Semantic Static Analysis of IoT Software

Pietro Ferrara
JuliaSoft SRL
Verona, Italy
pietro.ferrara@juliasoft.com

Paul Anderson
GrammaTech, Inc.
Ithaca NY, U.S.A.
paul@grammatech.com

***Abstract*— The IoT paradigm brings together two historically different software worlds. On the one hand, embedded software runs on "things", and it is written in low level programming languages like C and C++. Such software manages locally the device interacting with the physical world, and it might be safety critical. On the other hand, application software running nowadays on the cloud is often implemented in higher level programming languages like Java and C#. It usually manages the data and the business processes of an individual or an organization, and it does not directly interact with the physical world, thus it is not safety critical.**

Rigorous sound semantic static analyses have been widely applied to safety-critical software to detect runtime errors that might compromise the reliability of the system. Compared to non-semantic (e.g., syntactic) analyses, they discover all possible bugs of a given type. For these reasons, many standards like ISO 26262 require the application of semantic static analysis. Similar analyses were available also for application software despite an inferior interest in this world, since bugs and security vulnerabilities impacted only data and processes, and not the physical safety of users. However, the application and cloud software of an IoT system indirectly interacts with the physical world, where a software issue might compromise the safety of the overall system.

In this article, we discuss how static analysis was applied to the embedded and the application software, what new challenges arise from the IoT revolution, and what existing analyses might be reused, adapted and extended in this novel scenario.

Keywords—static analysis; IoT; security

I. INTRODUCTION

In the Internet of Things (IoT) paradigm, many things (aka, devices) are interconnected in a local network and to Internet. In this way, physical devices that are not Internet-enabled can be connected to cloud services, retrieve and store data on the cloud, and allow the user to interact with these devices remotely. IoT became a key enabling technology for many of the new disruptive technologies such as smart manufacturing and smart cities.

IoT brings together two distinct software worlds. On the one hand, embedded software (that is, software running locally on physical devices) has been widely adopted during the last decades to manage locally physical devices. For instance, airplanes and cars often contains several 100KLOCs to provide various functionalities to the pilot or the driver. Such software is often safety critical, this is, its failure could cause deaths, injuries or physical damages. On the other hand, application software is aimed at providing different functionalities to a user on a computer. Such software usually does not interact with physical components (thus it is not safety critical), but it supports various tasks of an individual or an organization.

These two software worlds adopted different programming languages. Embedded software was mostly developed with low level programming languages such as C, C++ and assembly code because of their efficiency at runtime, and easiness to interact with physical components. Instead, since usually application software implements complex functionalities and runs on computationally powerful computers, object programming languages, such as Java and C#, have been widely adopted to develop this software. Other more server- and web-based languages, like JavaScript and PHP, have been adopted more recently in this context as well.

Experience shows that writing bug-free software is impossible, and debugging programs is an extremely difficult task. Therefore, various approaches and tools have been introduced. Static analysis is aimed at detecting 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 the different paths of execution. However, it needs to introduce some forms of approximation in order to represent the execution of a program and prove properties on them.

Safety and security have been key aspects of both embedded and application software. In particular, bugs in embedded software have caused various catastrophic failures. In the field of static analysis, the most famous bug was the one that caused the explosion of the Ariane 5 flight on 4 June 1996 [1]. The explosion happened because of a numerical overflow due to data conversion. Such bug was discovered by PolySpace [9], a static analyzer of embedded software. Many other bugs in safety critical embedded software caused various crashes, and fatalities [2]. In this field, static analysis has been widely applied to detect runtime errors like divisions by zero, and arithmetic overflows. This led to various commercial static analyzers like GrammaTech CodeSonar [3], and ASTREE [4].

On the other hand, during the last decade application (and in particular Web and cloud) software has been the target of various cyber-attacks. SQL injections have been largely exploited to read, modify and delete data from various databases, while cross-site scripting (XSS) vulnerabilities allowed attackers to inject code and redirect users to malicious websites, or intercept confidential information like usernames

and passwords. SQL injections have been largely exploited in data breaches as well: the Heartland Payment Systems data breach, based on an SQL injection, exposed the data of 134 million credit cards to the attackers, and the company had to pay about 145M$ in compensation for fraudulent payments [8]. In this field, various static analysis tools, like Julia [6] and FindBugs [7], have been widely applied to detect injections and XSS vulnerabilities in industrial software. However, attacks exploiting these vulnerabilities have caused huge business damages, but usually no impact on the physical world. In fact, while embedded software directly interacts with the physical world through sensors and actuators, this software provides logical functionalities to support individuals and organizations, and it is not safety critical.

By bringing together these two software worlds, IoT will enable advanced remote physical functionalities through embedded and application software. However, this will expose such systems to various security vulnerabilities. This is one of the most critical factors for a wide adoption of IoT systems [5]:

Security will be a major concern wherever networks are deployed at large scale. There can be many ways the system could be attacked—disabling the network availability; pushing erroneous data into the network; accessing personal information; etc.

But what vulnerabilities might impact IoT software? The runtime errors of embedded software, injection vulnerabilities of cloud software, or something else? And what vulnerabilities static analysis can detect?

In the rest of this paper, we discuss how static analysis has been applied to embedded and application software, what software security vulnerabilities might be particularly relevant in an IoT system, and which ones could be detectable by static analysis.

II. Static Analysis of Embedded Software

A safety critical system is defined as "a system whose failure or malfunction may result in one (or more) of the following outcomes: (i) death or serious injury to people, (ii) loss or severe damage to equipment/property, (iii) environmental harm" [16]. Therefore, software bugs causing the failure of a safety-critical system might have catastrophic impact, and a relevant effort was focused on the debug of safety critical embedded software [17]:

Safety-critical embedded software has to satisfy stringent quality requirements. A system failure or malfunctioning can have severe consequences and cause high costs. Testing and validation consume a large – and growing – fraction of development cost. (…)

An important goal when developing critical software is to prove that no such errors can occur at runtime. Software testing can be used to detect errors, but since usually no complete test coverage can be achieved, it cannot provide guarantees. Semantics-based static analysis allows to derive such guarantees even for large software projects. The success of static analysis is based on the fact that safe overapproximations of program semantics can be computed.

In this context, static analysis became a mandatory part of several certification. For instance, the DO-178C "Software Considerations in Airborne Systems and Equipment Certification" describes in details the purpose of software verification, and how it should be applied during development, reviews and analyses. Similarly, the US Food and Drug Administration suggested the adoption of static analysis to improve the safety of medical software.

Therefore, many commercial analyzers developed several analyses to discover software vulnerabilities that might have catastrophic consequences in safety critical embedded software. These analyses targeted mainly runtime errors like null pointer dereferences, and buffer overrun and underrun.

However, over the time more and more functionalities have been added to embedded software, and this opened the door to cyber-attacks, and therefore commercial static analyzers developed some analyses to detect this kind of software vulnerabilities [18]:

As embedded applications become more feature-rich, the risks of security vulnerabilities are increasing. (…) Until recently, hackers had not been frequently targeting embedded systems because doing so required physical access to the device. However, with the accelerating network connectivity of embedded devices, malicious hackers have a new, virtual attack path. Despite the trend toward greater device connectivity, awareness of the risks of insecure code is still low among embedded developers.

Some of the semantics static analyses in this field are based on taint analysis [19], a standard way to track how user input might flow through the different components of the software and potentially reach sensitive API.

III. Static Analysis of Application Software

Static analysis was widely adopted to debug also application software. However, while in embedded software bugs could have catastrophic consequences, application software's bugs have much less impact, since this software is not safety critical. Therefore, bug hunting in application software has been focused towards finding some (and not all) bugs in order to improve the overall quality of the software product, but not to eradicate any possible bug like what happened for safety critical embedded software.

A recent article [12] gave very deep and interesting insights about the lessons learned at Google when applying static analyzers to their (huge) codebase. Their analysis targeted generic software engineers in Google, and they focus on relatively simple static analyzers (that is, analyzers that are not interprocedural or whole-program). In particular, a warning is considered as an effective false positive "if developers did not take positive action after seeing the issue", despite the fact that the warning effectively points to a real bug or not. Starting

from this developer-oriented view, they noticed that "the presence of effective false positives caused developers to lose confidence in the tool", and then they require that static analyzers "produce less than 10% effective false positives" in order to be integrated in the code-review process. Such approach is radically different from what happens in the safety critical software world, where it is essential to avoid any (potentially catastrophic) bug.

However, during the last few years, Web and cloud application software partially changed this scenario. Security vulnerabilities in this software, like SQL injections and XSS, could potentially cause relevant damages to the enterprise using such software, as mentioned in the Introduction. For these reasons, so-called Static Application Security Testing (SAST) became a common practice in application software producers [11] with the goal of eradicating all security vulnerabilities [10]:

The goal of SAP's Product Standard "Security" is to ensure that SAP products are secure by default as well as easy to operate securely. A prerequisite to achieve this goal is to ensure that SAP products are free of implementation related security vulnerabilities such as SQL injection, path traversal, memory corruption, or buffer overflow

This new scenario led to the development of several static analyzers targeting security vulnerabilities in Web and cloud application software. But what software vulnerabilities have the biggest impact? The OWASP Top 10 project released and updated, starting from 2010, a list of the 10 most dangerous software vulnerabilities. The version released in 2017 is the following one:

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging&Monitoring

Static analyzers have been proven to be effective in identifying some of these vulnerabilities, and in particular Injection, XSS and (more recently) Sensitive Data Exposure. All these properties can be reduced to discovering whether some user input or sensitive data might flow into a sensitive API call (e.g., executing an SQL statement, some code, or sending some data to Internet) without being sanitized (e.g., properly escaped or encrypted) before. Some analyzers focused on efficient yet imprecise local syntactic checks (e.g., use of API potentially injectable or non-constant strings passed as parameter), while others compute a precise but expensive taint analysis [19] of how user input might traverse the software, and if it can reach a sensitive API. Therefore, different static analyzers achieve radically different coverage and precision, and various projects like the OWASP Benchmark [13,14] have been developed to compare the effectiveness of these tools.

IV. Static Analysis of IoT Software

IoT systems interact with physical components, provide remote functionalities through cloud services, and they are usually safety critical. On the one hand, the embedded portion of IoT software directly interacts with physical components that could put the physical safety of a user at risk. These components might be equipped with sensors and actuators. The first ones allow the program to retrieve (potentially sensitive) information about the physical environment (thus posing privacy issue), while the latter actuates the (potentially safety-critical) devices.
On the other hand, another portion of the IoT software implements back-end or cloud services that might indirectly interact with the physical components of the system through the embedded software it communicates with. Therefore, even if this software is not embedded in the device and indeed is usually executed in the cloud or remotely, it should be still considered safety critical, since its failures or vulnerabilities might have catastrophic results in an IoT safety-critical system.

Imagine for instance a smart home where an IoT system allows a user to lock and unlock the doors. The embedded portion of the IoT software will directly interact with the physical lock, while, for instance, a mobile application might rely on some cloud services in order to check the status of the locks, and to lock or unlock the doors remotely. While the mobile application does not interact directly with physical components, its functionalities are still safety critical: a thief might exploit a security vulnerability in the mobile application in order to unlock the door of an apartment.
Such structure is quite common in IoT systems, where an edge device is usually adopted as a centralized gateway to connect and interact with various devices in the physical environment, while the cloud services provide remote control and monitoring of the physical system [15].

For all these reasons, several recent efforts have been focused towards the definition of what software vulnerabilities would be particularly relevant in an IoT system. For instance, the OWASP IoT project released the IoT Top 10 [20]. Similarly, to OWASP Top 10, this standing "represents the top ten things to avoid when building, deploying, or managing IoT systems". The IoT Top 10 2018 list is the following one:

1. Weak Guessable, or Hardcoded Passwords
2. Insecure Network Services
3. Insecure Ecosystem Interfaces
4. Lack of Secure Update Mechanism
5. Use of Insecure or Outdated Components
6. Insufficient Privacy Protection

7. Insecure Data Transfer and Storage
8. Lack of Device Management
9. Insecure Default Settings
10. Lack of Physical Hardening

Static analysis can help to address few of these issues, and in particular the third one (Insecure Ecosystem Interfaces):

Insecure web, backend API, cloud, or mobile interfaces in the ecosystem outside of the device that allows compromise 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.

These vulnerabilities correspond to several OWASP Top 10 categories like Injection, XSS, and Broken Authentication. As discussed in the previous Sections, several commercial analyzers have developed various analyses to detect these issues in application software, and tools for embedded software integrated similar analyses as well. However, IoT systems pose new challenges: the interaction among software components implementing different (e.g., embedded and cloud) layers written usually in different programming languages might expose security vulnerabilities that cannot be caught only when considering the overall IoT system.

Taint analysis has been proved to be an effective approach to detect injection, XSS, and other vulnerabilities. Various commercial static analyzers, like GrammaTech CodeSonar and Julia, apply this approach, and their application to industrial software helped to debug and find security vulnerabilities both in embedded and application software. However, existing static analyses are in position to reason on single software components written all in the same programming language, while typically IoT systems comprehend several different software layers written in different programming languages. How these tools and analyses can be extended to this novel scenario is still an open topic.

REFERENCES

[1] Wikipedia, Cluster (spacecraft) , https://en.wikipedia.org/wiki/Cluster_(spacecraft)

[2] Wikipedia, List of software bugs , https://en.wikipedia.org/wiki/List_of_software_bugs

[3] GrammaTech, CodeSonar, https://www.grammatech.com/products/codesonar

[4] The Astrée Static Analyzer, http://www.astree.ens.fr/

[5] Jayavardhana Gubbia, Rajkumar Buyyab, Slaven Marusica, and Marimuthu Palaniswamia: "Internet of Things (IoT): A vision, architectural elements, and future directions", in Future Generation Computer Systems, Volume 29, Issue 7, September 2013, Pages 1645-1660

[6] JuliaSoft, the Julia static analyzer, http://www.juliasoft.com

[7] FindBugs, http://findbugs.sourceforge.net/

[8] CSO: "Heartland: 'Largest Data Breach Ever'", 20 January 2009, https://www.csoonline.com/article/2123599/malware-cybercrime/heartland---largest-data-breach-ever-.html

[9] Mathworks, Polyspace: https://www.mathworks.com/products/polyspace.html

[10] Achim D. Brucker and Uwe Sodan: "Deploying static application security testing on a large scale", GI Sicherheit 2014

[11] M. Howard and S. Lipner: "The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software", Microsoft Press, 2006.

[12] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, C. Jaspan: "Lessons from Building Static Analysis Tools at Google", Communications of the ACM (CACM), 58-66, volume 61 Issue 4, 2018

[13] OWASP Benchmark, https://www.owasp.org/index.php/Benchmark

[14] Pietro Ferrara, Elisa Burato, Fausto Spoto: "Security Analysis of the OWASP Benchmark with Julia", Proceedings of ITASEC 2017

[15] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel: "Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities", CoRR abs/1809.06962, 2018

[16] Wikipedia, Safety-critical system, https://en.wikipedia.org/wiki/Safety-critical_system

[17] Daniel Kaestner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné and Xavier Rival: "Astrée: Proving the Absence of Runtime Errors", in Proceedings of Embedded Real Time Software and Systems (ERTS 2010)

[18] GrammaTech: "Protecting Against Tainted Data in Embedded Apps with Static Analysis", white paper, https://resources.grammatech.com/whitepapers/protecting-against-tainted-data-in-embedded-apps-with-static-analysis

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

[20] OWASP Internet of Things (IoT) Top 10 2018, https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10