AQL-System: Manual (https://github.com/FoelliX/AQL-System/wiki)

Table of contents

Menu

AQL

The Android App Analysis Query Language (AQL)

The AQL consists of two main parts, namely AQL-Queries (compositions of AQL-Questions) and AQL-Answers. AQL-Queries enable us to ask for Android specific analysis subjects in a general, tool-independent way.

Tutorials

- Queries & Questions
- Answers

Links

Query-Question-Grammar
Answers-Schema

Queries & Questions

AQL-Queries & -Questions

AQL-Questions can be used to ask various analysis tools for certain analysis subjects in a general way. AQL-Queries represent compositions of AQL-Questions. All possible AQL-Queries are defined by this grammar.

AQL-Questions

Basically an AQL-Question always has the following structure:
`<Analysis Subject>` `<Analysis Target>` `<Ending Symbol>`

Optionally with, features (or uses) can be specified:
`<Analysis Subject>` `<Analysis Target>` (`<Features>`)? (`<Uses>`)? (`<With>`)? `<Ending Symbol>`

In the following each of these elements are described in more detail.

Analysis Subjects (Subject of interest - SOI)

Each question can ask for one of the following analysis subjects:

- `Arguments` : To ask for arbitrary information.
- `Flows` : A flow symbolizes the transfer of information from one program location to another. To describe such a flow, these two locations have to be specified.
- `Intents` : Intents are used for inter-component communication, where one component sends an intent to another component. In case of an explicit intent the receiver can be identified by a component reference. But in case of an implicit intent the receiver needs to be recognized through the information triple action, category and data.
- `IntentFilters` : Specifies the types of intents that an activity, service, or broadcast receiver can respond to.
- `IntentSinks` : Intent-sinks are special intents, so they can be described in the same way as intents can be described. The difference is that intent-sinks must always come with a reference to a program location *(Required to connect answers -- See `CONNECT` operator)*.
- `IntentSources` : Intent-sources represent the counterpart of intent-sinks. An intent-source might be the starting location of an information flow path. They can be described just like intent-sinks and intents with one small but important difference: The reference has to refer to a statement that, for instance, extracts information from an intent. *(Also required to connect answers -- See `CONNECT` operator)*.
- `Permissions` : Shows which permissions are used in a certain reference (e.g. an entire app or a specific component).
- `Sinks` : Collect sinks (statements that leak information to the outside) from an analysis target. Sinks are represented through references.
- `Slice` : Compute a program slice based on the analysis target (in this case slicing criteria) provided.
- `Sources` : Collect sources (statements that extract sensitive information) from an analysis target. Sources are represented through references.

Analysis Target (Reference)

The analysis target is specified by a chain consisting of the following elements
`Statement -> Method -> Class -> App`
In such a chain the elements `Statement` , `Method` and `Class` are optional. In a question we can ask for information inside one target (`IN`) or for information between two targets `FROM` ... `TO`). Furthermore, preprocessors can be assigned to be applied on apps.

Ending symbols

Any question contained in a query ends with one of the following symbols: `?` , `!` or `.` .

- `?` : As answer an AQL-Answer is expected,

- `!`: A file of any type is expected as answer,
- `.`: The content contained in a file is used as answer.

AQL-Queries

An AQL-Query can be a single or multiple questions combined or not via operators. An operator is always put around one or more questions. A non-trivial (more than one question only) query usually has the following structure:
`<Operator>` [`<Question(s)>`] `<Ending Symbol>`

Operators

The following operators are specified in the AQL and implemented in the AQL-System, hence, these operators are also called *default operators*.

- `FILTER`: This operator can be used in various ways.
    - When used without specifying a specific filter, it outputs the input set, but beforehand it removes all elements (permissions, intent-sinks and -sources, ...) whose reference does not appear in any flow contained in the answer. Elements without a reference are kept.
    - When an analysis subject is given as filter, the operator filters out all elements of the selected subject of interest.
    - When using a name-value-pair as filter, only the elements that contain this name-value-pair as attribute are kept.
    - A reference can also be used as filter. Only elements that refer to this reference are kept then.
- `UNIFY`: Collects all information from different AQL-Answers and puts it into one.
- `INTERSECT`: Extracts the information that appears in all AQL-Answers provided.
- `MINUS`: Takes the information contained in one AQL-Answer and removes the information also contained in another.
- `CONNECT`: Works as `UNIFY`, however, it additionally computes transitive flows and flows that can be determined by connecting intent-sinks with intent-sources. Additionally, incomplete intents-sources (only naming a component) are completed by matching them with intent-filters (naming the same component). Lastly, it adds backward flows whenever there is a intent-sink connected to an intent-source, e.g. from `setResult(..)` in the intent-source's component to another intent-source in the `onActivityResult(..)` method of the intent-sink's component.
- `CONNECT~`: Same as `CONNECT` but while connecting intent-sinks with intent-sources only the intent's and intent filter's action is taken into account (category and data are ignored).
- `SIGN`: Automatically signs the given app (using the scripts and key stored in:data/sign)
- `TOFD` / `TOAD`: These two operators transform a single AQL-Answer, holding sources and sinks, into a tool specific Source and Sink file. `TOFD` to a FlowDroid compatible `SourcesAndSinks.txt` and `TOAD` to a AmanDroid compatible `TaintSourcesAndSinks.txt`.

To identify boundaries of operators, `[` and `]` are used.

It is also possible to define your own operator (see theconfiguration tutorial).

Variables

Variables can be used to substitute parts in a following query. For example, variable`var1` can refer to query `Q`. Then `$var1` can be used one or multiple times in a following query `Q'` as a substitute of `Q`. Thereby, partial queries must not be repeated and large queries can be structured more easily.

Examples

Question Examples

The following question asks for flows inside app*A*:

```
Flows IN App('A.apk') ?
```

The next one for flows between app*A* and *B*:

```
Flows FROM App('A.apk') TO App('B.apk') ?
```

To ask for the permission(s) used by a specific statement inside app*A*, the following question can be constructed:

```
Permissions IN Statement(sendTextMessage(..))->App('A.apk') ?
```

Let us assume we got a preprocessor associated with the keyword *TEST*. To ask for Intents in a preprocessed version of*A* we formulate:

```
Intents IN App('A.apk' | 'TEST') ?
```

To influence the tool selection a specific tool can be chosen:

```
Flows IN App('A.apk') USES 'AwesomeDroid' ?
```

Also the version can be attached `<name>-<Version>`):

```
Flows IN App('A.apk') USES 'AwesomeDroid-1.3.3.7' ?
```

Or the tool with the highest priority for a certain set of features can be selected:

```
Flows IN App('A.apk') FEATURING 'TEST', 'Awesome' ?
```

Asking for a Slice of app A can be done as follows:

```
Slice FROM
    Statement('from()')->Method('a()')->Class('A')->App('A.apk')
TO
    Statement('to()')->Method('b()')->Class('B')->App('A.apk')
!
```

Having such a slice we could ask for flows in it:

```
Flows IN App({
    Slice FROM
        Statement('from()')->Method('a()')->Class('A')->App('A.apk')
    TO
        Statement('to()')->Method('b()')->Class('B')->App('A.apk')
    !
}) ?
```

To simplify such queries variables can be used (e.g. `var1`):

```
var1 = {
    Slice FROM
        Statement('from()')->Method('a()')->Class('A')->App('A.apk')
    TO
        Statement('to()')->Method('b()')->Class('B')->App('A.apk')
    !
}

Flows IN App($var1) ?
```

By using `WITH` queries can be constructed that reuse information of other queries:

```
Flows IN App('A.apk') WITH 'Sources' = {
    Sources IN App('A.apk') ?
} ?
```

of course also constants can be used:

```
Flows IN App('A.apk') WITH 'num' = '42' ?
```

Here is an example that may forward a source and sink list to let's say FlowDroid:

```
Flows IN App('A.apk') WITH 'SourcesAndSinks' = {
    TOFD [
        UNIFY [
            Sources IN App('A.apk') ?,
            Sinks IN App('A.apk') ?
        ] ?
    ] !
} USING 'FlowDroid' ?
```

Another example that uses a question ending with `.`:

```
Flows IN App('A.apk') FEATURING {
    Arguments IN App('A.apk') .
} ?
```

Query Examples

Let us assume we want to know which permission protected statements are connected. The question we could ask is:

```
UNIFY [
    Flows IN App('A.apk') ?,
    Permissions IN App('A.apk') ?
] ?
```

Assuming we downloaded an answer telling us which permission uses can be found in app *A* we could use the following query:

```
UNIFY [
    Flows IN App('A.apk') ?,
    'downloaded_permission_answer.xml' !
] ?
```

We could further filter this result by adding the `FILTER` operator. In this case we would only get Permissions that are somehow related to at least one flow:

```
FILTER [
    UNIFY [
        Flows IN App('A.apk') ?,
        Permissions IN App('A.apk') ?
    ] ?
] ?
```

To connect two answers determined for two different apps (*A*, *B*) the `CONNECT` operator can be used. Thereby, flows are generated for each pair of matching intent-sinks and -sources:

```
CONNECT [
    IntentSinks IN App('A.apk') ?,
    IntentSources IN App('B.apk') ?
] ?
```

Similarly, based on the component specified for intent-filters and -sources, they can complete each other by applying the connect operator:

```
CONNECT [
    IntentFilters IN App('A.apk') ?,
    IntentSources IN App('B.apk') ?
] ?
```

The `CONNECT` operator can also be used to compute the transitive closure of a set of flows:

```
CONNECT [
    Flows IN App('A.apk') ?
] ?
```

Transformation Example

The AQL-System also allows to configure transformation rules to declare more complex analysis strategies hidden in a simple query. To that effect the following query:

```
Flows FROM App('A.apk') TO App('B.apk') ?
```

may be transformed into:

```
FILTER [
    CONNECT [
        Flows IN App('A.apk')?,
        Flows IN App('B.apk')?,
        CONNECT [
            IntentSinks IN App('A.apk') ?,
            IntentSources IN App('B.apk') ?
        ] ?
    ] ?
] ?
```

(see Rules for more information)

Answers

AQL-Answers

The answer to an AQL-Question (ending with `?`) always comes in form of an AQL-Answer. If a question ends with an `!` the file (of any type) is used as answer. When `.` ends a question the content of the file represents the answer.
AQL-Answers are used to represent results of analysis tools in a generalized but accurate way. The following analysis information can currently be represented through AQL-Answers:

- Flows (e.g. taint flows)
- Intents
- Intent-Filter
- Intent-Sinks
- Intent-Sources
- Sources
- Sinks
- Permissions

The structure of AQL-Answers is precisely defined through an XML Schema Definition. Consequently, every answer is represented by an .xml document. Such an .xml document is structured as follows:

```
<answer>
   <intentsources>
      <intentsource>
         ...
      </intentsource>
      ...
   </intentsources>
   <flows>
      <flow>
         ...
      </flow>
      ...
   </flows>
   ...
</answer>
```

Each `flow`, `intent`, `intentfilter`, `intentsink`, `intentsource`, `source`, `sink` and `permission` element can hold arbitrary many `attribute` elements consisting of name-value-pairs to represent additional information as shown in the following.

Example 1: Flows

One taint flow detected by a tool such as FlowDroid can be represented in the following way:

```xml
...
<flow>
    <!-- Flow starts from -->
    <reference type="from">
        <statement>
            <statementfull>$r4 = virtualinvoke $r3.&lt;android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()&gt;()</statementfull>
            <statementgeneric>android.telephony.TelephonyManager: java.lang.String getSimSerialNumber()</statementgeneric>
            <linenumber>26</linenumber>
        </statement>
        <method>&lt;de.foellix.aql.aqlbench.api19.interappstart1.MainActivity: void source()&gt;</method>
        <classname>de.foellix.aql.aqlbench.api19.interappstart1.MainActivity</classname>
        <app>
            <file>/media/sf_share/fix/InterAppStart1.apk</file>
            <hashes>
                <hash type="MD5">2aafeb4bd6e436f66fc06083fda3beb1</hash>
                <hash type="SHA-1">a4619c4448047436e96ec4397dff343e6702c532</hash>
                <hash type="SHA-256">627af4963cbd3b31a0e9c3ef4a029cfb62534689c2dd620c97252ba55c72ac15</hash>
            </hashes>
        </app>
    </reference>

    <!-- Flow ends in -->
    <reference type="to">
        <statement>
            <statementfull>virtualinvoke $r3.&lt;android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,androi
d.app.PendingIntent,android.app.PendingIntent)&gt;("123456789", null, $r2, null, null)</statementfull>
            <statementgeneric>android.telephony.SmsManager: void sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingInt
ent,android.app.PendingIntent)</statementgeneric>
            <linenumber>26</linenumber>
            <parameters>
                <parameter>
                    <type>java.lang.String</type>
                    <value>"123456789"</value>
                </parameter>
                <parameter>
                    <type>java.lang.String</type>
                    <value>null</value>
                </parameter>
                <parameter>
                    <type>java.lang.String</type>
                    <value>$r2</value>
                </parameter>
                <parameter>
                    <type>android.app.PendingIntent</type>
                    <value>null</value>
                </parameter>
                <parameter>
                    <type>android.app.PendingIntent</type>
                    <value>null</value>
                </parameter>
            </parameters>
        </statement>
        <method>&lt;de.foellix.aql.aqlbench.api19.interappend1.MainActivity: void sink()&gt;</method>
        <classname>de.foellix.aql.aqlbench.api19.interappend1.MainActivity</classname>
        <app>
            <file>/media/sf_share/fix/InterAppEnd1.apk</file>
            <hashes>
                <hash type="MD5">fc850d773145bbc65694c58213f6cb6f</hash>
                <hash type="SHA-1">edd9bf89ef2b01be8abf7f6616dd5867c96a79ac</hash>
                <hash type="SHA-256">79dd06fa366c4313f84913b73ce8fe157dd6dab8b0946d1a54ef60b604f2a26d</hash>
            </hashes>
        </app>
    </reference>

    <!-- Attribute showing this is a complete flow -->
    <attributes>
        <attribute>
            <name>complete</name>
            <value>true</value>
        </attribute>
    </attributes>
</flow>
...
```

The statement where the `flow` starts is described by a `reference`. Reference elements are used to describe certain program locations inside an app. This reference's `type` attribute is set to *from* to reflect that the flow starts at the referenced program location. The program location is described by four inner elements, namely `statement`, `method`, `classname` and `app`. For any statement two String representations can be provided. In this example the `statementfull` element refers to the full Jimple statement, whereas the `statementgeneric` element only refers to the program independent part of it. Additionally, each statement has a list of name-value-pairs attached to it in order to represent the parameters of e.g. a function call. `method` refers to the signature of the method which holds the previously described statement. `classname` again refers to the class containing the beforehand described method. Lastly, the `app` element references the app where this class can be found. To do so, the app's

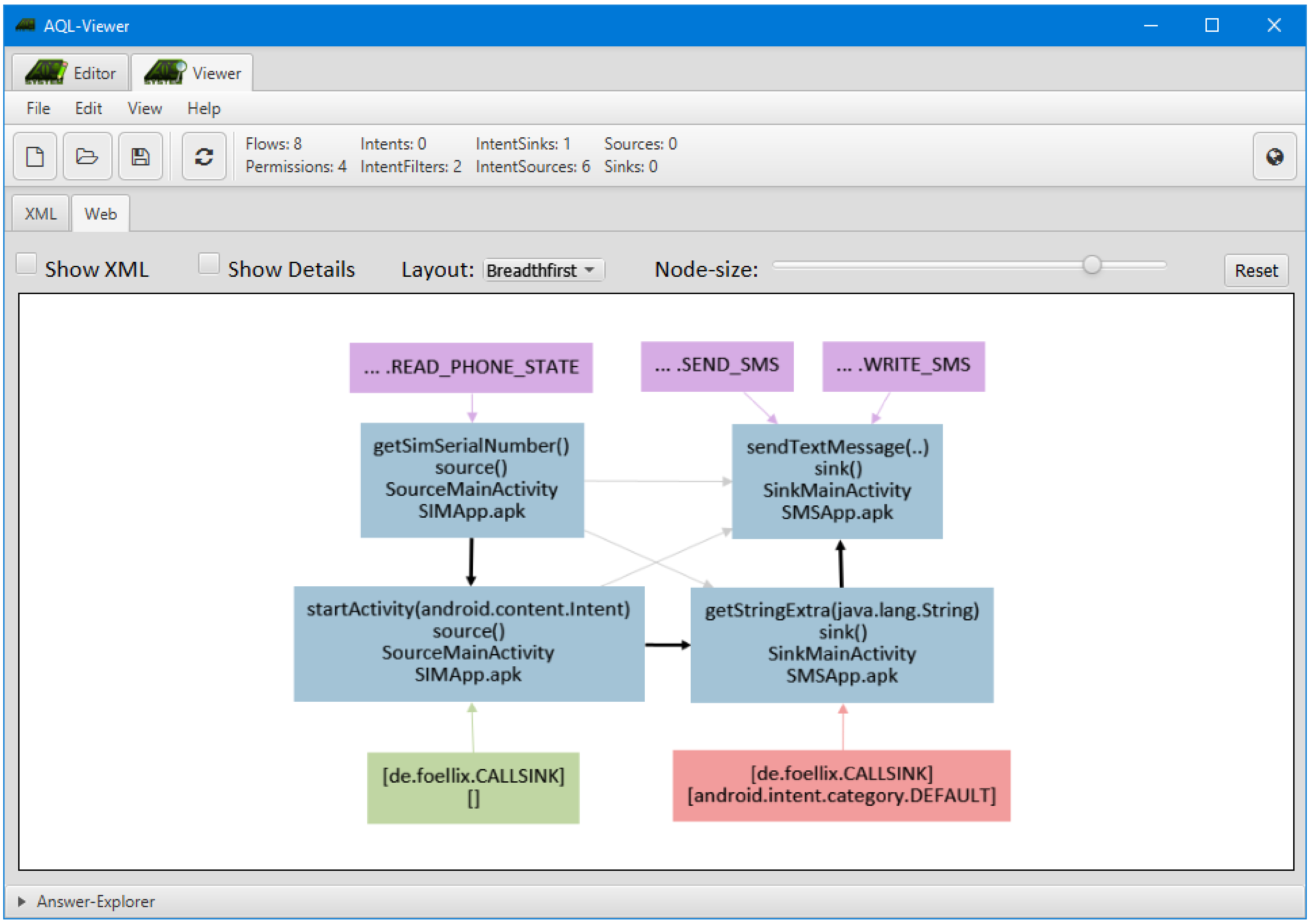
.apk file can be specified along with arbitrary hashes to recognize the app on other systems.

The second `reference` element's `type` attribute is set to *to*, thereby we know that the flow ends in this program location. Its structure is the same. Finally, the flow is fully described.

The *attributes* list at the end of the flow description holds one `attribute`. The name-value-pair (*complete = true*) representing this attribute refers to the fact, that this flow is complete - It leads from a tainted source to a sink.

Example 2: Connected answer

The following figure shows an AQL-Answer in a graphical way. Such graphs can be generated with the AQL-System on the basis of an AQL-Answer .xml file.


AQL-Viewer
Editor
Viewer
File
Edit
View
Help
Flows: 8
Permissions: 4
Intents: 0
IntentFilters: 2
IntentSinks: 1
IntentSources: 6
Sources: 0
Sinks: 0
XML
Web
Show XML
Show Details
Layout: Breadthfirst
Node-size:
Reset
... .READ_PHONE_STATE
... .SEND_SMS
... .WRITE_SMS
getSimSerialNumber() source() SourceMainActivity SIMApp.apk
sendTextMessage(..) sink() SinkMainActivity SMSApp.apk
startActivity(android.content.Intent) source() SourceMainActivity SIMApp.apk
getStringExtra(java.lang.String) sink() SinkMainActivity SMSApp.apk
[de.foellix.CALLSINK] []
[de.foellix.CALLSINK] [android.intent.category.DEFAULT]
Answer-Explorer

The blue nodes and edges represent the different flows. The green and red ones represent the intentsinks and -sources. Lastly, the purple nodes on top represent permissions.
To produce this answer four different analysis tools where asked by means of the AQL:

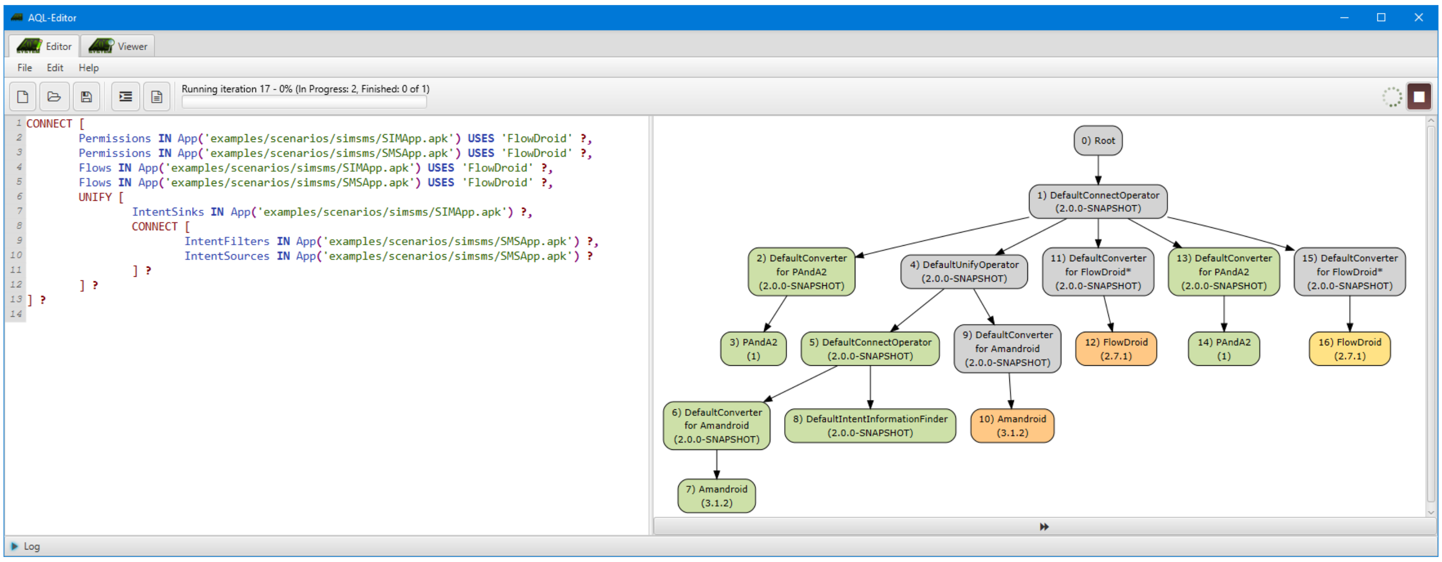
- FlowDroid for Flows inside the SIMApp and SMSApp
  Thereby, among others a flow
  - from `getSimSerialNumber()` to `startActivity(..)` and a flow
  - from `getStringExtra(..)` to `sendTestMessage(..)` could be found.
    In particular, the complete flow
  - from `getSimSerialNumber()` to `sendTextMessage(..)` could **not** be found, yet!
- Amandroid for IntentFilters and IntentSinks in these apps
  Its result especially holds one IntentSink and one IntentFilter described by the action string*de.upb.fpauck.CALLSINK*.
- DefaultIntentInformationFinder (default tool that comes with the AQL-System) for IntentSources which are combined with IntentFilters found by Amandroid through the `CONNECT` operator

- and PAndA² for Permission uses. The result shows that, on the one hand, the*READ_PHONE_STATE* permission is required by the `getSimSerialNumber()` statement and, on the other hand, the `sendTextMessage(..)` statement required the *SEND_SMS* and *WRITE_SMS* permission.

The AQL-Operator `CONNECT` has to be applied to connect the answers. While doing so, the IntentSink is matched to the combined IntentSource, resulting in a new flow from `startActivity(..)` to `getStringExtra(..)` which is the missing piece to finish the puzzle. Now the three flows found can be used to construct the complete flow (from `getSimSerialNumber()` to `sendTextMessage(..)`) in a transitive manner.

AQL-System

The Android App Analysis Query Language - System (AQL-System)


AQL-Editor
Editor
Viewer
File
Edit
Help
Log
0) Root
1) DefaultConnectOperator (2.0.0-SNAPSHOT)
2) DefaultConverter for PAndA2 (2.0.0-SNAPSHOT)
4) DefaultUnifyOperator (2.0.0-SNAPSHOT)
11) DefaultConverter for FlowDroid* (2.0.0-SNAPSHOT)
13) DefaultConverter for PAndA2 (2.0.0-SNAPSHOT)
15) DefaultConverter for FlowDroid* (2.0.0-SNAPSHOT)
3) PAndA2 (1)
5) DefaultConnectOperator (2.0.0-SNAPSHOT)
9) DefaultConverter for Amandroid (2.0.0-SNAPSHOT)
12) FlowDroid (2.7.1)
14) PAndA2 (1)
16) FlowDroid (2.7.1)
6) DefaultConverter for Amandroid (2.0.0-SNAPSHOT)
8) DefaultIntentInformationFinder (2.0.0-SNAPSHOT)
10) Amandroid (3.1.2)
7) Amandroid (3.1.2)

```
Running iteration 17 - 0% (In Progress: 2, Finished: 0 of 1)
```

```
CONNECT [
        Permissions IN App('examples/scenarios/simsms/SIMApp.apk') USES 'FlowDroid' ?,
        Permissions IN App('examples/scenarios/simsms/SMSApp.apk') USES 'FlowDroid' ?,
        Flows IN App('examples/scenarios/simsms/SIMApp.apk') USES 'FlowDroid' ?,
        Flows IN App('examples/scenarios/simsms/SMSApp.apk') USES 'FlowDroid' ?,
        UNIFY [
                IntentSinks IN App('examples/scenarios/simsms/SIMApp.apk') ?,
                CONNECT [
                        IntentFilters IN App('examples/scenarios/simsms/SMSApp.apk') ?,
                        IntentSources IN App('examples/scenarios/simsms/SMSApp.apk') ?
                ] ?
        ] ?
] ?
```

The AQL consists of two main parts, namely AQL-Queries (compositions ofAQL-Questions) and AQL-Answers. The AQL-System takes AQL-Queries as input and outputs AQL-Answers.

Tutorials

- Video tutorials
- Runthrough
- Install, Compile, Develop
- Launch parameters
- Configuration
    - Variables
- (Transformation) Rules
- Query execution
- Add your tools

Changelog

Video tutorials

Videos

https://github.com/FoelliX/AQL-System/wiki/Video_tutorials

Runthrough

Runthrough

The following instructions deal with the installation of the AQL-System. Along with that Amandroid will be installed. Hence, the AQL-System will be setup to use Amandroid only. (The operating system considered is Linux.)

1. Download the latest version of the AQL-System:here
   - Unzip it!
2. Download Amandroid: https://bintray.com/arguslab/maven/argus-saf/3.1.2
   (direct link: https://bintray.com/arguslab/maven/download_file?file_path=com%2Fgithub%2Farguslab%2Fargus-saf_2.12%2F3.1.2%2Fargus-saf_2.12-3.1.2-assembly.jar)
3. Download the `DirectLeak1` app from DroidBench 3.0: https://github.com/secure-software-engineering/DroidBench/raw/develop/apk/AndroidSpecific/DirectLeak1.apk
4. Setup a configuration
   - Create file `config_amandroid.xml` located in the directory of the AQL-System
   - Copy and Paste the following content:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<config>
<androidPlatforms>/path/to/android/platforms/</androidPlatforms>
<androidBuildTools>/path/to/android/buildTools</androidBuildTools>
<maxMemory>8</maxMemory>
<tools>
  <tool name="Amandroid" version="312">
      <priority>1</priority>
      <execute>
         <run>/path/to/Amandroid/aqlRun.sh %APP_APK% %MEMORY%</run>
         <result>/path/to/Amandroid/outputPath/%APP_APK_FILENAME%/result/AppData.txt</result>
         <instances>0</instances>
         <memoryPerInstance>4</memoryPerInstance>
      </execute>
      <path>/path/to/Amandroid</path>
      <questions>IntraAppFlows</questions>
      <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
      <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
  </tool>
</tools>
</config>
```

   - Adjust the configuration:
     - *1:* Adjust the path to your Android SDK's platforms directory (`<androidPlatforms>/path/to/android/platforms/</androidPlatforms>`)
       - *2 (Optional):* The build tools are not required here, still feel free to adjust the respective path as well (`<androidBuildTools>/path/to/android/buildTools</androidBuildTools>`)
     - *3:* Adjust the path for Amandroid (`<path>/path/to/Amandroid</path>`) (The directory should contain the previously downloaded .jar file.)
     - *4:* Use the same path in `<run>` and `<result>`
     - *5:* Adjust the path to flushMemory.sh and killpid.sh to the path of the AQL-System in `<runOnExit>` and `<runOnAbort>`.
     - *6:* Lastly adjust `<maxMemory>` and `<memoryPerInstance>`. The latter has to be less than or equal to the first value. Both values are given in gigabytes. (If sufficient memory is provided, a tool might be executed multiple times in parallel.)
5. Make sure `flushMemory.sh` and `killpid.sh`, located in the AQL-Systems directory, are executeable:

```
chmod u+x flushMemory.sh killpid.sh
```

6. Create launch script

```
cd /path/to/Amandroid
nano aqlRun.sh
```

7. Copy and Paste the following:

```
#!/bin/bash
rm -R outputPath
java -Xmx${2}g -jar argus-saf_2.12-3.1.2-assembly.jar t -o outputPath ${1}
```

8. Save *(Ctrl+o)* and exit *(Ctrl+x)* nano

9. Make the script executable:

```
chmod u+x aqlRun.sh
```

10. Finally, launch the AQL-System:

```
cd /path/to/AQL-System
java -jar AQL-System-1.1.1.jar -config config_amandroid.xml -d detailed -gui
```

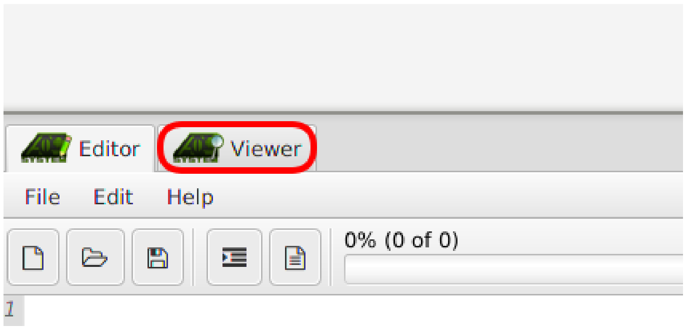
11. Type in the query (do not forget to adjust the contained path `/path/to/DirectLeak1.apk`):

```
Flows IN App('/path/to/DirectLeak1.apk') ?
```

12. Click on *Ask query* (Green play button on the right in the toolbar).

13. Wait for Amandroid to finish its execution.

14. View the AQL-Answer in "Viewer" tab (More information about AQL-Answers can be found: here)



Install, Compile, Develop

1. Install

To simply install the AQL-System you must

- download the current release: here
- and unzip it
- done!

For a *hello world* like tutorial follow the runthrough tutorial.

2. Compile

Requirements:

- **Java 16** or newer (tested with Oracle JDK only)
- **Maven 3.8.1** or newer (required for Java 16)

To compile the AQL-System by yourself follow these steps:

- Clone the repository
- Build the Maven project by:

  ```
  cd /path/to/project/AQL-System
  mvn
  ```

  (Test might not be completely up-to-date, consider skipping: `mvn -DskipTests`)

3. Develop

Include the AQL-System as a library.

- Therefore download the current release: here
- or add it as Maven dependency:

  ```
  <dependency>
    <groupId>de.foellix</groupId>
    <artifactId>AQL-System</artifactId>
    <version>2.0.0</version>
  </dependency>
  ```

  (Version probably must be adapted to the up-to-date one.)

Using the AQL

The following code shows a very simple example. One AQL-Answer is parsed, edited and returned as an Java object.

```
import ...;
import de.foellix.aql.datastructure.Answer;
import de.foellix.aql.datastructure.Flow;
import de.foellix.aql.datastructure.handler.AnswerHandler;

public class AQLUser {
    public Answer use(File aqlAnswerFile) {
        // Parse an AQL-Answer
        Answer answer = AnswerHandler.parseXML(aqlAnswerFile);

        // Create a new flow
        Flow flow = new Flow();
        ...

        // Add it to the answer
        answer.getFlows().getFlow().add(flow);

        return answer;
    }
}
```

Using the AQL-System

Another simple example. An AQL-System is initialized. A query is issued and the AQL-Answer produced upon is returned.

```
import ...;
import de.foellix.aql.datastructure.Answer;
import de.foellix.aql.system.AQLSystem;

public class AQLUser {
    public Answer query(File apkFile) {
        // Initialize aqlSystem
        AQLSystem aqlSystem = new AQLSystem();

        // Execute a query
        return (Answer) aqlSystem.queryAndWait("Flows IN App('" + apkFile.getAbsolutePath() + "') ?").iterator().next();
    }
}
```

Important Classes

The following table hightlights some important classes that may become useful while developing with the AQL-System.

Helpers & Handlers

Class	Capabilities
AQL-System, Options	The AQL-System itself and its Options
AnswerHandler	Parsing and dealing with AQL-Answers
ConfigHandler	Parsing and using configurations
RulesHandler	Parsing and using (transformation) rules
Helper	Contains, for example, all toString() methods for generated classes such as AQL-Answers
HashHelper	For generating hashes
EqualsHelper, EqualsOptions	For equals operations on generated classes such as AQL-Answers

Hooks

Along with version 1.2.0 hooks are introduced. You can execute your own code before and after a task (e.g. analysis tool or preprocessor execution) is completed:

```
import ...;
import de.foellix.aql.Log;
import de.foellix.aql.config.ConfigHandler;
import de.foellix.aql.config.Tool;
import de.foellix.aql.system.AQLSystem;

import de.foellix.aql.system.ITaskHook;
import de.foellix.aql.system.task.Task;

public class HookExample {
    public Collection<Object> query(File appFile) {
        // Initialize aqlSystem
        AQLSystem aqlSystem = new AQLSystem();

        // Apply hook
        Tool awesomeDroid = ConfigHandler.getInstance().getToolByName("AwesomeDroid");
        ITaskHook hook = new AwesomeHook();
        List<ITaskHook> hooks = new ArrayList<>();
        hooks.add(hook);
        aqlSystem.getTaskHooksBefore().getHooks().put(awesomeDroid, hooks);

        // Execute a query
        return aqlSystem.queryAndWait("Flows IN App('" + appFile.getAbsolutePath() + "')");
    }

    private class AwesomeHook implements ITaskHook {
        @Override
        public void execute(Task task) {
            Log.msg("Awesome!", Log.DEBUG);
        }
    }
}
```

Launch parameters

# Launch Parameters The AQL-System can be launched with the parameters mentioned in the table below.

Parameter	Meaning
`-help`, `-h`, `-?`, `-man`, `-manpage`	Outputs a very brief manual, which contains a list of all available parameters.
`-query "X"`, `-q "X"`	This parameter is used to assign an AQL-Query. `X` refers to this query.
`-config "X"`, `-cfg "X"`, `-c "X"`	By default the `config.xml` file in the tool's directory is used as configuration. With this parameter a different configuration file can be chosen. `X` has to reference the path to and the configuration file itself. `X` can also be an online file or combination of a link and login credentials associated with an AQL-WebService. Furthermore, this parameter can be given multiple times - configurations will be merged into the last one given in this case.)
`-rules "X"`	By default the `rules.xml` file in the tool's directory is used as (transformation) rules / strategy configuration. With this parameter a different file can be chosen. `X` has to reference the path to and the rules file itself.
`-timeout "X"s/m/h "Y"`, `-t "X"s/m/h "Y"`	With this parameter the maximum execution time of each tool can be set. If it expires the tool's execution is canceled. `X` refers to this time in seconds (e.g. 10s), minutes or hours. Y is optional and defines how to handle timeout values given in configuration files. Y can have the following values: 1. "min" - minimal timeout used, 2. "max" - maximal timeout used and 3. "override" use X always.
`-nr`, `-noRetry`	Disables retrying with next-highest-priority tool, if the tool with highest priority fails.
`-configwizard`, `-cw`	If this parameter is applied the Config Wizard will be started at the beginning.
`-output "X"`, `-out "X"`, `-o "X"`	The answer to a query is automatically saved in the `answers` directory. This parameter can be used to store it in a second file. `X` has to define this file by path and filename.
`-debug "X"`, `-d "X"`	The output generated during the execution of this tool can be set to different levels. X may be set to `none`, `important`, `error`, `special`, `warning`, `normal`, `debug`, `detailed`, `verbose`, `all` (ascending precision from left to right). Additionally it can be set to `short`, the output will then be equal to `normal` but shorter at some points. By default it is set to `normal`.
`-df "X"`, `-dtf "X"`, `-debugToFile "X"`	Sets the log level (`X`) that should be logged to file (into `log.txt`). The default value is `important`.
`-backup`, `-b`	When this launch parameter is provided, the current storage of the AQL-System is backed up on start.
`-reset`, `-r`	By this parameter the storage of the AQL-System is resetted on start. Whenever a backup is scheduled as well, it will be generated before the reset.
`-gui`	If this or no parameters at all are provided the graphical user interface is started. It allows to formulate queries and display answers in a handy way. (Additionally use -ns, -noSplash to skip the splashscreen.)
`-dg`, `-draw`, `-drawGraph`	Decides whether to draw a graph (on a GUI) representing the query and its questions.
`-v`, `-view`	The determined AQL-Answer will be shown in the GUI after executing the specified query.

Configuration

Configuration

Any AQL-System has to be configured via an *.xml* file. Its structure is described by this XML Schema Definition file. Such a configuration defines a few environmental properties and most importantly which analysis tools, preprocessors, operators and converters are available in addition to the default tools, operators and converters. Any of these is represented by a `<tool>` element inside the configuration file.

By default any AQL-System tries to use a `config.xml` file (which is located in the same directory as the system's `.jar` file) as configuration. The launch parameter `-c` can be used to define another config that should be used instead (see Launch parameters). Parameter `-c` allows three options (Parameter `-c` can be given multiple times - configurations will be merged into the last one given in this case.):

- *Local file:* `-c /path/to/config.xml`
- *Online file:* `-c http://FoelliX.de/path/to/config.xml`
- *AQL-WebService*: `-c "http://FoelliX.de/AQL-WebService/config, username, password"` (see AQL-WebService)

There exists two possibilities to create or edit a configuration:

- Edit the `.xml` file directly
- Use the Configuration Wizard

Option 1: Edit the .xml file directly

The following code shows a basic, shortened version of a configuration:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<config>
    <!-- Environment -->
    <androidBuildTools>path/to/android/build-tools</androidBuildTools>
    <androidPlatforms>path/to/android/platforms</androidPlatforms>
    <maxMemory>6</maxMemory>

    <!-- Tools -->
    <tools>...</tools>
    <preprocessors>...</preprocessors>
    <operators>...</operators>
    <converters>...</converters>
</config>
```

- Inside the `<androidBuildTools>` tag the path to the android build tools has to be specified.
- Inside the `<androidPlatforms>` tag the path to the android platform files has to be specified.
- The memory element tells the AQL-System how much memory shall be used at most.
- `<tools>`, `<preprocessors>`, `<operators>` and `<converters>` each contain a list of `<tool>` elements. Each `<tool>` element describes one analysis tool, preprocessor, operator or converter, respectively. In the following the differences and commonalities of these four are described along with one example for each type.

Analysis tools

```
<tool name="AwesomeDroid" version="1.3.3.7" external="false">
   <priority>1</priority>
   <priority feature="TEST">2</priority>

   <execute>
      <run>/path/to/AwesomeDroid/run.sh %MEMORY% %ANDROID_PLATFORMS% %APP_APK%</run>
      <result>/path/to/AwesomeDroid/results/%APP_APK_FILENAME%_result.txt</result>
      <instances>0</instances>
      <memoryPerInstance>4</memoryPerInstance>
   </execute>

   <path>/path/to/AwesomeDroid</path>
   <questions>IntraAppFlow</questions>

   <runOnEntry>/path/to/AQL-System/start.sh</runOnEntry>
   <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
   <runOnSuccess>/path/to/AwesomeDroid/success.sh</runOnSuccess>
   <runOnFail>/path/to/AwesomeDroid/fail.sh</runOnFail>
   <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
</tool>
```

- Any tool is identified by the values assigned to the two attributes `name` and `version`. In this example the tool's name is `AwesomeDroid` and its version is `1.3.3.7`.
- `<priority>`: If there are two or more tools available which are capable of answering the same AQL-Questions the priority decides which tool is executed.
  - There may be multiple `<priority>` elements, however, only one without a feature attribute. In the example AwesomeDroid has a priority of `1`, which becomes `1 + 2 = 3` if the associated AQL-Question assigns the feature `TEST`.

How to run any tool is specified through the `<execute>` element. Tools can generally be of two types:

1. Internal Tools

Tools that are execute on the same system as the AQL-System. We also refer to such tools as *internal tools*. All tools of this type must assign the `external="false"` attribute to the respective `<tool>` element (see example above). Any internal tool's `<execute>` element must define the following sub-elements:

- `<run>`: The run tag contains a command line command that can be used to run the associated tool (e.g. a command that runs a script - see example above).
- `<result>`: This describes where the result can be found once a tool finishes successfully. *(A *-symbol can be used inside this tag to reference an arbitrary substring.)*
- `<instances>`: This element defines how often the associated tool can be executed at the same time. *(0 means infinite times)*
- `<memoryPerInstance>`: This tag defines how much memory (in GB) is required and provided to each instance of the associated tool.

2. External tools

External tools (`external="true"`) will be executed remotely by communicating with an AQL-WebService (see AQL-WebService). For this type of tools the following sub-elements must be defined:

- `<url>`: The URL of the AQL-WebService. For example: `http://FoelliX.de:8080/AQL-WebService/askAQL`
- `<username>` & `<password>`: The credentials required to access the targeted webservice.

For both types a `<path>` must be defined. It describes a path to a directory in which the respective tool is going to be executed.

In the example above five variables are used to define the tags mentioned above:

Variable	Meaning
%APP_APK%	The .apk file referenced in an AQL-Question
%APP_APK_FILENAME%	The filename of the .apk file without path and ending
%ANDROID_PLATFORMS%	The Android platforms folder (Specified through)

Variable	Meaning
%MEMORY%	The memory available to an instance of a tool (Specified through)
%PID%	The tools process ID during execution

(A full list of all available variables can be foundhere)

- `<questions>`: The content of this tag describes which AQL-Questions can be answered with the associated tool. The following options are available (Exemplary associated AQL-Questions can be found in the brackets behind each option):
    - Arguments (`Arguments IN App('A.apk') ?`)
    - Permissions (`Permissions IN App('A.apk') ?`)
    - Sources (`Sources IN App('A.apk') ?`)
    - Sinks (`Sinks IN App('A.apk') ?`)
    - Intents (`Intents IN App('A.apk') ?`)
    - IntentFilters (`IntentFilters IN App('A.apk') ?`)
    - IntentSources (`IntentSources IN App('A.apk') ?`)
    - IntentSinks (`IntentSinks IN App('A.apk') ?`)
    - IntraAppFlows (`Flows IN App('A.apk') ?`)
    - InterAppFlows (`Flows FROM App('A.apk') TO App('B.apk') ?`)
    - Slice (`Slice FROM Statement('from()')->Method('a()')->Class('A')->App('A.apk') TO Statement('to()')->Method('b()')->Class('B')->App('A.apk') !`)

There are five more elements which optionally can be specified, namely `<runOnEntry>`, `<runOnExit>`, `<runOnSuccess>`, `<runOnFail>` and `<runOnAbort>`. Each refers to a command or a script which will be executed on certain tool events.

- `<runOnEntry>` is run before a tool is executed.
- `<runOnExit>` is always run after tool execution or abortion.
- `<runOnSuccess>` and `<runOnFail>` get executed depending on whether the tool has finished successfully or not.
- `<runOnAbort>` is run if the tool is aborted.

(Only the variables which are always available can be used inside these tags.)

Preprocessors

```xml
<tool name="AwesomePreprocessor" version="1.3.3.8" external="false">
   <priority>1</priority>

   <execute>
      <run>/path/to/AwesomePreprocessor/run.sh %APP_APK%</run>
      <result>/path/to/AwesomePreprocessor/results/%APP_APK_FILENAME%_preprocessed.apk</result>
      <instances>0</instances>
      <memoryPerInstance>4</memoryPerInstance>
   </execute>

   <path>/path/to/AwesomePreprocessor</path>
   <questions>TEST</questions>

   <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
   <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
</tool>
```

Preprocessors are specified the same way as analysis tools with one exception: The `<questions>` element now holds a list of keywords, separated by `,`, that is assigned to the associated preprocessor. In the above example only one keyword is assigned (`TEST`).

Operators

```
<tool name="AwesomeOperator" version="1.3.3.9" external="false">
   <priority>1</priority>

   <execute>
      <run>/path/to/AwesomeOperator/run.sh %ANSWERS%</run>
      <result>/path/to/AwesomeOperator/results/%ANSWERSHASH%.xml</result>
      <instances>1</instances>
      <memoryPerInstance>4</memoryPerInstance>
   </execute>

   <path>/path/to/AwesomeOperator</path>
   <questions>CONNECT(*)</questions>

   <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
   <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
</tool>
```

Operators are specified the same way as analysis tools with two exceptions:

- Different variables, for instance the two appearing in the example above `%ANSWERS%`, `%ANSWERSHASH%`), can be used (see Variables for more information).
- The `<questions>` element refers to the operators name and specifies its number of parameters
  In the example `CONNECT(*)` tells us that the default `CONNECT` operator gets overwritten by an operator which takes arbitrary many (`*`) AQL-Answers as input.

Converters

```
<tool name="AwesomeDroidConverter" version="1.3.3.7" external="false">
   <priority>1</priority>

   <execute>
      <run>/path/to/AwesomeDroidConverter/run.sh %RESULT_FILE% results/%APP_APK_FILENAME%.xml</run>
      <result>/path/to/AwesomeDroidConverter/results/%APP_APK_FILENAME%.xml</result>
      <instances>0</instances>
      <memoryPerInstance>4</memoryPerInstance>
   </execute>

   <path>/path/to/AwesomeDroidConverter</path>
   <questions>AwesomeDroid</questions>
</tool>
```

Converters again are specified the same way as analysis tools with two exceptions:

- The additional variable `%RESULT_FILE%` can be used (see Variables for more information).
- The `<questions>` element in this case refers to the analysis tools (separated by `,`) associated with this converter. (In the example only `AwesomeDroid` is associated.)

Complete Example

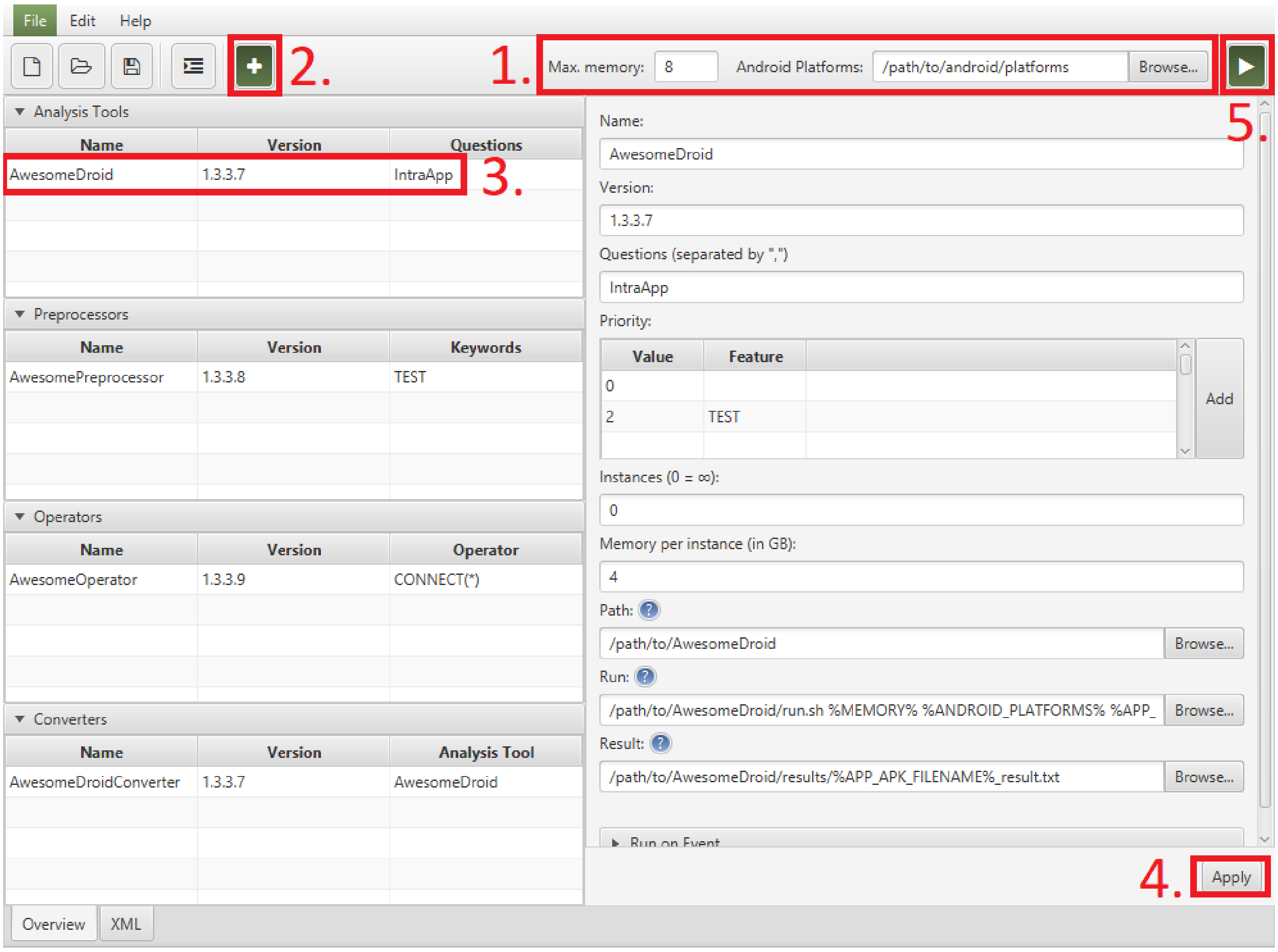
```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<config>
   <androidPlatforms>path/to/android/platforms</androidPlatforms>
   <androidBuildTools>path/to/android/build-tools</androidBuildTools>
   <maxMemory>8</maxMemory>
   <tools>
      <tool name="AwesomeDroid" version="1.3.3.7" external="false">
         <priority>1</priority>
         <priority feature="TEST">2</priority>
         <execute>
            <run>/path/to/AwesomeDroid/run.sh %MEMORY% %ANDROID_PLATFORMS% %APP_APK%</run>
            <result>/path/to/AwesomeDroid/results/%APP_APK_FILENAME%_result.txt</result>
            <instances>0</instances>
            <memoryPerInstance>4</memoryPerInstance>
         </execute>
         <path>/path/to/AwesomeDroid</path>
         <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
         <runOnSuccess>/path/to/AwesomeDroid/success.sh</runOnSuccess>
         <runOnFail>/path/to/AwesomeDroid/fail.sh</runOnFail>
         <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
         <questions>IntraApp</questions>
      </tool>
   </tools>
   <preprocessors>
      <tool name="AwesomePreprocessor" version="1.3.3.8" external="false">
         <priority>1</priority>
         <execute>
            <run>/path/to/AwesomePreprocessor/run.sh %APP_APK%</run>
            <result>/path/to/AwesomePreprocessor/results/%APP_APK_FILENAME%_preprocessed.apk</result>
            <instances>0</instances>
            <memoryPerInstance>4</memoryPerInstance>
         </execute>
         <path>/path/to/AwesomePreprocessor</path>
         <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
         <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
         <questions>TEST</questions>
      </tool>
   </preprocessors>
   <operators>
      <tool name="AwesomeOperator" version="1.3.3.9" external="false">
         <priority>1</priority>
         <execute>
            <run>/path/to/AwesomeOperator/run.sh %ANSWERS%</run>
            <result>/path/to/AwesomeOperator/results/%ANSWERSHASH%.xml</result>
            <instances>1</instances>
            <memoryPerInstance>4</memoryPerInstance>
         </execute>
         <path>/path/to/AwesomeOperator</path>
         <runOnExit>/path/to/AQL-System/flushMemory.sh</runOnExit>
         <runOnAbort>/path/to/AQL-System/killpid.sh %PID%</runOnAbort>
         <questions>CONNECT(*)</questions>
      </tool>
   </operators>
   <converters>
      <tool name="AwesomeDroidConverter" version="1.3.3.7" external="false">
         <priority>1</priority>
         <execute>
            <run>/path/to/AwesomeDroidConverter/run.sh %RESULT_FILE% results/%APP_APK_FILENAME%.xml</run>
            <result>/path/to/AwesomeDroidConverter/results/%APP_APK_FILENAME%.xml</result>
            <instances>0</instances>
            <memoryPerInstance>4</memoryPerInstance>
         </execute>
         <path>/path/to/AwesomeDroidConverter</path>
         <questions>AwesomeDroid</questions>
      </tool>
   </converters>
</config>
```

Option 2: Use the Config Wizard

You find the *ConfigWizard* in the *Help* menu of the GUI.
Alternatively you can launch it as follows:

```
java -jar AQL-System-*.jar -cw
```

The screenshot below shows the Config Wizard. All elements explained before can easily be edited here as well.



Max. memory: 8

Android Platforms: /path/to/android/platforms

Analysis Tools

Name	Version	Questions
AwesomeDroid	1.3.3.7	IntraApp

Preprocessors

Name	Version	Keywords
AwesomePreprocessor	1.3.3.8	TEST

Operators

Name	Version	Operator
AwesomeOperator	1.3.3.9	CONNECT(*)

Converters

Name	Version	Analysis Tool
AwesomeDroidConverter	1.3.3.7	AwesomeDroid

Name: AwesomeDroid

Version: 1.3.3.7

Questions (separated by ","): IntraApp

Priority:

Value	Feature
0	
2	TEST

Instances (0 = ∞): 0

Memory per instance (in GB): 4

Path: /path/to/AwesomeDroid

Run: /path/to/AwesomeDroid/run.sh %MEMORY% %ANDROID_PLATFORMS% %APP_

Result: /path/to/AwesomeDroid/results/%APP_APK_FILENAME%_result.txt

- The environmental properties can be defined at **1.**
- New tools of any kind can be added at **2.**
- To edit a tool:
  - Select it, for example by clicking on **3.**
  - Edit its properties on the right hand side
  - Apply the changed by clicking at **4.**
- To continue with the configuration you have set up click on **5.**

Variables

Configuration: Variables

Variables can be used while defining `<run>`, `<result>` or `<path>` sub-elements of a `<tool>` element inside configuration files. They will be replaced with their actual values once these values are available and requested.

Default Variables

Which variables can be used can be seen by the lists contained in the respective `TaskInfo` classes:

Context	`TaskInfo` **class**
Always available	TaskInfo.java
Preprocessors	PreprocessorTaskInfo.java
- Analysis tools	ToolTaskInfo.java
- Converters	ConverterTaskInfo.java
Operators	OperatorTaskInfo.java
- Filter-Operators	FilterOperatorTaskInfo.java

Custom Variables

The list of available variables can be extended by `WITH` definitions inside AQL-Questions. For example, if the question `Flows IN App('A.apk') WITH 'SourcesAndSinks' = 'SourcesAndSinks.txt' ?` is asked, the variable `%SourcesAndSinks%` becomes available. It will be replaced with `SourcesAndSinks.txt` once the content of a tag using it is resolved.

File Variables

Variables can have arbitrary names, however, if they follow the naming convention `%FILE_*%` they will be treated as files. Variables will also be treated as files if they reference an existing file and their value has the following format: `x.y` (where `x`, `y` only include `Aa0-Zz9`, but `x` may also hold path separators as well as `-` or `_` and `y` must have a length of 1-5). By default variables are treated like strings.

Variables as Priority Tiebreaker

If two tools have the same priority to answer a question, the following is done to break the tie:

- Internal tools are preferred over external tools
- If one of two internal tools uses more variables (denotes more in its `<run>`-tag) than this one is selected.
  - On another tie, the one which requires less custom variables is selected.
  - Again on another tie, the one that uses more default variables is selected.
- In any other case the first (mentioned earlier in the respective configuration file) tool is selected.

(Transformation) Rules

(Transformation) Rules

Rules were introduced along with BREW (1.2.0). With AQL-System (2.0.0) they got improved and moved into the AQL-System. The XML Schema Definition file defines how rules can be structured. Any `rules.xml` file adhering to this XSD may hold arbitrarily many rules to implement various analysis strategies. Two different types of rules may be defined - both types are introduced below.

Type 1: Simple Rules

Two types of rules can be specified. The so-called simple rules are defined by only one element `<query>`). All variables that can be used in queries are also available for simple rules (see Variables). In addition, the variables listed in the table below can be used:

Variable	Meaning
%QUERY%	The original query before applying the rule without question mark, if the original query ends with a question mark
%FILE_i%	File number i (i in [1, n]) from the original query
%FEATURE_i%	Feature number i (i in [1, n]) from the original query
%FEATURES%	All features from the original query

Example: Let us consider the following query `Flows IN App('AwesomeApp.apk') FEATURING 'Awesome' ?`.
With the rule-set below in place, it gets transformed to `FILTER [ Flows IN App('AwesomeApp.apk') FEATURING 'Awesome' ? ]` since only the rule with the highest priority is applied.

```
<rules>
   <rule name="SimpleRule1">
      <priority>1</priority>
      <query>UNIFY [ %QUERY% ?, Permissions IN App('%FILE_1%') ? ]</query>
   </rule>
   <rule name="SimpleRule2">
      <priority feature="Awesome">2</priority>
      <query>FILTER [ %QUERY% ? ]</query>
   </rule>
</rules>
```

The first rule included is always applied (see attribute `always="true"`) independently of the features mentioned in the query. However, since its priority is only `1` the sencond rule gets applied with a priority of `2` for this query.

Type 2: Input-Output Rules

Input-output rules are defined by only two elements: `<inputQuery>`, `<outputQuery>`.

Example: Let us consider the rule-set below:

```
<rule name="InOutRule1">
   <priority>0</priority>
   <priority feature="Combiner">1</priority>
   <inputQuery>Flows FROM App('%FILE_1%') TO App('%FILE_2%') *3*</inputQuery>
   <outputQuery>Flows IN App('%FILE_1%, %FILE_2%' | 'COMBINE')*3*</outputQuery>
</rule>
```

With this set of rules the query `Flows FROM App('A.apk') TO App('B.apk') FEATURING 'Combiner' ?` would be transformed into `Flows IN App('A.apk, B.apk' | 'COMBINE') ?`.

Further Examples

More examples can be found here:
examples/rules.xml

The rules defined there come into play when executing the following Test:

Query execution

Query Execution

An AQL-Query can be executed using the AQL-System via

- a command line interface
  OR
- a graphical user interface

Anyway it has to be configured first (see theconfiguration tutorial).

Command Line Interface (CLI)

To execute a query via command line, launch the AQL-System `java -jar AQL-System-2.0.0.jar` with the `-query` parameter and provide a query. The following example issues a query that asks for Flows inside one app (`A.apk`):

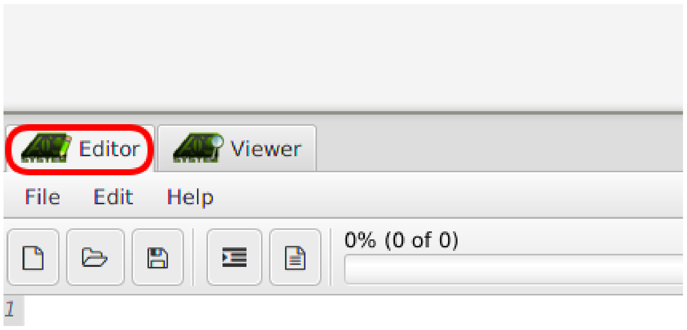
```
java -jar /path/to/AQL-System-2.0.0.jar -query ”Flows IN App('A.apk') ?”
```

Graphical User Interface (GUI)

- To execute a query via the GUI, launch it first:

  ```
  java -jar /path/to/AQL-System-2.0.0.jar -gui
  ```

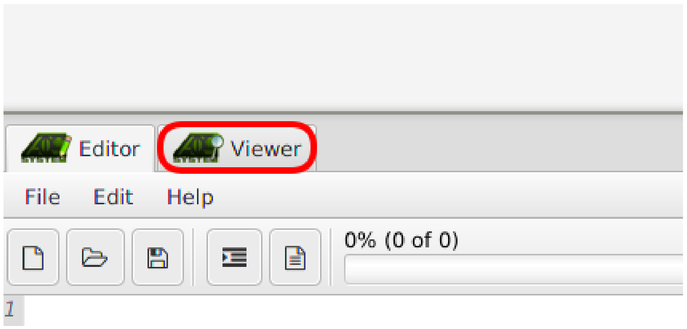
- Then switch to the editor tab:



- Type in an AQL-Query inside the editor window:

  ```
  Flows IN App('A.apk') ?
  ```

- Click "Run" *(Green play button at the right in the toolbar)*
- Wait for the AQL-System to compute the answer
- To view AQL answer, switch to viewer tab



- AQL-Answers are provided in two views:
  - textual view: Allows to view and edit the .xml file representing the AQL-Answer.

- web view: Shows the flows, permissions, intent-sources and -sinks in a single graph.

Add your tools

Add tools

In order to add a new analysis tool, preprocessor, operator or converter to the AQL-System, its configuration has to be adapted (See the configuration tutorial). Adding an analysis tool may require a new converter to be added as well. To do so, three options are available:

1. Existing Converter

If one of the converters implemented in the AQL-System fits your needs. You just have to assign the associated toolname for the analysis tool in the configuration. Currently the following converters are available:

- Amandroid
- DIALDroid *(not supported from version 2.0.0 onwards - If you plan to use this converter, please adapt `data/converter/dialdroid_config.properties`)*
- DidFail
- DroidSafe
- FlowDroid
- IC3
- IccTA
- PAndA^2

(the provided tool's version may decide which converter exactly is used. For example, there are two FlowDroid converters available one for FlowDroid >2.5.0 and one for earlier versions.)

2. Add new converter (internal)

Implement your own converter and compile the AQL-System on your own (See the install & compile tutorial).

1. Include your converter by implementing the interface `IConverter`
2. Add it to the `ConverterRegistry` inside the `de.foellix.aql.converter` package by adding the following line *(after Line 41 in ConverterRegistry.java)*:

   ```
   this.converters.add(
   new Identifier(
   new DefaultConverter("AwesomeDroid", AwesomeDroidConverter.class)
   , "AwesomeDroid")
   );
   ```

   (In this example `AwesomeDroid` refers to the toolname of the tool you want to add, which is defined in the configuration. The associated converter is `AwesomeDroidConverter`.)

3. Add new converter (external)

1. Develop your own converter...
   - taking the result file of the tool you want to add as input
   - and generating an AQL-Answer as output.
2. Specify this converter in the configuration (See the configuration tutorial).
   - Make sure to assign the toolname of all tools, for which the converter should be used, in `<questions>`. (e.g. `<questions>AwesomeDroid1, AwesomeDroid2</questions>`)