GATOR: Program Analysis Toolkit For Android

PRESTO Research Group

September 9, 2019

1 Overview

GATOR is a Prog**r**am **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 18.04 and Mac OS X 10.14. The toolkit takes as input an APK file and runs analyses on top of the Soot program analysis framework.[1]

This release (version 3.7) includes the source code for the static analyses described in our CGO'14 [1], ICSE'15 [4], ASE'15 [5], CC'16 [2] and JASE'18 [3] 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, JASE'18] 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.

Compared to the last release (version 3.6), the following changes were made:

- Bug fixes and performance enhancements, based on extensive testing with over 17K popular apps from Google Play.
- Faster version of the GUI structural analysis (option `-fast`) useful for initial testing and debugging of client analyses.
- Options to enable string analysis of GUI widget text properties.

2 Setup

GATOR can be run as a Docker container without additional manual setup. To run in a native environment on a local machine, JDK, Python 3 and Android SDK are required.

[1] https://sable.github.io/soot

2.1 JDK

JDK 1.8+ is required to run GATOR. Please refer to https://www.oracle.com/technetwork/java for details of how to obtain a copy of the JDK.

2.2 Python

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

2.3 Android SDK

Android SDK is required to run GATOR. It can be downloaded from https://developer.android.com. After installing the SDK, platform files for individual API levels should be installed. For example, if you want to analyze an Android application developed for API level 17, platform files for `android-4.2.2` must be installed. Android Studio includes a user-friendly SDK Manager with graphical user interface.[2] As a backup, GATOR will try its best to install missing files via `sdkmanager`[3] if specific API levels are not present.

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`

3 Usage

3.1 Build GATOR

3.1.1 With Docker

```
$ docker build -t gator . # build image
$ docker run -it --rm --name running-gator gator sh # start a shell
```

3.1.2 In Local Environment

Two environment variables need to be defined, `GatorRoot` should be assigned the path which contains the `AndroidBench` and `gator` subfolders and `ANDROID_SDK` should be assigned the path to the Android SDK, following the commands below:

```
$ export GatorRoot=/path/to/root/of/gator
$ export ANDROID_SDK=/path/to/android/sdk
```

[2] Note that some API levels and their Google APIs are only visible after unchecking the "Hide Obsolete Packages" box.

[3] https://developer.android.com/studio/command-line/sdkmanager

GATOR uses Gradle[4] to manage its dependencies. To build GATOR, go to `gator` sub-directory and run the `gator` script as follows:

```
$ cd $GatorRoot/gator
$ ./gator b
```

3.2 Run Analyses

3.2.1 Using The `gator` Script

The release provides a Python script at `$GatorRoot/gator/gator` to invoke GATOR for the analysis of any arbitrary application in APK format. The basic options for `gator` is:

```
gator analyze [-h] [-d] [-v] [--sdk ANDROID_SDK] [--log LOG_FILE]
                             [-t TIMEOUT] [-g] [--api API_LEVEL] -p APK
```

The usage of these options is printed when `-h` is provided. Below we give some examples. Option `-v` enables verbose mode of GATOR, which will increase the details of logs printed to the `stdout`. Option `--api` overrides the API level information sent to GATOR. By default, GATOR will use the application's target API level. But we have seen cases that applications use APIs from API levels higher than their target API levels that may cause unexpected problems with GATOR. 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 `-t` sets the timeout (3600 seconds by default). In our experience with thousands of apps, a value of 300 seconds works reasonably well for most apps.

The options shown above is specific for the `gator` script. Parameters and options of GATOR (i.e., the main Java program) should be added to control the behavior of the static analyses in GATOR. As an example, if you want to perform analysis on an APK located at `/tmp/example.apk` using `WTGDemoClient`, use following command:

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

The `./gator a` here is a shortcut for `./gator analyze`. The `-client WTGDemoClient` (unknown to the `gator` script) is passed to GATOR. The client is eventually invoked inside GATOR via reflection. As another example, if you want to perform analysis on the same APK using `EnergyAnalysisClient`, use following command:

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

Here `-cp` is a shortcut for passing *client parameters* to GATOR. Please note that, for some obfuscated apps, the `apktool`, which we used to extract APKs, may fail to decode correct tag names in the layout XML files. Unless `apktool` fix this issue, GATOR may crash when performing analyses on these apps. We will introduce more about parameters for GATOR (the main program, not the script) in Section 3.2.3.

3.2.2 Using Provided Script for Demo Applications

The release includes a script at `$GatorRoot/AndroidBench/guiAnalysis.py` that allows to run GATOR on applications in the [CGO'14], [ICSE'15], [ASE'15] and [CC'16] papers.

There are several options to run the `guiAnalysis.py` script. The easiest is as follows:

```
$ cd $GatorRoot/AndroidBench
$ ./guiAnalysis.py runAll
```

[4] https://gradle.org

This performs analyses on applications with default clients. If you only want to perform analysis on applications in [CGO'14] and [ASE'15], replace the option `runAll` with `runAllCGO`. If you only want to perform analysis on applications in [CC'16], 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` and `xbmc` from [CGO'14] and [ASE'15], and `droidar`, `droidar-fixed`, `osmdroid`, `osmdroid-fixed`, `recycle-locator`, `recycle-locator-fixed`, `sofia`, `sofia-fixed`, `ushahidi`, `ushahidi-fixed`, `speedometer`, `heregps`, `whereami`, `locdemo` and `wigle` from [CC'16].

If you want to change the analysis client used for analysis, please modify the configurations in `$GatorRoot/AndroidBench/cgo.json` and `$GatorRoot/AndroidBench/cc16.json`.

Note that in 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 indeed contains a defect. However, it is not accessible by the user and therefore we removed it from the published paper.

3.2.3 Options and Parameters

GATOR provides several options to control its analyses. There are two categories of options, *regular parameters* (denoted by `PARAM` in the JSON config file mentioned above) and *client parameters* (denoted by `CLIENT_PARAM`). Regular parameters control the behavior of GATOR before the client analysis, while client parameters are defined per client analysis. There is no constraint for these parameters and you could define them upon your client's needs. For a complete list of regular parameters/options allowed in GATOR, please refer to the `parseArgs` method in class `presto.android.Main`.

As an example, `-worker NUM_OF_THREADS` is used to define the maximum number of threads GATOR should use. In default setting, GATOR analyzes an application using 16 threads. However, in rare cases, it may experience concurrency issues since part of the underlying Soot framework is not thread-safe. In this case, you can put `-worker 1` in `PARAM`. If the `gator` script is used, you can add it at the end of the command, e.g., `./gator a -p some.apk -client SomeClient -worker 1`.

As another example, `-enableStringPropertyAnalysis` is used to enable the string analysis feature of GATOR, which allows the GUI structural analysis to process the text attribute of each widget defined in the app's layout XML files, as well as set programmatically by calls to `view.setText(String)` and `view.setText(int)`. Another option `-enableStringAppendAnalysis` allows GATOR to process `StringBuilder.append` operations for string concatenation. Both options will increase the overhead of GATOR so they are not enabled by default.

As the last example, option `-fast` is intended for prototyping. If enabled, only one iteration of the fixed point computation is performed. It does not produce a complete solution, but it may be useful initial testing and debugging of client analyses.

As introduced above, client parameters can be used to transfer parameters to the analysis client. For example, for applications used in the [CC'16] paper, we use the option `-clientParam WTPK5` (or `-cp WTPK5` for short) 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`. All `CLIENT_PARAM` can be accessed programmatically via the following global variable anywhere in your client:

```
public class presto.android.Configs {
  public static Set<String> presto.android.Configs.clientParams;
```

```
}
```

4 Creating Your Own Client

In this section, we show how to create a customized GUI analysis client from scratch.

4.1 `GUIAnalysisClient`

In order to implement a customized `GUIAnalysisClient`, one needs to add her 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);
}
```

A client is invoked after GATOR's default GUI analysis in [CGO'14], including reading XML files, determining the relationships between widgets, creating abtract GUI objects, etc. When this is finished, if one has specified some client via options for `gator` (described in the previous section), the `run` method in the client will be called. The parameter `output` of the `run` method provides the results from the GUI analysis in [CGO'14], which can be further used to build the GUI hierarchy or the Window Transition Graph (WTG).

4.2 Build the GUI Hierarchy

We provide the `GUIHierarchyPrinterClient` for GUI hierarchy printing purposes. 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());
    }
    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 numeric ID of the `View` object can be retrieved using `getId()` method. The ID name of the `View` object can be retrieved using `getIdName()` method, as shown in the `dumpWindow` method in the example. For more information, please refer to the `StaticGUIHierarchy` class.

4.3 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 ao = new WTGAnalysisOutput(output, wtgBuilder);
    WTG wtg = ao.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 creates a WTG from the result saved in the `output` parameter. All WTG nodes and WTG edges are stored in the `WTG wtg` variable. It then prints the number of WTG nodes and edges on screen.

4.3.1 WTG Related APIs

We provide several APIs to access these nodes. As shown in the example above, `WTG.getEdges()` will return all available edges in the WTG and `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 a list of window stack operations, each element of which is a **push** operation or a **pop** operation on a window. This information can be accessed by:

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

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.

4.3.2 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 ao = new WTGAnalysisOutput(output, wtgBuilder);
    WTG wtg = ao.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 prints out details information in the WTG nodes and WTG edges with the APIs introduced above. There is another example client, `presto.android.gui.clients.ASE15Client`, in GATOR's source code. More advanced usages of the WTG could be found in that client.

5 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 will record the path when certain users' requirements are satisfied. One of its factory methods of this class is as follows:

```
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
   */
  boolean match(List<WTGEdge> P, Stack<NObjectNode> S);

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

Whenever the `DFSGenericPathGenerator` generates a path, it calls 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);
}
```

When the `DFSGenericPathGenerator` tries to add a new WTG edge into the current temporary path, it calls 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 starting 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 edges are allowed in the generated path. If it is set to `true`, the generated path might contain the same edge for multiple times, which causes loops.

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 WTG paths 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 ao = new WTGAnalysisOutput(output, wtgBuilder);
    WTG wtg = ao.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("Demo", "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("Demo", "K = " + 3 );
```

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

This example code uses the `DFSGenericPathGenerator` class to generate any path from any `Activity` node of 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 is invokde when the DFS path generation starts. 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, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev. Static window transition graphs for Android. *International Journal of Automated Software Engineering*, 25(4):833–873, Dec. 2018.

[4] 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.

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