UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Specification and Modelling of Software Systems

Master's Thesis

Submitted to the Specification and Modelling of Software Systems Research Group
in Partial Fullfilment of the Requirements for the Degree of

Master of Science

Android App Analysis Benchmark Case Generation

by
Stefan Schott

Thesis Supervisor:
Prof. Dr. Heike Wehrheim
and
Prof. Dr. Eric Bodden

Paderborn, April 8, 2021

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Contents

1 Introduction

Smartphones nowadays are an omnipresent part of everyone's life. In contrast to desktop computers they record and deal with a lot of sensitive information like, for example, various sensor data like the device's GPS location, the user's contact data and other sensitive information such as the device's IMEI number, which is used to uniquely identify a device. In most situations this information cannot be leaked to the outside world. However, such leaks happen many times and can happen due to various reasons. Such a leak might be due to malicious reasons, because the developer of the application wants to steal data from the user. Another possible reason for a leak may be because of a vulnerability inside of an application which is only accidentally part of the application. Leaking sensitive information to the outside world however, may also happen on purpose, if, for example, the user of the device wants to share a contact with a friend. For this reason it is important to have adequate analysis tools which are able to detect leaks of sensitive data in mobile applications and distinguish possibly malicious leaks from intended leaks. Since Android devices make up by far the biggest share of the worldwide mobile operating system market [19], it is especially important to have efficient and reliable analysis tools for Android applications.

An analysis able to deal with sensitive data leakage is called *taint analysis*. A taint analysis tracks the data flow of sensitive information throughout the application before it gets leaked to the outside world. Such a leaking flow is called *taint flow*. Therefore, the main goal of a taint analysis is to uncover all taint flows contained in applications.

Currently there are multiple tools available, which are able to perform a taint analysis on Android applications. While AMANDROID [49] and FLOWDROID [26] currently are state of the art, there are still many more analysis tools [25, 28, 31, 37, 38, 39, 40, 35, 54]. Two distinct studies [44, 45] showed that many of these tools have some deficits and are not able to detect sensitive data leaks in all kinds of applications. In the constant race of attacker, who tries to steal data, and defender, who tries to create analysis tools to uncover stealing attempts before they happen, such deficits can be very critical. In order to reveal these deficits and improve taint analysis tools, many versatile *benchmark cases* are needed that can be used to evaluate the performance of the various taint analysis tools. In general, benchmark cases are applications that are specifically used to evaluate the performance of hardware or software. In the context of Android taint analysis a benchmark case consists of a regular Android application which contains one or more taint flows. Each taint flow consists of (1) a *source* that introduces sensitive data into the application, (2) some data flow the sensitive data is passed through, and (3) a *sink* that subsequently leaks the data to the outside world. Each of these parts of a taint flow can come in different variations implementing, for example, reflection, recursion, various obfuscations and

data structures, or even all of the mentioned *aspects* combined. This means that there is a vast amount of possibilities each taint flow can look like. Therefore, there needs to be a large amount of versatile benchmark cases to be able to properly evaluate the performance of taint analysis tools.

The current approach to evaluate Android taint analysis tools is the combined usage of *benchmark suites*, which are collections of multiple benchmark cases. Almost all of them consist solely of hand-crafted *micro benchmark cases* (very simple applications that usually contain only one simple taint flow), that have been specifically created for the purpose of analysis tool evaluation. This approach contains multiple drawbacks and issues. One issue is that the manual creation of benchmark cases offers the potential for frequent implementation mistakes (Issue 1). This is due to the vast amount of benchmark cases needed to cover all aspects a taint flow can contain. Furthermore, the manual creation of these benchmark cases requires a lot of time and effort from the creator. Another issue is that the reliance on micro benchmark cases may enable an *over-adaptation* of analysis tools to micro benchmark cases (Issue 2). An analysis tool may be able to uncover taint flows that contain reflection and recursion individually, but not combined. The developer of the analysis tool would not detect this behavior, since micro benchmark cases almost never combine multiple aspects. Another even clearer example for an over-adaptation of an analysis tool to a benchmark case is that the file name of the micro benchmark case could contain the aspect that is present in its taint flow. The benchmark case's file name could be, for example, `ReflectionBenchmarkCase1.apk`. The analysis tool could check the file name and decide which analysis strategy to apply, based on the keyword `Reflection`. This may lead to mistakenly good evaluation results, as this strategy can clearly not be successfully performed for real-world applications. This issue may be in part overcome by the integration of real-world applications in the analysis tool evaluation process. However, for almost all available applications there is no *ground-truth* available that contains the expected analysis results (Issue 3). Without this information one cannot determine whether the analysis tool yielded the right analysis results as only the amount of uncovered taint flows can be reported. The amounts of falsely detected taint flows and taint flows that do not have been detected at all, are still unknown. This thesis presents an Android app analysis benchmark case generation concept that is able to help overcome the three above mentioned issues.

A template based approach for a benchmark case generator is developed that enables an on-the-fly benchmark case generation. This approach allows for a generation of arbitrarily complex benchmark cases containing an arbitrary amount of taint flows, each containing various user-desired aspects. By automatically generating benchmark cases, the possibility of implementation mistakes, as well as the required effort from the creator, drastically decreases. Furthermore, this approach provides a mean for the user to easily extend the possible variety of taint flows and therefore always be up-to-date with newly identified vulnerabilities. This generation process helps overcome Issue 1, as only reusable code-snippets have to be created, in contrast to creating a full application for each desired benchmark case.

Based on this concept a benchmark case generator called GenBenchDroid is implemented to test the concept in practice and evaluate its performance. This evaluation results in GenBenchDroid being a powerful tool that is able to efficiently regenerate hand-crafted benchmark cases on-the-fly, containing various taint flows. Furthermore, GenBenchDroid's capability of efficiently generating versatile benchmark cases of various complexities is shown. By being able to generate benchmark cases on-the-fly that combine all kinds of aspects, an over-adaption, like described in Issue 2, can be prevented. Additionally, the evaluation shows that GenBenchDroid is also able to generate benchmark cases that are comparable to real-world applications and provide a proper ground-truth for them. This generation of a corresponding ground-truth for benchmark cases that are comparable to real-world applications and integrating them into

the evaluation process helps overcome Issue 3. Therefore, GenBenchDroid can help overcome all of the above mentioned issues, by generating numerous, arbitrarily complex benchmark cases that are comparable to real-world applications which are suitable to be used in the process of Android analysis tool evaluation. In addition to that, GenBenchDroid provides a *fuzzing* capability that allows for an automatic generation of random, but valid, benchmark cases of differing complexities, with no effort from the user required. The evaluation of this fuzzing capability shows that GenBenchDroid is able to generate a meaningful benchmark suite with an average deviation of around 1% for all investigated analysis metrics (in comparison to other generated benchmark suites) by generating just 500 benchmark cases. This further shows GenBenchDroid's strength of generating a vast amount of various benchmark cases to make an over-adaptation unlikely.

1.1 Thesis' Contents

The first part of this thesis introduces fundamentals that are used in each chapter of the thesis. Since the main purpose of the concept is the generation of analyzable Android applications, at first various specifics of the Android system and the development of Android applications are described (see Section 2.1). Next, the fundamentals of program analysis and more specifically taint analysis are explained on the basis of a running example that stretches throughout the thesis. In addition, the benchmarking process in general, its requirements and used metrics are introduced. Furthermore, an overview of benchmark suites available for Android taint analysis is given (see Section 2.2). Last, a short overview on the topic of fuzzing is given (see Section 2.3), because the developed concept is designed to allow for an additional fuzzing capability.

In Chapter 3, the Android app analysis benchmark case generation concept is described. All requirements, the framework of a possible generator, as well as all components of its framework are described in this chapter. Furthermore, all inputs, as well as the outputs are presented in detail. Additionally, an overview of how to integrate a benchmark case generator into the evaluation process of an analysis tool is given.

Based on the presented concept, a benchmark case generator (GenBenchDroid) is implemented in Chapter 4. This chapter describes how the implementation of the resulting benchmark case generator is connected to the underlying concept and also contains a user manual for the benchmark case generator. The implemented benchmark case generator is evaluated in Chapter 5. This chapter shows that all requirements which are introduced in Section 3.1 are fulfilled and furthermore offers an assessment of the performance of the benchmark case generator.

Afterwards an overview about similar benchmark case generator approaches for other application areas, as well as current benchmarking solutions for Android devices is given (see Chapter 6). In addition to the related work, an overview over threats to the validity of the benchmark case generation concept, as well as possible subjects of further work, is given in Chapter 7. Finally, this thesis concludes with a summary of its results and achievements (see Chapter 8).

2 Fundamentals

This chapter introduces various topics that are used throughout the thesis. Since the application area of this thesis is the Android system, several features that are unique to Android systems are introduced, alongside fundamentals regarding program and especially taint analysis. At last a short introduction to the topic of fuzzing is given.

2.1 Android Basics

This section describes various Android specifics that are fundamental to understanding the behavior of Android applications. It introduces topics like basic components an application is made off, lifecycles of these components and further Android application specific features.

2.1.1 Application Components

Each Android application is built from four different types of components [1]:

1. Activities
2. Services
3. Broadcast receivers
4. Content providers

Each type of component serves a distinct purpose.

An *activity* is responsible for user interaction. Each activity comes with its own user interface. An application usually contains multiple activities that interact with each other. However, an activity is also independent from other activities and can be started on its own. An example for this behavior is the user's contacts application. The user can use this application to view his/her contacts, add a new contact or initiate a call to one of his/her contacts. Each of these actions is encapsulated in its own activity. All of them are working together to form a fluent user experience. However, each of these activities is also independent from the others. This behavior can be observed when the user clicks on a text message containing shared contact-information and instead of the contacts application's main screen opening, the screen for adding contacts is opened. Usually activities are the most common building blocks for Android applications and also contain the biggest share of the application logic.

A *service* is responsible for handling actions that are performed in the background. Because of this, a service does not provide a user interface. A service is usually started by another component like an activity, and either runs until it finishes its task or until the Android system needs to free up RAM for tasks that are more important at this point in time than the service. The service running until it finishes is usually the case for services that the user is aware of, like playing music in the background. The service being terminated by the Android system usually happens for services that the user is not aware of, like syncing data in the background.

A *broadcast receiver* is a component that allows the system to deliver events to the application, that are not performed by the application's user. These events, that are emitted by the system, are system-wide broadcast announcements that are delivered to all applications. A broadcast receiver enables the application to react to these announcements without the application currently running on the device. There are multiple use-cases for system-wide broadcast announcements, such as notifying each application when the screen has been turned off, the battery is low or a picture has been taken. Besides the system, other applications are also able to broadcast announcements which can be picked up by a broadcast receiver. This might for example happen, when a download is finished. While a broadcast receiver component does not provide its own user interface, it can create a notification in the device's status bar to notify the user when a broadcast event occurs. However, usually a broadcast receiver is only intended to start up another component which then handles the received event.

A *content provider* is a component that is responsible for managing sets of data that are shared across multiple applications. If a content provider allows the modification of a certain data set, another application is able to access and modify the data that is provided by the content provider. One example for this is that an application with correct permissions is able to access the user's contacts, that are provided by a content provider, and read, as well as modify them. This allows the Android system to share data sets across multiple applications, while still maintaining a form of security through permissions that have to be requested by an application before being able to access the data.

2.1.2 Inter-Component Communication (ICC)

Since an Android application is composed of multiple components, there needs to be a way for a component to start up another component and to communicate with it. This communication happens via *intents*. An intent defines a message that activates a component. There are two types of intents: (1) *explicit intents* and (2) *implicit intents*. While an explicit intent is used to activate a specific component, an implicit intent is used to activate any component that is capable of performing a desired action. An explicit intent can, for example, be used to activate a specific activity called `TakePictureActivity`, while an implicit intent can be used to activate an arbitrary component that is able to take a picture.

An intent is not limited to components inside a single application. An application can, for example, send an implicit intent that requests a component which is able to take a picture and a component from a completely different application can receive this intent and start up its component that is responsible for taking pictures.

An intent for an activity or service defines the action to be performed by the activated component. It can also contain data that the component is requested to perform an action on. In some cases the requested component needs to return a result to the application that requested it once it finishes its task. This return of the result also happens via an intent. One example for this is an activity that requests the contacts application to open and the user to select one of his/her contacts. This selected contact is then returned to the original activity, that sent the intent to the contacts application, via another intent [1].

2.1.3 Android Manifest File

The *manifest file* is an important configuration file for Android applications that serves multiple purposes. It contains a declaration of all components of the application, the permissions that the application requires, requested hardware and software features and various other information. Components have to be declared so they can be seen by the system and started, if an explicit or implicit intent is issued. Furthermore, the component's capabilities have to be declared, so the system knows what actions the component is able to perform. Required permissions are declared, so that users have an overview about which permissions are required by the application before installing it [3]. The required software and hardware features need to be declared, so that an installation of the application on a device that does not provide enough capabilities can be prevented [2].

Listing 2.1 shows an example of a manifest file. It is represented in an XML format and depicts a photography application that consists of two activities. Each activity comes with its own *intent-filter*. An intent-filter depicts the capabilities of the component and tells the system to what kind of implicit intent this component is able to react. The first component that is represented in the example is an activity called `MainActivity`. The action that is specified by its intent-filter means that this activity is the main entry point to the application and it starts when the application is initially launched when the user presses the launcher icon on his/her device. Furthermore, it means that this activity does not need an intent to be started. The second activity is called `TakePictureActivity`. This activity is responsible for taking the pictures of the photography application. This capability is depicted by its intent-filter. When an application issues an implicit intent of the type `IMAGE_CAPTURE`, this activity is able to react to the intent. Since the application requires the device's camera, the required permission, as well as the requested hardware feature also have to be specified in the manifest file [5].

```
<manifest ...>
  <application ...>
    <activity android:name="MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        ...
      </intent-filter>
    </activity>

    <activity android:name="TakePictureActivity">
      <intent-filter>
        <action android:name="android.media.action.IMAGE_CAPTURE" />
        ...
      </intent-filter>
    </activity>
  </application>

  <uses-permission android:name="android.permission.CAMERA" />

  <uses-feature android:name="android.hardware.camera" />
</manifest>
```

Listing 2.1: Android manifest file example (shortened)

2.1.4 Application Resources and Layout

An Android application usually consists of more resources than just its source code. Some examples of additional resource files are image files, audio files and most importantly *layout files* that describe the visual presentation of the application. Layout files contain the whole user interface which usually consists of view elements like, for example, labels or dialogues, and input elements like, for example, buttons, text-fields or check-boxes. This layout definition is represented in XML format with a series of nested `View` and `ViewGroup` objects. A `View` represents an element that is shown on the screen which the user often can interact with, like for example a button or an element that provides information to the user, like a label. A `ViewGroup` defines an invisible container that can be used to model the layout structure. Each user interface element of an Android application belongs to one of these two categories. The layout definition of the application is contained in its own file, so that the program and the presentation logic are separated. Usually each activity comes with its own layout file. However, a single layout file can also be used for multiple activities [6, 1].

2.1.5 Events and Callbacks

Because of the nature of mobile devices and applications, *events* play a big role in the development of Android applications. Events can be incoming calls, specific sensor values that are reached, the battery level falling below a certain threshold or any kind of user input. Events are handled by so called *event listeners*. Event listeners are functions that are executed as soon as the according event, that is listened to by the event listener, is detected. They are implemented by using *callbacks*. A callback is a function that is passed as an argument to another part of the application code and is expected to be executed at a given time, in the case of event listeners whenever a specific event happens.

Figure 2.1 illustrates an example for listening to an event. When a user presses a button, he/she triggers a button-pressed-event. The Android system monitors the button and detects when the button is pressed by the user. Once the Android system detects that the button was pressed by the user, it calls the supplied callback function which is then executed [4].

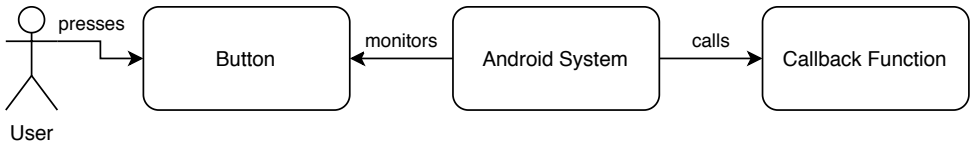


Figure 2.1: Listening to a button-pressed-event

2.1.6 Activity Lifecycle

An Android application consists of many components. Furthermore, in contrast to many other programming paradigms that provide a single entry point to an application like the `main()` method in programming languages like *Java* and *C*, an Android application may provide multiple entry points to the same application. This means that each component of an Android application can be in different execution states during its lifetime. If a user opens his/her contacts application to call another person, the application opens and shows the user a list of all of his/her contacts. But if the user clicks on shared contact information that has been sent to him/her by someone, the contacts application directly opens the screen for adding contact information with the forms already filled out with the shared contact information. This means that one component of the

application can already be in a running state, while the other component has not yet been started at all. Furthermore, a mobile application often has to react to various events, which additionally strengthens the need for multiple component states. This means that there is the need for callback functions to handle these state transitions. One example for this is, if the user watches a video on his/her device and receives a call, the video, as well as the component playing the video, need to be paused and resumed once the call is finished.

Figure 2.2 illustrates the *lifecycle* of the activity component. After the launch of the activity, the callback method `onCreate()` is executed. This method is always the first method that is executed once an activity is started. Afterwards the activity reaches the state called *Started* and directly thereafter enters the state *Resumed*, in which it stays until it is paused. When the activity is no longer in focus, but parts of it are still visible on the device it enters the *Paused* state. This may happen if the user's device is in multi-window mode and the focus of the user is on another application, while the paused one is still visible. When the user focuses the paused application again, the currently paused activity is resumed again. If the user opens up another application that covers the device's whole screen and puts the current application in the background, it reaches the *Stopped* state. Once the activity has finished its task or is terminated by the user, it executes the `onDestroy()` callback, that is usually used to perform cleanup operations like purging the memory or unregistering listeners, before it reaches the final *Destroyed* state. Between every state transition there is a callback function that is executed before the state transition that can be used to perform necessary operations. [8]

The service, broadcast receiver and content provider components have unique lifecycles as well, that differ from the activity lifecycle. However, since activities are usually the most important and common building block of an Android application, the lifecycles of the other components will not be explained in detail.

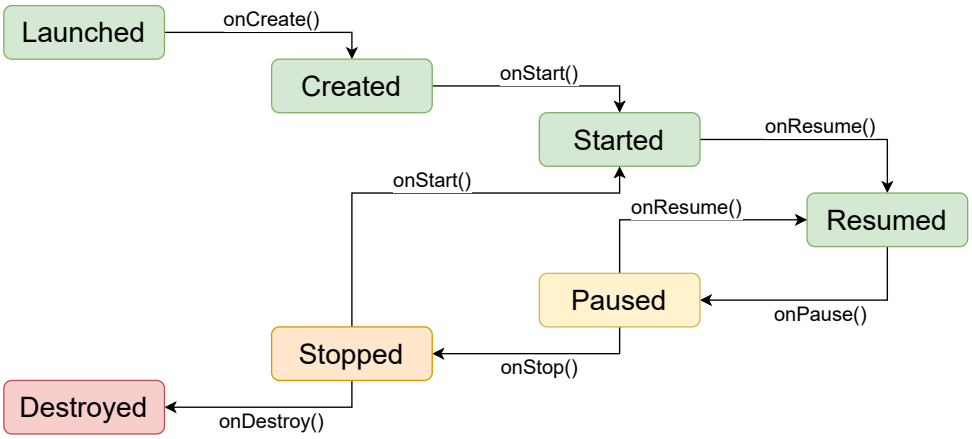


Figure 2.2: Illustration of the Android Activity Lifecycle, based on [8]

2.1.7 Android Package File

An executable Android application comes in form of an *Android Package (APK) file*. The APK file is a container that contains, among other data, the application's compiled source code, the manifest file, the layout files and all the application's resources, like, for example, images and videos. Before an application can be installed on a device, Android requires the APK file to be digitally *signed* with a certificate. The certificate contains the public key of a private/public key pair and other metadata of the owner of the certificate. When an application is signed, the certificate is attached to it and therefore the application is assigned to the owner of the certificate

and his/her private key. Throughout the lifetime of the application the same certificate has to be used. This way it is ensured that all future updates come from the owner of the certificate [1, 7].

2.2 Program Analysis

Program analysis is the process of analyzing the behavior of applications in regard to some properties. These properties can be, for example, safety, security, or correctness [43].

Program analysis can be divided into two categories, (1) *static program analysis* and (2) *dynamic program analysis*. Static program analysis is typically run on an external system outside of the operating environment. The analyzed application is not executed during the analysis, only its source code is analyzed. Dynamic program analysis is run while the application to be analyzed is executed. It usually is run on the same operating environment as the application. Since dynamic program analysis is performed during the execution of the application under test, it has access to all of the runtime information, allowing it to analyze data flows that are too complex for a static program analysis. However, the downside of dynamic program analysis is that it can only analyze data flows that are executed, which leads to static analysis usually having a higher degree of code coverage. [29]

One of the techniques that is used to analyze a program is called *data flow analysis*. Data flow analysis tracks the internal values of a program at each point in time and listens for changes of these values [43].

A more specific type of data flow analysis, that is focused on the tracking of sensitive user data, is called *taint analysis*. Taint analysis is performed to track flows of sensitive data, which are called *taint flows*, throughout an application and to analyze whether the sensitive data is leaked to the outside world. Figure 2.3 depicts such a taint flow. The sensitive data is introduced into the application through a *source*. It is then propagated through some kind of data flow throughout the application until it reaches a *sink*, where the sensitive data is leaked to the outside world. The goal of a taint analysis is to find taint flows that introduce sensitive data into the application and afterwards leak it to the outside world.

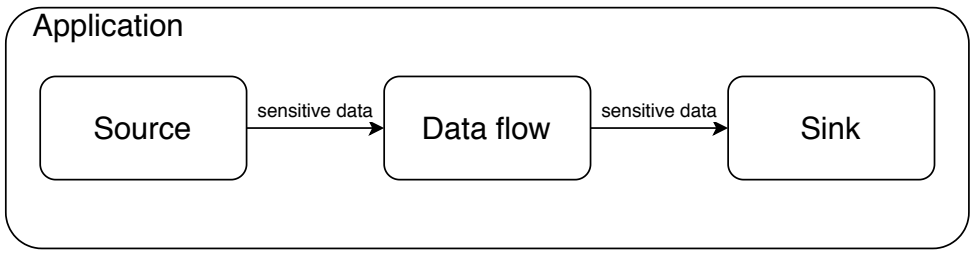


Figure 2.3: Depiction of a taint flow

Figure 2.4 illustrates a simple example of a taint flow. Sensitive data is introduced into the application by reading the device's IMEI (International Mobile Equipment Identity) number, which serves as a unique identifier for a device. This depicts the source of the taint flow. Afterwards the IMEI number is stored inside an array. Finally, the IMEI number is retrieved from the array and sent to another device via SMS, which represents the sink of the taint flow. Therefore, sensitive data, in form of the device's IMEI number, has been introduced into the application and subsequently leaked to the outside world. This example will be used as a running example throughout the thesis.

Taint flows are not always as simple as depicted in Figure 2.4. Figure 2.5 shows a more complex

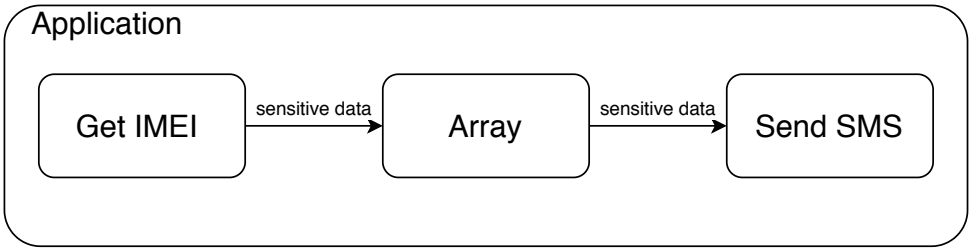


Figure 2.4: Simple taint flow (running example)

taint flow example. The user's private contacts are introduced into the application. Afterwards the data flow is branched into two flows. The first flow puts the user's contacts into an array which are then subsequently retrieved and passed on to another component of the application via Inter-Component Communication. This component leaks the user's contacts to the outside world via sending an Email. The second flow writes the user's contacts into a file and then uploads the file to an external server. This means that there are two possible taint flows inside the application.

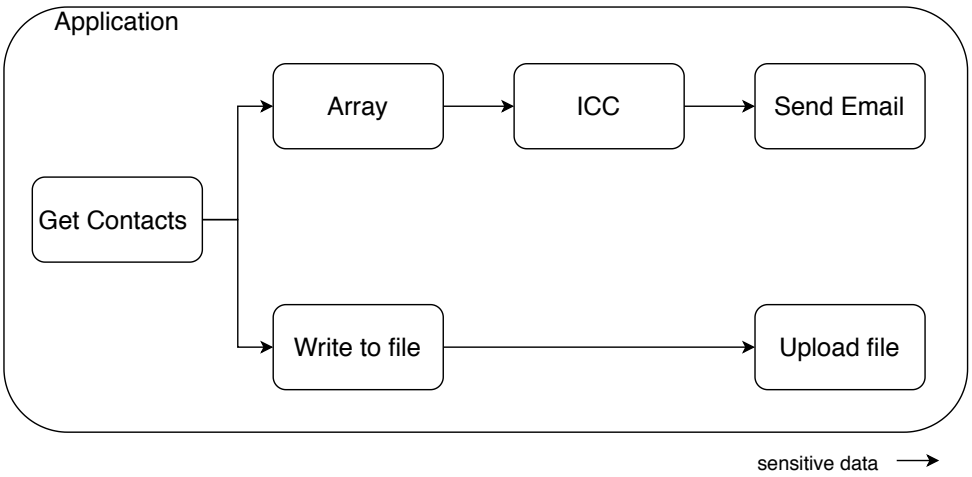


Figure 2.5: More complex taint flow example

There are many possible scenarios for taint flows. A device contains a lot of sensitive data that can be leaked in many ways. This data could be introduced by a reflectively loaded class, an external library or a native function call. There could be multiple split flows and the sensitive data could be *sanitized* (overwriting the data by something non-sensitive) before it reaches a sink, rendering it a *negative taint flow*, which in contrast to a *positive taint flow* is a flow that stretches from a source to a sink, but does not leak sensitive data. This variety of possibilities that have to be handled during the analysis make taint analysis a non-trivial problem.

2.2.1 Taint Analysis Tools for Android

Currently there are multiple taint analysis tools available for Android applications. These tools all provide different capabilities. One example for this discrepancy in capabilities is ICC. While some tools are able to handle ICC, many other tools do not have this capability. There are dynamic as well as static taint analysis tools available [51, 26, 49]. However, most of the available tools perform only static analysis.

The taint analysis tools that are currently the state of the art in Android taint analysis are FlowDroid [26, 14] and Amandroid [49, 10]. Both of these tools are static analysis tools and provide largely the same set of capabilities, with the exception of ICC, which is officially supported by FlowDroid, but still has some unresolved issues [23, 24]. However, FlowDroid is still the most widely used static-analysis tool for Android [44].

2.2.2 Benchmarks

A *benchmark* is a process in which the relative performance of hardware or software is assessed. Often benchmarks are used to compare the performance of different hardware components of a computer system like, for example, comparing the relative performance between multiple CPUs. However, there are as well so called *software-benchmarks* that are used to compare the relative performance of programs to each other. A typical use-case for software-benchmarks is the assessment of the performance of analysis tools.

In the context of the evaluation of Android taint analysis tools, a benchmark consists of multiple Android applications that have certain features [44]. Each individual application is called *benchmark case*. Therefore, a benchmark consists of multiple benchmark cases. This collection of benchmark cases is often also called *benchmark suite*. For the evaluation of Android taint analysis tools there are multiple types of benchmark cases that are used. Often used benchmark case types are (1) *micro benchmark cases* and (2) *real-world benchmark cases*.

Micro benchmark cases are relatively small applications that only contain a few, often only a single, taint flow. They are explicitly created to be used for the evaluation of analysis tools. Listing 2.2 shows a micro benchmark case that contains the taint flow from the running example (see Figure 2.4). As soon as the activity `MainActivity` is created, the device's IMEI number is put into an array. Afterwards it is retrieved from that array and sent to another phone via SMS. This application represents a typical micro benchmark case. It contains only a single taint flow with only a single source and sink. The purpose of this micro benchmark case is to check whether the analysis tool is able to handle taint flows that pass the sensitive data through an array and whether the tool is able to distinguish different array indexes. Other micro benchmark cases focus on different aspects of applications like, for example, reflection, ICC or different kinds of data structures. Usually each micro benchmark case only contains one of these aspects and they are not combined inside a single benchmark case.

Real-world benchmark cases are applications that either are real-world applications or are comparable to real-world applications. Real-world benchmark cases are usually bigger and more complex than micro benchmark cases. They often contain multiple taint flows. They also often contain very complex taint flows, which for example may contain reflectively loaded classes, ICC throughout many components and multiple obfuscations inside a single taint flow. Real-world benchmark cases are important in order to assess the performance of analysis tools when applied to real-world applications. They help to avoid over-adaptation of analysis tools to benchmarks that only contain micro benchmark cases.

```
import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.telephony.TelephonyManager;

public class MainActivity extends Activity {
```

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    TelephonyManager tm =
      (TelephonyManager)getSystemService(
        Context.TELEPHONY_SERVICE
      );

    String[] array = new String[10];
    array[5] = tm.getDeviceId();
    array[4] = "no taint";

    SmsManager sm = SmsManager.getDefault();
    sm.sendTextMessage("+49123", null, array[5], null, null);
  }
}
```

Listing 2.2: Micro benchmark case example

Ground-Truth

A benchmark alone is only able to determine the relative performance of analysis tools, based on the amount of vulnerabilities that have been found. If one tool is able to find more vulnerabilities than the other it is regarded as the superior tool. In order to assess a tool's performance independently, a *ground-truth* is needed [32]. In the context of Android taint analysis the expected results of the taint analysis are contained inside the ground-truth. It contains information about all taint flows that are contained inside a benchmark case. It usually contains information like the location of the source and the sink and also which statements comprise the source and the sink. Furthermore, the ground-truth contains the information whether the taint flow is a positive or a negative taint flow. If the taint flow is a negative taint flow, it should not be detected by an analysis tool. This way it is possible to check a tool's proneness to false warnings.

While micro benchmark cases usually come with a ground-truth, since the applications are explicitly created for the evaluation of analysis tools, real-world applications most of the time do not provide a corresponding ground-truth. This matter of fact makes it often infeasible to incorporate real-world benchmark cases into the evaluation process.

Android Benchmark Suites

Currently there are multiple benchmark suites for the evaluation of Android taint analysis tools available. Each of them comes with a set of multiple benchmark cases that can be used to evaluate the performance of an Android taint analysis tool in regard to different aspects.

DroidBench [13] is a benchmark suite that is exclusively comprised of micro benchmark cases. It contains 190 different Android applications. Although it can be used for the evaluation of static, as well as dynamic taint analysis tools, the benchmark cases provided by DroidBench model problems that are mainly interesting for static analysis tools. DroidBench divides its micro benchmark cases into 18 different categories. Some of these categories are Reflection, Lifecycle and ICC. The Reflection category contains applications that implement reflectively loaded classes, which contain the source or the sink of the application's taint flow. The Lifecycle category contains applications that model components at different lifecycle states, while the ICC category contains applications with different types of ICC. DroidBench does not

provide ground-truths that explicitly describe the contained taint flows of the benchmark case. The contained locations of the sinks, sources and the flow between them are not provided in a machine-readable format. The only provided information is the amount of leaks that the benchmark case contains, in form of a comment inside the benchmark case's source code. Furthermore, the provided amount of leaks is sometimes even completely wrong. This fact makes it impossible to automatically compare actual results against expected results [44].

ICC-Bench [17] is a benchmark suite that is as well exclusively comprised of micro benchmark cases. In comparison to DroidBench, ICC-Bench does not provide multiple categories of benchmarks, but only consists of benchmark cases that implement ICC. However, it contains in total 24 micro benchmark cases and therefore enables a more thorough analysis in regard to ICC than DroidBench does. The provided ground-truth information is as imprecisely specified as it is by DroidBench. It does not contain any information regarding the sources, sinks and flows between them. The only information provided is the amount of leaks that are contained in the application. This information is also provided in form of a comment inside a benchmark case's source code [44].

DIALDroid-Bench [12] is a benchmark suite that exclusively contains real-world benchmark cases. It contains 30 real-world Android applications. For each application only the APK file is provided. The source code of the applications is not provided. Furthermore, there is no ground-truth information provided at all. This fact makes it impossible to determine whether an analysis tool was able to find all taint flows inside an application from DIALDroid-Bench.

TaintBench [22] is a benchmark suite that contains multiple real-world benchmark cases. It contains 39 malicious real-world Android applications. Each of these applications comes with its source code and a detailed ground-truth description. The ground-truth comes in form of a JSON file that contains every source and sink, as well as their location inside the source code. Furthermore, all intermediate flows between a source and a sink are described as well. It also contains information that describes whether the taint flow is a positive or a negative flow. This very detailed ground-truth allows for a automatic comparison of expected and actual analysis results and therefore allows for a thorough evaluation of Android taint analysis tools.

Furthermore, there exists a tool which helps with the refinement of the mentioned missing ground-truths and the subsequent execution of the benchmarks called BREW [44, 11]. It provides aid in identifying the location of sources and sinks of a benchmark cases, as well as declaring taint flows as positive or negative flows. However, these steps are still a manual process, which have to be performed by a user. BREW only provides assistance in these steps.

Experimental Metrics

In order to evaluate the performance of Android taint analysis tools, different *experimental metrics* have to be introduced. Table 2.1 introduces the four outcomes that an analysis tool can produce and in which cases they are produced in the context of taint analysis. The expected result describes each taint flow that is actually contained inside the analyzed application. The goal of the analysis tool is to produce a correct analysis for each taint flow. Whenever the analysis yields the same result as is expected, a correct analysis has been performed. Therefore, a *true positive* and *true negative* outcome indicate a correct analysis of the taint flow, while a *false positive* and *false negative* outcome show a false analysis of the taint flow.

In order to evaluate the performance of an analysis tool on a benchmark suite, one can use the analysis tool to analyze all benchmark cases that are part of the benchmark suite and accumulate all analysis outcomes. Afterwards one can calculate the *precision* and the *recall* based on accumulated outcomes. The precision describes how many of the as positive declared taint flows are actually positive flows and can be calculated as follows:

Outcome	Expected result	Analysis result	Correct Analysis
True Positive (TP)	taint flow	taint flow	✓
False Positive (FP)	-	taint flow	✗
True Negative (TN)	-	-	✓
False Negative (FN)	taint flow	-	✗

Table 2.1: Possible analysis outcomes

$$\text{precision} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FP}|}$$

The recall describes how many of the encountered positive taint flows have been declared as such and can be calculated as follows:

$$\text{recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}$$

In order to combine precision and recall into a single measure the *F-measure* is used. The F-measure is the harmonic mean of the precision and the recall and can be calculated as follows:

$$\mathcal{F} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

These three measures are often used for detection-related experiments and therefore are commonly used to evaluate the performance of analysis tools [36].

2.3 Fuzzing

Fuzzing is a test strategy where random input strings are generated for a program to uncover failures, when the program is executed with these generated inputs. It uses a *fuzz generator* that is responsible for generating random strings. These random strings are then used as input for the program under test. This process is repeated until the program under test hangs, crashes, produces an undesired output or asserts any kind of undesired behavior [53].

Figure 2.6 illustrates the application of a Fuzzer to test a program. The Fuzzer generates a random input and supplies it to the program under test. Then the program is executed with the supplied input. If the program behaved unexpectedly, a failure has been uncovered that was triggered by the supplied input. If there was no unexpected behavior, the whole process is repeated.

Since just generating arbitrary random inputs until a program, which is receiving this input, crashes is a fairly naive and inefficient approach, there are many variations of fuzzing that aim to integrate user-knowledge into the fuzzing process and thereby provide a more efficient approach.

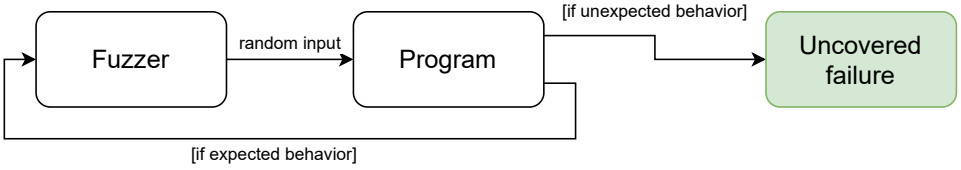


Figure 2.6: Fuzzing example

2.3.1 Grammar-based Fuzzing

Grammar-based fuzzing is an advanced fuzzing technique that uses grammars to generate only syntactically correct inputs for the program under test. This allows for a very systematic and efficient testing process, since there are no invalid inputs supplied to the program under test.

Listing 2.3 depicts a grammar for arithmetic expressions. If, for example, one would want to test a program that takes an arithmetic expression as its input, this grammar can be used. A regular fuzzer would generate many inputs that do not form valid arithmetic expression and only very few valid expressions. However, a grammar-based fuzzer supplied with the grammar from Listing 2.3 will only generate valid inputs for the program under test. The fuzzer starts with the `<start>` symbol and then expands one symbol after the other, choosing random alternatives every time, until it reaches a valid arithmetic expression. This generated arithmetic expression will then be supplied to the program under test. This approach clearly increases the efficiency of the testing process by a large margin [53].

```
<start>  ::= <expr>
<expr>   ::= <term> + <expr> | <term> - <expr> | <term>
<term>   ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= + <factor> | - <factor> | (<expr>) | <int> | <int>.<int>
<int>    ::= <digit><int> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Listing 2.3: Arithmetic Expression Grammar, based on [53]

3 Concept

The performance of benchmarks that are applied to Android taint analysis tools is strongly dependent on the quality of the benchmark cases that are part of the benchmark. The proper creation and selection of benchmark cases is not a trivial task and can contain many difficulties as can be seen in Section 2.2. Prominent issues that contribute to these difficulties are:

1. Possible implementation mistakes in benchmark cases
2. Possibility of over-adaptation of analysis tools to micro benchmark cases
3. Missing, incomplete or incorrect ground-truth information for real-world benchmark cases

Taint flows usually come in many possible variations. Some taint flows may contain reflection, while other taint flows may contain different kinds of obfuscation or ICC. This fact indicates that a thorough evaluation of a taint analysis tool calls for a vast amount of micro benchmark cases which all contain different aspects of possible taint flows. Creating this big amount of micro benchmark cases is a time consuming and error prone task. Furthermore, an implementation mistake in the benchmark case may lead to a wrong ground-truth description. If, for example, a sink may be unreachable due to an unnoticed implementation mistake, the creator of the benchmark case still assumes that the sink is reachable. Because of this assumption the creator will provide a ground-truth that contains a taint flow that does in fact not exist inside the benchmark case. This may lead to a false taint analysis tool evaluation.

Another issue that may happen due to an improper benchmark case selection is the possible over-adaptation of taint analysis tools to the used benchmark cases. If the evaluation of a taint analysis tool mostly relies on micro benchmark cases, the tool might struggle with the detection of complex taint flows. A taint analysis tool might be able to detect taint flows that contain reflection and taint flows that contain ICC individually, but it may struggle with taint flows that contain reflection and ICC combined. Therefore, it might happen that a taint analysis tool might get really high scores during the evaluation process, but still may struggle while analyzing real-world applications.

The second issue might in part be mitigated by the incorporation of more real-world benchmark cases into the evaluation process. However, the integration of real-world benchmark cases is often infeasible, because real-world applications almost never provide a corresponding ground-truth. Furthermore, the manual creation of real-world benchmark cases is also a very tedious task, since it usually takes a long time to create a single real-world Android application or the corresponding ground-truth. This process is even more error prone than the creation of micro

benchmark cases since the degree of complexity of a real-world benchmark case is far larger. Because of this, the evaluation of taint analysis tools has to mainly rely on the few real-world benchmark cases that are currently available [22]. This may also lead to an over-adaptation of taint analysis tools to the provided benchmark cases.

In order to help overcome these difficulties one possibility is to design a benchmark case generator which is able to automatically generate benchmark cases of arbitrary complexity and versatility. This chapter focuses on a conceptual design of such a generator.

3.1 Generator Requirements

A benchmark case generator has to fulfill multiple requirements to be able to help overcome the above mentioned issues. Above all it has to be able to generate proper Android applications in the form of APK files, since Android taint analysis tools require an APK file to perform a taint analysis on. To fulfill this requirement, first the generator needs to be able to generate all files that are required to compile an Android application. This means it has to generate the source code of the application, the Android manifest file and the layout file, since these are the minimum required files needed. Android apps can either be written in Java, Kotlin or C++ [1]. This thesis focuses on a generator concept that generates Java source code. Furthermore, the generator needs to be able to generate applications that contain various kinds of taint flows. This means that it needs to be able to produce taint flows that contain different *aspects* like, for example, ICC, reflective class loads or different kinds of data structures. Additionally, it needs to be able to generate arbitrary combinations of these aspects in a single taint flow. To fulfill this requirement a modular concept is needed that allows the insertion of different kinds of aspects into each taint flow the generated application contains. Additionally, the generator needs to be able to generate applications of arbitrary size and complexity, to be able to produce benchmark cases that are comparable to real-world applications. On this occasion it is also very important to automatically generate the corresponding ground-truth for each benchmark case, since a ground-truth is required in order to be able to use a benchmark case to properly evaluate an analysis tool. Another important requirement for the generator is a mean of extensibility. The generator should have the capability to offer the user a way to provide further aspects that can be contained inside the generated taint flows. This means that, for example, if the user wants to integrate a specific callback function from the Android activity lifecycle into the taint flow, he/she should be able to do so. To summarize this gives us the following requirements:

1. Generate proper Android applications in form of APK files
2. Generated applications need to contain various types of taint flows of arbitrary complexity
3. Generate a corresponding ground-truth for each application
4. The generator should be extendable

The next section illustrates a design concept that is able to fulfill these requirements.

3.2 Generator Overview

Figure 3.1 shows an overview of the processing pipeline of the benchmark case generator. The first input of the generator is a single *template* which denotes the generated benchmark case's starting structure, containing the structure for the source code, the Android manifest file and the layout files. The second input is a set of *modules*. The modules contain information about how the templates of the source code, the Android manifest file and the layout file (a single

layout file for all activities) have to be filled. Each module contains information like specified components, required permissions and the code snippet that should be inserted into the template. They are successively inserted into the template. This combination of the template and the modules is called *template/modules configuration* (TMC). The TMC determines the content of the benchmark case after the generation. In the first step of the pipeline, the selection of the template and modules is verified against a supplied grammar by the *verificator*. This step is needed in order to verify, for example, that there is exactly one template provided in the TMC and that the configuration is representing the right data flow branching structure, meaning that if a module requires a specific amount of follow-up modules, the needed amount is specified in the configuration. The verificator verifies that this amount of modules is always correct.

The grammar used by the verificator additionally enables fuzzing. This means that random, but always valid, TMCs can be generated automatically.

If the provided TMC is valid, the specified modules and the template are supplied to the *preprocessor*. The preprocessor takes care of Java language specific details like handling imports and multiple declarations of variables across the modules. Furthermore, it analyzes the supplied modules and generates a corresponding ground-truth that contains all taint flows, the location of each taint flow's source and sink, and whether the contained taint flow is a positive or a negative flow (see Section 2.2).

The preprocessed template and modules are then passed forward to the *source generator*. The source generator is responsible for inserting the modules into the template at the correct position. This step will generate Java source code files, as well as the corresponding Android manifest file and the layout file.

The generated files are then passed to a *build tool*. A build tool is a program that is capable of compiling Android applications. It is responsible for compiling an APK file out of the provided source files.

The outputs of the benchmark case generator are an APK file and its corresponding ground-truth in form of an *AQL-Answer* [44], which contains information about all taint flows contained in the generated benchmark case.

In the next sections each component, as well as the inputs and outputs, of the processing pipeline of the benchmark case generator will be described in detail.

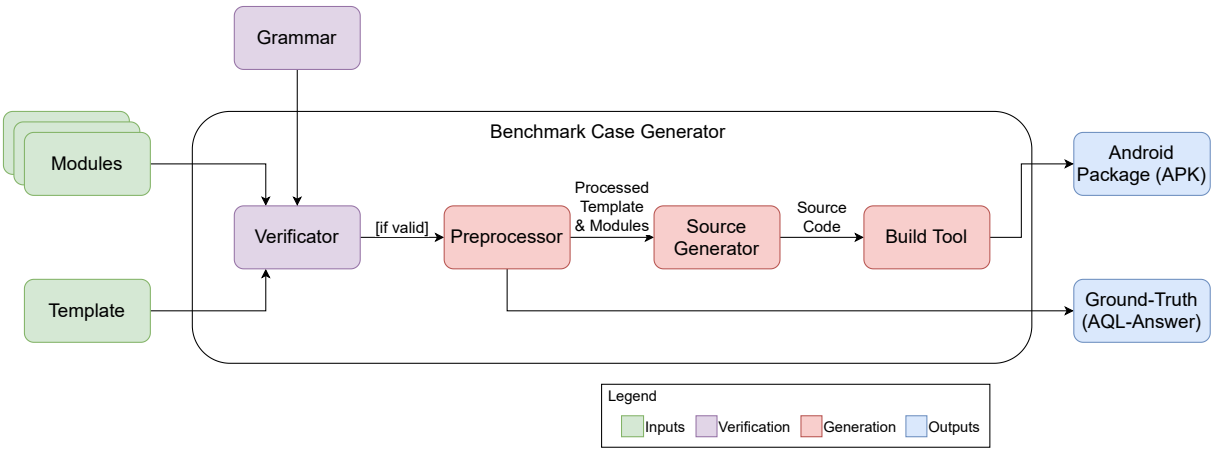


Figure 3.1: Overview of the processing pipeline of the benchmark case generator

3.3 Inputs

This section introduces the inputs of the benchmark case generator. It describes the template, as well as the modules in detail.

A *template* denotes the starting structure for every generated benchmark case. Each provided TMC starts with a template. A template $\mathcal{T}$ contains the following information:

$$\mathcal{T} = (S, M, L, F)$$

S denotes the structure of the source code of the starting component. M denotes the initial structure of the Android manifest file, while L represents the initial structure of the layout file. F describes the flow of the template.

The source code contains all necessary statements for the starting component, like the class definition, the initial method definition and required imports. Furthermore, it contains a placeholder which denotes where in the template the first module has to be inserted. Additionally, it contains multiple placeholders that optionally can be filled by code-snippets provided by modules.

The Android manifest file contains the declaration of the first component that is implemented in S. Furthermore, it contains two placeholders. The first placeholder indicates the location at which components that are newly introduced by a module have to be declared. The second placeholder indicates the location where permissions, as well as needed software or hardware features that are required by a module have to be declared.

The layout file contains the initial layout of the starting component. Even if the component is not an activity and therefore does not require a layout file, as it provides no user interface, the layout structure still has to be defined, in the case that a further module creates an activity which needs a layout file. Additionally, it contains a placeholder for the location at which view elements can be inserted by a module. This placeholder is necessary if a benchmark case has to be generated that contains some kind of user interaction. This way a button can be placed into the layout file and a callback function can be provided, that is executed, as soon as the button is pressed.

In order to generate a proper ground-truth that contains information about each taint flow, a declaration of the *flow* of the template is necessary. The flow information F is defined as

$$F = (className, methodSignature)$$

with $className$ denoting the class name of the component and $methodSignature$ denoting the signature of the method containing the placeholder for the next module insertion.

Modules are the core building blocks of the benchmark case generator. While the template denotes the starting point and the initial structure for the benchmark case to be generated, modules represent the content of the benchmark case. They specify the content of the application's source code, the content of the Android manifest file and the content of the layout file. Additionally, each module specifies a placeholder for the insertion location of the next module. A module $\mathcal{M}$ can be described as following:

$$\mathcal{M} = (T, C, A, F)$$

This means that each module contains four properties.

T describes the type of the module. T has the following value:

$$T = \text{Source} \vee \text{Bridge} \vee \text{Sink}$$

C defines the code that the module generates. It also contains created methods, classes and components.

A defines additional info that is necessary for the technical implementation, but not needed in order to explain the concept. It contains information like required imports, permissions and also component definitions, as well as view element definitions. This property can also be empty if there are no such requirements for the module.

F is a set that describes the flows of further modules. This property is mandatory and can not be empty, since it is necessary in order to generate a proper ground-truth. F can be defined as following:

$$F = \{(className_i, methodSignature_i, statementSignature_i, leaking_i)_i \mid 1 \leq i \leq |\text{Flows}|\}$$

|Flows| denotes the number of new branches a module creates. Each module creates at least one branch. However, some modules may create more than only a single branch. Therefore, each branch that is newly generated by the module needs to be tracked.

Like for the template, F contains the class name of the component, as well as the method signature of the method containing the next module placeholder. But in contrast to templates, this information may be empty inside a module. This is the case if the module does not alter the program flow by creating new methods or classes. If the program flow is moved to another method, but stays inside the same class as for the previous module, the $className$ field may be empty, but the $methodSignature$ field is still specified.

The $statementSignature$ field describes the signature of the statement, that interacts with the sensitive data. This is especially the case for source and sink modules, where the sensitive data is introduced into the application and leaked to the outside world.

The *leaking* field is a value that describes whether the module changes the current taint flow, that it has been placed in, to a negative one. The value of the *leaking* field is 1, if the module does not alter the taint flow. If the module sanitizes the taint flow by, for example, replacing the sensitive data with some insensitive data, the value of *leaking* is 0. If the module introduces unreachable code, the value of the *leaking* field is -1. This differentiation is necessary, since after the insertion of a module that inserts unreachable code, the taintflow can never be rendered into a positive one again, even if a new source is introduced.

The following sections will specify the available types of modules, present some examples of the different module types and demonstrate how the application from the running example (see Figure 2.4) can be replicated by this template and modules concept.

3.3.1 Template

The template that is used to generate a benchmark case which is semantically equivalent to the hand-crafted benchmark case from the running example (see Figure 2.4) is called $\mathcal{T}_{\text{Basic}}$ and is defined as following:

$$\mathcal{T}_{\text{Basic}} = (S_{\text{Basic}}, M_{\text{Basic}}, L_{\text{Basic}}, F_{\text{Basic}})$$

Listing 3.1 shows the content of S_{Basic}. S_{Basic} contains the structure of an activity. It contains all necessary imports that each Android activity requires, the class declaration of the activity `MainActivity`, the `onCreate` callback, which is executed as soon as the activity is started and also the declaration of the variable `sensitiveData` that will hold the sensitive data throughout the application. Furthermore, it contains exactly five placeholders that indicate the positions for the following module that is inserted. Placeholders are depicted in green color and also by being surrounded by double curly brackets.

Some modules may require additional imports or global fields that have to be declared, therefore an *imports* and a *globals* placeholder are provided. Other modules might come with their own methods and therefore need to be able to insert them into the *methods* placeholder. In order to be able to generate benchmark cases containing, for example, ICC or reflection, a *classes* placeholder, where new components or classes are inserted into is also necessary. The *module* placeholder continues the current program flow. The code statements that are inserted into the *module* placeholder are executed first.

```
import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
{{ imports }}

public class MainActivity extends Activity {
  {{ globals }}

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);
    String sensitiveData = "";
    {{ module }}
  }
  {{ methods }}
}
{{ classes }}
```

Listing 3.1: Source code part of the template for the running example (see Figure 2.4) (S_{Basic})

Listing 2.1 shows the content of M_{Basic}. It contains the structure of the Android manifest file. Furthermore, it already contains the declaration of the component `MainActivity` as the starting component, which should be launched, as soon as the application is started. Additionally, it contains three placeholders. The *permissions* placeholder indicates the location where necessary permissions and required hardware or software features have to be inserted. The *components* placeholder indicates the insertion location for further Android components. If, for example, a module creates a new activity to integrate ICC into the benchmark case, the newly created activity needs to be declared there.

```
<manifest ...>
  {{ permissions }}

  <application>
    <activity android:name="MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    {{ components }}

  </application>
```

```
</manifest>
```

Listing 3.2: Android manifest part of the template for the running example (see Figure 2.4) (M_{Basic})

Listing 3.3 shows the contents of L_{Basic}. It contains the layout structure of the generated benchmark case. The `RelativeLayout` tag just serves as a container for possible future views that are inserted by modules. L_{Basic} can contain any arbitrary `ViewGroup` element, which is able to serve as a container for views, as a root element. Besides the `ViewGroup`, L_{Basic} contains one placeholder: the *views* placeholder. It indicates the insertion location for objects of type `View` like, for example, buttons.

```
<RelativeLayout ...>
  {{ views }}
</RelativeLayout>
```

Listing 3.3: Layout part of the template for the running example (see Figure 2.4) (L_{Basic})

The last element of $\mathcal{T}_{\text{Basic}}$ is F_{Basic}. F_{Basic} has the following content:

$$F_{\text{Basic}} = (\texttt{MainActivity}, \texttt{void onCreate(android.os.Bundle)})$$

This means that the class name of the component containing the *module* placeholder is `MainActivity` and the signature of the method containing the *module* placeholder is `void onCreate(android.os.Bundle)`. This information is needed in order to determine the exact taint flows that are contained in the generated benchmark case. This means that if, for example, the first module that is inserted into the template is a source module, the exact location of this source can be written inside the corresponding ground-truth.

3.3.2 Modules

Modules can come in different forms. However, there are three main types of modules:

1. Source modules
2. Sink modules
3. Bridge modules
   - 3.1. Basic modules
   - 3.2. Branching modules
   - 3.3. Invoke modules
   - 3.4. Sanitizing modules

Source modules contain a statement that can introduce sensitive data into the application. They represent a source inside a taint flow. The first module that is inserted into the template is usually a source module. But this is not always necessarily the case.

Sink modules contain statements that have the capability to leak sensitive information to the outside world. They represent a sink inside a taint flow. There are many possibilities how such a sink module could look like. The sensitive data could be sent via SMS, via email or uploaded to a server and therefore be leaked to the outside world.

Each taint flow concludes with a sink module. This means that each taint flow which is started from a source module has to be concluded with a sink module in order to form a proper taint flow. However, this conclusion with a sink module is not mandatory.

Bridge modules describe the data flow of the generated benchmark case. They usually are placed in between a source module and a sink module. While each taint flow typically contains only one source module and one sink module, a taint flow can contain an arbitrary number of bridge modules. Bridge modules are used to integrate various aspects that obfuscate the taint flow between a source and a sink. This means that bridge modules could, for example, implement different kinds of data structures, various callbacks or ICC. However, bridge modules may also be placed before or after a source or a sink module to further inflate the benchmark case.

The main type of the module defines the value of the type property T of $\mathcal{M}$.

Bridge modules can be further divided into four sub-types:

Basic modules do not alter the program flow in any way. This means that the next module will be inserted in the exact same method of the exact same component in which the basic module has been inserted. Basic bridge modules typically are used to pass the sensitive data through some kind of data structure like an array or a list. However, they can also be used to implement, for example, reflective class loads.

Branching modules branch the current program flow into two or more program flows. This means that there are multiple follow-up modules that have to be inserted next. Therefore, a branching module has to specify more than just one placeholder for the insertion location for the next module. They have to specify an additional placeholder for each additional program branch that should be able to be continued by a follow-up module. Branching modules are typically used to implement different control structures like if and else blocks, where one module is placed into the if branch and another one is placed into the else branch. But they can also be used to implement multiple conditional method calls, where each method handles another following module. These methods may even be contained inside different classes or components.

Invoke modules alter the program flow, but do not create multiple program flow branches. This means that the invoke module creates a new method or even a new component and shifts the program flow to this newly created method or component. Invoke modules are used to implement ICC. They can also be used to implement callback functions that are, for example, invoked once a user event happens.

Sanitizing modules are special modules that are responsible for sanitizing the taint flow. This means that the sanitizing module replaces the sensitive data that is contained inside a variable with something that is not sensitive. Sanitizing modules therefore render a positive taint flow into a negative one. Modules that introduce unreachable code into the generated benchmark case are also considered sanitizing modules.

To generate a benchmark case that is semantically equivalent to the hand-crafted benchmark case from the running example (see Figure 2.4) three modules are needed in addition to the template, described in the previous section. The first module can be defined as following:

$$\mathcal{M}_{\text{IMEI}} = (\text{Source}, C_{\text{IMEI}}, A_{\text{IMEI}}, F_{\text{IMEI}})$$

This module represents a basic source module that reads the device's IMEI number.

Listing 3.4 shows the content of C_{IMEI}. This code snippet is responsible for reading the device's IMEI number and storing it inside a variable. Furthermore, it also contains the placeholder for the next module insertion (). This means that the next module will be inserted at the placeholders location. Since this module does not require global fields, additional methods, further classes, component or views, the according fields are empty.

module:
```
TelephonyManager tm = (TelephonyManager) getSystemService(
    Context.TELPHONY_SERVICE);
sensitiveData = tm.getDeviceId();
{{ module }}
```
Listing 3.4: Content of C_{IMEI}

In order to be able to read the device's IMEI number, one import and one permission from the Android system are required. The `READ_PHONE_STATE` permission is needed in order for the application to be able to access the IMEI number of the device. Furthermore, the `android.telephony.TelephonyManager` class has to be imported, since it provides the necessary capabilities to access the device's IMEI number. Both of these pieces of information are contained inside A_{IMEI}

The last field, F_{IMEI}, describes the flow of the module:

$$F_{\text{IMEI}} = \{(\epsilon, \epsilon, \texttt{...TelephonyManager: java.lang.String getDeviceId()}, 1)\}$$

M_{IMEI} is a basic module and creates no branching flows, therefore F_{IMEI} contains only one element. Since the module neither creates a new class nor a new method where the program flow is shifted to, the first two fields are just empty. The third field describes the signature of the method that introduces the IMEI number into the application. The fourth field describes whether the sensitive data is sanitized by this module or not. In module $\mathcal{M}_{\text{IMEI}}$ this is not the case and therefore the value is 1.

The second module $\mathcal{M}_{\text{Array}}$ that is needed can be defined as following:

$$\mathcal{M}_{\text{Array}} = (\text{Bridge}, C_{\text{Array}}, A_{\text{Array}}, F_{\text{Array}})$$

$$A_{\text{Array}} = \varnothing$$

This module represents a basic bridge module that puts the sensitive data into an array and retrieves it afterwards. Since arrays are part of the standard library of Java, no further imports are needed. Furthermore, arrays do not require any permissions from the Android system and therefore there are also no needed permissions specified. Views are also not a part of the module. This means that the property A_{Array} is empty. There are also no new components, methods or global fields required. Therefore, only content for the *module* placeholder has to be specified.

Listing 3.5 lists the content of C_{Array}. This code snippet is responsible for creating an array with ten slots and then putting the sensitive data into the slot with index 5. Slot 4 of the array is filled with data that is not sensitive. Afterwards the sensitive data is retrieved from the array. Finally, a placeholder for the insertion location for the next module is provided. This module can be used as an indicator during the evaluation process, whether a taint analysis tool is able to differentiate between different array indexes.

module:
```
String[] array = new String[10];
array[5] = sensitiveData;
array[4] = "unsensitive data";
sensitiveData = array[5];
{{ module }}
```
Listing 3.5: Content of C_{Array}

The flow of the module is described by F_{Array}:

$$F_{\text{Array}} = \{(\epsilon, \epsilon, \epsilon, 1)\}$$

Module $\mathcal{M}_{\text{Array}}$ is not a branching module, therefore there is only a single element in F_{Array}. Module $\mathcal{M}_{\text{Array}}$ does not alter the flow of the application. There is no class or method that is created by this module. Additionally, this module has no statement in it that introduces or leaks sensitive data, therefore there is no statement signature that needs to be specified. Since the sensitive data is not sanitized inside this module, the last value of the flow description is 1.

The final module $\mathcal{M}_{\text{SMS}}$ that is needed to generate a benchmark case containing the taint flow from the running example (see Figure 2.4) can be defined as following:

$$\mathcal{M}_{\text{SMS}} = (\text{Sink}, C_{\text{SMS}}, A_{\text{SMS}}, F_{\text{SMS}})$$

$\mathcal{M}_{\text{SMS}}$ is a basic sink module that leaks the sensitive data via SMS to the outside world.

Listing 3.6 shows the content of C_{SMS}. This code snippet is responsible for sending the sensitive data via SMS to another device and therefore leaking the sensitive data. A generated benchmark case could end with this module insertion, as a full taint flow is present in the application, once a source and sink module have been provided. However, there is still a placeholder for the next module insertion provided, in case the user of the generator would like to continue the taint flow.

In order to be able to send an SMS to another device the `SEND_SMS` permission is required from the Android system. Furthermore, to be able to utilize the SMS sending capability the `android.telephony.SmsManager` class has to be imported into the application. This means that both of these pieces of information are contained inside A_{SMS}

module:

```
SmsManager sm = SmsManager.getDefault();
sm.sendTextMessage("+49 12345", null, sensitiveData, null, null);
{{ module }}
```

Listing 3.6: Content of C_{Array}

The flow F_{SMS} of the module $\mathcal{M}_{\text{SMS}}$ is described by:

$$F_{\text{SMS}} = \{(\epsilon, \epsilon, \texttt{android.telephony.SmsManager: void sendTextMessage(...)}, 1)\}$$

Like the previous two modules, $\mathcal{M}_{\text{SMS}}$ is not a branching module and also does not generate any methods or classes and therefore does not alter the program flow. However, the statement `sendTextMessage()` is used to leak the sensitive data to the outside world, and therefore its signature is specified in F_{SMS}. Furthermore, the module does not sanitize the sensitive data and therefore the last field of the flow description is 1 as well.

Module Examples

Since the running example only uses a basic bridge module which is usually the simplest type of modules, this section will present some more module examples in order to showcase the differences between each type. Furthermore, it will also present a more complex basic module in order to show that basic modules may also be more complex than $\mathcal{M}_{\text{Array}}$.

- **Invoke-Example:** An example for an invoke module is the module $\mathcal{M}_{\text{ICC}}$ which employs ICC into the generated application. It can be defined as following:

$$\mathcal{M}_{\text{ICC}} = (\text{Bridge}, C_{\text{ICC}}, A_{\text{ICC}}, F_{\text{ICC}})$$

A_{ICC} contains the for the module required imports, as well as the component definition of the newly created activity that has to be inserted into the Android manifest file.

Listing 3.7 lists the content of C_{ICC} of $\mathcal{M}_{\text{ICC}}$. It contains two separate code snippets. One of them is inserted into the *module* placeholder, while the other one is inserted into the *classes* placeholder of the template. This module alters the program flow of the application. It moves the flow to the newly generated activity `NextActivity`. The altering of the program flow requires a new set of placeholders, as further modules have to be inserted into new positions that are generated by the invoke module. Because of this, in addition to the *module* placeholder, the previous *globals* and *methods* placeholders are removed and new placeholders are provided. Furthermore, a new *classes* placeholder is provided, so further modules can insert new classes.

F_{ICC} can be defined as following:

$$F_{\text{ICC}} = \{(\texttt{NextActivity}, \texttt{void onCreate(android.os.Bundle)}, \epsilon, 1)\}$$

Since the program flow changes to the `onCreate` method of the class `NextActivity`, this has to be mentioned in the flow property F_{ICC} of $\mathcal{M}_{\text{ICC}}$.

module:

```
Intent i = new Intent(this, NextActivity.class);
i.putExtra("leak", sensitiveData);
startActivity(i);
```

classes:

```
public class NextActivity extends Activity {
  {{ globals }}

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);
    Intent i = getIntent();
    String sensitiveData = i.getStringExtra("leak");
    {{ module }}
  }
  {{ methods }}
}
{{ classes }}
```

Listing 3.7: Content of C_{ICC}

- **Branching-Example:** One example for a branching module is the module $\mathcal{M}_{\text{RandomIfElse}}$ which creates an if-statement with one if- and one else-branch. It can be defined as following:

$$\mathcal{M}_{\text{RandomIfElse}} = (\text{Bridge}, C_{\text{RandomIfElse}}, A_{\text{RandomIfElse}}, F_{\text{RandomIfElse}})$$

$$A_{\text{RandomIfElse}} = \varnothing$$

$A_{\text{RandomIfElse}}$ is empty because no additional information, like for example imports or permissions are required by $\mathcal{M}_{\text{RandomIfElse}}$.

Listing 3.8 lists the content of $C_{\text{RandomIfElse}}$ of $\mathcal{M}_{\text{RandomIfElse}}$. The module generates a random number between 1 and 0 and executes the if-branch, if the number is bigger than 0.5 and executes the else-branch if the random number is less than or equal to 0.5. If the else-branch is executed, a method called `handleElse` is executed and the sensitive data is passed to it as a parameter. (The out-sourcing of the else-branch into the `handleElse` method is only used to demonstrate the capability of one branch altering the program flow, while the other branch not altering it. A module directly containing the *module* placeholder inside the else-branch would be semantically equivalent to $\mathcal{M}_{\text{RandomIfElse}}$.) This means that each branch has an execution probability of 50%. Because there are two possible branches that can be executed, the program flow is branched and requires multiple new slots for module insertions. One module can be inserted into the if-branch and one module can be inserted into the method invoked by the else-branch. Because of the presence of branches in the module, $F_{\text{RandomIfElse}}$ has to contain two elements, since each *module* placeholder requires its own flow description:

$$F_{\text{RandomIfElse}} = \{(\epsilon, \epsilon, \epsilon, 1)_1, (\epsilon, \texttt{void handleElse(java.lang.String)}, \epsilon, 1)_2\}$$

The first *module* placeholder does not alter the program flow, therefore the first element of $F_{\text{RandomIfElse}}$ only contains empty values, besides the leaking attribute. However, the second *module* placeholder does alter the program flow, since it is placed inside the `handleElse` method. This altering of the program flow can be seen in the second element of $F_{\text{RandomIfElse}}$.

module:

```
if (Math.random() > 0.5) {
  {{ module }}
} else {
  handleElse(sensitiveData);
}
```

methods:

```
public static void handleElse(String data) {
  String sensitiveData = data;
  {{ module }}
}
{{ methods }}
```

Listing 3.8: Content of $C_{\text{RandomIfElse}}$

- **Sanitization-Example:** The last remaining type of bridge modules is a sanitization module. Sanitization modules are important for generating negative taint flows. In doing so it is important to fully sanitize a taint flow by using a sanitization module, as partial sanitizations like, for example, replacing the sensitive data by a string with the same length or only partially replacing it would still allow for conclusions on the sensitive data.

Modules that provide the above mentioned functionality would therefore not fall into the category of sanitization modules.

One example for a sanitization module is the module $\mathcal{M}_{\text{Sanitize}}$ which can be defined as following:

$$\mathcal{M}_{\text{Sanitize}} = (\text{Bridge}, C_{\text{Sanitize}}, A_{\text{Sanitize}}, F_{\text{Sanitize}})$$
$$A_{\text{Sanitize}} = \varnothing$$

Listing 3.9 shows the content of C_{Sanitize}. $\mathcal{M}_{\text{Sanitize}}$ is a very simple module that only consists of a single statement. It overwrites the variable containing the sensitive data and therefore fully sanitizes the taint flow by not allowing any conclusions on the sensitive data. The content of F_{Sanitize} is as following:

$$F_{\text{ICC}} = \{(\epsilon, \epsilon, \epsilon, 0)\}$$

The program flow is not altered by $\mathcal{M}_{\text{Sanitize}}$, therefore the flow property is mostly empty. However, the module sanitizes the taint flow, therefore the leaking attribute of the element in F_{ICC} has a value of 0.

module:
```
sensitiveData = "untaint";
{{ module }}
```

Listing 3.9: Content of C_{Sanitize}

$\mathcal{M}_{\text{Sanitize}}$ is a very basic sanitization module that may be used in most of the use cases where a full sanitization of the taint flow is required. However, there are other sanitization options than just fully sanitizing the taint flow by replacing the contents of the variable containing the sensitive data. A taint flow can be sanitized by introducing unreachable code into it. If a source or a sink cannot be reached inside a taint flow, the taint flow is sanitized as well.

- **More complex Basic-Example:** Often, basic modules may create additional methods or classes that do not alter the program flow of the application. The module $\mathcal{M}_{\text{Recursion}}$ is an example for this behavior. $\mathcal{M}_{\text{Recursion}}$ is a basic module that incorporates recursion into the generated benchmark case. It can be defined as following:

$$\mathcal{M}_{\text{Recursion}} = (\text{Bridge}, C_{\text{Recursion}}, A_{\text{Recursion}}, F_{\text{Recursion}})$$
$$A_{\text{Recursion}} = \varnothing$$

Listing 3.10 shows the content of $C_{\text{Recursion}}$. $\mathcal{M}_{\text{Recursion}}$ creates a method called `rec` that recursively calls itself until a provided depth has been reached and then returns a supplied value. At first, the sensitive data is temporarily stored inside another variable and the `sensitiveData` variable is sanitized. Then the recursive function is called with a depth parameter of 1000 and the temporarily stored sensitive data. After 1000 recursive calls the sensitive data is returned. This value is then assigned to the previously sanitized variable `sensitiveData`. Since the program flow has not been altered by $\mathcal{M}_{\text{Recursion}}$ the value of $F_{\text{Recursion}}$ looks like the following:

$$F_{\text{Recursion}} = \{(\epsilon, \epsilon, \epsilon, 1)\}$$

module:
```
String recData = sensitiveData;
sensitiveData = "";
sensitiveData = rec(1000, recData);
{{ module }}
```

methods:
```
public static String rec(int depth, String data){
  if (depth > 0) {
    return rec(depth - 1, data);
  } else {
    return data;
  }
}
{{ methods }}
```

Listing 3.10: Content of $C_{\text{Recursion}}$

A list of all provided templates and modules together with a short description can be found in Appendix A.1.

3.4 Verification

This section describes the capabilities and responsibilities of the verificator component in detail and presents the grammar that is used by it.

3.4.1 Grammar

In order to determine whether a TMC is valid, a *grammar* is employed. Listing 3.11 shows the grammar that represents the set of valid TMCs of the benchmark case generator. Symbols that are surrounded by angle brackets denote non-terminals, while symbols inside quotes denote terminals. The grammar starts at a `<start>` symbol which can be derived to a `<template>` and a `<module>` symbol. The `<template>` symbol can be directly derived to a non-terminal that indicates which template is used in the TMC. The three dots indicate that all available templates are listed there and therefore the `<template>` symbol can be derived to any of these templates. There is no empty rule for the `<template>` symbol. This signals that a template has to be provided as the first part of the TMC in order to be a valid configuration.

The `<module>` symbol has three possible derivations. Based on the program flow that a module can provide, it can be derived to the `<linear>` symbol, if there are no branches in the module or to a `<branching>` symbol, if there is a branch in the program flow present in the module. Furthermore, a module can also be derived to an empty word, meaning that there is no necessity to provide a module at every position a `<module>` symbol is present. A linear module only needs one follow-up module and therefore the `<linear>` symbol can directly be derived to a `<linearModule>` symbol, which in turn can then be derived to a non-terminal representing the module with linear program flow. In addition to the `<linearModule>` symbol, another `<module>` symbol is derived, which then can be derived to the follow-up module or, if the end of the TMC is reached, to an empty word. A branching module needs multiple follow-up modules, depending on the amount of branches created by the module. Therefore, the `<branching>` symbol can be derived to the symbol `<2Branches>` if it contains two branches, to the symbol `<3Branches>` if

it contains three branches and so on. Since there are no modules with infinite branches, the amount of required follow-up modules is finite and therefore the amount of possible derivations is finite. The amount of branches a module creates and therefore the amount of follow-up modules required, is denoted by parenthesis pairs which each contain a `<module>` symbol. This means that a module that creates two branches comes with two parenthesis pairs, a module that creates three branches comes with three parenthesis pairs and so on. Because of this precise amount of follow-up modules, each needed amount of parenthesis pairs has to be explicitly defined in the grammar instead of allowing the generation of an arbitrary amount of parenthesis pairs after a branching module. This is important because of the fuzzing capability of the benchmark case generator, since the generation of an arbitrary amount of parenthesis pairs could lead to a wrong TMC. This way there could be, for example, a generation of three parenthesis pairs even though the supplied branching module only creates two branches. This would lead to an invalid TMC.

Two branches can, for example, be created by an *IfElse* module that creates an If-statement with one if- and one else-case. Three branches can be created by a *3Switch* module that creates a Switch-Statement containing three different cases.

The grammar is specifically designed to be easily extensible, in order to allow for a proper extensibility of the generator. If the user creates his/her own template or module he/she can easily add it to the provided grammar with as few needed changes as possible. Because of this extensibility aspect, the grammar is supplied to the benchmark case generator as an additional input. However, this input is only changed, whenever the user adds another module or template.

```
<start>          ::= <template> <module>
<module>         ::= <linear> | <branching> | ϵ
<linear>         ::= <linearModule> <module>
<branching>      ::= <2Branches> | <3Branches> | ...
<2Branches>      ::= <2BranchModule> "(" <module> ")" "(" <module> ")"
<3Branches>      ::=
   <3BranchModule> "(" <module> ")" "(" <module> ")" "(" <module> ")"
...
<template>       ::= "Basic" | ...
<linearModule>   ::= "IMEI" | "Array" | "SMS" | ...
<2BranchModule>  ::= "IfElse" | ...
<3BranchModule>  ::= "3Switch" | ...
...
```

Listing 3.11: Grammar that is used by the verificator

3.4.2 Verificator

The *verificator* is the first component of the processing pipeline of the benchmark case generator which is depicted in Figure 3.1. It serves two distinct purposes: (1) verifying that the provided TMC is a valid one and (2) generating random, but valid TMCs in order to enable fuzzing capabilities.

The verificator uses the grammar described in Section 3.4.1 to verify whether the provided TMC is valid or not. It does so by checking if there is a possible derivation that leads to the provided TMC. If there is one, the provided configuration is valid, if not it is invalid. Figure 3.2 shows the derivation tree for the TMC $\mathcal{C}$ with

$$\mathcal{C} = (\text{Basic}, \text{IMEI}, \text{Array}, \text{SMS})$$

which is used in the running example (see Figure 2.4). The existence of such a derivation tree for a TMC implies that this configuration is valid. If there is no derivation tree for a TMC, the configuration is invalid.

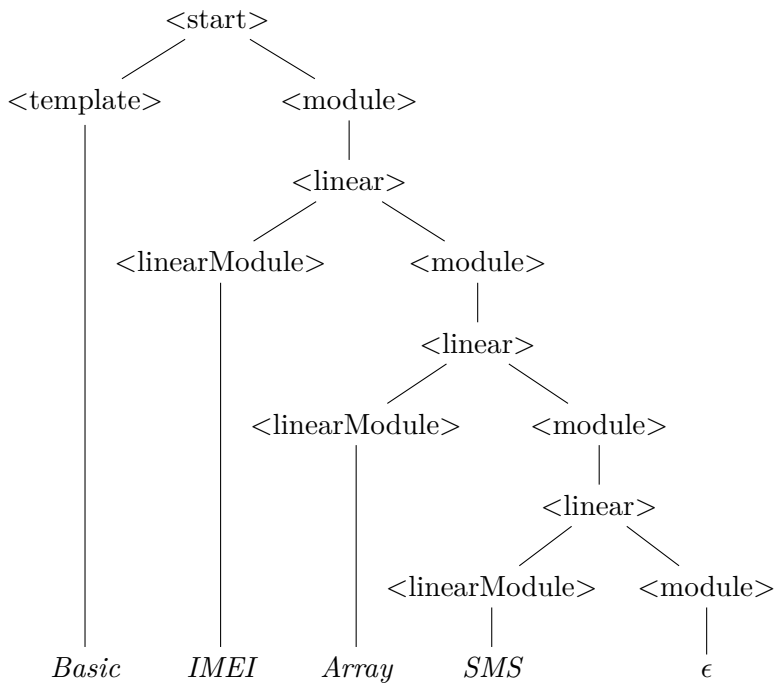


Figure 3.2: Derivation tree for the TMC $\mathcal{C}$

The second capability, that the verificator in combination with the grammar provides, is fuzzing. Having this grammar at hand allows to apply grammar-based fuzzing, which is a significantly more efficient approach than plain fuzzing. In order to generate only valid TMCs, the verificator that is used as a fuzzer starts at the `<start>` symbol and chooses a random derivation. Since there is only one possible derivation, this one will be chosen. Afterwards there are multiple possibilities. Therefore, one random derivation of the possible ones will be chosen. This process is continued until there are only terminals left. At this point the derivation process is finished and cannot be further continued.

Every template and module that is available to the generator can be part of the generated configuration. This also means that not every configuration contains a taint flow. Configurations that, for example, only use bridge modules are possible.

In order to generate a proper benchmark case that is desired by the user, only configurations which do contain modules that are defined by the user are continued throughout the processing pipeline of the benchmark case generator. Other generated TMCs are just discarded.

3.5 Generation

This section focuses on the generation of the necessary source code and resource files and the subsequent build process of the Android application. It will introduce the three necessary components: the preprocessor, the source generator and the build tool, in detail (see overview in Figure 3.1).

3.5.1 Preprocessor

The *preprocessor* is a component that is partly responsible for technical aspects. It preprocesses the template and each module before inserting the modules into the template.

The first responsibility of the preprocessor is conflict resolving. Java is a programming language that does not allow the declaration of multiple variables inside the same scope with the same identifier. But this scenario may happen if, for example, the same module that does not change the scope of the application, but declares a variable, is inserted twice, consecutively. One example for this is the module $\mathcal{M}_{\text{Array}}$ with the generated code snippet from Listing 3.5. It creates an array with the identifier `array` and assigns the sensitive data to it. Having this module twice, consecutively, in the TMC leads to invalid Java source code, since the variable `array` is declared twice inside the same scope. This invalid source code cannot be compiled into an Android application. But not only variables with the same identifier can create such invalid Java source code. Method names and class names that are created by a module can also be declared twice and therefore invalidate the source code.

In order to prevent this situation from happening, the preprocessor adds a unique numeral identifier to each identifier. This means that each identifier has a number appended to it, which is incremented whenever a new identifier has been found by the preprocessor. The only exception to this rule is the variable `sensitiveData` which contains the sensitive data. Since each module needs to be able to access the variable containing the sensitive data, no unique numeral identifier is appended to this variable.

The second responsibility of the preprocessor is the handling of required permissions and imports. It ensures that the specified, required imports and permissions are requested and that there are no duplicate requests.

The third responsibility of the preprocessor is to keep track of branches inside the source code and the according placeholders for the next module inside each branch. Whenever a branch is created by inserting a branching module, the current source code contains multiple placeholders for the next module insertion location. This number of branches, and the number of open placeholders resulting from this, can be arbitrarily high, since branching modules can be nested arbitrarily deep, creating many new branches. In order to be able to determine which module has to be inserted into which placeholder, the preprocessor attaches a unique identifier to each placeholder, as well as to the belonging module that needs to be inserted at the placeholder's location, whenever a new program flow branch is created. This way the source generator knows into which placeholder each module has to be inserted.

The most important responsibility of the preprocessor is the generation of the corresponding ground-truth for the generated benchmark case. It does so by generating a graph containing the flow information of the template and each module. Since the template and each module are processed by the preprocessor, the preprocessor is able to exactly track every taint flow inside the application that is generated by the benchmark case generator. Each template and module contains a property F, that specifies the flow of the corresponding template or module and also the statements that are used to process the sensitive data throughout the application. Together with the type information T of each module, the preprocessor is able to track every flow that stretches from a source to a sink and therefore represents a taint flow. Furthermore, with the value contained in F that describes whether the module sanitizes the taint flow or not, the preprocessor can determine if the taint flow is a positive or a negative one. If a single module along the taint flow sanitizes the taint flow, the whole taint flow becomes a negative one.
Figure 3.3 shows the information provided by each individual template and module of the TMC, the generated ground-truth graph after the preprocessing of the TMC $\mathcal{C}$ and the corresponding ground-truth of the running example (see Figure 2.4). The upper part of the Figure shows the template and each module of the TMC before the preprocessing step. It shows the type and

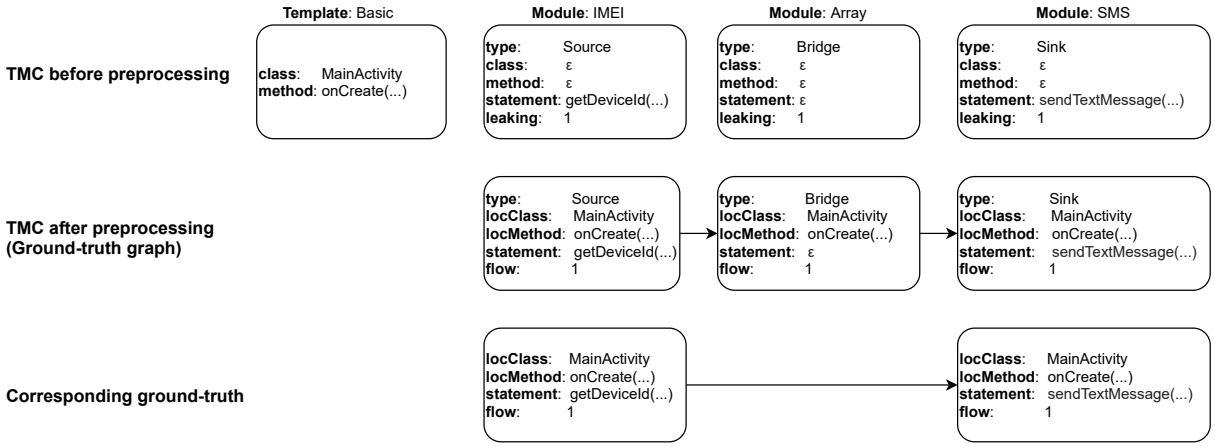


Figure 3.3: TMC before and after preprocessing of the running example (see Figure 2.4)

flow information the template and each module provide (normally the statement and method signatures are provided, but to enhance readability in the figure, the identifiers are used instead). The flow information of each template and module is explained in detail in the Sections 3.3.1 and 3.3.2.

The middle section of the figure shows the generated ground-truth graph after the template and each module have been preprocessed. Each node $\mathcal{N}$ in the graph represents one module and contains the following information:

$$\mathcal{N} = (type, locClass, locMethod, statement, flow)$$

The properties *type* and *statement* are directly adopted from the module that the node represents. The properties *locClass* and *locMethod* indicate the position in the source code where the corresponding module that is represented by the node is located at after insertion. The property *flow* indicates whether there was a sanitization of the sensitive data beforehand or not. The *flow* property of each node inside the graph is not equal to the *leaking* property of the module. If the data has been sanitized before (insertion of a sanitization module with a *leaking* value of 0), the value of *flow* for all following modules will change to 0 until the end of the path is reached or a source introduces new sensitive data, changing the value back to 1 for all following nodes.

The first node in the ground-truth graph represents the module $\mathcal{M}_{\text{IMEI}}$. It is a source module that introduces sensitive data into the application. Since it is the first module, it is located at the class and method specified by the template $\mathcal{T}_{\text{Basic}}$. The second node represents the bridge module $\mathcal{M}_{\text{Array}}$. The previous module in the graph, $\mathcal{M}_{\text{IMEI}}$, does not change the insertion location of the module to another class or method. Therefore, the previously by $\mathcal{T}_{\text{Basic}}$ specified location is further propagated. The same holds for the sink module $\mathcal{M}_{\text{SMS}}$. Neither $\mathcal{M}_{\text{Array}}$ nor $\mathcal{M}_{\text{IMEI}}$ provide a new location, therefore the location specified by $\mathcal{T}_{\text{Basic}}$ is further propagated. None of the used modules sanitizes the taint flow, therefore the value of *flow* continues to be 1.

The lower section of the figure shows the ground-truth that can be generated from the ground-truth graph. It can easily be generated by traversing the ground-truth graph and directly connecting source to sink nodes.

The ground-truth graph is not always linear. It can also come in form of a tree. Figure 3.4 shows a more complex example of a ground-truth graph in form of a tree. It shows a benchmark case that retrieves the device's IMEI number and then splits the program flow with a branching

module that creates an if-statement with one if- and one else-branch. The first branch uses ICC to create a new activity called `NextActivity` and propagates the sensitive data to it, where it is subsequently leaked via SMS. This means that the module $\mathcal{M}_{\text{ICC}}$ moves the program flow and therefore the insertion location for the follow-up module to the `onCreate()` method of the class `NextActivity`. This can be observed in the properties of the node representing $\mathcal{M}_{\text{SMS}}$ in the upper branch.

The second branch continues the flow with the sanitizing module $\mathcal{M}_{\text{Sanitizer}}$ before leaking the data via SMS. However, since $\mathcal{M}_{\text{Sanitizer}}$ sanitized the taint flow, it becomes a negative taint flow. Therefore, there is no leak of sensitive data, which is indicated by the propagated value of *flow* at the sink module.

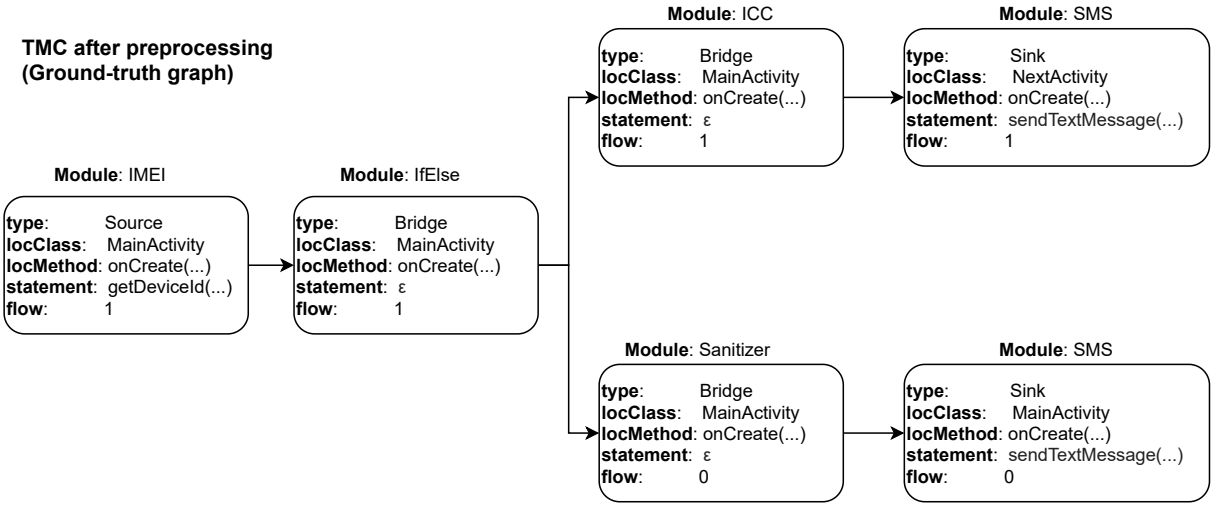


Figure 3.4: Ground-truth graph of a more complex benchmark case

Every path inside the ground-truth graph that starts at a node representing a source module and ends at a node representing a sink module (the source node has to come before the sink node), represents a taint flow. The value of *flow* at a node representing a sink module indicates whether the taint flow is a positive or a negative one.

3.5.2 Source Generator

The *source generator* is responsible for inserting the modules into the template. In order to generate proper source code for a valid Android application, the modules have to be inserted into the template one by one in the correct order. The correct order for the insertion process is the order of the modules that is specified in the TMC, from left to right. In case that there are branching modules present in the TMC, the modules are inserted in a breath-first manner.

Whenever a module $\mathcal{M}$ is inserted into a template $\mathcal{T}$, the code snippets provided by $\mathcal{M}$ are inserted into the corresponding placeholders in $\mathcal{T}$. Additionally, all placeholders that are provided by $\mathcal{T}$ are replaced by the placeholders provided by $\mathcal{M}$. In case that placeholders in $\mathcal{M}$ are left empty, the previous placeholders are retained. After the insertion of module $\mathcal{M}$ into template $\mathcal{T}$ a new template $\mathcal{T}_1$ is created in which the next module is inserted into. This process continues until all modules have been inserted. Once the last module in the TMC has been inserted, a cleanup of all remaining placeholders is performed. Furthermore, all newly created public classes are split into their own source file, since this is required in order to be able to compile a valid Android application.

The following section describes the insertion process to generate the benchmark case from the running example 2.4. Listing 3.12 shows the source code part of the template $\mathcal{T}_1'$ after module $\mathcal{M}_{\text{IMEI}}$ is inserted into template $\mathcal{T}_{\text{Basic}}$. The inserted parts are highlighted in green color. The code snippet contained in module $\mathcal{M}_{\text{IMEI}}$ is now part of the new template $\mathcal{T}_1'$. Furthermore, the location of the next *module* placeholder changed, since it was replaced by the placeholder in $\mathcal{M}_{\text{IMEI}}$. All other placeholders have been retained, since $\mathcal{M}_{\text{IMEI}}$ provided no further content for the other placeholders.

```
...
public class MainActivity extends Activity {
{{ globals }}

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);
    String sensitiveData = "";

    TelephonyManager tm = (TelephonyManager) getSystemService(
        Context.TELPHONY_SERVICE);
    sensitiveData = tm.getDeviceId();
    {{ module }}

  }
  {{ methods }}
}
{{ classes }}
```

Listing 3.12: $\mathcal{T}_1'$: After the insertion of $\mathcal{M}_{\text{IMEI}}$ into $\mathcal{T}_{\text{Basic}}$

Listing 3.13 shows the source code part of the resulting template $\mathcal{T}_2'$ after inserting the module $\mathcal{M}_{\text{Array}}$ into the newly generated template $\mathcal{T}_1'$. The code snippet of $\mathcal{M}_{\text{Array}}$ is inserted into the template. The *module* placeholder has been moved, while the other placeholders have been retained.

```
...
public class MainActivity extends Activity {
  {{ globals }}

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);
    String sensitiveData;

    TelephonyManager tm = (TelephonyManager) getSystemService(
    Context.TELPHONY_SERVICE);
    sensitiveData = tm.getDeviceId();
```

```
        String[] array = new String[10];
        array[5] = sensitiveData;
        array[4] = "unsensitive data";
        sensitiveData = array[5];
        {{ module }}
    }
    {{ methods }}
}
{{ classes }}
```

Listing 3.13: $\mathcal{T}_2'$: After the insertion of $\mathcal{M}_{\text{Array}}$ into $\mathcal{T}_1'$

Listing 3.14 shows the final source code of template $\mathcal{T}_{\text{Final}}$. This is the resulting template once the last module of the TMC, $\mathcal{M}_{\text{SMS}}$, has been inserted into $\mathcal{T}_2'$ and final cleanup operations have been performed. The last code snippet is inserted and all remaining placeholders are removed.

The same insertion and subsequent cleanup process is also performed for the Android manifest file, as well as for the layout file.

```
...
public class MainActivity extends Activity {

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity);
    String sensitiveData;

    TelephonyManager tm = (TelephonyManager) getSystemService(
    Context.TELPHONY_SERVICE);
    sensitiveData = tm.getDeviceId();

    String[] array = new String[10];
    array[5] = sensitiveData;
    array[4] = "unsensitive data";
    sensitiveData = array[5];

    SmsManager sm = SmsManager.getDefault();
    sm.sendTextMessage("+49 12345", null, sensitiveData, null, null);
  }

}
```

Listing 3.14: $\mathcal{T}_{\text{Final}}$: After the insertion of $\mathcal{M}_{\text{SMS}}$ into $\mathcal{T}_2'$ and final cleanup

3.5.3 Build Tool

The build tool is the last component in the processing pipeline of the benchmark case generator. It is used to automatically manage and control the compilation process, as well as manage the

required dependencies of the application. Its main task is to compile the source code into a set of executable files that can be run on an Android system. These executable files are then packaged together with the Android manifest file, the layout file and other resource files that are required by the application into a container file. Finally, the build tool is responsible for signing the resulting file, so it can be executed on an Android system. This container file is the resulting APK file.

3.6 Outputs

The benchmark case generator generates two output files: (1) the APK file containing the generated benchmark case and (2) the corresponding ground-truth description, that contains information about the taint flows inside the generated benchmark case.

The generated APK file contains the Android application that represents the resulting benchmark case. This file can be used to install the application on an Android device or it can be used as input for an Android taint analysis tool.

The ground-truth description is needed in order to correctly assess the performance of the tested evaluation tool on the generated benchmark case. It contains the source-to-sink connections from the ground-truth graph that is generated by the preprocessor (see Section 3.5.1). The ground-truth comes in form of an AQL-Answer [44], which is in XML format. The AQL-Answer format is a format that is used to represent analysis results in a standardized form. This format allows for an automatic comparison of the ground-truth to the analysis results of Android taint analysis tools for the generated benchmark case.

Listing 3.15 shows the ground-truth of the generated benchmark case from the running example (see Figure 2.4) in form of an AQL-Answer.

The AQL-Answer contains exactly one taint flow. The `reference` tag where the attribute `type` has the value `from` contains the source of the taint flow. It shows that the `getDeviceId()` statement inside the `onCreate()` method of the `MainActivity` class is responsible for introducing the sensitive data. Furthermore, it contains the path to the APK file and various hashes of the file. The `reference` tag with the value `to` of the `type` attribute contains the taint flow's sink. It shows that the statement `sendTextMessage()` is responsible for leaking the sensitive data to the outside world. Since the location of the `sendTextMessage()` statement is the same as for the source statement, the `method` and `classname` values are the same as for the source. In addition to the flow it contains one attribute called `leaking`. This attribute is provided for every taint flow that is described inside the ground-truth. If the value of this attribute is `true`, the taint flow is in fact a leaking one and therefore a positive taint flow. If the value is `false`, the taint flow is a negative one.

This ground-truth description can easily be generated by traversing the ground-truth graph from Figure 3.3.

```
<answer>
  <flows>
    <flow>
      <reference type="from">
        <statement>... getDeviceId() ...</statement>
        <method>... onCreate(...) ...</method>
        <classname>... MainActivity</classname>
        <app>
          <file>.../ExampleBenchmark.apk</file>
```

```
          <hashes>...</hashes>
        </app>
      </reference>
      <reference type="to">
        <statement>... sendTextMessage(...) ...</statement>
        <method>... onCreate(...) ...</method>
        <classname>... MainActivity</classname>
      </reference>
      <attributes>
        <attribute>
          <name>leaking</name>
          <value>true</value>
        </attribute>
      </attributes>
    </flow>
  </flows>
</answer>
```

Listing 3.15: Ground-truth in form of an AQL-Answer [44] for the running example (see Figure 2.4) (shortened)

The benchmark case generator is also able to generate a more detailed ground-truth description for a benchmark case, by including every node from the generated ground-truth graph, instead of only depicting the source-to-sink connections. This can, as well, be easily achieved by simply traversing the ground-truth graph.

3.7 Integration of the Benchmark Case Generator in the Benchmarking Process

Figure 3.5 shows the integration of the benchmark case generator (BCG) into the benchmarking process of an Android taint analysis tool. At first the BCG generates an Android application in form of an APK file that represents a benchmark case and a ground-truth that contains the expected analysis results. This can either happen by providing a specific TMC to the BCG or by randomly generating an Android application via fuzzing. The generated application is then provided to the Android taint analysis tool. This analysis tool will perform an analysis on the supplied application and try to find all positive taint flows contained inside it. The output of the analysis tool, after finishing the analysis process, is an analysis-result, containing all taint flows that have been found inside the application. If the analysis tool has been executed by the AQL-System [44], the analysis result comes in form of an AQL-Answer. This analysis-result can then be compared against the ground-truth that has been generated alongside the application by the BCG, which contains a description about all taint flows contained inside the application. This comparison can be automatically performed, since the analysis-results and the ground-truth are both in the same format. If the found taint flows inside the analysis-result equal all the positive taint flows listed inside the ground-truth, a correct analysis for the supplied application has been performed. If the found flows inside the analysis-result do not equal the positive taint flows listed inside the ground-truth, the analysis of the taint analysis tool on the application is wrong. It means that the tool either was not able to find a positive taint flow that is inside the generated application or it found a negative taint flow that just looks like a positive taint flow,

but does not leak sensitive data to the outside world. Based on the results, the performance metrics, introduced in Section 2.2.2, can be determined.

This process can be repeated with various TMCs that represent applications with taint flows that are containing different aspects like ICC, different kinds of data structures or reflection in order to determine the strengths and weaknesses of the Android taint analysis tool.

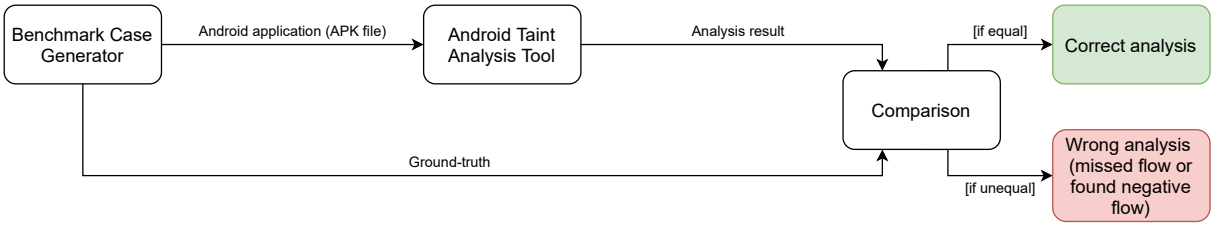


Figure 3.5: Example benchmarking process of an Android taint analysis tool

4 Implementation

While the previous chapter focused on the theoretical concept for a benchmark case generator, this chapter will describe the implementation of the mentioned concept in form of an application called GenBenchDroid, which is implemented in form of a Node.js application [21].

The first part of this chapter describes some details regarding the implementation of GenBenchDroid and the most important technologies used during the implementation. Afterwards the structure of GenBenchDroid is presented shortly. At the end of this chapter, a user manual which describes the usage of GenBenchDroid and the corresponding GenBenchDroid Editor, which is used for the creation of new templates and modules, is presented.

4.1 Implementation Details

The steps in the chapter describing the conceptual design of a benchmark case generator (see Chapter 3) showed a detailed description of how each step in the processing pipeline has to be performed in order to create a benchmark case generator. These steps can be summarized to four main steps in GenBenchDroid:

1. **Verification/Fuzzing:** In this step, if the user provided his/her own TMC, the provided TMC is verified and afterwards parsed into a tree structure. If the user did not provide his/her own TMC, a random, but valid, TMC is generated.

2. **Input Loading**: The template and modules that have been specified in the TMC are loaded.

3. **Source Generation:** During this step the necessary information to generate a ground-truth is extracted. Furthermore, the specified template and modules are interweaved in order to create the source code and additional resources needed to generate the benchmark case.

4. **Output Generation:** Once the necessary source code and resource files are generated, they can be compiled into an Android application in this step. Additionally, the ground-truth file is generated.

In the first step, the user-provided TMC, which comes in form of a string, is verified. In order to verify the TMC, a parser, which is generated from a provided grammar (see Section 3.4.1), is used. The parser is generated by nearley.js [30], which is a parser generator for Node.js

applications. This parser generation approach comes with the advantage that the grammar can easily be extended and a new corresponding parser can be generated automatically from the newly extended grammar. If the user did not provide a TMC and instead used the fuzzing mode, a TMC, in form of a string, is automatically generated by using nearley.js's `unparse` functionality, which generates a random string based on the provided grammar. This way the generated TMC will always be valid. Afterwards the TMC will be parsed into a tree structure by using a tree parser in order to correctly handle the usage of branching modules.

The files containing the template and modules information corresponding to the specified template and modules in the TMC are loaded in the Input Loading step. The files are in JSON format and can therefore be automatically parsed by Node.js without the need of other libraries.

In the Source Generation phase the template and each module are preprocessed in order to ensure that the finally generated source code is valid Java source code. Furthermore, the flow information from the template and each module is extracted in order to create a corresponding ground-truth graph. After preprocessing a module, it is inserted into the provided template. For this insertion process, a custom template engine based on the existing template engine Handlebars [16] is used. There is a multitude of other template engines available, which provide the same capabilities as Handlebars. However, Handlebars was selected, as it is a logic-less template engine. This means that the biggest part of the logic is not embedded in the template, but rather in the processing of the template. This makes the creation of new templates and modules a lot easier and more convenient, since the required logic is already provided by GenBenchDroid. Once all modules have been inserted, the necessary source code and resource files are generated.

The Output Generation step is performed, after all necessary source code and resource files are generated and are ready to be compiled into an executable Android application. To perform this compilation process the build tool Gradle [15] is used. Gradle is the build tool that is most commonly used for compiling Android applications, since it is the default build tool used by Android Studio. It compiles the provided source code, as well as the resource files, and generates an executable Android application if form of an APK-file. Additionally, the ground-truth graph, which was generated during the preprocessing of the modules, is traversed and a ground-truth file in form of an AQL-Answer is generated.

A list of further used technologies can be found in Appendix A.2.

4.2 Structure

The UML class diagram in Figure 4.1 illustrates the structure of GenBenchDroid. Cardinalities, as well as some attributes and methods are omitted for more clarity. It consists of one central `System` class. The `System` class has access to most of the other components which are used one after another. The `System` class is invoked via the `CLI` class, which lets the user interact with GenBenchDroid. The method `start(...)` in the `CLI` class is used to start GenBenchDroid. Based on the provided parameters either the fuzzing or the manual mode is executed. In case of the manual mode, the user directly provides a TMC in a string format. This provided TMC in string format is verfied by the `Verificator` class. The `Verificator` class uses a `Grammar` to perform this verification. If the user selects the fuzzing mode, the `Verificator` class will automatically generate a valid TMC in string format by using the `Grammar`. This TMC in string format is then parsed into a `TMC` object by the `Tree Parser` class. Based on the provided or generated `TMC` object, the `System` class uses the `load(...)` method to load all corresponding `Template` and `Module` objects. The loaded `Template` and `Module` objects are preprocessed by the `Preprocessor` class before they are forwarded to the `Template Engine` class. The `Template Engine` class inserts each `Module` object one by one into the `Template` object and stores the intermediate results in the `sourceContent`, `manifestContent` and `layoutContent`

attributes. Once this procedure is finished, the `System` class invokes the `getFileContents()` method of the `Template Engine` class to request the contents that have to be written into source files. To generate these files the `generateFile(...)` method of the `File Generator` class is invoked. After finishing the file generation, the `build()` method of the `Build tool` class is called, which generates the corresponding `Android Application` object. Additionally, after a successful compilation, the `generateGroundTruth(...)` method of the `Preprocessor` class is invoked to generate the `Ground-Truth` object that corresponds to the generated application.

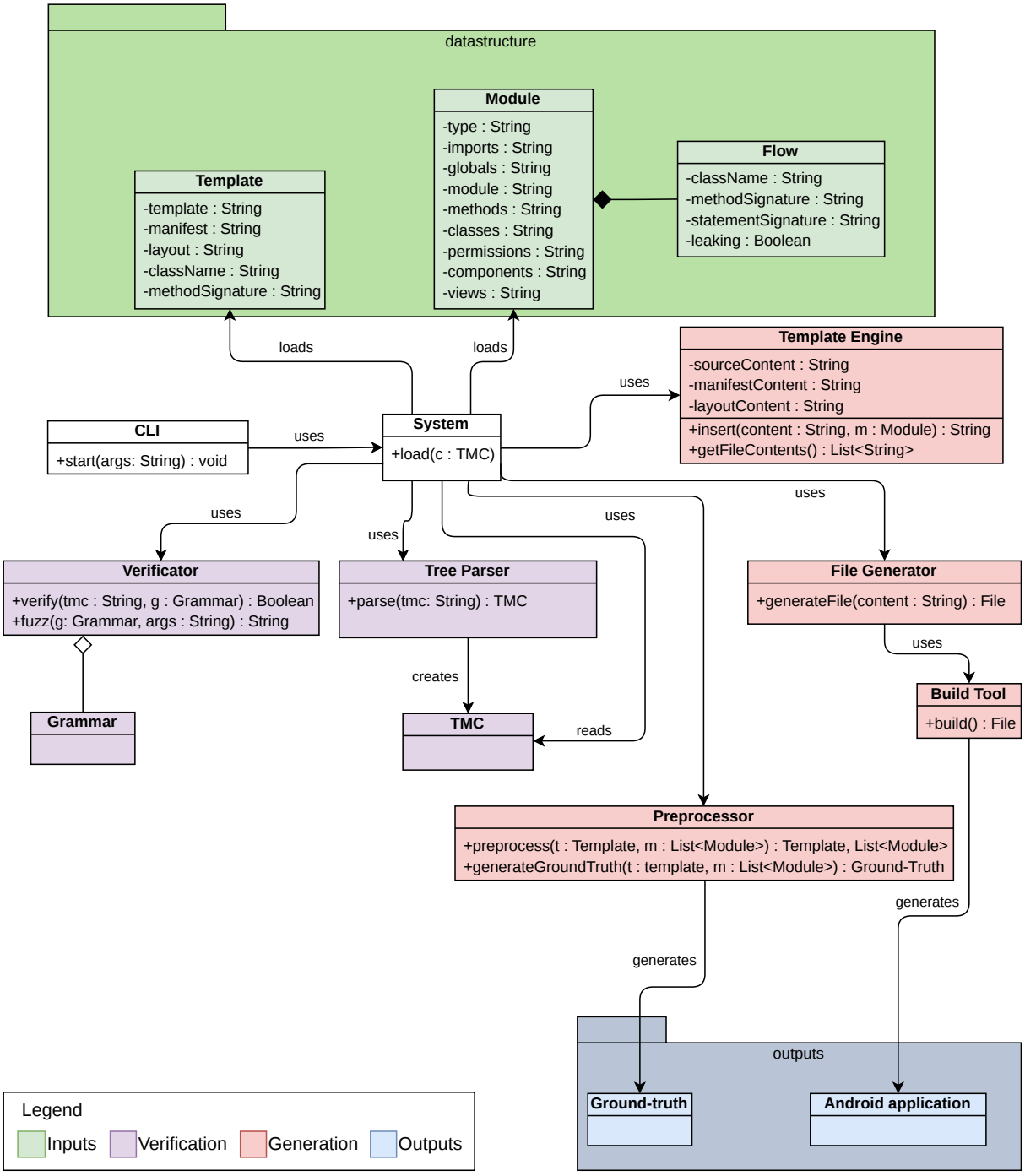


Figure 4.1: UML Class Diagram of GenBenchDroid (simplified)

4.3 Manual

This section explains the installation, configuration and usage of GenBenchDroid. It also explains how to extend GenBenchDroid by creating new templates and modules.

4.3.1 Installation

The GenBenchDroid application is attached to this thesis (see Appendix A.4). It is contained inside a repository that consists of several files. To be able to run the application, a Node.js [21] version of 14.5 or higher has to be installed on the system. Furthermore, in order to automatically build an Android application with Gradle, the Android Software Development Kit (SDK) [9], which is already shipped with Android Studio, as well as the Java Development Kit (JDK) [18] are additionally required. The following list shows where the mentioned applications can be obtained from:

- **Node.js:** `https://nodejs.org/`
- **Android SDK:** `https://developer.android.com/studio/`
- **JDK:** `https://oracle.com/de/java/technologies/javase-jdk15-downloads.html`

Before the first execution of GenBenchDroid, several dependencies have to be installed. This process can be automatically handled by the Node Package Manager (NPM) that is automatically installed with Node.js. In order to trigger the installation process, the following command has to be executed from the command-line inside the directory containing GenBenchDroid:

```
npm install
```

After executing the command, the installation process will take some seconds and after finishing the process, a new folder named `node_modules` will appear inside the GenBenchDroid directory. This folder contains all the required dependencies. After this step GenBenchDroid is ready to be configured.

4.3.2 Configuration

GenBenchDroid has to be configured before its usage. The configuration file is called `.env` (conventional way of configuring Node.js applications) and can be found in the root directory of GenBenchDroid. To configure GenBenchDroid the user has to set the following key-value pairs:

- TEMPLATE_DIR: Specifies the location of the templates that are provided to GenBenchDroid.
- MODULE_DIR: Specifies the location of the modules that are provided to GenBenchDroid.
- OUTPUT_DIR: Specifies the location of where the generated files, like the APK file containing the generated application and the corresponding ground-truth file should be placed.
- PROJECT_NAME: Specifies the project name of the generated application. This property will be used inside the Android Manifest file, as well as the package name for the generated application.

- ANDROID_SDK_DIR: Specifies the location of the Android SDK.
- JDK_DIR: Specifies the location of the JDK.

Listing 4.1 shows one example configuration for GenBenchDroid. After the configuration, GenBenchDroid is ready to be used.

```
TEMPLATE_DIR = templates
MODULE_DIR = modules
OUTPUT_DIR = output
PROJECT_NAME = com.generated.app
ANDROID_SDK_DIR = path/to/Android/Sdk
JDK_DIR = path/to/Java/jdk
```

Listing 4.1: Example content of the `.env` configuration file

4.3.3 Usage

GenBenchDroid allows for two different usage modes: (1) the *manual mode* and (2) the *fuzzing mode*.

Manual mode

The manual mode allows the user to provide his/her own TMC. This way the user is able to generate a benchmark case that is fully customized to his/her desires. In order to start GenBenchDroid, the following command has to be executed from the command-line inside the GenBenchDroid directory:

```
node app --configuration <TMC>
```

The placeholder <TMC> is replaced with the desired TMC in string format. The following command shows the execution of GenBenchDroid with the running example (see Figure 2.4) as argument:

```
node app --config "BasicTemplate ImeiSource ArrayBridge SmsSink"
```

This command will generate the benchmark case from the running example (see Figure 2.4). The templates and modules are specified by their individual file names without their file extension. The file that contains the template T_{Basic} is called `BasicTemplate.json`. Therefore, the string containing the TMC specifies the usage of this template by the token `BasicTemplate`.

The desired template has to always be the first token inside the provided string. Afterwards no more templates can be provided, only further modules are valid tokens. Every token inside the string containing the TMC has to be separated by spaces. Furthermore, the insertion order of the modules is specified by the order of the configuration, from left to right.

In case that one of the provided modules is a branching module, each branch is indicated by a parenthesis pair directly following the branching module. For every branch that is created by the module, one parenthesis pair has to be provided. The following command shows the creation of a benchmark case with a branching module contained inside the TMC:

```
node app --config "BasicTemplate RandomIfElseBridge ( ArrayBridge ) ( SmsSink )"
```

The module $\mathcal{M}_{\text{RandomIfElse}}$, which is contained inside the file `RandomIfElseBridge.json` is a branching module that creates two branches. Therefore, two parenthesis pairs are specified directly afterwards. The content inside the first parenthesis pair shows the modules that are inserted into the first branch, while the content inside the second parenthesis pair shows the modules that are inserted into the second branch.

Parenthesis pairs may be nested arbitrarily deep, by inserting more branching modules. However, they may also be left empty. This indicates that no further module is inserted into the corresponding branch.

After the execution of the command, GenBenchDroid verifies that the provided configuration is valid. If it is valid, GenBenchDroid will start the generation process. After finishing the process successfully, the generated Android application and the other files can be found inside the output directory that has been specified inside the configuration file.

Fuzzing mode

The fuzzing mode will generate a benchmark case from an automatically randomly generated TMC. In order to execute the fuzzing mode, the following command has to be executed from the command-line inside the GenBenchDroid directory:

```
node app --fuzz
```

After executing this command, GenBenchDroid will generate a random, but valid, TMC and use this TMC to generate a benchmark case. It will use a random subset of all templates and modules that are specified inside the grammar to create the TMC. The fuzzing mode can be configured with a set of further parameters.

The `depth` parameter indicates the maximum amount of derivations that can be used to generate the TMC. The higher this value is set, the longer the TMC can become. However, short TMCs are still possible. The following command generates a random TMC with a maximum of 100 derivations:

```
node app --fuzz --depth 100
```

The `minLength` parameter indicates the minimum length of the generated TMC. The length of a TMC is defined by the amount of tokens inside the TMC. The following command generates a random TMC with a minimum length of 10 tokens.

```
node app --fuzz --minLength 10
```

The `taintflow` parameter indicates whether the generated TMC has to definitely contain a taintflow inside it. The following command generates a random TMC that has a guaranteed taintflow inside it.

```
node app --fuzz --taintflow
```

The `contains` parameter indicates that one or multiple specific modules have to be contained inside the generated TMC. It can also handle substrings in order to not only specify one specific module or template, but specify a set of templates or modules containing the substring. The following command generates a random TMC which contains the module *ArrayBridge* and also a random module that has *Sink* in its name.

```
node app --fuzz --contains ArrayBridge Sink
```

The `ignore` parameter is the counter-part to the `contains` parameter. It will only generate benchmark cases that do not contain the specified modules.

All of the above mentioned parameters can also be combined. However, the user has to consider to provide only sensible combinations of parameters, since no TMC with a minimum length of 50 can be generated with a maximum derivation depth of 20.

4.3.4 Extension

One important capability of GenBenchDroid is its extendibility. The user can creates his/her own templates and modules and provide them to GenBenchDroid. The templates and modules are encoded in JSON format and can be created manually. To ease the creation process an application called GenBenchDroid Editor has been developed, which is attached to the thesis (see Appendix A.4). Figure 4.2 shows the GUI of the GenBenchDroid Editor. It consists of two tabs, one for template creation and one for module creation. It contains all fields that can be specified for a template and a module. Additionally, it contains a panel where the flow of the template and, in case of module creation, a set of flows can be specified. It also allows the user to load already existing templates or modules and modify them.

Before the first execution of the Modulebuilder, the following command has to be executed from the command-line inside the directory containing GenBenchDroid Editor, in order to install all needed dependencies:

```
npm install
```

After the installation process finished, the GenBenchDroid Editor can be started by executing the following command:

```
npm start
```

Creating Templates

To create a new template, each of the specified text fields in the GenBenchDroid Editor have to be filled in. The Source Code text field has to contain the structure of the Android component that represents the entry point to the generated application. The following five placeholders have to be specified inside it (placeholders are always wrapped inside double curly brackets):

Placeholder	**Location Description**
{{ imports }}	required imports
{{ globals }}	created global fields of further modules
{{ module }}	source code of the next module
{{ methods }}	created methods of further modules
{{ classes }}	created classes/components of further modules

Table 4.1: Placeholders for created templates in the Source Code field

Additionally, in case of the component being an activity, it has to contain the `onCreate` callback where the corresponding layout file is set via the statement `setContentView(R.layout.Activity)`. Furthermore, a variable of type `String` called `sensitiveData` has to be declared and initialized, right before the location of the *module* placeholder. This can happen in an arbitrary lifecycle callback. And the final requirement is that the context of the activity has to

be stored in a global field called `context`.

The Android Manifest text field has to contain the structure of the Android manifest file for the generated application. The first component that is created by the template has to be directly defined in the Android Manifest text field. Furthermore, the following three placeholders have to be specified inside it:

Placeholder	**Location Description**
{{ project }}	every location where the name of the project is required (can be set inside the configuration file (see Section 4.3.2))
{{ permissions }}	permission requests and used features declarations
{{ components }}	description of components that are created by further modules

Table 4.2: Placeholders for created templates in the Android Manifest field

The Layout text field has to contain the structure of the layout file for the generated application. It has to contain a `ViewGroup` as the outermost element and also specify one placeholder:

Placeholder	**Location Description**
{{ views }}	description of `View` elements that are created by further modules

Table 4.3: Placeholders for created templates in the Layout field

The Class Name text field inside the Flow section of the GenBenchDroid Editor has to contain the name of the class that the *module* placeholder is placed in. The Method Signature field has to contain the signature of the method the *module* placeholder is located at in a *Jimple* [46] representation.

Creating Modules

When creating a new module, at first the type of the module has to be selected. Furthermore, a unique name for the module has to be specified. If the component is a source module, it has to contain the word "Source" inside its name. The same holds for sink modules (This greatly increases the performance of the fuzzing mode, since not every module has to be loaded beforehand to determine whether the generated application will contain a taint flow). The text fields Imports, Globals, Methods, Classes, Permissions, Components and Views can be left empty if not required. The only mandatory text field is the Module field. This field represents the code snippet, that is inserted in the *module* placeholder of the previous module. It is the first code of the module that will be executed by the generated application. This field can therefore be used to invoke methods or classes that are created by the module. Each module has access to the `sensitiveData` variable which contains the sensitive data and can freely interact with it. If the created module changes the program flow by, for example, invoking a newly created method, the user needs to make sure to make that variable, as well as the current context in a variable called `context`, available to the next module.

There are only four placeholders that have to be defined inside the module:
If the program flow is not altered and therefore the location of the *globals*, *methods* and *classes* placeholders does not change, the placeholders stay in the same text field. However, if the program flow changes, the location of them may as well change. Since the Module text field is mandatory, the location of the *module* placeholder always changes.

If the newly created module is a branching module, each of the four placeholders have to be inserted for each created branch at the appropriate location. This means that if there a two

Placeholder	Location Description
{{ globals }}	created global fields of further modules
{{ module }}	source code of the next module
{{ methods }}	created methods of further modules
{{ classes }}	created classes/components of further modules

Table 4.4: Placeholders that have to be specified inside newly created modules

branches in the module, each of the four placeholders has to be specified twice. This also has to be performed, even if there are multiple placeholders of the same type right after each other.

Furthermore, whenever a new identifier is introduced by the user (besides the `sensitiveData` and `context` identifier) every occurrence of it has to be wrapped inside a § and a $ symbol (e.g. int §`data`$ = 123).

Describing the flow of the module is mandatory. The description of the flow is performed like for the creation of templates. However, if the program flow is not altered by the module and the location of the *module* placeholder does not change to another class or method, these fields can be left empty. Furthermore, there are three additional properties. The Statement Signature field has to contain the signature of the function that interacts with the sensitive data in a *Jimple* [46] format. While this field is mandatory for source and sink modules, it can be left empty for bridge modules. The leaking checkbox can be used to indicate whether the module continues the taint flow or sanitizes it (if unchecked, the taint flow is rendered negative). The reachable checkbox indicates that the following code is unreachable, meaning that the taint flow can never be turned positive again. For every branch that is created by the module, one individual flow description has to be created. The first flow description corresponds to the *module* placeholder that is the furthest up in the generated source code, while the last one corresponds to the placeholder that is the furthest down. The flow descriptions can be easily rearranged via drag and drop.

Adding to the Grammar

Whenever a new template or module has been created, it has to be added to the grammar, so it can be used by GenBenchDroid. In order to add it to the grammar, the user has to modify the file called `grammar.ne`, that is located inside the GenBenchDroid directory. At the bottom of the file there are two symbols. If the user created a template he simply has to add it to the `template` symbol. To add it he/she has to add a | symbol (indicates an OR-symbol) to the last defined template and attach the name of the newly created template without the file extension to it. The same procedure has to be performed for created modules. However, the created modules need to be attached to the `module` symbol and also need to have an extra symbol attached to them, based on the type of each created module. If the user created a linear module, he/she has to attach a `linear` symbol to the module name. If the user created a branching module with two branches, he/she has to attach a `2Branches` symbol, if the created module contains three new branches, the `3Branches` symbol has to be attached. The number in the symbol indicates the number of created branches.

Listing 4.2 shows an example of adding newly created modules and templates to the grammar. Newly created modules and templates are highlighted in green color.

```
...
template -> "BasicTemplate" | ... | "CreatedTemplate"
module   -> "ImeiSource" linear | ... | "CreatedLinearModule" linear |
```

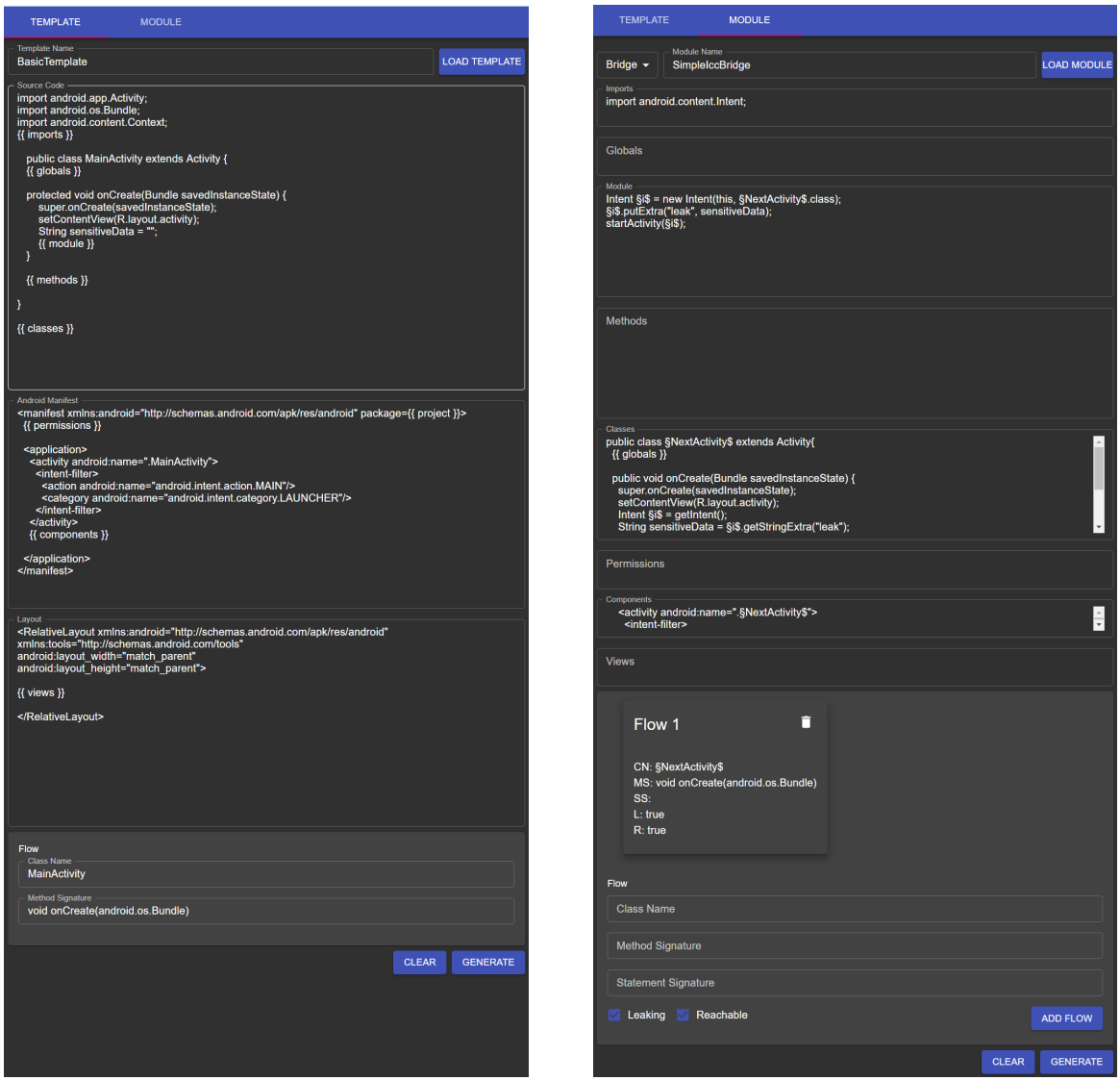
```
"CreatedBranchingModule" 2Branches
```

Listing 4.2: Adding newly created templates and modules to the grammar

After the grammar has been modified, a new parser has to be compiled from the grammar. In order to do so, the following command has to be executed from the command-line inside the GenBenchDroid directory:

```
node app --compile
```

The modified grammar is then compiled into a parser that can be used by GenBenchDroid. Now the newly created templates and modules can be used by it.



(a) Template creation tab

(b) Module creation tab

Figure 4.2: GUI of the GenBenchDroid Editor

4.3.5 Available Parameters

Table 4.5 shows all parameters that are available to GenBenchDroid as well as their functionality and an example usage.

Parameter	**Functionality**	**Example Usage**
--configuration, --config, -c	Generate a benchmark case with the provided TMC.	node app -c "BasicTemplate ImeiSource SmsSink" Generates a basic benchmark case.
--fuzz, -f	Activates fuzzing mode and generates a random benchmark case with a randomly generated TMC.	node app -f Generates a random benchmark case.
--depth, -d	Can only be used inside fuzzing mode. Specifies the maximum amount of derivations allowed to generate the TMC. Default value: 25	node app -f -d 20 Generates a random benchmark case with at most 20 derivations.
--minLength, -m	Can only be used inside fuzzing mode. Specifies the minimum length of the generated TMC. Default value: 1	node app -f -m 10 Generates a random benchmark case with a minimum TMC length of 10 tokens.
--taintflow, -t	Can only be used inside fuzzing mode. Specifies that there has to be a guaranteed taint flow inside the generated TMC.	node app -f -t Generates a random benchmark case with a guaranteed taint flow inside.
--contains	Can only be used inside fuzzing mode. Specifies strings that have to contained inside the generated TMC.	node app -f --contains ArrayBridge Sink Generates a random benchmark case with a TMC that contais an ArrayBridge and a random module containing the word Sink.
--ignore	Can only be used inside fuzzing mode. Specifies strings that cannot be contained inside the generated TMC.	node app -f --ignore ArrayBridge Generates a random benchmark case with a TMC that does not contain an ArrayBridge.
--compile	Recompiles the grammar into a parser. Has to be executed, whenever the grammar has been changed or extended.	node app --compile Recompiles the parser from the grammar.

Table 4.5: Available Parameters

5 Evaluation

The previous sections described a concept for a benchmark case generator. Based on this concept a generator called GenBenchDroid has been implemented. In this chapter the performance of GenBenchDroid will be evaluated in regard to various aspects. A direct comparison to other tools however is not possible since there exist none which are similar enough (see Chapter 6).

- **RQ1:** Is GenBenchDroid able to generate existing micro benchmark cases?

  Furthermore, do the generated micro benchmark cases provide the same analysis results as their hand-crafted counter-parts when used to evaluate Android taintflow analysis tools?

  To answer these questions, several micro benchmark cases from DroidBench will be generated by GenBenchDroid and compared to their manually created counter-parts. A manual review of the source code, as well as a comparison of analysis results will be performed. Additionally, the executability of the generated benchmark cases will be tested on an Android device.

- **RQ2:** How long does GenBenchDroid take to generate a benchmark case?

  How does the amount of used modules influence the generation time?

  In order to answer these questions, multiple benchmark cases, using TMCs of various length, will be generated and the generation time will be measured. Furthermore, the relation between benchmark case generation time and analysis time will be investigated.

- **RQ3:** How many benchmark cases have to be fuzzed to generate a meaningful benchmark suite?

  To answer this question multiple benchmark suites containing various amounts of benchmark cases will be generated by GenBenchDroid's fuzzing mode and supplied to an analysis tool in order to determine various statistical measures.

- **RQ4:** Is GenBenchDroid able to generate benchmark cases that are comparable to real-world applications?

  To answer this question a case study will be performed that investigates a generated benchmark case that contains the same amount of taintflows with the same aspects that a real-world application from TaintBench contains.

The following sections will address the research questions in detail. Furthermore, a summary of the answers can be found at the end of this chapter.

5.1 RQ1: Is GenBenchDroid able to generate existing micro benchmark cases?

To answer RQ1, benchmark cases have been generated, using GenBenchDroid that are semantically equivalent to micro benchmark cases from DroidBench 3.0 [13]. DroidBench consists of 190 micro benchmark cases that are distributed among 18 categories. In order to cover a wide variety of different categories of benchmark cases, templates and modules have been created to regenerate one benchmark case from 13 categories of DroidBench. Five categories have been omitted for the following reasons: three of the omitted categories of DroidBench (DynamicLoading, Native, Selfmodification) require the usage of external files, like, for example, native libraries. The usage of external files is currently not supported by GenBenchDroid and is subject of further work. The other two omitted categories (InterAppCommunication, Reflection_ICC) can be constructed by combining other categories of DroidBench.

Figure 5.1 illustrates the evaluation process. At first, one benchmark case from each category of DroidBench (besides the above mentioned five categories) has been selected. This selection has been performed in a way that the alphabetically first benchmark case of each category has been selected. However, there were some derivations in order to ensure a mixture of benchmark cases that contain a positive and benchmark cases that contain a negative taint flow. Based on the selected benchmark cases, modules and templates have been created, so GenBenchDroid is able to regenerate the selected benchmark cases. The created modules and templates have then been supplied to GenBenchDroid in order to generate the benchmark cases (the configuration used for the generation of each benchmark case can be found in Appendix A.3). A ground-truth has been generated as well, however it was not needed for the current evaluation process, because the topic of interest was only whether the regenerated benchmark cases yield the same analysis results as the hand-crafted ones from DroidBench. After the generation of a benchmark case, the APK-file has been decompiled and a manual review has been performed in order to verify that the generated benchmark case contains the same taint flows as its manually created counter-part and also to check that both benchmark cases are semantically equivalent. If these properties were not achieved, the template/modules have been adjusted accordingly. Once the taint flows did match each other and both benchmark cases were semantically equivalent, the original benchmark case from DroidBench, as well as the one generated by GenBenchDroid have been supplied to FlowDroid 2.7.1 [26] and Amandroid 3.1.2 [49] for an analysis.

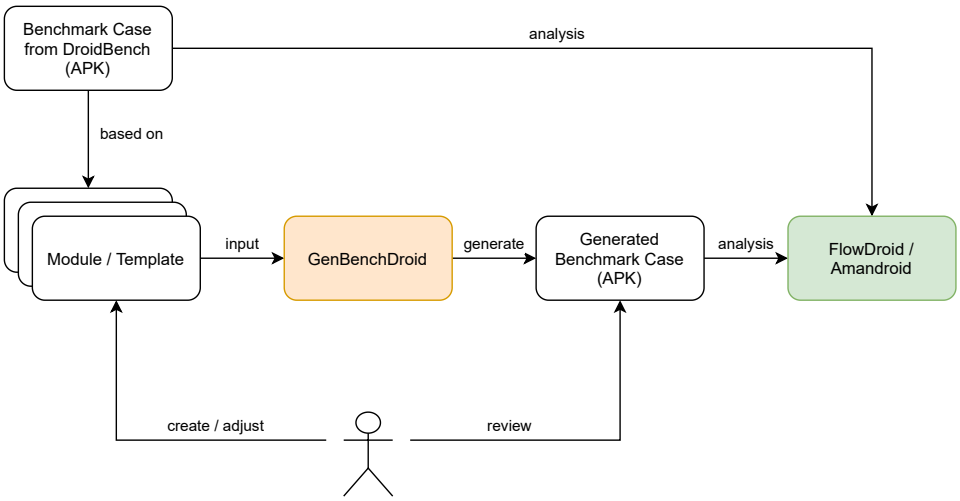


Figure 5.1: Benchmark case evaluation process

Table 5.1 shows the results of the analysis. FlowDroid and Amandroid both yielded five differing analysis results. FlowDroid was not able to uncover positive taint flows contained in the benchmark cases PublicAPIField1, Clone1, ImplicitFlow1 and ImplicitFlow5. Furthermore, it yielded a false warning for the benchmark case ArrayAccess1 that contains a negative taint flow. Amandroid only failed to uncover the taint flow in the benchmark cases ImplicitFlow1 and ImplicitFlow5. However, Amandroid yielded three false warnings for the benchmark cases FlowSensitivity1, ArrayAccess1 and SimpleUnreachable1. The most important result however, is that the generated benchmark cases almost always yielded the same analysis results as their manually created counter-parts.

This result indicates that GenBenchDroid is able to regenerate micro benchmark cases from most categories of DroidBench 3.0.

⊛= correct warning, ⋆= false warning, ◯= missed leak, ◯= no leak and no report,
multiple symbols = multiple taint flows inside benchmark case

Category	Benchmark Case	FlowDroid		Amandroid	
		DroidBench	Generated	DroidBench	Generated
Aliasing	FlowSensitivity1	◯	◯	⋆	⋆
AndroidSpecific	PublicAPIField1	◯	◯	⊛	⊛
ArraysAndLists	ArrayAccess1	⋆	⋆	⋆	⋆
Callbacks	Button1	⊛	⊛	⊛	⊛
EmulatorDetection	Bluetooth1	⊛	⊛	⊛	⊛
FieldAndObjectSensitivity	FieldSensitivity2	◯	◯	◯	◯
GeneralJava	Clone1	◯	◯	⊛	⊛
ImplicitFlows	ImplicitFlow1	⊛◯	⊛⊛	◯◯	◯◯
ImplicitFlows	ImplicitFlow5	◯	◯	◯	◯
ICC	ActivityCommunication1	⊛	⊛	⊛	⊛
Lifecycle	ActivityLifecycle4	⊛	⊛	⊛	⊛
Reflection	Reflection1	⊛	⊛	⊛	⊛
Threading	AsyncTask1	⊛	⊛	⊛	⊛
UnreachableCode	SimpleUnreachable1	◯	◯	⋆	⋆

Table 5.1: Analysis results of FlowDroid and Amandroid on benchmark cases from DroidBench and benchmark cases generated by GenBenchDroid

Nevertheless, this does not mean that GenBenchDroid is able to regenerate any benchmark case from DroidBench 3.0 and obtain the same analysis result from all Android taint flow analysis tools. During the evaluation process one benchmark case has been detected that, even though it is semantically equivalent and contains the same taint flows, did not yield the same analysis result from FlowDroid. The micro benchmark case ImplicitFlow1 of the ImplicitFlows category from DroidBench contains two positive taint flows. In both flows the device's IMEI number is read and leaked by writing it to a log. However, in the first taint flow, the numbers contained in the IMEI are directly replaced by characters, while in the second taint flow the numbers are replaced by characters based on ASCII values. Listing 5.1 shows the decompiled source code of the original benchmark case from DroidBench. Listing 5.2 shows the decompiled source code of the by GenBenchDroid regenerated benchmark case. The `obfuscateIMEI` method of the original benchmark case is equivalent to the `obfuscateData2` method of the regenerated benchmark case (besides some identifiers), and therefore both are omitted in the listings. The same holds for the `obfuscateIMEI` and `obfuscateData` methods. Both benchmark cases contain the same two positive taint flows and are semantically equivalent. However, the original benchmark case, in comparison to the regenerated one, only reads the device's IMEI number once and stores it inside a variable. Furthermore, it also outsources the method call, where the sensitive data is written into a log into its own method.

Even when altering the modules used to generate the benchmark case, GenBenchDroid was not able to regenerate an equal benchmark case due to compiler code changes.

While Amandroid was not able to uncover any of the taint flows contained inside the original benchmark case, nor the regenerated one, FlowDroid managed to uncover one taint flow inside the original benchmark case and even both taint flows in the regenerated one. This may be due to an *overshadowing* effect that is present in FlowDroid, which prevents it from finding a second taint flow, if two taint flows share a common part of the flow [22].

This result shows that even though GenBenchDroid may be able to generate semantically equivalent micro benchmark cases, in some cases these regenerated benchmark cases do not offer the same analysis results as their manually created counter-parts.

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_implicit_flow1);
  String imei = ((TelephonyManager) getSystemService("phone"))
                .getDeviceId();
  writeToLog(obfuscateIMEI(imei));
  writeToLog(copyIMEI(imei));
}

private void writeToLog(String message) {
  Log.i("INFO", message);
}

...
```

Listing 5.1: ImplicitFlow1 from DroidBench

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity);
  Log.i("INFO", obfuscateData2(((TelephonyManager)
    getSystemService("phone")).getDeviceId()));
  Log.i("INFO", copyData5(((TelephonyManager)
    getSystemService("phone")).getDeviceId()));
}

...
```

Listing 5.2: ImplicitFlow1 regenerated by GenBenchDroid

In addition to the analysis results, the executability of the regenerated benchmark cases has been investigated. GenBenchDroid compiles all generated source code into Android APK-files. This means that every generated APK-file does represent a valid Android application that contains no compile-time errors. However, there may be still runtime errors present in the generated APK-file. To determine whether the generated benchmark cases contain any runtime errors, each application has been installed and executed on a virtual Android device (Google Pixel 2), running Android 7.0 (API level 24). Every generated benchmark case was able to be

executed on the Android device without any runtime error. This result indicates that benchmark cases generated by GenBenchDroid are also suitable for Android taint flow analysis tools that perform a dynamic analysis.

GenBenchDroid offers additional capabilities compared to DroidBench. It is able to generate benchmark cases for categories that are not present in DroidBench.

One of these categories not present in DroidBench is *Recursion*. There are no benchmark cases in DroidBench that make use of recursive function calls. However, such benchmark cases can easily be generated by GenBenchDroid by using the provided `SimpleRecursionBridge`.

Furthermore, GenBenchDroid allows for a quick adjustment to new Android API releases.

Usually, whenever there is a new API release, many methods and classes become deprecated and are replaced by something else. Instead of updating every benchmark case that contains a deprecated method or class, for GenBenchDroid only the modules/templates using these deprecated methods or classes have to be updated. Once they have been updated, benchmark cases using the updated module/template can easily be regenerated. Especially for modules/templates that are used frequently, this saves the user a lot of time and effort.

5.2 RQ2: How long does GenBenchDroid take to generate a benchmark case?

To answer RQ2, benchmark cases with various amounts of modules have been generated by GenBenchDroid. Six benchmark cases with different module counts have been generated. The used modules have been selected randomly.

Table 5.2 shows the six benchmark cases with their individual amount of modules. Furthermore, it shows what kind of benchmark case is represented with each module amount and how many taint flows are approximately contained inside such a benchmark case.

Module count	**Represents**	**Approx. Taint flows**
4	Simple micro benchmark case	1
10	Complex micro benchmark case	1 – 2
25	Simple real-world application	3 – 6
100	Complex real-world application	20 – 40
500	Very complex real-world application	50 – 100
1000	Very complex real-world application	100 – 200

Table 5.2: Benchmark cases used for execution-time measurements

Figure 5.2 shows the results of the measurement of the execution-time of GenBenchDroid. All executions and measurements have been performed on a desktop computer running Windows 10 with an AMD Ryzen 5 3600 CPU and 32 GB of physical memory. Each benchmark case has been generated a total of 20 times and the average time of all generations has been calculated and can be seen in the chart (The first startup of the Gradle daemon takes around eight seconds and has been omitted in the evaluation as it only has to be started once after the computer is switched on). The execution-time is divided into three different phases. In the verification phase the provided TMC is validated against the supplied grammar. In the source-files generation phase the required templates and modules are loaded, preprocessed and all necessary source-files, as

well as the ground-truth, are generated. In the compilation phase all source-files are compiled into an executable Android APK-file.

One can observe that the time needed for the verification of the TMC can almost be neglected, as it only takes a very small percentage of the total generation time. Furthermore, one can observe that the time needed for the source-file generation also only takes a small percentage of the total generation time when the module count is rather small. However, at some point at around 500 modules the time needed for the generation of the source-files takes the biggest share of the generated time, as the compilation time stays fairly constant, independent from the amount of provided modules. Because of this behavior the time needed to generate very small micro benchmark cases and real-world benchmark cases is roughly the same and takes around three seconds for each generated benchmark case. Only when really complex benchmark cases with a fairly high module count are generated the generation time substantially increases.

This result indicates that a full benchmark suite like, for example, DroidBench, which consists of 190 micro benchmark cases, can be generated in less than ten minutes.

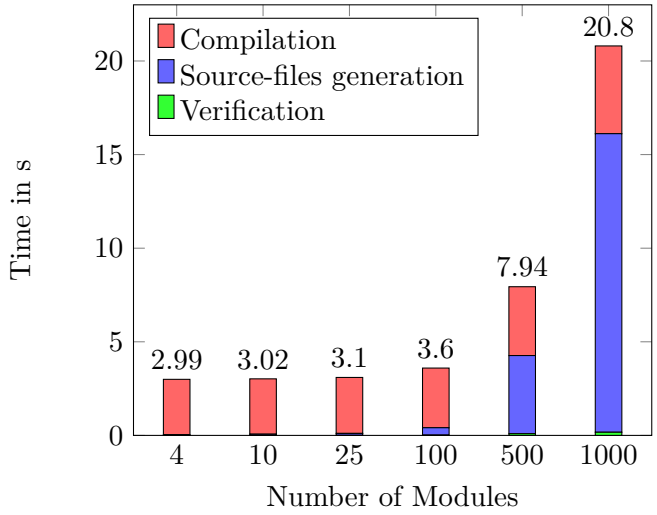


Figure 5.2: Time needed for GenBenchDroid to generate a benchmark case

Another interesting question is the relation between the generation time and the analysis time of benchmark cases. Figure 5.3 shows the relation between the generation time needed by GenBenchDroid and the analysis time needed by FlowDroid and Amandroid to analyze the generated benchmark cases. One can see that the generation time is, as already mentioned before, fairly constant until the module count becomes rather high. Only then the required generation time increases substantially. Furthermore, it can be observed that benchmark cases consisting of 25 modules or less take around the same time to be generated as they take to be analyzed by FlowDroid and Amandroid. Only for more complex benchmark cases with more than 25 modules a disparity between the generation and the analysis time can be observed. For FlowDroid the analysis time stays fairly low compared to the generation time. The generation of the benchmark case takes more and more time compared to the analysis. But this disparity only becomes clearly noticeable, once the module count of the benchmark case becomes rather high. The generation time for a benchmark case consisting of 500 modules takes around eight seconds, while the analysis of this benchmark case with FlowDroid takes around four seconds. This disparity further increases, as the module count increases.

However, for Amandroid the opposite can be observed. The increase of analysis time compared to the generation time is really significant once the generated benchmark cases become

bigger. The analysis of a benchmark case consisting of 500 modules takes Amandroid almost four times longer than the generation of the benchmark case (more than 27 seconds). This disparity still holds, even when the module count further increases.

These results indicate that for smaller benchmark cases (less than 25 modules) the generation and the analysis almost have the same share among the total time needed for generation and analysis of a benchmark case. For bigger benchmark cases however, the results depend on the used analysis tool.

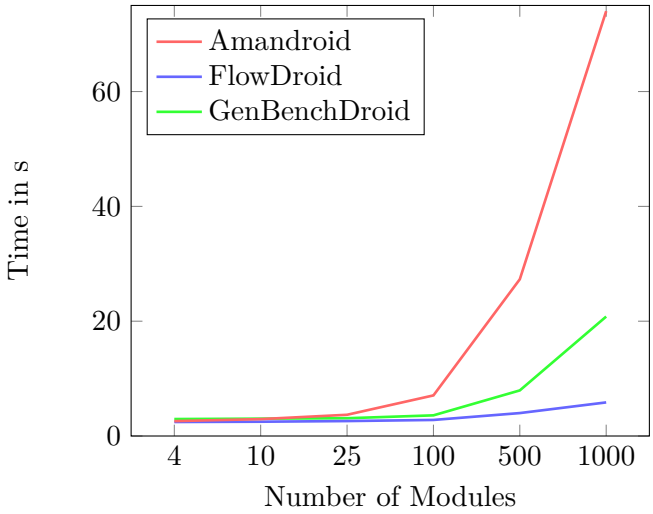


Figure 5.3: Relation between generation and analysis time

This experiment shows another capability of GenBenchDroid:

> GenBenchDroid can be used to efficiently analyze the execution-times of Android app analysis tools, since the size and complexity of the generated benchmark cases can easily be scaled up.

This can help to uncover execution-time related flaws in analysis tools for large applications.

5.3 RQ3: How many benchmark cases have to be fuzzed to generate a meaningful benchmark suite?

To answer RQ3, multiple benchmark suites consisting of various amounts of randomly generated benchmark cases by GenBenchDroid's fuzzing mode have been generated. The configuration used for GenBenchDroid's fuzzing mode in this experiment was the following:

1. The `taintflow` option has been provided in order to only generate benchmark cases containing at least one positive or negative taint flow. However, a generated benchmark case may also contain more than one taint flow.

2. The `minLength` option has been set to four. This means that every generated benchmark consists of one template and at least three modules. This prevents the generation of benchmark cases that only consist of one source and one subsequent sink.

In total, 70 benchmark suites have been generated consisting of 50, 100, 175, 250, 375, 500 and 750 benchmark cases. Ten benchmark suites have been generated consisting of each of the mentioned amounts of benchmark cases. This means there were ten benchmark suites consisting of 50 benchmark cases, ten benchmark suites consisting of 100 benchmark cases and so on. This

was done in order to compare the performance of the analysis tool using benchmark suites with the same amount of benchmark cases.

After generating the benchmark suites, each of them was provided to FLOWDROID 2.7.1 for analysis. The exact analysis results can be found in Appendix A.3.

Based on the analysis results, the precision, recall and F-measure (see Section 2.2.2) have been calculated for each benchmark suite.

Figure 5.4 shows the maximum difference between the analysis results (difference between the benchmark suite with the highest measure and the one with the lowest). One can observe that the maximum difference is very high for benchmark suites containing less than 200 benchmark cases. Especially the recall diverts heavily for small benchmark suites. However, one can also observe that at 250 benchmark cases the maximum difference for the precision does not decrease further and mostly stays constant. For the recall and the F-measure this behavior can first be observed at 500 benchmark cases.

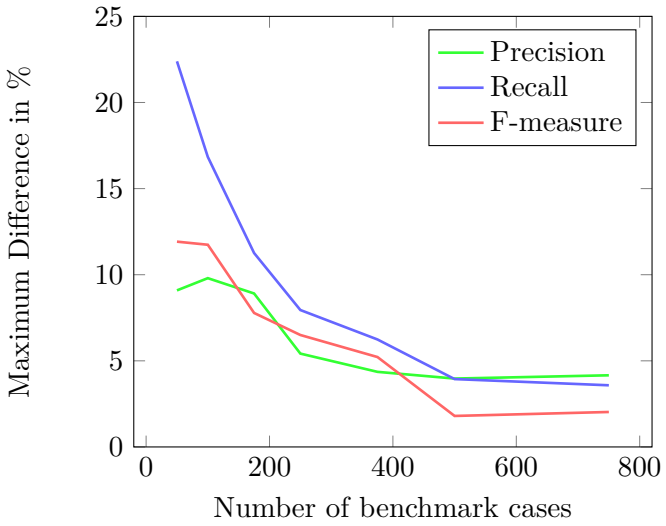


Figure 5.4: Maximum Difference between evaluation iterations with generated benchmark suites

Figure 5.5 shows the standard deviation of the analysis results. The standard deviation describes the variation across all analysis iterations. A low value indicates that all values are close to the mean.

One can observe that the forms of the curves are very similar to the curves in Figure 5.4. There is a big decrease in the standard deviation until 250 benchmark cases are reached. For the precision the biggest part of the decrease stops at this point. However, the recall and the F-measure do still slightly decrease until 500 benchmark cases are reached. From there on all curves stay mostly constant at a standard deviation of around 1%, which indicates very accurate results for every test suite that will be generated with this amount of benchmark cases.

These results show that GENBENCHDROID can generate a meaningful benchmark suite with its fuzzing mode by generating 500 random benchmark cases with an expected deviation from the average of around 1% for precision, recall and F-measure.

However, if the user is only interested in the precision or the F-measure of the analysis tool, 250 benchmark cases are sufficient to generate a meaningful benchmark suite with an expected deviation of less than 2%.

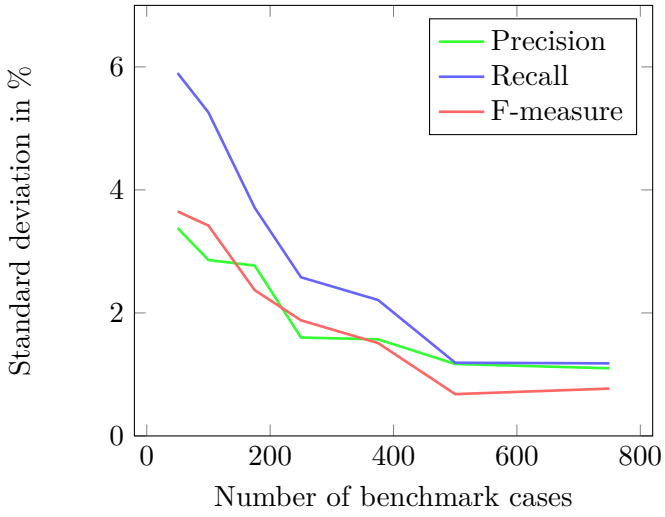


Figure 5.5: Standard deviation between evaluation iterations with generated benchmark suites

5.4 RQ4: Is GenBenchDroid able to generate benchmark cases that are comparable to real-world applications?

To answer RQ4 a case-study was performed. Modules and templates have been created to be able to generate a benchmark case that contains taint flows that are based on taint flows contained inside a real-world application. The generated benchmark case is based on a application from TaintBench [22]. TaintBench is a benchmark suite that consists of 39 real-world Android applications that intentionally try to leak sensitive data from the device. TaintBench contains the source code of these applications, as well as other important information like, for example, the amount of positive, as well as negative, taint flows inside the application and furthermore the various aspects contained in each taint flow.

An application from TaintBench contains, on average, five positive and one negative taint flows. Since there is no application containing exactly five positive and one negative taint flows, the application Samsapo has been selected as archetype for the benchmark case generated by GenBenchDroid for the case-study, which is closest with four positive and one negative taint flows.

Samsapo is a malicious application that sends a hyperlink, leading to the malicious APK-file, to all of the user's contacts and uploads SMS messages, as well as, phone numbers to a remote server. Table 5.3 shows the aspects contained inside each of Samsapo's taint flows. The first taint flow is a positive one that consists of four different aspects. *Array* indicates that the sensitive data is inserted into an array at some point. *Collections* means that the sensitive data is inserted into a Java collections objects. This could, for example, be a `LinkedList` or a `Map` object. *Lifecycle* indicates that there are Android lifecycle callbacks involved in the taint flow. Finally, *threading* indicates that there are threading mechanism involved in the taint flow. All these aspects are included in the taint flow before the sensitive data is leaked to the outside world. The second taint flow consists of all the aspects which are also present in the first taint flow. However, in addition, the sensitive data is appended to a string at some point. The third taint flow consists only of one aspect. At some point, before leaking the sensitive data, reflection is part of the taint flow. The fourth taint flow consists of all aspects that are also present in the second one. While all previous taint flows are positive ones that leak data, the fifth taint flow is a negative one that looks like a taint flow that stretches from a source to a sink, but does not leak sensitive data.

+ = positive taint flow, − = negative taint flow

Taint flow 1 +	**Taint flow 2 +**	**Taint flow 3 +**	**Taint flow 4 +**	**Taint flow 5 −**
array	appendToString	reflection	appendToString	
collections	array		array	
lifecycle	collections		collections	
threading	lifecycle		lifecycle	
	threading		threading	

Table 5.3: Taint flows contained inside the Samsapo application from TaintBench

While the generated benchmark case (configuration can be found in Appendix A.3) is not semantically equivalent to Samsapo, it contains the same amount of positive, as well as negative, taint flows, which all contain the same aspects as Samsapo's taint flows.

Once a module has been created that represents a certain aspect, it can easily be integrated into any taint flow inside the generated benchmark case. This is due to the concept of GenBenchDroid that allows for an arbitrary combination of templates and modules, allowing the creation of arbitrary long and complex taint flows. The created taint flows can easily be extended by further aspects by simply creating new modules or using previously created modules that represent the desired aspect.

This shows that GenBenchDroid is able to generate benchmark cases that contain similar taint flows, which are present in real-world applications.

These generated benchmark cases can also be artificially inflated by inserting modules in between the taint flows that are not related to the sensitive data. This way arbitrary complex benchmark cases, for example, in regard to line of codes or depth of the call graph, can be generated which are even more comparable to real-world applications, since real-world application usually contain a lot of code that is unrelated to the contained taint flows.

5.5 Summary

At first in RQ1 it has been shown that GenBenchDroid is able to regenerate micro benchmark cases from almost all categories of the benchmark suite DroidBench. However, it also has been shown that due to compiler code optimizations one benchmark case from DroidBench could not be regenerated, as the regenerated benchmark case yielded different results when analyzed by FlowDroid.

In RQ2 it has been shown that the generation of a benchmark case consisting of 100 modules or less takes GenBenchDroid around three seconds. This is almost the same time an analysis of the resulting benchmark case by FlowDroid or Amandroid takes. A benchmark suite like DroidBench can therefore be completely regenerated in less than 10 minutes. Furthermore, it has been shown that Amandroid's analysis time, in comparison to FlowDroid's drastically increases for benchmark cases with a high module count, revealing GenBenchDroid's ability to efficiently uncover execution-time related flaws in analysis tools.

Next, in RQ3 it has been shown that a meaningful benchmark suite can be created by using GenBenchDroid's fuzzing mode to generate 500 random benchmark cases, with an expected deviation of around 1% from the average for precision, recall and F-measure.

Finally, in RQ4 GenBenchDroid's ability to generate benchmark cases that are comparable to real-world applications has been demonstrated in form of a case-study. An average malicious application from TaintBench has been selected and a benchmark case containing similar taint

flows as the selected application has been generated.

All research questions could be answered and alongside, the capabilities of GenBenchDroid could be presented.

6 Related Work

Multiple benchmark case generators have been developed for various application areas. Yang et al. have developed a benchmark case generator called CSMITH [52] that aims at testing C compilers for their correctness. It uses a grammar that spans a subset of the C language to generate random, but valid C applications and provides them to a compiler in the hope to uncover bugs. However, CSMITH is only able to generate random applications and also does not provide a ground-truth alongside the generated benchmark case.

Furthermore, there are multiple benchmark case generators that have been developed for the area of energy consumption analysis. Bertran et al. have developed a benchmark case generator called MICROPROBE [27]. MICROPROBE aims at generating benchmark cases that can be used to analyze the energy consumption of multi-core/multi-threaded systems. The generated benchmark cases are represented in an internal representation which is specific to MICROPROBE and can then be synthesized into a representation suiting the desired target system. However, MICROPROBE is limited to generating micro benchmark cases with little complexity. Another benchmark case generator which aims at generating benchmark cases that are suitable for analyzing energy consumption is called GENEE [34], which has been developed by Eichler et al. GENEE, in comparison to MICROPROBE, aims at creating benchmark cases for the analysis of cyber-physical systems. It assembles randomly or by the user selected code-pieces, which are responsible for (de-)activating device's, like sensors or motor drivers, into an application. Alongside the benchmark case, GENEE also generates the applications energy-consumption profile and therefore provides a ground-truth that can be used to verify the energy-consumption analysis of the benchmark case.

A benchmark case generator that has been developed by Wägemann et al. called GENE [48] aims at creating benchmark cases for worst-case execution time analysis tools. GENE uses so called patterns that contain code-snippets and insertion points for further patterns, which are inserted into each other to generate a benchmark case. The user can also provide his/her own patterns. In addition to the benchmark case, GENE generates a file that contains, among other information, value ranges of variables, loop bounds or values triggering the most expensive path contained inside the benchmark case. To generate a benchmark case, the user has to specify desired benchmark characteristics. GENE's design which uses user provided patterns with insertion points is similar to GENBENCHDROID's modular design.

Despite the existence of many benchmark case generators, to the best of our knowledge, there exists no generator that aims at Android app analysis. Currently benchmark suites containing hand-crafted benchmark cases are used to evaluate the performance of Android taint analysis tools. There are several benchmark suites which provide different types of benchmark cases.

DroidBench [13] is a benchmark suite that consists solely of micro benchmark cases which span over a wide variety of categories. However, DroidBench does not provide proper ground-truth information (see Section 2.2.2). It only provides the number of leaks each benchmark case contains. This is an area that GenBenchDroid improves upon by automatically generating detailed ground-truth information for every generated benchmark case. TaintBench [22] is a benchmark suite that contains real-world benchmark cases. It provides 39 malicious applications and detailed information about each taint flow inside them. ICC-Bench [17] is a benchmark suite that aims at ICC. It consists of 24 micro benchmark cases that contain some form of ICC. Like DroidBench, ICC-Bench also only provides the amount of taint flows contained inside each application, instead of more detailed information. To evaluate an Android taint analysis tool usually a combination of these benchmark suites is used.

To overcome the issue of imprecise or missing ground-truth information for taint analysis in general, a tool for vulnerability-injection called LAVA (Large-scale Automated Vulnerability Addition) [33] has been developed by Dolan-Gavitt et al. LAVA uses real-world applications and inserts artificial vulnerabilities into them. These artificially malicious applications can be used to evaluate taint analysis tools, since the vulnerabilities contained inside them are known by the user. Such a vulnerability injection can also be performed by GenBenchDroid by using a real-world application as template with an insertion placeholder for further modules.

Despite no benchmark case generators being available for the area of Android app analysis, there are generators for other Android specific areas available. These generators mostly aim at generating test inputs for Android applications in the hope to uncover bugs contained inside them. Sapienz [42] is a tool that generates test inputs by using a combination of fuzzing and systematic, as well as search-based exploration. IntelliDroid [50] also generates test inputs for Android applications. However, IntelliDroid can be specifically configured to adjust to the capabilities of dynamic analysis tools. DroidBot [41] also is a tool for test input generation for Android applications. However, in contrast to the first two tools, DroidBot does not rely on instrumentation (modification of the original application) and therefore the user does not have to worry about inconsistencies between the tested and the original version of the application.

7 Threats to Validity and Future Work

GenBenchDroid and the concept it is based on still offer some capabilities to be added. GenBenchDroid and the underlying concept do not support the generation of benchmark cases that require external files like, for example, native libraries. By extending the concept by a way to include external files this could be achieved. Furthermore, GenBenchDroid relies on the variable `sensitiveData` that contains the sensitive data throughout the application. This possibly allows for an exploitation by taint analysis tools, which could specifically track the flow of the variable `sensitiveData`. This may lead to an over-adaptation of analysis tools to benchmark cases generated by GenBenchDroid. Additionally, the concept of only one variable containing the sensitive data prohibits the creation of multiple taint flows which run in parallel with different sensitive data. By extending GenBenchDroid's concept by a way to store the sensitive data in arbitrary variables and allowing for a more specific connection between modules (specifying which sensitive variable is used by which module), these capabilities can be added to GenBenchDroid.

Another improvement that can be employed in GenBenchDroid is the integration of a more refined fuzzing procedure. Currently GenBenchDroid utilizes the unparsing functionality of the underlying parser to generate random configurations, in order to only require a single grammar, due to the desired extensibility. However, this implementation requires that GenBenchDroid has to generate multiple configurations and discard the ones that do not fulfill the specified criteria. Instead of this discarding process, a separate fuzzer that only generates configurations with the desired properties could be integrated into the framework of GenBenchDroid. This can either be done by implementing a grammar-based fuzzer which utilizes the grammar that is already used by GenBenchDroid to verify configurations or by using an already proven grammar-based fuzzer like, for example, Mozilla's dharma [20] and employing a model transformation mechanism [47] between the grammar used by GenBenchDroid and the grammar used by the fuzzer, so that the grammar used by the fuzzer will be automatically extended, whenever GenBenchDroid's grammar is extended.

Furthermore, GenBenchDroid's concept can be extended to be able to generate benchmark cases for other platforms than just Android like, for example, iOS.

8 Conclusion

The main goal of the thesis is the development of a concept for an Android app analysis benchmark case generator that is able to generate benchmark cases of arbitrary complexity, which can be used to evaluate the performance of Android taint analysis tools. This goal is achieved by the concept that has been presented in this thesis.

The developed concept is based on a modular design, where each generated benchmark case consists of a template, that denotes the starting point of the application and a set of modules, that denote the content of the application. Furthermore, the concept of TMCs has been introduced in order to be able to specify the desired order of the modules and to generate ground-truth information that precisely describes every taint flow inside the generated benchmark case. Each template and module contains various placeholders that denote the insertion location for the next module. Based on this concept, a template and multiple modules can be combined to generate a desired benchmark case. A benchmark case generator called GenBenchDroid has been implemented based on this modular concept.

An evaluation of GenBenchDroid in regard to different aspects has been performed as well. The evaluation showed that GenBenchDroid is able to regenerate micro benchmark cases from all categories of DroidBench that do not require additional external files. GenBenchDroid is even able to generate benchmark cases with properties that are not contained in DroidBench. Furthermore, GenBenchDroid offers the capability to quickly adjust to new Android API releases, which is problematic in current benchmark suites.

Additionally, it has been shown that GenBenchDroid is able to generate benchmark cases that are comparable to real-world applications in regards to complexity and contained taint flows. Alongside the benchmark case, GenBenchDroid also provides a ground-truth that contains the expected analysis results of the generated benchmark case.

An execution time analysis of GenBenchDroid has been performed and compared to the analysis time of FlowDroid and Amandroid which revealed a vast discrepancy between the analysis time of FlowDroid and Amandroid for large benchmark cases. Because of the easy scalability of benchmark cases generated by GenBenchDroid, these benchmark cases can efficiently be used in analysis time evaluations of Android taint analysis tools.

GenBenchDroid's fuzzing capabilities have been evaluated with the result that at 500 randomly generated benchmark cases a meaningful benchmark suite can be created that is able to provide meaningful data when used to evaluate an analysis tool.

In summary, the concept, and its implementation in the form of GenBenchDroid, are a powerful solution to the current issues that the evaluation of Android taint analysis tools presents, which are (1) implementation mistakes in benchmark cases, (2) over-adaptation of

analysis tools to micro benchmark cases and (3) missing ground-truth information for real-world benchmark cases. GenBenchDroid can easily be extended and adjusted to the user's needs and enable a thorough evaluation of analysis tools in regards to precision, recall and execution time, by efficiently generating benchmark cases of varying complexity and corresponding ground-truths, containing the expected analysis results.

Bibliography

[1] Android Developers Guide: App fundamentals. Accessed 2020-11-17. URL: https://developer.android.com/guide/components/fundamentals#Components.

[2] Android Developers Guide: App Manifest Overview. Accessed 2020-11-18. URL: https://developer.android.com/guide/topics/manifest/manifest-intro.

[3] Android Developers Guide: Declare app permissions. Accessed 2020-11-18. URL: https://developer.android.com/training/permissions/declaring.

[4] Android Developers Guide: Input events overview. Accessed 2020-11-18. URL: https://developer.android.com/guide/topics/ui/ui-events.

[5] Android Developers Guide: Intents and Intent Filters. Accessed 2020-11-18. URL: https://developer.android.com/guide/components/intents-filters#ExampleSend.

[6] Android Developers Guide: Layouts. Accessed 2020-11-18. URL: https://developer.android.com/guide/topics/ui/declaring-layout.

[7] Android Developers Guide: Sign your app. Accessed 2020-11-19. URL: https://developer.android.com/studio/publish/app-signing.

[8] Android Developers Guide: Understand the Activity Lifecycle. Accessed 2020-11-13. URL: https://developer.android.com/guide/components/activities/activity-lifecycle.

[9] Android Studio and Android SDK. Accessed 2021-01-09. URL: https://developer.android.com/studio/.

[10] Argus-SAF. Accessed 2020-12-01. URL: https://github.com/arguslab/Argus-SAF.

[11] Benchmark Refinement and Execution Wizard. Accessed 2020-12-01. URL: https://foellix.github.io/BREW/.

[12] DIALDroid-Bench. Accessed 2020-11-25. URL: https://github.com/amiangshu/dialdroid-bench.

[13] DroidBench 3.0. Accessed 2020-11-25. URL: https://github.com/secure-software-engineering/DroidBench/tree/develop.

[14] FlowDroid. Accessed 2020-12-01. URL: https://github.com/secure-software-engineering/FlowDroid.

[15] Gradle build tool. Accessed 2021-01-20. URL: https://gradle.org/.

[16] Handlebars template engine. Accessed 2021-01-20. URL: https://handlebarsjs.com/.

[17] ICC-Bench. Accessed 2020-11-25. URL: https://github.com/fgwei/ICC-Bench.

[18] Java Development Kit. Accessed 2021-01-09. URL: https://www.oracle.com/de/java/technologies/javase-jdk15-downloads.html.

[19] Mobiel Operating System Market Share Worldwide. Accessed 2021-03-19. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[20] MozillaSecurity dharma. Accessed 2021-03-10. URL: https://github.com/MozillaSecurity/dharma.

[21] Node.js. Accessed 2021-01-09. URL: https://nodejs.org/.

[22] TaintBench. Accessed 2020-11-25. URL: https://taintbench.github.io/.

[23] Usage of IccTA in Flowdroid. Accessed 2020-12-01. URL: https://github.com/secure-software-engineering/FlowDroid/issues/219.

[24] Wrong IccLink error using IccTA. Accessed 2020-12-01. URL: https://github.com/secure-software-engineering/FlowDroid/issues/117.

[25] Maqsood Ahmad, Valerio Costamagna, Bruno Crispo, and Francesco Bergadano. Teicc: targeted execution of inter-component communications in android. In *Proceedings of the symposium on applied computing*, pages 1747–1752, 2017.

[26] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[27] Ramon Bertran, Alper Buyuktosunoglu, Meeta S Gupta, Marc Gonzalez, and Pradip Bose. Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211. IEEE, 2012.

[28] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85, 2017.

[29] Dan Boxler and Kristen R Walcott. Static taint analysis tools to detect information flows. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 46–52. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2018.

[30] Kartik Chandra and Tim Radvan. nearley: a parsing toolkit for JavaScript, 2014. URL: https://github.com/kach/nearley, doi:10.5281/zenodo.3897993.

[31] Xingmin Cui, Jingxuan Wang, Lucas CK Hui, Zhongwei Xie, Tian Zeng, and Siu-Ming Yiu. Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 1–12, 2015.

[32] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[33] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[34] Christian Eichler, Peter Wägemann, and Wolfgang Schröder-Preikschat. Genee: A benchmark generator for static analysis tools of energy-constrained cyber-physical systems. In *Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*, pages 1–6, 2019.

[35] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[36] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[37] Yu Feng, Isil Dillig, Saswat Anand, and Alex Aiken. Apposcopy: automated detection of android malware (invited talk). In *Proceedings of the 2nd International Workshop on Software Development Lifecycle for Mobile*, pages 13–14, 2014.

[38] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[39] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[40] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE, 2015.

[41] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[42] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[43] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.

[44] Felix Pauck, Eric Bodden, and Heike Wehrheim. Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341, 2018.

[45] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186, 2018.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[47] Dániel Varró. Model transformation by example. In *International Conference on Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[48] Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. Gene: A benchmark generator for wcet analysis. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[49] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[50] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[51] Z. Xu, C. Shi, C. C. Cheng, N. Z. Gong, and Y. Guan. A dynamic taint analysis tool for android app forensics. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 160–169, 2018. doi:10.1109/SPW.2018.00031.

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[53] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[54] Daojuan Zhang, Rui Wang, Zimin Lin, Dianjie Guo, and Xiaochun Cao. Iacdroid: Preventing inter-app communication capability leaks in android. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 443–449. IEEE, 2016.

A Appendix

A.1 Provided Templates and Modules

Templates that have been provided:

Name	Description
BasicTemplate	Creates a basic activity. Modules are placed inside the `onCreate` callback.
OnPauseTemplate	Creates a basic activity. Modules are placed inside the `onPause` callback.
OnStartTemplate	Creates a basic activity. Modules are placed inside the `onStart` callback.

Table A.1: Provided templates

Modules that have been provided:

Name	Type	Description
AliasingSanitizerBridge	Bridge (Sanitizing)	Creates two instances of a newly created class. One class receives the sensitive data, while the other does not. Data from class without sensitive data is further propagated.
AppendToStringBridge	Bridge (Basic)	Sensitive data is appended to another string via `StringBuilder`.
ArrayBridge	Bridge (Basic)	Sensitive data is inserted into an array.
ArraySanitizerBridge	Bridge (Sanitizing)	Sensitive data is inserted into an array. However, another field from the array, not containing the sensitive data is further propagated.
AsyncTaskBridge	Bridge (Invoke)	A new `AsyncTask` is created and executed. The sensitive data is passed to the `AsyncTask`'s `doInBackground` method.
BluetoothDetectionBridge	Bridge (Basic)	The device is checked for Bluetooth device's. (This module can be used to check for Emulator behavior.)
ButtonCallbackBridge	Bridge (Invoke)	A button is created and the program flow is continued when the button is pressed.
DatacontainerBridge	Bridge (Sanitizing)	A class with multiple data fields is created. The sensitive data is assigned to one datafield. However, the program flow is continued with data from a data field that does not contain the sensitive data.

Name	Type	Description
IccGlobalFieldBridge	Bridge (Invoke)	A new activity is created that contains a static global field. The sensitive data is assigned to the global field.
ImeiSource	Source	The device's IMEI is stored inside the variable containing the sensitive data.
ImplicitSmsSink	Sink	Based on the length of the sensitive data an SMS with the text "Hello World" is sent. This module implicitly leaks info about the sensitive data (length).
ListBridge	Bridge (Basic)	Sensitive data is inserted into a linked list.
ListCloneBridge	Bridge (Basic)	Sensitive data is inserted into a liked list. Afterwards the list is cloned.
LogSink	Sink	The sensitive data is leaked by writing it to a log.
Obfuscation1Bridge	Bridge (Basic)	Numbers contained inside the sensitive data are converted to characters.
Obfuscation2Bridge	Bridge (Basic)	Numbers contained inside the sensitive data are converted to characters based on ASCII values.
PauseResumeLifecycleBridge	Bridge (Invoke)	Creates an activity with an `onPause` and `onResume` callback. The sensitive data is set up in the `onResume` callback. The program flow continues in the `onPause` callback.
PublicApiPointBridge	Bridge (Basic)	Creates an object of type `Point`. The sensitive data is used to create the object.
RandomIfElseBridge	Bridge (Branching)	Creates an if-else control block and branches the program flow. A random number is generated and based on the result either the if-case or the else-case is invoked. (Both have a 50% chance)
Reflection1Bridge	Bridge (Basic)	Creates a new class and loads the class using reflection. The sensitive data is provided to the loaded class.
SimpleIccBridge	Bridge (Invoke)	Creates a new activity and invokes it by creating an `Intent`. The sensitive data is stored inside the `Intent`.
SimpleRecursionBridge	Bridge (Basic)	Creates a recursive function containing the sensitive data that invokes itself 100 times.
SimpleSanitizationBridge	Bridge (Sanitizing)	Overwrites the variable containing the sensitive data.
SimpleUnreachableBridge	Bridge (Sanitizing)	Creates an if-block that can never be executed.
SmsSink	Sink	Leaks the sensitive data by sending an SMS containing it.

Table A.2: Provided modules

A.2 Used Technologies

Technologies that have been used for the creation of GenBenchDroid:

Name	Description	Used for	Obtainable from
Node.js	A runtime environment for JavaScript applications.	Executing the application	`https://nodejs.org/en`
Dotenv	Module that is able to store and load configuration outside the program-code.	Configuring the application	`https://github.com/motdotla/dotenv`
Handlebars	A logic-less template engine.	Foundation of the Template Engine	`https://handlebarsjs.com`
JS Beautifier	Module that is able to format source code.	Formatting the generated source code	`https://github.com/beautify-web/js-beautify`
Nearley.js	A parser generator.	Foundation of the Verificator	`https://nearley.js.org`
XML-formatter	Module that is able to format XML-files.	Formatting the generated Android Manifest and Layout file	`https://github.com/chrisbottin/xml-formatter`
Xmlbuilder2	Module that helps with the creation of XML-files.	Generating the ground-truth	`https://github.com/oozcitak/xmlbuilder2`
Yargs	Module that helps with the creation of interactive command-line applications.	Providing and reading command-line parameters	`https://yargs.js.org`
Gradle	A build-automation tool. Can be used to compile source code into executable applications.	Compiling the generated source code into an APK-file	`https://gradle.org`

Table A.3: Used technologies for GenBenchDroid

Technologies that have been used for the creation of the GenBenchDroid Editor:

Name	Description	Used for	Obtainable from
Node.js	A runtime environment for JavaScript applications.	Executing the application	`https://nodejs.org/en`
Electron	Framework that allows for the creation of GUIs in Node.js.	Creating the GUI of the application	`https://www.electronjs.org`
React	Library that helps with the creation of performant and reactive GUIs.	Creating the GUI of the application	`https://reactjs.org`
Material-UI	UI library which provides pre-styled React components.	Creating the GUI of the application	`https://material-ui.com`
react-beautiful-dnd (rnd)	Library that provides drag and drop capabilities for lists.	Creating the drag and drop functionality for the Flows section	`https://github.com/atlassian/react-beautiful-dnd`

Table A.4: Used technologies for the GenBenchDroid Editor

A.3 Evaluation Data

Configurations that have been used for generating the benchmark cases to answer RQ1 (see Section 5.1) and RQ4 (see Section 5.4):

Benchmark Case	**Configuration**
FlowSensitivity1	"BasicTemplate ImeiSource AliasingSanitizerBridge SmsSink"
PublicAPIField1	"BasicTemplate ImeiSource PublicApiPointBridge LogSink"
ArrayAccess1	"BasicTemplate ImeiSource ArraySanitizerBridge SmsSink"
Button1	"BasicTemplate ImeiSource ButtonCallbackBridge SmsSink"
Bluetooth1	"BasicTemplate ImeiSource BluetoothDetectionBridge SmsSink"
FieldSensitvity2	"BasicTemplate ImeiSource DatacontainerBridge SmsSink"
Clone1	"BasicTemplate ImeiSource ListCloneBridge LogSink"
ImplicitFlow1	"BasicTemplate ImeiSource Obfuscation1Bridge LogSink ImeiSource Obfuscation2Bridge LogSink"
ImplicitFlow5	"BasicTemplate ImeiSource ImplicitSmsSink"
ActivityCommunication1	"BasicTemplate ImeiSource IccGlobalFieldBridge SmsSink"
ActivityLifecycle4	"BasicTemplate ImeiSource PauseResumeLifecycleBridge SmsSink"
Reflection1	"BasicTemplate ImeiSource Reflection1Bridge SmsSink"
AsyncTask1	"BasicTemplate ImeiSource AsyncTaskBridge SmsSink"
SimpleUnreachable1	"BasicTemplate ImeiSource SimpleUnreachableBridge LogSink"
Samsapo	"BasicTemplate ImeiSource ArrayBridge ListBridge PauseResumeLifecycleBridge AsyncTaskBridge SmsSink ImeiSource AppendToStringBridge ArrayBridge ListBridge PauseResumeLifecycleBridge AsyncTaskBridge LogSink ImeiSource Reflection1Bridge SmsSink ImeiSource AppendToStringBridge ArrayBridge ListBridge PauseResumeLifecycleBridge AsyncTaskBridge LogSink SimpleSanitizationBridge SmsSink"

Table A.5: Used configurations for the evaluation of GenBenchDroid

Detailed results of the fuzzing experiment in RQ3 (see Section 5.3):

50 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
31	3	10	6	91.17%	83.78%	87.32%
36	1	13	7	97.29%	83.72%	90%
38	0	11	5	100%	88.37%	93.87%
32	2	9	12	94.11%	72.72%	82.05%
30	3	14	8	90.9%	78.94%	84.5%
34	1	10	8	97.14%	80.95%	88.31%
39	3	13	2	92.85%	95.12%	93.97%
36	0	9	9	100%	80%	88.88%
42	2	3	8	95.45%	84%	89.36%
37	3	6	7	92.5%	84.09%	88.09%

Table A.6: Results of the fuzzing experiment with 50 fuzzed benchmark cases

100 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
69	4	21	11	94.52%	86.25%	90.19%
67	5	18	11	93.05%	85.89%	89.33%
67	4	21	12	94.36%	84.81%	89.33%
64	8	20	14	88.88%	82.05%	85.33%
75	1	16	11	98.68%	87.2%	92.59%
57	3	22	24	95%	70.37%	80.85%
77	6	17	13	92.77%	85.55%	89.01%
65	5	17	20	92.85%	76.47%	83.87%
73	4	16	17	94.80%	81.11%	87.42%
70	1	14	17	98.89%	80.49%	88.6%

Table A.7: Results of the fuzzing experiment with 100 fuzzed benchmark cases

175 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
111	10	39	31	91.75%	78.16%	84.41%
117	4	32	28	96.69%	80.68%	87.96%
107	13	31	24	89.16%	81.67%	85.25%
121	14	35	19	89.62%	86.42%	88%
115	11	30	33	91.26%	77.7%	83.94%
115	16	33	20	87.78%	85.18%	86.46%
113	12	35	24	90.4%	82.48%	86.25%
129	7	27	16	94.85%	88.96%	91.81%
127	8	23	29	94.07%	81.41%	87.28%
126	10	31	20	92.64%	86.30%	89.36%

Table A.8: Results of the fuzzing experiment with 175 fuzzed benchmark cases

250 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
184	13	33	33	93.4%	84.79%	88.88%
167	11	52	43	93.82%	79.52%	86.08%
165	11	54	36	93.75%	82.08%	87.53%
156	15	57	47	91.22%	76.84%	83.42%
183	12	40	34	93.84%	84.33%	88.83%
178	9	46	33	95.18%	84.36%	89.44%
174	11	46	35	94.05%	83.25%	88.32%
166	17	55	31	90.71%	84.26%	87.36%
158	11	64	33	93.49%	82.72%	87.77%
174	7	51	32	96.13%	84.46%	89.92%

Table A.9: Results of the fuzzing experiment with 250 fuzzed benchmark cases

375 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
248	21	74	65	92.19%	79.23%	85.22%
262	19	70	49	93.23%	84.24%	88.51%
253	25	72	50	91%	83.49%	87.09%
241	18	82	57	93.05%	80.87%	86.53%
245	29	63	68	89.41%	78.27%	83.47%
248	18	69	60	93.23%	80.51%	86.41%
231	26	89	50	89.88%	82.2%	85.87%
241	16	78	63	93.77%	79.27%	85.91%
251	18	61	62	93.3%	80.19%	86.25%
251	18	90	46	93.3%	84.51%	88.69%

Table A.10: Results of the fuzzing experiment with 375 fuzzed benchmark cases

500 fuzzed benchmark cases:

TP	FP	TN	FN	Precision	Recall	F-Measure
318	29	109	72	91.64%	81.53%	86.29%
332	24	101	79	93.25%	80.77%	86.57%
337	39	79	70	89.62%	82.80%	86.07%
335	28	86	85	92.28%	79.76%	85.56%
329	24	113	73	93.20%	81.84%	87.15%
336	23	104	75	93.59%	81.75%	87.27%
310	27	92	77	91.98%	80.1%	85.63%
339	32	85	66	91.37%	83.7%	87.37%
332	27	95	75	92.47%	81.57%	86.68%
318	30	111	76	91.37%	80.71%	85.71%

Table A.11: Results of the fuzzing experiment with 500 fuzzed benchmark cases

750 fuzzed benchmark cases:

TP	**FP**	**TN**	**FN**	**Precision**	**Recall**	**F-Measure**
490	39	171	110	92.62%	81.66%	86.8%
504	46	154	117	91.63%	81.15%	86.08%
499	43	155	121	92.06%	80.48%	85.88%
513	39	144	102	92.93%	83.41%	87.91%
497	36	150	120	93.24%	80.55%	86.43%
501	40	152	98	92.6%	83.63%	87.89%
524	23	126	122	95.79%	81.11%	87.84%
514	37	141	128	93.28%	80.06%	86.16%
491	37	157	113	92.99%	81.29%	86.74%
495	39	153	112	92.69%	81.54%	86.76%

Table A.12: Results of the fuzzing experiment with 750 fuzzed benchmark cases

A.4 Digital Appendix

GenBenchDroid and the GenBenchDroid Editor are attached to this thesis. The versions that are released with this thesis can be found in their individual GitLab repositories.

GenBenchDroid can be found in the following repository:

- `https://git.cs.uni-paderborn.de/sschott/genbenchdroid`
- Branch: main
- Commit: c0720d6e
- Commit-date: Apr 8, 2021

The GenBenchDroid Editor can be found in the following repository:

- `https://git.cs.uni-paderborn.de/sschott/genbenchdroid-editor`
- Branch: main
- Commit: 8d737de8
- Commit-date: Apr 6, 2021