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@unive.it, amitmandal.nitdgp@gmail.com
[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 can then interact with all hardware components of the car. This can for instance provide new interaction mechanisms to the driver. However, this can also be misused, becoming 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 are usually connected to the cloud, opening the door to remote attacks from malicious users or the cyberspace. The OpenXC platform is an open source API that allows Android application to interact with the car's hardware. In this article, we study this library and show how it can be used to create injection attacks. Moreover, we introduce a novel static analysis that identifies such attacks before they occur in real life. This analysis has been implemented in the Julia static analyzer and finds injection vulnerabilities in actual apps published in the Google Play marketplace.

1 Introduction

Car industry is quickly moving to the introduction of Android devices in cars, to provide new infotainment functionalities to the driver. For instance, various existing Android apps connect to the car and provide information about the status of its hardware, the history of its movements or the driving style. Moreover, such apps connect to the Internet, hence they can 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 about the car, its history and its drivers. Moreover, they can send dangerous commands: they can lock or unlock the car, activate the brakes, turn the engine on or off, accelerate, turn on the windshield

wipers, and so on. Conjunction of these possibilities with Internet connection means that such apps need very high security standards, as otherwise might expose the driver and the passengers to serious physical threats. In particular, injection of data and commands from a malicious user or from the outside world should be forbidden, as well as the unconstrainted communication of sensitive data about the car and its sensors.

This article considers a specific library that allows the programmatic connection of Android apps embedded in cars to the hardware of the car. The OpenXC platform[5] is an open source Java library that interacts with several hardware services. The Google Play Store contains already different apps using this library, and a Linux Foundation Workgroup (Automotive Grade Linux - AGL) uses this library for low level access to the internal car information[6]; The "Traffic tamer app challenge"[7] (related to the traffic in London) uses that library. Apps connect to the library services, to read sensitive data or send commands, by using the methods of the OpenXC API. In terms of information flow and taint analysis [9], these 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, while 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 the automatic verification of apps using the OpenXC library. We report several examples of Android apps where our technique finds vulnerabilities, automatically, and compare the results with those of other static analysis tools.

This article takes the perspective of the user of the Julia analyzer, that wants to instantiate it to the analysis of OpenXC code. The theory and implementation of the injection analysis of Julia is already fully described in [5] and is only briefly introduced in Sec. 3. It is not, however, the topic of this article. The interested reader is referred to that work.

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[8]. The CAN bus protocol is the most widely adopted standard bus in both USA and Europe. It was designed to be fast and robust, hence communication is unauthenticated and unencrypted. However, the CAN is nowadays connected to the driver and the 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, as shown in [3, 6, 1]. Such articles exemplify how an attacker can be granted complete control over

[5] `http://openxcplatform.com`

[6] `http://docs.automotivelinux.org/docs/apis_services/en/dev/reference/signaling/architecture.html#reusing-existinglegacy-code`

[7] `https://traffic.devpost.com/`

[8] `https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf`

Fig. 1: A schematic description of the connection between car, smartphone and the Internet, through the OpenXC library. The latter exports a service that Android apps can bind to, to access the CAN. Globally, this establishes an insecure connection between the CAN and the Internet.

the vehicle's systems [3]. More recently, authentication has been added to protocols [11]; this increases the 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. The common aspect of the whole set of apps we have found (also for electrical vehicle) is the use of services for the communication between Android with the OBD-II protocol. This article focuses 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 are MirrorLink[9], largely used but shown to be insecure [7], and the new Automotive Grade Linux[10]. The results of this article can be extended to such alternatives once their injection sources and sinks are identified, by using the same approach described in Sec. 4.

Fig. 1 shows how data flows between car, smartphone and the Internet, through OpenXC. The hardware side is an OBD II device with an installed firmware, called Vehicle Interface (VI). It is configured by default in read-only mode, that allows one to access the vehicle's data by translating CAN messages into the OpenXC message format. Messages can then be pushed to a host device. Commands and data can also be sent (*written*) to the VI (and hence to the car) by setting the firmware bus configuration to `raw_writable`. The software side of OpenXC is a library whose API, in its Java version, allows Android apps, coded in Java, to read and write commands to the CAN. To pass these commands as messages, the library exports them as `Parcelable` objects consumed by `Service`s, as typical in Android programming. Services are Android 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 invok-

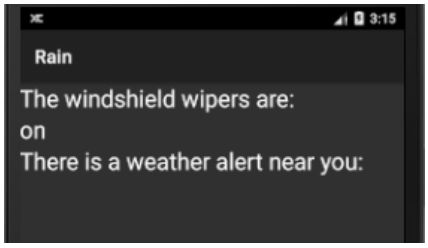
[9] `https://mirrorlink.com`
[10] `https://www.automotivelinux.org`

```
public class VehicleManager extends Service ... {
  public Measurement get(Class<? extends Measurement> msrmtType); // read from the CAN
  public VehicleMessage get(MessageKey key); // read from the CAN
  public VehicleMessage request(KeyedMessage msg); // read from the CAN, waiting up to 2 seconds
  public boolean send(Measurement msg); // send to the CAN
  public boolean send(VehicleMessage msg); // send to the CAN
  // send to the CAN and yields the result from the CAN
  public String requestCommandMessage(CommandType type);
  // register a listener for receiving updates
  public void request(KeyedMessage msg, VehicleMessage.Listener listnr);
  // register a listener for receiving updates to the given measurement
  public void addListener(Class<? extends Measurement> msrmtType, Measurement.Listener listnr);
  public String getVehicleInterfaceDeviceId(); // yield the device identifier
  public String getVehicleInterfaceVersion(); // yield the firmware version
}
public interface Measurement {
  public interface Listener {
    public void receive(Measurement measurement); // get notified about a measurement change
  }}
public class VehicleMessage ... {
  public interface Listener {
    public void receive(VehicleMessage message); // get notified about a new
message from the CAN
  }}
public class UserSink {
  // get notified about a measurement change from the CAN
  public void receive(VehicleMessage measurement);
}
public class ApplicationSource {
  // get notified about a new message from the CAN
  void handleMessage(VehicleMessage message);
}
public class UsbVehicleInterface {
  boolean write(byte[] bytes); // send raw data to the CAN
}
public class NetworkVehicleInterface {
  boolean write(byte[] bytes); // send raw data to the CAN
}
public class BluetoothVehicleInterface {
  boolean write(byte[] bytes); // send raw data to the CAN
}
```

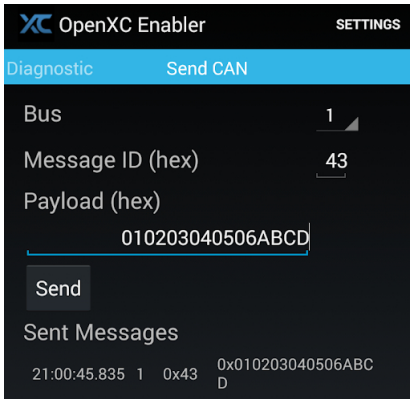
Fig. 2: Java classes of OpenXC and their methods that allow Android apps to interact with the CAN.

ing such methods, this technique allows direct and fast communication between software components.

The Android manifest of the OpenXC library exports an Android service `com.openxc.remote.VehicleService` towards the hardware of the car and another service `com.openxc.VehicleManager` towards the Java client app. Hence, 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` service. 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



(a) The Rain Monitor app.



(b) The OpenXC Enabler app.

Fig. 3: Screenshots from the first two Android apps that use the OpenXC library, analyzed in this article. These apps feature injections that our tool identifies, automatically.

available online[11]. Fig. 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.

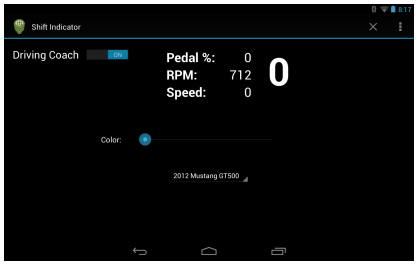
2 Examples of Injections in Android Apps using OpenXC

This section presents third-party apps that use the OpenXC library. Most are open-source, downloadable at `https://github.com/openxc`, except two of them that are downloadable from the Google's Play Store in Dalvik bytecode. For analyzing them we translate the their code into Java bytecode through tools `dex2jar` and `apktool`. Two apps are vulnerable to opposite kinds of injections. Another three, instead, deal with CAN data in a controlled way, in the sense that data does not go outside the app; hence, they are not vulnerable to injections. A last app lets CAN data flow into an SQL database, but in a way that is definitely safe against SQL-injections. It is important to consider also these *safe* apps, since they allow one to show that a static analysis tool does not issue too many spurious warnings.

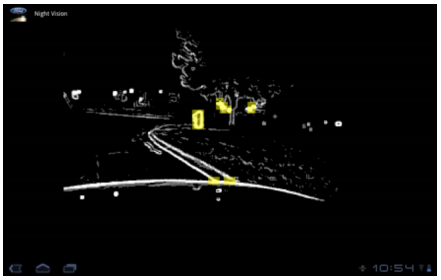
A Privacy Breaking App Rain Monitor[12] uses OpenXC to collect sensitive data, location of the car, as windshield status, and speed, and sends it to a remote web service, where it is collected and used to inform drivers of possible showers in their area. A screenshot is in Fig. 3a. Clearly, this app contains a

[11] `http://android.openxcplatform.com/reference/com/openxc/VehicleManager.html`

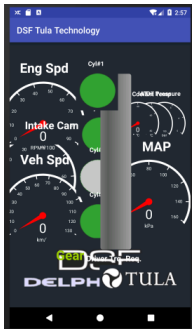
[12] `https://github.com/openxc/rain`



(a) The Shift Knob app.



(b) The Night Vision app.



(c) The DSF app.

(d) The MPG app.

Fig. 4: Screenshots from four more Android apps that use the OpenXC library, analyzed in this article. These apps do not feature any injection from the CAN nor to the CAN. Our tool confirms this.

leakage of sensitive data into the Internet, which can be seen as a privacy issue. Fig. 5 reports a code snippet where the OpenXC library is used to read the car location and windshield data, which at the end get sent to the Internet, without encryption nor authentication. The status of the HTTP request and of the windshield gets also reported in a log. These are instances of injections: flow of sensitive data into dangerous operations. In this case, the operations divulge sensitive information, violating privacy. Fig. 6 reports another code snippet from the same app. It 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.

An App that Injects Data into the CAN OpenXC Enabler[13] is a tutorial app meant to test and document most functionalities of OpenXC, bun-

[13] `https://play.google.com/store/apps/details?id=com.openxcplatform.enabler`

```
public class CheckWipersTask ... {
  private final String WUNDERGROUND_URL =
    "http://www.wunderground.com/weatherstation/VehicleWeatherUpdate.php";
  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);
  }

  private void uploadWiperStatus
      (Latitude latitude, Longitude longitude, WindshieldWiperStatus wiperStatus) {

    int wiperSpeed = 0;
    boolean wiperStatusValue = wiperStatus.getValue().booleanValue();
    if (wiperStatusValue)
      wiperSpeed = 1;
    String finalUri = WUNDERGROUND_URL + "?wiperspd=" + wiperSpeed +
                          "&lat=" + latitude + "&lon=" + longitude;
    ...
    HttpClient client = new DefaultHttpClient();
    // send the CAN data on the Internet and receive an ack back
    HttpGet request = new HttpGet(finalUri);
    HttpResponse response = client.execute(request); // line 111
    int statusCode = response.getStatusLine().getStatusCode();
    if (statusCode != HttpStatus.SC_OK)
      Log.w(TAG, "Error " + statusCode + // line 114
        " while uploading wiper status");
    else
      Log.d(TAG, "Wiper status (" + wiperStatus + ") uploaded successfully"); // line 117
  }
}
```

Fig. 5: A code snippet from the Rain Monitor app. Sensitive car data is sent to the Internet and logged.

dled with the library. It shows the possibility of typing and sending arbitrary messages to the CAN, as shown in Fig. 3b. The user format 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 for it, in JSON format, such as `{"bus": 1, "id": 43, "value": "0x0102003040506ABCD"}` That message gets delivered to the sensor or actuator, which will react accordingly if the safety flag is turn off. This app features a flow of information from user input into the CAN, that is, an injection of data into the CAN. Fig. 7 reports a code snippet where data is collected from GUI widgets, packed into a message and sent to the CAN. Again, no encryption nor authentication is used.

Apps that Let CAN Data Flow in their Internal Logic Only The Shift Knob app[14] monitors vehicle information, coming from the CAN, and provides an haptic and visual suggestions to the driver about good driving style, by

[14] `http://openxcplatform.com/projects/shift-knob.html`

```
public class FetchAlertsTask extends TimerTask {
  private final String TAG = "FetchAlertsTask";
  private final String API_URL = "http://api.wunderground.com/api/dcffc57e05a81ad8/alerts/q/";
  ...
  public void run() {
    Latitude latitude = (Latitude) mVehicle.get(Latitude.class);
    Longitude longitude = (Longitude) mVehicle.get(Longitude.class);
    double latitudeValue = latitude.getValue().doubleValue();
    double longitudeValue = longitude.getValue().doubleValue();
    ...
    Log.d(TAG, "Querying for alerts near " + latitudeValue + ", " + longitudeValue); // line 68
    ...
    StringBuilder urlBuilder = new StringBuilder(API_URL);
    urlBuilder.append(latitudeValue + "," + longitudeValue + ".json");
    URL wunderground = new URL(urlBuilder.toString()); // line 76
    ...  }  ... }
```

Fig. 6: Another code snippet from the Rain Monitor app. Sensitive car data is logged and used to build a URL address.

```
public class SendCanMessageFragment ... {
  private void onSendCanMessage
      (Spinner busSpinner, EditText idView, EditText payloadView) {
    ...
    // construct a message by attaching every parameter coming from the user
    CanMessage message = new CanMessage(
      Integer.valueOf(busSpinner.getSelectedItem().toString()),
      Integer.valueOf(idView.getText().toString(), 16),
      ByteAdapter.hexStringToByteArray(payloadView.getText().toString()) );
    // send the message to the CAN
    mVehicleManager.send(message); // line 110
    ... } }
```

Fig. 7: A code snippet from the OpenXC Enabler app. User data is read from GUI widgets and sent to the CAN.

vibrating the shift knob. Clearly, this app accesses CAN data, but this is then used in a controlled way, only inside the logic of the app. Data is also reported in the app's UI (Fig. 4a), but never divulged through external means, such as the Internet. Hence, this app does not feature any injection. Fig. 9 shows a code snippet from this app. It defines a listener that feeds, into a UI widget, CAN data about the car speed. That data does not propagate further.

The Night Vision app[15] "adds night vision to a car with off-the-shelf parts. The webcam faces forward from the dashboard and uses edge detection to detect objects on the road in the path of the vehicle". (Fig. 4b). This app uses OpenXC only for listening to the headlamps status. This is done by the listener shown in the code snippet in Fig. 10. When the headlamps are turned on, this 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 flow outside the device. Hence, this app does not feature any injection.

[15] http://openxcplatform.com/projects/nightvision.html

```
public abstract class VehiclePreferenceManager
  ...
  protected String getPreferenceString(int id) {
    return getPreferences().getString(mContext.getString(id), null);
  }  ... }

public class NetworkPreferenceManager extends VehiclePreferenceManager {
  private final static String TAG = "NetworkPreferenceManager";
  ...
  private void setNetworkStatus(boolean enabled) {
    Log.i(TAG, "Setting network data source to " + enabled);
    if (enabled) {
      String address = getPreferenceString(R.string.network_host_key);
      String port = getPreferenceString(R.string.network_port_key);
      String adrs = address + ":" + port;
      if (address == null  port == null  !NetworkVehicleInterface.validateResource(adrs)) {
        String error = "Network host URI (" + adrs + ") not valid";
        Log.w(TAG, error); // line 53
        ...
      } } }
}
```

Fig. 8: Another code snippet from OpenXC Enabler. Data coming from Android preferences, hence chosen by the user, is concatenated into `combinedAddress` and then logged.

```
Measurement.Listener mSpeedListener = new Measurement.Listener() {
  public void receive(Measurement measurement) {
    final VehicleSpeed updated_value = (VehicleSpeed) measurement;
    mVehicleSpeed = updated_value.getValue().doubleValue();
    runOnUiThread(new Runnable() {
      public void run() {
        // send vehicle speed with 1 decimal point
        mVehicleSpeedView.setText("" + Math.round(mVehicleSpeed * 10) / 10);
    } });
  } };
```

Fig. 9: A code snippet from the Shift Knob app. Sensitive car data coming from the CAN flows into the listener and then into a widget of the same app, but is not sent outside the device.

The Dinamic Skip Fire (DSF) app[16] is "used in conjunction with OpenXC OBD-II BT hardware and library dongle to showcase Tula Technology's DSF technology for cars[17]". It shows the fuel efficiency rate of 7-15% through optimized combustion and reduced engine pumping losses (Fig. 4c). We downloaded this app from the Play Store but could not find its source code. Thus, we have analyzed its behavior in the emulator and looked at the bytecode. As for the above apps, sensitive data is used inside the internal logic only.

An App that Lets CAN Data Flow into a Database, but in a Sanitized Way The MPG app[18] takes information about every trip with the car, calculates fuel consumption/fuel efficiency and saves the data to a local SQLite database. The same information is shown on the screen (Fig. 4d). The descrip-

[16] `https://apkpure.com/dsf/com.ntt.customgaugeview`

[17] `https://www.tulatech.com/dsf-overview/`

[18] `https://github.com/openxc/mpg`

```
Measurement.Listener mHeadlampListener = new Measurement.Listener() {
  public void receive(Measurement measurement) {
    final HeadlampStatus status = (HeadlampStatus) measurement;
    mHandler.post(new Runnable() {
      public void run() {
        if (status.getValue().booleanValue()) { // are headlamps on?
          if (!NightVisionActivity.isRunning()) {
            Intent intent = new Intent(VehicleMonitoringService.this, NightVisionActivity.class);
            intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            VehicleMonitoringService.this.startActivity(intent);
          } }
        else { sendBroadcast(new Intent(ACTION_VEHICLE_HEADLAMPS_OFF)); } }
    });
  } };
```

Fig. 10: A code snippet from the Night Vision app. Sensitive car data coming from the CAN flows into the listener, but it is not sent outside the device.

```
  public void saveResults(double dist, double fuel, double mileage, double start, double end) {
    double length = (end-start) / (1000*60);
    ContentValues values = new ContentValues();
    values.put(C_DISTANCE, dist);
    Timestamp time = new Timestamp((long) start);
    String stime = time.toString();
    values.put(C_TIME, stime);   values.put(C_LENGTH, length);
    values.put(C_FUEL, fuel);    values.put(C_MILEAGE, mileage);
    SQLiteDatabase db = getWritableDatabase();
    db.insertOrThrow(TABLE, null, values);
  }
```

Fig. 11: A code snippet from the MPG app. Sensitive car data flows from the CAN to a database. However, it is made of `double`s, not strings, hence it is hard to build an attack string from them. Moreover, the insertion method of the Android library sanitizes that data and avoids the risk of any SQL injection.

tion of the app indicates that it builds a flow of information from sensitive data coming from the CAN to a database. As such, this could be a dangerous data flow, leading to an SQL injection. Fig. 11 shows the code snippet where data is stored in the database. The arguments of method `saveResults` are tainted, since they are computed from CAN data. However, they have type `double`, hence it is hard to use them to build an SQL-injection attack string. Moreover, the `insertOrThrow` method of the Android library guarantees that the elements inside the `ContentValues` object undergo sanitization before the SQL query. That is, they are consistently escaped so that no SQL-injection attack can be built from them and no SQL-injection can occur.

3 Taint Analysis for Java and Android

Our work builds on the Julia static analyzer [10], an existing static analyzer 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 Java 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) the entry points are `main` methods; (ii) the entry points are public methods (this is the default); (iii) the entry points are public and protected methods (library mode). 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, since the execution model heavily relies on event handlers of various components. 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 synthetic method is an entry point for the analysis [8].

Among its checkers, Julia includes the Injection checker that implements 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 form 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 the taintedness 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 the 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 specific to 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, specific to Android. By default, Julia comes with a specification of the most used sources and sinks of the Android runtime. The analysis of a source forces the corresponding Boolean variable to be true. At each sink, the analyzer checks if the corresponding Boolean local variable is definitely false. If that is the case, no flow of tainted data into that sink is possible; otherwise, it 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 the scalability of the technique.

4 Instantiation for the OpenXC Library

Fig. 2 reports the methods of the OpenXC library that either produce (*sources*) sensitive, tainted data, that should not flow into sensitive locations, or receive (*sinks*) data that must not be tainted, since it might flow into the CAN device. This information was in the mind of the library developers, and it is not explicit in code. In order to use the taint analysis of Julia, it must be first made explicit, in a format that Julia can understand. Currently, Julia allows one to instantiate its taint analysis with a specification of additional 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. 12 are provided to Julia before the analysis. They reflect the methods in Fig. 2 where either sources (`@UntrustedDevice`) or sinks (`@DeviceTrusted`) occur, or both. 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. Hence, it is a sink, annotated as `@DeviceTrusted`. 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, annotated as `@UntrustedDevice`. 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` as well. Note that these annotations must be manually provided for the OpenXC library, once and for all. They cannot be statically nor dynamically inferred, automatically, since they follow from the intended semantics of the OpenXC library, 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 Experimental Results

We have analyzed the apps from Sec. 2 with the taint analysis of Julia, instantiated with the annotation in Fig. 12. The analyses require up to 3 minutes per app on a standard desktop Intel Core i7 machine with 16GB of RAM. We have monitored and captured the network traffic between every app, in the Android emulator of Android Studio and with the VI simulator[19] by WireShark[20]. Obviously, we have ignored all packages from/to IP addresses belonging to Google.

For the Rain Monitor app, the taint analysis issues the following five warnings about potential injections:

[19] `https://github.com/openxc/openxc-vehicle-simulator`
[20] `https://www.wireshark.org/`

```
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. 12: The specification of sources and sinks in the classes of OpenXC.

```
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>"
```

These correspond to the injections informally discussed in Sec. 2. In particular, as reported in Fig. 5 and detected by the first warning, sensitive data about the car flows into the `execute` method that builds an HTTP request. 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 which supplies Internet services such as a cloud database. This is definitely a dangerous injection, although not exactly a cross-site scripting (XSS) injection, as the analyzer 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, at the end of Fig. 5 (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 data about the car. In

Fig. 6, sensitive data (latitude and longitude) is read from the CAN, logged at line 68 (fourth warning) and later used to build 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 taint analysis of Julia issues five injection warnings on Rain Monitor and they are all true alarms, although inherent to the task the app has to perform.

For the OpenXC Enabler app, the taint analysis of Julia issues seven warnings about potential injections, including:

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

The former corresponds to the injection discussed in Sec. 2. Namely (Fig. 7), data coming from user-controlled widgets flows into the `send` method and hence to the CAN. The latter warning corresponds to the other discussed in the same section, about the flow of user-controlled preferences into the logs (Fig. 8). Another warning is similar to the first one (line 127 of `DiagnosticRequestFragment.java`). Four more warnings are similar to the second one, that is, they warn about data coming 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`). All seven warnings are true alarms. By analyzing the network traffic with Wireshark, we have found many ack packages sent to the VI simulator and some non-empty packages, unfortunately coded in hexadecimal, that correspond to the commands sent from the user to the CAN bus simulator.

For the apps Shift Knob, Night Vision and DSF discussed in Sec. 2, Julia issues no injection warnings. This is in line with the fact that sensitive data coming from the CAN flows in a controlled way inside those apps and is never used in a critical operation nor sent outside the device. By analyzing the network traffic of Shift Knob with Wireshark, we have found only the packages coming from the VI simulator, that is, the CAN information, and no other interesting IP address. We have no analysis for the other two apps since, during the emulation, they crashed repeatedly.

For the app MPG discussed in Sec. 2, Julia issues only one injection warning, at line 343 of `MpgActivity.java`. There, an integer 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 in the snippet of code in Fig. 11. We have verified that Julia does consider `values` as tainted. However, method `insertOrThrow` is not in the list of sinks provided to the analyzer, since it is known to sanitize data used to perform the SQL query. Hence, no warning is issued there. Unfortunately, we have no network traffic analysis since the app crashed repeatedly during the emulation.

The last results are important, since they show the power of the tool: not only it identified several real issues, but did not issue false alarms (*noise*), at least in these examples. In general, however, the analysis can have false alarms, as usual in static analysis.

These results have been obtained with the Injection checker of Julia, that performs a taint analysis of the apps. The Julia analyzer includes other checkers, that issue other warnings on these five apps. They are not injections, nor security issues, but mainly related to potential bugs and inefficiencies of the code. As such, they are not considered relevant to this article, that focuses on injections.

We have analyzed the same apps with other static analyzers: FindBugs[21], SpotBugs[22], SonarQube[23], Qark[24] and FlowDroid[25]. They do not identify any of the above injections. Some of them do issue some 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()` without checking the result 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. Furthermore, the DSF app has not been analyzed with these tools, that cannot 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 all simple syntactical checks of the code, in the sense that 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 simply pattern-matching. Julia avoids such false alarms since it performs a taintedness analysis of data, instead. Moreover, FlowDroid issues no warnings about information flow from/to the CAN bus. FindBugs and SpotBugs issue no security warning at all on the analyzed apps.

6 Conclusion

This article instantiated the taint analysis of Julia [5] with a specification of the sources and sinks of the OpenXC library that integrates Android apps into a car. The resulting taint analysis finds security vulnerabilities in actual third-party apps interacting with the car CAN bus. In particular, such vulnerabilities are injections, that is, either a safeness (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 are divulged to the external world). Comparison with five other tools for static analysis shows that only Julia is able to spot such issues.

[21] `http://findbugs.sourceforge.net`
[22] `https://spotbugs.github.io`
[23] `https://www.sonarqube.org`
[24] `https://github.com/linkedin/qark`
[25] `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 in an app.

We are currently investigating other libraries used for accessing the CAN of a car. It will be interesting to see if the same technique, that only requires the specification of sources and sinks, can be applied there. If this will be the case, the specification of external annotations can be seen as a way of personalizing the taint analysis of Julia, available to all its users.

References

1. O. Avatefipour, A. Hafeez, M. Tayyab, and H. Malik. Linking Received Packet to the Transmitter through Physical-Fingerprinting of Controller Area Network. In *IEEE Workshop on Information Forensics and Security (WIFS'17)*, pages 1–6, Rennes, France, December 2017.
2. R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Survey*, 24(3):293–318, 1992.
3. S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *20th USENIX Security Symposium*, San Francisco, CA, USA, August 2011. USENIX Association.
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
5. M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145, Suva, Fiji, 2015.
6. K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *31st IEEE Symposium on Security and Privacy (S&P 2010)*, pages 447–462, Berleley/Oakland, California, USA, May 2010. IEEE Computer Society.
7. S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy. A Security Analysis of an In-Vehicle Infotainment and App Platform. In *10th USENIX Workshop on Offensive Technologies (WOOT'16)*, Austin, TX, August 2016. USENIX Association.
8. É. Payet and F. Spoto. Static Analysis of Android Programs. *Information & Software Technology*, 54(11):1192–1201, 2012.
9. A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
10. F. Spoto. The Julia Static Analyzer for Java. In *Proc. of Static Analysis Symposium (SAS)*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57, Edinburgh, UK, 2016.
11. Q. Wang and S. Sawhney. VeCure: A Practical Security Framework to Protect the CAN Bus of Vehicles. In *4th International Conference on the Internet of Things (IOT'14)*, pages 13–18, Cambridge, MA, USA, October 2014. IEEE.