless network. In particular it, allows to prove the user's identity by testing the validity of his lightweight certificate and in case establish a secure connection between a such user and the sensor. *Giuliano et al.* [21] analyzes security aspects of IoT capillary networks for various IP and non-IP IoT devices and proposed an algorithm based on secure key renewal mechanism for safely accessing uni and bi-directional IoT devices. Although, they studied approaches enhances security, but majority of these methods incurs a substantial implementation complexity and imposes significant overhead to the already power constrained IoT devices. Further, the performance is also affected by the secure network channel between various IoT devices for data transmission. Beside traditional authentication methods which comprises key exchange or password/pin validation, biometric authentication has been used, since lasts years in several areas. It provides highest level of security and is more challenging to be compromised. A recent study in [22] points out that the Iris match approach can be used whitening IoT scenario, allowing to prove an identity through two-phase authentication process. *Kinikar et al.* [24] proposed an open authentication protocol *OAuth* based on authentication technique through access token stored on the server. *Farris et al.* [26] analyzed security features of network functions virtualization (NFV) and software defined network (SDN) from IoT security perspective. They also depicted the open challenges for SDN and NFV based security mechanisms for IoT. Whereas, *Shin et al.* [25] proposed a secure protocol using trust between Proxy Mobile IPv6 (PMIPv6) domain and IOT systems to address security issues. The proposed protocol supports features like: handover management, where mutual authentication, key exchange, etc.

### 2.1.3 IoT security, communication threats

The key feature of an IoT ecosystem is to be able to send measured or collected data to some end-point, where they could be elaborated. Beside a such data, IoT actuators usually requires some user's private information to be exchanged, thus it becomes important to garntee also a robust connection, encryption mechanisms to be applied in order to protect transferred data. This is another weak point of a such system, since traditional encryption algorithms requires a lot of computations which are not feasible with majority of embedded things. For this purpose *Kim et al.* [18] devised a mechanism by implementing proxy re-encryption towards managing data with comparatively less encryptions, and also a data sharing process for secure and efficient data transmission. Whereas *Sahay et al.* [27] propose a mechanism for detecting the malicious nodes responsible for the version number attack, thus preventing the flow of malicious data in the system. *Hou et al.* [28] devised a three-dimensional approach for exploring IoT security by combining the concept of IoT architectures and data life cycles. Unlike many other security measures it considers the data flow from IoT end-point devices through the Internet to a cloud or vice versa. However, it is focused on secure usage of IoT data, it does not ensure the taintedness of the data.

Beside the lack of advanced and specific cipher mechanisms, another issue is increasing spread of various thingbots(*e.g.* *VPN Filter*, *UPnProxy*, *OWARI*, *DoubleDoor*, *JenX*), which infects routers and another intermediate network nodes. *f5* public resource, investigated in [13] that the thingbots above, allows intruders to perform attacks like, *crypto-jacking*, *DDoS*, *PDos*, installing tor nodes, make packet sniffing, *crypto-jacking*. This is due to the absence of specifically devised techniques and tools which be able to recognize and block malicious botnets. To this regard there was presented some scientific literature, to gain insight the issue. *Hguyen et al.* [29] offered a botnet detection technique based Machine learning approach. They analyzed (SOHO) IoT device firmware through PSI graph and discover with use of convolution neural networks whether it contains malicious strings. The accuracy of



nearly 92% confirms that this is a good starting point address an issue. *Prokofiev et al.* [30] used statistical methods(*e.g. logistic regretion*) to evaluate the probability of an IoT device belonging to some malicious botnet. Again this method reports very good accuracy(*nearly 97.3%*) and precision(*94%*) and the capability to detect Telnet/SSH based botnet attacks. Whereas *Choi et al* [31] evaluated the DNS traffic and defined a key features to distinguish the malicious botnet DNS requests. In particular devised an algorithm which is able to identify botnet queries by using previous key features. Another allowing factor to this kind of attacks is so called hidden backdoors. Manufacturers of IoT things usually sales their devices with some opened communication ports (*e.g. Telnet, SSH, Sip, UPnP*) it makes easier the procedure of support but also allows attacks to gain additional insight into device. To this regard, *Tien et al.* [32] provided an algorithm *ShDep* which is able to analyze an IoT thing's firmware, based on various scripts and discover if there is some opened service ports(*hidden backdoor*). To this end, the graph based approach is applied in order to represent the communication among scripts. Then by traversing a path it's possible to discover telnet or other unsafe service connection.

#### **2.1.4 IoT security, static analyses as possible solution**

In order to mitigate previously discussed threats, the static program analysis in [33] (in particular data-flow taint analysis) may be very useful, since allows to discover data propagating across different systems and it's effect on the transmitted IoT ecosystem. This technique also helps make IoT software more robust and solve most security bugs by it's manufacturer, during the development phase of software life cycle. In general, there are lots of solutions based on this approach, and our technique is one of them, for example *Click et al.* [34] proposed a mechanism for conditional constant propagation based on lattice representation by defining special flow functions over the composed domain. *Pioli and Hint* [35] devised a flow function by combining constant propagation and pointer analysis. However, most of the studied approaches discussed about

propagation of some specific type of data instead of taint propagation in general. Recently, a some other static analysis approaches have been used to analyzes the security of IoT systems. *Huuck et al.* [36] discussed the uses static code analysis to detect some of these issues. Similarly *Cleik et al* [37] identified security and privacy issues of five IoT platforms, and applied existing static analyzers to detect these issues. These approaches pointed out that "a suite of analysis tools and algorithms targeted at diverse IoT platforms is at this time largely absent". The biggest company that produces static analyzers(*GrammarTech*) has also claimed in [12], that the only way to keep software, developed for IoT landscape both, design and coding error free is to adapt the mandatory use of static code analysis across development projects. A such technique offers a vast gamma of source code controls(*constant propagation, live variable analysis, taint analysis*). Further, the taint analysis should be performed over multiple programs in order to be complete, as IoT systems composed of multiple interactive components executing independently. Therefore, the current IoT security landscape demands a mechanism for analyzing the security vulnerabilities of the IoT system which facilitates cross-interface data propagation. In this regard the existing taint analysis techniques can be very useful, but they can only analyze a program in isolation. Therefore, the taint analysis should be enhanced to support the analysis of a multiple interactive programs running independently.

# Chapter 3

## Preliminaries

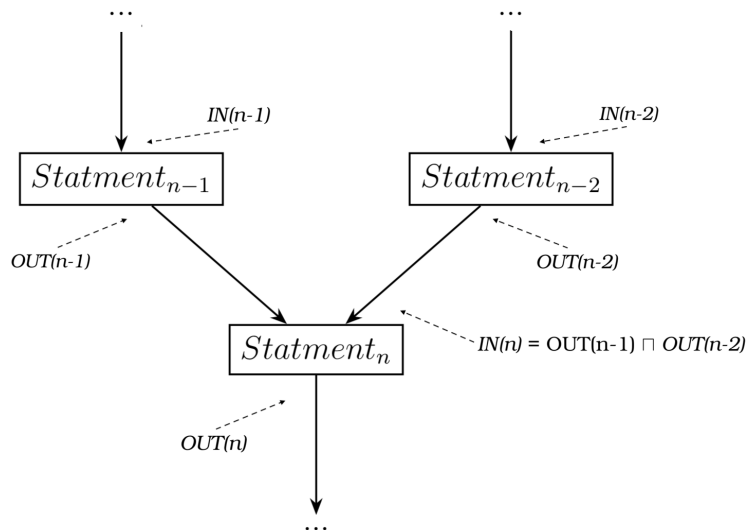
This chapter will provide the fundamentals of *Data-flow analysis*, one of the most popular technique used within a static analysis. It will show how data-flow, and in particular tainted-flow analysis may be useful in order to point out potential security issues. Moreover it will provide the role of it's components and security threats and contain an example of tainted flow analysis. The chapter will be broken down into two sections, for clarity: *3.1 Data flow analysis* and *3.2 IoT Case study*. The first section introduces data-flow and tainted flow analysis. The second section illustrates a specific scenario of IoT, discusses possible security issues and mitigation methods.

### 3.1 Data-Flow analysis

In general, *Data-Flow* analysis can be defined as a technique which allows, without an explicit execution to discover information about flow of data of a computer program. Where *data* may assume various types: constants, variables, expressions, depending on a specific instance of analysis. Usually, this technique is widely used by a compilers in order to optimize source code as well as by bug-finding tools to gather some common issues of program code. To this regard, the program is expressed in terms of a *Control Flow Graph(CFG)*. A CFG is an oriented graph where each node represents a statement and each edge shows possible control flow prop-

agation. Among the set of all vertices, entry nodes have no predecessors while exits nodes have no successors. The control flow enters through the entry node and leaves through the exit node.

To represent properties of program's data, the notion of *fact* is defined. In particular a fact characterizes the value of the property at each program point. For this purpose, on the Control Flow Graph, each node has the set of facts which are valid before entering(*IN*) and after exiting(*OUT*) such node. Moreover, since within a statement some information may be generated or killed(*e.g. no more valid*), each node has also a set of *GEN* and *KILL* facts. The following figure shows an example of a portion of CFG reporting the sets of facts:



Depending on specific instance of data-flow analysis,  $IN(n)$  and  $OUT(n)$  can be defined with respect to the  $GEN(n)$ ,  $KILL(n)$  or  $IN(n+1)$ ,  $OUT(n-1)$ , where  $n+1$  are successors and  $n-1$  predecessors respectively of the node  $n$ . However there might be a case when a given node  $n$  on CFG, has more then one predecessor/successor, so the *meet*( $\sqcap$ ) operator is applied to all *OUT* or *IN* sets of predecessors/-successors respectively. In general, for a particular data-flow analysis, *IN* and *OUT* set's definitions are resumed with a system of

equations.

The aim of the data-flow analysis is to evaluate, for each node  $n$  of CFG, the sets  $IN(n)$  and  $OUT(n)$  by iterating through previously defined system of equations, until the sets change (*not joined the fixed point*). Once the  $IN(n) = \{fact_1, fact_2, \dots, fact_k\}$  and  $OUT = \{fact_1, fact_2, \dots, fact_l\}$  has been computed for each node  $n$ , some measures to reduce code complexity or common bugs management are made. For example by applying *Reaching definition* analysis at [9] to a given source code, a lot of computations (*constant values*) may be performed even before executing the program. Since it allows to discover for each statement, what are the earlier defined statements with target variable which can be assignment to a current instruction. In this case,  $IN(n)$  and  $OUT(n)$  equations are defined as follows:

$$IN(n) = \begin{cases} \emptyset, & \text{if } n \text{ is entry node;} \\ \bigcup_{\forall s \in pred(n)} OUT(s); & \text{otherwise;} \end{cases}$$

$$OUT(n) = GEN(n) \cup (IN(n) - KILL(n));$$

Where  $pred(n)$  is the set of predecessors of node  $n$ .

Another instance of data-flow analysis is *very busy expressions* in [11]. In particular, the aim is to get for all program's points  $p$ , so called "*busy*" expressions which, no matter the selected path on CFG from  $p$ , they are used before any of their variables are redefined. This allows code checker tool to reduce code size, especially on embedded IoT devices, where there is some memory constrain. The *fact* is represented by an expression and the  $IN(n)$  and  $OUT(n)$  equations are defined as follows:

$$OUT(n) = \begin{cases} \emptyset, & \text{if } n \text{ is exit node;} \\ \bigcap_{\forall s \in succ(n)} IN(s) & \text{otherwise;} \end{cases}$$

$$IN(n) = (OUT(n) - KILL(n)) \cup GEN(n);$$

Where  $succ(n)$  is a set of all successors on CFG of node  $n$ .

The last example, I want to report is called *live variable analysis* in [10]. The main goal in this case is to determine, for each program's point  $p$ , which are the "live" variables at the exiting from  $p$ . In general the variable is said to be live, if there exists a path on CFG from some node  $k$  to a usage node  $l$  and on that path a such variable is not redefined. This technique allows to reduce again the variables set of some program, especially in case, there is a little number of register. What concerns the data-flow equations, they are defined as:

$$OUT(n) = \begin{cases} \emptyset, & \text{if } n \text{ is exit node;} \\ \bigcup_{\forall s \in succ(n)} IN(s) & \text{otherwise;} \end{cases}$$

$$IN(n) = (OUT(n) - KILL(n)) \cup GEN(n);$$

Data-flow analysis is a wildly used technique which allows code optimization, performance improving but also finding program vulnerabilities and information leakage. In the following sub-section, I would like to focus attention on one more technique of data-flow framework. In particular, *tainted flow analysis* and how it can be used to discover potential code vulnerabilities.

### 3.1.1 Tainted Flow analysis

*Tainted analysis* can be defined as a technique which tracks an un-sanitized data propagation, between a previously defined *source* and *sink* program points. In particular, the main goal of the analysis is to find out, whether there exists a path from source to sink which does not go across a sanitizer, if so mark a such path as tainted. For tainted analysis, a path from source to sink on the CFG, can be seen as a chain of data dependencies. To this regard, [16] defined taint analysis as a quadruple:

$$Tainted = (P, Src, Snk, S)$$

Where  $P$  is a program defined with CFG,  $Src$  is a set of sources, the program points where a particular data has been generated.  $Snk$  is a set of sinks, program points where a particular data is consumed. Finally  $S$  represents a set of sanitizers, might be some predefined functions or user controls of data. In general, as presented by *James Clause et al.* [38], there exists two kinds of tainted flow propagation:

- *Explicit*: which is due to the explicit value dependencies of the program variables. For example, if some variable,  $a$  contains tainted value, and then it appears within expression assigned to a variable  $x$ , this would affect  $x$  making it tainted as well:

```
a = 5; "Tainted";
x = a * 2;
```

Figure 3.1: Example of explicit tainted flow

- *Implicit*: which is due to the implicit control flow dependencies in the code. For example, let's consider again the tainted variable  $a$ , and the *if-then-else* construct, where variable  $x$  appears within both: *then* and *else* blocks. The *if* condition depends on  $a$  value. So as a consequence, also the value assigned to  $x$  will depend on tainted value of  $a$ , thus become tainted as well.

```
a = 5; "Tainted";
if(a > 2) then
  x = 10;
else
  x = 20;
```

Figure 3.2: Example of implicit tainted flow

To address the issue several implementations has been presented within the recent literature. *Cao et al.* [39] implemented tool to perform the *flow-sensitive tainted analysis* of modern version of *PhP* web applications. They used a built-in(*PhP*) feature to

construct a CFG, then discover the source node and traverse and analyze all paths starting from a such node towards a potential sinks. However the proposed solution supports only procedure-oriented fashion and need to be integrated for OOP case. Another proposal was made by *Zhang et al.* [40], where they implemented a low-overhead system for binary taint flow analysis. In particular, the system inspects a binary code stream of the program, and checks whether it contains a data used for control transfer target, if so marks the source as tainted and arises a warning.

With such technique it's possible to discover code vulnerabilities like *SQLi* [6], *Cross-Site scripting* [7], since the effect on execution of malicious payload, introduced by intruder can be pointed out during compile time. Although tainted analysis is even built-in feature of some SDKs(e.g. *tainted checking in Perl* [3], *Ruby* [4]) and some enterprise check tools are available(*Julia soft* [18]), they are devised to be used mostly with Desktop or Smartphone applications. Furthermore allows an isolated analysis, which means discovering only tainted paths within a single program. This is the major bottleneck for the IoT oriented software, where in general more programs and communication among them should be considered. That's why we aimed to device an automatized tool, which extends this technique to the IoT field. In particular, apply an already available taint checker to all components of an IoT ecosystem and combine obtained results in order to get global tainted paths.

## 3.2 IoT ecosystem, case study scenario

In this section, I will focus attention on a real world IoT ecosystem, which is used to collect environmental data(e.g. *temperature, humidity, soil-moisture*). Provide an example of the cross-interface taint propagation, and apply tainted flow analysis using one of the available solutions on the market(*JuliaSoft*), in order to point out the issue. Finally discuss about obtained results.



### 3.2.1 Illustrative scenario

In this sub-section I would like to provide a real world example of IoT ecosystem based on 4 main components: *IoT back-end*, *cloud storage*, *servlet*, *user front-end*.. The communication scenario among those components is depicted on the following figure.

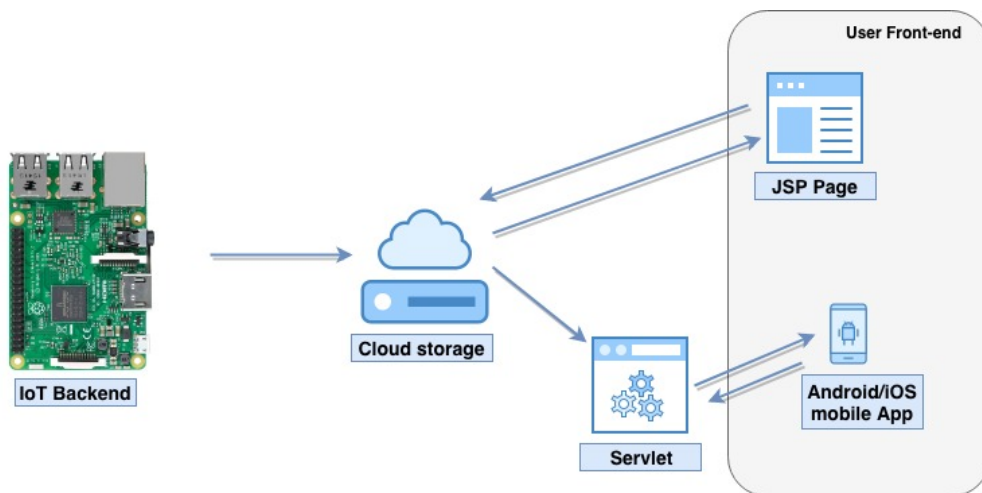


Figure 3.3: Illustrative scenario of IoT ecosystem

In particular, it's a simplified Plant Monitoring System(*thing*) with *Raspberry 326 Pi 3B+* with 64-bit, Quad-Core, Broadcom BCM2837B0 CPU 327 running at 1.4GHz and 1GB of LPDDR2 SDRAM. The thing runs a java program in order to collect measurements from two different sensors: *temperature & humidity sensor and an soil-moisture sensor*. Retrieved data is then send to a remote *hadoop* database. Which is afterwards accessed by a user front-end, through either dynamic *JSP* web page or android application. Android app is built on *Android API 25* which in turn interacts with the java servlet in order to utilize data repository. Here, java servlet is a simple servlet which reads the data from *Hbase* in [22] tables and sends them as requested by the app.

These apps are then analyzed using Julia static analyzer for taint propagation in the software [41]. In general taint analysis checks which program points can possibly be affected by the unsanitized user input. It performs tainted analysis by statically

racking explicit information flow in the program, from the selected sensitive data locations *source* into the selected leakage locations (*sinks*). In this way Julia is able to return an exhaustive report to identify potential data leaks with respect to the source and sink. In particular, the checker is based on two phases, *Init* and *Report*. Within the *Init* phase, Excel file is produced from the program under analysis. A such file contains the possible candidate points that can become leakage locations, as well as possible sensitive data source locations. These are then tagged with the category of sensitive data they retrieve, or leakage points they disclose information to. The report phase instead analyzes an application by applying the specification provided through the annotated Excel file created by *Init* phase. By the end of this phase, an exhausted report, with all possible data flow graphs representing potential un-allowed leakages, is provided. With the following sub-sections, I will show the particular task of each component and highlight possible tainted flow propagation.

### 3.2.2 IoT back-end

As mentioned before, it's based on java application. The Figure 3.4 depicts the code snippet of a such IoT back-end program. At line 13 and 19 the system performs a read of sensor data, then it makes some computation and finally adds the data to the database objects from line 27 to line 30. In the *init* phase the analyzer generates the specification file with the all possible source and sink methods. Thus, we may tag the *readline()* method as source, whereas *add()* method as sink. Afterwords, in the report phase we supplied the tagged specification. The final results report possible leakage of data between source and sink depicted in Figure 3.7, i.e the flow from the sensor reading to the storage is tainted.

---

```

1   public class Server{
2   private static Socket socket1;
3   private static Socket socket2;
4   public static void main(String[] args){
5   try{
6   int rowNo = 1;
7   ...
8   while(true) {
9   // sensor 1
10  InputStream is1 = socket1.getInputStream();
11  InputStreamReader isr1 = new InputStreamReader(is1);
12  BufferedReader br1 = new BufferedReader(isr1);
13  String msg1 = br1.readLine();
14  String[] readings1 = msg1.split("\\t+");
15  // Sensor 2
16  InputStream is2 = socket2.getInputStream();
17  InputStreamReader isr2 = new InputStreamReader(is2);
18  BufferedReader br2 = new BufferedReader(isr2);
19  String msg2 = br2.readLine();
20  String[] readings2 = msg2.split("\\t+");
21  //Compute Average of two Sensor
22  humidity =
      (Float.parseFloat(readings1[0])+Float.parseFloat(readings2[0]))/2;
23  tc = (Float.parseFloat(readings1[1])+Float.parseFloat(readings2[1]))/2;
24  tf = (Float.parseFloat(readings1[2])+Float.parseFloat(readings2[2]))/2;
25  // create an object with the measured data
26  Put p = new Put(Bytes.toBytes("row"+rowNo));
27  p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("humidity"),Bytes.toBytes(""+humidity));
28  p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("tempc"),Bytes.toBytes(""+tc));
29  p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("tempf"),Bytes.toBytes(""+tf));
30  p.add(Bytes.toBytes("soil"),Bytes.toBytes("moisture"),Bytes.toBytes(readings1[3]));
31  // insert data into the database
32  try {
33  table.put(p);
34  } catch (IOException e) {...}
35  rowNo++;}
36  ...
37  }}}

```

---

Figure 3.4: Code Snippet for IoT Backend.

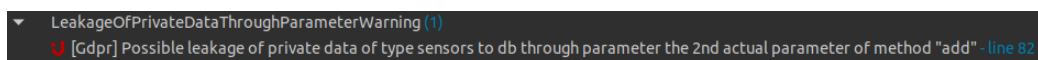


Figure 3.5: Taint Analysis Results of the Thing

### 3.2.3 Servlet

What concerns a servlet, it's again has been implemented with java(*JSP web page*). It main purpose is to perform access to the data and send them to the android application. The following figure depicts the code snippet of the servlet [24] which has been used

within the project.

---

```

1   @WebServlet("/HbaseConnection")
2   public class HbaseConnection extends HttpServlet {
3   public HbaseConnection() {
4   super();}
5   protected void doPost(HttpServletRequest request, HttpServletResponse
        response){
6   response.setContentType("text/html;charset=UTF-8");
7   try {
8   String sensType = request.getParameter("sensType");
9   String result = queryHbase(sensType);
10  response.setStatus(HttpServletResponse.SC_OK);
11  OutputStreamWriter writer = new
        OutputStreamWriter(response.getOutputStream());
12  writer.write(result);
13  writer.flush();
14  writer.close();
15  } catch (IOException e) {...}}
16  //query Hbase
17  public String queryHbase(String sensType) {
18  double val=0.0;
19  String result = "";
20  try {
21  Class.forName("org.apache.drill.jdbc.Driver");
22  Connection connection =
        DriverManager.getConnection("jdbc:drill:zk=192.168.1.6:2181/drill/drillbits1");
23  Statement st = connection.createStatement();
24  ResultSet rs1 = st.executeQuery("SELECT CONVERT_FROM(SensData." + sensType
        + ", 'UTF8') FROM hbase.SensData");
25  int count=0;
26  while(rs1.next()){
27  System.out.println("-->" + rs1.getString(1));
28  float temp = Float.parseFloat(rs1.getString(1));
29  val = val+temp;
30  count++;}
31  val = val/count*100;
32  val = Math.round(val);
33  val = val/100;
34  } catch (ClassNotFoundException | SQLException e) {...}
35  return ""+val;}}

```

---

Figure 3.6: Code Snippet for the Servlet.

In particular, at line 28 the value sent by IoT back-end is retrieved using *parseFloat()* method from the database whereas at line 12 with *write()* method it is sent through the Internet. These two methods are respectively target as source and sink in the specification file, generated within init phase of checker. Further, the tagged specification file is supplied for the report phase of the analysis. The taint analysis results depicted in Figure shows that tainted flow does exist from the identified source and sink.

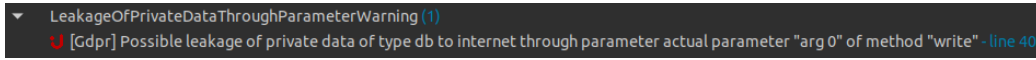


Figure 3.7: Taint Analysis Results of the Servlet

### 3.2.4 Front-end

For the user front-end, we considered an android App. It's actually consist of two different components: *BackgroundWorker*, primarily responsible for collecting data from servlet, and *Main activity*, which takes the values received by the Background Worker and perform the intended action. The Figures 3.8, 3.9 reports both code snippets: of Main Activity and Background Worker.

---

```

1     public class MainActivity extends AppCompatActivity {
2         TextView textView;
3         RadioGroup radioGroup;
4         RadioButton radioButton;
5         protected void onCreate(Bundle savedInstanceState) {
6             super.onCreate(savedInstanceState);
7             setContentView(R.layout.activity_main);
8             textView = (TextView) findViewById(R.id.tvResult);
9             radioGroup = (RadioGroup) findViewById(R.id.rgSensType);}
10        public void onClick(View view) {
11            String sensor="";
12            String type = "mul";
13            BackgroundWorker backgroundWorker = new BackgroundWorker(this);
14            String result = null;
15            int sensType = radioGroup.getCheckedRadioButtonId();
16            if(sensType!=-1) {
17                radioButton = (RadioButton) findViewById(sensType);
18                String str = (String) radioButton.getText();
19                if(str.equals("Soil Moisture")){sensor = "soil.moisture";}
20                else if(str.equals("Temperature in Celsius")){sensor = "ambiance.tempc";}
21                else if(str.equals("Temperature in Fahrenheit")){sensor = "ambiance.tempf";}
22                else if(str.equals("Humidity")){sensor = "ambiance.humidity";}
23                ...
24                textView.setText("Average "+str+": "+result);}
25            ...}}

```

---

Figure 3.8: Code snippet of Android App (Main Activity).

If we perform the analysis of a such android application, with the *readLine()* method of Background Worker at line 19 as a source and *setText()* method of the Main Activity at line 24 as a sink. Then the analysis results, at Figure 3.10 shows that a tainted flow still exists among them.

---

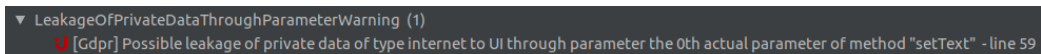
```

1  public class BackgroundWorker extends AsyncTask<String, Void, String> {
2  ...
3  protected String doInBackground(String... params) {
4      String type = params[0];
5      String temp = params[1];
6      String servletURL = "http://192.168.1.6:9080/IOTQuery/HbaseConnection";
7      if(type.equals("mul")){
8          try {
9              String result, line;
10             URL url = new URL(servletURL);
11             HttpURLConnection httpURLConnection = (HttpURLConnection)
12                 url.openConnection();
13             ...
14             BufferedWriter bufferedWriter = new BufferedWriter(new
15                 OutputStreamWriter(outputStream, "UTF-8"));
16             String post_data = URLEncoder.encode("sensType",
17                 "UTF-8")+"="+URLEncoder.encode(temp, "UTF-8");
18             bufferedWriter.write(post_data);
19             ...
20             InputStream inputStream = httpURLConnection.getInputStream();
21             BufferedReader bufferedReader = new BufferedReader(new
22                 InputStreamReader(inputStream, "UTF-8"));
23             while((line = bufferedReader.readLine())!=null){
24                 result += line;}
25             bufferedReader.close();
26             inputStream.close();
27             httpURLConnection.disconnect();
28             return result;
29             ...}
30             return null;}
31             protected void onPreExecute() {
32                 alertDialog = new AlertDialog.Builder(context).create();
33                 alertDialog.setTitle("Connection Status");}
34             protected void onPostExecute(String result) {}
35             protected void onProgressUpdate(Void... values) {
36                 super.onProgressUpdate(values);}

```

---

Figure 3.9: Code snippet of Android (Background Worker).



▼ LeakageOfPrivateDataThroughParameterWarning (1)  
 [Gdpr] Possible leakage of private data of type internet to UI through parameter the 0th actual parameter of method "setText" - line 59

Figure 3.10: Taint Analysis Results of the Android App

The performed analysis detects the tainted path from the specified source and sink for individual components. If we take a close look at these analysis results we can observe that tainted data did propagated over the Internet in the following fashion:

*sensor* → *database* → *servlet* → *Androidapp*

This gives us the complete path for the tainted data propagation across the different interfaces of IoT. However, the project we have tested comprises just a couple of components but in general an IoT system consists of a hundreds of different parts which in turn might include lots of sources and sinks. Thus making this kind of manual analysis nearly infeasible. The existing static analysis techniques efficiently detects the taint propagation for a program in isolation. But, when multiple programs running independently interacts with each other over some communication medium, the tainted path may extend to the other program modules participating in interaction, which may appear safe when analyzed in isolation. Thus, in order to get the complete flow, the existing taint analysis techniques need to be enhanced to support analysis of multiple interactive programs executing independently.

# Chapter 4

## Cross-interface taint analysis

In this section we will enhance the existing taint analysis mechanism to support the interactive multi-program system. For this propose we used the inter-procedural data flow approach discussed in [12] and [5] as the basis for our formalization.

### 4.1 Formal model

As we have discussed in *Chapter 3*, each time we are dealing with the IoT ecosystem, we actually must consider multiple components(*e.g. back-end embedded devices, cloud storage, servlets, front-end user devices*). Due to their different nature, all of them will run a program which is independent from others, is written with different programming language, runs on it's own stack and manages independently possessed resources. However that programs share some data across a communication medium(*e.g. wired network IEEE 802.3, Wi-Fi IEEE 802.11, LTE, 5G, Bluetooth, Zig-bee, Zig-wave*) in order to produce a common result. For this reason we aimed to discover possible ways, of not properly sanitized data, been propagated across the system. Thus define a formal model and a set of properties, to be able to implement a sound taint propagation analysis and run it on a real world example.

The rest of this chapter is organized as follows: at 4.1.1 we provide a formal representation of a program of IoT ecosystem, at 4.1.2 formal aspects and role of a communication medium, at 4.1.3



we define an IoT ecosystem using super-graph technique, at 4.1.4 provide a formal definition of Juliasoft static analyzer used functionalities. Finally provide the algorithm to construct the graph representing tainted paths obtained from the earlier results.

### 4.1.1 Program

What concerns a program, it can be represented as a graph  $G_p$  as follows:

**Definition 4.1.** An IoT ecosystem program  $P$  is defined as a flow graph:  $G_p = \{N_p, E_p, S_p, Ex_p\}$

Where  $N_p$  is the set of nodes corresponding to statements in the program.  $S_p$  is the set of all unique entry points of different procedures/functions. Similarly  $Ex_p$  represents the set of all unique exit points. Beside start and exit nodes, the statements of  $G_p$  can be classified into a call node ( $Call_p$ ) or a return node ( $Ret_p$ ), where  $Call_p$  represents statements which calls some other functions, whereas  $Ret_p$  represents statements where control returns to the calling site. Thus  $\{Call_p, Ret_p\} \in N_p$ . Again,  $E_p$  represents the finite set of directed edges, showing the relation among nodes in  $G_p$ . It can be defined as:

$$E_p = (S_p \times N_p) \cup (N_p \times N_p) \cup (N_p \times Call_p) \cup (Call_p \times S_p) \cup (Ex_p \times Ret_p) \cup (N_p \times Ex_p)$$

Where:

$(S_p \times N_p)$ : connects a start node to a statement within the same program  $p$ ;

$(N_p \times N_p)$ : represents an edge between statements within a program  $p$ ;

$(N_p \times Call_p)$ : represents an edge from a statement to a call site node within the program  $p$ ;

$(Call_p \times S_p)$ : represents an edge between a call-site node and a start node of another procedure within a program  $p$ ;

$(Ex_p \times Ret_p)$ : represents an edge between an exit node and a return-site node of another procedure within a program  $p$ ;

$(N_p \times Ex_p)$ : represents an edge between a statement and an exit node within the program  $p$ ;

### 4.1.2 Communication medium

IoT ecosystem is made up of multiple programs  $\{P_1, P_2, \dots, P_n\}$  running on different devices. That programs communicate with each other through some communication channel. A communication channel ( $C$ ) consists of type of communication medium, sending( $I_S$ ) and receiving( $I_R$ ) interfaces. These interfaces consist of a set of communication functions which enable a program to send ( $f^s \in I_S$ ) or receive ( $f^r \in I_R$ ) data/signal to and from the communication channel. Therefore formally it can be stated as:

**Definition 4.2.** An IoT ecosystem channel  $C$  is defined as a flow graph:  $C = \{medium, I_S, I_R\}$

Where:  $I_S = \{f_1^s, f_2^s, \dots, f_h^s\}$  and  $I_R = \{f_1^r, f_2^r, \dots, f_k^r\}$

### 4.1.3 IoT Ecosystem

IoT ecosystem consisting of various interactive programs can be represented as a directed graph  $G^*$ , connecting all the programs over the communication channel and can be defined as follows:

**Definition 4.3.** An IoT ecosystem is defined as a directed flow graph:  $G^* = \left\{ \{G_{p_1}, G_{p_2}, \dots, G_{p_n}\}, \{E_{c_1}, E_{c_2}, \dots, E_{c_m}\} \right\}$

Where,  $G_{p_i}(1 \leq i \leq n)$  is a graph, which represents the program  $P_i$  and  $E_{c_i}(1 \leq i \leq m)$  represents the edges which connect nodes belonging to different programs of the ecosystem, it corresponds to the program connection through communication channel. Programs communicating over the channel  $C$  implement the communication functions to *send/receive* data.

Thus, an edge  $E_c$  between two programs  $h$  and  $k$ , respectively  $G_{ph}$  and  $G_{pk}$  can be defined as:

**Definition 4.4.** An inter-program edge through communication channel  $C$  is  $E_c = (n_h^s, n_k^r)$ , where  $n_h^s \in G_{ph}$ ,  $n_k^r \in G_{pk}$  and  $h \neq k$

#### 4.1.4 Data source and sink

Taint analysis technique performs a check between a given pair of source and sinks, across which tainted data may propagate to the other programs. Here the source ( $\sigma_{src}$ ) and sink ( $\sigma_{snk}$ ) are the program points from where the specific data is generated or consumed. They generally are defined by the user, as annotations for some specific field (*e.g. data*) of interest for the overall system. However in ecosystem with multiple interconnected programs there might present a multiple internal sources( $\sigma_{src}^i$ ) and sinks respectively ( $\sigma_{snk}^i$ ) across the communicating interfaces, as each program executes independently. Therefore, formally sources and sinks can be defined as:

**Definition 4.5.** A set of source and sinks in an IoT ecosystem is  $\sigma_{src} = \sigma_{src}^e \cup \sigma_{src}^i$ ,  $\sigma_{snk} = \sigma_{snk}^e \cup \sigma_{snk}^i$  and  $e \neq i$

#### 4.1.5 Julia functionalities and final algorithm

In general, the taint analysis can be defined as a mechanism to check the possibility of a data (*field of interest*) being modified in an un-sanitized way. To analyze the taintedness of a program we used the Julia static analyzer in [18]. The analysis of the Julia are organized in two phases:

1. **Init:** where the analyzer takes an individual program  $G_P$  as an input and produces a "sink-configuration" ( $conf_p$ ) file. A such file consists of all possible sources( $\sigma_{src}^P$ ) and sinks( $\sigma_{snk}^P$ )

of a given program. Thus the init process ( $Analysis^i$ ) can be defined as follows:

$$(Analysis^i) : (G_p) \rightarrow conf_p \text{ where,}$$

$$conf_p = \{\{\sigma_{src}^p, \sigma_{snk}^p, tg_p\} : tg_p \neq null\}$$

2. **Report:** the second phase takes programs and "source-sink configuration" ( $conf'_p$ ) as an input and returns a tainted path among those source and sinks. Therefore formally the report phase of the analysis, ( $Analysis^r$ ) of program p can be defined as:

$$(Analysis^r) : (G_p \times conf'_p) \rightarrow R_p \in \Phi(G_p) \text{ and,}$$

$$conf'_p = \{\{\sigma_{src}^p, \sigma_{snk}^p, tg_p\} : \exists tg_p \neq null\}$$

The obtained results need to be integrated in order to get a complete trace of the tainted data propagation, for this purpose we also devised *Algorithm 1*. It thus takes interactive programs :  $\{G_{p_1}, G_{p_2}, \dots, G_{p_n}\}$ , user defined source and sinks  $\{\sigma_{src}^e, \sigma_{snk}^e\}$  and API for the communication channels  $\{E_{c1}, E_{c1}, E_{cm}\}$  as input. The algorithm first of all generates all possible sources ( $\sigma_{src}$ ) and sinks ( $\sigma_{snk}$ ), by combining the user provided sources and sinks with that of the communication APIs. After that it applies *init* phase to each program( $G_{pi}$ ) passed as input. A such phase produces the source-sink configuration file for each program( $conf_p$ ). Later *TagConfig()* procedure, tags the source-sink-configuration file, based on the generated sources ( $\sigma_{src}$ ) and sinks ( $\sigma_{snk}$ ) and returns the modified source-sink configuration ( $conf'_pi$ ). Afterwords the second phase, (*report*) of the analyzer is carried out, it provides the set of tainted paths( $R_i$ ) for each program. Then by evaluating the last sets, we discover the so called "common edges", in particular those which interconnects two different programs by relative source and sink. It thus allows us to reconstruct the overall graph and get only the reachable paths between user specified sources ( $\sigma_{src}^e$ ) and sinks ( $\sigma_{snk}^e$ ).

**Algorithm 1** Integrating Multiple Analysis

- 
- 1: **Input:**  $\{G_{p1}, G_{p2}, \dots, G_{pn}\}, \{\sigma_{src}^e, \sigma_{snk}^e\}, \{E_{c1}, E_{c2}, \dots, E_{cm}\}$
  - 2:  $\sigma_{src} \leftarrow \sigma_{src}^e \cup \left(\bigcup_{i=0}^m (f^r \in E_{ci})\right)$
  - 3:  $\sigma_{snk} \leftarrow \sigma_{snk}^e \cup \left(\bigcup_{i=0}^m (f^s \in E_{ci})\right)$
  - 4:  $conf_{pi} \leftarrow Analysis^i(G_{pi})$
  - 5:  $conf'_{pi} \leftarrow tagConfig(conf_{pi}, \sigma_{src}, \sigma_{snk})$
  - 6:  $R_i \leftarrow Analysis^r(G_{pi}, conf'_{pi})$
  - 7:  $CommonEdges ::= \bigcup_{k=0}^m \{(n_i^s, n_j^r) \in E_{ck} : n_i^s \in R_i, n_j^r \in R_j\}$
  - 8:  $overallGraph ::= \bigcup_{k=0}^m \{(R_i, R_j) : \exists (n_i^s, n_j^r) \in CommonEdges : n_i^s \in R_i, n_j^r \in R_j\}$
  - 9:  $reachable ::= \{(\sigma_{src}^e, \sigma_{snk}^e) : \exists path(\sigma_{src}^e, \sigma_{snk}^e) \in overallGraph\}$
  - 10:  $taintedGrapg ::= \Pi_{reachable(\sigma_{src}^e, \sigma_{snk}^e)}(overallGraph)$
  - 11: **procedure** TAGCONFIG( $conf_{pi}, \sigma_{src}, \sigma_{snk}$ )
  - 12:     **if**  $(\sigma_{src}^{pi} \in \sigma_{src} \wedge \sigma_{snk}^{pi} \in \sigma_{snk})$  **then**
  - 13:          $tg_{pi} \leftarrow "ti"$  where  $tg_{pi} \in conf_{pi}$
  - 14:     **end if**
  - 15:     **return**  $conf_{pi}$
  - 16: **end procedure**
- 

## 4.2 Implementation

The mechanism discussed above, for cross-interface taint analysis has been implemented on Julia static analyzer, which is a commercial static analyzer for java and .NET bytecode, based on abstract interpretation. It currently features 45 different checkers such as: *Nullness Checker, Injection checker etc.* The taint analyzer of Julia has been widely utilized to detect various security vulnerabilities such as SQL injections and XSS as well as to the detection of leakages of sensitive data. The Injection checker can be instrumented with additional sources and sinks by adding specific Java annotations to the analyzed code . We utilized this analysis mechanism to develop our own "*cross-interface taint checker*". Figure 4.1 depicts the working principle of the devised checker where the number indicates the order in which different tasks are performed. Where it

first generates the possible set of source and sinks from the given program in the form of an Excel file. These are then tagged using the communication channels API's. This tagged sources and sinks are then send to the analyzer along with the program for the taint analysis. Afterwords the analyzer provides detail information about tainted paths in a XML file. This XML file later parsed to determine the connectivity in the tainted paths of interactive programs which helps in construction of tainted graph.

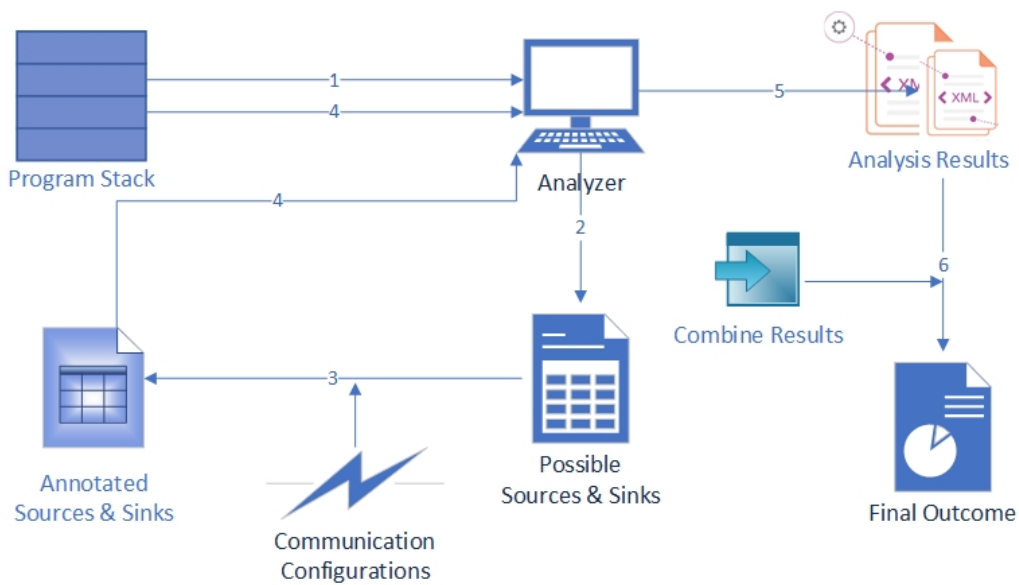


Figure 4.1: Working principle of devised technique

### 4.3 Plant monitoring system analysis

In the section 3.2 we illustrated the real world example of the IoT ecosystem and possible taint propagation by applying the manual cross-interface tainted flow analysis. However in real world systems it will be very difficult to investigate tainted data propagation across the different interfaces manually. To automate the detection of cross-interface taint propagation, in section 4.1 we devised a mechanism which allows to detect the tainted path across the

interfaces for a given communication channel. In order to demonstrate the functionality and feasibility of the devised approach we analyzed the same example used in Section 3.2. To this regard I would like to discuss, in the following sub-sections about the process of analysis through our algorithm.

To start analysis, we first of all passed the jar files corresponding to the IoT back-end [25] *HbaseStorageRPI.jar*( $G_{p1}$ ), Servlet [24] *IoT-Query.jar*( $G_{p2}$ ) finally Android application [26]  $G_{p3}$  to the devised algorithm. Moreover, it also requires the sources and sinks (*external and intermediate*) and communication media as an input. In particular, for the tested project the external source is Reading sensor data ( $\sigma_{src}^e$ ) whereas the the external sink is displaying it on application UI ( $\sigma_{snk}^e$ ). What concerns the communication channel API's for sending( $f_i^s$ ) and receiving( $f_i^r$ ) data from IoT back-end to database ( $E_{c1}$ ) and database to android app ( $E_{c2}$ ) are considered as the intermediate sources and sinks respectively.

The IoT back-end stores measurements into *HBase* database and Servlet retrieves them and sends to the Android Application over the internet. Therefore the *Hbase* database is the medium of communication ( $E_{c1}$ ) between IoT back-end ( $G_{p1}$ ) and the Servlet ( $G_{p2}$ ). Further, the data read by the Servlet is sent to the App over the Internet. Thus, Internet is another communication channel between the Servlet and the application.

### 4.3.1 Tagging process

In order to explain, in detail the execution of the analysis, I will refer to the Figure 4.2 which highlights, by using *YED graph* the tainted data flow across programs.

The init phase of the analysis ( $Analysis^i(G_{p1})$ ) generates the source-sink configuration ( $conf_{p1}$ ) for the IoT back-end program, where the method *readline()* at line 57 of *Server.Java* is considered by the user as external source ( $\sigma_{src}^e$ ). Environmental measurements of IoT back-end is then sent to the *Hbase* storage. For this purpose, the method *add()* at line 80 – 83 of *Server.Java* is used and can be easily identified as the intermediate sink ( $\sigma_{snk}^e$ ). The process of tagging of the intermediate sources and sinks in  $conf_{p1}$  has been

automated by considering the communication channel ( $E_{c1}$ ) procedures responsible for storing data in *Hbase* and matching them with the possible sinks. Thus the method *add()* is target as the intermediate sink. The result, target configuration ( $conf'_{p1}$ ) file, along with  $G_{p1}$  are passed to the report phase of checker to get the tainted path(s) ( $R_1$ ) between source and sink.

Moving to the servlet ( $G_{p2}$ ), it receives the data from the *Hbase* database and sends it to the Android App ( $G_{p3}$ ) over the Internet  $E_{c2}$ . Here, the source and sink both are intermediate, facilitated by the communication *Hbase* and Internet respectively. The init phase ( $Analysis^i(G_{p2})$ ), produces the the source-sink configuration ( $conf_{p2}$ ) with empty tags. During the tagging process, the method *getString()* at line 69 of *HbaseConnection.java* in the servlet program has been considered as an intermediate source ( $\sigma^i_{src}$ ). In particular it's responsible for reading the data from the *Hbase* database  $E_{c1}$ . Similarly, for the intermediate sink the tagging process matched the method used for sending data over the Internet in the generated source-sink configuration ( $conf_{p2}$ ) file. To this regard the method *write()* of [*java.io.Writer*] class has been tagged as the intermediate sink ( $\sigma^i_{snk}$ ). This tagged source-sink configuration ( $conf'_{p2}$ ) file as well as the Servlet program ( $G_{p2}$ ) were analyzed in report phase ( $Analysis^r(G_{p2})$ ) for the tainted paths  $R_2$ .

Finally, what concerns the Android application ( $G_{p3}$ ), it monitors the sensor data in real-time, for this regard it interacts with the servlet ( $G_{p2}$ ). In particular, app receives data from the servlet over the internet ( $E_{c2}$ ) (*intermediate source*) and displays it locally within the app (*user defined external sink*). As usual, first the source-sink configuration file ( $conf_{p3}$ ) is generated within init phase of the analyzer. Afterwords, the tagging process identifies the method *readLine()* of [*java.io.BufferedReader*] class at line 52 of *Backgroud-Worker.java* as intermediate source ( $\sigma^i_{src}$ ) by matching the generated source with the methods responsible for receiving data from the input stream of HTTP ( $E_{c2}$ ). Whereas the method *setText()* of [*android.widget.TextView*] class is identified as the external sink  $\sigma^e_{snk}$  by the user and tagged accordingly. The report phase of the analysis provides the tainted path  $R_3$  between the identified source and sink.



### 4.3.2 Combining the analysis

Report phase of Julia analyzer produced tainted paths  $(R_1, R_2, R_3)$  for different programs  $(G_{p1}, G_{p2}, G_{p3})$ . A such paths are then combined by checking if  $\text{sources}(\sigma_{src})$  and  $\text{sinks}(\sigma_{snk})$  belong to the same communication channel  $(E_c)$  in order to generate the *overall graph*. Afterwards a reachability analysis of the various nodes in the overall graph is performed and the reachable portion of the overall graph is projected as the tainted graph. The figure 4.2 reports the tainted paths grouped together and connected to each component by using the inter-program edges which corresponds to the communication channels. In particular, the yellow square box represents the nodes in the tainted path which involves in some operation with/using the data coming from the source. The thin arrowed lines, on the figure represents edges of tainted path within the same program.

For instance, the tainted graph of IoT back-end points out that at line 57, there is a reading of sensor's value and it's marked as tainted. Afterwards it propagates through lines 58 – 60 to reach the sink at line 82 of the original source code. Here data is stored in the database which is then retrieved by Servlet. This inter-program iteration over the communication channel is shown as a thick edge with the communication channel name  $(DB)$  as label. The servlet, gets values at line 69 of the source code which them propagates to statements at line 63 and line 83 respectively. Finally at line 83 tainted data are first passed to the method *doPost()* line 35 and then transmitted to the Android Application, at line 40 over the internet. This lats communication is represented again with thick edge labeled *Internet*. This transmitted tainted data is received by the Android App in the *doInBackground* class at line 52 of the source code. Inside the application the tainted data retrieved at line 52 is then used at lines 53 and 59 where reaches the user interface of the app  $(\sigma_{snk}^e)$ .

The Figure 4.2 clearly reports us that the tainted IoT back-end sensor's data propagates to the front-end user application across two different communication channels.

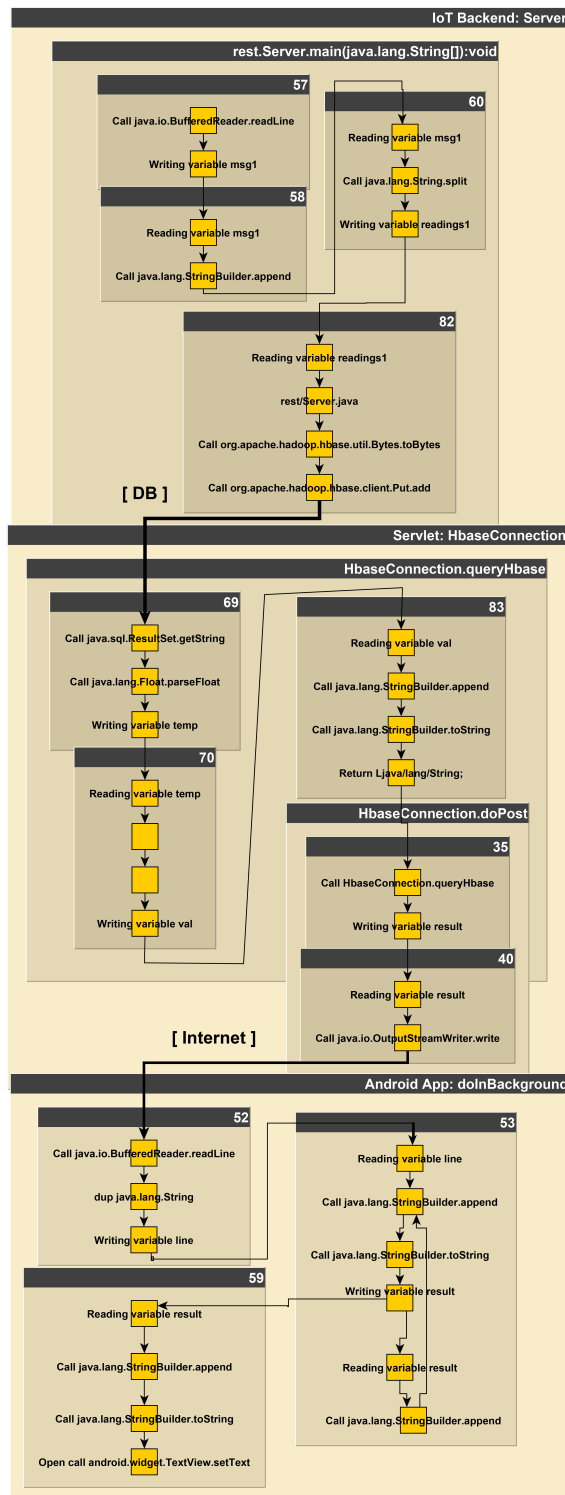


Figure 4.2: End-to-End taint propagation of the Plant monitoring system

# Chapter 5

## Experimental results

In this Chapter I would like to point out some results we got by testing devised cross-interface taint checker on a couple of IoT projects, picked up from GitHub repositories. In particular IoT projects based on Android Things, where each component (*Things and Android App*) interacts through different communication channels over some network. IoT Thing based on android facilitates building Applications on hardware platforms such as, Raspberry Pi, where, the Android Things console helps in building and delivering the Software images to target devices. It's necessary to mention the advantage of the Board Support Package (*BSP*) of Google, which in turn gives a trusted platform for developing Apps with standard updates and fixes. This makes it the most popular development platform in IoT field. In addition, it also supports advanced connectivity over cloud, bluetooth and Near Field Communication (*NFC*) among the different components of the IoT system. To this regard, we selected the projects that should satisfy the following criteria: *they must have at-least an Android or Web Application along with the Things App*. A such selection narrowed down the available repositories to a very small number. Among them majority of the repositories are functionality wise repetitive and many did not compile because of missing resources, or incorrect Gradle Build files.

The rest of the chapter is organized on 4 sections. Where we report tainted path(s) discovered by our mechanism on some repositories (*IoT systems*), categorized on the communication channel which

facilitate inter App interaction. In particular, the section 5.1 illustrates some example of tainted analysis on the IoT systems with cloud based communication channel, whereas 5.2 an example of system based on NFC communication and finally 5.4 we provide a safe IoT system.

## 5.1 Communication Channel: Cloud (Firebase)

In this sub-section we report results of testing two GitHub projects namely: *Doorbell* [27] and *Electricity Monitor* [28]. In particular, both repositories represents Android IoT applications, based on cloud communication channel - "Firebase" by Google. Actually there exists a thousands of IoT projects which uses cloud services as medium of communication between the thing and the Android Application.

### 5.1.1 Doorbell

This repository represents an IoT "smart" doorbell system. In particular, it allows to capture the image of the user who is pressing the bell button. The obtained image from the Android thing camera is then processed using *Google's Cloud Vision API*. Afterwords the image, Cloud Vision annotations and metadata are uploaded to a Firebase database. This data in the Firebase can be finally accessed by the companion Android App.

In order to perform analysis, we provided intermediate sources and sinks represented by method signatures for sending and receiving data from the Firebase. Further, we provided the external source (*acquireLatestImage()* of [android.media.ImageReader]) class and sink (*load()* method of the [com.example.androidthings.doorbell.GlideRequests]) class for the system. As showed before, the init phase of the analysis generates the source-sink configuration files which are then tagged. In particular, *acquireLatestImage()* (of [android.media.ImageReader] class) within *DoorbellActivity* class (line 162 of the source code) responsible for acquiring the

image from the camera module is tagged as external source. The *putBytes()* method of class [com.google.firebase.storage.StorageReference] (used at line 181 in DoorbellActivity) is target as sink by the automated tagging process facilitated by the communication channel method signatures. What concerns the android application, *getReferenceFromUrl()* method of [com.google.firebase.storage.FirebaseStorage] class which is used at line 88 in DoorbellEntryAdapter class, is intermediate source. Whereas *load()* method of the class [com.example.androidthings.doorbell.GlideRequests] used at line 91 of the source code of DoorbellEntryAdapter has been passed as the external sink. The programs along with the tagged source-sink configuration files are then analyzed for the report phase. The Figure 5.1.1 represents the tainted data propagation for the Thing and the Android Application. Additionally, the result of the reachability analysis after combining the tainted graph using the devised mechanism shows that tainted data did propagated from the thing to the Android App over the Firebase communication medium.

### 5.1.2 Electricity Monitor

The second project represents an IoT system which is used to track the unavailability of the electricity and notifies the user about the same as a notification by an Android App. Similarly to the Doorbell, it uses the Firebase as the medium of communication between the Thing and the Android App.

As usual, we started the analysis of the system by providing methods used for sending and receiving data from Firebase, as a set of intermediate source and sinks to the analyzer. We also provided the *electricityLog* object of [za.co.riggaroo.electricitymonitor.ElectricityLog] class as external source and *timeOn* members of the [za.co.riggaroo.electricitymonitor.ElectricityViewModel] class as the external sink. After that, the init phase generated the source-sink configuration, which has been tagged as follows: the source provided before has been target as external one, whereas the sink has been tagged by matching the the source-sink configuration with the sinks of the Firebase, send/receive methods. Further, the *set-*

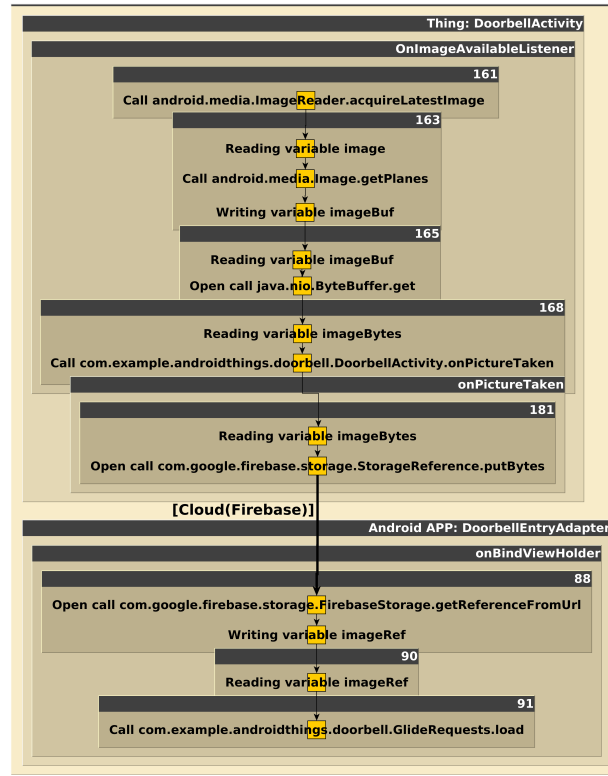


Figure 5.1: End-to-End taint propagation for cloud based communication channel (DoorBell)

*Value()* of `[com.google.firebase.database.DatabaseReference]` class used at line 41 within `ElectricityMonitorActivity`, is tagged as the intermediate sink. For the android application, the *getValue()* method of `[com.google.firebase.database.DataSnapshot]` class used at line 74 within `OverviewPresenter`, has been tagged by the automated process as intermediate source. Finally, *timeOn*, *timeOff* members of the `[za.co.riggaroo.electricitymonitor.ElectricityViewModel]` class has been set as the external sink. All the programs along with tagged source-sink configuration have been then analyzed in the report phase of Julia checker. The analysis generated us end-to-end taint propagation graph for the IoT system over the communication medium Firebase, which is summarized in the Figure 5.1.2.

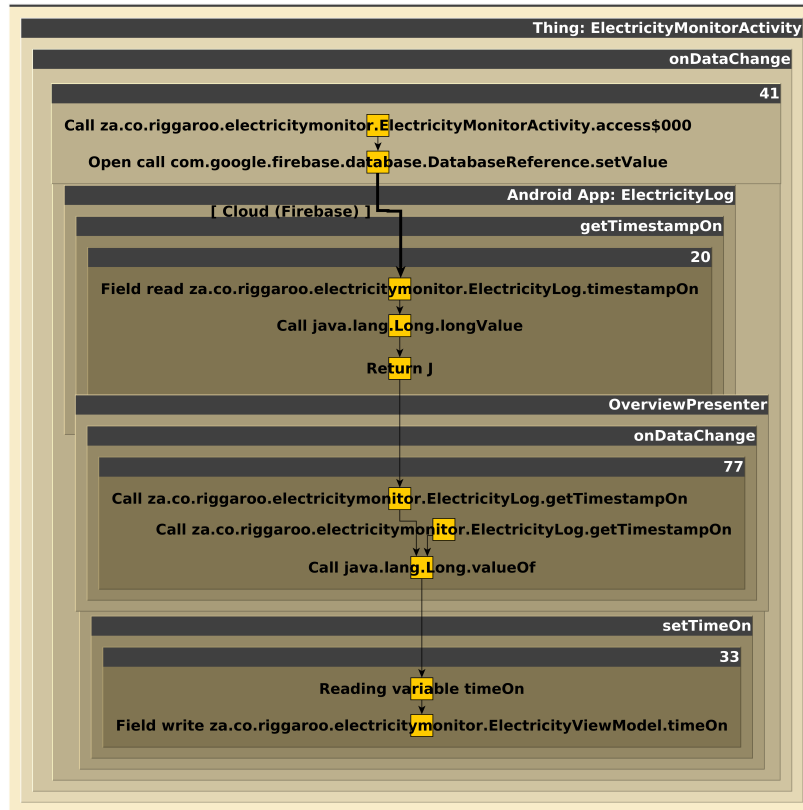


Figure 5.2: End-to-End taint propagation for cloud based communication channel (ElectricityMonitor)

## 5.2 Communication Channel: NFC

The third example we want to report is based on Near Field Communication (NFC) medium to interaction between the Things and the Android App. In particular the GitHub repository of the project is called *Color-Thing* [29] and it allows to change the color of the LED by an Android App. For this it sends the RGB Values to the Things using NFC.

What concerns the taint analysis of this project, we first of all specified sending and receiving method signatures. This will help in tagging the intermediate source and sink generated in the init phase of the analysis. In particular, in the Android application, the method *onColorSelected()* of the class [com.holgis.colorconnection.client.ControllerActivity] used in line 271 is marked as the ex-

ternal source. Whereas In the Thing, the Method *setPWM()* of [com.holgis.colorthings.PCA9685] class used at line 80 – 82 of ColorActivity is set as external sink. After the the init phase, a such information is updated in generated source-sink configuration file for the corresponding programs. Thus, for the Android Application the method *sendReliableMessage()* of [com.google.android.gms.nearby.connection.Connections] class is tagged as the intermediate source by the automated tagging process. Similarly for the Thing, the method *onMessageReceived()* of [com.google.android.gms.nearby.connection.Connections.MessageListener] class is tagged as the intermediate sink by the devised automated mechanism. These programs and tagged source-sink configuration files are used to perform the analysis in the report phase, where it generates the tainted path for the individual programs. Later this tainted paths are combined to generate the complete flow as depicted in Figure 5.3

### 5.3 Communication Channel: Bluetooth

Another wireless communication medium wildly used within IoT field is Bluetooth. For this purpose we analyzed the project: "Bluetooth Low-Energy (BLE) fun - Android (Things)" [30]. A such IoT system allows to use Bluetooth Low Energy to communicate between an Android Things board and an Android device. And in particular it sends a counter from the thing to the Android App using bluetooth.

The analysis of the project, first generates the source-sink configuration file from the init phase of the Julia checker. Here the *getInt()* of [android.content.SharedPreferences] class used at line 18 of AwesomenessCounter in the things program has been set as external source. Whereas the method *setText()* of the [android.widget.Button] class used at line 39 of InteractActivity class in the app program has been set as external sink. What concerns intermediate source and sinks, they have been tagged with the help of the bluetooth communication API. In particular, for the Things app the *sendResponse()* method of [android.bluetooth.BluetoothGattServer] class has been marked as intermediate sink. And the instance



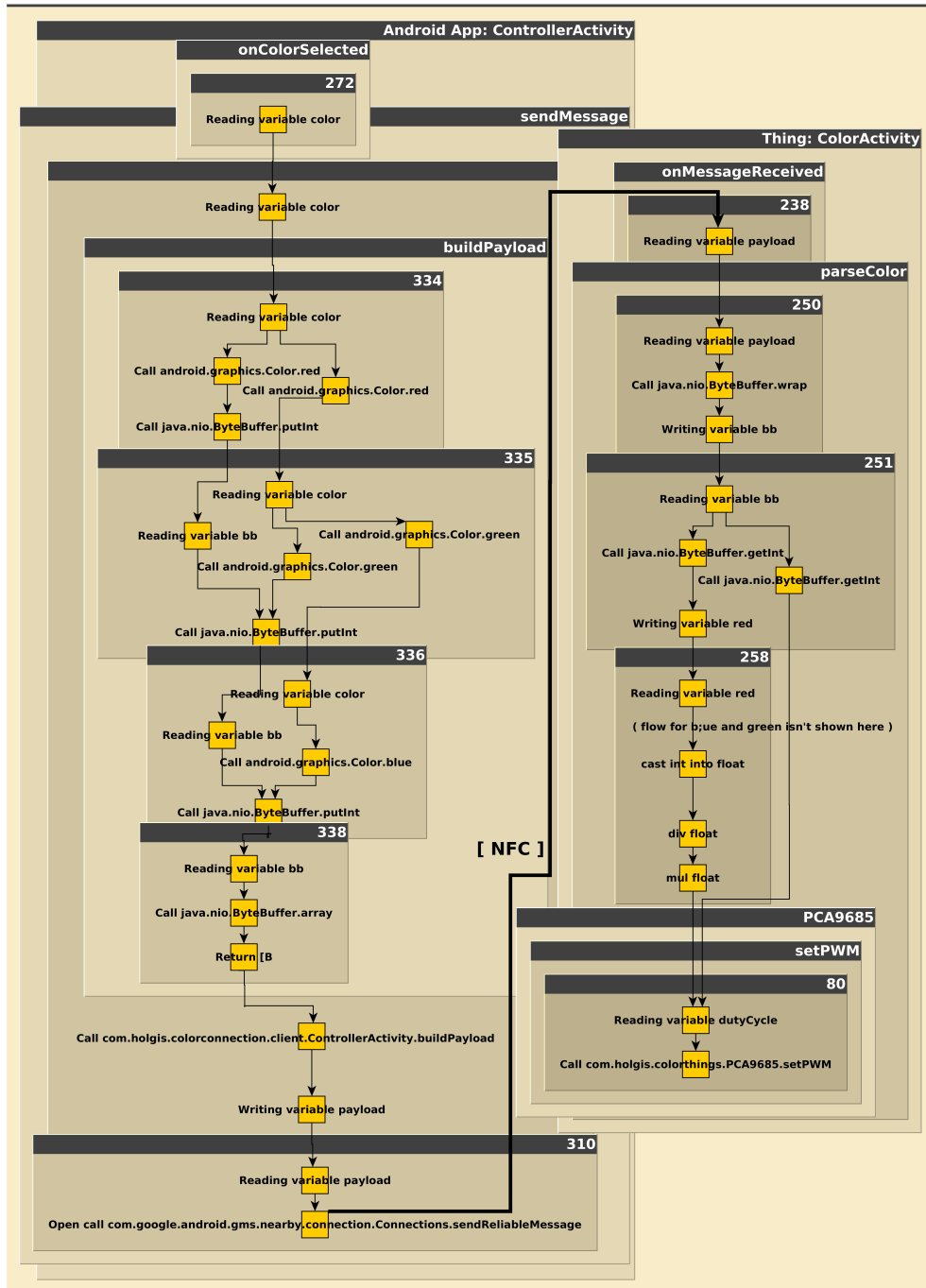


Figure 5.3: End-to-End taint propagation for NFC based communication channel

of *android.bluetooth.BluetoothGattCharacteristic* used in *onCharacteristicRead()* method of *GattClient* class has been tagged as an intermediate source for the Android Application. This tagging process has been automated in the devised approach. Afterwards the tagged source-sink and the programs are analyzed within the report phase, where it generates the tainted paths. Finally, the results of the analysis are depicted on Figure 5.3. In particular, it shows that our devised mechanism tracked the path from the external source to the external sink via various program points over the bluetooth communication medium.

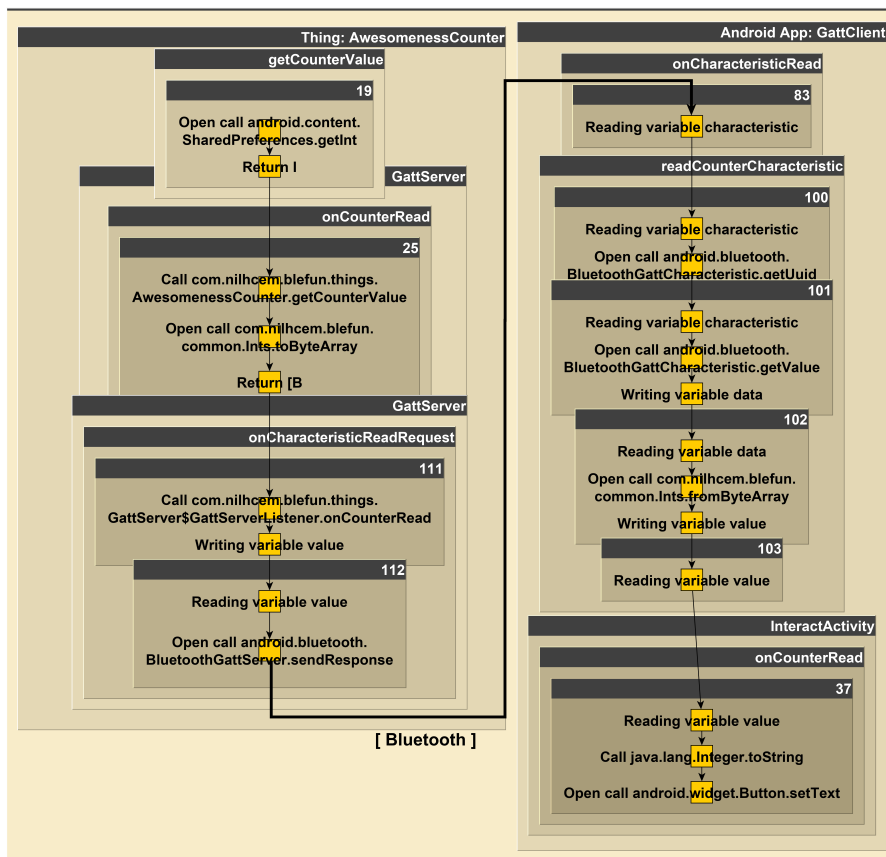


Figure 5.4: End-to-End taint propagation for Bluetooth based communication channel

## 5.4 Robocar

Finally, we want to report an example of a safe IoT system which does not comprise any cross-interface taint propagation. To this regard we analyzed Robocar GitHub repository [5]. In particular it allows to control, over the internet or manually with a regular joystick, the robotic car using the android thing.

In order to analyze the system, the program of the robotic car and the android app are made available for the init phase of Julia checker. It allowed to produce the initial source-sink configuration file. In which then, the *handleSlideButtonEvent()* method of *GameControllerActivity* class has been tagged as the external source for the Thing. The *changeSpeed()* method of the *Localhost-Driver* class has been defined as the external sink. Further, the automated process tags for the android app *setSpeed()* method of the *RobocarClient* service as the intermediate sink and *Robocar-Response()* method of the *Main Activity* class as the intermediate source. Afterwards, robotic car program and android application as well as tagged source-sink file have been analyzed in report phase. This reported in the robotic car the tainted received data which can reach the intended sink i.e. *changeSpeed()* method and can cause serious damage as depicted in Figure . But what concerns the android application, it does not include any tainted path for the given source-sink configuration. In particular, by analyzing the code it appears that the app only sends the directional command, not the speed of the robotic car. Instead the speed is determined locally at the Thing's side. Thus the analysis of the Robocar IoT system did not report any taint propagation from the perspective of cross-interface.

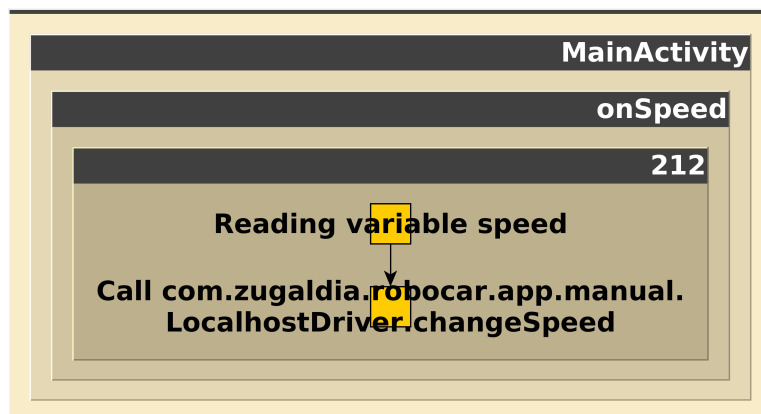


Figure 5.5: A safe IoT System Using cloud based communication channel

# Chapter 6

## Conclusions

OWASP IoT Top 10 [17] reports, the cross-interface propagation of tainted data is the most critical point of an interactive IoT system. With this work we aimed to devise an automated approach which will be capable of detecting cross-interface taint data propagation by integrating analyses of different interactive IoT programs in a sort of black-box framework. To this regard we first of all defined a formal model for detecting cross-program taint propagation based on graph representation. We considered both: the control flow graph of each program of particular IoT system and the communication channels they use to exchange data. Then we constructed a graph  $G^*$  which represents the iteration of each single component and data propagation of the overall IoT ecosystem. We further implemented this model by utilizing the existing taint analysis mechanism of Julia in order to develop a new cross-interface taint checker. In particular it works in two phase. In the first phase it generates a configuration file which represents a possible set of source and sinks for the given program. These sources and sinks are then tagged by an automated process with the help of communication channel's API. In the second phase we analyze the programs for identified (*tagged*) sources and sinks to discover the tainted paths in the program. Afterwards, the obtained result is parsed to determine the connectivity in the tainted paths among the interactive programs to construct the tainted graph for the IoT system. The overall analysis is carried out statically, so with no need to explicitly execute analyzed IoT system. Aside from, reducing the time

of a manual interconnection of isolated analysis, provides a technique to prevent security vulnerabilities like, SQLi, XSS, CSRF, by testing the system before its actual release.

In the next sections I would like to highlight the results we got by analyzing a set of real world IoT systems with our devised mechanism. Further discuss about possible future developments and integration of the approach. Finally conclude with mentioning the analysis, development, work in team and results interpretation, learned skills during this period of work.

## 6.1 Experimental work

Once we have implemented our mechanism of detecting tainted data propagation with use of JuliaSoft, we went further and tried to apply it to some of the IoT systems available on GitHub. In the Chapter 5, we showed in details that our automated approach is capable of detecting security vulnerabilities of multiple related programs having some common communication channel. In particular, it successfully pointed out end-to-end tainted data propagation of IoT systems based on android things, (*e.g Doorbell, Color-Thing, BLE fun*). For each of the tested projects we also reported the graph representation which indicates discovered tainted path(s). We grouped the projects to be tested on different communication channels and showed that un-sanitized data can be actually propagated independently of the communication medium. However in case the overall system does not comprises any tainted path(s), our mechanism correctly notifies about safety of the IoT ecosystem. In the following, we list the results obtained for each possible communication medium analyzed with this work:

- *Doorbell*: gives a possibility to a tainted data be propagated from the android Thing(*IoT back-end*) across cloud based communication medium(*Firebase*) towards the Android application. Our automated mechanism was able to detect tainted path through interfaces of the system components. Detailed information about tainted path has been reported on 5.1.1;

- *Color-Thing*: comprises possible tainted data propagation from the Android Application across NFC communication channel to the IoT back-end, Raspberry Pi component (*LED system*). Again, by setting the correspondent source and sink program points, our technique was able to discover the tainted path, which we have also reported at 5.2;
- *(BLE) fun*: analysis have reported the possible tainted data propagation from IoT Thing across Bluetooth communication channel to Android application. By analyzing each single program and Bluetooth API with JuliaSoft and afterwards combining results, we were able to track the tainted path from source to sink point which we have also reported at 5.3;
- *Robocar*: in this case, our mechanism identified only a possible taint data propagation within IoT back-end component, since then a such data are not sent to the front-end Android App. Thus the overall system has been considered as a safe;

## 6.2 Future work

Security within IoT field still leaves opened a huge number of vulnerabilities. As discussed before, most of them are due to reduced computational capabilities of Thing, lack of common standards, need to satisfy time to market factor by a manufacturer of IoT component, heterogeneity of different devices, etc. With this work, we aimed to devise a tool which will help to produce a safe software for an IoT ecosystem. It's necessary to mention that this is the first step towards modern and secure Internet of Things. With a future work we want to extend our automated static checker to accept further inspections of an IoT ecosystem software. To this regard we are now studying the following two cases:

### 6.2.1 Cross-program argument type control

An IoT ecosystem, as we showed above, consists of multiple components which run it's own software. These programs are mostly

independent, they have different execution environment, they run on different architectures, and are written with different programming languages. For instance, to get more control of the heap or to have further management of hardware of the Thing rather than to reduce execution latency, some low-level(*C/C++*) programming languages are used to develop software for the IoT back-end. Instead, what concerns front-end user devices, high-level(*Java*) programming language is preferred. Although different nature of components they use some communication media to exchange information in order to construct a final result. In general to share data and even execute methods/functions between *C/C++* and *Java* programs, the Java Native Interface(JNI) [8] is used. In particular it allows to share the execution environment of Java program, such that it's possible to access Java members from *C/C++*, it also allows to provide an implementation of Java method on *C/C++* side. Since neither *C/C++* compiler nor Java offers a static check of passed argument types across programs, some issue might arise during run-time of an IoT system. For instance, passing the double value from the java program and receiving a correspondent, wrong integer value is a feasible operation of JNI. For this purpose, we are now working on an automated tool which will be capable to discover this kind of incompatibilities during the static analysis of a such IoT system. In particular, our goal is to integrate the formal model by introducing the concept of a call graph [42] and again by considering also the JNI calls, define a unique graph representation of the overall system in terms of called method/functions. Then check whether there are some mismatch occurrences like:

$$\begin{aligned} &double \rightarrow int \\ &boolean \rightarrow double \\ &float \rightarrow short \end{aligned}$$

And in case arise an error message to avoid a future run time value misleading. Again, this argument type mismatches are not statically controlled and especially when programs are developed by several teams, it's difficult to discover the wrong value which propagates across the system causing wrong results.



### 6.2.2 Cross-program constant value propagation

Similarly to the previous case, using JNI as a communication channel among Java and C/C++ programs, it allows to access the java's global members from C/C++ code. To this regard JNI offers a set of methods which allows to retrieve an identifier of the member and then get access to it's value. The issue, we have faced out consists on accessing the constant java members from the C/C++ function which afterwards could be successfully modified. For instance the possible scenario is following one: in the java class Test there is a *const* member *pi* to which has been assigned a value 3.141592, then a such member is accessed through JNI and stored in some C++ variable of a class Compute. There is no controls whether an accessed variable is declared as constant or not, so it allows to update the value and "bypass" the global constant property.

For this purpose we want to enhance our previous approach to integrate also constant value propagation analysis. In particular, track each constant member on a Control Flow Graph representing the overall system and discover whether the constant property is preserved across inter-program calls and if is not, arise an error message.

## 6.3 Learned skills

With this work, we devised an automated tool to allow statically discover a set of vulnerabilities of an IoT ecosystem. Our work doesn't stop here, we aimed also to enhance tainted analysis with other two cases explained above. It's necessary to mention that behind all that work, analysis and implementation, as well as testing process, I improved my skills in static analysis techniques, understood better how the theory I've seen during the lectures (*e.g call graph construction, backward and forward analysis, communication protocols*) can be applied in a real world scenario in order to get a sound results. This work gave me opportunity to get aware how actually insecure is the IoT field and that it still requires a huge amount of work to be done until we get a first examples of safe Internet of Things.

## Online references

- [1] Report from security firm Symantec,  
[www.techrepublic.com](http://www.techrepublic.com)
- [2] The biggest IoT security failures of 2018,  
[www.techrepublic.com](http://www.techrepublic.com)
- [3] Perl 5 documentation on tainted checking ,  
[www.perldoc.perl.org](http://www.perldoc.perl.org)
- [4] Locking Ruby in the Safe ,  
[www.phrogz.net](http://www.phrogz.net)
- [5] Antonio Zugaldia. 2017. Android Robocar  
[www.github.com/zugaldia](http://www.github.com/zugaldia)
- [6] SQL injection vulnerability,  
[www.owasp.org](http://www.owasp.org)
- [7] XSS vulnerability,  
[www.owasp.org](http://www.owasp.org)
- [8] Java Native Interface Specification Contents,  
[www.docs.oracle.com](http://www.docs.oracle.com)
- [9] Material of Reaching definition technique  
<http://www.dsi.unive.it>
- [10] Material of Live variable technique  
<http://www.dsi.unive.it>
- [11] Material of Very Busy Expressions definition technique  
<http://www.dsi.unive.it>

- [12] The biggest IoT security failures of 2018,  
[www.embedded-computing.com](http://www.embedded-computing.com)
- [13] The Hunt for IoT: Multi-Purpose Attack Thingbots Threaten Internet Stability and Human Life,  
[www.f5.com](http://www.f5.com)
- [14] Shock at the wheel: your Jeep can be hacked while driving down the road,  
[www.kaspersky.com](http://www.kaspersky.com)
- [15] The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet,  
[www.csoonline.com](http://www.csoonline.com)
- [16] Tainted flow analysis,  
[www.laure.gonnord.org](http://www.laure.gonnord.org)
- [17] OWASP Internet of Things Project,  
[www.owasp.org](http://www.owasp.org)
- [18] Juliasoft code analysis,  
[juliasoft.com](http://juliasoft.com)
- [19] IoT analytics, The Top 10 IoT Segments in 2018 ? based on 1,600 real IoT projects,  
[iot-analytics.com](http://iot-analytics.com)
- [20] New trends in the world of IoT threats,  
[securelist.com](http://securelist.com)
- [21] Trend Micro Research Finds Major Lack of IoT Security Awareness,  
[www.smart-industry.net](http://www.smart-industry.net)
- [22] Welcome to Apache HBase [www.hbase.apache.org](http://www.hbase.apache.org)
- [23] Random Forests Leo Breiman and Adele Cutler,  
[www.stat.berkeley.edu](http://www.stat.berkeley.edu)

- [24] Amit Kr Mandal. 2019. Plant Monitoring System, Servlet.  
[jgithub.com/amitmandalnitdgp](https://github.com/amitmandalnitdgp).
- [25] Plant Monitoring System - IoT Back- end.  
[jgithub.com/amitmandalnitdgp](https://github.com/amitmandalnitdgp).
- [26] Android App for Plant Monitoring System.  
[jgithub.com/amitmandalnitdgp](https://github.com/amitmandalnitdgp).
- [27] Dave Smith. 2018. doorbell  
[jgithub.com/androidthings](https://github.com/androidthings).
- [28] Android things electricity monitor  
[jgithub.com/riggaroo](https://github.com/riggaroo).
- [29] Holger. 2016. Color-Things.  
[jgithub.com/holgi-s](https://github.com/holgi-s).
- [30] Gautier MECHLING. 2018. Bluetooth Low-Energy (BLE) fun  
Android (Things)  
[jgithub.com/Nilhcem](https://github.com/Nilhcem).

## Scientific literature

- [1] FRUSTACI, Mario, et al. Evaluating critical security issues of the IoT world: present and future challenges. *IEEE Internet of Things Journal*, 2017, 5.4: 2483-2495
  
- [2] GE, Mengmeng. A framework for automating security analysis of the internet of things. *Journal of Network and Computer Applications*, 2017, 83: 12-27.
  
- [3] ASPLUND, Mikael; NADJM-TEHRANI, Simin. Attitudes and perceptions of IoT security in critical societal services. *IEEE Access*, 2016, 4: 2130-2138.
  
- [4] DAS, Ashok Kumar; ZEADALLY, Sherali; HE, Debiao. Taxonomy and analysis of security protocols for Internet of Things. *Future Generation Computer Systems*, 2018, 89: 110-125.
  
- [5] TIP, Frank; PALSBERG, Jens. Scalable propagation-based call graph construction algorithms. *ACM*, 2000.
  
- [6] TWENEBOAH-KODUAH, Samuel; SKOUBY, Knud Erik; TADAYONI, Reza. Cyber security threats to IoT applications and service domains. *Wireless Personal Communications*, 2017, 95.1: 169-185.

- [7] MAVROPOULOS, Orestis. Apparatus: A framework for security analysis in internet of things systems. *Ad Hoc Networks*, 2018.
- [8] KHATTAK, Hasan Ali. Perception layer security in Internet of Things. *Future Generation Computer Systems*, 2019, 100: 144-164.
- [9] HASAN, Mahmudul. Attack and Anomaly Detection in IoT Sensors in IoT Sites Using Machine Learning Approaches. *Internet of Things*, 2019, 100059.
- [10] LOHACHAB, Ankur; KARAMBIR, Bidhan. Critical Analysis of DDoS - An Emerging Security Threat over IoT Networks. *Journal of Communications and Information Networks*, 2018, 3.3: 57-78.
- [11] MILOSLAVSKAYA, Natalia; TOLSTOY, Alexander. Internet of Things: information security challenges and solutions. *Cluster Computing*, 2019, 1-17.
- [12] REPS, Thomas; SAGIV, Mooly; HORWITZ, Susan. Interprocedural dataflow analysis via graph reachability. *Datalogisk Institut, Kobenhavns Universitet*, 1994.
- [13] ABDUL-GHANI, Hezam Akram; KONSTANTAS, Dimitri; MAHYOUB, Mohammed. A comprehensive IoT attacks survey based on a building-blocked reference model. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 2018, 9.3.
- [14] GAMUNDANI, Attlee M.; PHILLIPS, Amelia; MUYINGI, Hippolyte N. An Overview of Potential Authentication Threats and Attacks on Internet of Things (IoT): A Focus on Smart Home

- Applications. *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCoM) and IEEE Smart Data (SmartData)*, IEEE, 2018. p. 50-57.
- [15] LING, Zhen. An Overview of Potential Authentication An end-to-end view of iot security and privacy. *GLOBECOM 2017-2017 IEEE Global Communications Conference.*, IEEE, 2017. p. 1-7.
- [16] HAO, Peng; WANG, Xianbin; SHEN, Weiming. A Collaborative PHY-Aided Technique for End-to-End IoT Device Authentication, *IEEE Access*, 2018, 6: 42279-42293.
- [17] SHAH, Trusit; VENKATESAN, S. Authentication of IoT Device and IoT Server Using Secure Vaults. *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, IEEE, 2018. p. 819-824.
- [18] KIM, SuHyun; LEE, ImYeong. IoT device security based on proxy re-encryption. *Journal of Ambient Intelligence and Humanized Computing*, 2018, 9.4: 1267-1273.
- [19] CHALLA, Sravani. Secure signature-based authenticated key establishment scheme for future IoT applications. *IEEE Access*, 2017, 5: 3028-3043.
- [20] PORAMBAGE, Pawani. Two-phase authentication protocol for wireless sensor networks in distributed IoT applications. *2014 IEEE Wireless Communications and Networking Conference (WCNC)* IEEE, 2014. p. 2728-2733.

- [21] GIULIANO, Romeo. Security access protocols in IoT capillary networks. *IEEE Internet of Things Journal* 2016, 4.3: 645-657.
- [22] JOSHY, Annies; JALAJA, M. J. Design and implementation of an IoT based secure biometric authentication system. *2017 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES) IEEE*, 2017. p. 1-13.
- [23] SUJATHA, S. Mary; DEVI, Y. Usha. Design and implementation of IoT testbed with three factor authentication. *International Conference on Communication and Electronics Systems (ICCES) IEEE*, 2016. p. 1-5.
- [24] KINIKAR, Swati; TERDAL Sujatha. Implementation of open authentication protocol for IoT based application. *International Conference on Inventive Computation Technologies (ICICT) IEEE*, 2016. p. 1-4.
- [25] SHIN, Daemin. Secure and efficient protocol for route optimization in PMIPv6-based smart home IoT networks. *IEEE Access*, 2017, 5: 11100-11111
- [26] FARRIS, Ivan. TERDAL Sujatha. A survey on emerging SDN and NFV security mechanisms for IoT systems. *IEEE Communications Surveys & Tutorials IEEE*, 2018, 21.1: 812-837
- [27] SAHAY, Rashmi. Efficient Framework for Detection of Version Number Attack in Internet of Things. *International Conference on Intelligent Systems Design and Applications Springer, Cham*, 2018. p. 480-492



- [28] HOU, Jianwei; QU, Leilei; SHI, Wenchang. A survey on internet of things security from data perspectives. *Computer networks*, 2019, 148: 295-306
- [29] NGUYEN, Huy-Trung; NGO, Quoc-Dung; LE, Van-Hoang. IoT Botnet Detection Approach Based on PSI graph and DGCNN classifier. *International Conference on Information Communication and Signal Processing (ICICSP)*, IEEE, 2018. p. 118-122.
- [30] PROKOFIEV, Anton O.; SMIRNOVA, Yulia S.; SUROV, Vasiliy A. A method to detect Internet of Things botnets. *Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, IEEE, 2018. p. 105-108.
- [31] CHOI, Hyunsang. Botnet detection by monitoring group activities in DNS traffic. *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, IEEE, 2007. p. 715-720.
- [32] TIEN, Chin-Wei. UFO-Hidden Backdoor Discovery and Security Verification in IoT Device Firmware. *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2018. p. 18-23.
- [33] COUSOT, Patrick; COUSOT, Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, 1977. p. 238-252.
- [34] Cliff Click and Keith D Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17, 2 (1995), 181–196.

- [35] PIOLI, Anthony; HIND, Michael. Combining interprocedural pointer analysis and conditional constant propagation. *IBM Thomas J. Watson Research Division*, 1999.
- [36] HUUCK, Ralf. The internet of threats and static program analysis defense. *EmbeddedWorld 2015: Exhibition & Conferences*, 2015. p. 493.
- [37] Z Berkay Celik, Earlene Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2018. *Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities*, arXiv preprint arXiv:1809.06962 (2018).
- [38] CLAUSE, James; LI, Wanchun; ORSO, Alessandro. Dytan: a generic dynamic taint analysis framework. *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM, 2007. p. 196-206.
- [39] CAO, Kai. PHP vulnerability detection based on taint analysis. *International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, IEEE, 2017. p. 436-439.
- [40] ZHANG, Ruoyu; HUANG, Shan; QI, Zhengwei. Efficient taint analysis with taint behavior summary. *Third International Conference on Communications and Mobile Computing.*, IEEE, 2011. p. 11-14.
- [41] FERRARA, Pietro; SPOTO, Fausto. Static Analysis for GDPR Compliance. *ITASEC.*, 2018.

- [42] RYDER, Barbara G. Constructing the call graph of a program.  
*IEEE Transactions on Software Engineering*, 1979, 3: 216-226.