Static Analysis of Android Apps Interaction with Automotive CAN

Federica Panarotto[1], Agostino Cortesi[2], Pietro Ferrara[3(✉)], Amit Kr Mandal[2,4], and Fausto Spoto[1]

[1] Università di Verona, Verona, Italy
federica.panarotto@gmail.com,fausto.spoto@univr.it
[2] Università Ca' Foscari, Venezia, Italy
{cortesi,amitkr.mandal}@unive.it
[3] JuliaSoft Srl, Verona, Italy
pietro.ferrara@juliasoft.com
[4] BML Munjal Univesity, Gurgaon, Haryana, India
amitmandal.nitdgp@gmail.com

Abstract. Modern car infotainment systems allow users to connect an Android device to the vehicle. The device then interacts with the hardware of the car, hence providing new interaction mechanisms to the driver. However, this can be misused and become a major security breach into the car, with subsequent security concerns: the Android device can both read sensitive data (speed, model, airbag status) and send dangerous commands (brake, lock, airbag explosion). Moreover, this scenario is unsettling since Android devices connect to the cloud, opening the door to remote attacks by malicious users or the cyberspace. The OpenXC platform is an open-source API that allows Android apps to interact with the car's hardware. This article studies this library and shows how it can be used to create injection attacks. Moreover, it introduces a novel static analysis that identifies such attacks before they actually occur. It has been implemented in the Julia static analyzer and finds injection vulnerabilities in actual apps from the Google Play marketplace.

1 Introduction

Car industry is quickly introducing Android devices in cars, to provide new infotainment options to the driver. Various existing Android apps already connect to the car and provide info about the status of its hardware, the history of its movements or the driving style. Moreover, they connect to the Internet, hence gather information about the nearby area or the presence of parking slots. Such possibilities enhance the driving experience, but are also security concerns since apps can leak arbitrary data, including sensitive information on car, movements and drivers [7]. Moreover, they can send dangerous commands: lock or unlock the car, activate its brakes, turn the engine on or off, accelerate, turn on the windshield wipers, and so on. Hence, such apps need very high security standards, or the might otherwise expose driver and passengers to serious physical threats.

M. Qiu (Ed.): SmartCom 2018, LNCS 11344, pp. 114–123, 2018.
https://doi.org/10.1007/978-3-030-05755-8_12

In particular, injection of data and commands by a malicious user or by the outside world should be forbidden, as well as the unconstrainted communication of sensitive data about the car and its sensors.

This article targets OpenXC[1], an open-source library for the programmatic connection of Android apps embedded in cars to the hardware of the car. The Google Play Store already contains various apps that use this library. The Automotive Grade Linux Foundation Workgroup uses OpenXC for low level access to internal car information[2]. The *traffic tamer app challenge*[3] (dealing with the traffic in London) uses OpenXC. Apps connect to OpenXC services to read sensitive data or send commands, by using its API methods. In terms of information flow and taint analysis [10], such methods are sources and sinks of tainted data, respectively. Injection attacks occur when the user or the external world injects data or commands that reach a sink; privacy issues occur when sources are used to read sensitive data that flows towards the outside world. Static taint analysis of Java has already been widely applied to identify injection attacks, for instance in the Julia analyzer [5]. This article leverages and instantiates this approach to automatically verify apps that use OpenXC; it reports several examples of Android apps where our technique finds vulnerabilities, automatically, and compares the results with those of other static analyzers. The theory and implementation of the injection analysis of Julia is already fully described in [5] and is only briefly introduced in Sect. 3, since it is not the topic of this article.

Modern vehicles connect their embedded hardware, such as sensors and actuators, through an electronic bus. External devices can be plugged in the bus through an OBD II port and send AT commands. The most adopted connection device is the ELM327, whose AT commands are publicly available online[4]. The CAN bus protocol is the most widely adopted standard bus in both USA and Europe. It was meant to be fast and robust, hence uses unauthenticated and unencrypted communication. However, the CAN is nowadays connected to the driver and external world, even to the Internet, by using smartphones and tablets plugged in via Bluetooth or USB. This paves the way to security attacks to the car and to privacy leaks of the transferred data [1,3,6]. An attacker might even be granted complete control over the vehicle's systems [3]. More recently, authentication has been added [12]; this increases latency time but does not completely solve injection issues, nor applies to legacy systems.

There are a few software layers for connecting to the CAN, trying to become the industry standard. This article focuses on one such layer, namely, on OpenXC, since it is free, open-source and already distributed on Google Play. OpenXC is an automotive middleware and hardware platform supported by Ford Motors as an evolution of its AppLink technology. Alternatives layers

[1] http://openxcplatform.com
[2] http://docs.automotivelinux.org/docs/apis_services/en/dev/reference/signaling/architecture.html#reusing-existinglegacy-code
[3] https://traffic.devpost.com/
[4] https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf

Fig. 1. A schematic description of the connection between car and OpenXC.

```
public interface VehicleManager {
  public @UntrustedDevice Measurement get(@DeviceTrusted Class<? extends Measurement> msrmttp);
  public @UntrustedDevice VehicleMessage get(@DeviceTrusted MessageKey key);
  public @UntrustedDevice VehicleMessage request(@DeviceTrusted KeyedMessage msg);
  public boolean send(@DeviceTrusted Measurement msg);
  public boolean send(@DeviceTrusted VehicleMessage msg);
  public String requestCommandMessage(@DeviceTrusted CommandType type);
  public void request(@DeviceTrusted KeyedMessage msg, Listener lstnr);
  public void addListener(@DeviceTrusted Class<? extends Measurement> msrmttp, Listener lstnr);
  public @UntrustedDevice String getVehicleInterfaceDeviceId();
  public @UntrustedDevice String getVehicleInterfaceVersion();
  public @UntrustedDevice String getVehicleInterfacePlatform();
}
public interface Measurement {
  public interface Listener {
    public void receive(@UntrustedDevice Measurement msrmt);
  }
}
public interface VehicleMessage {
  public interface Listener {
    public void receive(@UntrustedDevice VehicleMessage msg);
  }
}
public interface UserSink {
  public void receive(@UntrustedDevice VehicleMessage msrmt);
}
public interface ApplicationSource {
  void handleMessage(@UntrustedDevice VehicleMessage msg);
}
public interface UsbVehicleInterface {
  boolean write(@DeviceTrusted byte[] bytes);
}
public interface NetworkVehicleInterface {
  boolean write(@DeviceTrusted byte[] bytes);
}
public interface BluetoothVehicleInterface {
  boolean write(@DeviceTrusted byte[] bytes);
}
```

Fig. 2. Java classes from OpenXC and their source/sink specifications for Julia.

are MirrorLink[5], largely used but shown insecure [8], and the new Automotive Grade Linux[6]. The results of this article can be extended to such alternatives once injection sources and sinks are identified, by using the same approach as in Sect. 4.

Figure 1 shows data flows between car, smartphone and the Internet, through OpenXC. The hardware side is an OBD II device with an installed firmware,

[5] https://mirrorlink.com
[6] https://www.automotivelinux.org

called Vehicle Interface (VI). It is configured by default in read-only mode, to access the vehicle's data by translating CAN messages into the OpenXC message format. Messages can then be pushed to a host device. To send commands and data to the VI (and hence to the car), the bus configuration must be set to `raw_writable`. The software side is OpenXC, a library whose Java API allows Android apps, coded in Java, to read and write commands to the CAN. To pass these commands as messages, OpenXC exports them as `Parcelable`s consumed by `Service`s, as typical in Android: there, *services* are abstractions of a remote data processor, where communication takes place, transparently, through remote procedure calls between the components of a distributed system. OpenXC services are *bound*, meaning that the app receives a stub object whose methods handle, transparently, the interprocess method calls. By invoking such methods, this allows direct and fast communication between software components. The OpenXC Android manifest exports a `com.openxc.remote.VehicleService` towards the hardware of the car and another `com.openxc.VehicleManager` service towards the Java client app. An app can bind the latter service and use Java code for creating objects of a class `VehicleMessage` to interact with the CAN. The OpenXC API consists of Java classes, including interfaces and stubs for the above services. The main class for interacting to the CAN is the above mentioned `VehicleManager`. It exports methods that allow an app to read and write measurements, send commands to the CAN, register listeners for receiving data updates and access sensitive information about the hardware VI. The full description of this API is available online[7]. Figure 2 reports just methods and listeners of `VehicleManager` that are relevant to this article. In terms of taint analysis, we anticipate that such methods are either sources of sensitive data or sinks of dangerous commands, or even both at a time.

2 Examples of Injections in Android Apps Using OpenXC

We analyzed open-source, third-party apps using OpenXC, mostly from https://github.com/openxc; two come from the Google's Play Store, in Dalvik bytecode, and have been translated into Java bytecode through `dex2jar` and `apktool`. We classify these apps on the basis of our findings.

A Privacy Breaking App. Rain Monitor[8] uses OpenXC to access sensitive data: car location, windshield status and speed. It sends it to a remote web service, that uses it to inform drivers about showers in their area. The status of the HTTP request and of the windshield are also logged. These are injections: flow of sensitive data into dangerous operations. In this case, the operations divulge sensitive data, violating privacy. Rain Monitor also reads the car position from the CAN and logs it. Hence, anybody with access to the logs can reconstruct the movements of the vehicle, a clear privacy issue. This code builds a URL by

[7] http://android.openxcplatform.com/reference/com/openxc/VehicleManager.html
[8] https://github.com/openxc/rain

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.

An App that Injects Data into the CAN. OpenXC Enabler[9] is a tutorial app meant to test and document most functionalities of OpenXC. It shows the possibility of typing and sending arbitrary messages to the CAN (see Fig. 3). The user formats the messages as requested by the protocol, *i.e.*, CAN bus number, ID of a target sensor or actuator and a value containing multiple CAN signals, in JSON format, such as {`"bus": 1, "id": 43, "value": "0x0102003040506ABCD"`}. That message gets sent to the sensor or actuator. This app features a flow of information from user input into the CAN, that is, an injection of data into the CAN.

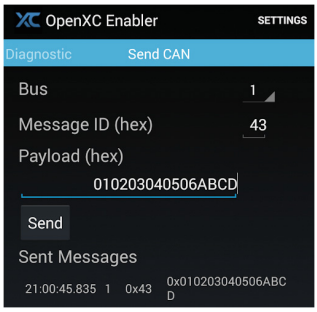

Fig. 3. OpenXC enabler apps.

Apps that Keep CAN Data Inside their Logic. Shift Knob[10] tracks vehicle information from the CAN and provides to the driver haptic and visual suggestions about good driving style, by vibrating the shift knob. Clearly, it accesses CAN data, but keeps it inside the app. Data is reported in the app's UI, but never divulged externally, for instance through the Internet. Hence, this app does not feature any injection. Night Vision[11] "adds night vision to a car with off-the-shelf parts. The webcam faces forward [...] and uses edge detection to detect objects on the road". It uses OpenXC only for listening to the headlamps status. When the headlamps are turned on, a listener starts the main activity of the app. Sensitive data (the state of the headlamps) is only used inside the logic of the app and does not escape from it. Hence, this app does not feature any injection. Dynamic Skip Fire[12] is "used [...] to showcase Tula Technology's for cars"[13]. It shows the fuel efficiency rate of 7–15% through optimized combustion and reduced engine pumping losses. We downloaded this app from the Play Store but could not find its source code. Hence, we analyzed its behavior in the Android emulator and looked at its bytecode. Also this app uses sensitive data inside its internal logic only.

An App Where CAN Data Flows into a Database, Sanitized. MPG[14] stores trips information, fuel consumption and efficiency into a local SQLite database. Hence this app builds an information flow from sensitive data from the CAN into a database. This could allow a dangerous SQL-injection, but data undergoes sanitization before the database update and no SQL-injection occurs.

[9] https://play.google.com/store/apps/details?id=com.openxcplatform.enabler
[10] http://openxcplatform.com/projects/shift-knob.html
[11] http://openxcplatform.com/projects/nightvision.html
[12] https://apkpure.com/dsf/com.ntt.customgaugeview
[13] https://www.tulatech.com/dsf-overview/
[14] https://github.com/openxc/mpg

3 Taint Analysis for Java and Android

Our work builds on the Julia static analyzer [11] for Java and Android bytecode, based on abstract interpretation [4]. Julia starts the analysis from a set of entry points and builds a semantic model of the execution of a program. Namely, all methods reachable, recursively, from the entry points get analyzed. The selection of the entry points can be done in three ways: (i) they are `main` methods; (ii) they are public methods (this is the default); (iii) they are public and protected methods. A larger set of entry points induces a larger set of reachable methods, weaker method call patterns and, in general, more warnings. The selection of the entry points is different in Android, whose execution model heavily relies on event handlers. Hence, Julia scans the Android manifest, looking for XML elements declaring services, activities, receivers and content providers. Then Julia creates a synthetic method that simulates the lifecycle of such components (*e.g.*, an activity starts with a call to `onCreate()`, followed by calls to `onStart()`, `onStop()` and `onResume()`). This method is then an entry point [9].

Among its checkers, Julia includes the Injection checker for a sound information flow analysis [5]. It propagates tainted data along all possible information flows. Boolean variables stand for program variables. Boolean formulas model explicit information flows. Namely, their models are a sound overapproximation of all taintedness behaviors for the variables in scope at a given program point. For instance, the abstraction of the `load k` bytecode instruction, that pushes on the operand stack the value of local variable k, is the Boolean formula $(\check{l}_k \leftrightarrow \hat{s}_{top}) \wedge U$, stating that the taintedness of the topmost stack element after this instruction is equal to that of local variable k before the instruction; all other local variables and stack elements do not change (expressed by a formula U); taintedness before and after an instruction is distinguished by using distinct hats for variables. There are such formulas for each bytecode instruction. Instructions that might have side-effects (field updates, array writes and method calls) need some approximation of the heap, to model the possible effects of the updates. The analysis of sequential instructions is merged through a sequential composition of formulas. Loops and recursion are saturated by fixpoint. The resulting analysis is a denotational, bottom-up taint analysis, that Julia implements through efficient binary decision diagrams [2].

The taint analysis of Julia uses a dictionary of sources and sinks for Android. Sources include methods accessing sensitive information about the user or device, or reading data from UI widgets; sinks include methods for logging or for database or network manipulation. The analysis of a source forces the corresponding Boolean variable to true. At each sink, the analyzer checks if the corresponding Boolean local variable is definitely false. In that case, no flow of tainted data into that sink is possible; otherwise, Julia issues a warning, reporting a potential flow of tainted data into the sink. This approach uses a single Boolean mark for all sources. Hence, it is inherently impossible to distinguish different origins of tainted data. However, this limitation justifies its scalability.

4 Instantiation to OpenXC

Figure 2 reports the methods of OpenXC that either produce (*sources*) sensitive, tainted data, that should not flow into sensitive locations, or receive (*sinks*) data that must be untainted, since it might flow into the CAN. This information was in the mind of the library developers, and is not explicit in code. To use the taint analysis of Julia, it must be first made explicit, in a format that Julia understands. Currently, Julia allows one to instantiate its taint analysis with the addition of sources and sinks, given either as an XML file or as annotated interfaces. This article exploits the latter possibility. Namely, the annotated interfaces in Fig. 2 are given to Julia before the analysis, with annotations for sources (`@UntrustedDevice`) or sinks (`@DeviceTrusted`). For instance, methods `get` receive a parameter that specifies the kind of information that must be read from the CAN. Hence, that parameter must not be freely in control of the user of the application, or otherwise she might be able to build an injection into the CAN device. That is, it is a sink. Moreover, the value returned by such `get` methods discloses sensitive information about the car. Consequently, it must be used in a proper way or otherwise privacy might be jeopardized. Hence, it is a source. Also the parameter of the `receive` method of the listeners is a source, since it carries data reporting updates about the car status. Hence, it is annotated as `@UntrustedDevice`. Note that these annotations must be manually provided for OpenXC, once and for all. They cannot be automatically inferred, either statically or dynamically, since they follow from the intended semantics of OpenXC, which is only described in its plain English documentation. Any other taint analyzer would need that same information.

Once Julia receives such annotated interfaces, it can perform a taint analysis, aware of those extra sources and sinks. Sources are marked as tainted during the analysis and propagated. Sinks are checked for taintedness at the end of the analysis: if they are tainted, Julia issues a warning about a potential injection.

5 Experiments

We have analyzed the apps from Sect. 2 with the taint analysis of Julia, instantiated with the annotation in Fig. 2. The analyses require up to 3 min per app on a standard desktop Intel Core i7 with 16GB of RAM. We have monitored and captured the network traffic of the apps, in the Android emulator of Android Studio and with the VI simulator[15] by WireShark[16].

The analysis of Rain Monitor issues the following injection warnings:

```
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>"
```

[15] https://github.com/openxc/openxc-vehicle-simulator
[16] https://www.wireshark.org/

These correspond to the injections informally discussed in Sect. 2. By analyzing the network traffic with Wireshark, we have found a package sent to the IP address 2.17.206.167, that corresponds to a company that supplies Internet services such as a cloud database. This is definitely a dangerous injection, but not exactly a cross-site scripting (XSS) injection, as Julia suggests, since it cannot distinguish the source of tainted data. Moreover, the status of the HTTP request and the status of the windshield get logged into a file (second and third warning). The former is an example of data coming from the external world (the HTTP server might be compromised and send any possible status); the latter is an example of sensitive car data. Sensitive data (latitude and longitude) is read from the CAN, logged (fourth warning) and later used to build a URL (fifth warning). The latter points to a remote web service that tracks the position of the car and the weather. This is a potential privacy breach. Hence, the analysis only issues true alarms here, although inherent to the task of the app.

The analysis of OpenXC Enabler issues seven injection warnings, including:

```
SendCanMessageFragment.java:110:Device injection into method "send"
NetworkPreferenceManager.java:53:Log forging into method "w"
```

The first corresponds to the injection discussed in Sect. 2. Namely, data coming from user-controlled widgets flows into the `send` method and hence to the CAN. The second corresponds to the other discussed in the same section, about the flow of user-controlled preferences into the logs. Another warning is similar to the first (line 127 of `DiagnosticRequestFragment.java`). Four more warnings are similar to the second, that is, they warn about data from the preferences of the app (hence under user control) that can flow into logs (line 480 of `SettingsActivity.java`, line 69 of `PreferenceManagerService.java` and line 72 of `TraceSourcePreferenceManager.java`) or into the specification of a file name (*path-traversal*: line 72 of `viewTraces.java`). These warnings are true alarms. The analysis of the network traffic shows many ack packages sent to the VI simulator and some non-empty packages, unfortunately coded in hexadecimal, corresponding to commands sent from the user to the CAN bus simulator.

For Shift Knob, Night Vision and Dynamic Skip Fire, Julia issues no warning. This is in line with the fact that sensitive data from the CAN flows in a controlled way inside those apps and never reaches critical operations nor leaves the device. The analysis of the network traffic reports packages from the VI simulator, that is, the CAN information, and from no other interesting IP address. We have no analysis for the other two apps since, during emulation, they crashed repeatedly.

For MPG, Julia issues only one warning, at line 343 of `MpgActivity.java`. There, an option from Android preferences (hence controlled by the user) is used in a call to `Thread.sleep`. This allows a denial-of-service injection by setting a large integer value in the preferences. More interestingly, Julia does not issue any warning where a method `insertOrThrow` is used to update an SQL database. Julia does consider the query to that method as potentially tainted, but that method is not in the list of sinks provided to Julia, since it is known to sanitize data used to perform the query. Hence, no warning is issued there. We have no network traffic analysis since the app crashed repeatedly during emulation.

The last results are relevant, since they show the power of the tool: not only did it identify several real issues, but it did not issue false alarms here. In these experiments, we have used the Injection checker of Julia, that performs a taint analysis. Julia includes other checkers, that issue other warnings on these apps. They are not injections, nor security issues, but mainly related to potential bugs and inefficiencies. As such, they are not considered here.

We have analyzed the same apps with other static analyzers: FindBugs[17], SpotBugs[18], SonarQube[19], Qark[20] and FlowDroid[21]. They do not identify any of the above injections. Some of them do issue warnings tagged as *security issues*, by using some syntactical check of the code. Namely, SonarQube complains about the fact that some public fields should have been declared as `final`, since they are never modified; or that some visibility modifier is too weak; it also complains about calls to `File.delete()` with no check on the returned value, which in Java is meant to inform about the outcome of the operation. Julia would issue the same warnings, had the corresponding checkers been turned on; however, it does not tag them as security issues but rather as bugs or inefficiencies. Dynamic Skip Fire has not been analyzed with these tools, since they do not work on bytecode. Qark issues warnings about a too small `minSdkVersion` in the `AndroidManifest.xml`, which is known to allow some security problems; it also warns about the run-time registration of Android broadcast receivers, that might allow some form of data hijacking. FlowDroid issues Android security warnings about writing information in a log file, since it warns at *all* logging calls. The same happens for method `putString`, that FlowDroid assumes to *always* inject tainted data into an intent. These are simple syntactical checks of the code, since the analyzers do not make any effort in proving that the risk is real or only potential, which results in false alarms. These analyses are only pattern-matching. Julia avoids such false alarms through a taintedness analysis of data. Moreover, FlowDroid issues no warning about information flow from/to the CAN. FindBugs and SpotBugs issue no security warnings on these apps.

6 Conclusion

This article instantiated the taint analysis of Julia [5] with a specification of sources and sinks for OpenXC. The resulting taint analysis finds security vulnerabilities in actual third-party apps interacting with the car CAN bus. They are injections, that is, either a safeness issue (the user of the app or the external world can control safety critical aspects of the car) or a privacy issue (sensitive data about the car escape into the external world). Comparison with five other tools for static analysis shows that only Julia is able to spot such issues.

[17] http://findbugs.sourceforge.net
[18] https://spotbugs.github.io
[19] https://www.sonarqube.org
[20] https://github.com/linkedin/qark
[21] https://github.com/secure-software-engineering/FlowDroid

The actual relevance of the injection issues depends from the level of privacy and security required by a car manufacturer. In any case, the importance of these results can also be read the other way around: since the Injection checker of Julia is sound (that is, it considers all execution paths), then there is no injection into the CAN if Julia does *not* issue any warning. This allows one to understand where are the only injection risks.

References

1. Avatefipour, O., Hafeez, A., Tayyab, M., Malik, H.: Linking received packet to the transmitter through physical-fingerprinting of controller area network. In: IEEE Workshop on Information Forensics and Security (WIFS 2017), Rennes, France, pp. 1–6, December 2017
2. Bryant, R.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
3. Checkoway, S., et al.: Comprehensive experimental analyses of automotive attack surfaces. In: 20th USENIX Security Symposium, SanFrancisco, CA, USA. USENIX Association, August 2011
4. 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)
5. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in Java. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 130–145. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_10
6. Koscher, K., et al.: Experimental security analysis of a modern automobile. In: 31st IEEE Symposium on Security and Privacy (S&P 2010), Berleley/Oakland, California, USA, pp. 447–462. IEEE Computer Society, May 2010
7. Mandal, A.K., Cortesi, A., Ferrara, P., Panarotto, F., Spoto, F.: Vulnerability analysis of android auto infotainment apps. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 183–190. ACM (2018)
8. Mazloom, S., Rezaeirad, M., Hunter, A., McCoy, D.: A security analysis of an in-vehicle infotainment and app platform. In: 10th USENIX Workshop on Offensive Technologies (WOOT 2016). USENIX Association, Austin, August 2016
9. Payet, É., Spoto, F.: Static analysis of android programs. Inf. Softw. Technol. **54**(11), 1192–1201 (2012)
10. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003)
11. Spoto, F.: The Julia static analyzer for Java. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 39–57. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_3
12. Wang, Q., Sawhney, S.: VeCure: a practical security framework to protect the CAN bus of vehicles. In: 4th International Conference on the Internet of Things (IOT 2014), Cambridge, MA, USA, pp. 13–18. IEEE, October 2014