International Journal on Software Tools for Technology Transfer (2021) 23:71–88
https://doi.org/10.1007/s10009-020-00592-x

FOUNDATION FOR MASTERING CHANGE

Special Issue: SPIoT 2019

Static analysis for discovering IoT vulnerabilities

Pietro Ferrara[1,2] · Amit Kr Mandal[3] · Agostino Cortesi[1] · Fausto Spoto[4]

Published online: 24 November 2020

Abstract
The Open Web Application Security Project (OWASP), released the "OWASP Top 10 Internet of Things 2018" list of the high-priority security vulnerabilities for IoT systems. The diversity of these vulnerabilities poses a great challenge toward development of a robust solution for their detection and mitigation. In this paper, we discuss the relationship between these vulnerabilities and the ones listed by OWASP Top 10 (focused on Web applications rather than IoT systems), how these vulnerabilities can actually be exploited, and in which cases static analysis can help in preventing them. Then, we present an extension of an industrial analyzer (Julia) that already covers five out of the top seven vulnerabilities of OWASP Top 10, and we discuss which IoT Top 10 vulnerabilities might be detected by the existing analyses or their extension. The experimental results present the application of some existing Julia's analyses and their extension to IoT systems, showing its effectiveness of the analysis of some representative case studies.

Keywords IoT security · Static analysis · OWASP IoT Top 10 · IoT privacy · Insecure IoT ecosystem interface · Static analysis

1 Introduction

In most of the attacks targeting Internet of Things (IoT) systems [11,19,41,69], a common IoT device is used to intrude into the system, and exploit the larger network to which IoT devices are connected. According to Gartner, by 2020, more than 25% of cyber-attacks on enterprises will target IoT systems [39]. Therefore, cyber-attacks are moving their targets from vulnerable computers to IoT devices. The ubiquitous nature of IoT ecosystems goes beyond the boundaries of traditional network security, and it widens the attack surface, as interconnected devices operate from different physical locations and network layers. In such scenarios, attackers may use automation tools to simulate authorized operations on legitimate devices to create a springboard effect where they may exploit minor vulnerabilities. IoT systems usually comprise at least three major components: devices, cloud, and companion applications [20]. Each of these components may contain security vulnerabilities, and when combined together such issues might increase their severity exponentially because of various computational and network features of IoT ecosystems.

In general, a "Thing" in IoT (aka, device) executes (embedded) software on microcontrollers (MCUs) with a small memory footprint, where autonomy, reconfiguration, safety, and fault tolerance are highly sought to meet functional safety requirements. Moreover, IoT devices rely on cloud services (e.g., to communicate and store data). This yields to communications between devices physically located at different places through different communication mediums supported by distinct protocols. This diversity may highly compromise the integrity of device data that could be

Work partially supported by the Project "ADditive Manufacturing and Industry 4.0 as innovation Driver (ADMIN 4D)".

✉ Pietro Ferrara
pietro.ferrara@unive.it

Amit Kr Mandal
amitmandal.nitdgp@gmail.com

Agostino Cortesi
cortesi@unive.it

Fausto Spoto
fausto.spoto@univr.it

1 Università Ca' Foscari, Venice, Italy
2 JuliaSoft, Verona, Italy
3 SRM University, Amaravati, AP, India
4 Università di Verona, Verona, Italy

sensitive or under IoT user's control. Tracking this across diversified communication mediums (e.g., WiFi, Bluetooth, and NFC) is difficult and error prone as it may propagate through multiple layers and devices. Furthermore, tracking malicious flow can be overwhelming especially when there are millions of devices communicating through the same channels. Finally, the cloud environment facilitates the development of sophisticated programs allowing the access and control of devices from remote locations. Such software may come in the form of Web or mobile applications, or even enterprise programs. This software, supported by ubiquitous connectivity, may become a suitable attack surface for intruders. Ideally, applications based on cloud services should use identity management, access management, identity governance, and authentication services to enforce security. However, due to computational constraints of IoT devices, a full-flagged implementation of these services may not be possible. This increases the risk of exploitation of weak points in the companion applications to gain significant remote control of IoT systems.

Several standards and certifications have been proposed over the years in order to prevent software vulnerabilities. The Open Web Application Security Project (OWASP) is probably the most notable and popular effort in this context. Among the many projects carried on by this foundation, the OWASP Top 10 project lists the most dangerous security vulnerabilities in Web applications. Similarly, the OWASP Internet of Things Top 10 project focuses on the 10 most critical risks for the IoT ecosystem. OWASP Top 10 policy refers to the Top 10 as an "awareness document" which may be adopted by industries to improve their product development processes in order to minimize and/or mitigate the most critical security risks. The vulnerabilities listed by OWASP IoT Top 10 in 2018 include, among others, weak and hardcoded passwords, insecure network interfaces, lack of update mechanisms, and insecure ecosystem interfaces. The diversity of these vulnerabilities poses a critical challenge to adopt a robust solution for their detection and mitigation.

The first version of OWASP Top 10 dates back to 2003. During the last 15 years, this list has widely impacted the processes to enforce cybersecurity in Web applications driving the development and adoption of various tools to prevent software vulnerabilities. Static analysis focuses on their detection at compile time without executing the program. Since this approach does not need to have concrete values to expose different execution paths, it can navigate the code more pervasively at the price of introducing some forms of approximation. Different kinds of static analyzers have been implemented and commercialized, ranging from syntactic analyzers (mostly considering small portions of the code in isolation) to tools based on formal methods (thus building up a complete semantic model of the programs and approximating how different software components might interact). OWASP Top 10 pushed industrial static analyzers to detect the software vulnerabilities listed among its categories. Therefore, various types of analyses have been developed in order to detect injection and XSS vulnerabilities, leakages of sensitive data, hardcoded passwords, as well as usage of weak cryptographic algorithms.

Can we therefore conclude that the application of static analysis to detect security IoT vulnerabilities is straightforward? Unfortunately, no. In particular, the IoT ecosystem comprises quite diversified types of software, like "Web, backend API, cloud or mobile interfaces," and embedded software as well. Each software components is potentially written in a different programming language (e.g., C for embedded software, and Java for Web and mobile applications), executes independently, and interacts over various communication channels. Existing static analyzers are mostly focused on individual programs. If independent programs interact with each other, then the analysis considers each program "in isolation," missing some potential vulnerabilities, or producing too many false alarms. In addition, existing static analyzers do not possess any knowledge or interface to specify the physical world of an IoT system.

In addition, the IoT ecosystem poses new challenges. For this reason, few years ago OWASP opened the IoT project[1] that released the IoT Top 10 list.[2] Like OWASP Top 10, this list is aimed at impacting how enterprises develop and debug their software in order to prevent vulnerabilities. However, this scenario is more recent and quickly evolving.

In this paper, we discuss each category of the OWASP IoT Top 10 list, and if and how such vulnerabilities can be prevented by means of static analysis. In particular, we compare them with OWASP Top 10 category, and how the static analyzers developed w.r.t. these vulnerabilities can be applied to IoT software as well. In addition, we extend an existing industrial static analyzer (Julia's Injection checker in particular) to properly address the novel challenges arising from IoT systems. We present some preliminary experimental results that show that our extension is in position to precisely discover security vulnerabilities specific of IoT systems.

The rest of the paper is structured as follows. Section 2 discusses related literature, while Sect. 3 introduces how static analysis has been applied to address some of the categories of the OWASP Top 10 list. Section 4 presents in detail the OWASP IoT Top 10 list, and for each category it discusses if and how static analysis can help to prevent the particular type of software vulnerabilities. Section 5 informally introduces the extension of the Julia static analyzer we developed to address some of the IoT Top 10 issues not yet covered by

[1] https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project.

[2] https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10.

existing analyses, while Sect. 6 discusses some preliminary experimental results of this implementation. Finally, Sect. 7 concludes.

2 Related work

The diversity of IoT devices continues to grow, and nowadays it ranges from simple sensing and actuating devices to complex systems like connected vehicles, smart televisions and cameras. This diversification is closely related to what happened in other computing paradigms such as wireless sensor network (similar core concepts), edge and cloud computing (similar adoption of existing technology). On the one hand, IoT takes advantage of the most sophisticated technologies. On the other hand, it also inherits and increases the security concerns of these technologies. To make it even worse, in addition to the existing vulnerabilities, it also introduces some unique security threats due to its specific system architecture.

In this context, Assiri et al. [4] reviewed the security and privacy issues associated with IoT ecosystem. Das et al. [18] carried out a comparative study of different security protocols of IoT. In this process, they presented a taxonomy for security protocols used in IoT which includes device authentication, access control, privacy preservation, etc. Similarly, Frustaci et al. [29] and Neshenko et al. [52] presented a taxonomy of the IoT security issues based on the perception, transportation, and application. Tweneboah-Koduah et al. [67] analyzed the taxonomy of various security issues, whereas Ge at al. [30] devised an IoT security model consisting of five phases (namely data processing, security model generation, security visualization, security analysis, and model updates). This model is capable of analyzing IoT security strategies based on well-defined security metrics. Mavropoulos et al. [50] proposed a class-based notation for the architectural modeling languages and corresponding mechanism for transition between different models. Khattak et al. [44] discussed the various components of an IoT architecture from the context of perception layers security. Here, they categorized and classified possible attacks at different layers in their architecture. Yoon et al. [71] proposed an architecture for remote security management of IoT devices to prevents various threats in advance. Urien [68] devised an architecture where a TLS server running on a secure chip is used to secure the communication among various devices. This server facilitates strong mutual authentication with the clients and serves as an identity module. However, most of these security frameworks only outlines the IoT system vulnerabilities, and they do not provide any analysis of the major security issues listed in OWASP IoT Top 10.

From another perspective, the limited computational resources prevent IoT devices from implementing advanced authentication mechanism. Thus, device authentication is a weak point in the IoT security landscape. The Mirai malware exploited such devices to launch a DDoS attack in the larger network. El-Hajj et al. [21] provided an analysis of the different authentication mechanisms based on the multi-criteria classification. They compared and analyzed the existing authentication protocols to evaluate their relative advantages and disadvantages. Hao et al. [36] devised a secure device authentication mechanism by integrating physical security with the asymmetric cryptography. The cryptographic key is generated by estimating the device features such as intermediate nodes and radio-frequency. The experimental results demonstrate a more effective protection against various common attacks. Bhawiyuga et al. [5] proposed a token-based authentication mechanism on the MQTT protocol for resource-constrained devices. The devised mechanism comprises four components (namely publisher, subscriber, MQTT broker, and token authentication server). Here, the publisher/subscriber first submits its credentials to the authentication server to get the token. After obtaining a valid token, it can store and use it for further authentication. Shah et al. [60] presented mutual authentication mechanisms based on multiple keys. A secure vault is used for sharing the keys among many IoT devices. Initially, keys of the secure vault are provided which changes after a successful communication session. Similarly, a signature-based authenticated mechanism for the IoT devices was introduced by Challa et al. [9]. The mechanism is simulated using NS2 and later analyzed using Burrows–Abadi–Needham logic. Finally, Alizai et al. [3] proposed a mechanism where devices are allowed into a network only if they pass a multi-factor authentication process. Such an approach helps in mitigating the common attacks like replay and man in the middle by using nonce and timestamps.

A fully flagged authentication process is often dubbed as a too-costly mechanism toward ensuring device security. Thus, in order to contrast such limitation one could secure the network by using technologies like enforcing various layers of defense, segregating devices into separate networks using firewalls, etc. [35]. In this context, Zaidan et al. [72] explored the security challenges of existing communication components for IoT-based smart homes. Sahay et al. [23] proposed a framework called CyberShip-IoT, which mitigates network traffic attacks by leveraging the software defined network (SDN) paradigm. Chze et al. [10] devised a multi-hop routing protocol for secure communication among IoT devices. This approach authenticates a device through multilayer parameters before forming a new network for enhancing the security of the communication. Farris et al. [24] analyzed security issues of network functions virtualization (NFV) and SDN from the perspective of IoT. They also depicted the critical security challenges needed to be addressed for SDN and NFV based security mechanisms when adopted by IoT systems. Kim et al. [45] devised a trustworthy networking system based on self-certifying ID (SCID), whereas Shin et al. [61] used trust between Proxy Mobile IPv6 (PMIPv6) domain and IoT systems to device a new protocol for addressing various security issues. The devised protocol supports features like handover management, mutual authentication, key exchange, etc. Giuliano et al. [32] analyzed IoT capillary networks for various IP and non-IP IoT devices to propose an algorithm based on secure key renewal mechanism for better network security. Although these approaches enhance security of IoT systems, they usually incur a substantial implementation complexity, and/or impose a significant runtime overhead which may hinder the performance as data is transferred using these secure network channels.

Typically, in an IoT system, the values measured from sensors often control the physical behavior of other devices. Thus, security of transmitted data in the IoT ecosystem is very important for preventing attacks [46]. However, the traditional cryptographic algorithms are not suitable for the resource constraint IoT devices. For this purpose, Kim et al. [46] devised a mechanism by implementing proxy re-encryption for transmitting data with minimal encryption overhead. Sahay et al. [59] prevented the flow of malicious data in the system by devising a mechanism for detecting the malicious nodes which are vulnerable to version number attacks, whereas Hou et al. [38] combined the concept of IoT architectures and data life cycles to devise a three-dimensional approach for exploring IoT security. Singh et al. [62] presented an overview of the blockchain technology and its implementation toward enhancing the security of IoT using blockchain, whereas Jeon et al. [42] uses MySQL's Mobius configuration to devise a novel IoT server platform supported by a blockchain for secure storage and retrieval of sensor data. Further, Sollins et al. [64] addressed the issue of conflicts in the collection, usage, and management of large volume of data from the perspective of IoT security and privacy requirements. However, like many other security enforcing mechanisms, these approaches consider the data flow from an IoT end-point devices through the Internet to a cloud layer (or vice versa) at runtime to ensure secure usage of IoT data, but they do not track how this data flows through different software layers statically.

In this context, static program analysis [14] can be very useful for determining taintedness of the data (e.g., sensitive or user controlled data) propagating across different IoT layers. In particular, taint analysis [53,65,66,66] tracks if something from a source (e.g., methods retrieving user input or sensitive data) flows into a sink (e.g., methods sending data to Internet or executing SQL queries) without being sanitized (e.g., encrypted or escaped). This approach has been widely applied to the detection of SQL injections in Web applications [66], leakages of sensitive data [26,27], etc. A first attempt to apply such approach to a scenario similar to IoT was performed by Mandal et al. [47] and Panarotto et al. [58], that utilized this approach to detect leakages and injection vulnerabilities in Android automotive apps [49]. Huuck [40] discussed the use static code analysis to detect some of these types of issues. Similarly, Celik et al. [8] 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." Further, taint analysis should be performed over multiple programs, 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.

3 Static analysis of OWASP Top 10 vulnerabilities

3.1 OWASP Top 10

OWASP Top 10 [57] is one of the flagship and most popular OWASP projects.

> The OWASP Top 10 is a powerful awareness document for Web application security. It represents a broad consensus about the most critical security risks to Web applications. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.
> We urge all companies to adopt this awareness document within their organization and start the process of ensuring that their Web applications minimize these risks. Adopting the OWASP Top 10 is perhaps the most effective first step toward changing the software development culture within your organization into one that produces secure code.

This project lists 10 categories of security vulnerabilities of Web applications in order of relevance. The first version of this classification was released in 2004, and it has updated several times. The first two columns of Table 1 report the 2017 version. Over the years, OWASP Top 10 kept the pace with the changes of the continuously evolving cybersecurity world, where new vulnerabilities are discovered and exploited as soon as the previous ones were detected and fixed. OWASP Top 10 heavily impacted the focus of security

Table 1 OWASP Top 10 2017

ID	OWASP Top 10 2017 (application security)	Coverage	Julia
A1	Injection	Full	Injection
A2	Broken authentication	Partial	Passwords, cryptography
A3	Sensitive data exposure	Full	GDPR
A4	XML external entities (XXE)	Full	XXE
A5	Broken access control		
A6	Security misconfiguration		
A7	Cross-site scripting (XSS)	Full	Injection
A8	Insecure deserialization		
A9	Using components with known vulnerabilities	Full	Vulnerable-components
A10	Insufficient logging and monitoring		

assessments, and various commercial tools focused on the detection of these software vulnerabilities.

3.2 Static analyzers

Static analysis detects bugs at compile time without executing the code. While dynamic analysis (e.g., testing) needs specific execution states in order to expose different execution paths, static analysis can abstractly reason about all different paths of execution. However, it needs to introduce some form of approximation in order to represent the execution of a program and prove properties on them. In particular, static analysis tools build a semantic model of a software at compile time without executing it, and then check various properties on that model. Nowadays, several standards and regulations (e.g., MISRA DO-178C, IEC 61508, ISO 26262, and IEC 62304) require the application of such tools. This pushed the development and commercialization of various industrial static analyzers like ASTREE [17] and GrammaTech CodeSonar [34].

Similarly, OWASP Top 10 pushed static analyzers to the detection of security vulnerabilities on the back-end of Web servers. In this paper, we will focus on the Julia static analyzer [43]. Note that, while other commercial analyzers like SonarQube [1] exist, they cover all the same types of vulnerabilities. Therefore, we chose Julia as a representative example since the most part of its analyses has been formalized and published.

Julia implements an abstract interpretation-based engine for the analysis and verification of Java bytecode and CIL. It contains a call graph builder as well as several denotational and constraint-based analyses that rely on an intermediate representation of bytecode. Julia currently features about 50 *checkers*, ranging from a sound taint analysis engine [22,65] to superficial analyses to detect a large set of typical errors in software, such as null-pointer accesses, nontermination, and wrong synchronization.

3.3 OWASP Top 10 coverage by static analyzers

The last column of Table 1 reports the coverage of the various OWASP Top 10 categorizes by static analysis.

In particular, categories A1 (Injection), A3 (Sensitive Data Exposure), A4 (XML External Entities), and A7 (Cross-Site Scripting) can be detected through taint analysis [66], that is, an analysis that tries to detect if a value coming from a source (e.g., methods retrieving some user input) flows into a sink (e.g., methods executing SQL queries) without being sanitized (e.g., properly escaped). While the set of sources and sinks is different for these analyses, the same taint analysis can be applied to detect security vulnerabilities such as SQL injections and XSS [6] (Julia's Injection checker) as well as to the detection of leakages of sensitive data [26,27] (GDPR checker).

Category A2 comprises a wide range of different checks. Some of them, like hardcoded passwords (checker Passwords) and weak cryptographic algorithms (checker Cryptography), can be (partially) detected by means of static analysis, while others, like the prevention of automated attacks, require runtime analyses and/or monitoring.

Instead, A9 (Using Components with Known Vulnerabilities) can be fully detected by specific tools such as the OWASP Dependency Check.[3] Julia embeds such detection in the VulnerableComponents checker.

4 Security vulnerabilities of IoT software

In March 2018, OWASP released the "2018 Internet of Things Top 10" [56] list of the high-priority security vulnerabilities for the IoT ecosystem.

> The OWASP Internet of Things Project is designed to help manufacturers, developers, and consumers better

[3] https://www.owasp.org/index.php/OWASP_Dependency_Check.

understand the security issues associated with the Internet of Things, and to enable users in any context to make better security decisions when building, deploying, or assessing IoT technologies.
The primary theme for the 2018 OWASP IoT Top 10 is simplicity. Rather than having separate lists for risks vs. threats vs. vulnerabilities—or for developers vs. enterprises vs. consumers—the project team elected to have a single, unified list that captures the top things to avoid when dealing with IoT Security

Various organizations released detailed guidelines for IoT security targeting different industries. Instead, the OWASP IoT Top 10 provided a generic vulnerability classification by creating a list consisting of very-critical issues relevant for manufacturers, enterprises, and consumers at the same time. This section re-caps the security vulnerabilities of OWASP, reinforcing it with more context and clarification. Furthermore, it also analyzes the feasibility of detecting such vulnerabilities by static analysis means. In the rest of this section, for each category of the OWASP IoT Top 10 list we recall its description, we introduce a code snippet explaining it, and we discuss if and how static analysis techniques can be applied to discover and/or prevent this kind of vulnerabilities.

I1: Weak, guessable, or hardcoded passwords

Use of easily brute forced, publicly available, or unchangeable credentials, including backdoors in firmware or client software that grants unauthorized access to deployed systems.

Keeping a default password during system development is (unfortunately) a common practice, and sometimes such password might have been hardcoded within the program. However, critical problems may occur when sensitive information (such as credentials, encryption keys, and certificates) is hardcoded. To make this issue even worse, the IoT developer/manufacturer keeps such information exactly same for all the instances of the device/applications. This is easily exploitable by the intruders to gain unauthorized access for the entire product-line in general by using simple brute force, or a reverse engineering approach.

Listing 1 Vulnerable use hardcoded IP Address.

```
class IPaddress {
  String ipAdd = "172.16.254.1";
  public final Connection getConnection() throws
        SQLException {
    return DriverManager.getConnection(
 "jdbc:mysql://"+ipAdd+"/dbName", "admin", "54gvvc'&");
  }
}
```

Examples For instance the code snippet reported in Listing 1 hard codes (i) the IP address of the server (field `ipAdd`), (ii) the username (`admin`), and (iii) the password (`54gvvc'&`). An attacker might reverse engineer this code obtaining the admin credentials.

Listing 2 Vulnerable use hardcoded cryptographic keys.

```
byte[] key = {1, 2, 3, 4, 5, 6, 7, 8};
SecretKeySpec spec = new SecretKeySpec(key, "AES");
Cipher aes = Cipher.getInstance("AES");
aes.init(Cipher.ENCRYPT_MODE, spec);
return aesCipher.doFinal(secretData);
```

Similarly, keeping cryptographic keys in the source code can cause serious security issues as source code can be widely shared in an enterprise environment and easily detectable. The code snippet in Listing 2 depicts such an example.

Instead, weak (e.g., `123456`) and guessable (that is, already disclosed and publicly known) passwords are not hardcoded inside the program, but they are stored in property files or a database, and often they are provided by the user.

Static analysis Hardcoded passwords within the program can be detected by static analysis means. For instance, Julia provides the Passwords checker[4] that detects hardcoded passwords (CWE 259) and passwords that are retrieved from property files (CWE 522). However, this covers only partially I1, since weak and guessable passwords can be detected only (i) through dynamic analysis (e.g., penetration testing) on a deployed system, or (ii) dynamic monitoring of the passwords selected by users checking that they conform some standards (e.g., at least 8 characters, special characters, etc.) and that they have not been previously disclosed in some public databases (e.g., https://haveibeenpwned.com/). Thus, category I1 can be partially detected using static analysis.

I2: Insecure network services

Unneeded or insecure network services running on the device itself, especially those exposed to the internet, that compromise the confidentiality, integrity/authenticity, or availability of information or allow unauthorized remote control.

On the one hand, a common approach to secure network services is to monitor an IT system through firewalls and intrusion detection systems in order to prevent, recognize, and block external attacks. In an IoT environment, this requires to implement similar network security measures, since things communicate over the very same Internet. On the other hand, software running in the system should use secure services (e.g., communications through `https` rather than `http`).

4 https://static.juliasoft.com/docs/latest/Passwords.html.

Listing 3 Vulnerable use of HTTP communication.

```java
public class HttpClientExample {
  private void sendGet() throws Exception {
    String url = "http://www.example.com/search?q=get";
    HttpClient client = new DefaultHttpClient();
    HttpGet request = new HttpGet(url);
    ...
  }
  private void sendPost() throws Exception {
    String url = "http://example.com/wcResults.do";
    HttpClient client = new DefaultHttpClient();
    HttpPost post = new HttpPost(url);
    ...
  }
}
```

Examples For instance, using unencrypted services over HTTP while sharing sensitive information may lead to critical attacks like DNS hijacking. Listing 3 depicts an insecure use of such service.

Listing 4 Vulnerable use of UPnP Service.

```java
private void doPortForwarding() {
  PortMapping[] desiredMapping = new PortMapping[2];
  desiredMapping[0] = new PortMapping(8123,
       InetAddress.getLocalHost().getHostAddress(),
       PortMapping.Protocol.TCP, " TCP POT Forwarding");
  desiredMapping[1] = new PortMapping(8123,
       InetAddress.getLocalHost().getHostAddress(),
       PortMapping.Protocol.UDP, " UDP POT Forwarding");
  UpnpService upnpService = new UpnpServiceImpl();
  RegistryListener registryListener = new
       PortMappingListener(desiredMapping);
  upnpService.getRegistry().addListener(registryListener);
  upnpService.getControlPoint().search();
}
```

Again, for hassle free set up of a new IoT connection devices often use a technology called UPnP to allow IoT devices to open certain ports in the router and allow traffic through them. However, researchers [2] uncovered serious security issues with UPnP particularly if a IoT device in the network is exploited UPnP can give intruders remote control, thus allowing them to steal sensitive information and access to the other devices connected to the network. Further, the compromised devices can be used to launch botnets to instigate distributed denial of service (DDoS) campaigns. The code snippet listed in Listing 4 shows that the port 8123 is opened for the public traffic.

Static analysis Static analysis can detect when a program relies on insecure network communications. For instance, a standard string prefix analysis [12,13] approximates the prefixes of a string, and it would be in position to detect what communication protocols are used when building up URLs. In addition, static analysis can automatically track and report what ports are used by a program. However, currently Julia does not implement such analyses.

Instead, securing the network services involves the monitoring of the overall system (and not only the analysis of a software), and thus it requires runtime analysis. Therefore, static analysis can cover partially category I2.

I3: Insecure ecosystem interfaces

> Insecure Web interface, supporting APIs, and mobile interfaces of the IoT ecosystem increases the attack surface of the device or its related components. Common issues include a lack of authentication/authorization, lacking or weak encryption, and a lack of input and output filtering.

This category encompasses various OWASP Top 10 categories, since these were designed to improve the security of a specific layer of the IoT ecosystem (that is, Web applications). In particular, the lack of authentication/authorization and weak encryption is part of A2 (Broken Authentication), while the lack of input and output filtering is part of A1 (Injection) and A7 (XSS).

An IoT system comprises various software layers that communicate each other. Therefore, a vulnerability in one of these layers might compromise the security of the overall IoT system. Therefore, security and integrity of an IoT system are challenged at multiple layers, as data flows through many devices, networks, and administrative boundaries.

Listing 5 Vulnerable IoT Device

```java
public class Server {
  private static Socket socket1;
  private static Socket socket2;
  public static void main(String[] args) {
    int rowNo = 1;
    ...
    while(true) {
      // sensor 1
      InputStream is1 = socket1.getInputStream();
      InputStreamReader isr1 = new InputStreamReader(is1);
      BufferedReader br1 = new BufferedReader(isr1);
      String msg1 = br1.readLine();
      String[] r1 = msg1.split("\t+");
      // sensor 2: same code below as before for sensor
           1, but flowing into r2
      // compute average of the two sensors
      hum = (Float.parseFloat(r[0]) +
           Float.parseFloat(r2[0]))/2;
      tc = (Float.parseFloat(r1[1]) +
           Float.parseFloat(r2[1]))/2;
      tf = (Float.parseFloat(r1[2]) +
           Float.parseFloat(r2[2]))/2;
      // create an object with the measured data
      Put p = new Put(Bytes.toBytes("row"+rowNo));
      p.add(Bytes.toBytes("ambiance"),
           Bytes.toBytes("hum"), Bytes.toBytes(hum));
      p.add(Bytes.toBytes("ambiance"),
           Bytes.toBytes("tempc"), Bytes.toBytes(tc));
      p.add(Bytes.toBytes("ambiance"),
           Bytes.toBytes("tempf"), Bytes.toBytes(tf));
      p.add(Bytes.toBytes("soil"),
           Bytes.toBytes("moisture"),
           Bytes.toBytes(r1[3]));
      // insert data into the database
      table.put(p);
      rowNo++;
    }
    ...
  }
}
```

Examples Consider a scenario where sensor data is sent to a database by the embedded software in the device, and then read by a Web app through a servlet. The code snippet of the IoT device in Listing 5 shows a possible leakage of data between the reading of sensor data (line 12) and the its storage to the database (line 26).

Listing 6 Vulnerable IoT Device

```
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 }
```

Listing 6 reports the implementation of the servlet of this IoT system that retrieves the data and sends it to a companion Android application. The sensitive data is retrieved from the database (line 18) and, after computing the average (line 16–18), it is transmitted through the servlet response (line 8). Therefore, the sensor data (or better, their average) is leaked by the servlet.

Listing 7 The `MainActivity` of the Android application.

```
1  public class MainActivity extends AppCompatActivity {
2    TextView textView;
3    RadioGroup radioGroup;
4    RadioButton radioButton;
5    public void onClick(View view) {
6      String sensor="";
7      String type = "mul";
8      BackgroundWorker backgroundWorker = new
            BackgroundWorker(this);
9      String result = null;
10     int sensType = radioGroup.getCheckedRadioButtonId();
11     if(sensType!=-1) {
12       radioButton = (RadioButton) findViewById(sensType);
13       String str = (String) radioButton.getText();
14       if(str.equals("Soil Moisture")){sensor =
              "soil.moisture";}
15       else if(str.equals("Temerature in C")){sensor =
              "ambiance.tempc";}
16       else if(str.equals("Temerature in F")){sensor =
              "ambiance.tempf";}
17       else if(str.equals("Humidity")){sensor =
              "ambiance.hum";}
18       ... // Wait and retrieve the data from
              BackgroundWorker
19       textView.setText("Average "+str+": "+result);
20     }
21   }
22 }
```

Listing 8 Code snippet of the Android `BackgroundWorker`

```
1  public class BackgroundWorker extends AsyncTask<String,
        Void, String> {
2    protected String doInBackground(String... params) {
3      String type = params[0];
4      String temp = params[1];
5      String servletURL = "...";
6      if(type.equals("mul")) {
7        String result, line;
8        URL url = new URL(servletURL);
9        HttpURLConnection httpURLConnection =
              url.openConnection();
10       ...
11       BufferedWriter bufferedWriter = new
              BufferedWriter(new
              OutputStreamWriter(outputStream, "UTF-8"));
12       String post_data = URLEncoder.encode("sensType",
              "UTF-8")+"="+URLEncoder.encode(temp,"UTF-8");
13       bufferedWriter.write(post_data);
14       ...
15       InputStream inputStream =
              httpURLConnection.getInputStream();
16       BufferedReader bufferedReader = new
              BufferedReader(new
              InputStreamReader(inputStream,"UTF-8"));
17       while((line = bufferedReader.readLine())!=null)
18         result += line;
19       bufferedReader.close();
20       inputStream.close();
21       httpURLConnection.disconnect();
22       return result;
23     }
24     return null;
25   }
26 }
```

Again, when we consider the companion Android application then it is clear that the tainted data received from the servlet is displayed to the user. The code snippet in Listing 7 and 8 reports the code of the Android application. The background worker retrieves (line 17 of Listing 8) and returns the data (line 22 of Listing 8) exposed by the servlet, while the main activity displays such data (line 19 of Listing 7) to the user. Therefore, if the device, cloud storage or companion application communicate correctly, they may propagate through the entire network the sensitive data and leak it.

Static Analysis As discussed in Sect. 3, categories A1, A2, and A7 are already covered by static analysis. In particular, Julia's Injection checker covers A1 and A7, while A2 is partially covered by Passwords and Cryptography checkers. However, the interaction between different layers in an IoT system poses new challenges to static analysis: existing techniques (like taint analysis) could only analyze each layer of the example above in isolation. Therefore, they are not in position to track how user input flows between different layers, and they can cover only partially or with very limited precision injection vulnerabilities in IoT software. Further investigation and solutions [48] are needed in order to address this novel scenario.

I4: Lack of secure update mechanism

Lack of ability to securely update the device which includes lack of firmware validation on device, lack of secure delivery (un-encrypted in transit), lack of anti-rollback mechanisms, and lack of notifications of security changes due to updates.

IoT systems might be exposed to malware and other attack techniques that exploit vulnerable components installed in the devices when the system was deployed. Therefore, it is important to update the software on a regular basis to prevent such attacks. A commonly used technique for this purpose is public key infrastructure (PKI) based system, which is capable of providing required security for update mechanisms. However, the majority of IoT devices lack the computation power to execute PKI efficiently. Again, if the key is not secure and can be extracted from the device, it is encrypted with a single symmetric key for all the device instances, or the encryption keys transferred along with the update for the device firmware, then the IoT system is at risk of hijacking the update process. Therefore, the lack of secure update mechanism refers as well to how the firmware is managed, and not specifically vulnerable components in the firmware.
Static analysis This type of vulnerability can be detected only after the IoT system is deployed, as it involves the overall set up of the different devices, and it cannot be detected by analyzing the software of the various IoT layers.

I5: Use of insecure or outdated components

Use of deprecated or insecure software components/libraries that could allow the device to be compromised. This includes insecure customization of operating system platforms, and the use of third-party software or hardware components from a compromised supply chain.

Studies [70] showed that 80% of the code in today's applications are using libraries and frameworks, but the vulnerabilities associated with these components has been largely undermined. An outdated vulnerable library may allow an intruder to exploit the full privilege of the application, which may include accessing sensitive data, executing transactions, etc. In this regard, the National Vulnerability Database [54] lists majority of the outdated vulnerable libraries. Therefore, the IoT applications shouldn't use third-party libraries which contain well-known vulnerabilities published in National Vulnerability Database.
Static analysis This type of vulnerability is equivalent to A9 (Using Components with Known Vulnerabilities) of the OWASP Top 10 list, and it can be easily detected statically using tools such as OWASP Dependency Checker as discussed in Sect. 3.2.

I6: Insufficient privacy protection

User's personal information stored on the device or in the ecosystem that is used insecurely, improperly, or without permission.

During the last few years, identity thefts and leakages of sensitive data are on the rise with increasing number of devices exposed to the Internet. Assessing the type and protecting sensitive data in various IoT layers is nowadays a critical topic. For instance, should the apps installed in the infotainment system communicate or store sensitive information like location, speed, etc. of the vehicle all the time? Thus, from the privacy perspective one should look for unnecessary communication and storing of sensitive personal identifiable information, encryption of all such data, and anonymization whenever feasible to protect the privacy of the user. This category is similar to A3 (Sensitive Data Exposure) of the OWASP Top 10 list.
Examples To explain the scenario of privacy breaking IoT applications, we considered the Rain Monitor app.[5] It uses OpenXC [55] to collect sensitive data (such as location, windshield status, and speed) of a car, and transmit it to a remote Web service, where it is collected and used to inform drivers of possible showers in their area. Clearly, this app contains a leakage of sensitive data to Internet, which can be seen as a privacy issue.

Listing 9 A code snippet from the Rain Monitor app. Sensitive car data is sent to the Internet and logged.

```
public class CheckWipersTask ... {
  private final String WUNDERGROUND_URL =
    "http://www.wunderground.com/...";
  private VehicleManager mVehicle;
  ...
  public void run() {
    // get messages from the CAN by means of the OpenXC
          API library
    Latitude latitude = (Latitude)
          mVehicle.get(Latitude.class);
    Longitude longitude = (Longitude)
          mVehicle.get(Longitude.class);
    WindshieldWiperStatus wiperStatus =
          (WindshieldWiperStatus)
          mVehicle.get(WindshieldWiperStatus.class);
    ...
    boolean wiperStatusValue =
          wiperStatus.getValue().booleanValue();
    ...
    uploadWiperStatus(latitude, longitude, wiperStatus);
```

[5] https://github.com/openxc/rain.

```
15    }
16
17    private void uploadWiperStatus
18      (Latitude latitude, Longitude longitude,
            WindshieldWiperStatus wiperStatus) {
19      int wiperSpeed = 0;
20      boolean wiperStatusValue =
            wiperStatus.getValue().booleanValue();
21      if (wiperStatusValue)
22        wiperSpeed = 1;
23      String finalUri = WUNDERGROUND_URL + "?wiperspd=" +
            wiperSpeed + "&lat=" + latitude + "&lon=" +
            longitude;
24      ...
25      HttpClient client = new DefaultHttpClient();
26      // send the CAN data on the Internet and receive an
            ack back
27      HttpGet request = new HttpGet(finalUri);
28      HttpResponse response = client.execute(request); //
            line 111
29      int statusCode =
            response.getStatusLine().getStatusCode();
30      if (statusCode != HttpStatus.SC_OK)
31        Log.w(TAG, "Error " + statusCode + // line 114
32          " while uploading wiper status");
33        else
34          Log.d(TAG, "Wiper status (" + wiperStatus + ")
                uploaded successfully"); // line 117
35    }
36  }
```

The code snippet in Listing 9 reports the snippet of the OpenXC application that reads the car location and windshield data, and sends it to the Internet, without encryption nor authentication. The status of the HTTP request and of the windshield is also logged. These are instances of injections: flow of sensitive data into dangerous operations. In this case, the operations divulge sensitive information, violating privacy.

Listing 10 Another code snippet from the Rain Monitor app. Sensitive car data is logged and used to build a URL address.

```
1   public class FetchAlertsTask extends TimerTask {
2     private final String TAG = "FetchAlertsTask";
3     private final String API_URL =
          "http://api.wunderground.com/...";
4     ...
5     public void run() {
6       Latitude latitude = (Latitude)
            mVehicle.get(Latitude.class);
7       Longitude longitude = (Longitude)
            mVehicle.get(Longitude.class);
8       double latitudeValue =
            latitude.getValue().doubleValue();
9       double longitudeValue =
            longitude.getValue().doubleValue();
10      ...
11      Log.d(TAG, "Querying for alerts near " +
            latitudeValue + ", " + longitudeValue); // line
            68
12      ...
13      StringBuilder urlBuilder = new StringBuilder(API_URL);
14      urlBuilder.append(latitudeValue + "," +
            longitudeValue + ".json");
15      URL wunderground = new URL(urlBuilder.toString()); //
            line 76
16      ...
17    }
18  }
```

The code snippet in Listing 10 reports another fragment of the source code from the same app. In this case, the application reads the car position from the CAN and logs it. Hence, anybody having access to the logs can reconstruct the movements of the vehicle, a clear privacy issue. At the end, this code builds a URL by using latitude and longitude. This is a URL injection (sensitive data flowing into an Internet address), possibly inherent to the task performed by this app.

Static analysis As discussed in Sect. 3.2, taint analysis can be applied to detect leakages of sensitive data. Such approach has been implemented in Julia's GDPR Checker. Therefore, this category can be covered by static analysis.

I7: Insecure data transfer and storage

> Lack of encryption or access control of sensitive data anywhere within the ecosystem, including at rest, in transit, or during processing.

Consider the scenario discussed in *I6: Insufficient Privacy Protection* where an application sends the location of a car to the cloud, and also logs some sensitive information coming from the CAN in the local log file as depicted in the code snippets of Listing 9 (line 31–32) and Listing 10 (line 15). With regard to this category, the problem is that the sensitive data is passed and stored in clear text. Therefore, anyone with access the network or the log can intercept and understand it. To protect this sensitive data, messages must be properly encrypted (e.g., with keys that are not directly accessible). These keys are usually stored in a keystore and protected with a password.

Listing 11 Malicious usage of a keystore.

```
1   KeyStore ks = KeyStore.getInstance("JKS");
2   char[] password = getPassword();
3   try (FileInputStream fis = new
          FileInputStream("keyStoreName")) {
4     ks.load(fis, password);
5   }
6   // get private key
7   KeyStore.ProtectionParameter protParam = new
          KeyStore.PasswordProtection(password);
8   KeyStore.PrivateKeyEntry pkEntry =
          (KeyStore.PrivateKeyEntry)
          ks.getEntry("privateKeyAlias", protParam);
9   PrivateKey myPrivateKey = pkEntry.getPrivateKey();
10  // save secret key
11  javax.crypto.SecretKey mySecretKey = ...;
12  KeyStore.SecretKeyEntry skEntry = new
          KeyStore.SecretKeyEntry(mySecretKey);
13  ks.setEntry("secretKeyAlias", skEntry, protParam);
14  // store in the keystore
15  try (FileOutputStream fos = new
          FileOutputStream("newKeyStoreName")) {
16    ks.store(fos, password);
17  }
```

Example The code snippet in Listing 11 uses such a keystore. If the parameter passed to the constructor of `PasswordProtection()` (line 4) and `load()` (line 7) contain input under user's control, then the security of the system could be compromised by an attacker. Moreover, many companion applications rely on weak cryptographic algorithms such as *SHA1PRNG* in *SecureRandom*. A common but potentially harmful use of this algorithm is the creation of encryption keys by using a password as a seed [31]. Due to some implementation issues, the key could become deterministic if the seed is generated with an unsafe algorithm. These issues are prominent in many IoT devices devices and the companion applications.

Static analysis Like category A2 of the OWASP Top 10 list, also this category can be partially covered. In particular, the use of unsafe cryptographic algorithms can be detected by simple analyses that checks if some APIs are called with some specific values. Such vulnerabilities are already detected by standard tools, like Julia's Cryptography checker.

Instead, taint analysis can be applied to detect if user input flows into cryptographic keys that are storing in a keystore. However, to the best of our knowledge, state-of-the-art industrial analyzers do not cover yet this scenario.

Finally, the lack of access control to sensitive data can be only discovered when the system is deployed (e.g., through some forms of dynamic analysis). Therefore, this category can be covered only partially by static analysis means.

I8: Lack of device management

> Lack of security support on devices deployed in production, including asset management, update management, secure decommissioning, systems monitoring, and response capabilities.

This may lead to unauthorized access to the device or the data. Due to poor configurations, devices may have debug ports open for interaction with the system. An intruder may communicate through these pin-outs to interact with the entire system. The level of vulnerable interaction and privilege exploitation is dependent on the type of communication protocol. In the configuration file there may be pin-outs for UART interface which enable intruders to access command shell, logger output, etc. Again, based on the device configuration an intruder may also get access to low-level interaction with the microcontroller using protocols such as JTAG and SWD, which can be used to read/write the internal flash, read/write register values, debug the OS/base firmware code.

Static analysis This category requires to analyze a whole deployed system (comprising firmware and runtime settings), and it goes far beyond the IoT software involving some forms of runtime monitoring. Therefore, it cannot be detected by static analysis.

I9: Insecure default settings

> Devices or systems shipped with insecure default settings or lack the ability to make the system more secure by restricting operators from modifying configurations.

Malware like Mirai scans IoT devices trying to get control by using the default username and password (e.g., admin/admin). If a malware is able to get root access to the device, it may exploit it for coordinating botnet attacks. Therefore, when configuring a device, it is critical to enforce the administrator to follow strict security regulations.

Static analysis In general, these default settings are stored in some properties file or databases, and thus, this category cannot be detected using the static analysis.

I10: Lack of physical hardening

> Lack of physical hardening measures, allowing potential attackers to gain sensitive information that can help in a future remote attack or take local control of the device.

One important aspect of IoT devices is that they are used regularly by multiple users over time. On top of the device usage, there is also the aspect of how a device is accessible, and what level of device access is really needed. Physical security weaknesses are present for instance when an attacker can disassemble a device to easily access the storage medium and any data stored on that. Weaknesses are also present when USB ports or other external ports can be used to access the device using features intended for configuration or maintenance. This could lead to unauthorized access to the device. An attacker could then steal confidential data from the device's memory and launch a spoofing attack.

Static analysis Physical presence of the intruders is required to carry out these kinds of attacks, and therefore, they can be prevented only by physical surveillance and access control of the devices. Thus, static analysis cannot help to detect such category of vulnerabilities.

4.1 Summary

The in-depth exploration of OWASP IoT Top 10 categories suggests that IoT security vulnerabilities can be broadly classified into three categories: software, system, and device hardware. Software vulnerabilities refer to security issues associated with the applications running on the IoT system at different layers. Instead, system vulnerabilities refer to the security issues related to the firmware or operating systems of the devices, as well as to the configuration of the deployed system. Finally, device hardware vulnerabilities are associated with the hardware components and the physical environment they are operating in.

Again, static analysis detects the program vulnerabilities without executing the code. Therefore, this approach is suitable for detecting application vulnerabilities, such as reading sensors data and sending it to the cloud or storing it locally. Instead, discovering vulnerabilities on the overall system

Table 2 OWASP IoT Top 10 2018

ID	IoT vulnerabilities	Application security	Category	Coverage	Julia
I1	Weak guessable, or hardcoded passwords	A2	System, software	Partial	Passwords
I2	Insecure network services		System, software	Partial	
I3	Insecure ecosystem interfaces	A1, A2, A7	Software	Partial	Injection, cryptography, passwords
I4	Lack of secure update mechanism		System		
I5	Use of insecure or outdated components	A9	Software	Full	Vulnerable-components
I6	Insufficient privacy protection	A3	Software	Full	GDPR
I7	Insecure data transfer and storage		System, software	Partial	Cryptography
I8	Lack of device management		Device, hardware		
I9	Insecure default settings		Device, hardware		
I10	Lack of physical hardening		Device, hardware		

involves often firmware and RTOS, and thus, it cannot be pursued by static analysis as it usually demands execution of the program to monitor runtime behaviors and device configuration (e.g., adjusting duty cycle, sending signal to the microcontroller's pins, communication protocols, managing credentials from the configuration files, etc.). Finally, the vulnerabilities associated with the device hardware and physical operational environment cannot be detected using static analysis as they have nothing to do with the application software. Table 2 summarizes the categories of the OWASP IoT top 10 2018 vulnerabilities, and their coverage using static analysis.
Discussion Six out of the top seven vulnerabilities of OWASP IoT Top 10 can be addressed by static analysis. Existing industrial solutions were developed mainly to address OWASP Top 10 vulnerabilities in Web applications, and in fact the four IoT categories that have a corresponding category in OWASP Top 10 are already covered by Julia. While the coverage of IoT categories might be less or more pervasive (e.g., for I1 this approach can detect only very specific cases), in two cases (I5 and I6) static analysis could provide deep coverage. Therefore, next section will briefly introduce the extension of Julia's analyses we developed, while Sect. 6 will present the experimental results of these analyses when applied to some IoT applications publicly available in GitHub.

5 Extending static analyzers to IoT systems

In this section, we describe how we extended Julia's taint analysis engine [7,22,65] in order to detect IoT security vulnerabilities such as leakages of sensitive data and interface interaction issues.

5.1 IoT privacy checker

An IoT device consists of multiple sensors, which provide their own APIs to access (potentially sensitive) sensor data. Thus, it is very difficult to provide a single solution applicable for all IoT devices and companion apps. For this purpose, we applied Julia's taint analysis to detect privacy issues related to sensitive sensor data. Figure 1 depicts the working mechanism of the IoT privacy checker, that relies on a dictionary of sources and sinks specific to the APIs of IoT devices under analysis. Here, sources refer to methods retrieving sensitive information about the device, whereas sinks include methods that potentially leak data (e.g., logging, database, or network manipulation). The analyzer tags as tainted (aka, with a Boolean flag set to true) all the value retrieved from some sources, and then it propagates such values throughout the whole program following the program's semantics. Then, the analyzer checks the flag associated to the values passed to the sink. If it is false then flow of tainted data into that sink is not possible; otherwise, it generates a warning, reporting a potential leakage of private data. We instantiated our approach to a specific library used by some of the examples we will discuss in Sect. 6.

5.2 Insecure ecosystem checker

Julia's taint analysis can provide an exhaustive report tracking how tainted data flows through the program up to a sink [25]. Such approach has been instantiated for the GDPR analysis [26,27], and it allows the user to specify sources and sinks through an Excel spreadsheet. This spreadsheet contains all the potential API calls that could retrieve or leak sensitive data. The user can then tag these with the category of sensitive data they retrieve or with leakage points they disclose information to. The GDPR checker then applies Julia's taint analysis engine with the specification provided through the annotated Excel file, and it returns an exhaustive report with all possible data flow graphs representing potential leakages. We extended the GDPR checker to work with multiple programs, for this we added intermediate sources and sinks in the boundaries of the different interfaces with the help of

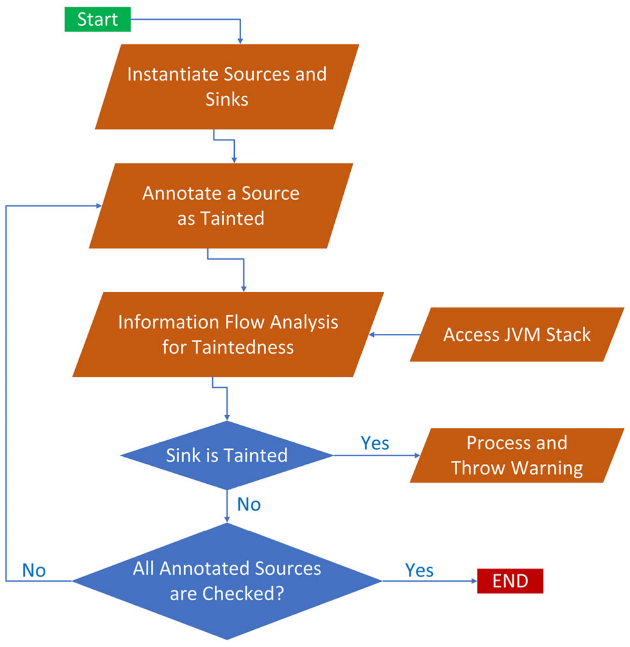
Start
Instantiate Sources and Sinks
Annotate a Source as Tainted
Information Flow Analysis for Taintedness
Access JVM Stack
Sink is Tainted
Yes
Process and Throw Warning
No
No
All Annotated Sources are Checked?
Yes
END

Fig. 1 IoT privacy checker (I6)

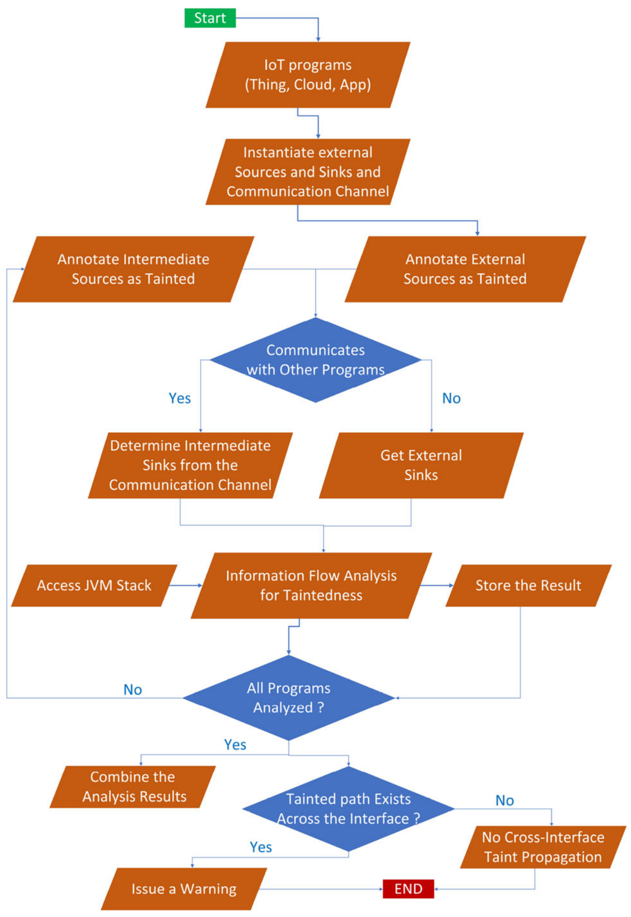
Start
IoT programs (Thing, Cloud, App)
Instantiate external Sources and Sinks and Communication Channel
Annotate Intermediate Sources as Tainted
Annotate External Sources as Tainted
Communicates with Other Programs
Yes
No
Determine Intermediate Sinks from the Communication Channel
Get External Sinks
Access JVM Stack
Information Flow Analysis for Taintedness
Store the Result
No
All Programs Analyzed ?
Yes
Combine the Analysis Results
Tainted path Exists Across the Interface ?
No
No Cross-Interface Taint Propagation
Yes
Issue a Warning
END

Fig. 2 Insecure ecosystem checker (I3)

the communication channel, and tracked the propagation of tainted data across the interfaces.

Figure 2 depicts the working principle. The analyzer first generates the possible set of sources and sinks from the given programs. The users may provide the set of external sources (primary origin) and sinks (final destination) for various types of sensitive data. Usually, we have a program (e.g., embedded software in an IoT device) that access the sensitive data and transmits it through some communication channel. In this case, the analysis considers as sources the ones defined by the user (that is, external sources), while the sinks are determined from the communication channel (termed as intermediate sinks). Then, we usually have another application that reads data from the communication channel and exposes it to other applications. Here, both sources and sinks are the ones specified for some specific communication channels. Finally, we could have a program that read data from the communication channel, and leaks it to some external sinks. In this latter case, the sources are the one defined for communication channels, while the sinks are the ones defined by the user (external sinks). A complete IoT system comprises several applications that are all analyzed by our approach, and then the results are combined into a final report.

6 Experimental results and discussion

The IoT checkers have been implemented on top of the commercial Julia static analyzer [43] for Java and .NET bytecode, based on abstract interpretation [15,16]. 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 [22]) analyses. 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.

6.1 IoT privacy

Listing 12 Vulnerability warnings for the Rain Monitor app.

```
CheckWipersTask.java:111:XSS-injection into method
      "execute"
CheckWipersTask.java:114:Log forging into method "w"
CheckWipersTask.java:117:Log forging into method "d"
FetchAlertsTak.java:68:Log forging into method "d"
FetchAlertsTak.java:76:URL injection into method "<init>"
```

In the Rain Monitor app, we introduced in the description of category I6 of the OWASP IoT Top 10 in Sect. 4, we instructed Julia's GDPR analysis with the sources of sensitive data about the car. Julia's taint analysis then issues the five warnings about potential injections reported in Listing 12. These correspond to the privacy issues informally discussed in the aforementioned section, and reported in Listing 9. The first warning reports that sensitive data about the vehicle flows into method `execute` which creates an HTTP request. Moreover, the status of the HTTP request and the status of the windshield get logged into a file, as shown in the code snippet of Listing 9. The analysis catches these issues in the second and third warning. Instead, in the code snippet of Listing 10, sensitive data (latitude and longitude) is read from the CAN, logged at line 68 (fourth warning), and later concatenated into a URL at line 76 (fifth warning). The latter points to a remote Web service that tracks the position of the car and the weather. Clearly, this is potentially a privacy breach. In conclusion, the privacy checker issues five injection warnings on Rain Monitor and they are all true alarms, although inherent to the main functionality the app performs.

6.2 Insecure ecosystem

To design and demonstrate the capabilities of Insecure Ecosystem checker, we scanned GitHub for repositories containing IoT systems made up of several interacting 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.

In particular, we selected IoT systems communicating through Google Firebase [33] (Doorbell [63] and Electricity Monitor [28]), Near Field Communication (Color Thing [37]), Bluetooth (Bluetooth Low Energy (BLE) fun [51]), and Internet (Robocar [73]). In the rest of this section, we discussed the results of the analysis when applied to these IoT programs.

6.2.1 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 [33]. We have considered two different IoT systems that communicate through Firebase: Doorbell [63] and Electricity Monitor [28].

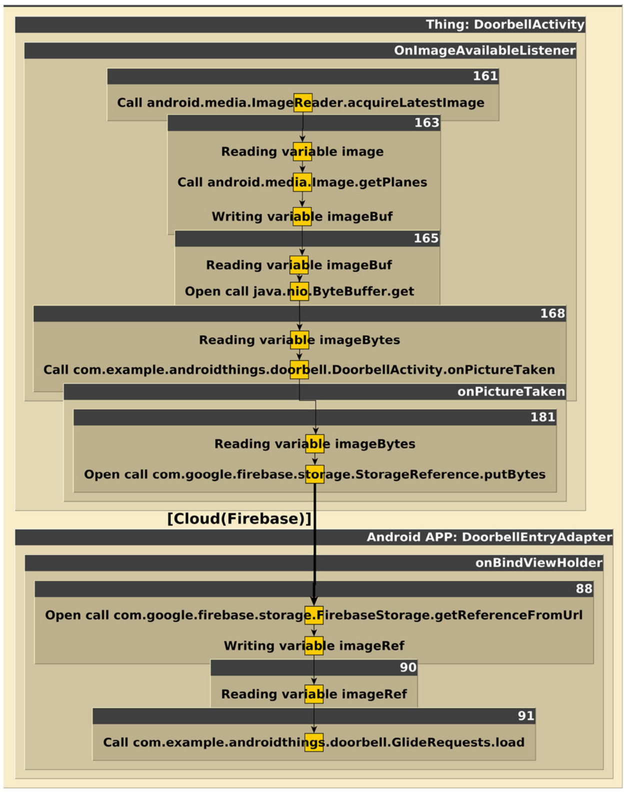
Thing: DoorbellActivity
OnImageAvailableListener
161
Call android.media.ImageReader.acquireLatestImage
163
Reading variable image
Call android.media.Image.getPlanes
Writing variable imageBuf
165
Reading variable imageBuf
Open call java.nio.ByteBuffer.get
168
Reading variable imageBytes
Call com.example.androidthings.doorbell.DoorbellActivity.onPictureTaken
onPictureTaken
181
Reading variable imageBytes
Open call com.google.firebase.storage.StorageReference.putBytes
[Cloud(Firebase)]
Android APP: DoorbellEntryAdapter
onBindViewHolder
88
Open call com.google.firebase.storage.FirebaseStorage.getReferenceFromUrl
Writing variable imageRef
90
Reading variable imageRef
91
Call com.example.androidthings.doorbell.GlideRequests.load

Fig. 3 Doorbell analysis result

Android Things Doorbell [63] 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 `GlideRe- quests.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 3 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, the bold arrow represents a data flow between components. This result shows that the edge software retrieves the image and stores it in Firebase,

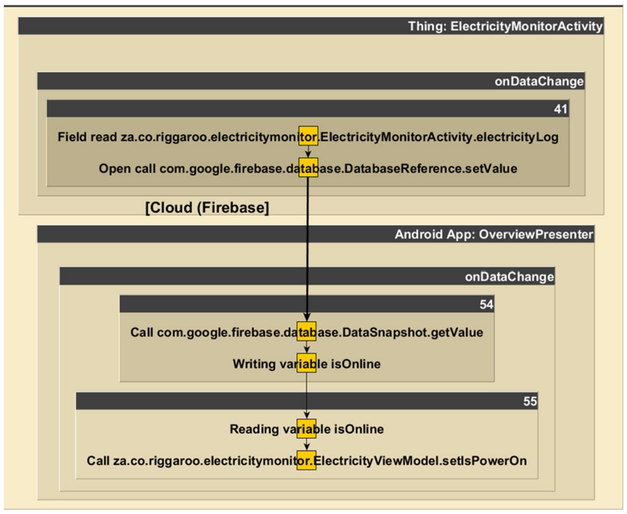
Thing: ElectricityMonitorActivity
onDataChange
41
Field read za.co.riggaroo.electricitymonitor.ElectricityMonitorActivity.electricityLog
Open call com.google.firebase.database.DatabaseReference.setValue
[Cloud (Firebase)]
Android App: OverviewPresenter
onDataChange
54
Call com.google.firebase.database.DataSnapshot.getValue
Writing variable isOnline
55
Reading variable isOnline
Call za.co.riggaroo.electricitymonitor.ElectricityViewModel.setIsPowerOn

Fig. 4 Electricity monitor analysis result

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).

The second IoT system in this category is Electricity Monitor [28], 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 `Database Reference. setValue` (called at line 41 of `Electricity Monitor Activity`), 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 Fig. 4. 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.

6.2.2 Near Field Communication: color thing

Near Field Communication (NFC) allows short-range wireless connectivity. The Color Thing program [37] relies on NFC to allow communication between an edge program and a mobile app, that changes the colors of some LEDs

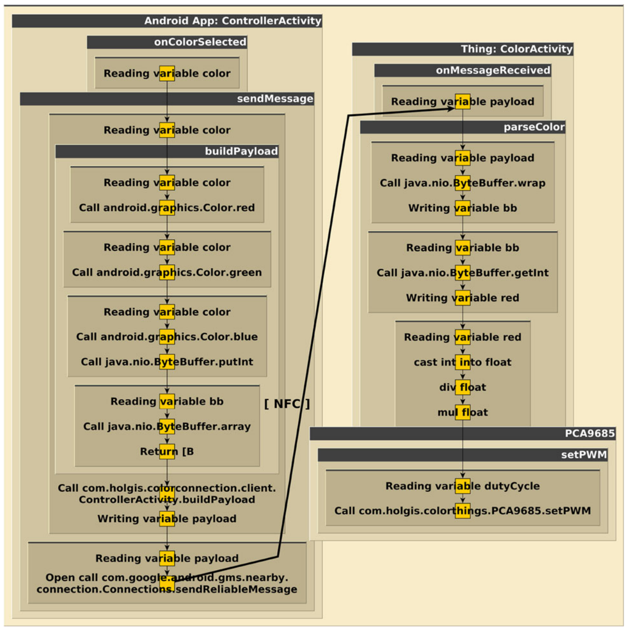
Android App: ControllerActivity
onColorSelected
Reading variable color
sendMessage
Reading variable color
buildPayload
Reading variable color
Call android.graphics.Color.red
Reading variable color
Call android.graphics.Color.green
Reading variable color
Call android.graphics.Color.blue
Call java.nio.ByteBuffer.putInt
Reading variable bb
Call java.nio.ByteBuffer.array
Return [B
[NFC]
Call com.holgis.colorconnection.client.ControllerActivity.buildPayload
Writing variable payload
Reading variable payload
Open call com.google.android.gms.nearby.connection.Connections.sendReliableMessage
Thing: ColorActivity
onMessageReceived
Reading variable payload
parseColor
Reading variable payload
Call java.nio.ByteBuffer.wrap
Writing variable bb
Reading variable bb
Call java.nio.ByteBuffer.getInt
Writing variable red
Reading variable red
cast int into float
div float
mul float
PCA9685
setPWM
Reading variable dutyCycle
Call com.holgis.colorthings.PCA9685.setPWM

Fig. 5 Color thing analysis result

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.se- nd ReliableMessage` (called at line 310 of `Controller Activity`) 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 Fig. 5, 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.

6.2.3 Bluetooth: BLE fun

Bluetooth is another way of communication between nearby devices. The Bluetooth Low Energy (BLE) fun—Android (Things) program [51] 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 `Bluetooth GattCallback. onCharacteristicRead`, accessed at line 83 of `GattClient`, and the parameter of `BluetoothGatt`

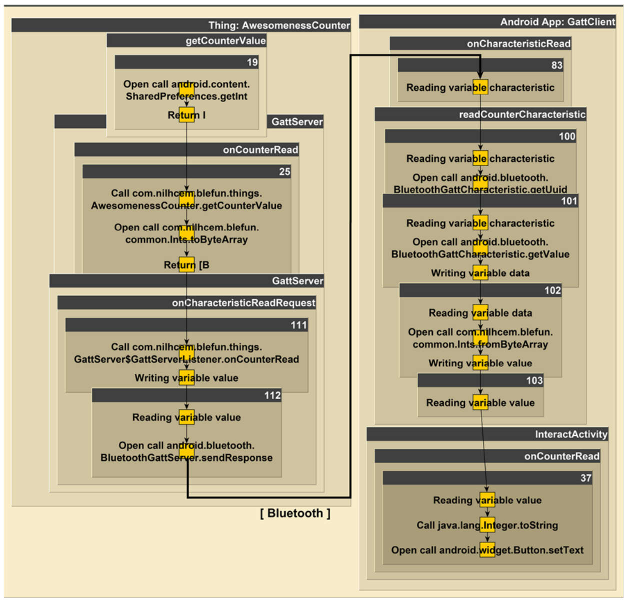
Thing: AwesomenessCounter
getCounterValue
19
Open call android.content.SharedPreferences.getInt
Return I
GattServer
onCounterRead
25
Call com.nilhcem.blefun.things.AwesomenessCounter.getCounterValue
Open call com.nilhcem.blefun.common.Ints.toByteArray
Return [B
GattServer
onCharacteristicReadRequest
111
Call com.nilhcem.blefun.things.GattServer$GattServerListener.onCounterRead
Writing variable value
112
Reading variable value
Open call android.bluetooth.BluetoothGattServer.sendResponse
[Bluetooth]
Android App: GattClient
onCharacteristicRead
83
Reading variable characteristic
readCounterCharacteristic
100
Reading variable characteristic
Open call android.bluetooth.BluetoothGattCharacteristic.getUuid
101
Reading variable characteristic
Open call android.bluetooth.BluetoothGattCharacteristic.getValue
Writing variable data
102
Reading variable data
Open call com.nilhcem.blefun.common.Ints.fromByteArray
Writing variable value
103
Reading variable value
InteractActivity
onCounterRead
37
Reading variable value
Call java.lang.Integer.toString
Open call android.widget.Button.setText

Fig. 6 BLE fun analysis result

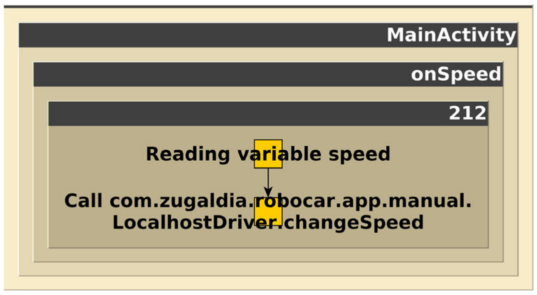
MainActivity
onSpeed
212
Reading variable speed
Call com.zugaldia.robocar.app.manual.LocalhostDriver.changeSpeed

Fig. 7 Robocar analysis result

`Server.sendResponse`, called at line 112 of `Gatt Server`, 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 6 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.

6.2.4 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 [73]. 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. handle- JoystickButtonEvent(View,Motion Event)`, since this activity manages the joystick of the mobile app. The external sink is `LocahostDriver. changeSpeed`, called at line 212 of `MainActivity`. Figure 7 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 `GameControllerActivity.handleJoystick Butt- onEvent`). 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.

7 Conclusion

In this paper, we discussed how static analysis can be adopted to prevent security vulnerabilities in IoT systems. In particular, we started by discussing how this type of tools is currently applied to prevent OWASP Top 10 vulnerabilities in Web applications. We then analyzed OWASP IoT Top 10, and discussed how these vulnerabilities are related to OWASP Top 10 and what existing static analyses can be re-used to detect IoT vulnerabilities. We then introduced two extensions of an existing industrial static analyzer (Julia) and presented some preliminary experimental results.

Overall, six out of the first seven vulnerabilities listed by OWASP IoT Top 10 can be covered at least partially by static analysis means. Five of these types of vulnerabilities are already provided by Julia's checkers, but in some cases its coverage is partial since the IoT scenario introduces novel complexity (in particular, communications between different software layers) that were not present in Web applications. Therefore, we extended these analyses (in particular, about leakages of sensitive data coming from sensors, and insecure ecosystem interfaces passing tainted data between different software layers) to address this scenario. The experimental results show that the proposed extensions are capable of fully detecting several issues related to these categories.

Funding Open access funding provided by Universitá Ca' Foscari Venezia within the CRUI-CARE Agreement.

References

1. Analyzing with sonarqube scanner. https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner. Accessed 05 Nov 2018
2. Akami: Akamai warns of upnp devices used in ddos attacks. https://www.akamai.com/us/en/about/news/press/2014-press/akamai-warns-of-upnp-devices-used-in-ddos-attacks.jsp. Accessed 05 Nov 2018
3. Alizai, Z.A., Tareen, N.F., Jadoon, I.: Improved iot device authentication scheme using device capability and digital signatures. In: 2018 International Conference on Applied and Engineering Mathematics (ICAEM), pp. 1–5 (2018). https://doi.org/10.1109/ICAEM.2018.8536261
4. Assiri, A., Almagwashi, H.: Iot security and privacy issues. In: 2018 1st International Conference on Computer Applications Information Security (ICCAIS), pp. 1–5 (2018). https://doi.org/10.1109/CAIS.2018.8442002
5. Bhawiyuga, A., Data, M., Warda, A.: Architectural design of token based authentication of mqtt protocol in constrained iot device. In: 2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA), pp. 1–4 (2017). https://doi.org/10.1109/TSSA.2017.8272933
6. Burato, E., Ferrara, P., Spoto, F.: Security Analysis of the OWASP Benchmark with Julia. In: Proceedings of ITASEC'17 (2017)
7. Burato, E., Ferrara, P., Spoto, F.: Security analysis of the OWASP Benchmark with Julia. In: Proceedings of ITASEC'17, Venice, Italy (2017)
8. Celik, Z.B., Fernandes, E., Pauley, E., Tan, G., McDaniel, P.: Program analysis of commodity iot applications for security and privacy: challenges and opportunities (2018). arXiv preprint arXiv:1809.06962
9. Challa, S., Wazid, M., Das, A.K., Kumar, N., Goutham Reddy, A., Yoon, E., Yoo, K.: Secure signature-based authenticated key establishment scheme for future iot applications. Access **5**, 3028–3043 (2017)
10. Chze, P.L.R., Leong, K.S.: A secure multi-hop routing for iot communication. In: 2014 IEEE World Forum on Internet of Things (WF-IoT), pp. 428–432 (2014). https://doi.org/10.1109/WF-IoT.2014.6803204
11. COSIC: KU-LEUVEN: Fast, furious and insecure: passive keyless entry and start in modern supercars (2018). https://www.esat.kuleuven.be/cosic/fast-furious-and-insecure-passive-keyless-entry-and-start-in-modern-supercars/. Accessed 05 Nov 2018
12. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: Proceedings of ICFEM'11, Lecture Notes in Computer Science. Springer (2011)
13. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw. Pract. Exp. **45**(2), 245–287 (2015)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
15. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th Symposium on Principles of Programming Languages (POPL). ACM (1977)
16. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of POPL'79. ACM Press (1979)
17. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Proceedings of ESOP '05, LNCS. Springer (2005)
18. Das, A.K., Zeadally, S., He, D.: Taxonomy and analysis of security protocols for internet of things. Future Gener. Comput. Syst. **89**, 110–125 (2018)
19. Dunn, J.E.: Pacemaker controllers still vulnerable 18 months after flaws reported (2018). https://nakedsecurity.sophos.com/2018/08/14/pacemaker-controllers-still-vulnerable-18-months-after-flaws-reported/. Accessed 05 Nov 2018
20. Eclipse IoT Working Group: The three software stacks required for iot architectures (2016)
21. El-Hajj, M., Chamoun, M., Fadlallah, A., Serhrouchni, A.: Analysis of authentication techniques in internet of things (iot). In: 2017 1st Cyber Security in Networking Conference (CSNet), pp. 1–3. IEEE (2017)
22. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in Java. In: Proceedings of LPAR'15, Lecture Notes in Computer Science. Springer (2015)
23. Estay, D.A.S.: Cybership-iot: a dynamic and adaptive SDN-based security policy enforcement framework for ships. Future Gener. Comput. Syst. **100**, 736–750 (2019)
24. Farris, I., Taleb, T., Khettab, Y., Song, J.: A survey on emerging SDN and NFV security mechanisms for iot systems. IEEE Commun. Surv. Tutor. **21**(1), 812–837 (2018)
25. Ferrara, P., Olivieri, L., Spoto, F.: Backflow: backward context-sensitive flow reconstruction of taint analysis results. In: Proceedings of VMCAI'20, LNCS. Springer (2020)
26. Ferrara, P., Spoto, F.: Static analysis for GDPR compliance. In: Proceedings of ITASEC '18 (2018)
27. Ferrara, P., Spoto, F., Olivieri, O.: Tailoring taint analysis to GDPR. In: Proceedings of APF'18 (2018)
28. Franks, R.: Android-things-electricity-monitor. https://github.com/riggaroo/android-things-electricity-monitor. Accessed 05 Nov 2018
29. Frustaci, M., Pace, P., Aloi, G., Fortino, G.: Evaluating critical security issues of the iot world: present and future challenges. Internet Things **5**(4), 2483–2495 (2018)
30. Ge, M., Hong, J.B., Guttmann, W., Kim, D.S.: A framework for automating security analysis of the internet of things. J. Netw. Comput. Appl. **83**, 12–27 (2017)
31. Giro, S.: Android developers blog: Security "crypto" provider deprecated in android n. https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html. Accessed 18 Aug 2018
32. Giuliano, R., Mazzenga, F., Neri, A., Vegni, A.M.: Security access protocols in iot capillary networks. Internet Things **4**(3), 645–657 (2017)

33. Google: Firebase. https://firebase.google.com/. Accessed 05 Nov 2018
34. Grammatech: Codesonar. https://www.grammatech.com/products/codesonar. Accessed 05 Nov 2018
35. Gurunath, R., Agarwal, M., Nandi, A., Samanta, D.: An overview: security issue in iot network. In: 2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), pp. 104–107 (2018). https://doi.org/10.1109/I-SMAC.2018.8653728
36. Hao, P., Wang, X., Shen, W.: A collaborative PHY-aided technique for end-to-end IoT device authentication. IEEE Access **6**, 42279–42293 (2018)
37. Holger: Color-things. https://github.com/holgi-s/ColorThings. https://github.com/holgi-s/ColorConnection. Accessed 05 Nov 2018
38. Hou, J., Qu, L., Shi, W.: A survey on internet of things security from data perspectives. Comput. Netw. **148**, 295–306 (2019)
39. Hung, M.: Leading the iot: Gartner insights on how to lead in a connected world (2017). https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf. Accessed 05 Nov 2018
40. Huuck, R.: Iot: The internet of threats and static program analysis defense. In: EmbeddedWorld 2015: Exibition and Conferences, pp. 493–495 (2015)
41. Invincea Labs: Breaking bhad: Abusing belkin home automation devices (2016). https://www.blackhat.com/docs/eu-16/materials/eu-16-Tenaglia-Breaking-Bhad-Abusing-Belkin-Home-Automation-Devices.pdf
42. Jeon, J.H., Kim, K., Kim, J.: Block chain based data security enhanced iot server platform. In: 2018 International Conference on Information Networking (ICOIN), pp. 941–944 (2018). https://doi.org/10.1109/ICOIN.2018.8343262. Accessed 05 Nov 2018
43. JuliaSoft: Julia static analyzer. https://juliasoft.com/
44. Khattak, H.A., Shah, M.A., Khan, S., Ali, I., Imran, M.: Perception layer security in internet of things. Future Gener. Comput. Syst. **100**, 144–164 (2019)
45. Kim, E., Chung, K., Jeong, T.: Self-certifying id based trustworthy networking system for iot smart service domain. In: 2017 International Conference on Information and Communication Technology Convergence (ICTC), pp. 1299–1301 (2017)
46. Kim, S., Lee, I.: Iot device security based on proxy re-encryption. Ambient Intell. Hum. Comput. **9**(4), 1267–1273 (2018)
47. Mandal, A.K., Cortesi, A., Ferrara, P., Panarotto, F., Spoto, F.: Vulnerability analysis of android auto infotainment apps. In: Proceedings of CF'18. ACM (2018)
48. Mandal, A.K., Ferrara, P., Khlyebnikov, Y., Cortesi, A., Spoto, F.: Cross-program taint analysis for iot systems. In: Proceedings of SAC'20. ACM (2020)
49. Mandal, A.K., Panarotto, F., Cortesi, A., Ferrara, P., Spoto, F.: Static analysis of android auto infotainment and odb-ii apps. Softw. Pract. Exp. **49**(7), 1131–1161 (2019)
50. Mavropoulos, O., Mouratidis, H., Fish, A., Panaousis, E.: Apparatus: a framework for security analysis in internet of things systems. Ad Hoc Netw. **92**, 101743 (2018)
51. Mechling, G.: Bluetooth low-energy (ble) fun—android (things). https://github.com/Nilhcem/blefun-androidthings. Accessed 05 Nov 2018
52. Neshenko, N., Bou-Harb, E., Crichigno, J., Kaddoum, G., Ghani, N.: Demystifying iot security: an exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. Commun. Surv. Tutor. **21**, 2702–2733 (2019)
53. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of NDSS'05. Internet Society (2005)
54. NIST: National vulnerability database. https://nvd.nist.gov/vuln. Accessed 05 Nov 2018
55. OpenXC: The openxc platform. http://openxcplatform.com/. Accessed 05 Nov 2018
56. OWASP: Owasp internet of things (iot) project. https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project. Accessed 05 Nov 2018
57. OWASP: Top 10 Project 2017 (2018). https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed 05 Nov 2018
58. Panarotto, F., Cortesi, A., Ferrara, P., Mandal, A.K., Spoto, F.: Static analysis of android apps interaction with automotive can. In: Proceedings of SmartCom'18, LNCS, vol. 11344, pp. 114–123. Springer (2018)
59. Sahay, R., Geethakumari, G., Mitra, B., Sahoo, I.: Efficient framework for detection of version number attack in internet of things. In: Abraham, A., Cherukuri, A.K., Melin, P., Gandhi N. (eds.) Proceedings of ISDA'18. Springer (2018)
60. Shah, T., Venkatesan, S.: Authentication of iot device and iot server using secure vaults. In: Proceedings of TrustCom/BigDataSE'18, pp. 819–824. IEEE (2018)
61. Shin, D., Sharma, V., Kim, J., Kwon, S., You, I.: Secure and efficient protocol for route optimization in pmipv6-based smart home iot networks. IEEE Access **5**, 11100–11117 (2017)
62. Singh, M., Singh, A., Kim, S.: Blockchain: A game changer for securing iot data. In: 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), pp. 51–55 (2018). https://doi.org/10.1109/WF-IoT.2018.8355182. Accessed 05 Nov 2018
63. Smith, D.: Doorbell. https://github.com/androidthings/doorbell. Accessed 05 Nov 2018
64. Sollins, K.R.: IoT big data security and privacy versus innovation. IEEE Internet Things J. **6**(2), 1628–1635 (2019). https://doi.org/10.1109/JIOT.2019.2898113
65. Spoto, F., Burato, E., Ernst, M.D., Ferrara, P., Lovato, A., Macedonio, D., Spiridon, C.: Static identification of injection attacks in Java. ACM Trans. Program. Lang. Syst. **4**(3), 18:1–518:8 (2019)
66. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: effective taint analysis of web applications. In: Proceedings of PLDI'09. ACM (2009)
67. Tweneboah-Koduah, S., Skouby, K.E., Tadayoni, R.: Cyber security threats to iot applications and service domains. Wirel. Pers. Commun. **95**(1), 169–185 (2017)
68. Urien, P.: An innovative security architecture for low cost low power iot devices based on secure elements: a four quarters security architecture. In: 2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC), pp. 1–2 (2018). https://doi.org/10.1109/CCNC.2018.8319309
69. US Dept. of Homeland Security: Alert (ta16-288a): Heightened ddos threat posed by Mirai and other botnets (2017). https://www.us-cert.gov/ncas/alerts/TA16-288A. Accessed 05 Nov 2018
70. Williams, J., Dabirsiaghi, A.: The Unfortunate Reality of Insecure Libraries. Aspect security. Inc., Columbia (2012)
71. Yoon, S., Kim, J.: Remote security management server for iot devices. In: 2017 International Conference on Information and Communication Technology Convergence (ICTC), pp. 1162–1164 (2017). https://doi.org/10.1109/ICTC.2017.8190885
72. Zaidan, A.A., Zaidan, B.B., Qahtan, M., Albahri, O., Albahri, A., Alaa, M., Jumaah, F.M., Talal, M., Tan, K.L., Shir, W., et al.: A survey on communication components for iot-based technologies in smart homes. Telecommun. Syst. **69**(1), 1–25 (2018)
73. Zugaldia, A.: Android robocar. https://github.com/zugaldia/android-robocar. Accessed 05 Nov 2018