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

Master's Thesis

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Master of Science

Empirical Evaluation of Forward and Backward Static Taint Analysis

by
Amit Kumar

Examiners:
Prof. Dr. Eric Bodden
and
Dr.-Ing. Steven Arzt

Paderborn, June 2, 2023

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

The presence of software vulnerabilities and the potential for their misuse presents significant risks [22] [15] [2] [29]. Over the past decade, taint analysis has gained significant attraction in finding vulnerabilities. Taint analysis can identify most security vulnerabilities - specifically, 17 out of 25 vulnerabilities from SANS/CWE-25 and six taint-style vulnerabilities from the OWASP Top 10 [17]. Due to this trend, well-established academic [30][11][9][1][16] and industrial [24][23] tools provide taint analysis as a dominant feature.

Since most users with malicious intent attack the software by manipulating the input, taint analysis considers this untrusted input as a tainted value and tracks its flow. Taint analysis is a data flow analysis that tracks the flow of data: a taint (user input or private data) from sources to security-sensitive sinks (insecure methods or channels like the internet) which may leak. Taint analysis can be broadly classified into two types: dynamic analysis, where code is monitored as it executes, reasoned upon runtime information, and static analysis, where analysis is performed over source code or compiled code without executing the program and incurring runtime overhead. Dynamic taint analysis cannot provide information about the source code which is not executed while static analysis can achieve high code coverage, including the code which is either unreachable or not executed at runtime. Tools that perform static analysis over the source code or compiled versions of code are called Static Application Security Testing (SAST) tools. In the present study, we will focus exclusively on static taint analysis.

Due to its static property where no run is required for the analysis, SAST tools provide early vulnerability detection in Software Development Lifecycle (SDLC). These tools allow developers to perform analyses while code is being written or compiled rather than delaying for an exhaustive runtime analysis. The developers get feedback from the analyses while they are working on the program, allowing them to remove vulnerabilities early, improving the code quality and overall security of their applications. The data flow analysis can require significant resources and be resource-intensive in terms of runtime, memory usage, especially for large codebases. Efficient taint analysis can make the development process more efficient by reducing the time it takes to find and fix security issues while memory usage optimization favors real-time analysis and detection on machines where such resources are limited, for example: developer's machine.

In general, the data flow of a program is directed, i.e. the analysis is either run in the forward direction of the control flow graph or backward. For instance, constant propagation is typically a forward analysis [28], while the liveness-based analysis runs backward [12]. On the other hand, taint analysis can be performed in both backward and forward directions [9] [1]. Forward taint analysis performs analysis from sources to sinks, while backward taint analysis analyzes taint flows from sinks to sources.

Lerch et al. in FlowTwist [9] stated that a backward analysis is efficient if there is a single sink but a large amount of possible sources. A few taint analysis tools like FlowDroid [1] and SecuCheck [16] can be configured with taint analysis direction before the analysis starts. Given that the taint analysis requires significant resources, it is crucial to select the most efficient direction for the analysis.

Theoretically, it appears that the present number of sources and sinks might affect the taint analysis efficiency. If fewer sources are found than the sinks, the forward analysis should be efficient. Similarly, the presence of a smaller number of sinks compared to sources should favor backward analysis. Therefore, in this work, we empirically evaluate forward and backward static taint analysis to understand the behavior of the taken analysis direction with respect to resource utilization and efficiency.
Therefore, our work studies the effects of taken direction respective to analysis efficiency and try to address the following two research questions:

RQ1 How do the number of sources and sinks affect the efficiency of forward and backward taint analysis, and how to do the two types of taint analysis compare in terms of their overall efficiency?

RQ2 How can we predict efficient taint analysis direction for a given application?

We required a benchmark suite containing many applications with desired properties like the number of present sources, sinks, application code complexity, and more to evaluate, perform analysis, and benchmark. Since finding many applications with the exact required properties is hard in the limited time of this work, we built our benchmark suite with an automatic benchmark application generator tool, GenBenchDroid. GenBenchDroid could only generate benchmark applications with a single tainted variable. Hence, we first extended the GenBenchDroid to generate real-world-like applications with multiple variables having parallel taint flows. Then, we created a benchmark suite with GenBenchDroid, containing 114 applications, including its complete taint flow information and ground truths. To perform an empirical evaluation of taint analysis, we chose two SAST tools: FlowDroid and SecuCheck, because both tools could be configured to run in forward or backward configuration. SecuCheck failed to track taint flow in generated benchmark suite. Therefore, we used FlowDroid in forward and backward configurations. We performed taint analysis over all generated applications for 10 iterations each in the forward and backward direction. With gathered results from the analysis of our generated benchmark suite, we perform empirical evaluation and answer the proposed research questions.

Outline Chapter 2 presents the background, concepts, and tools employed in the empirical analysis. Chapter 3 details the necessary modifications and implementations in GenBenchDroid, and offers a brief overview of the design of applications in the benchmark suite. In Chapter 4, we perform an empirical evaluation of the generated benchmark suite with FlowDroid in both forward and backward direction settings. Subsequently, we discuss the results acquired and address the research questions based on the evaluation. Chapter 5 discusses a comprehensive review of related work. Finally, Chapter 6 outlines our conclusions and provides directions for possible future work.

2 Background

This chapter presents an overview of the various topics and tools utilized throughout the thesis, with a specific focus on benchmarking and taint analysis. It also provides an introduction to the fundamental concepts and applications of taint analysis, as well as the taint analysis direction.

2.1 Taint Analysis

Over the past decade, static and dynamic taint analysis has gained significant attraction in finding vulnerabilities. Taint analysis can detect most security vulnerabilities of SANS/CWE-25 (17 out of 25 vulnerabilities), and OWASP TOP 10 comprises 6 taint-style vulnerabilities [17]. Due to this trend, well-established academic and industrial Static Application Security Testing (SAST) tools like FastDroid [30], COVA [11], FlowTwist [9], FlowDroid [1], and SecuCheck [16] provide taint flow analysis as a dominant feature.
A taint analysis identifies possible vulnerabilities and breaches in a software program. Taint analysis is a data flow analysis that tracks the flow of data: a taint (user input or private data) from sources to security-sensitive sinks like memory, file storage, unsecured methods, and channels like the internet, which may leak. Sources are essentially any point in the program at which data enters the system and it could be a user input or a file. The data from these sources is considered untrusted or tainted. While sinks are points in the program where the source data is used in a potentially dangerous way, such as a function call that executes a SQL query or sending an SMS message. If tainted data makes it to a sink without being properly sanitized or validated, this could lead to a vulnerability. Let us consider the example code snippet in Listing 2.1 which has three sources that take user input in statements 1, 2, and 5 and only one sink function in statement 7 which leaks the user input. Each function call to `source()` returns a secret key input by the user, and the key will leak if it reaches security sensitive `sink()`.

```
1    String x = source();
2    int a = source();
3    int b = a;
4    int c = b;
5    int y = source();
6    x = "";
7    sink(y);
```

Listing 2.1: Code snippet illustrating sources and sinks

Figure 2.1 represents the Exploded Super Graph (ESG) of taint analysis performed over the snippet in Listing 2.1 while tracking the data flow of the value from `source()` to security

sensitive call `sink()`. First, in statement 1, the value from `source()` is assigned to `x`. For the second time, again in statement 2, a call to `source()` is made, and the value is assigned to `a`. After that, the same value is propagated to `b` and then to `c`. Third, the `source()` is called again, and the value is assigned to `y`. In the end, in statement 7, the `sink()` is called with `y`, which contains the sensitive value. The value of `y`, `a`, `b`, `c`, `x` was tainted while `x` was overwritten again at statement 6, hence becoming untainted. Even though value of `y`, `a`, `b`, `c` are tainted at statement 7, the taint analysis would only report statement 7 as a leak since the value of `y` was propagated to security sensitive `sink()`.

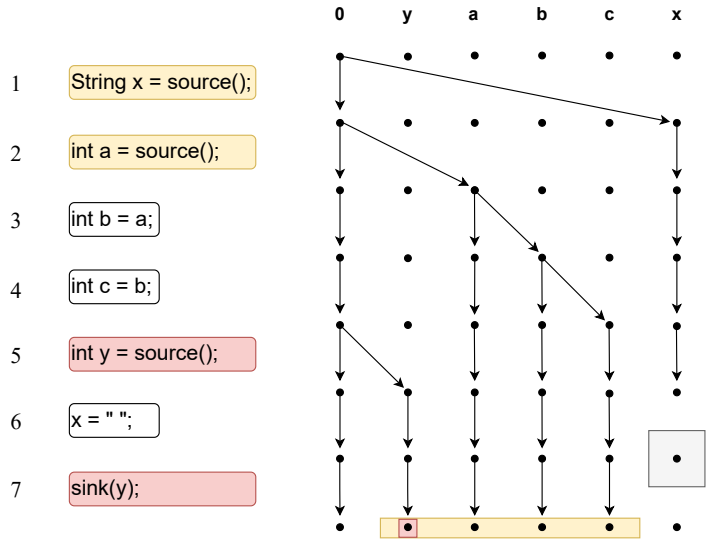

Figure 2.1: Exploded Super Graph of taint analysis over code snippet in Listing 2.1

In general, the data flow of a program is directed meaning that the analysis traverses the data flow graph in the forward direction of the control flow or in backward direction. For instance, constant propagation is a forward analysis [28], while the liveness-based analysis runs backwards [12]. However, taint analysis can be performed in both backward and forward directions [9] [1]. Forward taint analysis investigates the flow from sources to sinks, and backward taint analysis examines the flow from sinks to sources. We will explore it individually in the Chapter 2.1.1 and 2.1.2.

2.1.1 Forward Analysis

Taint analysis presented in Exploded Super Graph (ESG) in Figure 2.1 is a forward analysis. Forward analysis identifies security vulnerabilities by tracking the flow of sensitive or untrusted data (tainted) with the program's execution path called forward flow. Forward taint analysis considers sources of untrusted or sensitive data as tainted values. Statements 1, 2 and 5 in Figure 2.1 present sources that introduce such sensitive values. As the analysis proceeds forward, any data that comes into contact with tainted variables through operations like assignments, function calls, and arithmetic operations are considered tainted. `b` in statement 3 gets tainted once assigned with taint values from `a` and later tainting `c` in statement 4. The analysis continues until a sink is reached where unsafe tainted values are used or handled unsafely.
A potential security vulnerability is identified and reported if the tracked tainted data reaches a sink without proper validation or sanitization. The forward taint analysis would report a leak if the tainted data is used in critical operations and functions without proper validation and sanitization, which leads to security and data breaches, including injection attacks. In statement

7, `sink(y)` is called where `y` contains tainted values, hence a leak will be reported.

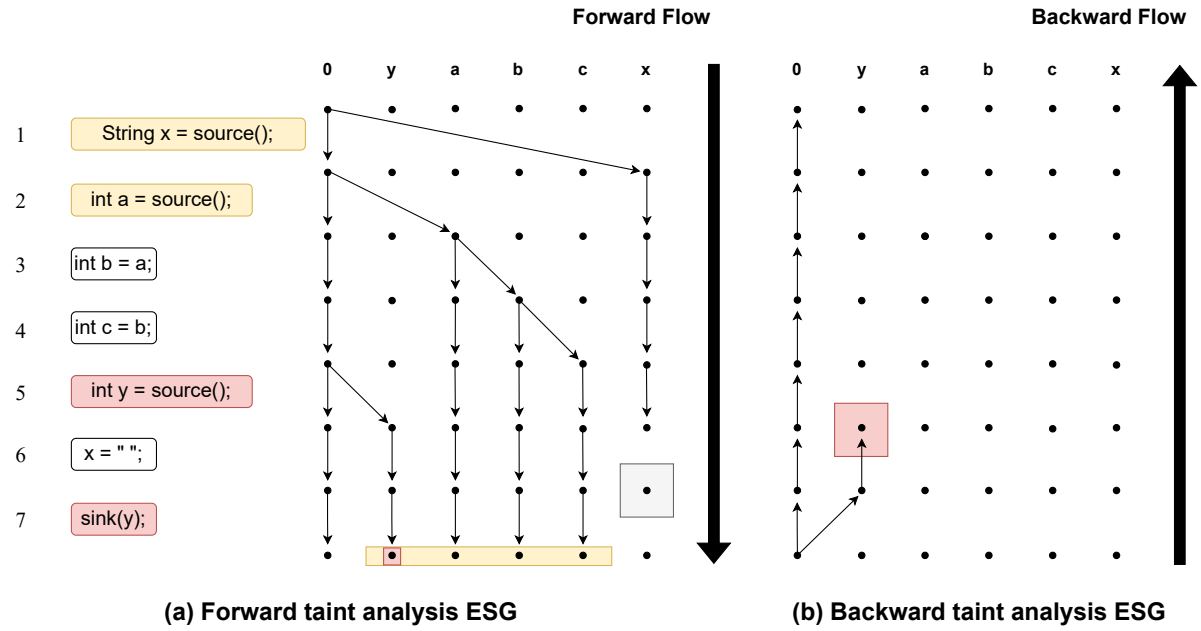

Figure 2.2: ESG of forward and backward taint analysis over code snippet in Listing 2.1

2.1.2 Backward Analysis

Similar to forward taint analysis, backward analysis reports vulnerabilities and leaks if the tainted data leak is traced back to source without undergoing proper validation or sanitization. The backward analysis tracks the data flow counter to the direction of the program flow, commencing from potential security-sensitive sinks and progressing towards the program's origin, or source. The analysis starts from a sink (where sensitive operations are performed), moving backwards in the program following the data flow and control flow to determine the source of the sensitive data used in the sink. Figure 2.2(b) represents backward analysis performed over code snippet in Listing 2.1. Compared to the ESG of forward analysis in Figure 2.2(a), it is noticeable that the flow in the ESG of backward analysis in Figure 2.2(b) is reversed. The analysis starts at the possible sink `sink()` at statement 7, which takes input `y` which is then traced back to statement 5 where `y` is assigned with security sensitive value from `source()` concluding the analysis and reporting the leak.

At a glance, Figure 2.2 shows the difference in the size of ESG of forward and backward taint analyses. In the case of forward taint analysis in Figure 2.2(a), analysis creates many edges to keep track of all taints until it reaches the sink. In Listing 2.1 many sources exist, and it is unclear in the forward analysis whether a source will reach sink until the analysis ends. On the other hand, in backward taint analysis in Figure 2.2(b), the size of ESG is smaller because very few edges exist. The backward analysis starts from the sink and is aware of the leaked value `y`. The analysis tracks the flow back to the source, which generates the leak value `y` ignoring other sources hence generates very few edges.

Also, the analysis does not track `y` anymore because it is undefined before statement 5 and the sink cannot reach another source hence terminating the analysis early. But in forward analysis, the analysis needs to track `y` even after statement 7 because it might reach another sink. Backward analysis performs analysis in the reverse direction of forward analysis and can come to the same conclusion with a smaller ESG, as shown in this particular example.

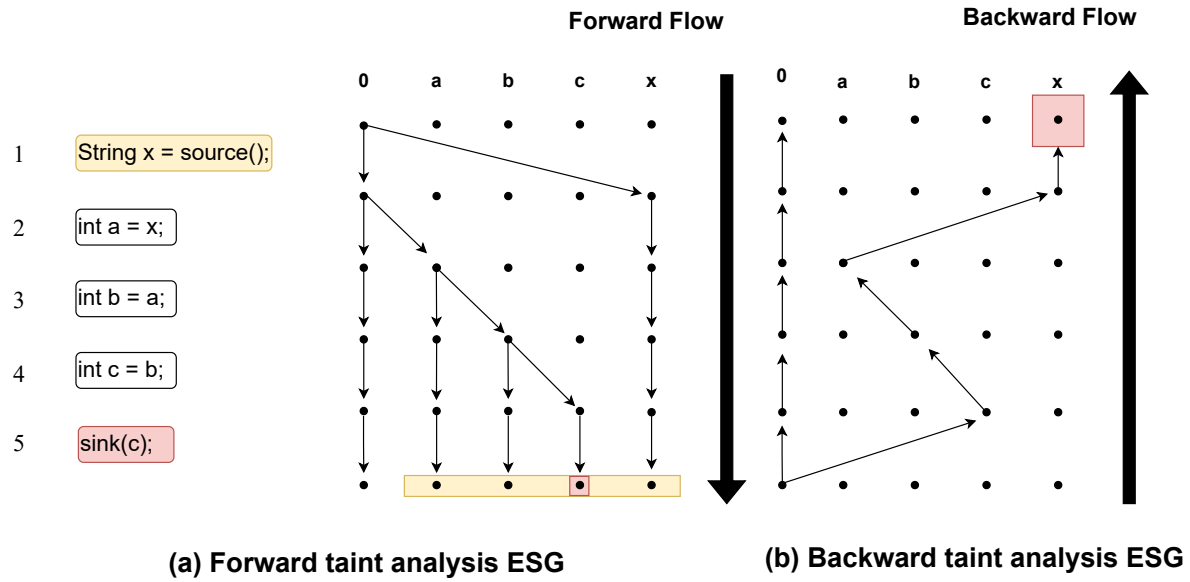

Figure 2.3: ESG presenting Right-to-Left flow with simple add operation

Another well-known property that may favor backward analysis is the right-to-left flow of the taint as presented in Figure 2.3. In general, most taint flows propagate via assign operations where the flow of the program is right to left as in all assignment statements of Figure 2.3. It is observable in backward analysis that the taint is immidiately killed for `a, b, c` since there exists no declaration of a variable while in forward analysis all taints are propagated till the end. If a program taint flow mostly contains assignment operation, the backward analysis may come as efficient compared to forward analysis.
However, both discussed scenarios of backward analysis being efficient is generally not true. Consider the ESG of a simple addition program in Figure 2.4 statement 5, where `add()` performs the addition of provided argument values.
Due to the nature of the operation performed `add()` that adds all the values and returns a sum, the backward analysis will consider all values tainted. Hence, to complete the analysis, backward analysis has to propagate to all declarations, leading many edge propagation compared to forward analysis. On the other hand in forward analysis, the analysis only propagates the tainted value initialized with the source. Hence, forward analysis performs less edge propagation compared to backward analysis.

2.2 FlowDroid

FlowDroid [1] is a static taint analysis tool for Android and Java applications that are context-, flow-, object-, and field-sensitive, providing precise analysis results. It has been actively maintained and utilized by the academic community since its inception. To perform taint analysis, FlowDroid uses the callgraph generated by Soot [27], a Java optimization framework that converts Java and Dalvik bytecode to Jimple intermediate representation. Due to the numerous entry points in Android's activity lifecycle (presented in Figure 2.5), FlowDroid generates a dummy main method as a single entry point for generating callgraph and simplifying modeling. Using its precise modeling of the Android lifecycle, FlowDroid first constructs a semantic model of the application code and then generates a dummy main method as the entry point for analysis. The semantic model includes the application's manifest file, the layout XML files, the

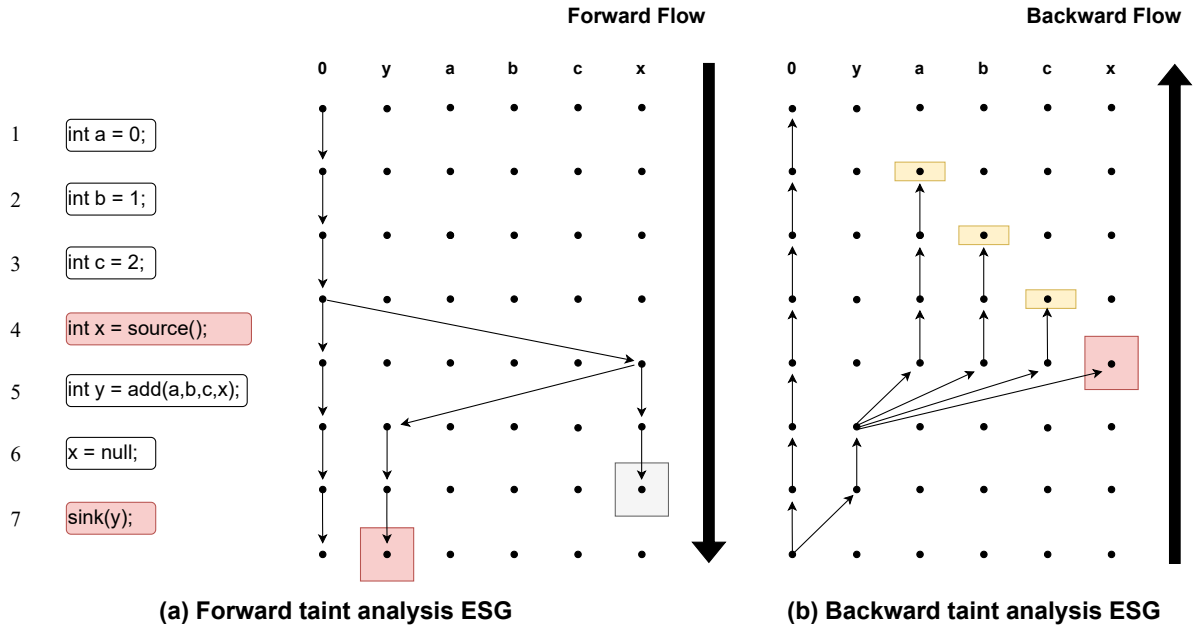

Figure 2.4: ESG of a simple addition program

compiled resources file, and the application's source code, which are all inter-leaved.

FlowDroid employs the IFDS framework [20] to model the taint-analysis problem. It combines analysis results immediately at any control-flow merge point. Consequently, FlowDroid can generate and efficiently analyze a dummy main method that accommodates every possible order of individual lifecycle components and callbacks. This approach eliminates the need to traverse all potential paths. FlowDroid uses the list of sources and sinks generated by an automatic tool SuSi [19], which categorizes sources and sinks to perform taint analysis and identify sources and sinks precisely in the Android framework. Furthermore, FlowDroid also offers the feature of a taint wrapper, which allows to define rules for commonly used methods, for example. *StringBuilder*. The taint wrapper allows FlowDroid to skip the analysis of such methods whose summaries are available in the taint wrapper. FlowDroid can use provided summaries and not analyze the method, reducing the analysis's runtime. These summaries can be created with StubDroid, a FlowDroid extension that allows precomputing of summaries with FlowDroid and can be used for tool-independent use. This feature helps calculate summaries for third-party libraries before the actual analysis, also, the summaries can be computed from the binary distribution of the libraries.

Initially, FlowDroid performed taint analysis in only a forward direction. A recent extension to FlowDroid by Lange [8] allows one to perform taint analysis in a backward direction and configure the analysis direction before the analysis starts. While comparing the results of the analysis performed over 200 applications from the Google Play Store, Lange reported that the backward analysis is as sound as the forward analysis.

The FlowDroid provides numerous settings to modify the analysis performance and constrain it to particular analysis attributes for better performance. For instance, users can choose to take into account or disregard features such as callbacks, reflection, or implicit flow among other aspects. Creating a call graph and subsequent analysis demands substantial resources, mainly regarding runtime and memory. To conduct analysis efficiently, FlowDroid also provides the functionality to set resource limitations. The table 2.1 presents primary settings used to set resource limits.

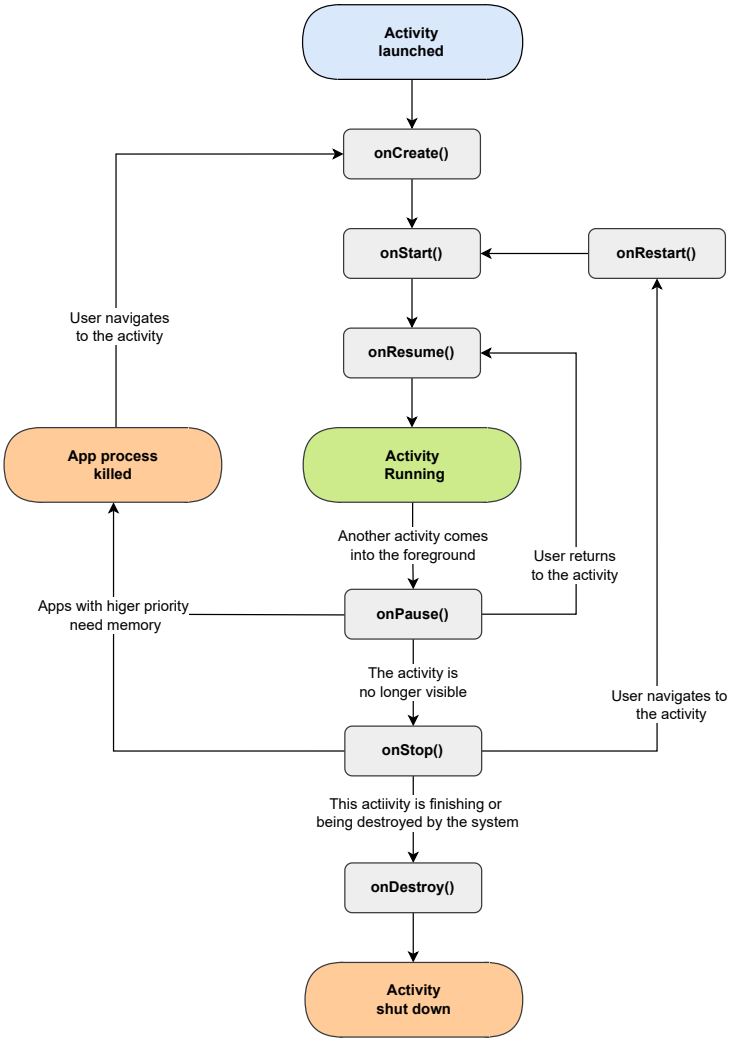

Figure 2.5: A simplified illustration of the activity lifecycle, reproduced from Android developer guide[1]

Configuration option	**Configuration option argument: <arg>**	**Configuration property**
–callbacktimeout	in seconds	Timeout for the callback collection phase
–timeout	in seconds	Timeout for the main data flow analysis
–resulttimeout	in seconds	Timeout for the result collection phase

Table 2.1: FlowDroid settings for resources

FlowDroid is also available in JAR packaged command line interface(CLI) tool, which is easy to use. FlowDroid, in the default configuration, can analyze applications if the taint wrapper summaries and list of sources and sinks are provided. In this work, we will utilize FlowDroid from CLI and perform analysis in its default settings.

[1]source: `https://developer.android.com/guide/components/activities/activity-lifecycle` (visited on 10.05.2023)

2.3 SecuCheck

SecuCheck [16] is a configurable (SAST) tool for Java applications that runs in multiple integrated development environments and is also available as a command-line tool for analysis. SecuCheck was created considering the developer at the center hence making it developer friendly in terms of developer-tool interaction, analysis configuration, and vulnerability reporting that improves the explainability. While the IDE plugin tool is interactive and easy to use, the CLI tool can be used for the scenario where analysis is run for a few minutes to several hours. CLI tool can also be used to perform batch analysis in the build pipeline.
To perform analysis over Java applications, SecuCheck requires configuration for sources and sinks. In addition to sources and sinks definition to analyze the taint flow, SecuCheck allows the configuration of propagators, which are responsible for maintaining the taint in the taint flow, and sanitizers that are responsible for removing the taint from propagated taint flow in the life-cycle of sources and sinks. This allows the tool to use fine-grained taint flow definitions that adjust to real-world application design and reduce false positives in vulnerability detection. SecuCheck uses a Java-internal domain-specific language fluentTQL [18] for specifying taint-flows, and additional YAML settings for its configuration.
FluentTQL allows SecuCheck to configure taint analysis properties like sources, sinks, and propagators, which helps to find violations in the required lifecycle of methods and sanitizers that kill the taint.
Consider fluentTQL specification presented in Listing 2.3. The specification presented is to detect SQL injection vulnerability. The specification will detect and warn the developer on analysis over code snippet presented in Listing 2.2. The specification specifies a definition for a source in statement 4, a sanitizer in statement 8, and a sink in statement 11 for the taint analysis. From such a definition, a complete specification is built that is presented in statements 13-24. According to specification, analysis performed over code snippets in Listing 2.2 will report a `SQLi vulnerability` since, no sanitizer function with name `sanitize(java.lang.String)` was called in the taint flow but sink `executeQuery(java.lang.String)` was called.

```
1  public static ResultSet GetEmployeeInformationWithoutSanitizer(){
2      String employeeID = new Scanner(System.in)nextLine();
3      try {
4          Connection conn= DriverManager.getConnection("jdbc:hsqldb:mem:EMPLOYEES"
       ,"user","pass");
5          Statement stmt = conn.createStatement();
6
7          ResultSet rs= stmt.executeQuery("SELECT * FROM EMPLOYEE where EID ="+
       employeeID);
8      .
9      .
10 }
```

Listing 2.2: Simple code snippet with SQL Injection vulnerability, reproduced from [16]

```
1  @FluentTQLSpecificationClass
2  public class SimpleSQLInjectionSpec implements FluentTQLUserInterface {
3      @OutFlowReturnValue
4      public Method source = new MethodSelector("String nextLine()");
5
6      @InFlowParam(parameterID = {0})
7      @OutFlowReturnValue
8      public Method sanitizer = new MethodSelector("String sanitize(java.lang.
       String)");
9
10     @InFlowParam(parameterID = {0})
```

```
11      public Method sink = new MethodSelector("ResultSet executeQuery(java.lang.
        String)");
12
13      public List<FluentTQLSpecification> getFluentTQLSpecification() {
14          TaintFlowQuery myTF = new TaintFlowQueryBuilder("SQLi vulnerability")
15                  .from(source)
16                  .notThrough(sanitizer)
17                  .to(sink)
18                  .report("SQL Injection - CWE89")
19                  .at(LOCATION.SOURCEANDSINK)
20                  .build();
21
22          List<FluentTQLSpecification> specs = new ArrayList<
        FluentTQLSpecification>();
23          specs.add(myTF);
24          return specs;
25      }
26  }
```

Listing 2.3: fluentTQL specification for detecting simple SQL Injection (the fully qualified names are omitted due to simplicity), reproduced from [16]

Another feature on which SecuCheck focuses is its entry point configuration that allows the handling of multiple entry points. Also, the developer can limit the scope of analysis by selecting individual entry points at the start of the analysis. SecuCheck can be configured to perform analysis with two different data flow solvers, Boomerang Synchronized Pushdown Systems (SPDS) [25] and Flowdroid, which uses the Interprocedural Finite Distributive Subset (IFDS) [20] framework. Boomerang SPDS extends the concept of pushdown systems to model the behavior of programs with recursive and context-sensitive features. It combines context sensitivity, flow sensitivity, and field sensitivity to perform a detailed analysis of the program's data-flow behavior. Boomerang SPDS is a powerful and precise analysis technique that can detect complex program vulnerabilities. In the case of FlowDroid configuration, SecuCheck utilizes FlowDroid in combination with its ability to be configured with different entry points, flexible sources, sinks, and sanitizer configurations. SecuCheck extension by Ozuni [13], allows the SecuCheck to perform backward analysis but only with Boomerang. To allow a change of direction in analysis setting a simple depending on the number of found sources and sink is utilized. Listing 2.4 presents the pseudo-code that decides how the direction of the analysis is chosen. While Ozuni [13] asserts that the backward analysis in his extension exhibits equivalent soundness to forward analysis, it appears that formal documentation and reporting of the tests conducted on the implementation have yet to be presented.

```
1  int sources = getSources () ;
2  int sinks = getSinks () ;
3
4  if( sources <= sinks ) {
5      startForwardAnalysis () ;
6  }
7  else {
8      startBackwardAnalysis () ;
9  }
```

Listing 2.4: Pseudo code for direction setting in SecuCheck extension, reproduced from [13]

2.4 GenBenchDroid

GenBenchDroid[21] is a tool for generating Android applications for benchmarking and test cases with complete ground truth. Benchmarking requires meta-information about the applications called ground truth to compare the observed behavior and reason about the results obtained. With a ground truth, the user can determine whether the analysis tool produced accurate results. Especially, in the case of taint analysis, ground truth can present taint flow, its position in the application, and other information required in benchmarking and analysis comparison. TaintBench [10] includes a benchmark suite with real-world applications including ground truth of the found taint flow in the applications. The ground truth of the TaintBench suite is handcrafted from the decompiled code of the Android application by the taint analysis researchers. The meta-information like ground truth is important for benchmarking. However, generating a ground truth from real-world applications can be time-consuming and labor-intensive, and it does not necessarily ensure an accurate taint flow. Expected properties and behaviour required for benchmarking and tool evaluations is also hard to find in real-world applications. In instances where a few properties of an application match the requirement, there always exist unexpected properties that may hinder the evaluation. Another possibility is to design required benchmark applications manually by hand, which is not feasible for large benchmark suites in the limited time of the research. Due to its modular design, GenBenchDroid also allows us to generate applications with expected properties like specific number of sources, sinks, taint flow design, and more. Almost all properties can be controlled and individually selected for an application. For these reasons, we use GenBenchDroid which generates Android applications, including automatically produced ground truth. GenBenchDroid can generate complex applications comparable to real-world applications, and provide complete ground truth in the form of an AQL-Answer [14] for the generated applications.

An Android application is essentially packaged in an APK(Android Package) format, which is a package file format used by Android operating system for distribution and installation. The APK file is a compressed file, similar to JAR(Java archive) files that include several different types of resource files, and configuration within it. An APK file can contain several different types of files, including DEX (Dalvik Executable) files: compiled bytecode of the application's Java or Kotlin code, manifest file: a special XML file that an Android application must have to contain essential information about the application, android system, application permissions, application components, and different types of resource files. For generating an Android application, the presence of such files and configuration is necessary. Android application logic can be written in a variety of languages but GenBenchDroid generates Android applications based only on Java source code.

GenBenchDroid uses a modular design to generate applications from the provided set of templates and configurations. In addition to the Java class definition, XML resources and settings in the template and module configuration files, GenBenchDroid requires extra meta information about the resources present in the configuration file. It uses meta information provided in the template and module files to calculate the ground truth about the complete application. Along with the designed template developer must provide metainformation about provided logic in the template. Template and module configuration files in GenBenchDroid are provided in the form of JSON. It does not support the addition of external files such as PNG, or data structures files in Android applications but generates applications relying on basic three configuration file types: Java, Android manifest, and XML resource file. All three configurations are generated by the tool by processing its template and module configurations. Figure 2.6 presents an overview of the processing pipeline of GenBenchDroid that takes three inputs, template, modules, and grammar and provides generated Android application with its ground truth as output.

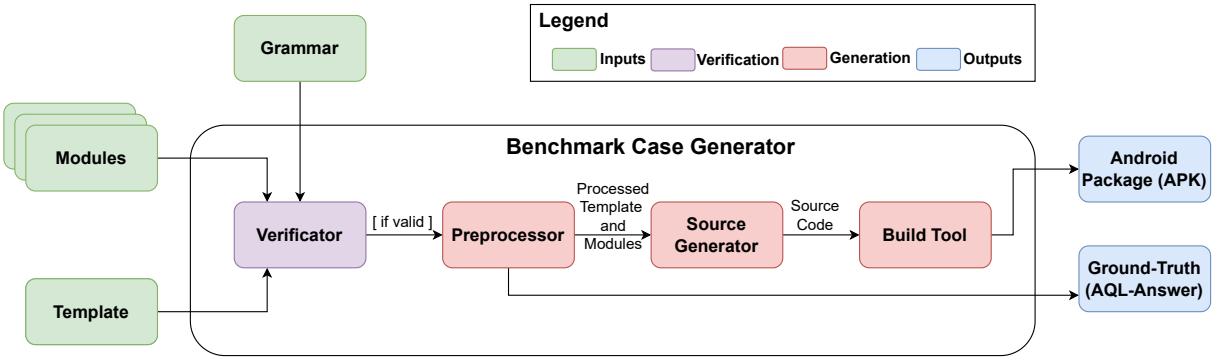

Figure 2.6: Overview of the processing pipeline of the benchmark case generator, reproduced from [21]

Template A template in GenBenchDroid is the genesis of the Android application and is present as a JSON configuration similar to the configuration in Listing 2.5.

```
{
"template": [
"import android.app.Activity;",
"import android.os.Bundle;",
"import android.content.Context;",
"{{ imports }}",
"    public class MainActivity extends Activity {",
"    private Context context = this;",
" {{ globals }}",
"    @Override",
"    protected void onCreate(Bundle savedInstanceState) {",
"        super.onCreate(savedInstanceState);",
"        setContentView(R.layout.activity);",
"        String sensitiveData = \"clear\";",
" {{ module }}",
"    }",
" {{ methods }}",
"}",
"{{ classes }}"
],
"manifest": [
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>",
    "<manifest xmlns:android=\"http://schemas.android.com/apk/res/android\"
    package={{ project }}>",
  " {{ 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>"
],
"layout": [
    "<?xml version=\"1.0\" encoding=\"utf-8\"?>",
    "<RelativeLayout xmlns:android=\"http://schemas.android.com/apk/res/android
    \"",
        "xmlns:tools=\"http://schemas.android.com/tools\"",
```

```
40          "android:layout_width=\"match_parent\"",
41          "android:layout_height=\"match_parent\">",
42   "{{ views }}",
43      "</RelativeLayout>"
44  ],
45  "className": "MainActivity",
46  "methodSignature": "void onCreate(android.os.Bundle)"
47  }
```

Listing 2.5: JSON template configuration named BasicTemplate, from GenBenchDroid tool [21]

A template defines an initial Java class named `MainActivity` in statements 3-19 that contains class definition with its defined android lifecycle `onCreate()`, a resource layout file in statements 37-43, and an initial setting of `AndroidManifest` in statements 22-34. The genesis template has placeholders in statements 6, 9, 15, 17, 19, 24, 32, and 42 at which the module configuration replaces the values. Additionally, the meta-information required by the tool to compile the taint flow ground truth is present in statements 45 and 46.

Modules Modules, on the other hand, extend the initial template by adding values to placeholders defined in its configuration. Consider Listing 2.6 that represent a module configuration of GenBenchDroid that has values for `imports`, `module`, and `permissions` placeholders in the genesis template. If the placeholder does not exist in the Genesis template then the module configuration for the placeholder is ignored. Few meta information like module type of `source` at the top, and the bottom definition of `flows` could be observed in the module configuration which is used to generate ground truth.

```
1  {
2  "type": "SOURCE",
3  "imports": [
4      "import android.telephony.TelephonyManager;"
5  ],
6  "globals": [
7      "{{ globals }}"
8  ],
9  "module": [
10     "TelephonyManager §tm$ = (TelephonyManager) getSystemService(context.
       TELEPHONY_SERVICE);",
11     "sensitiveData = §tm$.getDeviceId();",
12     "{{ module }}"
13 ],
14 "methods": [
15     "{{ methods }}"
16 ],
17 "classes": [
18     "{{ classes }}"
19 ],
20 "permissions": [
21     "<uses-permission android:name=\"android.permission.READ_PHONE_STATE\"/>"
22 ],
23 "flows": [
24     {
25     "className": "",
26     "methodSignature": "",
27     "statementSignature": "android.telephony.TelephonyManager: java.lang.String
       getDeviceId()",
28     "leaking": true,
29     "reachable": true
30     }
```

```
]}
```

Listing 2.6: JSON module configuration named ImeiSource for a taint source, from GenBenchDroid tool [21]

In GenBenchDroid, each module configuration has a defined type and differs in three possibilities: `SOURCE` that defines a source variable or taint in Android application, `BRIDGE` that defines a bridging module between sources and sinks containing operations over introduced sources, and `SINK` that ends the taint flow by leaking the value to security-sensitive sinks. It is important to note the similarities between a `SOURCE` module in Listing 2.6 and a `SINK` module in Listing 2.7. The distinction is used to process module configuration respective to its type and gather meta information like statement location, methods, and class information from the application at build time.

```
{
"type": "SINK",
"imports": [
    "import android.telephony.SmsManager;"  ],
"globals": [
    "{{ globals }}" ],
"module": [
    "SmsManager §sm$ = SmsManager.getDefault();",
    "§sm$.sendTextMessage(\"+49123\", null, sensitiveData, null, null);",
    "{{ module }}"  ],
"methods": [
    "{{ methods }}" ],
"classes": [
    "{{ classes }}" ],
"permissions": [
    "<uses-permission android:name=\"android.permission.SEND_SMS\"/>"  ],
"flows": [
    {
    "className": "",
    "methodSignature": "",
    "statementSignature": "android.telephony.SmsManager: void sendTextMessage(
    java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent
    ,android.app.PendingIntent)",
    "leaking": true,
    "reachable": true
    } ]
}
```

Listing 2.7: JSON module configuration named SmsSink for a taint sink, from GenBenchDroid tool [21]

TMC To create a benchmark application from the template and module configuration, GenBenchDroid requires a configuration, called *template/modules configuration* (TMC). Each TMC will have a template name at the first location, then module configuration follows. The configured template and arrangement of module configuration determine the design and the content of the application after the compilation. The TMC for a simple application with single taint flow including the template from 2.5, source from Listing 2.6 and sink from Listing 2.7 defined as presented in Listing 2.8.

```
BasicTemplate ImeiSource SmsSink
```

Listing 2.8: TMC for a simple application with single taint flow

According to the TMC arrangement, first `BasicTemplate` is parsed and its placeholder is replaced with values from `ImeiSource`, then resulting outputs are replaced with values from `SmsSink`. Additionally, `BRIDGE` modules can be logically classified into four sub-types out of which a type *branching module* stands out. Branching modules divide the existing program flow into multiple program flows, requiring the insertion of multiple follow-up modules. Consequently, a branching module needs to designate not only one placeholder for the next module's insertion location but also an additional placeholder for each extra program branch that may be continued by a follow-up module. Branching modules are commonly utilized to implement diverse control structures such as if-else blocks. Branching modules are distinguished by two additional follow-up modules provided enclosed in parenthesis and demonstrated in Listing 2.9 by a module named `RandomIfElseBridge` containing if and else block.

```
1 BasicTemplate ImeiSource RandomIfElseBridge ( ArrayBridge ) ( SmsSink )
```

Listing 2.9: Incorrect TMC

Listing 2.6 contains configuration for `ImeiSource` that initiates a variable for tainted data flow in statement 11 and Listing 2.7 contains configuration for `SmsSink` that uses the tainted data variable to leak at statement 9. The application generation and its configuration are majorly determined by the TMC provided to GenBenchDroid. To assure that the rules of the template, modules, and its branching settings GenBenchDroid utilizes a grammar-based verification approach.

```
1  <start >         ::= <template> <module>
2  <template>       ::= "BasicTemplate" | ...
3  <linear Module>  ::= "ImeiSource" | "ArrayBridge" | "SmsSink" | ...
4  <2BranchModule>  ::= "RandomIfElseBridge" | ...
5  <3BranchModule>  ::= "3Switch" | ...
6  ...
7  <module>         ::= <linear> | <branching> | ϵ
8  <linear >        ::= <linearModule> <module>
9  <branching>      ::= <2Branches> | <3Branches> | ...
10 <2Branches>      ::= <2BranchModule> "(" <module> ")" "(" <module> ")"
11 <3Branches>      ::=
12     <3BranchModule> "(" <module> ")" "(" <module> ")" "(" <module> ")"
13 ...
```

Listing 2.10: Grammar used by GenBenchDroid for TMC validation

Grammar Listing 2.10 represents the grammar configuration GenBenchDroid uses to verify the TMC. The terminal elements are employed within quotes, and non-terminal elements are enclosed in angle brackets. The grammar commences with a symbol `<start>` , which can be derived into two symbols, a `<template>` and a `<module>`. The `<template>` can be derived into a non-terminal indicating the template utilized in the TMC. The ellipsis denotes all available templates, permitting the `<template>` to be derived into any templates. Unlike `<module>` symbol in statement 7, the `<template>` symbol does not have an empty rule ϵ, indicating that a template must be provided as the initial component of a TMC for it to be deemed a valid TMC. On the other hand, the `<module>` symbol can be derived into three derivations. Depending on the program flow configured in a module, `<module>` can be derived into the `<linear>`, `<branching>` or empty value ϵ. `<linear>` is preferred if the module has no branches, while `<branching>` represents branches present in the program flow of the module.
A linear module only needs a single subsequent module, so the `<linear>` symbol can be directly derived into a `<linearModule>` symbol. The symbol `<linear>` signifies a module with a straight program flow. Beyond the `<linearModule>` symbol, another `<module>` is used, which

can be derived into the next module. Alternatively, if no TMC element exists, the TMC end is represented by ϵ.
On the other hand, a branching module requires multiple subsequent modules based on the number of branches the module generates. The `<branching>` symbol can be derived into the symbol `<2Branches>` if it has two branches, or `<3Branches>` if it contains three branches.
The required follow-up modules are finite for branches, leading to a finite number of possible derivations.
The number of parenthesis pairs in the TMC should correspond to the number of branches in a module. For instance, if a module has two branches, two pairs of parentheses should be utilized in the TMC. Similarly, a module with three branches requires using three pairs of parentheses in the TMC.
For example, a `RandomIfElseBridge` module can create two branches by generating one if-case and one else-case in the module. Similarly, a 3Switch module can produce three branches by creating a switch-case Statement with three distinct cases. The grammar is easily expandable, which facilitates the proper extensibility of the generator. Due to this, templates or modules can be incorporated into the provided grammar effortlessly with minimal required modifications. The grammar is extensible and it is an additional input to GenBenchDroid, including modules and templates. However, the grammar is only modified when users add more modules and templates to GenBenchDroid.

Verificator The Verificator works as the first component in GenBenchDroid's processing pipeline. It utilizes the grammar outlined in Listing 2.10 to validate the given TMC. The verification process involves checking for a potential derivation tree utilizing the grammar of the provided TMC. If such a derivation exists, the TMC is deemed valid or else invalid. Figure 2.7 illustrates an example of the derivation tree for the TMC defined in Listing 2.8.

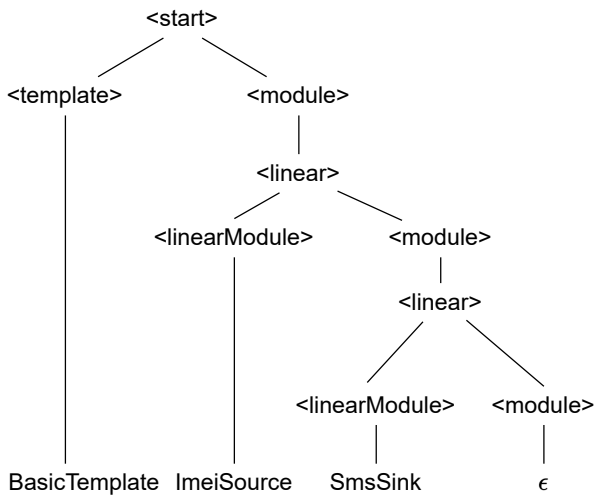

Figure 2.7: Derivation tree for the TMC defined in Listing 2.8, reproduced from [21]

Preprocessor The preprocessor in GenBenchDroid plays an important role. The preprocessor is responsible for handling Java source code for Android applications, including resolving conflicts, managing permissions and imports, tracking code branches, and generating ground truths for benchmarking. Java doesn't allow multiple variable declarations with the same identifier in the same scope. However, this could happen in GenBenchDroid if the same module declaring a variable is inserted twice, which would result in invalid Java source code. To prevent

this, the preprocessor adds a unique numeral identifier to each variable, except for the variable `sensitiveData` which is used to track sensitive data. The preprocessor distinguishes such variables using a combination of a pre-symbol § and a post-symbol $ that are attached to them. For example, at statements 10, 11 identifiers `tm` in Listing 2.6 and at statements 8, 9 identifiers `sm` in Listing 2.7. Also, the preprocessor ensures that required imports and permissions are parsed into Java code without duplications. If a branching module creates a new branch in the source code, the preprocessor assigns a unique identifier to each placeholder and the associated module. This aids the source generator in deciding where each module should be inserted. The preprocessor generates a corresponding ground-truth graph for each benchmark case, tracking every taint flow within the generated application. With the generated graph, preprocessor can identify if a module sanitizes the taint flow, changing it from a positive to a negative one with the help of meta-information. All functions of the preprocessor are dependent on the meta-information provided in the module configuration, for example, to mark a source and sink of taint flow `type` information provided in JSON module configuration is used, `import` for import handling, and values provided in `flows` are used to calculate the flow branch and respective flow leak and code reachability.

Source Generator The source generator inserts modules into the template to generate valid Android application source code. This process executes in the order specified by the TMC in left to right order, with branching modules inserted in a breadth-first manner. When a module is processed, its code snippets replace the corresponding placeholders in the template, and the placeholders provided by the template are replaced with those from the module. If the module leaves any placeholders empty, the previous placeholders are retained. This process continues until all modules from TMC have been inserted. Finally, a cleanup of any remaining placeholders is conducted, and all newly created public classes are split into their source files to allow the successful compilation of the Android application. The source generator leverages a JavaScript library *handlebars.js* to efficiently manage placeholders. *handlebars.js* is known for its effective templating capabilities, making it a valuable tool in Android applications' source code construction.

Build Tool The compilation process of the Android application is conducted using the Gradle build tool. Gradle is the most commonly utilized for compiling Android applications, primarily because it is the default option offered by Android Studio. Gradle's function include generation of an executable Android application (an APK file) with provided source code and resource files. Respective ground truth is calculated by the preprocessor along the Android application.

Limitations GenBenchDroid depends on a variable `sensitiveData` to hold sensitive information and create taint flows throughout the application as presented in statement 14 in Listing 2.5. Taint analysis tools could exploit this property, as they can track the flow of the `sensitiveData`. In the implementation of GenBenchDroid, a random string generated from English alphabets is used to disguise the name `sensitiveData` as presented in statement 10 of Listing 2.12. Although, the identifier is disguised, the same identifier is used throughout the application. The possibility of such string generation is limited and tends to fail while a large number of modules are used in application generation. Furthermore, this property still could potentially hint analysis tools to over-adapt to the benchmark cases created by GenBenchDroid as all taint uses the same identifier with a random string. Moreover, relying on a single identifier for propagating taint restricts the ability of GenBenchDroid to generate multiple parallel taint flows. Consider the TMC presented in Listing 2.11 that generates a simple Android application featuring two taint flow as presented in Java class in 2.12.

```
BasicTemplate ImeiSource SmsSink ImeSource SmsSink
```

Listing 2.11: TMC for a simple application with double taint flow

```
public class MainActivity extends Activity {
    private Context context = this;

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity);
        String HNuSIdrOQSeEyeAF = "clear";
        TelephonyManager tm0 = (TelephonyManager) context.getSystemService(
    Context.TELEPHONY_SERVICE);
        HNuSIdrOQSeEyeAF = tm0.getDeviceId(); // statementId: 0
        SmsManager sm1 = SmsManager.getDefault();
        sm1.sendTextMessage("+49123", null, HNuSIdrOQSeEyeAF, null, null); //
    statementId: 1
        TelephonyManager tm2 = (TelephonyManager) context.getSystemService(
    Context.TELEPHONY_SERVICE);
        HNuSIdrOQSeEyeAF = tm2.getDeviceId(); // statementId: 2
        SmsManager sm3 = SmsManager.getDefault();
        sm3.sendTextMessage("+49123", null, HNuSIdrOQSeEyeAF, null, null); //
    statementId: 3
    }
}
```

Listing 2.12: Java configuration generated by GenBenchDroid with TMC from Listing 2.11

In statement 7, we can observe the GenBenchDroid preprocessor declaring an identifier, which is disguised as `HNuSIdrOQSeEyeAF`, instead of the actual identifier `sensitiveData`. At statement 9, the identifier is assigned with a taint, and then at statement 11, it leaks, resulting in the first taint flow. Similarly, there is a second taint flow from a source at statement 13 to a sink at statement 15. Both taints flow use the same identifier, `HNuSIdrOQSeEyeAF`, and they occur in series, with the first one happening before the second one. This limitation prevents the simulation of real-world applications, where taint flow can propagate in parallel and utilize different identifiers.

Similarly, over-dependence on the use of `sensitivedata` to propagate the taint flow limits the application to mimic real-world behavior. Consider the module configuration named `ArrayBridge` presented in Listing 2.13.

```
{
    "type": "BRIDGE",
    "imports": [],
    "globals": [
        "{{ globals }}"
    ],
    "module": [
        "String[] §arrayData$ = new String[3];",
        "§arrayData$[0] = \"element 1 is tainted\";",
        "§arrayData$[1] = sensitiveData;",
        "§arrayData$[2] = \"neutral text\";",
        "sensitiveData = \"clear\";",
        "sensitiveData = §arrayData$[1];",
        "{{ module }}"
    ],
    "methods": [
        "{{ methods }}"
    ],
    "classes": [
        "{{ classes }}"
```

```
    ],
    "permissions": [],
    "components": [],
    "views": [],
    "flows": [
    {
        "className": "",
        "methodSignature": "",
        "statementSignature": "",
        "leaking": true,
        "reachable": true    } ]
}
```

Listing 2.13: JSON module configuration named ArrayBridge for taint propagation, from GenBenchDroid tool [21]

The taint from `sensitiveData` is assigned to the first element of the array in statement 10. To continue the propagation, the taint is re-assigned as the string `"clear"` in statement 12 and tainted again in statement 13. However, using the same variable that initiated the taint flow for further propagation is not suitable for real-world applications. Instead of using `sensitiveData` in statements 12 and 13, another identifier should be employed to carry on the taint flow.
Another shortcoming of GenBenchDroid is its limited handling of program structures. Consider, for instance, the TMC configuration from Listing 2.9, where a module configuration `RandomIfElseBridge`, is used to integrate two additional modules `ArrayBridge` and `SmsSink`. This occurrence is due to the presence of two separate placeholders at statements 9 and 11, as demonstrated in the JSON module configuration in Listing 2.14. The module configuration from `ArrayBridge` will be placed at statement 9 and the module configuration from `SmsSink` will be placed at statement 11. However, due to the absence of a third placeholder, the program flow cannot be extended to incorporate additional modules beyond the scope of if-else module `RandomIfElseBridge`. This limitation restricts the flexibility of application design and the representation of real-world taint flow.

```
{
    "type": "BRIDGE",
    "imports": [],
    "globals": [
        "{{ globals }}",
        "{{ globals }}" ],
    "module": [
        "if (Math.random() > 0.5) {",
        "  {{ module }}",
        "} else {",
        "  {{ module }}",
        "}" ],
    "methods": [
        "{{ methods }}",
        "{{ methods }}" ],
    "classes": [
        "{{ classes }}",
        "{{ classes }}" ],
    "permissions": [],
    "components": [],
    "views": [],
    "flows": [
        {
            "className": "",
            "methodSignature": "",
            "statementSignature": "",
```

```
27              "leaking": true,
28              "reachable": true
29          },
30          {
31              "className": "",
32              "methodSignature": "",
33              "statementSignature": "",
34              "leaking": true,
35              "reachable": true
36          }  ]
37  }
```

Listing 2.14: JSON module configuration named RandomIfElseBridge for a branch, from GenBenchDroid tool [21]

Above all discussed limitations are mentioned in the GenBenchDroid's future work and are necessary for the generation of a benchmark suite with applications closely comparable to real-world applications with taint flows.

3 Implementation

While the previous Chapter 2 focused on the introduction to taint analysis, SAST tools, and GenBenchDroid, this chapter discusses the necessary modifications performed over the tools. First, we outline the extension made to GenBenchDroid, which removes its limitations, allowing it to flexibly generate parallel multi taint flow benchmark applications. Next, we present a benchmark suite named Taint Analysis Benchmark Suite (TABS) generated by extended GenBenchDroid for empirical evaluation of forward and backward taint analysis. At the end of this chapter, we discuss changes and fixes made to SecuCheck's backward implementation, allowing analysis direction to be configured from YAML configuration before analysis starts.

3.1 GenBenchDroid extension

This section outlines the enhancements and adaptations applied to both the concepts and the implementation of GenBenchDroid. The objective of these changes is to address the limitations detailed in Chapter 2.4, thereby facilitating the generation of the necessary benchmark applications. Specifically, we focused on the three primary limitations previously noted:

1. The original design of GenBenchDroid was such that it generated applications featuring a single variable taint flow with identifier `sensitiveData`.
2. GenBenchDroid reinitializes the taint flow identifier `sensitiveData` repeatedly to propagate the taint.
3. The design flexibility of applications produced by GenBenchDroid was limited.

We address these three limitations by adaptation and extensions to GenBenchDroid, as presented in Chapter 3.1.1.

3.1.1 Limitation 1: single variable taint flows

Repeated use of the same modules in GenBenchDroid's TMC can result in duplicate identifier declaration in Java, leading to a compilation failure. GenBenchDroid adds numeral identifiers to such duplicate declarations to prevent this, except for the variable `sensitiveData`. The variable `sensitiveData` is declared in the template, for example in `BasicTemplate` in Listing 2.5 at statement 14, and further tainted in module configuration in Listing 2.6 statement 11. To allow the possibility of varying identifiers in taint flow, multiple variables like `sensitiveData`

with different identifiers should exist as well. Hence, we first extended the preprocessor to add numeral identifiers to `sensitiveData` to enable multiple taint flow. This was achieved in three steps. First, we moved the declaration of identifier `sensitiveData` from the template to individual modules that start the taint flow for example, a module with TMC named ImeiSource. Listing 3.1 presents the changes in statement 12.

```
{
    "type": "SOURCE",
    "pattern": "OUT",
    "imports": [
        "import android.telephony.TelephonyManager;"
    ],
    "globals": [
        "{{ globals }}"
    ],
    "module": [
        "TelephonyManager §tm$ = (TelephonyManager) getSystemService(context.
    TELEPHONY_SERVICE);",
 "String sensitiveData_ϵ = tm.getDeviceId();",
        "{{ module }}"
    ],
    "methods": [
        "{{ methods }}"
    ],
    "classes": [
        "{{ classes }}"
    ],
    "permissions": [
        "<uses-permission android:name=\"android.permission.READ_PHONE_STATE
    \"/>"
    ],
    "flows": [
        {
            "className": "",
            "methodSignature": "",
            "statementSignature": "android.telephony.TelephonyManager: Java.lang
    .String getDeviceId()",
            "leaking": true,
            "reachable": true
        }
    ]
}
```

Listing 3.1: Modified JSON module configuration named ImeiSource, from GenBenchDroid tool [21]

Second, to have more control over the designed taint flow, we separated the logic of numeral identifiers for sensitive data from rest of the identifiers. The modules of type source, like `ImeiSource`, get the numeral identifiers for `sensitiveData` from TMC. We modified the preprocessor logic to extract the numeral identifier from the module present in TMC. Consider the TMC for a simple application with modified changes in Listing 3.2.

```
BasicTemplate ImeiSource1 ImeiSource2 ImeiSource3
```

Listing 3.2: TMC for a simple application with single taint flow

The preprocessor extracts the suffix numeral digit from TMC configuration in Listing 3.2 **1** and adds it to the identifier `sensitiveData` present in the Java code of the module, for example, in Listing 3.1 statement 12. The presented TMC will create an application with three different identifiers sensitiveData1, sensitiveData2, and sensitiveData3. Third, to enable reliable

obfuscation of the identifier `sensitiveData` and remove the possibility of over-adaptation of the analysis tool, a new logic was introduced. Instead of a random string of alphabet, we chose to use the MD5 hash of the complete identifier, including the suffix numeral identifier with additional meta-information like the numeral id of TMC. The numeral id of a module in TMC is defined by its position in TMC, for example, in Listing 3.2 BasicTemplate holds id 0, ImeiSource1 holds id 1, and SmsSink1 holds id 2.

3.1.2 Limitation 2: repeated reinitialization of tainted variable `sensitiveData`

To enable multiple taint flows, the existence of multiple variable identifiers is necessary and was achieved by removing the limitation 1. But the problem of excessive use and reliance on identifier `sensitiveData` is still an issue, as presented in statements 12, 13 in Listing 2.13 of JSON module configuration `ArrayBridge`. To resolve this problem, we extended the numeral identifier concept built to solve Limitation 1 to all modules and created a concept of virtual taint flow in TMC. Consider the TMC configuration presented in Listing 3.3. With the present numeral identifier, the TMC element can be categorized into two taint flows since only 1 and 2 are suffix numeral identifiers to the module. The first element of all TMC, template element, is an exception as there exists no `sensitiveData` identifier declaration in the configuration since it was moved to module `ImeiSource`. `ImeiSource1` and `ImeiSource2` will generate two different `sensitiveData` identifiers, and `ArrayBridge1` will only propagate the taint from `ImeiSource1` to `SmsSink1` without disturbing the flow created by `ImeiSource2`.

```
1 BasicTemplate ImeiSource1 ImeiSource2 ArrayBridge1 SmsSink2 SmsSink1
```

Listing 3.3: TMC for a simple application with single taint flow

Now, numeral identifiers can be assigned to all elements of TMC, including templates and modules.
By assigning a numeral ID to each element in TMC, we can determine which module made changes, sanitized, or propagated the taint flow to a specific version of `sensitiveData`. If a numeral identifier is not provided for a TMC configuration element, its numeral value is set to 0. This behavior led to the development of an additional extension to GenBenchDroid, which enables the generation of variable identifiers on the go for propagators such as `ArrayBridge` in Listing 2.13. GenBenchDroid supports three types of JSON module configuration, namely `SOURCE`, `BRIDGE`, and `SINK`, defined by the `type` attribute as shown in Listing 3.1, line 2. Since there is no established convention regarding propagators that may require these on-the-go variables, we introduced a new definition in the JSON module configuration called `pattern`. The `pattern` property can be set to two options: `IN` and `OUT`. When `pattern` is set to `OUT`, the JSON module configuration generates a new MD5 hash for the `sensitiveData` identifier, creating a new identifier. On the other hand, when `pattern` is set to `IN`, the configuration only utilizes the generated identifier in the module configuration. For instance, the JSON module configuration for ImeiSource in Listing 3.1 demonstrates this extended feature with the `pattern` set to `OUT`. By incorporating this extended feature, the TMC configuration from Listing 3.3 generates the Java settings depicted in Listing 3.4.

```
1 protected void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(R.layout.activity);
4     TelephonyManager tm0 = (TelephonyManager) getSystemService(context.
    TELEPHONY_SERVICE);
5     String a297034ddcda4a1b1b1dccbc7ad749f02 = tm0.getDeviceId(); // statementId
    : 0
6     TelephonyManager tm1 = (TelephonyManager) getSystemService(context.
    TELEPHONY_SERVICE);
```

```
7       String a160fd8078374209957eeb0da036786d5 = tm1.getDeviceId(); // statementId
        : 1
8       String[] arrayData2 = new String[3];
9       arrayData2[0] = "element 1 is tainted";
10      arrayData2[1] = a297034ddcda4a1b1b1dccbc7ad749f02;
11      arrayData2[2] = "neutral text";
12      String a4a30f7748f2d9885a4c4b66f08c79b8d = arrayData2[1];
13      SmsManager sm3 = SmsManager.getDefault();
14      sm3.sendTextMessage("+49123", null, a160fd8078374209957eeb0da036786d5, null,
         null); // statementId: 3
15      SmsManager sm4 = SmsManager.getDefault();
16      sm4.sendTextMessage("+49123", null, a4a30f7748f2d9885a4c4b66f08c79b8d, null,
         null); // statementId: 4
17  }
```

Listing 3.4: Java configuration generated by extended GenBenchDroid with TMC from Listing 3.3

All the changes mentioned earlier can be witnessed in Listing 3.4. Instead of utilizing sensitiveData1 in statement 5, a MD5 hash of identifier is used. Furthermore, in statement 5, 7, and 12 three different MD5 hash can be observed replacing the previously generated one.

3.1.3 Limitation 3: limited flexibility in application program generation

As shown in Listing 2.14, GenBenchDroid, in its current configuration, lacks the necessary flexibility for generating real-world applications. While the primary issue is branching, the limitations also extend to the design implemented through TMC. By improving TMC, both the limitations in design and branching could be addressed. To illustrate this, consider the TMC and its associated JSON module configuration presented in Listing 3.5 and Listing 3.6. The JSON module configuration, IccGlobalFieldBridge, generates a new class and maintains the taint flow through an intent to the newly created class. However, the flow is continued in the new class, it becomes impossible to configure other flows in the old class. Consequently, this restricts the flexibility of the taint flow to emulate real-world applications.

```
1  BasicTemplate ImeiSource IccGlobalFieldBridge SmsSink
```

Listing 3.5: TMC for a simple application with single taint flow

```
1  {
2      "type": "BRIDGE",
3      "imports": [
4          "import android.content.Intent;"
5      ],
6      "globals": [],
7      "module": [
8          "Intent §i$ = new Intent(context, §NextActivity$.class);",
9          "§i$.putExtra(\"leak\", sensitiveData_e);",
10         "startActivity(§i$);"
11     ],
12     "methods": [],
13     "classes": [
14         "public class §NextActivity$ extends Activity{",
15         "   private Context context = this;",
16         "  {{ globals }}",
17         "  public void onCreate(Bundle savedInstanceState) {",
18         "    super.onCreate(savedInstanceState);",
19         "    setContentView(R.layout.activity);",
20         "    Intent §i$ = getIntent();",
21         "    String sensitiveData_ = §i$.getStringExtra(\"leak\");",
```

```
        "    {{ module }}",
        "  }",
        "{{ methods }}",
        "}",
        "",
        "{{ classes }}"
    ],
    "permissions": [],
    "components": [
    "    <activity android:name=\".§NextActivity$\">",
    "      <intent-filter>",
    "        <action android:name=\"android.intent.action.MAIN\"/>",
    "        <category android:name=\"android.intent.category.LAUNCHER\"/>",
    "      </intent-filter>",
    "    </activity>"
    ],
    "flows": [
        {
            "className": "§NextActivity$",
            "methodSignature": "void onCreate(android.os.Bundle)",
            "statementSignature": "",
            "leaking": true,
            "reachable": true
        }
    ]
}
```

Listing 3.6: Modified JSON module configuration named ImeiSource, from GenBenchDroid tool [21]

To overcome this limitation, we utilized a logical structure known as Branches, as mentioned in Listing 2.10. To allow for flexible TMC configuration, we replicated most JSON module configurations with single flow to grammar element `2Branches` and introduced new modules, identified by the prefix **D**. Additionally, modules with grammar element `2Branches` were replicated to `3Branches`. For example, if we consider the standard module `ArrayBridge`, we also created a corresponding module called `DArrayBridge`. Similarly, module configurations with a single placeholder like ImeiSource can now accommodate two branches, and the if-else branch, which traditionally had two branches, can now have three. Two for if-else, and additionally one for continuing the program flow.

```
BasicTemplate ImeiSource1 DArrayBridge1 ( IccGlobalFieldBridge1 SmsSink1  ) (
    SmsSink1 )
```

Listing 3.7: TMC for a simple application with single taint flow

Complete improved grammar is presented in Listing 3.8. In addition to modules with linear settings, a replica of each is created to support two branches (2Branches) as present in statement 55-78. If-else modules with two branches can now support three to extend the program design. With the improved grammar and additional double modules, the TMC in Listing 3.7 showcases an enhanced application design. It displays two taint flows: one concludes in a new class created by IccGlobalFieldBridge1, and the other remains within the same class where it can be continued.

```
start -> template | template __ module

linear -> __ module | null

2Branches ->  _ "(" innerModule ")" _ "(" innerModule ")"

3Branches ->  _ "(" innerModule ")" _ "(" innerModule ")" _ "(" innerModule ")"
```

```
8
9  innerModule -> _ module _ | _
10
11 _ -> [ ]:*
12
13 __ -> " "
14
15 # Insert position of new templates or modules
16
17 template ->
18     "BasicTemplate" |
19     "OnStartTemplate" |
20     "OnPauseTemplate"
21
22 module ->
23     "ImeiSource" linear |
24     "EmptySource" linear |
25     "AliasingSanitizerBridge" linear |
26     "AsyncTaskBridge" linear |
27     "ArraySanitizerBridge" linear |
28     "BluetoothDetectionBridge" linear |
29     "ButtonCallbackBridge" linear |
30     "DatacontainerBridge" linear |
31     "IccGlobalFieldBridge" linear |
32     "IccInactiveActivity" linear |
33     "IccParcel" linear |
34     "ListCloneBridge" linear |
35     "Obfuscation1Bridge" linear |
36     "Obfuscation2Bridge" linear |
37     "PauseResumeLifecycleBridge" linear |
38     "PublicApiPointBridge" linear |
39     "Reflection1Bridge" linear |
40     "ReflectionMethod1Bridge" linear |
41     "ReflectionMethod1NonSink" linear |
42     "SimpleIccBridge" linear |
43     "SimpleRecursionBridge" linear |
44     "SimpleSanitizationBridge" linear |
45     "SimpleUnreachableBridge" linear |
46     "SmsSink" linear |
47     "ImplicitSmsSink" linear |
48     "LogSink" linear |
49     "ListBridge" linear |
50     "AppendToStringBridge" linear |
51     "ArrayBridge" linear |
52     "ArrayExampleBridge" linear |
53     "StringBufferBridge" linear |
54     "RandomIfElseBridge" 3Branches |
55     "DImeiSource" 2Branches |
56     "DEmptySource" 2Branches |
57     "DAliasingSanitizerBridge" 2Branches |
58     "DArraySanitizerBridge" 2Branches |
59     "DBluetoothDetectionBridge" 2Branches |
60     "DButtonCallbackBridge" 2Branches |
61     "DDatacontainerBridge" 2Branches |
62     "DIccInactiveActivity" 2Branches |
63     "DIccParcel" 2Branches |
64     "DListCloneBridge" 2Branches |
65     "DObfuscation1Bridge" 2Branches |
66     "DObfuscation2Bridge" 2Branches |
67     "DPublicApiPointBridge" 2Branches |
68     "DSimpleRecursionBridge" 2Branches |
```

```
69      "DSimpleSanitizationBridge" 2Branches |
70      "DSimpleUnreachableBridge" 2Branches |
71      "DSmsSink" 2Branches |
72      "DImplicitSmsSink" 2Branches |
73      "DLogSink" 2Branches |
74      "DListBridge" 2Branches |
75      "DAppendToStringBridge" 2Branches |
76      "DArrayBridge" 2Branches |
77      "DArrayExampleBridge" 2Branches |
78      "DStringBufferBridge" 2Branches
```

Listing 3.8: Improved grammar used in GenBenchDroid extension

By addressing all three limitations, we were able to generate applications that closely resemble real-world scenarios while maintaining a flexible design. To minimize the need for extensive changes to the tool, we kept design modifications to a minimum, thereby avoiding significant alterations to the tool's framework. To speed up the source-code generation, we used module processing as asynchronous operations. The extension applied resource-consuming operations to GenBenchDroid and template engine *handlebar.js*. Hence, for efficiency, we changed our template engine to *nunjucks.js*, which works similarly to *handlebar.js*. Although a comparison between both was not performed, the extension performed well. Using numeral identifiers, the existing ground-truth calculation was repurposed for the new TMC configuration. All models underwent adjustments for pattern settings `IN`, `OUT`, and adaptation to handle `sensitivedata`. A few modifications were made to the Gradle build tool configuration to align with application requirements. For example, the compileSdkVersion was set to 28, and the minSdkVersion was set to 19. Additionally, the applications were compiled using Java version 8. The contributed extensions can be found in the repository provided in Appendix A.1. Also, a list of added JSON module configurations for flexible design is present in Table A.1.

3.2 TABS: Taint Analysis Benchmark Suite

This section describes the development process of a benchmark suite designed to emulate real-world scenarios respective to complexity and the required number of sources and sinks in the applications. For a comprehensive empirical evaluation of taint analysis, a benchmark suite consisting of real-world like applications with varying source-code properties are necessary. This requirement brought us to identify two essential criteria. First, taint analysis requires applications featuring a specific quantity of sources and sinks. Second, the benchmark suite should include applications with diverse program structures, code complexity, and known vulnerabilities.

However, finding Android applications suitable for taint analysis and evaluation that meet these criteria posed a challenge. Designing applications with a specific or similar count of sources and sinks functions is particularly challenging within the restricted timeframe of this work. Hence, we used to GenBenchDroid, a tool designed to generate Android applications according to our needs while replicating real-world like application behavior. Improvements made and limitations addressed in Chapter 3.1 allowed us to develop a benchmark suite named Taint Analysis Benchmark Suite (TABS). TABS consists of 114 applications of diverse and intricate designs, including its taint-flow information (AQL-Answers) to serve as the ground truth for the generated applications. The generated suite assists in evaluating the efficiency of both forward and backward taint analysis. Under the assumption that the number of sources and sinks within an application could influence the analysis's efficiency, we considered three specific design choices for the application generated:

1. number of sources and sinks present in the application,

2. design and complexity of the applications,

3. arrangement of present sources and sinks in the application.

The aforementioned three criterias define the property of 114 applications included in the benchmark suite.

3.2.1 Number of sources and sinks

To analyze the behavior of the analysis based on the chosen direction, we developed applications with varying numbers of sources and sinks. This allowed us to observe behaviors specific to each backward and forward analysis.
To evaluate the impact of the present number of sources and sinks in the applications on the taint analysis, we used different combinations of sources and sinks. The ratios of sources to sinks considered are between 10:1 to 10:10 and 1:10 to 10:10. We created 19 applications through this approach, each with a unique combination of sources and sinks, as outlined in Table 3.1. The present number of sources and sinks helps define the number of taint flow in their respective applications. If there are a certain amount of sources present in the application, and the number of sinks is equal to or less than the present sources, then an independent taint flow exists, and the analysis should detect it. Each source leads to an individual sink. Considering the application with 200 sources and 200 sinks, there exist 200 independent taint flows in the application. The present number of independent taint flows is mentioned in the table, and there exists no other taint flow. In the rest of the scenarios where this condition does not match, only sources or sinks with their respective complexity exist.

Sources:Sinks Ratio	**Sources:Sinks in numbers**	**Taint flow count**
1:10	20:200	20
2:10	40:200	40
3:10	60:200	60
4:10	80:200	80
5:10	100:200	100
6:10	120:200	120
7:10	140:200	140
8:10	160:200	160
9:10	180:200	180
10:10	200:200	200
10:1	200:20	20
10:2	200:40	40
10:3	200:60	60
10:4	200:80	80
10:5	200:100	100
10:6	200:120	120
10:7	200:140	40
10:8	200:160	160
10:9	200:180	180

Table 3.1: Sources to sink ratio, and amount of taint flow of 19 applications

3.2.2 Design and complexity

The applications are specifically designed with a range of 20 to 200 sources and a range of 20 to 200 sinks, as depicted in Table 3.1. In addition to analyzing the number of sources and sinks, we aimed to evaluate the impact of their placement within the applications. The analysis tool starts analysis from sources to sinks in a forward setting, and from sinks to sources in a backward setting. We observed the influence of source and sink placement on the created edges in the ESG, in Figure 2.2. When a taint flow is short, the sink is close to the source; it can lead to faster analysis. This placement also affects the analysis efficiency in terms of memory usage and runtime performance. However, the overall efficiency of forward and backward analysis may differ due to the application design.
With the flexibility in the application design, we had control over the placement of sources and sinks. Utilizing this advantage, we incorporated three different design choices in our applications to further investigate the impact of the taint flow structure.

simple design

This design includes taint flows present in a single class. Hence, all sources and sinks are within the same class. By examining the taint analysis behavior in a single class containing all taint flow, we can gain insights into its efficiency and resource utilization for such designs.

simple_icc design

In contrast to design 1, design 2 involved placing all sources within the same class while directing all sinks to a separate individual class. By creating an application where all sinks are leaked in a new class, we can study the behavior of the taint analysis in this scenario. This is done by utilizing Android's ICC component.

simple_icc_callback design

Similar to design 2, this design also involved placing all sources within the same class while directing all sinks to a separate individual class. However, design 3 introduced additional complexity by utilizing callback functions, resulting in taints being leaked in a new class.
Considering a total of 19 applications with varying numbers of sources and sinks, as outlined in Table 3.1, applying these three different designs over them generated the possibility of total of 57 applications.

3.2.3 Arrangement

To increase the depth of our analysis, we paid special attention to the arrangement of sources and sinks within the codebase, on top of the intricacy of the design. As the sources, sinks, and taint flow operations in the application contribute to a more complex call graph for taint analysis, evaluating the efficiency of analysis relative to their arrangement can highlight scenarios where one-directional setting outperforms another.
To explain our selected arrangement of sources and sinks, we introduce two terms:

1. Series: In a series configuration, all elements of taint flow, sources, operations over them, and sinks are organized sequentially within the application. For instance, the arrangement might look like source1, source2,...source200, followed by sink1, sink2,...sink200.

2. Iteration: In an iterative configuration, each taint flow is represented in the application sequentially. The program flow starts with generating a source, a sequence of operations

is then executed on these sources, then each source is subsequently leaked via a sink. This cycle repeats with the generation of a new source. This could be depicted as: source1,..., sink1, source2,..., sink2, continuing to source200,..., sink200.

By examining the number of sources and sinks present in the application, their arrangement, and the design of the taint flow, we can investigate the efficiency of the taint analysis in both forward and backward settings. With 57 unique applications generated, each with varying numbers of sources, sinks, design elements, and complexities, we also incorporate both arrangements, culminating in a total of 114 (57+57) applications. Each of these applications possesses distinct source code properties that we can use to evaluate the efficiency of taint analysis.

3.2.4 Patterns

Considering the number of sources and sinks from Section 3.2.1 , design and complexity from Section 3.2.2, and arrangement of same from Section 3.2.3, we produced 19 applications, each with varying quantities of sources and sinks, and designs as outlined in Table 3.1. These were developed in two arrangement styles and three designs, culminating in 114 applications for the benchmark suite. Consequently, each of 19 applications can be classified into one of six unique patterns, which are defined in Figure 3.1, Figure 3.2, Figure 3.3, Figure 3.4, Figure 3.5, and Figure 3.6. The benchmark application suite comprises these 19 applications distributed across the six patterns, collectively making up 114 applications. The following sections discuss the design of all six patterns.

P1: series_simple

Figure 3.1 illustrates the application design accommodating 200 sources, 200 sinks, and 200 unique taint flows. All these sources are initialized in the scope of `onCreate()` method within the `MainActivity`. The operations, represented as OPS in the figure, are multiple tasks performed on the source to add complexity to each taint flow. These operations (OPS) are executed on all 200 derived sources, which are subsequently directed to the sink `sendTextMessage()`. This pattern emerged from the combination of simple design, in which all taint flow is limited to one class and series arrangement. Hence, to refer to this particular set of applications with pattern P1, we define it as series_simple.

P2: series_simple_icc

Pattern P1 is analogous to pattern P2 displayed in Figure 3.2. In this pattern, all 200 sources are initialized at the start of the `onCreate()` method, followed by operations (OPS) on each of these sources. However, rather than allowing the flow to leak within the same class, it is redirected to a new class designed specifically for individual taint flows, named NextActivity. The `sendTextMessage()` sink, responsible for the leakage of the flow, is located within this NextActivity class. The sinks are leaked in a new class named NextActivity which in the applications is established through Android's Inter-component communication(ICC) model. Hence, we refer to this set of applications as series_simple_icc.

P3: series_simple_icc_callback

The design illustrated in Figure 3.3 is similar to the one in Figure 3.2 includes two additional steps in the individual taint flow. Once the taint flow is propagated to the class `NextActivity`, rather than leaking the taint through the sink `sendTextMessage()`, a callback function is initiated. This, in turn, leads back to a call to the sink. Hence, the application of this pattern P3 can be referred to as series_simple_icc_callback.

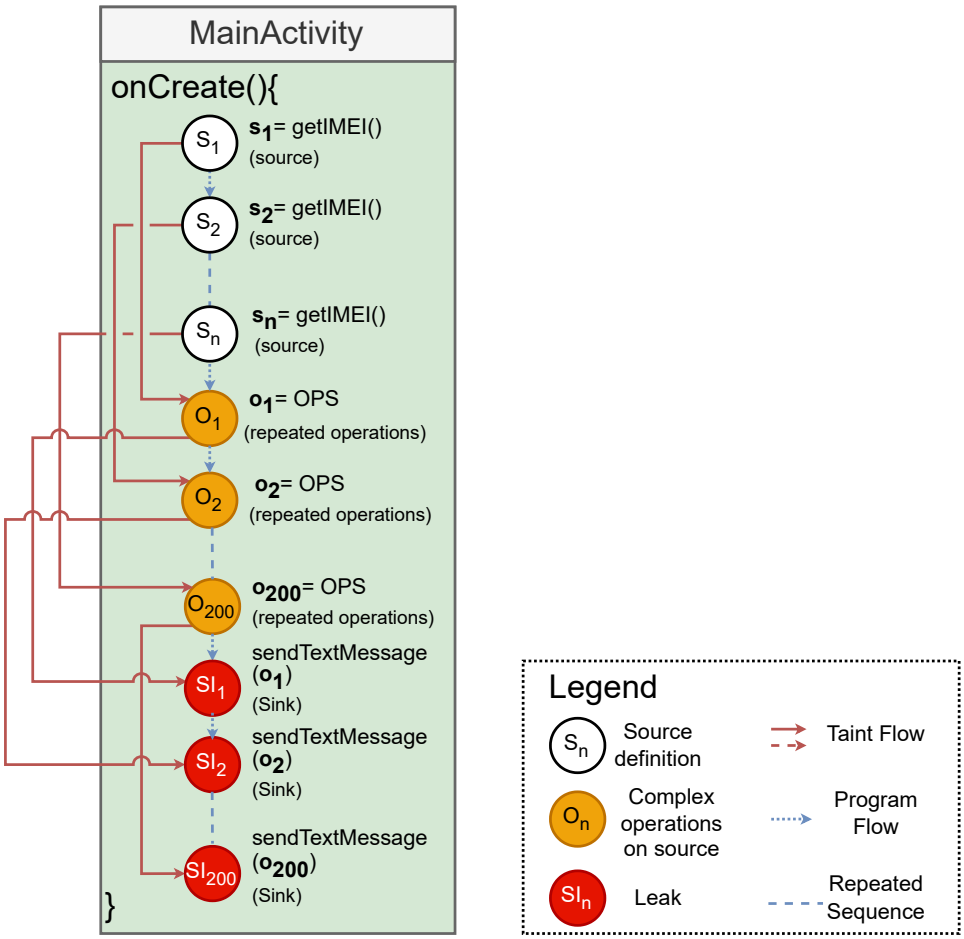

Figure 3.1: Implementation of design series_simple

P4: iteration_simple

The illustration in Figure 3.4, as opposed to a series arrangement, showcases the the iterative pattern of taint flow as outlined in 3.2.3. Initially, a single source is established, after which several operations are conducted. Subsequently, the derived flow is leaked to a sink, thereby concluding the taint flow. This procedure facilitates the continuous creation of 200 flows. All taint flows in this pattern persist within a single class, MainActivity. Similar to patterns P1, P2, and P3, this pattern P4 can be referred to as iteration_simple.

P5: iteration_simple_icc

The **iteration** pattern design depicted in Figure 3.5 is similar to the one presented in Figure 3.4. However, in this pattern P5, the sinks are specifically located within the individual class NextActivity. Similar to pattern P2, all sinks are propagated to NexActivity via ICC. To refer to this pattern, we use iteration_simple_icc.

P6: iteration_simple_icc_callback

Figure 3.6 illustrates the sixth pattern with an iterative structure, where each taint flow occurs serially in the MainActivity, and each sink leads to a distinct class called NextActivity. Similar to pattern P3, the callback function is used in this design to create a comparative complex

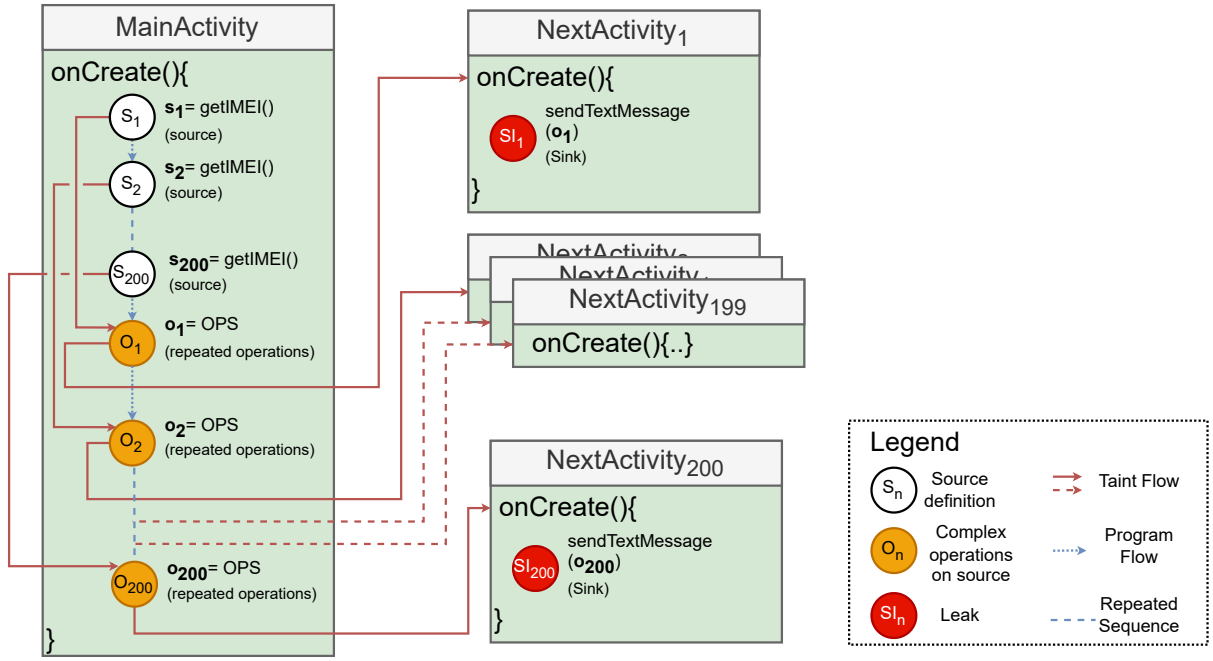

Figure 3.2: Implementation of design series_simple_icc

structure. Hence, we refer to this pattern P6 as iteration_simple_icc_callback.

The reference used for all six patterns are also used and distinguished in the benchmark suite TABS, made available with this work for which the git repository is provided in Appendix A.1. Table Appendix A.2 presents a complete list of modules used to generate TABS.
The complete list of applications contributed in the benchmark suite TAB is available in Table A.3 where code metrics **S** refers to number of sources, **SI** refers to number of sinks, **LOC** refers to Lines of code present, **Assign** refers to assignments, **FN** refers to funcitons, and **CLS** refers to classes present in the the respective applications. The presented code metrics do not include the *BuildConfig.java* file that holds information about the build and is automatically generated during the build process of an Android project.

3.3 SecuCheck Backward analysis

SecuCheck is a highly configurable taint analysis tool with two solvers, Boomerang SPDS [25] and FlowDroid. Initially, SecuCheck supported only forward analysis. Recent work done by Ozuni[13] on improving FluentTQL extends the tool to implement backward analysis with Boomerang. However, there is no implementation of backward analysis in SecuCheck with FlowDroid. However, the backward implementation of Boomerang is yet to be formally tested. The SecuCheck backward implementation must be precise as its forward taint analysis implementation before one could use SecuCheck for taint analysis evaluation. Hence, this chapter first discusses necessary changes to secucheck and related tests.
SecuCheck can be utilized and configured via an Integrated Development Environment (IDE) and is also available as a Command-Line Interface (CLI) tool. In CLI tool, configurations are supplied through YML configuration. In this section, we will solely focus on implementations related to the CLI tool. As presented in Listing 2.4, SecuCheck backward extension by Ozuni [13] applies backward analysis only when more sources exist than sinks. The backward analysis feature, extended by Ozuni [13], was available in an older version of SecuCheck 1.1.0. However,

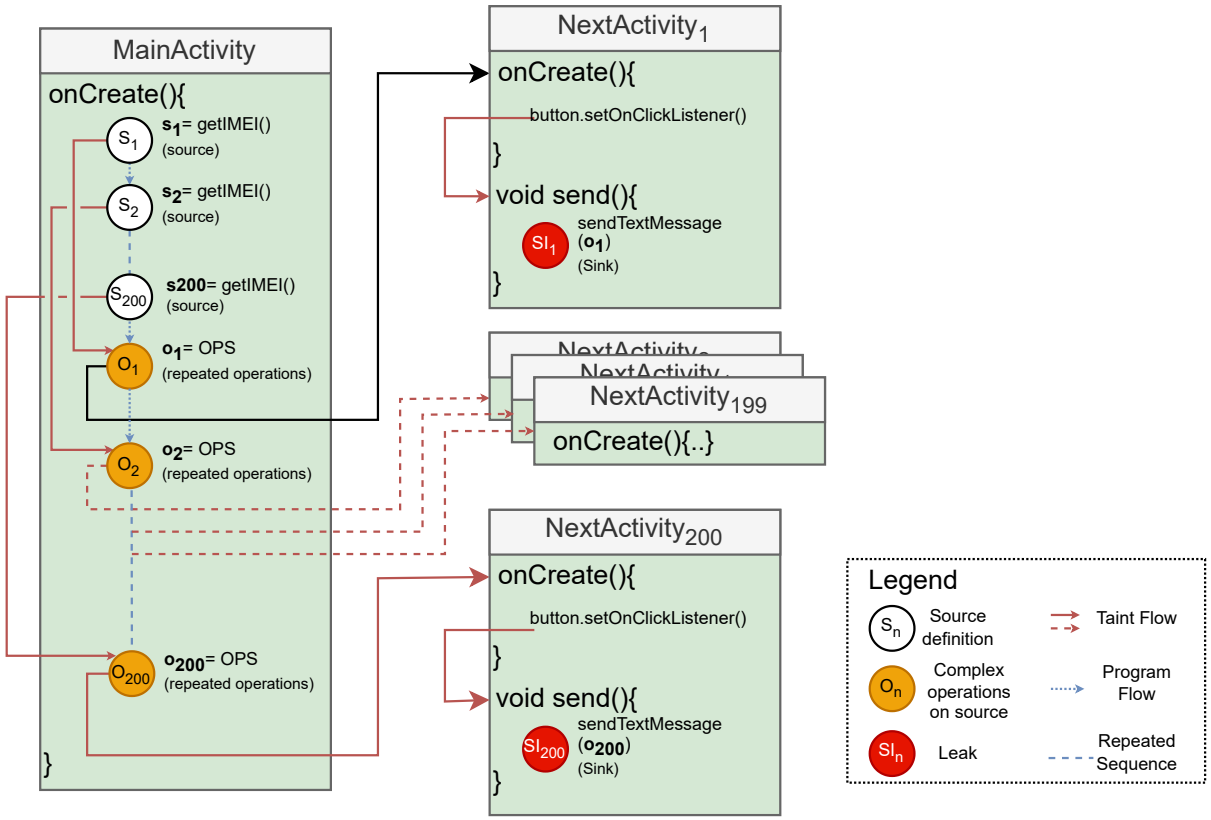

Figure 3.3: Implementation of design series_simple_icc_callback

the latest version, 1.2.0, conflicts with this backward implementation. As a result, we decided to continue our work on the older version.
To test backward implementation, we first required settings to choose the analysis directions manually. To independently change the direction, we employed the logic to change the analysis direction via YML settings. The Listing 3.9 provides a YAML configuration that is employed for the analysis of an application *application1*. Statement 6 represents the introduced feature, which allows the manual direction selection, replacing earlier used default logic. Setting `analysisDirection` can be set to two options `forward`, for forward analysis, and `backward` for backward analysis with Boomerang. To achieve it, respective changes were necessary for both *secucheck* tool and *secucheck-core*.

```
1 classPath: "C:\Users\user\projects\SecuCheck\application1\target\classes"
2 entryPoints:
3 specPath: "C:\Users\user\projects\secucheck-catalog\tabs-specification\target"
4 selectedSpecs:
5 solver: "Boomerang3"
6 analysisDirection: "backward"
7 isPostProcessResult: false
```

Listing 3.9: Example YML configuration for analysis with SecuCheck

In addition to the direction of analysis, Listing 3.9 also depicts other settings.

1. `classPath`: This setting refers to the directory path of the compiled class file of the application.

2. `specPath`: This option refers to the fluentTQL specification used to identify vulnerabilities.

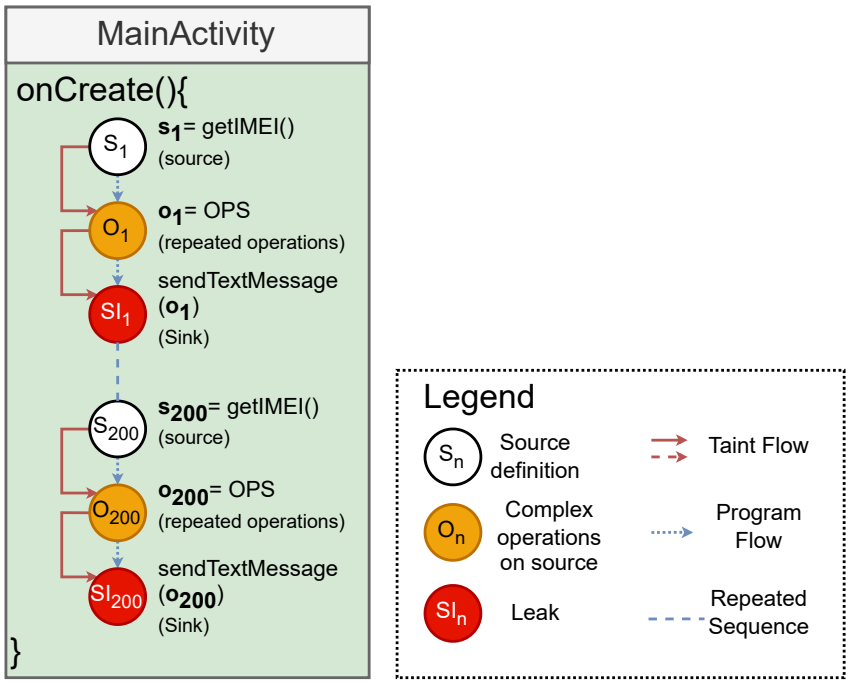

Figure 3.4: Implementation of design iteration_simple

3. `entryPoints` refers to the chosen entry point for analysis from the available classpaths.

4. `selectedSpec`: This refers to the available fluentTQL specifications. All specifications are chosen if no individual specification is specified.

5. `solver`: This option can be set to either 'Boomerang3' or 'FlowDroid'. In this implementation, we refer to the solver as Boomerang3.

SecuCheck was also extended to detect Hard-Coded Credentials (HCC), and Null Pointer Detection (NPD). Theoretically, in both scenarios, a backward analysis is always efficient for detecting such vulnerabilities from the sink where such values (credentials or null) are used back to the source. Hence, in both types of vulnerability detection, analysis is set to a backward analysis.

3.3.1 demo-project analysis

secucheck-catalog is an extension of secuCheck's work that consists of numerous applications and micro-benchmark tests for vulnerabilities, as well as corresponding FluentTQL configurations to detect these vulnerabilities.
With extended SecuCheck, analysis over *demo-project* was performed in both forward and backward configurations. *demo-project* is a to-do list application in *secucheck-catelog* repository that contains numerous vulnerabilities. The FluentTQL specifications for *demo-project* can be found in *demo-project-specification*.
The backward analysis could only find 33 leaks out of a total of 42, while the forward analysis detected all taints present in the application as presented in Table A.4 where AC refers to present actual taint flow count, while F refers to taint flow reported by forward and B refers to taint flow reported by backward analysis. A detailed report can be found in the repository provided in Appendix A.1. This difference in reported taints indicates that few edge cases exist where backward analysis fails to report vulnerabilities. Hence, further improvement on coverage is necessary.

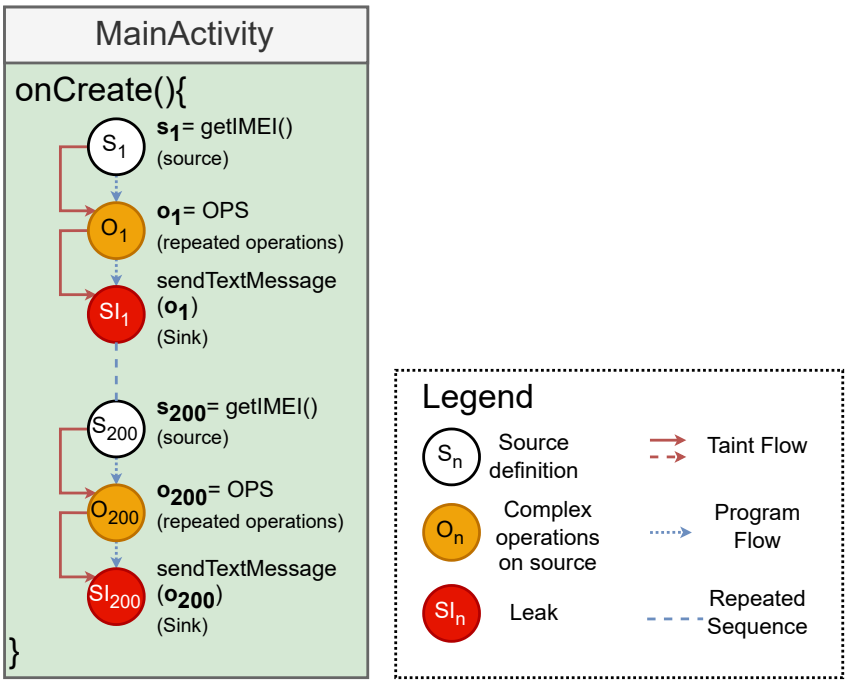

Figure 3.5: Implementation of design iteration_simple_icc

3.3.2 Android application analysis

Like the demo project, taint analysis was performed on Android applications to evaluate SecuCheck's taint analysis capability over Android applications. Unlike FlowDroid, SecuCheck has no Android modeling feature. Hence, it can only evaluate Java applications or available Java class files. Since extended GenBenchDroid allowed us to keep Java class files of the built application, we utilized the same class files for the analysis. For evaluation with SecuCheck, we required a fluentTQL specification; hence, a new specification *tabs-specification* in *secucheck-catelog* was created. To initially test the analysis, three simple applications were created.

1. simple_arraybridge_app: an application with a single source containing array operation on the source and then leaking the result to a sink.

2. simple_list_clone_app: an application with a single source containing a list clone operation with source and then leaking the tainted value to the sink.

3. simple_stringbuild_app: an application with a single source, array operation, and string builder. The result then leaks to a sink.

While forward and backward analysis of application simple_arraybridge_app reported the taint flow, analysis of the other two applications, simple_list_clone_app, and simple_stringbuild_app, did not report any taint flow in any direction.
It is unclear why SecuCheck analysis only reported taint flow with Array operation but did not report taint flows with list clone or StringBuilder operations even though both are core Java features.
Hence, SecuCheck requires a further extension and tests for Android application evaluation to fulfill its shortcomings. Since SecuCheck cannot report taint with common operations, we refrained from utilizing it for our evaluation.

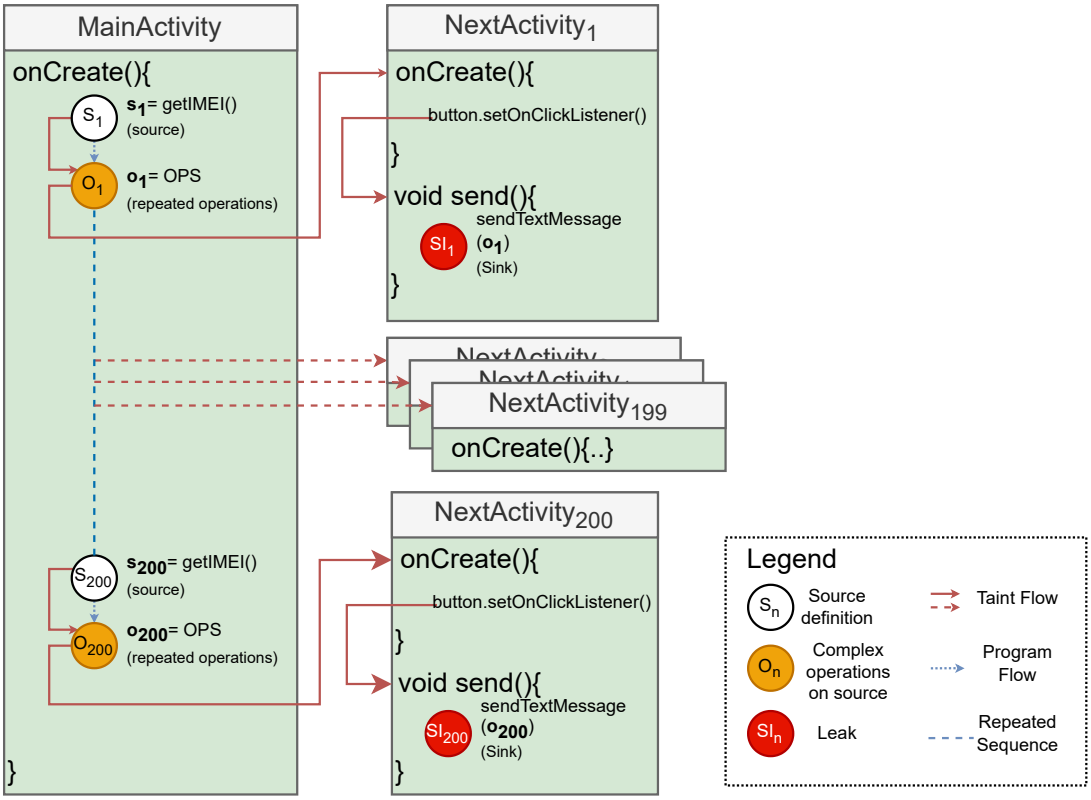

Figure 3.6: Implementation of design iteration_simple_icc_callback

4 Evaluation

4.1 Empirical Evaluation

The previous Chapter discussed the generation of a benchmark suite TABS. This chapter discusses the findings and associated metrics collected from evaluation of the TABS benchmark suite. Since SecuCheck could not find taints in the Android applications of the generated benchmark TABS, our evaluation focuses on the results obtained using FlowDroid. To encapsulate real-world application behavior, we also performed analysis over applications from TaintBench [10]. TaintBench benchmark suite contains 39 fully documented real-world applications with complete ground truth. However, we could not include the evaluation of TaintBench applications in our analysis. This is because only 7 out of 39 applications could be analyzed using FlowDroid, both in the backward and forward settings. This sample size was insufficient for our evaluation. Additionally, 32 applications failed to initiate the evaluation in the backward setting. We reported the failures to the maintainers of FlowDroid during our evaluation, and the issue[1] was later fixed. Hence the TaintBench evaluation could be a potential future work. Our work aims to better understand the behavior of taint analysis in different directions for efficiency and to explore the possibility of predicting the most efficient direction for taint analysis. With the gathered results from the analysis of our generated benchmark suite, we discuss the effects of the taken direction on the analysis efficiency and address the following two research questions:

RQ1 How do the number of sources and sinks affect the efficiency of forward and backward taint analysis, and how do the two types of taint analysis compare in terms of their overall efficiency?

RQ2 How can we predict efficient taint analysis direction for a given application?

Section 4.1.1 presents the used configuration for the FlowDroid and environment settings, while Sections 4.1.2 and 4.1.3 will address the research questions.

4.1.1 Configuration

We conducted our analysis using a dedicated machine equipped with an Intel Xeon E5-2695 2.30GHz processor and 128GB of RAM. In our initial trials, which are not part of the evaluation, we found that the Java Virtual Machine (JVM) required limited memory. As a result, we set a

[1] `https://github.com/secure-software-engineering/FlowDroid/pull/616`

40GB RAM limit for all analyses. However, the behavior differed if the minimum heap size was initially set. Consequently, we refrained from setting a minimum heap size, allowing the JVM to manage these settings while the maximum is 40GB.
We recognize that background services could influence analysis performance. Hence, each analysis run was performed ten times to ensure trustworthy and precise results, with a ten-second interval between each run. FlowDroid was configured to its default configuration with timeouts for data flow analysis. Analysis results were collected from the analysis logs after all runs were complete. We used FlowDroid built from the latest commit available at the time as mentioned in Appendix A.1.
We employed the default sources and sinks list provided with FlowDroid for our evaluation. The benchmark suite analysis, covering 114 applications in both forward and backward modes, was completed within an average of 93 minutes. Remarkably, no timeouts or failures were registered during the entire procedure.
Given the characteristics of sources and sinks, the taint flow design and program flow while developing the benchmark suite TABS, taint flows in all applications were identified. A recent amendment to FlowDroid contributed to achieving 100 percent accuracy in reporting taint flow in both forward, and backward setting. Details of the default FlowDroid settings and resource settings utilized in the evaluation are presented in Table 4.1.

Option	**Value**
Result timeout	10 minutes
data-flow timeout	10 minutes
Taint wrapper	EasyTaintWrapper
implicit flow	disabled

Table 4.1: FlowDroid settings used in the evaluation

In our analysis, a timeout of 10 minutes was set for both the data flow and result evaluations. For TABS applications, the data flow analysis runtime spans a few seconds, but the difference in time taken by the analysis is noticeable. To accurately depict the impact of the number of sources and sinks, we plotted the precise quantity of sources and sinks of each application across all evaluation graphs. Unlike the results from FlowDroid, where the terminology of sources and sinks changes according to the configured direction settings, our study maintains a consistent perspective. This study considers sources and sinks from a forward analysis perspective. A source is always referred to as an entity that introduces a taint, and a sink is always referred to as an entity that leaks the taint in both forward and backward analysis.

4.1.2 RQ1: How do the number of sources and sinks affect the efficiency of forward and backward taint analysis, and how do the two types of taint analysis compare in terms of their overall efficiency

We already introduced the Taint Analysis Benchmark Suite (TABS) created to evaluate the efficiency of the taint analysis that contains applications with a diverse number of sources and sinks, including six distinct designs. To address the complete RQ1, we ran taint analysis in two different direction settings, forward and backward, with FlowDroid. Since all analyses revealed 100% accuracy in taint flow detection, our focus shifts to the data flow time (time taken by the IFDS solver for taint propagation) and the maximum memory utilization by the analysis to compare the efficiency of taint analysis. It is essential to keep in mind that the Taint Analysis Benchmark Suite (TABS) consists of applications with six distinct patterns, as detailed in Section 3.2. Hence, evaluation of each pattern is necessary. P1, P2, and P3 are patterns that

follow a similar arrangement of sources, sinks, and taint flow operations within the applications. In Pattern P1, all taint flows are encapsulated within a single class where all sources are created, operated upon, and leaked. Pattern P2, conversely, starts taint flows in a single class, similar to P1, but each taint flow leaks in its distinct class. Pattern P3 closely mirrors P2, with taint flows beginning in a single class and leaking in individual classes. However, P3 differs in terms of the complexity of these taint flows. They involve more intricate operations, such as callbacks and function calls, increasing the overall complexity compared to P2. Similarly, P4, P5, and P6 are patterns that contain taint flows with shorter lifespans. These taint flows are created, operated upon, and then leaked, unlike those in P1, P2, and P3. While P5 follows this pattern, P6 adds an extra layer of complexity. Like P5, taint flows in P6 are short-lived, but they involve more complicated operations like callbacks and function calls, and distinction of P5 and P6 is much like the distinction between P2 and P3.

Average Evaluation

Table 4.2 presents the average results of all 114 TABS applications, including total edge propagation in the analysis. Notably, the backward analysis was approximately 50% more efficient than the forward analysis. Forward analysis, on average, took 7.24 seconds, while the average time taken for taint propagation in the backward analysis was 3.6 seconds.
Surprisingly, the obtained average data flow time with 50% efficiency from our evaluation corresponds to the analysis results obtained by Lange [8] in their evaluation of 200 real-world applications without timeouts. Also, the average memory consumption is found to be comparable. For TABS, forward analysis consumed about 1189 MB, while backward analysis utilized 1025 MB on average. While their backward analysis required few edge propagations and was in the 85th percentile, our evaluation reveals that backward analysis required much less propagating in the 35th percentile than forward analysis. We suspect this behavior is because of right-to-left flow in taint flows, since all applications in TABS contain such flow and heavily utilize assignment operations. The observed number of assignment operations present in individual applications is detailed in Table A.3.

Metric	**Forward Analysis**	**Backward Analysis**
Data flow time	7.24s	3.6s
Maximum memory consumption	1189 MB	1025MB
Total edge propagations	1897973	680002

Table 4.2: Average results of all 114 TABS applications

To answer the first part of the research question, we evaluate the applications of each design individually. First, we discuss the effects of the analysis direction on the data flow time. Afterward, we delve into the memory consumption by for each design and summarize the observed behavior at the end.

Data Flow Time

In Figure 4.1, we compare the time taken by forward and backward analyses to examine applications from pattern P1. On the x-axis, ratios of sources and sinks present in the application is depicted. The numerical value of this ratio consistently increases from left to right, maintaining the increasing order. We observed that in the forward analysis, the analysis time increased as the number of sources rose from 20 to 160, with the number of sinks remaining constant. Although, the analysis time remained unchanged when the number of sinks decreased.

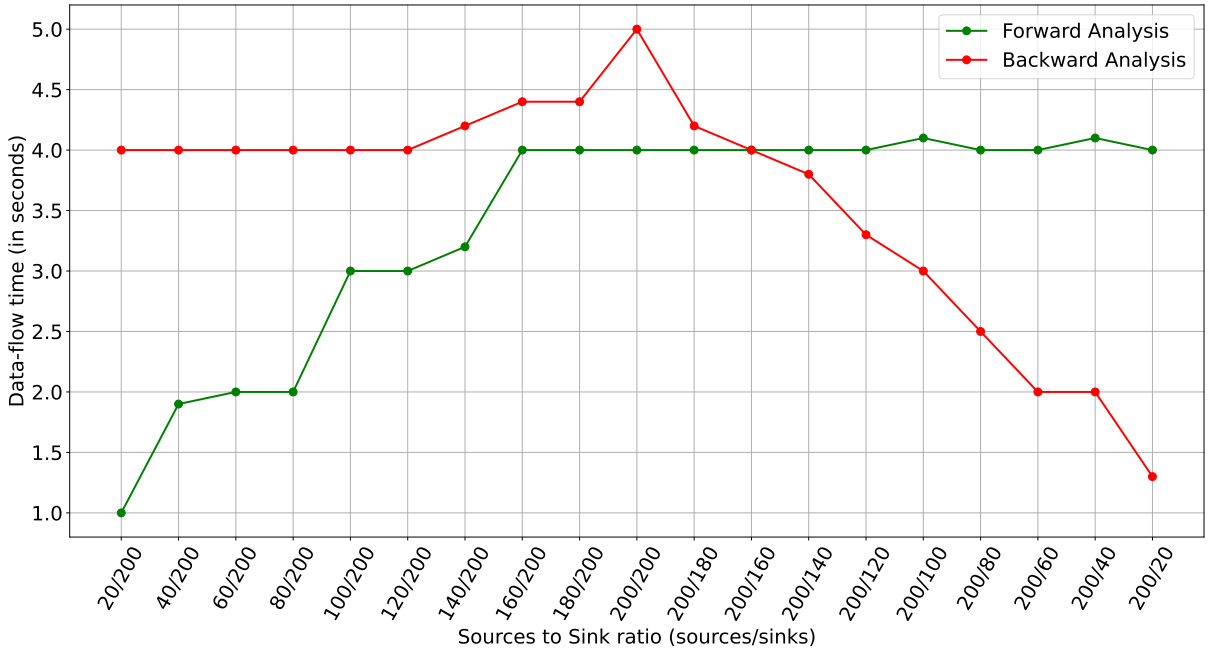

Figure 4.1: Time evaluation of 19 applications with pattern P1, series_simple

In contrast, to forward analysis, backward analysis reduces the analysis time when the number of sources remains constant, and the number of sinks decreases. The behavior depicted in Figure 4.1 confirms our assumption regarding the effects of present sources and sinks. Although, the behavior can also be attributed to the application's design. Overall, the analysis time is balanced, with the backward analysis taking a maximum of 5 seconds and the forward analysis taking 4 seconds, while the source count is 200 for both forward and backward analysis.

Figure 4.2, on the other hand, depicts a scenario in which backward analysis generally outperforms forward analysis, except in four cases where the number of sources is relatively small compared to the number of sinks. The time evaluation illustrates the duration of the analyses conducted on the application with the pattern P2. We observed that the maximum time required by both analyses nearly doubles, with backward analysis taking approximately 6 seconds and forward analysis reaching a maximum of 10 seconds. The difference becomes apparent when comparing the applications of pattern P2 to that of P1. In P1, all sinks are located within a single class, whereas P2 has individual classes for each sink, which affects the data flow time. Based on this observation, we conclude that sinks' placement impacts the analysis time.

Figure 4.3 depicts the time evaluation for the applications with pattern P3, which is similar to the design of P2. Comparing the evaluations of both designs, we notice that the time taken by the backward analysis remained similar, even with the introduction of callback complexity in P3. However, we observed an average time increase of approximately 78 percent in the forward analysis. This behavior hints toward reduced efficiency of forward analysis due to introduced complexity.

Figure 4.4 illustrates the evaluation of the time taken by the application using the pattern P4. Interestingly, both forward and backward analyses take approximately 1 second on average, demonstrating equality in their execution times. When comparing the same applications but with different source-sink arrangements in pattern P1, we observe that the placement of sources and sinks does indeed impact the analysis time. As presented in Figure 3.4, 3.5, and 3.6, the lifespan of taint flow 200 is shorter than that of taint flow 1 in the applications of iteration pattern. Additionally, we suspect that the code optimization performed by the FlowDroid over

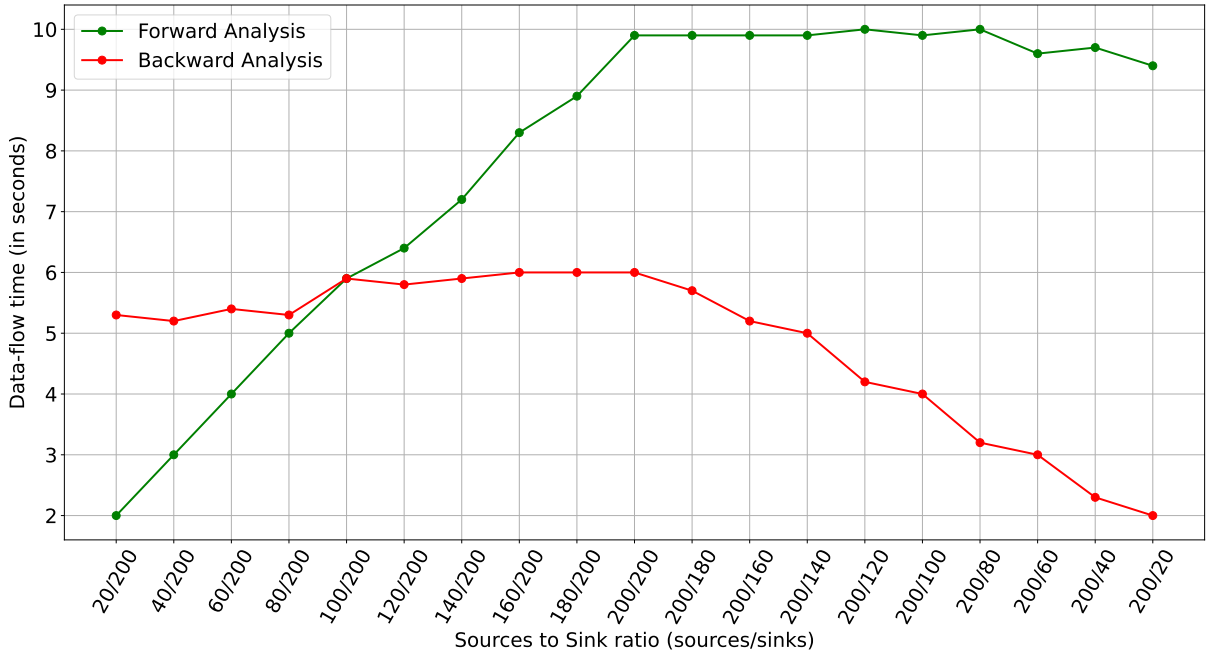

Figure 4.2: Time evaluation of 19 applications with pattern P2, series_simple_icc

repeated taint flow may have impacted the analysis.
Applications with the pattern P5 exhibit comparable results to P3, with a a slight decrease in their average time, as shown in Figure 4.5. Additionally, we notice a similar performance pattern between forward and backward analysis.

In summary, we have observed that backward analysis either outperforms or is comparable to forward analysis in terms of time when there exists complex taint flow with sinks placed at different classes. Although, we also acknowledge that the forward analysis will outperform the backward analysis in terms of data flow time when there are many sinks compared to the sources present in the applications.

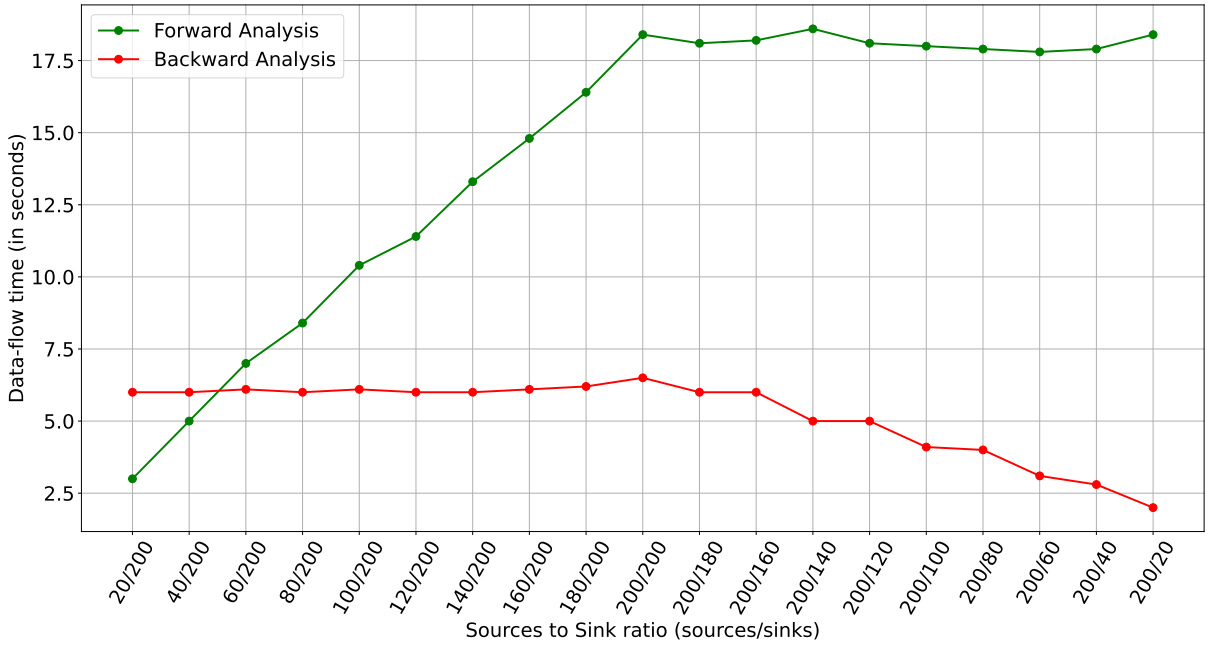

Figure 4.3: Time evaluation of 19 applications with pattern P3, series_simple_icc_callback

Maximum Memory Consumption

Unlike data flow time evaluations, we encountered a few challenges in memory evaluations due to the nature of our benchmark suite and JVM options. Our benchmark suite consists of varying sources and sinks, resulting in a certain level of taint flow with designed complexity. During the memory evaluation, we found that the minimum heap size we selected significantly impacted memory consumption during the analysis. The TABS application used approximately 4GB of memory for the analysis. Therefore, for the evaluation in this study, we did not specify a minimum heap size. Instead, we relied on the JVM (Java Virtual Machine) to automatically set it. The min default heap size set by JVM varies respectively to the Java version and the system configuration. In our evaluation, we observed the initial minimum heap size set to 2.1GB by JVM.

Figure 4.7 illustrates the peak memory usage for both forward and backward analysis of applications from the pattern P1. On average, forward analysis consumes substantially less memory than backward analysis. However, the forward analysis shows no apparent trend of increasing memory usage relative to sources and sinks. In contrast, the backward analysis tends to consume a significant amount of memory when there are many sinks and fewer sources while consuming less memory when there are fewer sinks.

In Figure 4.9 and Figure 4.8, the backward analysis presents increasing memory consumption when the number of sinks is increasing, though, the amount of sources remains the same. While on the contrary, when the number of sinks is a maximum of 200 and sources vary, memory consumption tends to decrease. This behavior can be attributed to the effect of few sources being present.

The memory consumption of the application using the pattern P3 is illustrated in Figure 4.9. The pattern of memory consumption in the backward analysis appears to be similar to that displayed in Figure 4.8. In contrast, the forward analysis consumes a greater amount of memory as the the number of sources increases. This escalation in memory usage is attributed to the additional complexity introduced by the callback function.

Similar to pattern P1, the application of design P4 present a non-linear graph, although similar-

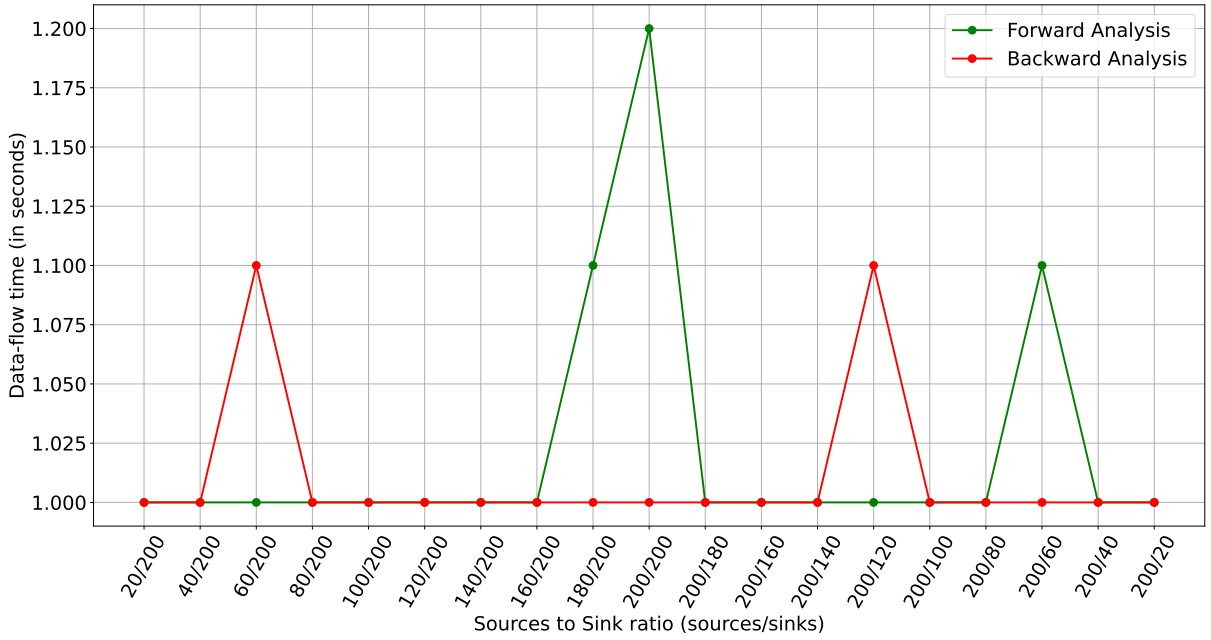

Figure 4.4: Time evaluation of 19 applications with pattern P4, iteration_simple

ities among forward and backward analyses are clear. It is important to note that the maximum and minimum memory consumption depicted is much lower than other patterns. Similarly, for pattern P4, data flow time averages 1 second for all runs, as depicted in Figure 4.4.
The similarity between the backward analysis in Figure 4.11 and Figure 4.12 is observable. However, the memory consumption for the forward analysis shows an increase in the pattern P6. This increase is due to the added complexity and confirms that forward analysis is less efficient in terms of memory consumption as well.

Pattern	Avg. time(s)		Avg. memory (MB)		Avg. total edges	
	F	B	F	B	F	B
P1	64.3	68.1	13492	19421	15137226	13931060
P2	148.9	91.4	18012.8	18544.9	38916090	19851230
P3	271.1	99	41005.1	22270	78371082	20013519
P4	19.4	19.2	5215.8	5534.1	425320	165348
P5	98.5	64	20760.6	24190.3	22039939	11698429
P6	222.7	69.6	37072.4	26896.7	61479292	11860697

Table 4.3: Average metrics of application of all patterns

Table 4.3 presents average data flow time, maximum memory consumption, and total edge propagations of 19 applications of all patterns individually respective to forward analysis indicated by `F` and backward analysis indicated by `B`. We can observe a correlation between the average time of the data flow analysis and its memory consumption, particularly in the pattern of complex flows in P2, P3, P5, and P6.
However, this correlation diminishes when all taint flows are confined to a single class, most notably in the pattern P1 and P4. Additionally, we have noticed a small difference in average memory consumption where backward analysis is notably efficient. The presented metrics related to runtime and memory consumption have addressed the first part of RQ1, *How do the number*

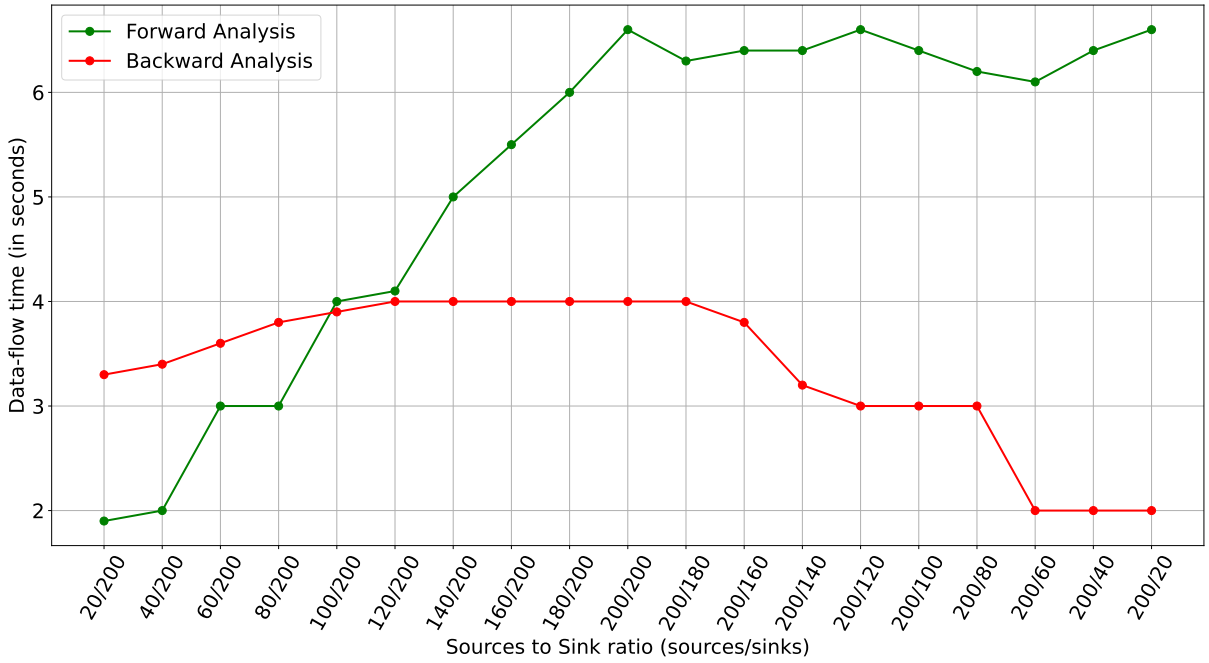

Figure 4.5: Time evaluation of 19 applications with pattern P5, iteration_simple_icc

of sources and sinks affect the efficiency of forward and backward taint analysis?
We discuss the effects of present sources and sinks in forward and backward analysis. To answer the second part of the RQ1, *how do the two types of taint analysis compare in terms of their overall efficiency?* We conclude with the observed results discussed in this section. After evaluating the Taint Analysis Benchmark Suite (TABS), which comprises a varying number of sources and sinks with different arrangements and complexities, we conclude the following:

1. Backward analysis proved to be efficient in terms of time, except in cases where the number of sources is significantly smaller than the number of sinks. The exception can be noticed in applications with the source-to-sink ratios 20:200, 40:200, and 60:200 in Figure 4.2, 20:200, 40:200 in Figure 4.3, 20:200, 40:200, 60:200, 80:200 in 4.5, and 20:200 in Figure 4.6.

2. In terms of memory consumption, backward analysis typically utilizes a relatively smaller yet comparable amount of memory for the analysis. However, backward analysis is more efficient in complex taint flow analysis. 18 out of 19 backward analysis of pattern P3 are efficient than its forward counterpart in Figure 4.9, and 15 out of 19 analysis of pattern P6 applications in Figure 4.12.

4.1.3 RQ2: How can we predict efficient taint analysis direction for a given application?

To answer RQ2, we considered various parameters: the number of sources, sinks, lines of code, classes, and functions that were already known or could be extracted from the source code at the pre-analysis stage. The TABS benchmark suite was carefully crafted using GenBenchDroid. Consequently, we had immediate access to a comprehensive ground truth containing taint flow information and available source code used to build individual applications enabling us to collect metrics. Table A.3 displays a detailed list of applications with metrics such as the number of lines of code, classes, functions, and assignments present in each application. We conducted analyses

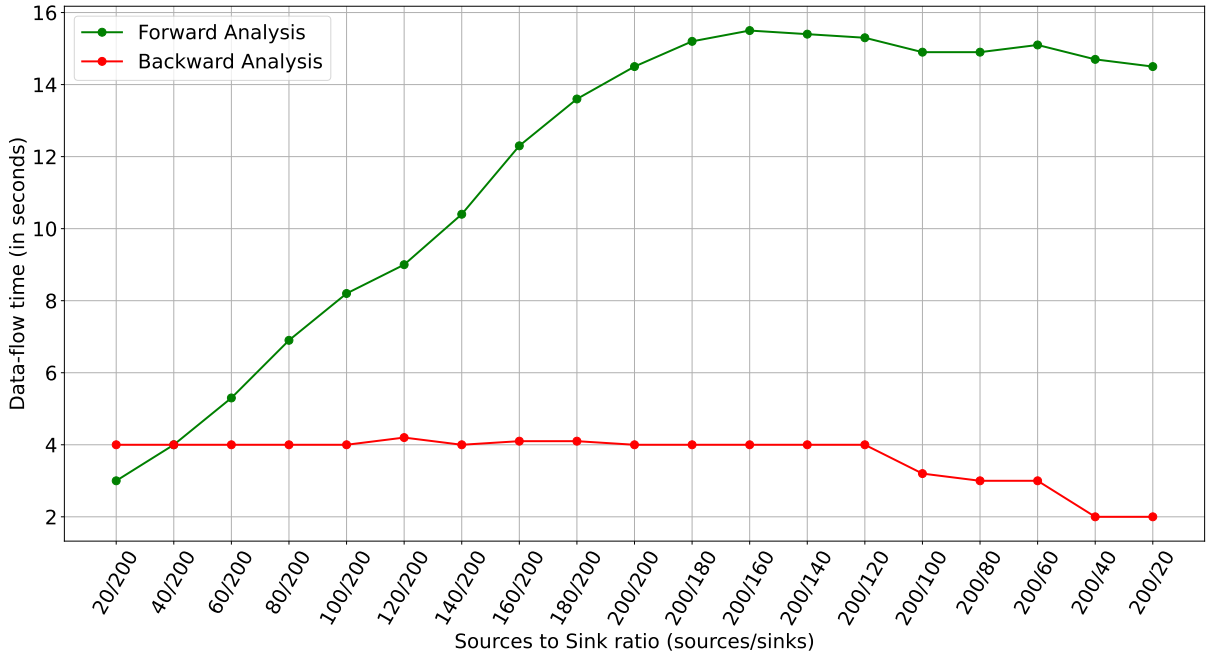

Figure 4.6: Time evaluation of 19 applications with pattern P6, iteration_simple_icc_callback

on all applications exhibiting different properties to address RQ1 in Chapter 4.1. The metrics collected from these analyses and the application's meta-information formed the foundation for our evaluation aimed at answering RQ2.

Figure 4.13, 4.14, and 4.15 present total lines of code, the total number of functions, and the total number of assignment operations present in 19 applications of each of six different patterns P1 to P6. All 19 applications are depicted on the x-axis as their sources-to-sink ratio, similar to memory and runtime evaluations. It should be noted that the applications of pattern P1 contain approximately similar lines of code, total classes, and total functions compared to pattern P4. Likewise, P2 contains code metrics similar to P5 and P3 similar to P6. The difference in the number of lines of code between applications following the same design is minimal, while number of functions in pattern P1 and P4 is 201, P2 and P5 contain 401 , P3 and P6 contain 801.

It is worth noting that the time taken for the analysis varies significantly among 19 applications of a pattern even when the number of functions is constant. The Figures 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6 present varying data flow times for 19 applications of different patterns.

Similarly, the number of classes for all applications of patterns P3, P4, P5, and P6 remain constant at 201 though the time is taken and memory utilization vary significantly, as presented in Figures 4.7, 4.8, 4.9, 4.10, 4.11, and 4.12.

If we compare the analysis time of different code patterns with higher numbers of lines of code due to additional taint flow, we cannot solely attribute the difference in memory consumption and time to only lines of code. The variations in code metrics between applications are minimal varying from 20 to 40 lines of code between applications. Hence, the number of lines of code alone does not significantly impact resource consumption during analysis. The same reasoning applies to the number of classes and functions since they remain constant within each pattern. However, sources, sinks, and taint flow can be contributing factors. Code metrics play a role in the application by adding taint flows but do not directly correlate with resource consumption. Therefore, we can conclude that metrics such as lines of code, number of functions, and classes do contribute to resource consumption of analysis, but only when involved in taint flows. Hence, they cannot be directly correlated with the efficiency of the analysis process.

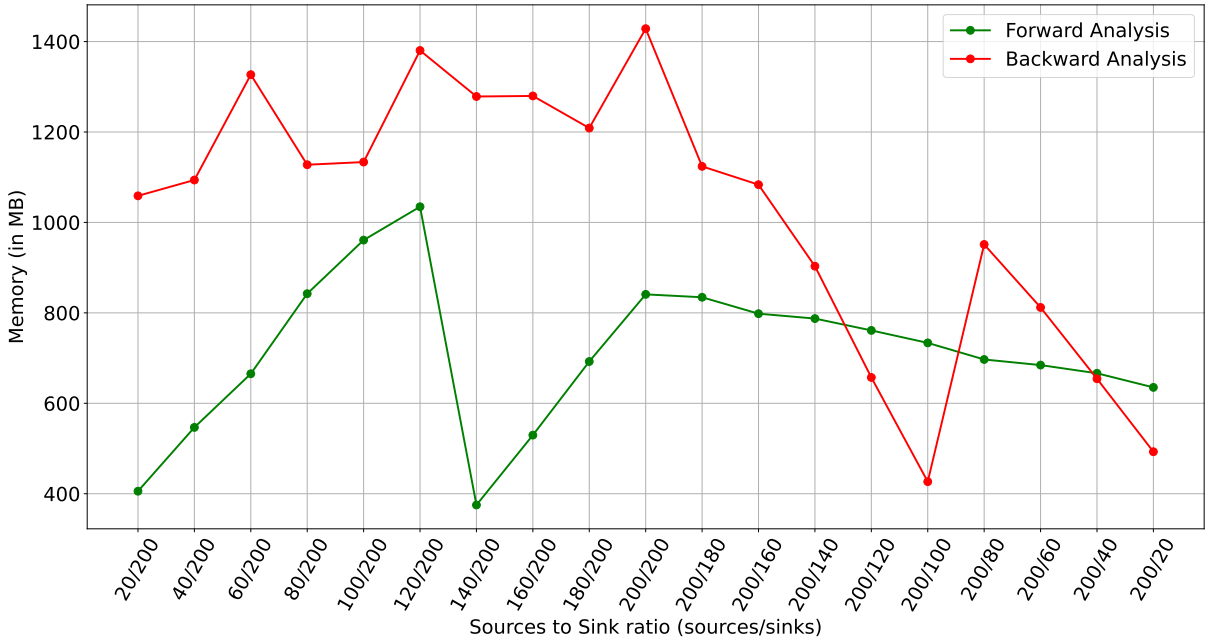

Figure 4.7: Memory evaluation of 19 applications with pattern P1, series_simple

Therefore, our initial consideration focuses primarily on taint flow and its available properties before the analysis. Among them is the mostly discussed amount of sources and sinks in the application. As presented in our evaluation in Chapter 4.1, the present amount of sources and sinks do impact the analysis. Additionally, backward analysis is efficient if the taint flow includes many classes in its propagation.
Notably, forward analysis is efficient in scenarios where the number of sources is substantially smaller than the number of sinks. Backward analysis always outperforms forward analysis in cases where the number of sinks is lower than the present number of sources given that there exist taint flows among multiple classes.
Additionally, our evaluation shows that backward analysis, on average, performs as efficiently as forward analysis in terms of runtime, even when the sources-to-sink ratio is in the range of 1:3 to 1:2. Applications present in TABS heavily utilize assignment operations to propagate and operate on the taint flow, resulting smaller number of edges in the backward analysis. As Table 4.2 presents, the difference in total edge propagation, backward analysis on average, utilizes around 1/3 of the forward analysis. The Figure A.1, A.2, A.3, A.4, A.5, and A.6 present comparison of total edge propagation of forward and backward analysis among all six TABS application patterns. It can be observed that in most analyses, backward analysis requires less edge propagation compared to forward analysis. Notably, in many cases, less than 50% compared to forward analysis. Since the TABS application heavily utilizes assignment operations, a higher number of assignment operations in taint flows might make backward analysis more advantageous due to fewer edge propagations. As analysis graph edges consume resources, the analysis's efficiency can be linked to the current number of assignment statements. However, similar to code metrics, this property does not directly correlate with efficiency prediction. Many applications might contain assignment-intensive operations that are not analyzed due to code unreachability or not being present in taint flow propagation.
Hence, to answer RQ2, we conclude the following:

1. The present number of sources and sinks can help predict efficient taint analysis, whereas a small number of sources compared to sinks favors forward analysis.

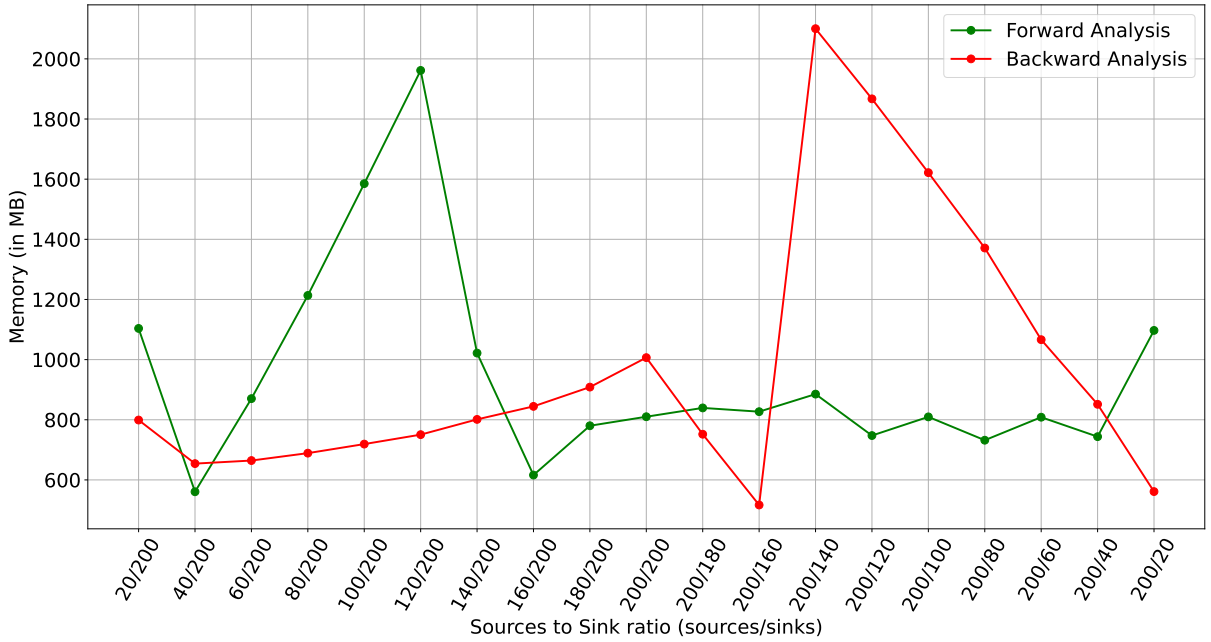

Figure 4.8: Memory evaluation of 19 applications with pattern P2, series_simple_icc

2. Code metrics like lines of code, number of classes, and functions can hint at the complexity of an application though it is not a reliable metric. Hence, the number of sources and sinks present in the application can be the primary indicator of efficient analysis.

4.1.4 Threats to validity

The application benchmark suite, TABS, created for empirical evaluation in this work, encompasses various aspects of application properties, such as a varying number of sources, sinks, taint flow, and underlying complexity, but not all cases are covered. The applications designed and included in TABS are well thought out to test specific behaviors. Consequently, the behaviors observed and results obtained are limited to the applications present in TABS. As we were unable to analyze the application using TaintBench, we could not validate the results against real-world applications. Furthermore, the analysis is performed using a single tool, FlowDroid. The efficiency of taint analysis can also be attributed to the tool performing it and tool's design. Therefore, empirical evaluation with a single tool presents a threat to the validity of the evaluation. A similar evaluation with multiple tools could overcome such a shortcoming and validate the obtained results.

4.1.5 Summary

In evaluation of taint analysis, we performed taint analysis with FlowDroid on an artificially generated benchmark suite named TABS, which contains many applications. We first observed the impact of the present number of sources and sinks in applications on taint analysis with respect to efficiency in terms of resource consumption, memory, and time. We found that the analysis time increases in relation to the present taint flows in the applications. Additionally, in forward analysis, time consumption is the lowest in case of lower number of sources while there exist many sinks, while in backward analysis, it is in case of smaller number of sinks and many sources exist. Backward analysis was found to be more efficient in many instances than forward analysis, especially when there is a taint flow in individual, different classes. Furthermore, on

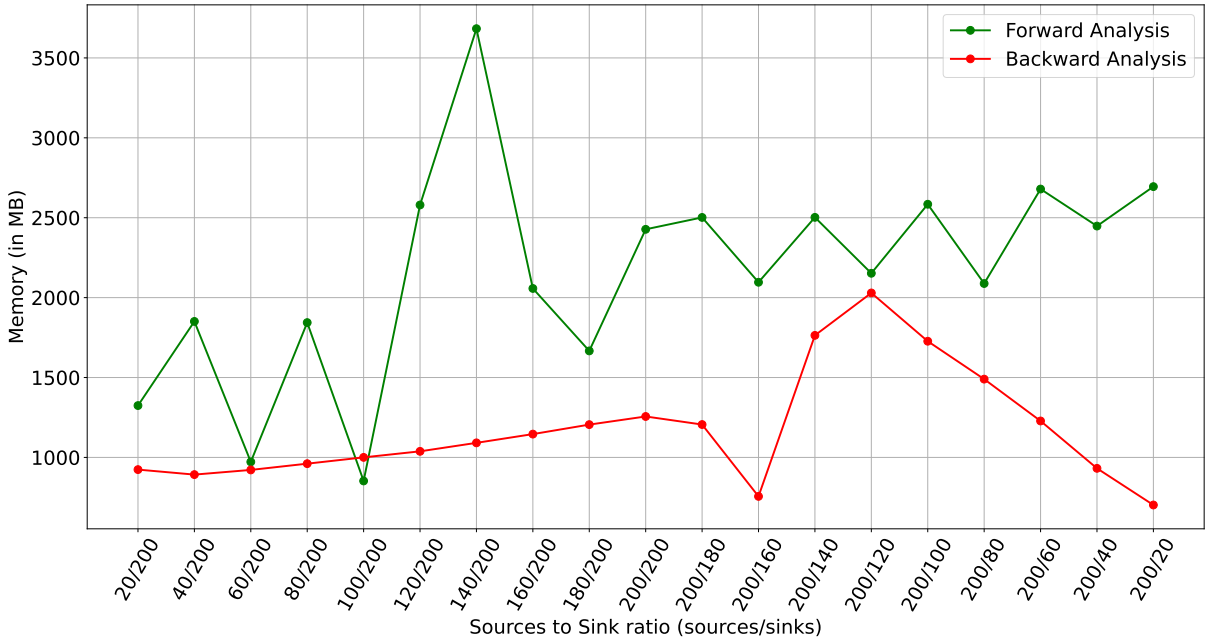

Figure 4.9: Memory evaluation of 19 applications with pattern P3, series_simple_icc_callback

average, backward analysis took 50% of the analysis time compared to forward analysis while consuming a comparable amount of memory. Next, to find an efficient analysis, we compared the code metrics of applications with their resource consumption. We found that code metrics such as lines of code and classes can hint towards efficient taint analysis but are unreliable for prediction. The present number of sources and sinks were found to be the primary indicators of efficient analysis.

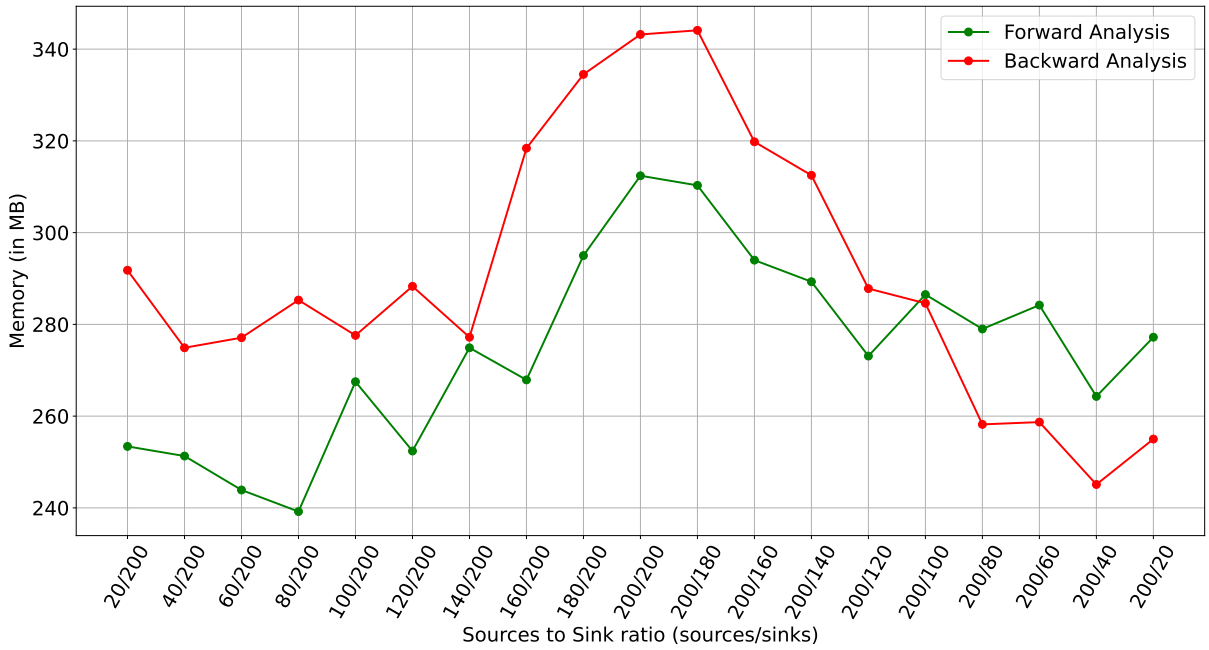

Figure 4.10: Memory evaluation of 19 applications with pattern P4, iteration_simple

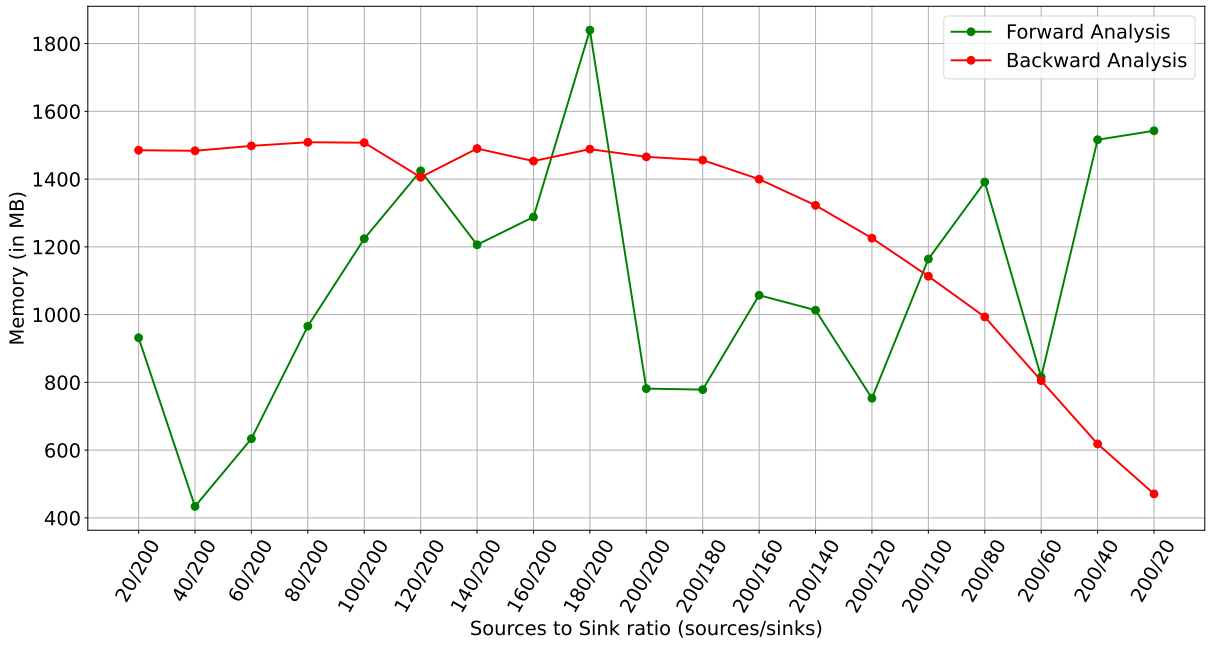

Figure 4.11: Memory evaluation of 19 applications with pattern P5, iteration_simple_icc

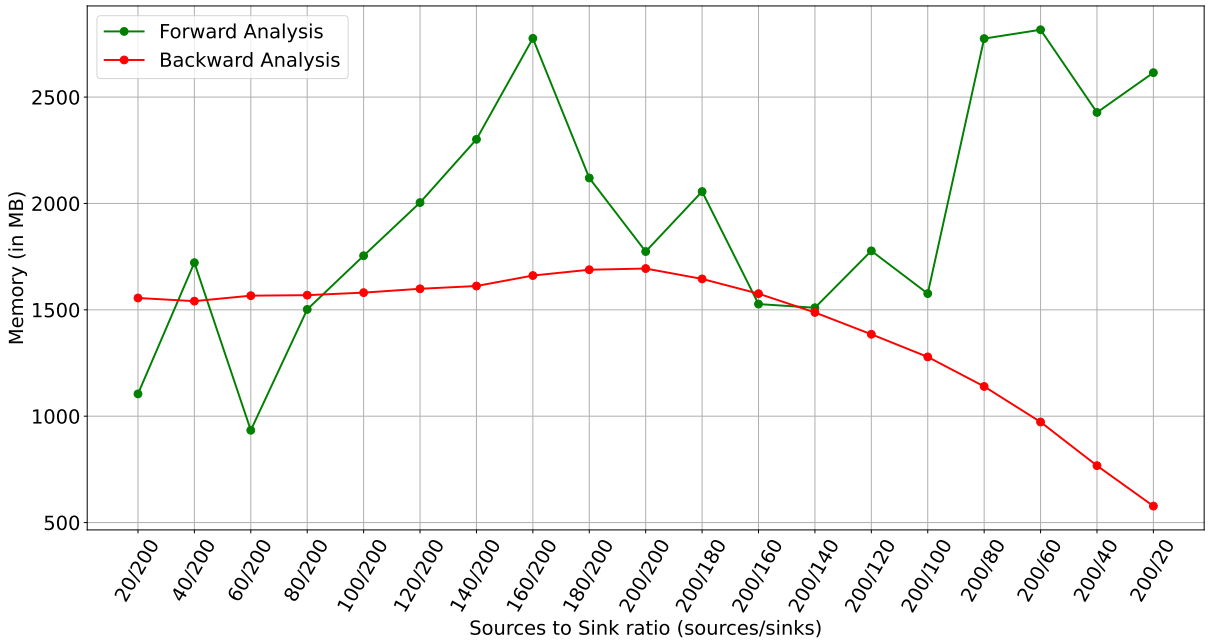

Figure 4.12: Memory evaluation of 19 applications with pattern P6, iteration_simple_icc_callback

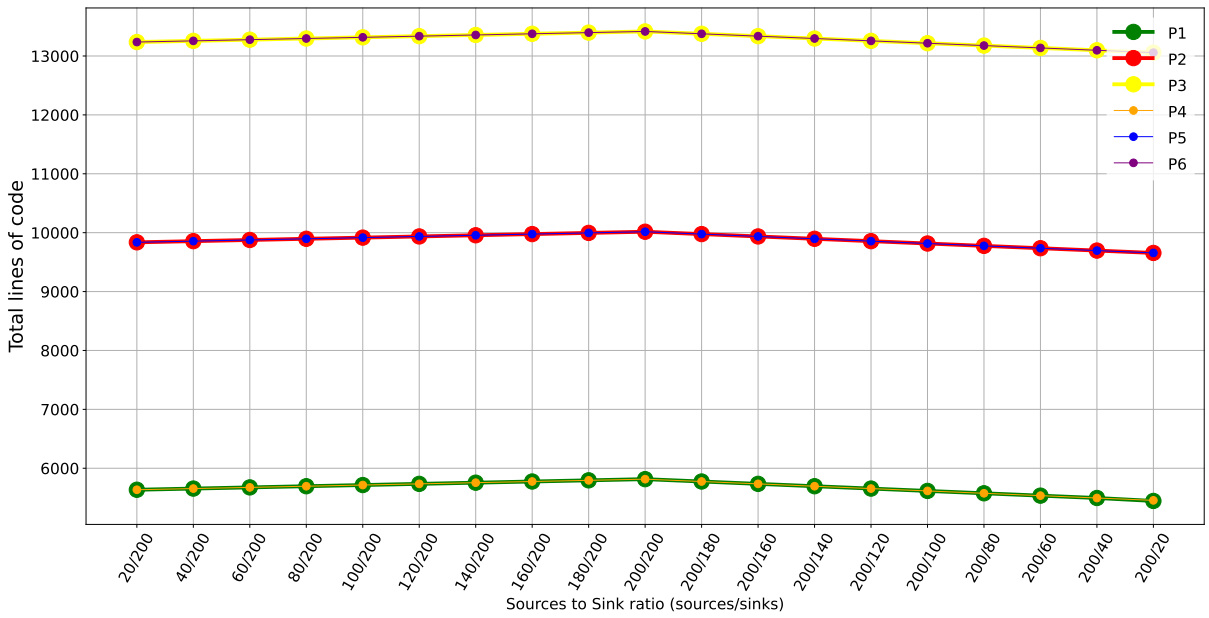

Figure 4.13: Total lines of code in 19 applications of pattern P1, P2, P3, P4, P5 and P6

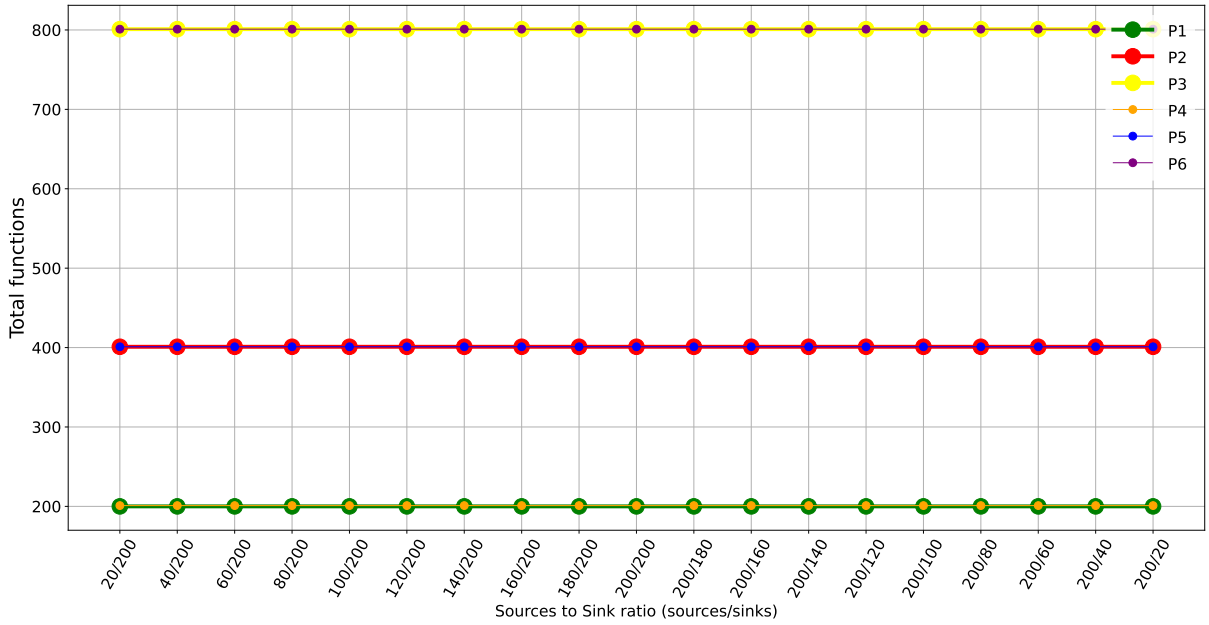

Figure 4.14: Total functions present in 19 applications of pattern P1, P2, P3, P4, P5 and P6

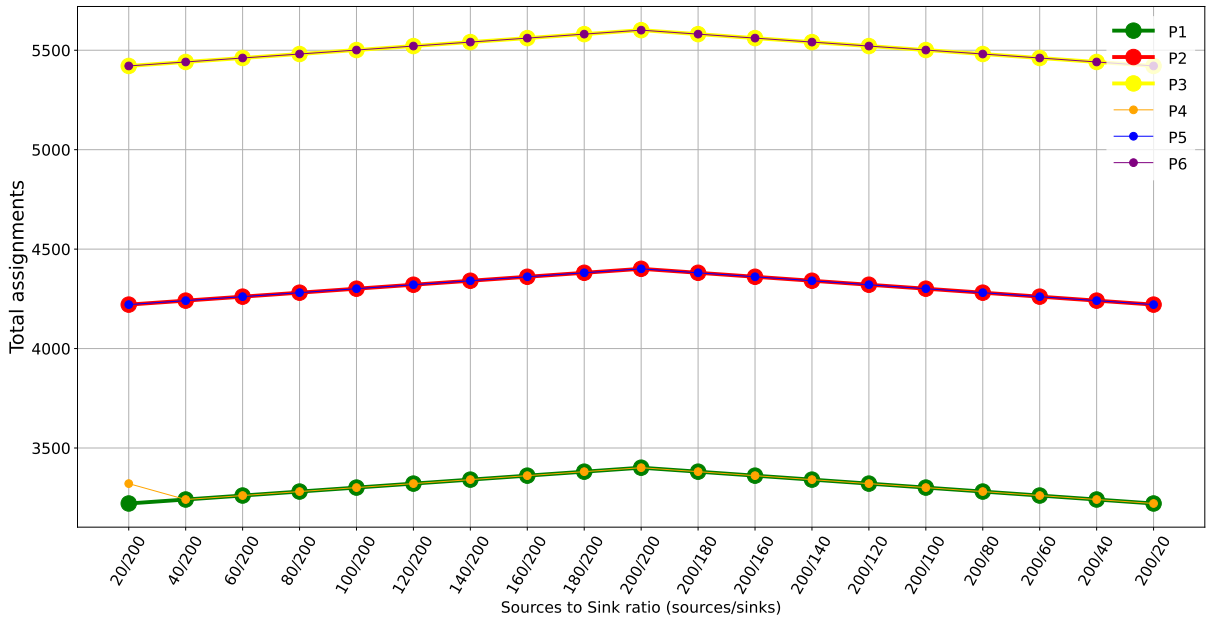

Figure 4.15: Total assignment operations present in 19 applications of pattern P1, P2, P3, P4, P5 and P6

5 Related Work

5.0.1 Taint analysis tools

Lerch et al., in their work with FlowTwist [9], found that a backward data flow analysis is advantageous when there are only a few sinks but a significant number of potential sources. FlowTwist introduced an inside-out taint analysis approach that uses the coordination of both forward and backward analysis based on the IFDS framework to detect confused deputy problems in Java Class Library, allowing analysis to scales better and perform faster than pure forward analysis.
FlowDroid [1] is a taint analysis tool based on IFDS for Android applications which are highly precise and models the Android lifecycle to manage Android callbacks. Work done by Lange [8] implements backward taint analysis in FlowDroid and presents initial benchmarking and experimentation in comparison to traditional forward analysis in FlowDroid. The benchmarking involved investigation of apriori known parameters such as code size, present sources, sinks, and a fast pre-analysis with 200 applications from the Google Play store to find favorable taint analysis direction in order of efficiency. Although, no known parameters correlate with efficient taint analysis direction prediction. The 60% of the tested applications in their application dataset demonstrated a clear advantage for per-app performance respective to the chosen direction. In the remaining 40%, the performance was found to be comparable.
FastDroid [31], is a novel static taint analysis tool based on the initial version of FastDroid[30] for efficient taint analysis of large-scale applications, while maintaining high precision and recall. Unlike analyses based on data flow analysis, FastDroid adopts a distinct taint analysis approach. It traces taint value propagation and constructs taint value graphs to represent the relationships among taint values in a flow-insensitive analysis. Then taint flows are identified by validating found potential taint flows from taint value graphs against control flow graph of the program. FastDroid was evaluated against 3 benchmarks and comparison with FlowDroid demonstrated FastDroid's superior precision and recall while also proving to be efficient in terms of processing time.
Many tools, such as FlowTwist, FlowDroid, and FastDroid, utilize IFDS [20] as a prominent choice for conducting taint analysis. Another framework, Synchronized pushdown systems (SPDS) [25], can also be used to formulate taint analyses. SPDS provides an alternative to IFDS and offers a precise context-sensitive, flow-sensitive, and field-sensitive data flow analysis. SPDS achieves this by combining two pushdown systems: one for matching calls and returns and the other for matching field stores and loads. Unlike k-limiting pushdown systems, SPDS delivers a concise and precise representation without the need for any k-limiting, comparable to k =

1. Synchronized pushdown systems can be easily applied to perform backward analysis, which determines where the content of a variable originates from in field-sensitive information. An example of a backward taint analysis extension using SPDS is found in SecuCheck's backward taint analysis implementation [13]. Additionally, multiple tools like SWAN [26], and CogniCrypt [6] utilize SPDS for data flow analysis. SWAN is a static program analysis framework designed for data flow analysis in Swift applications. It can detect API misuse through typestate analysis and identify security vulnerabilities using taint analysis. CogniCrypt, on the other hand, identifies cryptographic API misuses based on rules written in a domain-specific language CrySL [7].
Benchmarking plays an important role in evaluating analysis tools and used frameworks within. Since developing such benchmarks to test analysis tools and include evolving feature is not feasible in terms of cost and efficiency, a single benchmark is usually developed to test all features and metrics, or an automated benchmark generation tool such as GenBenchDroid [21] is used.
DroidBench [1] is an open micro-benchmark suite used to evaluate the effectiveness of taint-analysis tools for Android applications. It was initially developed to assess FlowDroid and compare it with other tools. The suite includes test cases for various static-analysis challenges, such as field sensitivity, object sensitivity, and access-path lengths. It also covers Android-specific challenges like modeling the application lifecycle, handling asynchronous callbacks, and interacting with the user interface. Both static and dynamic taint analyses can be assessed using DroidBench.
ICC-Bench [5] is another micro-benchmark suite that only consists of benchmark cases that implements Inter-component communication (ICC). ICC-Bench contains 24 micro-benchmark cases enabling a more thorough analysis for ICC. Whereas DroidBench 2.0 consists of 120 and DroidBench 3.0 consists of 190 cases covering many analysis problems like aliasing, callbacks including ICC, and more.

5.0.2 Benchmarks

TaintBench [10], on the other hand, is the first real-world malware benchmark suite with documented tait flows. It contains 39 real-world applications with comprehensive taint flow information, enabling analysis tools to compare their results. It also defines a benchmark framework for real-world taint analysis that improves performance and usability in taint flow documentation and inspection.
LAVA (Large-scale Automated Vulnerability Addition) [4] is another dynamic taint analysis-based technique for producing ground truth of bugs and taint flow by injecting them into the program source code. The resulting program can evaluate taint analysis tools and approaches against the generated ground truth.
Automated Benchmark Management (ABM) [3] introduces a methodology for automating and managing reliable, domain-specific benchmark suites for program analysis and tool evaluation, ensuring they are kept up-to-date. ABM uses filters to create benchmarks within a specific domain consistently. By repeatedly applying the same set of filters, it releases updates to the benchmark suite, ensuring relevance over time. ABM methodology created a suite of 139 projects within the domain of Java web applications sourced from GitHub.

6 Conclusion and Future Work

The main goal of this work is to analyze the behavior of taint analysis in different directions and reason about its efficiency respective to present and known metrics in the application to predict the efficient taint analysis direction. We could not find applications with a specific number of sources, sinks, and taint flow, with ground truth. Hence, to evaluate the taint analysis efficiency we built our applications. We first extended the automated benchmark case generator tool GenBenchDroid and removed many limitations. Then with extended GenBenchDroid, we generated a special benchmark suite, TABS, with 114 Android applications containing different numbers of sources, sinks, and behavior with six distinct styles. To analyze the behavior of taint analysis we employed FlowDroid in forward and backward analysis setting. An in-depth evaluation of runtime and memory was performed to understand the taint analysis behavior. Additionally, other application metrics like lines of code, number of classes, and the number of assignments were considered in the evaluation. The evaluation showed a greater correlation between the present number of sources and sink and taint flow with efficiency. The backward analysis was found to outperform forward analysis by efficiency of 50% in terms of time, and with slightly lower memory consumption comparable to forward analysis. Forward analysis is found to be efficient where the number of sources is significantly lower than that of sinks. Overall, the efficiency of the analysis was observed to be highly relatable with the present number of sources, sinks, and taint flow. In summary, this work presents an in-depth evaluation of forward and backward taint analysis with respect to many application properties while strengthening the understanding of the behavior of taint analysis in order to predict the efficient taint analysis direction.

6.0.1 Future Work

This work utilized GenBenchDroid, a benchmark generation tool, and FlowDroid, an analysis tool. Therefore, two concrete future work is introduced in the following sections.

Benchmarking

114 applications in TABS do not cover all possible evaluation cases for evaluating the analysis efficiency. Hence the extended GenBenchDroid can be further utilized to generate various Android application cases. TABS contains taint flows with sinks in individual classes, whereas sources are always in a single class. Alternative, many taint flow patterns can be tested in future work. For example, all sinks leaking in a single class while sources are located in many distinct

classes. Furthermore, extended GenBenchDroid can be evolved by adding more module settings and code abstracts to expand the possibility of generating applications with variety of feature for evaluation.

Analysis tool

Our test evaluation with SecuCheck revealed the possibility of evaluation with an Android application but the analysis failed in many cases and revealed no results. Hence, SecuCheck can be extended to evaluate Android and Java applications. Considering empirical evaluation in this study with FlowDroid, a similar study could be performed for Java applications using SecuCheck or Android application after extensions . It could further extend to comparing the data flow analysis framework in taint analysis, given that SecuCheck and FlowDroid utilizes two different solvers.

Bibliography

[1] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[2] Abdullah Al Murad Chowdhury and Shamsul Arefeen. Software risk management: importance and practices. In *IJCIT, ISSN*, pages 2078–5828. Citeseer, 2011.

[3] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, page 13–17, New York, NY, USA, 2016. Association for Computing Machinery.

[4] 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, 2016.

[5] ICC-Bench. Accessed 2023-03-15 url: https://github.com/fgwei/icc-bench.

[6] Stefan Krüger. Cognicrypt - the secure integration of cryptographic software. Universitaetsbibliothek Paderborn, October 2020.

[7] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. volume 47, pages 2382–2400, 2021.

[8] Tim Lange. Implementation and evaluation of a static backwards data flow analysis in flowdroid. Bachelor thesis, Technische Universität, Darmstadt, 2022.

[9] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014.

[10] L. Luo, F. Pauck, and G. et al. Piskachev. Taintbench: Automatic real-world malware benchmarking of android taint analyses. In *Empir Software Eng 27, 16*, 2022.

[11] Linghui Luo, Eric Bodden, and Johannes Späth. A qualitative analysis of android taint-analysis results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019.

[12] Robert Muth. Register liveness analysis of executable code. Citeseer, 1998.

[13] Enri Ozuni. Extending fluenttql, specifying taint flows through a dsl. 2021.

[14] 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*, ESEC/FSE 2018, page 331–341, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Frank Piessens. A taxonomy of causes of software vulnerabilities in internet software. In *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 47–52. Citeseer, 2002.

[16] Goran Piskachev, Ranjith Krishnamurthy, and Eric Bodden. Secucheck: Engineering configurable taint analysis for software developers. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021.

[17] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. Codebase-adaptive detection of security-relevant methods. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[18] Goran Piskachev, Johannes Späth, Ingo Budde, and Eric Bodden. Fluently specifying taint-flow queries with fluenttql. volume 27, page 1–33. Springer, 2022.

[19] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. 01 2014.

[20] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.

[21] Stefan Schott and Felix Pauck. Benchmark fuzzing for android taint analyses. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 12–23, 2022.

[22] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781, 2012.

[23] SNYK. What is taint analysis and why do i care? url: https://snyk.io/product/snyk-code visited on may 13th, 2023.

[24] Sonarsource. What is taint analysis and why do i care? url: https://www.sonarsource.com/blog/what-is-taint-analysis/ accessed on may 9th, 2023.

[25] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[26] Daniil Tiganov, Jeff Cho, Karim Ali, and Julian Dolby. Swan: A static analysis framework for swift. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1640–1644, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[28] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. volume 13, page 181–210, New York, NY, USA, apr 1991. Association for Computing Machinery.

[29] Awad A. Younis, Yashwant K. Malaiya, and Indrajit Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8, 2014.

[30] Jie Zhang, Cong Tian, and Zhenhua Duan. Fastdroid: Efficient taint analysis for android applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 236–237, 2019.

[31] Jie Zhang, Cong Tian, and Zhenhua Duan. An efficient approach for taint analysis of android applications. volume 104, page 102161, 2021.

A Appendix

A.1 Digital Appendix

All the extended versions of the tools and new implementations can be found in individual repository under branch named **tab**. Since, all applications except GenBenchDroid are a Java application, they can be built with `mvn clean install`. For detailed arrangement and use of the respective tool, a descriptive readme in each project is provided. GenBenchDroid utilizes a gitlab pipeline to generate applications respective to provided TMC while there exist a python script that is triggered by pipeline to completely build the benchmark suite.
GenBenchDroid's extension can be found in the following repository and all TMC configuration used to build the TABS resides in `appConfig` directory of this repository.

- url: https://git.cs.uni-paderborn.de/empirical-analysis/tools/genbenchdroid
- branch: tab
- commit-id: c31618a8903e175733d21675e0efc357bbc895fb
- commit-date: May 15, 2023
- pipeline script: .gitlab-ci.yml
- application generation script: startup.py
- comment: application generation script can be changed to any directory containing TMC

The SecuCheck implementation has 3 repository which are utilized for different purposes. SecuCheck, that host cli-tool, require SecuCheck-core to be built first. All other repository can be built in any order. No pipeline setting is available for SecuCheck related repository and can be built individually with `maven clean install`.

1. SecuCheck
    - url: https://git.cs.uni-paderborn.de/empirical-analysis/tools/secucheck
    - branch: tab
    - commit-id: 996be0f03076631e0ba29a6d551270c208af82ee
    - commit-date: May 18, 2023

2. SecuCheck-core
    - url: https://git.cs.uni-paderborn.de/empirical-analysis/tools/secucheck-core
    - branch: tab
    - commit-id: 39abc01aaa01cd5b4c6f37e298ea200d18db48b3
    - commit-date: May 18, 2023
3. SecuCheck-catelog
    - url: https://git.cs.uni-paderborn.de/empirical-analysis/tools/secucheck-catelog
    - branch: tab
    - commit-id: bab18a5bf064c740793535c2b18cddfe1351a231
    - commit-date: May 18, 2023
    - comment: contains fluentTQL specifications for demo-project, and tabs-specification

Taint Analysis Benchmark Suite (TABS) contains the complete benchmark suite with GenBenchDroid's TMC configurations used to generate Android application, FlowDroid executable (.jar) used for empirical analysis, its required configurations, and all analysis results and logs. A detailed description is provided for its all available contents in readme with the attached commit. Additionally, the analysis can be repeated via gitlab pipeline where the analysis is handled by the python script called by the pipeline.

- url: https://git.cs.uni-paderborn.de/empirical-analysis/tools/tab
- branch: tab
- evaluation script path: startup.py
- pipeline script path: .gitlab-ci.yml
- simple_arraybridge_app: at path `simple_arraybridge_app` from root of repository.
- simple_list_clone_app: at path `SecuCheck/simple_list_clone_app` from root of repository.
- simple_stringbuild_app: at path `SecuCheck/simple_stringbuild_app` from root of repository.
- secucheck-demo: results at path `SecuCheck/secucheck-demo` from root of repository.

For empirical evaluation the application from the following commit id is used:

- commit-id: 2bbf9a53f64e44eb78389c9c3e046a1ee29c4c2f
- commit-date: May 5, 2023

For SecuCheck's taint analysis over android applications the following commit id is used:

- commit-id: db5e4f813d1f2ba0bf6cad5f31205c64d5e2db4a
- commit-date: May 18, 2023

FlowDroid version used for empirical evaluation:

- url: https://github.com/secure-software-engineering/FlowDroid
- branch: develop
- commit-id: c5b1f669d7e499d0f34871767bee01face5c8c73
- commit-date: April 20, 2023

Name	Type
DImeiSource	SOURCE
EmptySource	BRIDGE
DImplicitSmsSink	SINK
DLogSink	SINK
DReflectionMethod1NonSink	SINK
DSmsSink	SINK
DArrayBridge	BRIDGE
DArrayExampleBridge	BRIDGE
DArraySanitizerBridge	BRIDGE
DAppendToStringBridge	BRIDGE
DStringBufferBridge	BRIDGE
DSimpleUnreachableBridge	BRIDGE
DAliasingSanitizerBridge	BRIDGE
DBluetoothDetectionBridge	BRIDGE
DButtonCallbackBridge	BRIDGE
DDatacontainerBridge	BRIDGE
DEmptySource	BRIDGE
DListBridge	BRIDGE
DListCloneBridge	BRIDGE
DObfuscation1Bridge	BRIDGE
DObfuscation2Bridge	BRIDGE
DPublicApiPointBridge	BRIDGE
DSimpleRecursionBridge	BRIDGE
DSimpleSanitizationBridge	BRIDGE

Table A.1: New modules added to GenBenchDroid for flexible design

Name	Type
BasicTemplate	TEMPLATE
ImeiSource	SOURCE
ListCloneBridge	BRIDGE
ArrayBridge	BRIDGE
AppendToStringBridge	BRIDGE
SimpleRecursionBridge	BRIDGE
ListBridge	BRIDGE
SmsSink	SINK
DListBridge	BRIDGE
IccGlobalFieldBridge	BRIDGE
SmsSink	BRIDGE
ButtonCallbackBridge	BRIDGE
EmptySource	BRIDGE

Table A.2: GenBenchDroid modules used to generate TABS

Table A.3: List of applications contributed in TABS

id	Pattern	S	SI	LOC	Assign	FN	CLS
1	series_simple (P1)	20	200	5635	3221	200	1
2	series_simple (P1)	40	200	5655	3241	200	1
3	series_simple (P1)	60	200	5675	3261	200	1
4	series_simple (P1)	80	200	5695	3281	200	1
5	series_simple (P1)	100	200	5715	3301	200	1
6	series_simple (P1)	120	200	5735	3321	200	1
7	series_simple (P1)	140	200	5755	3341	200	1
8	series_simple (P1)	160	200	5775	3361	200	1
9	series_simple (P1)	180	200	5795	3381	200	1
10	series_simple (P1)	200	200	5815	3401	200	1
11	series_simple (P1)	200	180	5775	3381	200	1
12	series_simple (P1)	200	160	5735	3361	200	1
13	series_simple (P1)	200	140	5695	3341	200	1
14	series_simple (P1)	200	120	5655	3321	200	1
15	series_simple (P1)	200	100	5615	3301	200	1
16	series_simple (P1)	200	80	5575	3281	200	1
17	series_simple (P1)	200	60	5535	3261	200	1
18	series_simple (P1)	200	40	5495	3241	200	1
19	series_simple (P1)	200	20	5445	3221	200	1
20	series_simple_icc (P2)	20	200	9836	4221	401	201
21	series_simple_icc (P2)	40	200	9856	4241	401	201
22	series_simple_icc (P2)	60	200	9876	4261	401	201
23	series_simple_icc (P2)	80	200	9896	4281	401	201
24	series_simple_icc (P2)	100	200	9916	4301	401	201
25	series_simple_icc (P2)	120	200	9936	4321	401	201
26	series_simple_icc (P2)	140	200	9956	4341	401	201
27	series_simple_icc (P2)	160	200	9976	4361	401	201
28	series_simple_icc (P2)	180	200	9996	4381	401	201
29	series_simple_icc (P2)	200	200	10016	4401	401	201
30	series_simple_icc (P2)	200	180	9976	4381	401	201
31	series_simple_icc (P2)	200	160	9936	4361	401	201
32	series_simple_icc (P2)	200	140	9896	4341	401	201
33	series_simple_icc (P2)	200	120	9856	4321	401	201
34	series_simple_icc (P2)	200	100	9816	4301	401	201
35	series_simple_icc (P2)	200	80	9776	4281	401	201
36	series_simple_icc (P2)	200	60	9736	4261	401	201
37	series_simple_icc (P2)	200	40	9696	4241	401	201
38	series_simple_icc (P2)	200	20	9656	4221	401	201
39	series_simple_icc_callback (P3)	20	200	13237	5421	801	201
40	series_simple_icc_callback (P3)	40	200	13257	5441	801	201
41	series_simple_icc_callback (P3)	60	200	13277	5461	801	201
42	series_simple_icc_callback (P3)	80	200	13297	5481	801	201
43	series_simple_icc_callback (P3)	100	200	13317	5501	801	201
44	series_simple_icc_callback (P3)	120	200	13337	5521	801	201

Continued on next page

Table A.3 List of applications contributed in TABS

id	Pattern	S	SI	LOC	Assign	FN	CLS
45	series_simple_icc_callback (P3)	140	200	13357	5541	801	201
46	series_simple_icc_callback (P3)	160	200	13377	5561	801	201
47	series_simple_icc_callback (P3)	180	200	13397	5581	801	201
48	series_simple_icc_callback (P3)	200	200	13417	5601	801	201
49	series_simple_icc_callback (P3)	200	180	13377	5581	801	201
50	series_simple_icc_callback (P3)	200	160	13337	5561	801	201
51	series_simple_icc_callback (P3)	200	140	13297	5541	801	201
52	series_simple_icc_callback (P3)	200	120	13257	5521	801	201
53	series_simple_icc_callback (P3)	200	100	13217	5501	801	201
54	series_simple_icc_callback (P3)	200	80	13177	5481	801	201
55	series_simple_icc_callback (P3)	200	60	13137	5461	801	201
56	series_simple_icc_callback (P3)	200	40	13097	5441	801	201
57	series_simple_icc_callback (P3)	200	20	13057	5421	801	201
58	iteration_simple (P4)	20	200	5635	3321	201	1
59	iteration_simple (P4)	40	200	5655	3241	201	1
60	iteration_simple (P4)	60	200	5675	3261	201	1
61	iteration_simple (P4)	80	200	5695	3281	201	1
62	iteration_simple (P4)	100	200	5715	3301	201	1
63	iteration_simple (P4)	120	200	5735	3321	201	1
64	iteration_simple (P4)	140	200	5755	3341	201	1
65	iteration_simple (P4)	160	200	5775	3361	201	1
66	iteration_simple (P4)	180	200	5795	3381	201	1
67	iteration_simple (P4)	200	200	5815	3401	201	1
68	iteration_simple (P4)	200	180	5775	3381	201	1
69	iteration_simple (P4)	200	160	5735	3361	201	1
70	iteration_simple (P4)	200	140	5695	3341	201	1
71	iteration_simple (P4)	200	120	5655	3321	201	1
72	iteration_simple (P4)	200	100	5615	3301	201	1
73	iteration_simple (P4)	200	80	5575	3281	201	1
74	iteration_simple (P4)	200	60	5535	3261	201	1
75	iteration_simple (P4)	200	40	5495	3241	201	1
76	iteration_simple (P4)	200	20	5455	3221	201	1
77	iteration_simple_icc (P5)	20	200	9836	4221	401	201
78	iteration_simple_icc (P5)	40	200	9856	4241	401	201
79	iteration_simple_icc (P5)	60	200	9876	4261	401	201
80	iteration_simple_icc (P5)	80	200	9896	4281	401	201
81	iteration_simple_icc (P5)	100	200	9916	4301	401	201
82	iteration_simple_icc (P5)	120	200	9936	4321	401	201
83	iteration_simple_icc (P5)	140	200	9956	4341	401	201
84	iteration_simple_icc (P5)	160	200	9976	4361	401	201
85	iteration_simple_icc (P5)	180	200	9996	4381	401	201
86	iteration_simple_icc (P5)	200	200	10016	4401	401	201
87	iteration_simple_icc (P5)	200	180	9976	4381	401	201
88	iteration_simple_icc (P5)	200	160	9936	4361	401	201
89	iteration_simple_icc (P5)	200	140	9896	4341	401	201

Continued on next page

Table A.3 List of applications contributed in TABS

id	Pattern	S	SI	LOC	Assign	FN	CLS
90	iteration_simple_icc (P5)	200	120	9856	4321	401	201
91	iteration_simple_icc (P5)	200	100	9816	4301	401	201
92	iteration_simple_icc (P5)	200	80	9776	4281	401	201
93	iteration_simple_icc (P5)	200	60	9736	4261	401	201
94	iteration_simple_icc (P5)	200	40	9696	4241	401	201
95	iteration_simple_icc (P5)	200	20	9656	4221	401	201
96	iteration_simple_icc_callback (P6)	20	200	13237	5421	801	201
97	iteration_simple_icc_callback (P6)	40	200	13257	5441	801	201
98	iteration_simple_icc_callback (P6)	60	200	13277	5461	801	201
99	iteration_simple_icc_callback (P6)	80	200	13297	5481	801	201
100	iteration_simple_icc_callback (P6)	100	200	13317	5501	801	201
101	iteration_simple_icc_callback (P6)	120	200	13337	5521	801	201
102	iteration_simple_icc_callback (P6)	140	200	13357	5541	801	201
103	iteration_simple_icc_callback (P6)	160	200	13377	5561	801	201
104	iteration_simple_icc_callback (P6)	180	200	13397	5581	801	201
105	iteration_simple_icc_callback (P6)	200	200	13417	5601	801	201
106	iteration_simple_icc_callback (P6)	200	180	13377	5581	801	201
107	iteration_simple_icc_callback (P6)	200	160	13337	5561	801	201
108	iteration_simple_icc_callback (P6)	200	140	13297	5541	801	201
109	iteration_simple_icc_callback (P6)	200	120	13257	5521	801	201
110	iteration_simple_icc_callback (P6)	200	100	13217	5501	801	201
111	iteration_simple_icc_callback (P6)	200	80	13177	5481	801	201
112	iteration_simple_icc_callback (P6)	200	60	13137	5461	801	201
113	iteration_simple_icc_callback (P6)	200	40	13097	5441	801	201
114	iteration_simple_icc_callback (P6)	200	20	13057	5421	801	201

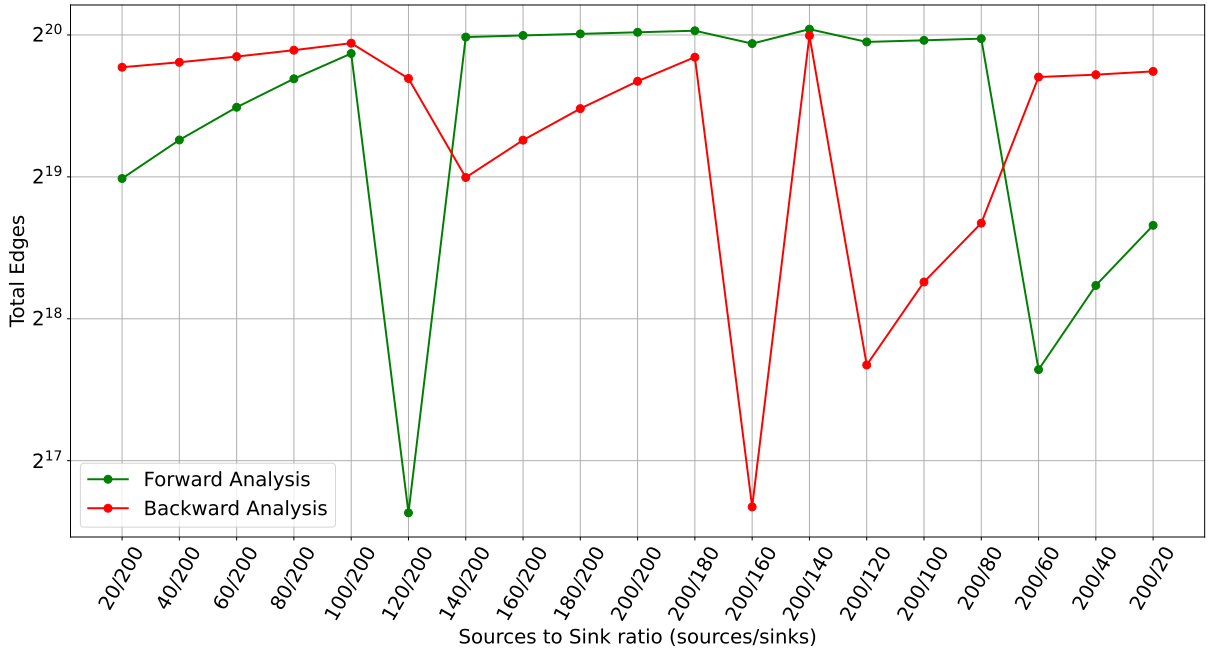

Figure A.1: Total edge comparison of 19 applications with pattern P1

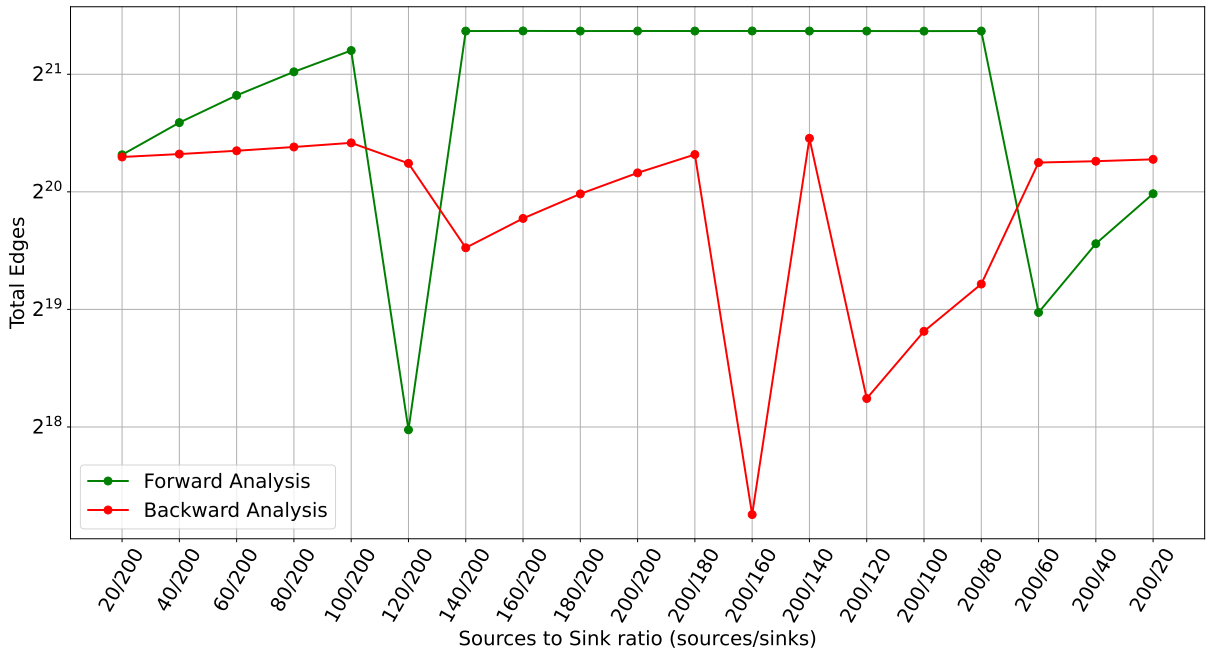

Figure A.2: Total edge comparison of 19 applications with pattern P2

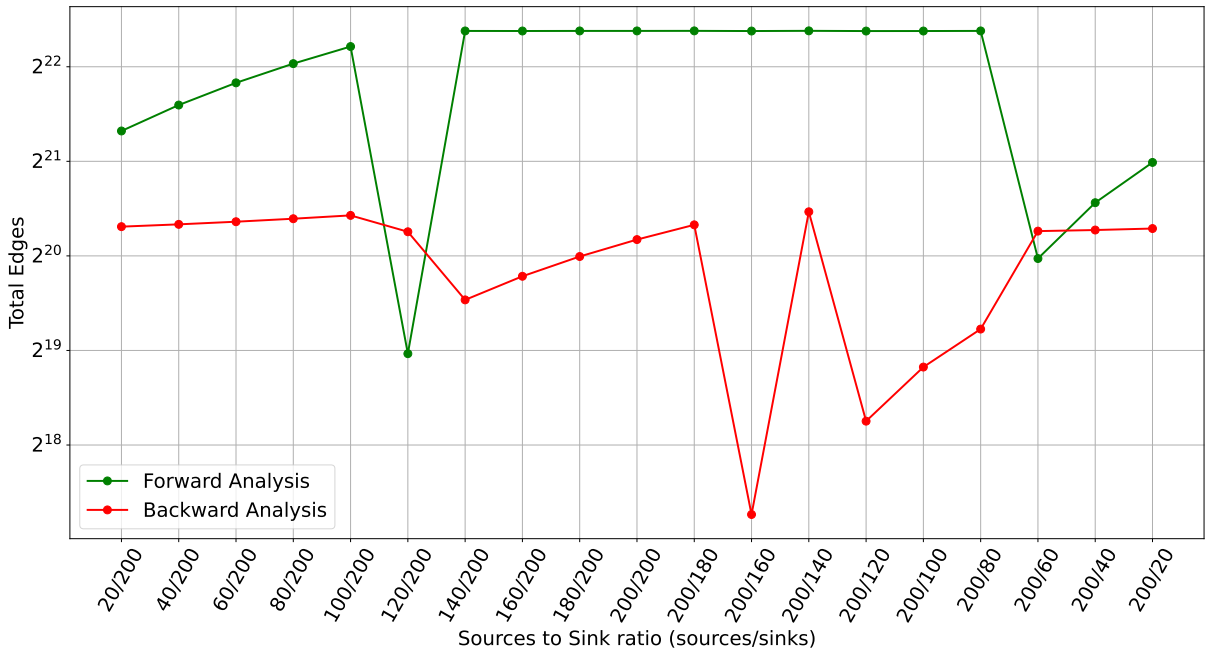

Figure A.3: Total edge comparison of 19 applications with pattern P3

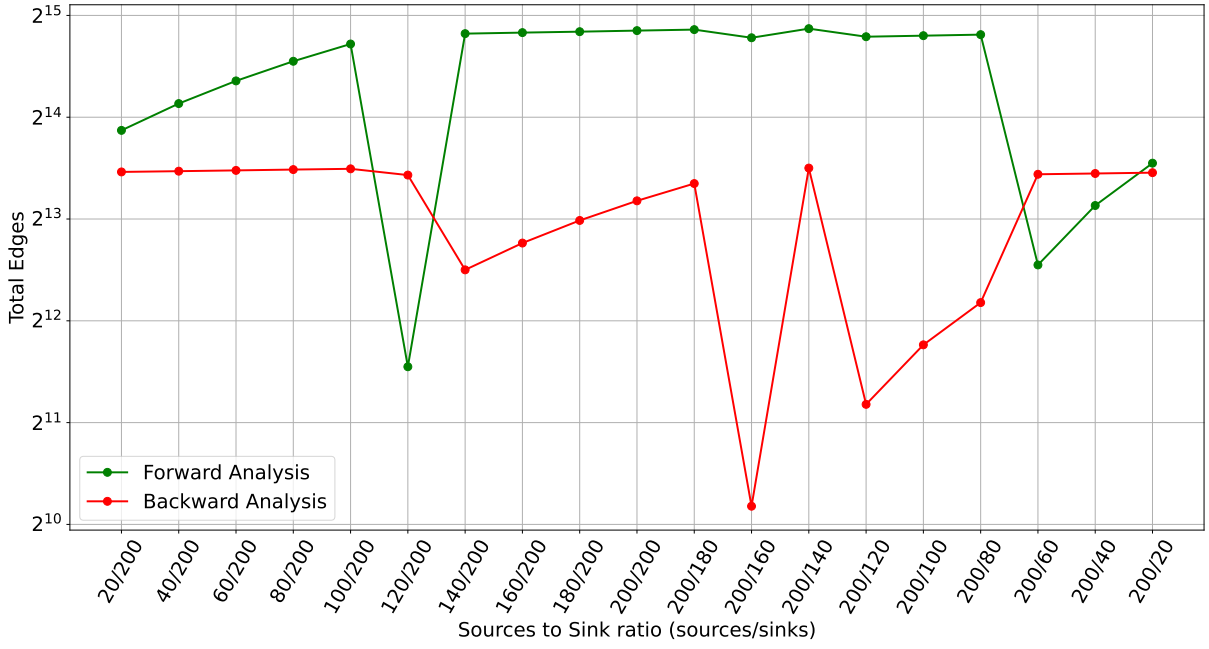

Figure A.4: Total edge comparison of 19 applications with pattern P4

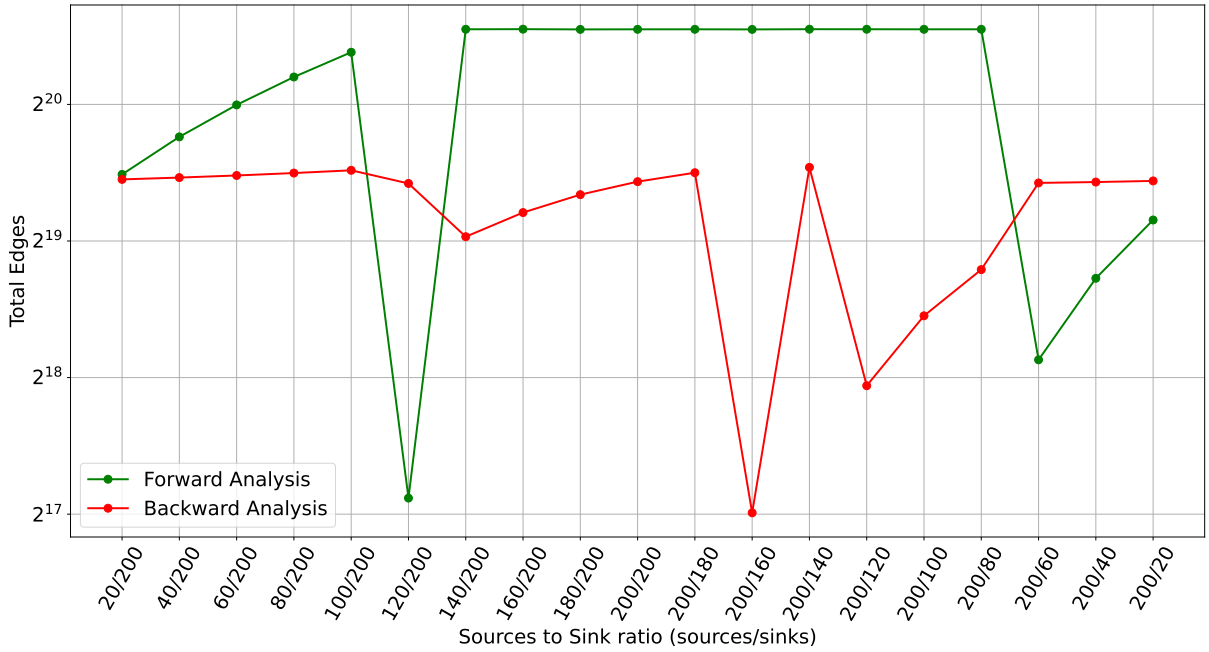

Figure A.5: Total edge comparison of 19 applications with pattern P5

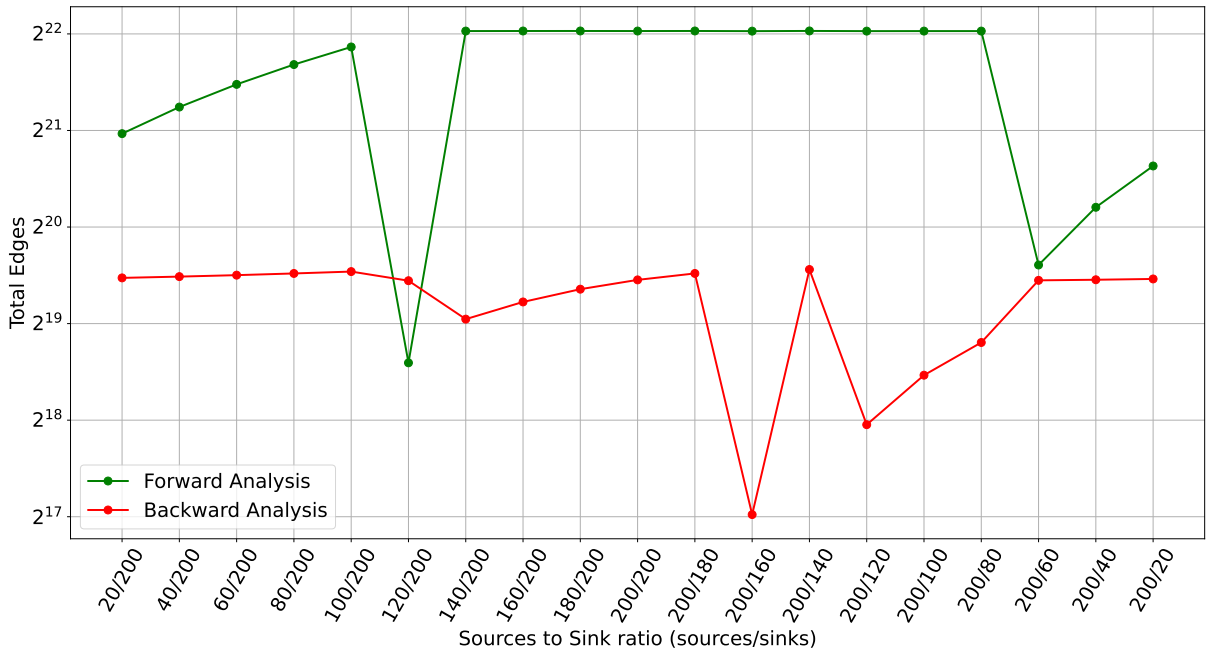

Figure A.6: Total edge comparison of 19 applications with pattern P6

ID	fluentTQL Specifications	AC	F	B
1	CWE643_XpathInjection_TF1_WithMethodSign	1	1	-
2	CWE89_SqlInjection_TF1_WithEntryPoints	1	1	1
3	CWE89_SqlInjection_TF2_WithEntryPoints	1	1	1
4	CWE89_SqlInjection_TF3_WithEntryPoints	1	1	-
5	CWE79_CrossSiteScripting_WithMethodSign	1	1	1
6	CWE601_OpenRedirect_TF1_WithMethodSign	1	1	1
7	CWE601_OpenRedirect_TF2_WithMethodSign	1	1	-
8	CWE601_OpenRedirect_TF3_WithMethodSign	1	1	1
9	CWE89_SqlInjection_TF1	1	1	1
10	CWE89_SqlInjection_TF2	1	1	1
11	CWE89_SqlInjection_TF3	1	1	-
12	CWE79_CrossSiteScripting_WithEntryPoints	1	1	1
13	CWE78_OsCommandInjection_TF1	1	1	1
14	CWE78_OsCommandInjection_TF2	1	1	1
15	CWE643_XpathInjection_TF1	1	1	-
16	CWE22_PathTraversal_WithMethodSign	1	1	1
17	CWE311_MissingEncryption_WithEntryPoints	2	2	2
18	CWE20_ImproperInputValidation	1	1	1
19	CWE78_OsCommandInjection_TF1_WithEntryPoints	1	1	1
20	CWE78_OsCommandInjection_TF2_WithEntryPoints	1	1	1
21	CWE311_MissingEncryption_WithMethodSign	2	2	2
22	CWE78_OsCommandInjection_TF1_WithMethodSign	1	1	1
23	CWE78_OsCommandInjection_TF2_WithMethodSign	1	1	1
24	CWE20_ImproperInputValidation_WithEntryPoints	1	1	1
25	CWE643_XpathInjection_TF1_WithEntryPoints	1	1	-
26	CWE89_SqlInjection_TF1_WithMethodSign	1	1	1
27	CWE89_SqlInjection_TF2_WithMethodSign	1	1	1
28	CWE89_SqlInjection_TF3_WithMethodSign	1	1	-
29	CWE20_ImproperInputValidation_WithMethodSign	1	1	1
30	CWE22_PathTraversal	1	1	1
30	CWE22_PathTraversal_WithMethodSign	1	1	1
31	CWE601_OpenRedirect_TF1_WithEntryPoints	1	1	1
32	CWE601_OpenRedirect_TF2_WithEntryPoints	1	1	-
33	CWE78_OsCommandInjection_TF1_WithEntryPoints	1	1	1
34	CWE79_CrossSiteScripting	1	1	1
35	CWE22_PathTraversal_WithEntryPoints	1	1	1
36	CWE311_MissingEncryption	2	2	2
37	CWE601_OpenRedirect_TF1	1	1	1
38	CWE601_OpenRedirect_TF2	1	1	-
39	CWE601_OpenRedirect_TF3	1	1	1
	Total Count	42	42	33

Table A.4: Taint flow reported by SecuCheck's forward and backward annalysis for demo-project