# GATOR Documentation

PRESTO

August 21, 2018

## 1 Overview

GATOR is a Pro**g**ram **A**nalysis **T**oolkit F**o**r And**r**oid. It requires a Unix-like operating system to run, and has been tested on Ubuntu 16.04 and Mac OS X 10.10. The toolkit takes as input either Java bytecode or APK files, which are processed with the Soot program analysis framework (`http://www.sable.mcgill.ca/soot/`). GATOR's analyses are built on top of Soot.

This release (version 3.5) includes the source code for the static analyses described in our CGO'14 [1], ICSE'15 [3], ASE'15 [4] and CC'16 [2] papers:

- GUI structural analysis [CGO'14] with extensions and modifications
- Callback control flow-analysis [ICSE'15] with minor extensions
- Analysis for constructing the window transition graph (WTG) [ASE'15] with minor extensions
- The analysis from [ICSE'15] is included as a building block of WTG construction and is not intended for independent use
- The analysis from [CC'16] is included as a client based on the WTG
- The Android programs used in the experiments for these papers are also included

And compared to the last release (version 3.4), it adds following features:

- Bug fixes and performance enhancement of GATOR
- Build system is changed to Gradle
- GUIHierarchyPrinterClient will print out widget's title
- Support of Android Eclipse projects is deprecated

## 1.1 Prerequisite Setup

### 1.1.1 JDK

JDK 1.7+ is required to run GATOR. Please refer to `http://www.oracle.com/technetwork/java` for details of how to obtain a copy of the JDK.

## 1.2 Python

Python 3 is required to run GATOR. Please refer to `https://www.python.org/downloads/` for details of how to obtain a copy of Python 3 runtime.

### 1.2.1 Android SDK

Android SDK is required to run GATOR. It can be downloaded from `http://developer.android.com/sdk`. After the SDK is installed, support files for individual API levels should be installed as well. For example, if you want to analyze an Android application developed for API level 17, support files for android-4.2.2 must be installed. To do that, you can use the Android SDK manager, whose documentation can be found at `http://developer.android.com/tools/help/sdk-manager.html`. In order to run GATOR on example apps included in this package, at least following API levels and Google APIs should be installed:

- android-8
- android-10
- android-14
- android-15
- android-16
- android-17

Note: Some API levels and their Google APIs are only visible after clicking "obsolete" in Android SDK Manager.

# 2 Usage

## 2.1 Build Gator

Before running GATOR analysis on applications the GATOR needs to be built. There are two ways to build GATOR: (1) you can import the project rooted at gator directory into IntelliJ IDE and build it; or (2) you can use the `gator` script under the gator directory to compile the project, just run following command under the gator directory:

```
./gator b
```

## 2.2 Use the Analyses

Before running any analyses using GATOR, there are 2 environment variables need to be defined. The `GatorRoot` should be assigned the path which contains the `AndroidBench` and `gator` directories. The `ANDROID_SDK` should be assigned the path to the Android SDK. Under `bash`, this can be done using following command:

```
export GatorRoot=PATH_TO_ROOT_DIRECTORY_OF_GATOR
export ANDROID_SDK=PATH_TO_ANDROID_SDK
```

### 2.2.1 Perform Analysis on Demo Applications

If this is the first time you use GATOR, we provide a script `AndroidBench/guiAnalysis.py`, which allows you to run the GATOR on applications we used in the [CGO'14], [ICSE'15] and [CC'16] paper.

You should change the current working directory to the `$GatorRoot/AndroidBench` using

```
cd $GatorRoot/AndroidBench
```

There are several options to run `guiAnalysis.py` script. The easiest one is use:

```
./guiAnalysis.py runAll
```

It will perform analysis on applications we used in [CGO'14] and [CC'16] with default clients. If you only want to perform analysis on applications in [CGO'14], you can replace the option "`runAll`" with "`runAllCGO`". If you only want to perform analysis on applications in [CC'16], you can replace the option with "`runAllEnergy`". It is also possible to perform analysis on a single application. e.g. To perform analysis on `apv`, you can use:

```
./guiAnalysis.py apv
```

The applications available are : `apv`, `astrid`, `barcodescanner`, `beem`, `connectbot`, `fbreader`, `k9`, `keepassdroid`, `mileage`, `mytracks`, `notepad`, `npr`, `openmanager`, `opensudoku`, `sipdroid`, supergenpass, tippytipper, vlc, vudroid, xbmc, osmdroid, osmdroid-fixed, recycle-locator, recycle-locator-fixed, sofia, sofia-fixed, ushahidi, ushahidi-fixed, droidar, droidar-fixed, speedometer, heregps, whereami, locdemo, wigle.

If you need to change the analysis client used during the analysis, you need to modify the configuration information in the $GatorRoot/AndroidBench/cgo.json and $GatorRoot/AndroidBench/cc16.json.

Note on the CC'16 analysis results for ushahidi and ushahidi-fixed: the detected energy defects reported by GATOR are for an activity named LocationMap. This activity does have an energy defect if it can be opened. However, the activity is not possible to open by the user; therefore, we removed it from the published paper.

### 2.2.2 Using new Python script gator

Besides the AndroidBench/guiAnalysis.sh, we provide another Python script gator under the gator directory to initiate GATOR to perform analysis on applications. From this GATOR release, the only format of the application we supported is the applications' APK files. To run GATOR on an application, the basic options for this script is:

```
./gator a [-v] [-g] [--api API_LEVEL] -p PATH_TO_APK
                        [GATOR_PARAM] [-client] [GATOR_CLIENT]
                        [-clientParam] [GATOR_CLIENT_PARAM]
```

For optional options, option '-v' enables verbose mode of GATOR, which will increase the details of logs printed to the stdout. Option '-g' will allow GATOR to load Google Play Service libraries if they exist under ANDROID_SDK directory. Some old applications requires this option. Option '–api' will override the API level information sent to GATOR. By default, GATOR will use the application's target API level. But we have see some cases that some applications use APIs from API levels higher than their target API levels, causing issues with GATOR.

If we want to perform analysis on an apk located at /tmp/example.apk using WTGDemoClient, we can use following command:

```
./gator a -p /tmp/example.apk -client WTGDemoClient
```

If we want to perform analysis on the same apk using EnergyAnalysisClient, we can use following command:

```
./gator a -p /tmp/example.apk -client EnergyAnalysisClient
                                            -clientParam WTPK5
```

Please note, for some obfustcated apps, the apktool, which we used to extract the apk package, may failed to decode correct tag names in the layout xml files. Unless apktool fix this issue, Gator may crash when performing analysis on these apps.

#### 2.2.3 GATOR Option Parameters

In GATOR, we provide several options to control the way GATOR analyze the application. There are two categories of options, one is `PARAM`, the other is `CLIENT_PARAM`. Currently we provide 3 `PARAM` options, which are `-worker NUM_OF_THREAD`, `-enableSetTextAnalysis` and `-enableStringAnalysis`. The `-worker NUM_OF_THREAD` is used to define the maximum number of threads GATOR should use. In default configuration, GATOR will analyze the application using 16 threads. However, in rare cases, it may experience concurrency issues as some of part of Soot framework is not thread-safe. In this case, you can put `-worker 1` in the `PARAM`. The other 2 options, the `-enableSetTextAnalysis` and `-enableStringAnalysis` are used to enable the new string analysis feature of GATOR, which allows the `GUIAnalysisClient` to include the text attribute of each widget in the output layout xml files. By default, GATOR only process the text defined in layout xml files. By using `-enableSetTextAnalysis` option, calls to `view.setText(String)` and `view.setText(int)` will be processed by GATOR. The other option, `-enableStringAnalysis` will allow GATOR to process Java `StringBuilder` related options such as string concatenation. Both options will increase the running time of GATOR so they are not enabled by default.

The `CLIENT_PARAM`, on the other hand, can be used to transfer parameters to the analysis client. For example, for applications used in CC'16[2] paper, we use the option `-clientParam WTPK5` for the `EnergyAnalysisClient` to define the maximum length of WTG path it should generate. If you want to change this limit to 3, you can replace this option to `-clientParam WTPK3`.

If you define your own analysis client, in your source code, you can access all `CLIENT_PARAM` by accessing following global variable:

```
Set<String> presto.android.Configs.clientParams;
```

## 3 Customize GUIAnalysisClient

In this section, we will show the way to create a customize GUIAnalysisClient from scratch.

### 3.1 GUIAnalysisClient

In order to implement a customized `GUIAnalysisClient`, user needs to add his own class which implements the `GUIAnalysisClient` interface in `presto.android.gui.clients` package. The declaration of `GUIAnalysisClient` interface is:

```
public interface GUIAnalysisClient {
  public void run(GUIAnalysisOutput output);
}
```

When the GUI analysis of GATOR is finished, if the user specified the name of user implemented `GUIAnalysisClient` in the GATOR options mentioned in the previous section, the `run` method in this interface would be called. The parameter `output` of the `run` method provides the results from the GUI analysis ([CGO'14]), which can be further used to build the GUI Hierarchy or the Window Transition Graph.

### 3.2 Build the GUI Hierarchy

We provide the `GUIHierarchyPrinterClient` for GUI hierarchy print out purposes. However, if you would like to access the GUI hierarchy of the application within your own `GUIAnalysisClient` programmatically, there are a few APIs to do that.

A basic example that print the GUI hierarchy to stdout is like this:

```
public class YourOwnClient implements GUIAnalysisClient {
  @Override
  public void run(GUIAnalysisOutput output) {
    GUIHierarchy guiHier = new StaticGUIHierarchy(output);
    List<GUIHierarchy.Activity> activities = guiHier.activities;
    for (GUIHierarchy.Activity act : activities) {
      dumpWindow(act, 0);
    }
  }

  private String genIndent(int indent) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < indent; i++)
      sb.append(" ");
    return sb.toString();
  }

  private void dumpWindow(GUIHierarchy.ViewContainer w, int indent) {
    if (w instanceof GUIHierarchy.Window) {
      GUIHierarchy.Window win = (GUIHierarchy.Window) w;
      Logger.verb("DUMPHIER", genIndent(indent)
              + "Window " +win.getName());
    } else if (w instanceof GUIHierarchy.View) {
      GUIHierarchy.View v = (GUIHierarchy.View) w;
      Logger.verb("DUMPHIER", genIndent(indent)
              + " View " + v.getType()
              + " " + v.getIdName() + " "
```

```
              + v.getTitle() + " " + v.getHint());
    }
    for (GUIHierarchy.View v : w.getChildren()) {
      dumpWindow(v, indent + 2);
    }
  }
}
```

The programmatic representation of the GUI hierarchy of an application can be built using `GUIHierarchy guiHier = new StaticGUIHierarchy(output)` statement. The `GUIHierarchy` has a field named `activities` which is a list of `Activity` objects, containing all declared activities of this application. The views (widgets) declared in an activity can be accessed using `getChildren()` method, which returns a list of `View` objects. The type of the `View` object can be retrieved using `getType()` method. The id number of the `View` object can be retrieved using `getId()` method. The id name of the `View` object can be retrieved using `getIdName()` method. The title and hint of this `View` can be retrieved using `getTitle()` and `getHint()` APIs, as shown in the `dumpWindow` method in the example. For more information, please read the source code of `StaticGUIHierarchy` class.

## 3.3 Build the Window Transition Graph

The window transition graph (WTG) can be build inside a `GUIAnalysisClient`. A basic example is like this:

```
public class TestingClient implements GUIAnalysisClient {
  @Override
  public void run(GUIAnalysisOutput output){
    WTGBuilder wtgBuilder = new WTGBuilder();
    wtgBuilder.build(output);
    WTGAnalysisOutput wtgAO = new WTGAnalysisOutput(output, wtgBuilder);
    WTG wtg = wtgAO.getWTG();
    Collection<WTGEdge> edges = wtg.getEdges();
    Collection<WTGNode> nodes = wtg.getNodes();
    Logger.verb(mtdTag, "Number of nodes: "
      +nodes.size() + "\tNumber of edges: "+ edges.size());
  }
}
```

The example code shown above will create a WTG from the result saved in the `output` parameter. All WTG nodes and WTG edges are stored in the `WTG wtg` variable. And the number of WTG nodes and edges will be printed on the screen.

### 3.3.1 WTG related APIs

We provide several APIs to access these nodes. As shown in the example above, API `WTG.getEdges()` will return all available edges in the WTG and API `WTG.getNodes()` will return all available nodes in the WTG.

Every application has a launcher node which stands for starting the application from the launcher. This node can be accessed by using:

```
public WTGNode WTG.getLauncherNode();
```

For each WTG node. The window (activity/dialog/menu) it represents can be accessed through:

```
public NObjectNode WTGNode.getWindow();
```

Any inbound WTG edges of a WTG node can be accessed by:

```
public Collection<WTGEdge> WTGNode.getInEdges();
```

Any outbound WTG edges of a WTG node can be accessed by:

```
public Collection<WTGEdge> WTGNode.getOutEdges()
```

For each WTG edge, its source and target window can be accessed through following APIs:

```
public WTGNode WTGEdge.getSourceNode();
public WTGNode WTGEdge.getTargetNode();
```

Each WTG edge is associate with an `EventType`, for example, it can be clicking on a button, or pressing the **BACK** button. This information can be accessed through:

```
public EventType WTGEdge.getEventType();
```

The event handler triggered in this edge can be accessed through:

```
public Set<SootMethod> WTGEdge.getEventHandlers();
```

In some cases several possible event handlers may be associated with the same event; thus, this method returns a set.

If the edge triggers window life cycle callbacks, these callback methods can be accessed by:

```
public List<EventHandler> WTGEdge.getCallbacks();
```

The sequence of lifecycle callbacks is provided as a list (i.e., the callbacks are ordered). These lifecycle callbacks will occur after the GUI event handlers returned by method getEventHandlers() described earlier. For historic reasons, this methods returns a helper EventHandler object. The `EventHandler` object above is a wrapper for the `SootMethod`, which can be accessed via `EventHandler.getEventHandler()` method.

Each WTG edge is annotated with window stack operations, which can be **push** a window or **pop** out a window. This information can be accessed by:

```
public List<StackOperation> WTGEdge.getStackOps();
```

And the declaration of the `StackOperation` class is:

```
public class StackOperation {
  public boolean isPushOp();
  public NObjectNode getWindow();
}
```

The `isPushOp()` method will return whether current window stack operation is **push**. It will return `false` if the window stack operation is **pop**. The `getWindow()` method will return the window this stack operation is pushing or popping.

### 3.3.2 WTG usage example

Here is a demo of the APIs introduced above:

```
public class WTGDemoClient implements GUIAnalysisClient {
  @Override
  public void run(GUIAnalysisOutput output){
    VarUtil.v().guiOutput = output;
    WTGBuilder wtgBuilder = new WTGBuilder();
    wtgBuilder.build(output);
    WTGAnalysisOutput wtgAO = new WTGAnalysisOutput(output, wtgBuilder);
```

```
    WTG wtg = wtgAO.getWTG();

    Collection<WTGEdge> edges = wtg.getEdges();
    Collection<WTGNode> nodes = wtg.getNodes();


    Logger.verb("DEMO", "Application: " + Configs.benchmarkName);
    Logger.verb("DEMO", "Launcher Node: " + wtg.getLauncherNode());

    for (WTGNode n : nodes){
      Logger.verb("DEMO", "Current Node: " + n.getWindow().toString());
      Logger.verb("DEMO", "Number of in edges: "
              + Integer.toString(n.getInEdges().size()));
      Logger.verb("DEMO", "Number of out edges: "
              + Integer.toString(n.getOutEdges().size()) + "\n");
    }

    for (WTGEdge e : edges){
      Logger.verb("DEMO", "Current Edge ID: " + e.hashCode());
      Logger.verb("DEMO", "Source Window: "
              + e.getSourceNode().getWindow().toString());
      Logger.verb("DEMO", "Target Window: "
              + e.getTargetNode().getWindow().toString());
      Logger.verb("DEMO", "EventType: " + e.getEventType().toString());
      Logger.verb("DEMO", "Event Callbacks: ");
      for (SootMethod m : e.getEventHandlers()) {
        Logger.verb("DEMO", "\t"+ m.toString());
      }
      Logger.verb("DEMO", "Lifecycle Callbacks: ");
      for (EventHandler eh : e.getCallbacks()) {
        Logger.verb("DEMO", "\t"+ eh.getEventHandler().toString());
      }
      Logger.verb("DEMO", "Stack Operations: ");
      for (StackOperation s : e.getStackOps()){
        if (s.isPushOp())
          Logger.verb("DEMO", "PUSH " + s.getWindow().toString());
        else
          Logger.verb("DEMO", "POP " + s.getWindow().toString());
      }
    }
  }
}
```

This example, will print out details information in the WTG nodes and WTG edges with the APIs introduced in the previous section. There is another example client which is `presto.android.gui.clients.ASE15Client` in the GATOR's source code. It provides more advanced usage of the WTG.

# 4 Path Generation

We provide a generic class to perform WTG Path generation. The name of the class is `DFSGenericPathGenerator`. As the name suggests. It performs depth-first traversal on the Window Transition Graph and it will record the path when it satisfies users' requirements. One of its factory methods of this class is:

```
public static DFSGenericPathGenerator create(
          List<IPathFilter> pathFilters,
          List<IPreEdgeFilter> edgeFilters,
          List<WTGEdge> initEdges,
          Map<String, List<List<WTGEdge>>> matchedPath,
          boolean stopAtMatch,
          boolean allowRepeatedEdge,
          int K)
```

There are 2 interface objects required by this class. The first one is the interface `IPathFilter`, which determines if the path traversed by `DFSGenericPathGenerator` satisfies the requirement of the user. The declaration of this interface is:

```
public interface IPathFilter {
  /***
   * Specify the stop rule for the DFS traversal
   * @param P
   * @param S
   * @return
   */
  boolean match(List<WTGEdge> P, Stack<NObjectNode> S);

  /***
   * Return the name of the filter
   * @return the name of the filter.
   */
  String getFilterName();
}
```

Every time when the `DFSGenericPathGenerator` generates a path, it will call the `match` method in the `IPathFilter`, if the `match` method returns true, it means that the path is matched by the pattern defined in this IPathFilter. The matched path will be recorded in `matchedPath` passed in the factory method.

Another interface, `IEdgeFilter` is used to determine if a WTG edge should be added to the generated WTG path during the path expansion. The declaration of this interface is:

```
public interface IEdgeFilter {
```

```
  /***
   * Specify if Edge e should be discarded
   * @param e Current edge
   * @param P Current Path
   * @param S Current WindowStack
   * @return return true if this edge should be discarded. Otherwise
   *         return false
   */
  boolean discard(WTGEdge e, List<WTGEdge> P, Stack<NObjectNode> S);
}
```

Every time when the `DFSGenericPathGenerator` adds a new WTG edge into the current temporary path, it will call the `discard` method in the `IEdgeFilter`. If the `discard` returns true, it means the edge does not satisfy the requirement defined in this `IEdgeFilter`. The WTG Edge will be discarded.

The `List<WTGEdge> initEdges` parameter defines the start point of the path generation. Every WTG edge inside this list will be put in the first place in the generated path. This parameter should not be empty.

The boolean parameter `stopAtMatch` defines the behavior of the DFS traversal when the `match` method in `IEdgeFilter` returns `true`. When this boolean flag is set to `true`, which is its default value, the DFS traversal will stop at this depth when all `IPathFilter` have been evaluated. The DFS traversal will return the previous depth. If this boolean flag is set to `false`, the DFS traversal will continue no matter what is returned by the `match` method.

The boolean parameter `allowRepeatedEdge` defines whether repeated edge is allowed in the generated path. If it is set to `true`, the generated path might contain the same edge for multiple times, which will cause a loop.

The integer parameter `K` defines the maximum length of the path. In our energy analysis, this value is set to 5.

Here is an example which generates WTGPath from any activity with maximum length of 3:

```
public class PathGenerationDemoClient implements GUIAnalysisClient {
  @Override
  public void run(GUIAnalysisOutput output) {

    //Perform WTG Construction
    WTGBuilder wtgBuilder = new WTGBuilder();
    wtgBuilder.build(output);
    WTGAnalysisOutput wtgAO = new WTGAnalysisOutput(output, wtgBuilder);
    WTG wtg = wtgAO.getWTG();

    //Create a placeholder filter class
    IPathFilter ph = new IPathFilter() {
      @Override
```

```
  public boolean match(List<WTGEdge> P, Stack<NObjectNode> S) {
    return true;
  }

  @Override
  public String getFilterName() {
    return "PlaceHolder";
  }
};

List<IPathFilter> pathFilterList = Lists.newArrayList();
pathFilterList.add(ph);

//Create Initial Edges.
//The path generation will begin from these
//Initial Edges

List<WTGEdge> initEdges = Lists.newArrayList();
for (WTGNode n : wtg.getNodes()){
  if(!(n.getWindow() instanceof NActivityNode)){
    //Ignore any window that is not Activity
    continue;
  }
  List<WTGEdge> validInboundEdges = Lists.newArrayList();
  for (WTGEdge curEdge : n.getInEdges()){
    switch (curEdge.getEventType()) {
      case implicit_back_event:
      case implicit_home_event:
      case implicit_rotate_event:
      case implicit_power_event:
        continue;
    }
    List<StackOperation> curStack = curEdge.getStackOps();
    if (curStack != null && !curStack.isEmpty()) {
      StackOperation curOp = curStack.get(curStack.size() - 1);
      //If last op of this inbound edge is push
      if (curOp.isPushOp()) {
        NObjectNode pushedWindow = curOp.getWindow();
        WTGNode pushedNode = wtg.getNode(pushedWindow);
        if (pushedNode == n) {
          validInboundEdges.add(curEdge);
        }
      }
    }
  }
  initEdges.addAll(validInboundEdges);
}
```

```
    Logger.verb("PathGenDemo", "Total Init Edges: " + initEdges.size());

    //Create Output Map
    Map<String, List<List<WTGEdge>>> outputMap = Maps.newHashMap();

    DFSGenericPathGenerator dg = DFSGenericPathGenerator.create(
            pathFilterList, null, initEdges, outputMap, false,false, 3);

    dg.doPathGeneration();

    Logger.verb("PathGenDemo", "K = " + 3 );
    Logger.verb("PathGenDemo", "Total path count: "
      + outputMap.get(ph.getFilterName()).size());
  }
}
```

This example code will use the `DFSGenericPathGenerator` class to generate any path from any Activity node with its length less or equal to 3. It implements a `IPathFilter` that will always return `true`, as the only requirement for the generated path is its length, which is already defined in the parameter `K`.

The `DFSGenericPathGenerator.doPathGeneration()` method will start the DFS path generation. After the execution of this method, the recorded path can be accessed from parameter `matchedPath` passed in the factory method. The key of this map is the filter name defined in `IPathFilter.getFilterName()` method.

## References


[1] A. Rountev and D. Yan. Static reference analysis for GUI objects in Android software. In *International Symposium on Code Generation and Optimization*, pages 143–153, 2014.

[2] H. Wu, S. Yang, and A. Rountev. Static detection of energy defect patterns in Android applications. In *International Conference on Compiler Construction*, pages 185–195, 2016.

[3] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *International Conference on Software Engineering*, pages 89–99, 2015.

[4] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *International Conference on Automated Software Engineering*, pages 658–668, 2015.