Cross Programming Language Taint Analysis for the IoT Ecosystem

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

Amit Kr Mandal
Università Ca' Foscari, Venezia, Italy
BML Munjal Univesity, Gurgaon, Haryana, India
amitmandal.nitdgp@gmail.com

Agostino Cortesi
Università Ca' Foscari, Venezia, Italy
cortesi@unive.it

Fausto Spoto
Università di Verona, Verona, Italy
fausto.spoto@univr.it

The Internet of Things (IoT) is a key technology for the next disruptive technologies. However, IoT merges together several diverse software layers: embedded, enterprise, and cloud programs interact with each other. In addition, security and privacy vulnerabilities of IoT software might be particularly dangerous due to the pervasiveness and physical nature of these systems. During the last decades, static analysis, and in particular taint analysis, has been widely applied to detect software vulnerabilities. Unfortunately, these analyses assume that software is entirely written in a single programming language, and they are not immediately suitable to detect IoT vulnerabilities where many different software components, written in different programming languages, interact. This paper discusses how to leverage existing static taint analyses to a cross programming language scenario.

1 Introduction

The Internet of Things (IoT) paradigm is considered a key-enabling component of many of the next disruptive technologies: smart health, smart manufacturing, and smart cities are just three of the most notable examples. In an IoT system, many different *things* (*i.e.*, devices with embedded software) coordinate and communicate through the Internet (*e.g.*, a local gateway and/or the cloud). Often, these systems can be accessed and managed remotely through cloud services. Because of the expected pervasiveness of these systems (Gartner estimates that there will be more than 20 billions IoT devices by 2020 [15]), the IoT revolution introduces major challenges for building safe, reliable, and privacy-preserving solutions.

Figure 1 reports a common IoT architecture depicted by the IoT Eclipse Working Group [10]. In particular, "a typical IoT solution is characterized by many **devices** (*i.e.*, things) that may use some form of **gateway** to communicate through a network to an enterprise back-end server that is running an **IoT platform** that helps integrate the IoT information into the existing enterprise". Therefore, three main components are involved in an IoT system, each with its own software: (i) devices (aka things), (ii) gateways, and (iii) cloud platforms. Therefore, the various layers of an IoT system comprise a wide stack of software that might be written in extremely heterogeneous programming languages.

IoT CyberSecurity and Static Analysis: IoT devices have been already the target of several cyber-attacks, and their safety and security is a key aspect to ensure their wide adoption. For instance, the malware Mirai exploited IoT devices (such as IP cameras and home routers) that were not properly configured to create a botnet and launch DDoS attacks on a large scale, such as the Dyn cyberattack on October, 21st 2016[1].

[1] https://en.wikipedia.org/wiki/2016_Dyn_cyberattack

Submitted to:
InterAVT 2019 (ETAPS)

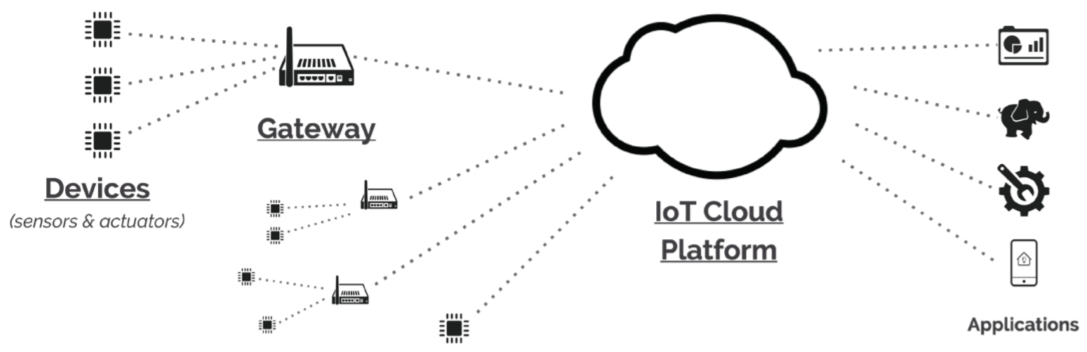


Figure 1: A standard IoT architecture.

How could we then improve the cybersecurity of IoT systems? Static analysis [7] has been widely applied in this field during the last few decades. They build a semantic model of software at compile time, without executing it, and check various properties on that. Nowadays, several standards and regulations (*e.g.*, MISRA DO-178C, IEC 61508, ISO 26262, and IEC 62304) require the application of such tools. However, their application to IoT 5cloud component has been limited up to now. On the other hand, various static analyzers have focused on the detection of various types of security vulnerabilities (such as SQL injection and cross site scripting attacks) on the back-end of web servers [1, 26].

Taint analysis: A standard approach to address these issues has been taint analysis [28], 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). This generic schema has been instantiated to several critical security vulnerabilities in the OWASP Top 10 list [23], such as (i) SQL injection, where sources are methods returning user input, sinks are methods executing SQL queries, and sanitizers are methods escaping the input; (ii) cross-site scripting (XSS), where sinks are methods executing the given data; (iii) redirection attacks, where sinks are instead parameters of methods opening an Internet connection; and (iv) leakages of sensitive data. Taint analysis achieved impressive industrial results, detecting many vulnerabilities in real-world software (e.g., web servers) and achieving amazing results [4], in comparison to other (usually pattern-based) approaches. Such analyses have been widely applied to detect privacy leaks as well [21, 13]. Unfortunately, existing static analyzers focus on a single programming language; if programs written in different languages interact with each other, then the analysis considers each program "in isolation".

1.1 Related Work

Most IoT applications use cross programming language programs to interact with the hardware, that is, to execute native calls. For instance, this is the case of most Java programs that rely on native code for this purpose. Thus, inter-procedural dataflow analysis provides a general framework for program analysis. Matthews et al. [20] provided a formalization of the inter-operation among two high-level functional languages with a shared memory. Therefore, this work mainly focuses on the interaction through this space, and not on taint propagation. Click and Cooper [6] devised a lattice-based representation of conditional constant propagation by defining special flow functions over the composed domain. Pioli and Hind [25] provided a single analysis which combines constant propagation and pointer analysis by using a combined flow function. Whereas, Mandal et al. [18] and Panarotto et al. [24] demonstrated the effectiveness of taint analysis for leakage detection in android automotive apps. However, the majority of these approaches provides insight about propagation of a particular type of data, and do not focus specifically on taint propagation. Recently, few other approaches have been focused towards the security of IoT systems, and what program analysis can address in this space. Huuck [17] discussed the security threats of IoT devices, and advocated the use of static code analysis to detect some of these issues. Similarly, Celik et al. [5] identified security and privacy issues of five IoT platforms, and applied existing static analyzers to detect these issues. Their conclusion is that ”a suite of analysis tools and algorithms targeted at diverse IoT platforms is at this time largely absent”.

2 Illustrative Example

We now introduce an illustrative example to explain how different software components might interact in an IoT system. The software we introduce in this section manages a robotic car. In particular, it allows the user to control the car through a joystick. In this section, we report a simplified snippet of code of the application; the interested reader can find the full implementation at `https://github.com/amitmandalnitdgp/IOTJoyCar`. This software has been installed on a Raspberry Pi 3B+, with 64-bit, Quad-Core, Broadcom BCM2837B0 CPU running at 1.4GHz and 1GB of LPDDR2 SDRAM. For steering control, we relied on a TowerPro SG90 micro servo, and for accelerating the car on a 130 DC 1V - 6V Micro Motor. Finally, to control the system we adopted an Analog B103 joystick. In the implementation, we used PCF8591 and L293D as device driver for the analog motor and the joystick.

Figure 2 reports the Java code implementing the front-end of the IoT system. This program, running on the gateway, interacts with the joystick to read its data, and with the car in order to move it accordingly to the input received by the user through the joystick. In particular, the `main` method indefinitely loops (line 9) while reading the input from the joystick and moving the car accordingly (line 10). However, in order to physically interact with the components (that is, to run the motor and turn the car), it relies on two native methods implemented in C++ (lines 2–3).

```
 1  class JoyCar {
 2    public native int readUpDown();
 3    public native void runMotor(int value);
 4
 5    public static void main(String[] args) {
 6      JoyCar rc = new JoyCar();
 7      // Initialization
 8      ...
 9      while(true){
10        rc.runMotor(rc.readUpDown());
11        //Turn based on joystick input
12        ...
13      }
14    }
15  }
```

Figure 2: Java code.

```
 1  JNIEXPORT jint JNICALL Java_JoyCar_readUpDown
 2                              (JNIEnv *env, jobject o){
 3  % return readAnalog(A1);
 4  }
 5  long map(long val,long fl,long fh,long tl,long th){
 6    return (th-tl)*(val-fl) / (fh-fl) + tl;
 7  }
 8  void motor(int ADC){
 9    int value = ADC -130;
10    // initialize the direction
11    softPwmWrite(enablePin,map(abs(value),0,130,0,255));
12  }
13  JNIEXPORT void JNICALL Java_JoyCar_runMotor
14                         (JNIEnv *env, jobject o, jint val){
15    motor(val);}
```

Figure 3: Embedded C++ code.

Figure 3 reports the code of these native methods. This code relies on WiringPi[2], a PIN-based GPIO

[2]`http://wiringpi.com/`

access library written in C for the Raspberry Pi. This library allows one to send instructions to the car-components through GPIO pins, and facilitates I2C communication required for many devices through various extensions[3]. In our example, the joystick uses PCF8591 to read the analog value by calling the `analogRead(A1)` function of the *pcf8591.h*[4] I2C library at line 3; in this way the program reads the analogical Y-axis value of the joystick. WiringPi also allow one to send pulse width modulation (PWM) signals to the devices through `softPwmWrite(pin,ms)`; for instance, line 14 sends a signal to the motor.

Therefore, Figure 3 reports the implementation of the two native methods of the class `JoyCar` in Figure 2. Method `Java_JoyCar_readUpDown` simply returns the Y-axis value of the joystick (line 3). Instead, method `Java_JoyCar_runMotor` sends a value to the motor that is the result of some arithmetical computation on the value passed by the Java program.

Potential Attack: What kind of attack might happen in the IoT software we just introduced? Imagine that an attacker can inject through this code an unbounded numerical value to the motor: this might cause the car to break, or the engine to overheat and potentially get on fire. So the question is: is our code vulnerable to such an attack? In order to give an answer to this question, we need to consider both the Java and the C++ code. Intuitively, we want to detect if a user input (that is, a source) might flow into the motor (that is, a sink) without being sanitized (that is, properly cleaned). In the example above, the user input is retrieved at line 3 of the C++ code in Figure 3. Therefore, the call to `readUpDown()` at line 10 of the Java code in Figure 2 retrieves a tainted value and passes it to `runMotor`. Then, the `motor(val)` call at line 19 of the C++ code passes a tainted value, that is then processed and passed to `softPwmWrite`. This method sends a value to the motor, that is, `softPwmWrite` is a sink.

3 Cross Programming Language Taint Analysis

The main contribution of our work is providing evidence of the effectiveness of a Cross Programming Language Taint Analysis approach to security vulnerability detection in IoT systems. We have combined two existing commercial analyzers to analyze the example introduced in Section 2.

Julia [26] is a commercial static analyzer for Java and .NET (i.e., CIL [11]) bytecode, based on abstract interpretation [7]. Julia currently features 45 *checkers*, including the Injection checker based on the sound taint analysis defined in [12]. Julia's taint analysis has been widely applied to detect security vulnerabilities such as SQL injections and XSS [4] as well as to the detection of leakages of sensitive data [13, 14]. The Injection checker can be instrumented with additional sources and sinks by adding specific Java annotations to the analyzed code[5].

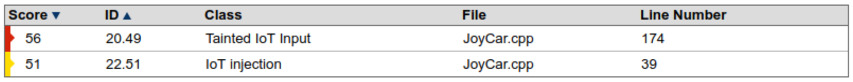
CodeSonar is an advanced, whole program static analyzer developed by GrammaTech for C, C++ and binary programs. Its analyses comprise a taint analysis engine [2] that performs an automatic taint analysis propagation. CodeSonar also allows the user to inspect the complete flow of the tainted data inside the program in order to guide him towards the problem detected by the analysis. In addition, the taint analysis engine can be instrumented with additional sources and sinks.

Results on the Illustrative Example: Taint analysis receives as input a set of sources (*e.g.*, methods returning user input), sinks (*e.g.*, safety critical methods), and sanitizers (*e.g.*, methods *cleaning* the user input). Therefore, we start by defining a set of sources and sinks for the Java and C++ programs of our Illustrative Example. In particular, the C++ code contains one source (the value returned by

[3]`http://wiringpi.com/extensions/`

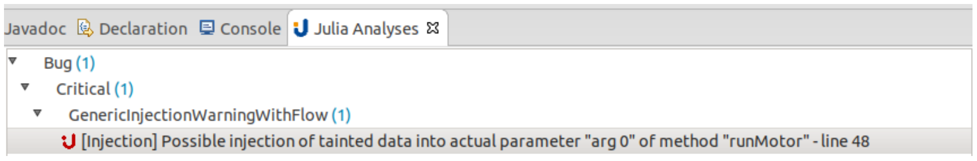
[4]`http://wiringpi.com/extensions/i2c-pcf8591/`

[5]`https://static.juliasoft.com/docs/latest/Injection.html` contains the detailed description of the Injection checker and the annotations it understands.



Score ▼	ID ▲	Class	File	Line Number
56	20.49	Tainted IoT Input	JoyCar.cpp	174
51	22.51	IoT injection	JoyCar.cpp	39

Figure 4: CodeSonar warnings.



Bug (1)
Critical (1)
GenericInjectionWarningWithFlow (1)
[Injection] Possible injection of tainted data into actual parameter "arg 0" of method "runMotor" - line 48

Figure 5: Julia warnings.

`readAnalog`) and one sink (the second parameter of method `softPwmWrite`), while the Java code does not contain any source nor sink. In addition, we need to further instrument the taint analysis in order to detect if tainted values are passed from one program to the other. Starting from the Java Native Interface specification, we tag as a sink (i) the value returned by `Java_JoyCar_readUpDown` in the C++ program, and (ii) the first paramenter of `JoyCar.runMotor` in the Java program. In this way, we will detect if a tainted value flows between the boundaries of these two applications, and we will further instrument and iterate the taint analysis to propagate cross programming language tainted values.

We instrumented Julia and CodeSonar taint analyses with these sources and sinks through their standard instrumentation mechanism (Java annotations for Julia, and specific method calls and stubs in CodeSonar). `https://github.com/amitmandalnitdgp/IOTJoyCar` contains the instrumented code. We then run the taint analysis with these sources and sinks on both the programs. Since we do not have any source in the Java software, no tainted flow is propagated there. Instead, the CodeSonar taint analysis on the C++ program produces the first warning in Figure 4 reporting that method `Java_-JoyCar_readUpDown` could return tainted data. Therefore, we taint the value returned by `JoyCar.read-UpDown` as a source in the Java program to propagate it.

We run again the taint analysis on the C++ and Java programs. In the first case, we obtain the same result of the previous run since no source or sink was added. Instead, Julia taint analysis gets augmented with the new source, and it produces a warning at line 10 of the Java program as reported in Figure 5, since a tainted value flows into the first parameter of method `JoyCar.runMotor`. We add this parameter in the C++ program as a source to propagate the warning reported in Figure 5. The C++ taint analysis produces an additional warning reporting that a tainted value might flow into `softPwmWrite`. Figure 4 reports this as the second warning. This warning detects exactly the potential attack we discussed at the end of Section 2. However, one might assume that the `map` method sanitizes the value passed to the motor, and thus this is not a real problem. Therefore, we further instrumented our C++ program annotating the `map` function at lines 6–8 of Figure 3 as a sanitizer. Then CodeSonar at the end produces only the first warning in Figure 4 telling that `Java_JoyCar_readUpDown` returns a tainted value.

4 Conclusion

This paper shows how existing taint analyses can be leveraged to detect IoT software vulnerabilities where software components written in different programming languages interact. It shows the feasibility of the extension of existing commercial taint analyses to a rather simple IoT case study, but its automation and scalability to IoT industrial programs is still an open question, left as future work.

References

[1] *Analyzing with SonarQube Scanner.* `https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner`.

[2] *Protecting against Tainted data in embedded apps with static analysis.*

[3] R. Bryant (1992): *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams.* ACM Computing Survey 24(3), pp. 293–318.

[4] E. Burato, P. Ferrara & F. Spoto (2017): *Security Analysis of the OWASP Benchmark with Julia.* In: Proceedings of ITASEC '17.

[5] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan & Patrick McDaniel (2018): *Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities.* arXiv preprint arXiv:1809.06962.

[6] Cliff Click & Keith D Cooper (1995): *Combining analyses, combining optimizations.* ACM Transactions on Programming Languages and Systems (TOPLAS) 17(2), pp. 181–196.

[7] P. Cousot & R. Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* In: POPL, pp. 238–252.

[8] Patrick Cousot & Radhia Cousot (1979): *Systematic design of program analysis frameworks.* In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, pp. 269–282.

[9] Dorothy E. Denning & Peter J. Denning (1977): *Certification of Programs for Secure Information Flow.* Commun. ACM 20(7).

[10] Eclipse IoT Working Group (2016): *The Three Software Stacks Required for IoT Architectures.*

[11] ECMA (2012): *Standard ECMA-335: Common Language Infrastructure (CLI).*

[12] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon & F. Spoto (2015): *Boolean Formulas for the Static Identification of Injection Attacks in Java.* In: Proceedings of LPAR '15, Lecture Notes in Computer Science, Springer.

[13] P. Ferrara & F. Spoto (2018): *Static Analysis for GDPR Compliance.* In: Proceedings of ITASEC '18.

[14] P. Ferrara, F. Spoto & O. Olivieri (2018): *Tailoring Taint Analysis to GDPR.* In: Proceedings of APF '18.

[15] Gartner (2016): *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015.* Available at `https://www.gartner.com/newsroom/id/3165317`.

[16] Grammatech (Accessed On: 05-Nov-2018): *CodeSonar.* `https://www.grammatech.com/products/codesonar`.

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

[18] Amit Kr Mandal, Agostino Cortesi, Pietro Ferrara, Federica Panarotto & Fausto Spoto (2018): *Vulnerability Analysis of Android Auto Infotainment Apps.* In: Proceedings of the 15th ACM International Conference on Computing Frontiers, CF '18, ACM, New York, NY, USA, pp. 183–190, doi:10.1145/3203217.3203278. Available at `http://doi.acm.org/10.1145/3203217.3203278`.

[19] Mathworks (Accessed On: 05-Nov-2018): *Polyspace.* `https://www.mathworks.com/products/polyspace.html`.

[20] Jacob Matthews & Robert Bruce Findler (2007): *Operational semantics for multi-language programs.* ACM SIGPLAN Notices 42(1), pp. 3–10.

[21] Andrew C. Myers (1999): *JFlow: Practical Mostly-static Information Flow Control.* In: Proceedings of POPL '99, ACM.

[22] Oracle (Accessed On: 09-Nov-2018): *Java Native Interface.* `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`.

[23] OWASP (2018): *Top 10 Project 2017*. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[24] Federica Panarotto, Agostino Cortesi, Pietro Ferrara, Amit Kr Mandal & Fausto Spoto (2018): *Static Analysis of Android Apps Interaction with Automotive CAN*. In Meikang Qiu, editor: *Smart Computing and Communication*, Springer International Publishing, Cham, pp. 114–123.

[25] Anthony Pioli & Michael Hind (1999): *Combining interprocedural pointer analysis and conditional constant propagation*. IBM Thomas J. Watson Research Division.

[26] F. Spoto (2016): *The Julia Static Analyzer for Java*. In: *Proceedings of SAS '16*, Lecture Notes in Computer Science, Springer.

[27] Gang Tan (2015): *JNI Light: An operational model for the core JNI*. *Mathematical Structures in Computer Science* 25(4), pp. 805–840.

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