Cross-Program Taint Analysis for IoT Systems

Amit Mandal
SRM University, AP - Amaravati, India

Pietro Ferrara
Universitá Ca' Foscari di Venezia

Yuliy Khlyebnikov
Universitá Ca' Foscari di Venezia

Agostino Cortesi
Universitá Ca' Foscari di Venezia

Fausto Spoto
Universitá di Verona

ABSTRACT

Cross-program propagation of tainted data (such as sensitive information or user input) in an interactive IoT system is listed among the OWASP IoT top 10 most critical security risks. When programs run on distinct devices, as it occurs in IoT systems, they communicate through different channels in order to implement some functionality. Hence, in order to prove the overall system secure, an analysis must consider how these components interact. Standard taint analyses detect if a value coming from a source (such as methods that retrieve user input or sensitive data) flows into a sink (typically, methods that execute SQL queries or send data into the Internet), unsanitized (that is, not properly escaped). This work devises a cross-program taint analysis that leverages an existing inter-program taint analysis to detect security vulnerabilities in multiple communicating programs. The proposed framework has been implemented above the inter-program taint analysis of the Julia static analyzer. Preliminary experimental results on multi-program IoT systems, publicly available in GitHub, show that the technique is effective and detects inter-program flows of tainted data that could not be discovered by analyzing each program in isolation.

1 INTRODUCTION

The Internet of Things (IoT) paradigm extends computing and network capabilities to various types of devices by enabling seamless device-to-device interactions. IoT is a key enabling technology for the next Industry 4.0 revolution. The latest Gartner estimations predict that 25 billion IoT devices will be interconnected by 2021. This trend accelerates the implementation of Industry 4.0 by leveraging a sophisticated mechanism to gather and exploit system data [27]. In general, a typical IoT system consists of many devices (*things*) that rely on some form of network communication to an enterprise back-end server. Figure 1 shows the IoT architecture proposed by the Eclipse working group [14], whose three main components are devices (things), gateways, and cloud platforms. These systems are pervasive; each component runs its own software and communicates with the others over some channels; it can collect data from the external environment (through sensors or cameras) and take action over it (through actuators). This rises potential privacy and security issues. Namely, it is essential to protect data flowing through IoT devices along its life cycle and to avoid that an attacker injects arbitrary input that jeopardizes safety. The lack of tools and techniques for detecting security vulnerabilities and privacy leaks is actually a major issue for the expansion and pervasiveness of IoT systems. At the same time, the diversity of the devices in such systems continues to grow, ranging from simple sensors and actuators to complex hardware like in connected vehicles, smart TVs and cameras. This diversified adaption of IoT testifies the wide potential and genericity of this technology, but complicates any form of analysis.

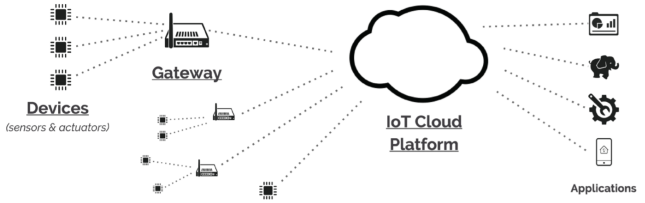

Figure 1: A standard IoT architecture.

Recent literature has largely studied the security of IoT authentication and authorization mechanisms as most of the devices have limited computing capabilities for complex authentication procedures [7, 30]. In order to deal with the flexibility of the communication network, IoT often relies on software defined networks (SDN). The OWASP IoT Top 10 2018 [44] highlights *insecure ecosystem interfaces* as a major security threat. Attacks to the open interface of the SDN and their mitigation have been widely discussed [17, 34, 47]. In addition, several protocols [25, 32, 49] have been devised to ensure security in unidirectional and bidirectional communications among devices. However, since IoT comprises multiple heterogeneous components, a holistic framework [22, 39] to assess security is desirable, but a single solution, for all security issues, is prohibitive in practice. Thus, insecure web, back-end API, cloud storage and mobile interfaces in the IoT ecosystem can still play a crucial role in compromising the security of the devices and related components.

Static program analysis [11, 12] has been already widely applied to identify security vulnerabilities and privacy leaks of software [51, 53], typically on web applications. However, such techniques are focused on single software components and assume, conservatively, that data coming from the external environment is *dangerous* [18]. Hence, they are too conservative for the analysis of the software components of an IoT system, that interact in a well-orchestrated way. In addition, security and integrity of these systems is challenged in multiple stages, as data flows through many devices, networks, and administrative boundaries. In contrast to run-time analysis, such as testing, that covers only a (small) portion of the security and privacy issues in IoT [33] and can explore only some execution paths, for very specific inputs and run-time contexts, static program analysis can achieve full code coverage, as it analyzes the program by abstracting its behavior. Hence, it does

not need specific run-time values to explore different execution paths. In this context, several static analyses have been introduced to identify security and privacy issues of IoT platforms [6, 31]. Some studies have focused on data leakages and security vulnerabilities in specific IoT environments, such as car infotainment systems [38, 45]. However, these approaches focus on specific vulnerabilities and contexts. As far as we know, taint analysis has not been generalized, yet, to the detection of security vulnerabilities and privacy leaks in IoT systems.

1.1 Contribution

This paper extends the static, single-program taint analysis of Julia [51, 52] to devise a cross-program taint analysis that detects security vulnerabilities and privacy leaks in multiple programs that share data through some communication channels. Taint analysis detects if a value coming from a *source* flows into a *sink*, unsanitized. Starting from this intra-program definition, this work formalizes a model for cross-program taint propagation.

Namely, intra-program taint analysis receives as input the set of sources and sinks. Therefore, our technique generates these sets for all the programs of the IoT system under analysis. In particular, it distinguishes between external and communication channel sources and sinks. The former are standard components (that might for instance retrieve user input or send a command to a hardware device); the latter involve the interfaces of communication channels to receive (source) and send (sink) data through them. Our technique then runs the intra-program taint analysis on all the programs, obtaining data flow graphs that represent data flows from a source to a sink. These are connected by adding the edges between programs that communicate through each given communication channel. The resulting inter-program flow graph is then projected on the paths that connect external sources and external sinks.

We applied the implementation of our technique to five IoT systems publicly available on GitHub, as well as to an illustrative example devised in order to test our technique. In all cases but one, our technique reconstructed the data flow connecting the external source to the external sink. In one case, there was no explicit data flow, thus the taint analysis engine did not produce any flow of data from the external source to the communication channel. This paper will discuss all these cases.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides and discusses an illustrative example that demonstrates cross-program taint propagation across the interfaces of different interactive IoT programs running on different devices. Section 4 formalizes our technique. Section 5 provides details about the implementation and discusses the results obtained on five IoT applications. Section 6 concludes.

2 RELATED WORK

IoT relies on the most sophisticated cloud technologies, and therefore inherits cloud security concerns. To make things even worse, IoT poses new security challenges, since it allows cloud interactions between remote users (potentially attackers) of the IoT system and physical devices. Several different approaches have been recently developed to address these issues. Frustaci *et al.* [21] and Neshenko *et al.* [41] provide a taxonomic analysis of the IoT security panorama that considers three important aspects of the IoT system model: perception, transportation, and application. Similarly, Das *et al.* [13] perform a comparative study of different security IoT protocols. Whereas, Ge *at al.* [22] and Mavropoulos *et al.* [39] focus on generic security frameworks for IoT to analyze IoT security strategies through some well-defined security metrics. Again, Hao *et al.* [28], Shah *et al.* [48] and Challa *et al.* [7] provide cryptography-based authentication mechanisms to mitigate various computation-based impersonation attacks. Beside authentication, secure networks are also essential to protect IoT systems from cyber-attacks. In this regard, Zaidan *et al.* [55] highlight the challenges of existing communication components for IoT-based smart homes. Instead, Sahay *et al.* [16] devise a framework called CyberShip-IoT, that mitigates network traffic attacks by leveraging the Software-Defined Networking (SDN) paradigm. Finally, Farris *et al.* [17] analyze security features of Network Functions Virtualization (NFV) and SDN from an IoT security perspective. The majority of these approaches enhances security, but incurs in a substantial implementation complexity and imposes significant computational overhead to power-constrained IoT devices.

The integrity and confidentiality of data transmitted in an IoT environment is critical [32]. However, conventional cipher algorithms do not match memory and computational limits of IoT devices. For this reason, Kim *et al.* [32] devise a mechanism that implements proxy re-encryption for secure and efficient data transmission. Sahay *et al.* [47] propose a mechanism to detect the malicious nodes responsible for version number attacks. Whereas, Hou *et al.* [30] devise a three-dimensional approach for exploring IoT security by combining the concept of IoT architectures with data life cycles. Unlike many other security approaches, this work considers the data flow from IoT end-point devices through the cloud and vice versa. However, it focuses on secure usage of IoT data, but it does not track data taintedness.

In contrast to that, static program analysis [11] can be very useful to determine taintedness of data propagating across different systems and its effect on the IoT system at compile time. Among its various applications to security vulnerabilities, taint analysis has been widely adopted to detect issues like SQL injections and cross-site scripting [3, 8, 42, 51, 53]. Recently, the application of other static analysis techniques to IoT systems has been considered. Huuck [31] discusses the usage of static code analysis to detect some IoT security vulnerabilities. Xu *et al.* [54] devise a mechanism of firmware code analysis for the assembly code successfully discovering vulnerabilities in Siemens PAC4200 power meters. Nobakht *et al.* [43] develop a static analysis tool called PGFIT to identify overprivileged issues in apps for Google Fit. Alnaeli *et al.* [2] uses static analysis to detect usage of unsafe functions in popular IoT systems. Blanchard *et al.*[4] demonstrate the effectiveness of FRAMA-C for discovering vulnerabilities in the Contiki operating system widely used in IoT. Alasmary *et al.* [1] carry out a detailed graph-based comparative analysis of Android and IoT malwares. Kim *et al.* [32] adopt static analysis to detect unwanted system halts and suppress the fast propagation of the device-freezing status in the entire IoT system. Similarly, Celik *et al.* [6] identify security and privacy issues of five IoT platforms, and they apply existing static analyzers to detect them. These approaches point out that "a suite of analysis

tools and algorithms targeted at diverse IoT platforms is at this time largely absent". However, taint analysis should consider how multiple programs of an IoT system interact and communicate, in order to detect issues on the overall system, while this approach has been applied to programs *in isolation* up to now. Therefore, the current IoT security landscape demands a mechanism for analyzing the security vulnerabilities of the IoT system, that is aware of cross-program data propagation. In this regard, existing taint analysis techniques can be very useful, if extended in that direction. This is exactly the contribution of this paper.

3 ILLUSTRATIVE EXAMPLE

This section introduces an illustrative example used to explain how the different formal components work in practice on an IoT system. For this purpose, we developed a simplified plant monitoring system as *edge* software on a Raspberry Pi 3B+ with 64-bit, Quad-Core, Broadcom BCM2837B0 CPU running at 1.4GHz and 1GB of LPDDR2 SDRAM[1]. This Java program reads temperature, humidity and soil-moisture through sensors. Collected data is then sent to a remote Hadoop 3.2.0 database. A simple Java servlet reads it from the database and makes it available to an Android mobile application (available at [37]), developed for Android API 25. Figure 2 reports the overall architecture of the system.

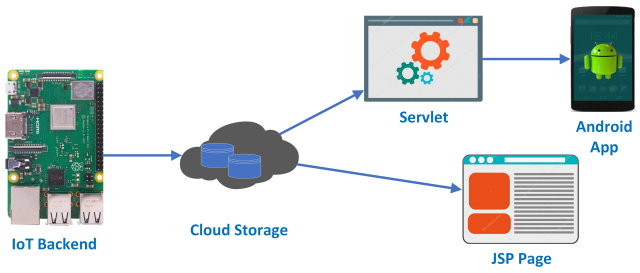

Figure 2: Example scenario.

3.1 Julia Taint Analysis

As a starting point, we analyzed our code with the Julia static analyzer [52], by applying its extension of taint analysis [5, 15, 51] to track sensitive data [19]. Julia's taint analysis has been applied to various properties, like SQL injection, cross-site scripting and leakage of sensitive data. The latter analysis (called GDPR checker) provides an exhaustive report tracking how sensitive data flows through the program up to a sink, and it allows the user to specify sources and sinks through an Excel spreadsheet built during the Init phase of the checker. This spreadsheet contains the potential program points that could retrieve sensitive data, as well as potential locations of leakage. The user can then tag these with the category of sensitive data they retrieve or with leakage points they disclose information to. The Report phase then applies the taint analysis engine with the specification provided through the annotated Excel file. It returns an exhaustive report with all possible data flow graphs representing potential leakages.

[1] Available at https://github.com/amitmandalnitdgp/IOT-EcoSyatem

3.2 Edge Software

Figure 3 reports a code snippet of the IoT edge software [37] for the plant monitoring system. Such software executed in the IoT backend (e.g., a Raspberry Pi device) of Figure 2. It reads sensor data at line 12 and performs some computation on it, to compute average humidity and temperature (lines 17–19). At the end, it stores the data in the database (line 27).

In the Excel spreadsheet for the edge software, we manually tag readline() as source and add() as sink. We then apply Julia's taint analysis using this specification. Its results show a possible leakage of data between the source and the sink, shown in Figure 4. This leakage is a flow from the sensor sensitive data to the database storage.

```
1  public class Server {
2    private static Socket socket1;
3    private static Socket socket2;
4    public static void main(String[] args) {
5      int rowNo = 1;
6      ...
7      while(true) {
8        // sensor 1
9        InputStream is1 = socket1.getInputStream();
10       InputStreamReader isr1 = new InputStreamReader(is1);
11       BufferedReader br1 = new BufferedReader(isr1);
12       String msg1 = br1.readLine();
13       String[] r1 = msg1.split("\t+");
14       // sensor 2: same code below as before for sensor 1, but flowing into r2
15       ...
16       // compute average of the two sensors
17       hum = (Float.parseFloat(r[0])+Float.parseFloat(r2[0]))/2;
18       tc = (Float.parseFloat(r1[1])+Float.parseFloat(r2[1]))/2;
19       tf = (Float.parseFloat(r1[2])+Float.parseFloat(r2[2]))/2;
20       // create an object with the measured data
21       Put p = new Put(Bytes.toBytes("row"+rowNo));
22       p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("humidity"),Bytes.toBytes(hum));
23       p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("tempc"),Bytes.toBytes(tc));
24       p.add(Bytes.toBytes("ambiance"),Bytes.toBytes("tempf"),Bytes.toBytes(tf));
25       p.add(Bytes.toBytes("soil"),Bytes.toBytes("moisture"),Bytes.toBytes(r1[3]));
26       // insert data into the database
27       table.put(p);
28       rowNo++;
29     }
30     ...
31   }
32 }
```

Figure 3: The edge software.

3.3 Java Servlet

The Java servlet retrieves data from the database (where it was stored by the edge software and hence to be considered as sensitive) and makes it available to other components (in particular, to the Android app). Figure 5 reports a code snippet of this servlet [36]. It receives as parameter the type of sensor (sensType) that the client wants to access (line 4) and queries the database (line 5). After the connections is established (lines 15–17), it retrieves data corresponding to the client request (line 18), computes the average of the stored data (lines 19–27) and returns the result (line 29). Then the servlet issues a response with such data (lines 6–9).

The Excel spreadsheet for this servlet contains getString() (reading data from the database) as source, and write(String) (writing data into the servlet response) as sink. Figure 6 reports the results of the taint analysis reporting a flow from source to sink.

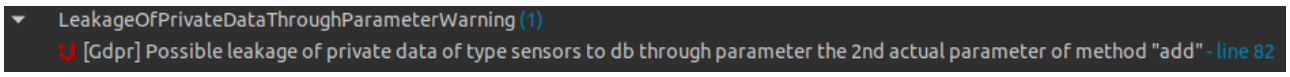

Figure 4: Results of taint analysis on the edge software.

```
1  @WebServlet("/HbaseConnection")
2  public class HbaseConnection extends HttpServlet {
3    protected void doPost(HttpServletRequest request, HttpServletResponse
         response){
4      String sensType = request.getParameter("sensType");
5      String result = queryHbase(sensType);
6      response.setStatus(HttpServletResponse.SC_OK);
7      OutputStreamWriter writer = new
           OutputStreamWriter(response.getOutputStream());
8      writer.write(result);
9      ...
10   }
11   public String queryHbase(String sensType) {
12     double val=0.0;
13     String result = "";
14     try {
15       Class.forName("org.apache.drill.jdbc.Driver");
16       Connection c = DriverManager.getConnection("jdbc:drill:zk=...");
17       Statement st = c.createStatement();
18       ResultSet rs1 = st.executeQuery("SELECT SensData." + sensType + " FROM
             hbase.SensData");
19       int count=0;
20       while(rs1.next()) {
21         float temp = Float.parseFloat(rs1.getString(1));
22         val = val+temp;
23         count++;
24       }
25       val = val/count*100;
26       val = Math.round(val);
27       val = val/100;
28     } catch (ClassNotFoundException | SQLException e) {...}
29     return ""+val;
30   }
31 }
```

Figure 5: The Java servlet.

3.4 Android Application

The Android application [35] consists of two different classes: BackgroundWorker is responsible for collecting data from the servlet, while MainActivity takes the values received by the BackgroundWorker and shows data to the user. The latter (Figure 7) identifies which type of sensor information the user wants to see (lines 10–17), then retrieves such data through the background worker (line 18) and shows it into a text view (line 19). The background worker (Figure 8) connects to the Java servlet of Section 3.3 (lines 8–9), queries it for the type of sensor data passed by the main activity (lines 11–13), reads the data (lines 15–20) and returns it (line 22).

In the Excel spreadsheet, we specified readLine() (line 17 of Figure 8) as source and setText(String) (line 19 of Figure 7) as sink. Figure 9 reports the results of taint analysis, showing a potential flow from source to sink.

3.5 Discussion

The taint analysis is able to detect the path between the specified sources and sinks, for individual components of the system, *when considered in isolation*. If we take a closer look at these results, we notice that tainted data propagated in the following way: sensor → database → internet → app. Each software component builds one of these arrows: the edge software is responsible for sensor → database; the servlet for sensor → internet; and the mobile application for internet → app. However, the analysis of the three components in isolation requires some manual integration of the results, which is impractical for real world applications whose components might be many and large and include many sources, sinks and communication channels. Instead, the tainted paths of a single component may extend into other components and this extension must be automated. This is our contribution, presented in the next sections.

4 CROSS-PROGRAM TAINT ANALYSIS

This section formalizes our technique that extends existing static taint analysis to multi-program systems.

4.1 Single Component Analysis

We start by formalizing the taint analysis on a single component. We rely on the inter-procedural data flow approach introduced by [46] as basis for our formalization.

A program p is represented as a 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, E_p is the set of directed edges between statements in the program, *i.e*, its control flow, S_p is the set of entry points of different procedures/functions, and Ex_p is the set of exit points. The statements of G_p can be classified into call nodes (Call_p) (representing the statements where calls to other functions occur) and return nodes (Ret_p) (representing the statements where control returns to the caller). Thus $\text{Call}_p \cup \text{Ret}_p \subset N_p$.

The set of edges E_p can be defined as:

$$E_p = (S_p \times N_p) \cup (N_p^i \times N_p^j) \cup (N_p \times \text{Call}_p) \cup (\text{Call}_p \times S_p) \cup (Ex_p \times \text{Ret}_p) \cup (N_p \times Ex_p)$$

where:

- $(S_p \times N_p)$ connects a start node to a statement within the same program,
- $(N_p^i \times N_p^j)$ is an edge between the i^{th} and j^{th} statement of the program,
- $(N_p \times \text{Call}_p)$ is an edge from a statement to a call site node within the program,
- $(\text{Call}_p \times S_p)$ is an edge between a call-site node to a start node of another procedure,
- $(Ex_p \times \text{Ret}_p)$ is an edge between an exit node to a return-site node of another procedure, and
- $(N_p \times Ex_p)$ is an edge between a statement to the exit node within the program.

4.2 Communication Channels

Different programs communicate and exchange data through some communication channel (*e.g.*, a database or a Bluetooth connection). A communication channel $C = \{medium, f^s, f^r\}$ consists of the type of its communication medium and of functions to send ($f^s \in I_S$) and receive ($f^r \in I_R$) data. Namely, $I_S = \{f_1^s, f_2^s, \ldots, f_h^s\}$ and $I_R = \{f_1^r, f_2^r, \ldots, f_k^r\}$.

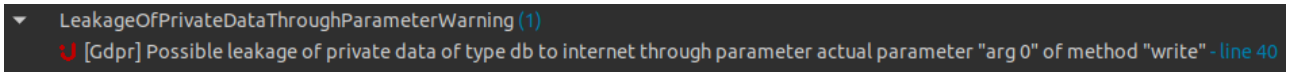

Figure 6: Results of taint analysis on the Java servlet.

```
public class MainActivity extends AppCompatActivity {
  TextView textView;
  RadioGroup radioGroup;
  RadioButton radioButton;
  public void onClick(View view) {
    String sensor="";
    String type = "mul";
    BackgroundWorker backgroundWorker = new BackgroundWorker(this);
    String result = null;
    int sensType = radioGroup.getCheckedRadioButtonId();
    if(sensType!=-1) {
      radioButton = (RadioButton) findViewById(sensType);
      String str = (String) radioButton.getText();
      if(str.equals("Soil Moisture")){sensor = "soil.moisture";}
      else if(str.equals("Temerature in C")){sensor = "ambiance.tempc";}
      else if(str.equals("Temerature in F")){sensor = "ambiance.tempf";}
      else if(str.equals("Humidity")){sensor = "ambiance.humidity";}
      ... // Wait and retrieve the data from BackgroundWorker
      textView.setText("Average "+str+": "+result);
    }
  }
}
```

Figure 7: The MainActivity **of the Android application.**

```
public class BackgroundWorker extends AsyncTask<String, Void, String> {
  protected String doInBackground(String... params) {
  String type = params[0];
  String temp = params[1];
  String servletURL = "...";
  if(type.equals("mul")) {
    String result, line;
    URL url = new URL(servletURL);
    HttpURLConnection httpURLConnection = url.openConnection();
    ...
    BufferedWriter bufferedWriter = new BufferedWriter(new
          OutputStreamWriter(outputStream, "UTF-8"));
    String post_data = URLEncoder.encode("sensType",
          "UTF-8")+"="+URLEncoder.encode(temp,"UTF-8");
    bufferedWriter.write(post_data);
    ...
    InputStream inputStream = httpURLConnection.getInputStream();
    BufferedReader bufferedReader = new BufferedReader(new
          InputStreamReader(inputStream,"UTF-8"));
    while((line = bufferedReader.readLine())!=null)
      result += line;
    bufferedReader.close();
    inputStream.close();
    httpURLConnection.disconnect();
    return result;
  }
  return null;
}
```

Figure 8: Code snippet of the Android BackgroundWorker**.**

4.3 IoT System

An IoT system is composed of multiple interconnected programs $\{P_1, P_2, \ldots, P_n\}$ running on different hardware. Thus, such system can be represented as a graph G^*, connecting the programs through the communication channels. Formally,

$$G^* = \left\{\{G_{p_1}, G_{p_2}, \ldots, G_{p_n}\}, \{E_{c_1}, E_{c_2}, \ldots, E_{c_m}\}\right\}$$

where $G_{p_i} (1 \leqslant i \leqslant n)$ is a graph representing a program p_i, and $E_{c_i} (1 \leqslant i \leqslant m)$ is a set of edges connecting nodes belonging to different programs that communicate through some channels. When two programs communicate through a channel c, they call the functions to send and receive data introduced in Section 4.2. Namely, an edge E_c between two programs G_{p_h} and G_{p_k} can be formalized as

$$E_c = (f_h^s, f_k^r) \text{ where } f_h^s \in G_{p_i}, f_k^r \in G_{p_j} \text{ and } i \neq j.$$

4.4 Sources and Sinks

Taint analysis detects unsanitized flows from sources to sinks. Therefore, it needs a set of sources and sinks that includes the communication channels' interfaces. In general, the user manually provides sources σ^e_{src} and sinks σ^e_{snk}. However, in a system with communication channels, we must augment this set to take into account their interfaces. Therefore, we introduce a set of communication sources (σ^i_{src}) (*i.e.*, functions retrieving data from the channel) and a set of communication sinks (σ^i_{snk}) (*i.e.*, functions sending data to the channel). Formally, sources and sinks are then defined as $\sigma_{src} = \sigma^e_{src} \cup \sigma^i_{src}$ and $\sigma_{snk} = \sigma^e_{snk} \cup \sigma^i_{snk}$.

4.5 Algorithm

Algorithm 1 Multi-Program Taint Analysis

Input: $\{G_{p_1}, \cdots, G_{p_n}\}, \{\sigma^e_{src}, \sigma^e_{snk}\}, \{E_{c_1}, \cdots, E_{c_m}\}$
$\sigma_{src} \leftarrow \sigma^e_{src} \bigcup_{i=0}^{m} (f^r \in E_{c_i})$
$\sigma_{snk} \leftarrow \sigma^e_{snk} \bigcup_{i=0}^{m} (f^s \in E_{c_i})$
$conf_{p_i} \leftarrow Analysis^i(G_{p_i})$
$conf'_{p_i} \leftarrow \text{TAGCONFIG}(conf_{p_i}, \sigma_{snk}, \sigma_{snk})$
$R_i \leftarrow Analysis^r(G_{p_i}, conf'_{p_i})$
$cE ::= \bigcup_{k=0}^{m} \left\{(f_i^s, f_j^r) \in E_{c_k} : f_i^s \in R_i,\ f_j^r \in R_j\right\}$
$uGr ::= \bigcup\{(R_i, R_j) : \exists (f_i^s, f_j^r) \in cE : f_i^s \in R_i, f_j^r \in R_j\})$
$reachable ::= \{(\sigma^e_{src}, \sigma^e_{snk}) : \exists path(\sigma^e_{src}, \sigma^e_{snk}) \in uGr\}$
$taintedGraph ::= \Pi_{reachable(\sigma^e_{src}, \sigma^e_{snk})}(uGr)$
procedure TAGCONFIG($conf_{p_i}, \sigma_{snk}, \sigma_{snk}$)
    if $(\sigma^{p_i}_{src} \in \sigma_{src} \wedge \sigma^{p_i}_{snk} \in \sigma_{snk})$ **then**
        $tg_p \leftarrow$ "ti" *where* $tg_p \in conf_{p_i}$
    return $conf_{p_i}$

As discussed in Section 3, our approach relies on Julia's taint analysis engine and in particular on its GDPR checker, that allows the user to specify sources and sinks through an Excel spreadsheet. Such checker consists of two phases: Init and Report. The Init phase takes as input a program (G_p) and produces a *"source-sink-configuration"* ($conf_p$) file (another Excel spreadsheet). The configuration file consists of all possible sources (σ^p_{src}) and sinks (σ^p_{snk}) and of an empty set of tags (tg_p) (representing the types of sources and sinks) in the given program. Therefore, we formalize the init process ($Analysis^i$) as

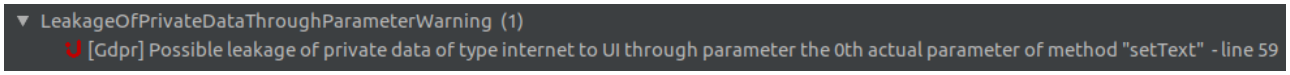

Figure 9: Results of taint analysis on the Android application.

$$Analysis^i : G_p \rightarrow \{\{\sigma^p_{src}, \sigma^p_{snk}, tg_p\} : tg_p = null\}$$

Instead, the Report phase takes a program and the tagged *"source-sink-configuration"* ($conf'_p$) (aka, the Excel spreadsheet specifying sources and sinks) as input and returns a set of flow graphs representing the paths of tainted data from source to sink. Formally, the report phase of the analysis ($Analysis^r$) of a program G_p is defined as

$$Analysis^r : (G_P \times conf'_p) \rightarrow R_p, \text{ where } R_p \subset \wp(G_P) \text{ and}$$
$$conf'_p = \{\{\sigma^p_{src}, \sigma^p_{snk}, tg_p\} : tg_p \neq null\}$$

At the end, we must integrate the results of the taint analysis on each program in isolation to get the complete trace of tainted data propagation across multiple programs. Algorithm 1 shows the process. The inputs of the system (line 1) are a set of programs $\{G_{p_1}, G_{p_2}, \ldots, G_{p_n}\}$, a set of user defined sources and sinks $\{\sigma^e_{src}, \sigma^e_{snk}\}$, and the APIs of the communication channels $\{E_{c_1}, E_{c_2}, \ldots, E_{c_m}\}$. We then generate all possible sources (σ_{src}) and sinks (σ_{snk}), combining user defined sources and sinks with that of the communication APIs (lines 2–3). Afterwards, we analyze individual programs (G_{p_i}) to generate the source-sink-configuration ($conf_{p_i}$) through the Init phase of Julia's GDPR checker (line 4). Then TAGCONFIG procedure (line 5, procedure defined at lines 11–14) tags the source-sink-configuration file based on the generated sources (σ_{src}) and sinks (σ_{src}) and returns the tagged source-sink-configuration ($conf'_{p_i}$). This last set is then used to analyze each program G_{p_i} with the Report phase of the GDPR checker (line 6) where the actual taint analysis is performed and returns all the tainted paths R_i. The algorithm then computes the edges cE connecting different programs through the same communication channel (line 7); these edges represent how data transmitted through channels might flow from a program to another. These are then added to obtain a unique graph uGr for the whole system (line 8). Finally, we compute the paths connecting user-provided sources to user-provided sinks (line 9) and project the graph on them (line 10).

4.6 Analysis of the Illustrative Example

Section 3 introduced an illustrative example where several programs (edge software, a Java servlet and an Android application) interact through different communication channels (database and Internet). We now apply the inter-program analysis Algorithm 1 to that example system.

4.6.1 Input. Our system is composed of three programs: the IoT backend [37] G_{p_1}, the Java servlet [36] G_{p_2} and the Android application [35] G_{p_3}. We specify to the algorithm the method readLine(), that reads sensor data, as user-provided source (σ^e_{src} = {BufferedReader.readLine()}) and the method setText(String), that sets the text of an Android text view, as sink (σ^e_{snk} = TextView. setText(String)). We consider, as communication channels, (i) the database where data is sent and retrieved (E_{c_1} = (DB, Put.add(byte[]...), ResultSet.get String(int)), and (ii) the Internet where data is sent and retrieved (E_{c_2} = (Internet, Output StreamWriter.write(String), BufferedReader.readLine())).

4.6.2 Taint Analyses in Isolation. Section 3 presented the results of taint analysis on each single component of the system, by using Julia's taint analysis. Here we relate them to Algorithm 1 (lines 2–6).

Figure 10 shows the complete graph obtained at the end of Algorithm 1. Julia's taint analysis represents the graph of flows of tainted data as hierarchical graphml files that can be visualized by using the yED graph editor (https://www.yworks.com/products/yed)[2].

Edge Program. The Init phase of the analysis ($Analysis^i(G_{p_1})$ at line 4 of Algorithm 1) generates the source-sink configuration ($conf_{p_1}$) for the IoT edge software, where readline() at line 12 of the Server class (Figure 3) is responsible for reading sensor data and is tagged as source (σ^e_{src}). This sensor data is then sent to the Hbase database through method add(byte[]...) (lines 22–25), tagged as sink since it belongs to the communication channel DB. This is why the warning in Figure 4 is issued. The data flow reported by Julia consists of the upper part of Figure 10.

Java Servlet. The Java servlet [36] (G_{p_2}) receives data from the Hbase database (through method getString(int) at line 21 of Figure 5, hence a source of communication channel DB), and sends it to the Android app (G_{p_3}) over the Internet (through method write(String) at line 8, hence a sink of communication channel Internet). Here, both source and sink involve communication channels. Therefore, the init phase ($Analysis^i(G_{p_2})$ at line 4 of Algorithm 1) generates a source-sink configuration file with empty tags, that gets augmented with the source and sink of the two communication channels (procedure TAGCONFIG). This tagged source-sink configuration ($conf'_{p_2}$) file and the Servlet program (G_{p_2}) are then analyzed in the report phase ($Analysis^r(G_{p_1})$ at line 6) producing the warning reported in Figure 6, together with the data flow reported in the middle of Figure 10.

Android Application. The Android application [35] (G_{p_3}) retrieves data provided by the Java servlet and shows it to the user of the app. In particular, the app receives data sent from the servlet through Internet (method readLine() at line 17 of Figure 8) and displays it locally within the app through a user-defined sink (method setText(String) at line 19 of Figure 7). By using this configuration file, the report phase of the analysis (line 6 of Algorithm 1) produces the warning reported in Figure 9 and the data flow in the lower part of Figure 10.

4.6.3 Inter-program Graph. Algorithm 1 combines the results obtained by the taint analysis of each software component. In particular, line 7 adds the edges between different programs that communicate through some channel: those from lines 22–25 of the edge software in Figure 3 to line 21 of the Java servlet in Figure 5, since

[2]Note that line numbers in this figure refer to the source code available on GitHub and not to the simplified code snippets from Section 3.

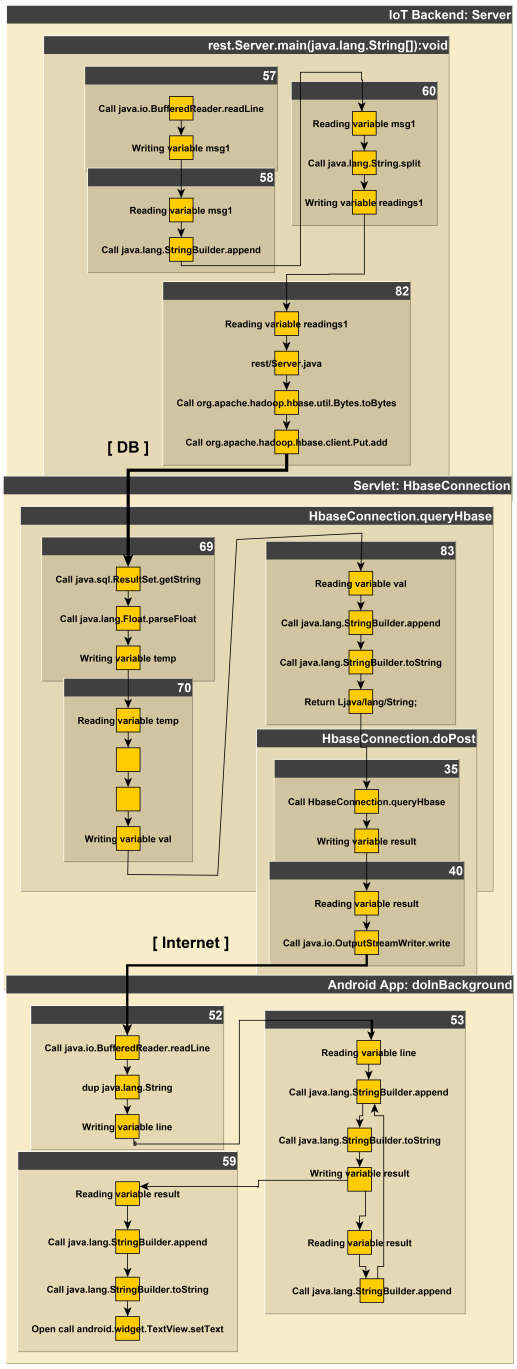

Figure 10: End-to-End Taint Propagation for the Illustrative Scenario.

these communicate through DB; those from line 8 of the Java servlet and line 17 of the background worker of the Android app in Figure 8, since these communicate through Internet. They are denoted in bold in Figure 10 and annotated with the type of communication channel that is related to that inter-program edge.

```
TAINT PROPAGATES AMONG THE INTERFACES...

It propagates from "sensor" to "UI"
through "add()" method of program "HbaseStorageRPi"
to "getString()" method of program "IOTQuery.jar""
to "Write()" method of program "IOTQuery.jar"
```

Figure 11: Result of the Cross-Program Taintedness Propagation.

Algorithm 1 then builds the whole graph (line 8), computes the paths from user-defined sources to user-defines sinks (line 9) and projects the graph on them (line 10). In the illustrative example, this produces the graph reported in Figure 10 where the user-defined source is the call to readLine() at line 12 of Figure 3 (in the upper left part of Figure 10) and the user-defined sink is the call to setText(String) at line 19 of Figure 7 (in the lower right part of Figure 10). This graph reports, exactly, a possible flow of tainted (that is, sensitive) data from source to sink, passing through several different components of an IoT system. Figure 11 reports the command-line output of such analysis showing how sensitive data crosses various software components.

5 EXPERIMENTAL RESULTS

This section discusses the implementation of the formal approach from Section 4, describes the IoT programs selected for analysis and the findings obtained on them.

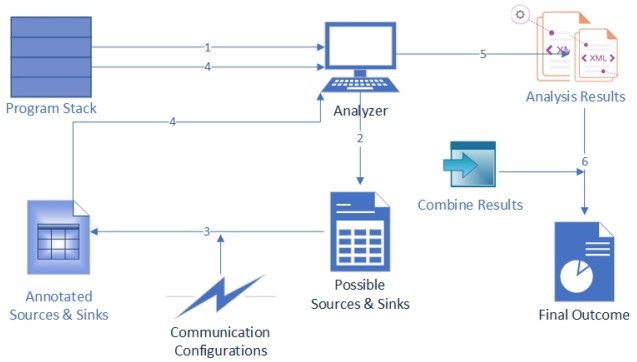

Figure 12: Architecture of the Implementation.

5.1 Implementation

The cross-program taint analysis has been implemented on top of the commercial Julia static analyzer [52] for Java and .NET bytecode, based on abstract interpretation [11, 12]. As of version 2.7.0.2, Julia implements 48 different checkers, divided into two main groups. The *basic* checkers perform simple yet comprehensive semantic controls of software issues. Instead, the *advanced* checkers perform deep semantic controls that need a complete inspection of the call graph, a precise abstraction of the heap, as well as other supporting (*e.g.*, flow [15]) analyses. As discussed in Section 3, the GDPR advanced checker[3] performs a taint analysis between a set of sources

[3]https://static.juliasoft.com/docs/latest/Gdpr.html

Application	Channel	Edge	Mobile	Flow
Doorbell	Firebase	15.42"	56.39"	From picture or surveillance camera to mobile app (Fig. 14)
Electricity Monitor	Firebase	105.40"	50.31"	From timestamp to mobile app (Fig. 15)
Color Thing	NFC	17.06"	457.23"	From user input to LEDs (Fig. 16)
BLE fun	Bluetooth	18.18"	94.11"	From shared preferences to mobile app (Fig. 17)
Robocar	Internet	51.51"	55.67"	None (Fig. 18)

Figure 13: Summary of the Results.

and sinks that the user specifies through an Excel spreadsheet. We used this analysis to develop the cross-program taint analysis.

Figure 12 shows the architecture of our analyzer. The number on the arrows specifies the order of the distinct tasks: our implementation first generates the possible set of sources and sinks from the given program, as an Excel file (edges 1 and 2). These are then tagged with the communication channels (edge 3). These tagged sources and sinks are then sent to the analyzer, along with the program, for a taint analysis based on the Report phase of the GDPR checker (edge 4). At the end of the analysis, Julia reports detailed information about tainted paths (edge 5). These are then combined into a final report (edge 6).

5.2 Experimental Setup

The analyses have been executed on a r5.xlarge Amazon Web Service machine, that features a Xeon Platinum 8000 series (Skylake-SP) processor with a sustained all core Turbo CPU clock speed of up to 3.1 GHz and 32 GB of RAM.

We scanned GitHub for repositories containing IoT systems made up of several programs. We ended up selecting five repositories based on Android Things, where an edge program and an Android application communicate through some channels. Android Things supports cloud, Bluetooth and Near Field Communication (NFC) connectivity between different IoT components. We selected the repositories that have at-least an Android or Web application along with an Android Things application (that is, an edge program). This narrowed the available repositories, since the majority is functionally repetitive, and many did not compile because of missing resources, or incorrect Gradle Build files. However, we did not find in GitHub any suitable MQTT or 6LowPAN based IoT system (composed by at least two programs) developed in Java where multiple components interacts with each other via some interfaces [23, 24].

In particular, we selected IoT systems communicating through Google Firebase [26] (Doorbell [50] and Electricity Monitor [20]), Near Field Communication (Color Thing [29]), Bluetooth (Bluetooth Low-Energy (BLE) fun [40]), and Internet (Robocar [56]).

5.3 Discussion of the Results

Figure 13 summarizes the results of the experiments. **Application** is the application name; **Channel** the communication channel; **Edge** and **Mobile** are the analysis time on the edge software and on the mobile app, respectively; **Flow** is a description of the identified flow (if any). Appendix A discusses the analysis results in details.

In four out of five cases, our approach identifies a flow from an external source to an external sink. The flow graphs produced by the analysis show, in detail, how programs process data and let it flow between components. The fact that we found a flow of sensitive data (*e.g.*, camera picture) or user input into a leakage point (*e.g.*, code that shows something to the user of a mobile app) or into a critical sink (*e.g.*, code that sets the speed of a physical device) does not mean that the program is bugged. Actually, in all cases these flows were intended to support the functionality of the app. In a single case (Robocar), manual code inspection revealed that there was no explicit flow of user input to the sink, since the mobile app correctly checks the values of the joystick and sends appropriate constant values for setting the car speed. Such results show however how taint analysis can be extended to a multi-program (*e.g.*, IoT) context and automatically infer data flows across different components.

6 CONCLUSION

Cross-program propagation of tainted data in an interactive IoT systems is highly critical and allows one to detect issues (insecure ecosystem interfaces, in particular) among the OWASP IoT Top 10 2018 [44]. This work extends the existing Julia's taint analysis to a system where multiple components communicate through some channels, as it is typical of IoT systems. It formalizes the approach and applies it to an illustrative example and to five IoT systems, publicly available on GitHub, composed of two programs. Experiments show that the technique traces tainted data flows across multiple programs in an IoT system.

This work shows the practical feasibility of leveraging taint analysis to a multi-program scenario and its effectiveness on some small case studies. However, it has limitations. In particular, it only considers communication channels as monolithic: a method that reads data from a channel is conservatively assumed to yield tainted data if any tainted data ever reached that channel. Although this is sound, it is likely overly conservative for big industrial IoT applications, that might use the same channel for several different communications. For instance, only some (columns of) tables of a database might hold tainted data and only few database queries might consequently access it, but our technique does not consider this. Since most communication occurs through strings (*e.g.*, SQL queries, IP addresses), we plan to apply some abstraction of string values [9, 10] to discard some spurious cross-program edge introduced by our technique.

REFERENCES

[1] Hisham Alasmary, Afsah Anwar, Jeman Park, Jinchun Choi, DaeHun Nyang, and Aziz Mohaisen. 2018. Graph-based comparison of IoT and android malware. In *International Conference on Computational Social Networks*. Springer, 259–272.

[2] S Alnaeli, Melissa Sarnowski, M Aman, Ahmed Abdelgawad, and Kumar Yelamarthi. 2017. Source Code Vulnerabilities in IoT Software Systems. *Advances in Science, Technology and Engineering Systems Journal* 2, 3 (2017), 1502–1507.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive

and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of PLDI '14*. ACM.

[4] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. 2018. A Lesson on Verification of IoT Software with Frama-C. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 21–30.

[5] E. Burato, P. Ferrara, and F. Spoto. 2017. Security Analysis of the OWASP Benchmark with Julia. In *Proceedings of ITASEC '17*. Venice, Italy.

[6] Z Berkay Celik, Earlence 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).

[7] S. Challa, M. Wazid, A. K. Das, N. Kumar, A. Goutham Reddy, E. Yoon, and K. Yoo. 2017. Secure Signature-Based Authenticated Key Establishment Scheme for Future IoT Applications. *Access* 5 (2017), 3028–3043.

[8] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of ISSTA '07*. ACM.

[9] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Proceedings of ICFEM '11 (Lecture Notes in Computer Science)*. Springer.

[10] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2015. A suite of abstract domains for static analysis of string values. *Softw., Pract. Exper.* 45, 2 (2015), 245–287.

[11] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints *(Proceedings of the 4th Symposium on Principles of Programming Languages (POPL))*. ACM.

[12] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks *(Proceedings of POPL '79)*. ACM Press.

[13] Ashok Kumar Das, Sherali Zeadally, and Debiao He. 2018. Taxonomy and analysis of security protocols for Internet of Things. *Future Generation Computer Systems* 89 (2018), 110 – 125.

[14] Eclipse IoT Working Group. 2019. The Three Software Stacks Required for IoT Architectures. Retrieved June, 26th 2019 from https://iot.eclipse.org/resources/white-papers/Eclipse%20IoT%20White%20Paper%20-%20The%20Three%20Software%20Stacks%20Required%20for%20IoT%20Architectures.pdf

[15] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. 2015. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proceedings of LPAR '15 (LNCS)*. Springer.

[16] Daniel Alberto Sepúlveda Estay. 2019. CyberShip-IoT: A Dynamic and Adaptive SDN-Based Security Policy Enforcement Framework for Ships. *Future Generation Computer Systems* (2019).

[17] Ivan Farris, Tarik Taleb, Yacine Khettab, and Jaeseung Song. 2018. A survey on emerging SDN and NFV security mechanisms for IoT systems. *IEEE Communications Surveys & Tutorials* 21, 1 (2018), 812–837.

[18] Pietro Ferrara, Amit Mandal, Agostino Cortesi, and Fausto Spoto. 2019. Static Analysis for the OWASP IoT Top 10 2018. In *Proceedings of SPIoT '19*.

[19] Pietro Ferrara and Fausto Spoto. 2018. Static Analysis for GDPR Compliance. In *Proceedings of ITASEC 2018*.

[20] Rebecca Franks. 2019. android-things-electricity-monitor. Retrieved June, 26th 2019 from https://github.com/riggaroo/android-things-electricity-monitor

[21] M. Frustaci, P. Pace, G. Aloi, and G. Fortino. 2018. Evaluating Critical Security Issues of the IoT World: Present and Future Challenges. *Internet of Things* 5, 4 (2018), 2483–2495.

[22] Mengmeng Ge, Jin B Hong, Walter Guttmann, and Dong Seong Kim. 2017. A framework for automating security analysis of the internet of things. *Journal of Network and Computer Applications* 83 (2017), 12–27.

[23] GitHub. 2019. 6LowPAN Repositories (Java). Retrieved June, 26th 2019 from https://github.com/search?l=Java&q=6lowpan&type=Repositories

[24] GitHub. 2019. MQTT Repositories (Java). Retrieved June, 26th 2019 from https://github.com/search?l=Java&q=MQTT&type=Repositories

[25] R. Giuliano, F. Mazzenga, A. Neri, and A. M. Vegni. 2017. Security Access Protocols in IoT Capillary Networks. *Internet of Things* 4, 3 (2017), 645–657.

[26] Google. 2019. Firebase. Retrieved June, 26th 2019 from https://firebase.google.com/

[27] F. Griffiths and M. Ooi. 2018. The fourth industrial revolution - Industry 4.0 and IoT [Trends in Future I&M]. *Instrumentation Measurement Magazine* 21, 6 (2018), 29–43.

[28] Peng Hao, Xianbin Wang, and Weiming Shen. 2018. A Collaborative PHY-Aided Technique for End-to-End IoT Device Authentication. *IEEE Access* 6 (2018), 42279–42293.

[29] Holger. 2019. Color-Things. Retrieved June, 26th 2019 from https://github.com/holgi-s/ColorThings,https://github.com/holgi-s/ColorConnection

[30] Jianwei Hou, Leilei Qu, and Wenchang Shi. 2019. A survey on internet of things security from data perspectives. *Computer Networks* 148 (2019), 295 – 306.

[31] Ralf Huuck. 2015. Iot: The internet of threats and static program analysis defense. In *EmbeddedWorld 2015: Exibition & Conferences*. 493–495.

[32] SuHyun Kim and ImYeong Lee. 2018. IoT device security based on proxy re-encryption. *Ambient Intelligence and Humanized Computing* 9, 4 (01 Aug 2018), 1267–1273.

[33] M. G. Kukkuru. 2019. Testing IoT Applications - A Perspective. Retrieved June, 26th 2019 from https://www.infosys.com/IT-services/validation-solutions/Documents/testing-iot-applications.pdf

[34] Y. Liu, Y. Kuang, Y. Xiao, and G. Xu. 2018. SDN-Based Data Transfer Security for Internet of Things. *Internet of Things* 5, 1 (2018), 257–268.

[35] Amit Kr Mandal. 2019. Android App for Plant Monitoring System. Retrieved June, 26th 2019 from https://github.com/amitmandalnitdgp/IOTApp

[36] Amit Kr Mandal. 2019. Android App for Plant Monitoring System - Servlet. Retrieved June, 26th 2019 from https://github.com/amitmandalnitdgp/IOT-EcoSyatem/blob/master/HbaseConnection.java

[37] Amit Kr Mandal. 2019. Plant Monitoring System - IoT Backend. Retrieved June, 26th 2019 from https://github.com/amitmandalnitdgp/IOT-EcoSyatem/blob/master/Server.java

[38] Amit Kr Mandal, Federica Panarotto, Agostino Cortesi, Pietro Ferrara, and Fausto Spoto. 2019. Static Analysis of Android Auto Infotainment and ODB-II Apps. *Software: Practice and Experience* (2019). In Press.

[39] Orestis Mavropoulos, Haralambos Mouratidis, Andrew Fish, and Emmanouil Panaousis. 2018. Apparatus: A framework for security analysis in internet of things systems. *Ad Hoc Networks* (2018), 101743.

[40] Gautier MECHLING. 2019. Bluetooth Low-Energy (BLE) fun - Android (Things). Retrieved June, 26th 2019 from https://github.com/Nilhcem/blefun-androidthings

[41] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. 2019. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-scale IoT Exploitations. *Communications Surveys Tutorials* (2019).

[42] J. Newsome and D. Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS '05)*. Internet Society.

[43] Mehdi Nobakht, Yulei Sui, Aruna Seneviratne, and Wen Hu. 2018. Permission Analysis of Health and Fitness Apps in IoT Programming Frameworks. In *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, 533–538.

[44] OWASP. 2019. OWASP Internet of Things (IoT) Project. Retrieved June, 26th 2019 from https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

[45] Federica Panarotto, Agostino Cortesi, Pietro Ferrara, Amit Kr Mandal, and Fausto Spoto. 2018. Static Analysis of Android Apps Interaction with Automotive CAN. In *Proceedings of SmartCom '18 (LNCS)*. Springer.

[46] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Proceedings of POPL '95)*. ACM.

[47] Rashmi Sahay, G. Geethakumari, Barsha Mitra, and Ipsit Sahoo. 2019. Efficient Framework for Detection of Version Number Attack in Internet of Things. In *Intelligent Systems Design and Applications*, Ajith Abraham, Aswani Kumar Cherukuri, Patricia Melin, and Niketa Gandhi (Eds.). Springer. In press.

[48] Trusit Shah and S Venkatesan. 2018. Authentication of IoT Device and IoT Server Using Secure Vaults. In *Proceedings of TrustCom/BigDataSE '18*. IEEE, 819–824.

[49] Daemin Shin, Vishal Sharma, Jiyoon Kim, Soonhyun Kwon, and Ilsun You. 2017. Secure and efficient protocol for route optimization in PMIPv6-based smart home IoT networks. *IEEE Access* 5 (2017), 11100–11117.

[50] Dave Smith. 2019. doorbell. Retrieved June, 26th 2019 from https://github.com/androidthings/doorbell

[51] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2019). To appear.

[52] JuliaSoft srl. 2019. Julia Static Analyzer. Retrieved June, 26th 2019 from https://juliasoft.com/

[53] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of PLDI '09*. ACM.

[54] Yifei Xu, Ting Liu, Pengfei Liu, and Hong Sun. 2018. A Search-based Firmware Code Analysis Method for IoT Devices. In *2018 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–2.

[55] Aws Alaa Zaidan, Bilal Bahaa Zaidan, MY Qahtan, OS Albahri, AS Albahri, Mussab Alaa, Fawaz Mohammed Jumaah, Mohammed Talal, Kian Lam Tan, WL Shir, et al. 2018. A survey on communication components for IoT-based technologies in smart homes. *Telecommunication Systems* 69, 1 (2018), 1–25.

[56] Antonio Zugaldia. 2019. Android Robocar. Retrieved June, 26th 2019 from https://github.com/zugaldia/android-robocar

A DETAILS ABOUT EXPERIMENTAL RESULTS

In this Appendix, we discuss the details of the results of the analysis on the programs we reported in Figure 13.

A.1 Firebase: Doorbell and Electricity Monitor

Most IoT systems rely on different cloud services as communication channel between the edge software and the mobile applications. A very popular choice is Google Firebase [26]. We have considered two different IoT systems that communicate through Firebase: Doorbell [50] and Electricity Monitor [20].

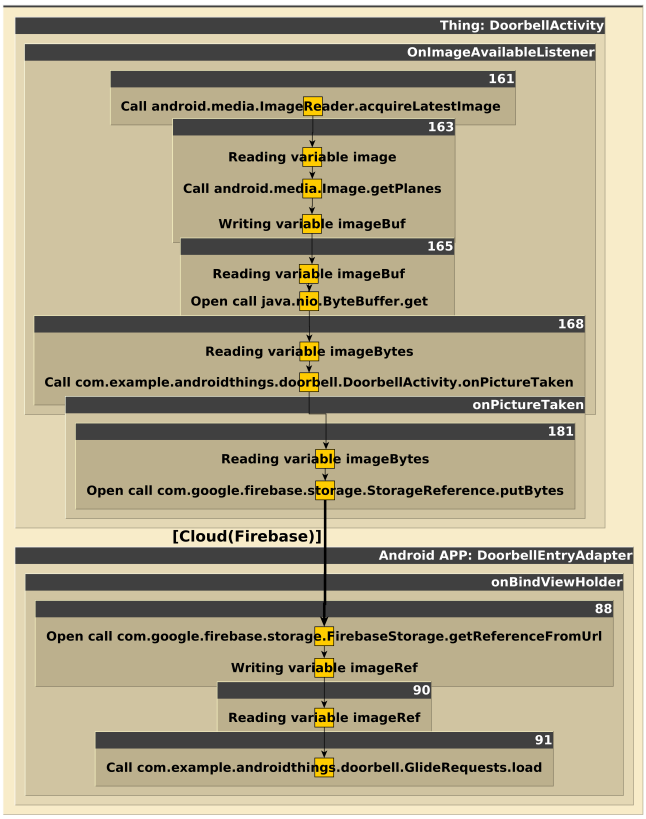

Figure 14: Doorbell Analysis Result.

Android Things Doorbell [50] implements a smart doorbell that captures the image of the visitor who presses the bell button. The picture obtained from the camera is processed through Google's Cloud Vision API; the edge software then uploads it to a Firebase database, together with Cloud Vision annotations and metadata. The companion Android app accesses the database and presents data to the user. For the analysis of this program, we tagged as sources and sinks of the communication channel StorageReference.putBytes (called at line 181 of DoorbellActivity) and FirebaseStorage.getReferenceFromURL (called at line 88 of DoorbellEntryAdapter), respectively. Furthermore, we specified ImageReader.acquireLatestImage as external source (*i.e.*, the Android API that retrieves a camera image, called at line 162 of DoorbellActivity) and GlideRequests.load as external sink (*i.e.*, the method of the mobile app that displays an image, called at line 91 of DoorbellEntryAdapter). The programs along with the Excel spreadsheet tagging these sources and sinks are passed to Julia's GDPR checker. Figure 14 reports the flow graph produced by the taint analysis, where the results on the two programs have been connected (Thing and Android App). In addition, Algorithm 1 adds the bold arrow that represents a data flow between components. This result shows that the edge software retrieves the image and stores it in Firebase, where the mobile app retrieves it to show it to the user. Therefore, our approach detects the information flow from source (picture taken from a camera) to sink (image shown in a mobile app).

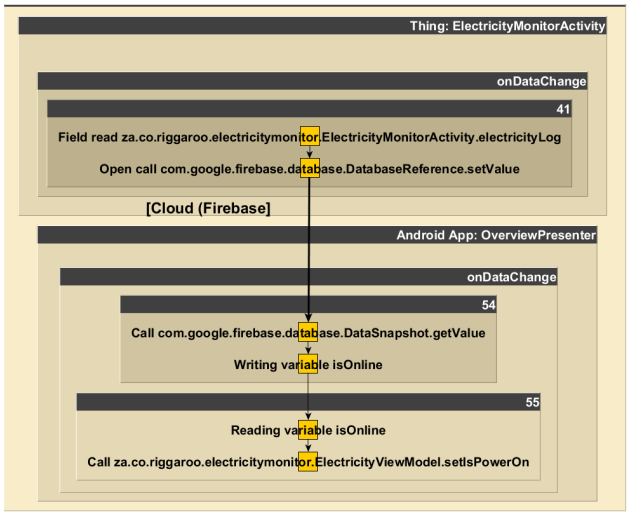

Figure 15: Electricity Monitor Analysis Result.

The second IoT system in this category is Electricity Monitor [20], that tracks the availability of electricity and notifies the user about black-outs, in an Android app. It uses Firebase as communication means between the edge software and the mobile app. Similarly to Doorbell, we add as source and sink of the communication channel methods DataSnapshot.getValue (called at line 74 of OverviewPresenter) and DatabaseReference.setValue (called at line 41 of ElectricityMonitorActivity), respectively. The external source is the ElectricityLog instance in field ElectricityMonitorActvity.electricityLog (since it keeps track of the status of electricity). The external sink is setIsPowerOn of ElectricityViewModel, since the information contained there is shown to the user of the mobile app. Note that we might have chosen additional external sinks since the view model contains several other fields, but we focused on a single specific value since the other cases would be identical. The analysis of these programs, with this configuration, generates the flow in Figure 15. It shows that the edge software accesses the log of the status of electricity and immediately retransmits it to Firebase; moreover, the mobile app accesses this data and shows the electricity status to the user, by passing this data to the view model.

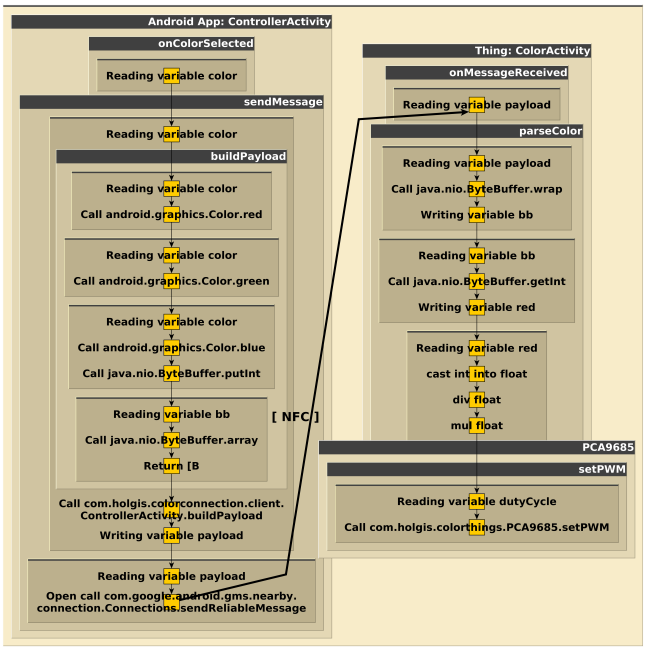

Figure 16: Color Thing Analysis Result.

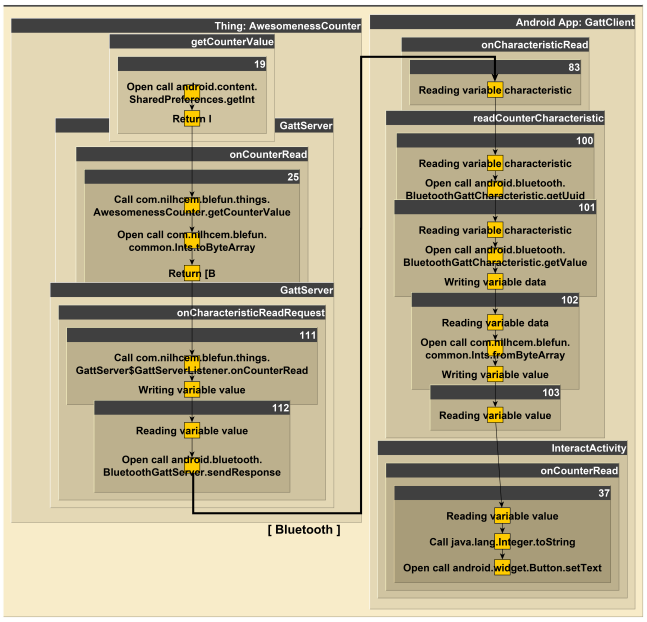

Figure 17: BLE Fun Analysis Result.

A.2 Near Field Communication: Color Thing

Near Field Communication (NFC) allows short-range wireless connectivity. The Color Thing program [29] relies on NFC to allow communication between an edge program and a mobile app, that changes the colors of some LEDs connected to a Raspberry Pi 3. The source of the communication channel is the parameter of the event listener Activity.onMessageReceived (used at line 238 of ColorActivity). Method Connections.sendReliableMessage (called at line 310 of ControllerActivity) is a sink of the communication channel. The external source is the method parameter of the event listener ControllerActivity.onColorSelected, that receives the input of the user through the mobile app at line 272 of this activity. The external sink is PCA9685.setPWM, that sets the LEDs' color. As in the other examples, the programs are analyzed with this configuration. The result, in Figure 16, shows that the Android app reads the user input, elaborates an adequate payload transforming the user input into an RGB color and transmits it through NFC; the edge software receives this input, processes the value and transmits a coherent value to the hardware device to set the LEDs' color.

A.3 Bluetooth: BLE Fun

Bluetooth in another way of communication between nearby devices. The Bluetooth Low-Energy (BLE) fun - Android (Things) program [40] relies on Bluetooth Low Energy technology to communicate between an Android Things program and a mobile app. It simply sends a counter from the edge software to the Android app. For this communication channel, source and sink are the second parameter of the event listener BluetoothGattCallback.onCharacteristicRead, accessed at line 83 of GattClient, and the parameter of BluetoothGattServer.sendResponse, called at line 112 of GattServer, respectively. The initial value of the counter is read by calling SharedPreferences.getInt at line 19 of AwesomenessCounter and is consequently tagged as external source. The external sink is Button.setText, called at line 37 of InteractActivity, since it shows the value to the user of the mobile app. Figure 17 reports the analysis results: the edge software accesses a counter stored in shared preferences and transmits it, after some computation, through Bluetooth; the mobile app receives it and shows it to the user.

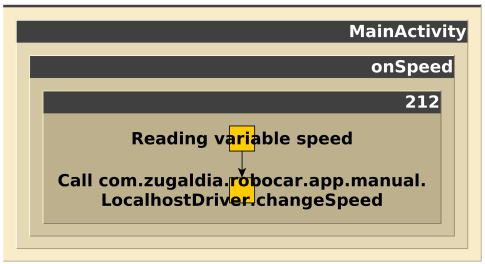

Figure 18: Robocar Analysis Result.

A.4 Internet: Robocar

As a last example, we considered an IoT system that allows one to drive a little, remote-controlled, autonomous car built with Android Things [56]. The system consists of a mobile app to control the car and of some edge software, that sends driving instructions to the car. The two programs communicate through standard HTTP requests and responses. The repository provides specific interfaces to send and receive data. Hence, source and sink of the communication channel are the parameter of HTTPRequestListener.onSpeed(RobocarSpeed) and the value passed to RobocarClient.setSpeed(int,int), respectively. In addition, the external source

is the second parameter of GameControllerActivity.handleJoystick-Button Event(View,MotionEvent), since this activity manages the joystick of the mobile app. The external sink is LocahostDriver. changeSpeed, called at line 212 of MainActivity. Figure 18 reports the result of the analysis that, this time, did not find any explicit flow from the external source to the external sink. Although the edge software does receive data from the communication channel and sends it to the external sink (as the figure reports), the mobile app does not send data to the Internet: it just performs some checks of user input (retrieved through the external source) and sends distinct constant values based on such checks (see GameController-Activity.handleJoystickButtonEvent). Hence, there is no explicit flow from the external source to the communication channel, while there is an implicit flow, not detected by taint analysis. From our point of view, this means that the program correctly translates the user input into *sanitized* (namely, constant) values. In that way, the software prevents the user (and potentially an attacker) from sending arbitrary speed values that might damage the device.