Finding Data Leaks in Xamarin Apps by Performing Taint Analysis on CIL Code

DEIS1027F18

MIKKEL CHRISTIAN LYBECK CHRISTENSEN
SIMON ELLEGAARD LARSEN
SØREN AKSEL HELBO BJERGMARK
THOMAS PILGAARD NIELSEN
Software Engineering
Aalborg University

SPRING 2018

AALBORG UNIVERSITY
STUDENT REPORT

Cassiopeia
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Tlf. 9940 9940
http://www.cs.aau.dk

Title
Finding Data Leaks in Xamarin Apps by Performing Taint Analysis on CIL Code

Theme
Master Thesis

Project period
Spring Semester 2018

Project group
deis1027f18

Members:
Mikkel Christian Lybeck Christensen
Simon Ellegaard Larsen
Søren Aksel Helbo Bjergmark
Thomas Pilgaard Nielsen

Supervisor
René Rydhof Hansen
Magnus Madsen

No. of printed copies 0

No. of pages 94

Appendices A - D

Completed 2018-06-15

Abstract:

Today, smartphones handle a lot of personal and private data of their users. With the implementation of GDPR, more focus is put on the handling of private data, and the importance of preventing data leaks.

In this report, we present a tool to perform taint analysis on Common Intermediate Language (CIL) code with the intent of analyzing Android apps made using Xamarin.

An intermediate language called Simple CIL (SCIL) is created to simplify the analysis on CIL code. SCIL is formally defined, and a flow analysis in regards to a control flow analysis (CFA) is defined as well. To accompany SCIL, Simple CIL Analyzer (SCIL/A) and Flix Analyzer (Flix/A) are presented to perform the analysis.

Together, SCIL/A and Flix/A are able to scan a Xamarin app and find potentially insecure data flow. This involves using a control flow graph (CFG), transforming to static single assignment (SSA) form, and resolving dynamic methods, in order to analyze the flow through the app. In an evaluation of SCIL/A and Flix/A where 2,866 Xamarin apps were scanned, 20% were flagged with potential problems.

Summary

In this report, a taint analysis of Common Intermediate Language (CIL) is presented with the intention of scanning Android apps made with Xamarin. The intermediate language Simple CIL (SCIL) is presented along with Simple CIL Analyzer (SCIL/A) and Flix Analyzer (Flix/A), which are tools to perform the analysis on the SCIL code. SCIL/A transforms CIL code to SCIL and then to Flix facts, where Flix/A performs the taint analysis on the Flix facts.

The project starts with the initial problem statement in Section 1.1:

> Apps have access to private data and the user does not know what happens to the data. How can a tool be created, which can track data through a smartphone app made using Xamarin?

From this initial problem statement, the problem area is analyzed first with a description of General Data Protection Regulation (GDPR) and then existing tools examined. The tool FlowDroid is used to perform a taint analysis on native Android apps, where 11.7% of 66,969 apps are found to potentially have data leakage. It is expected that the Xamarin apps have a similar number of apps with potential problems. This leads to the problem statement in Section 2.4, which is the basis for the rest of the report:

> How can a static taint analysis tool be constructed to examine the flow of data in apps developed using Xamarin?

In the theory chapter, various subjects are investigated for the purpose of generating requirements and obtaining the necessary knowledge for the implementation of a tool to analyze Xamarin apps, which leads to the requirements for the tool in Section 3.7.

In Chapter 4, the definition of SCIL and the implementation of SCIL/A and Flix/A are described. First, the structure, components and operational semantics for SCIL are defined and explained. The definition of SCIL is then used to define abstract domains and flow logic rules to perform a control flow analysis (CFA) of SCIL. This CFA is the foundation for the taint analysis.

With SCIL defined, SCIL/A is implemented with focus on the following:

- Android Package Kit (APK) parsing
- The creation of a control flow graph (CFG) and a call graph
- Conversion to static single assignment (SSA)
- Handling branching
- Handling asynchronous tasks

The output of SCIL/A is Flix facts for further analysis.

With the implementation of SCIL/A complete, Flix/A is implemented. Flix/A is executed with the Flix facts from SCIL/A, where Flix facts are used to determine taint propagation through the code. In Flix/A, extra effort was placed on implementing branching, method calls, and asynchronous tasks. In addition to the taint analysis of Flix/A, a simple string analysis based on character inclusion is performed.

To evaluate SCIL/A and Flix/A, automated tests in the form of unit tests are created, which test different aspects of the analysis. Furthermore, a general evaluation is performed by scanning apps with the tool, where statistics are collected. Furthermore, three detected apps are investigated, to check if the apps were correctly flagged by the analysis.

To round off the project, a discussion is done, where selected shortcomings of the project are discussed. In the conclusion, it is investigated if we succeeded in what was planned for the project in regards to the problem statements and the requirements. Finally, future work for the project is discussed, outlining what the next step for SCIL, SCIL/A, and Flix/A is.

Preface

This report and the associated product were developed as a project on the master's program in Software Engineering at Aalborg University. The project is based on the problem oriented model from AAU.

Basic knowledge about the structure of Android apps is expected of the reader. Throughout the report, any terms that the reader is not expected to know, will be explained.

In order to ensure reproducibility of the results in this projects, all source code can be found at: `https://github.com/sahb1239/SCIL`. Apps used in this report can be requested by contacting the authors of this report.

Reading guide

Source reference

References to source material use the Vancouver style. The number in square brackets at the end of a given statement refers to an entry in the bibliography at the end of the report. The following is an example of a source reference on a simple statement.

Aalborg University offers a master's degree in Software Engineering [0].

Bibliography

The Vancouver system also specifies the way individual entries in the bibliography are structured. The information is listed as follows: author(s), title of article/section, relevant pages, book title, editor, publisher, year of publication and ISBN. Any information that is unavailable or does not apply to a given entry may be excluded.

Figure and tables reference

All figures and tables in the report are assigned a unique number that can be referenced repeatedly throughout the report. The first number in the reference refers to the chapter of the figure, while the second number indicates the position in the sequence of figures/tables in the chapter. Immediately below, a short description is found. All figures and tables with no indicated source have been produced by the project group. An example is seen in Figure 1.

Figure 1: *Android logo [0]*

Listings

All listing adhere to the same rules as figures and tables, numbered separately. The code shown may have some parts removed that are irrelevant to the example, which will be marked with comments in the code. All listings are followed by the name of the programming language used in the example. An example of a code listing is seen in Listing 1.

```
1 public static void main(String[] args) {
2     System.out.println("Hello, World");
3 }
```

Listing 1: *Example of a listing (Java)*

Table of Contents

Chapter 1

Introduction

Smartphones are growing more popular every year, and the amount of personal data they have access to is growing. Apps that make use of this data have a responsibility to their users to keep it secure, but this is not always the case. Regardless of whether the apps are malicious or benign, some apps do leak personal information. To strengthen and unify data protection for citizens of the European Union (EU), the European Parliament has approved the General Data Protection Regulation (GDPR) [1]. This regulation enforces a Privacy by Design (PbD) approach, which means that companies are obligated to integrate privacy concerns into their design. Personal data is only to be processed when needed by the system.

In modern Android app development, developers have multiple choices on how to develop apps, e.g. native, web or cross-platform apps. To develop cross-platform apps, there exists different frameworks, e.g. Xamarin[1], React Native[2] and Unity[3] to make the development as smooth and easy as possible. Xamarin, acquired by Microsoft in 2016 [2], makes it possible to develop Android apps by using C#. In this report, we will continue the work from our project last semester[3] and focus on Xamarin apps. Android apps are most commonly distributed through Google Play Store[4], which contains around 3.6 million Android apps as of March 2018 [4], but there are no official numbers of how many of these apps are made with Xamarin.

Ferrara and Spoto [5] published a paper called "Static Analysis for GDPR Compliance" at ITASEC18, an Italian conference on cyber security. This paper raises awareness of how static program analysis can be used to check whether apps violate the GDPR. They point out that taint analysis can be used for checking if privacy leaks can occur. Many taint analysis tools for native Android apps already exist. However, to the best of our knowledge, such a tool does not exist for apps developed with .NET based cross-platform tools, such as Xamarin.

1.1 Initial Problem Statement

With the complexity of modern smartphone apps it is nearly impossible for a regular user to figure out what happens to the personal data given to an app, often on the premise that it is necessary for the app or service to function normally. Furthermore, the complexity also makes analyzing app binaries difficult for third parties, for instance researchers, to verify that a particular app does not leak personal information. This leads to the following initial problem statement:

> Apps have access to private data and the users do not know what happens to the data. How can a tool be created, to track data through a smartphone app made using Xamarin?

With the initial problem statement defined, the problem area can be analyzed, with the pur-

[1] `https://www.xamarin.com/`
[2] `https://facebook.github.io/react-native/`
[3] `https://unity3d.com/`
[4] `https://play.google.com/store`

pose of gaining knowledge about the problem. This knowledge will then lead to a final problem statement, which will be the foundation for the project.

Chapter 2

Problem Area Analysis

In this chapter, we analyze the problem area, including a description of the GDPR (Section 2.1), existing taint analysis tools for Android (Section 2.2) and a discussion of the potential target audiences of a taint analysis tool for Xamarin (Section 2.3). This analysis leads up to a final problem statement in Section 2.4, which forms the basis for the rest of the report.

2.1 General Data Protection Regulation

The General Data Protection Regulation (GDPR) is a regulation originally proposed by the European Commission in 2012 in order to strengthen and unify data protection for citizens of the EU. The regulation was adopted in 2016 and became enforceable on 25 May 2018 after a two year post-adoption grace period. [1]

2.1.1 Background

GDPR originates from the EU's single digital market strategy, which is intended to simplify the rules for companies inside the EU. The goal of the regulation is to strengthen citizens' rights, facilitate business and reduce administrative burdens. Specifically, the GDPR protects the personal data of EU citizens and the free movement of such data. Personal data is only allowed to be collected at the time when it is needed, and the data is required to be protected and only used for legal purposes. [1]

The GDPR replaces the existing data protection directive from 1995, implementing several important changes [6]. Among these changes are increased territorial scope, as the GDPR applies to all companies processing personal data of citizens of the EU, regardless of the location of the company. Other changes include an increase in the potential penalty for non-compliance and strengthened conditions for requesting consent from users. Additionally, the regulation offers more legislative power compared to the existing directive, since regulations are legally binding in their entirety, while directives simply set a goal that individual countries can decide how to achieve. [1]

One important goal of the regulation is to secure a number of obligations for companies that process personal data, and rights for the subjects of this data processing. It will become mandatory to notify customers in the case of a data breach within 72 hours, if the data breach poses a risk to the customers. Customers will have the right to know what personal data concerning them is being processed, where and for what purpose. Data controllers are also obligated to provide a copy of the data for free in an electronic format. Data subjects reserve the right to be forgotten, i.e. have the data controller erase personal data, stop any further processing of the data and stop any potential third parties' processing of the data. This right is invoked when the data is no longer relevant to its original purpose or if the data subject withdraws consent. Data subjects also have the right to receive any personal data in a commonly used machine readable format, and to transmit this data to another data controller. Lastly, the GDPR requires systems to be designed with data protection in mind, known as Privacy by Design (PbD). [1]

2.1.2 Privacy by Design

Privacy by Design (PbD) is not a new concept, but with the GDPR it is becoming a legal requirement for developing systems that they must include data protection by default, not as an addition. Data controllers will have to implement technical and organizational measures in order to sufficiently protect the rights of data subjects. This approach is characterized by proactive rather than reactive measures: privacy compromising events are anticipated and prevented before they happen. [7]

2.2 Existing Tools

In this section, two of the existing tools for performing static analysis on programs are examined. These tools are examined to find out if they contain features that could serve as inspiration for a new tool. The tools we examine are FlowDroid and Gendarme.

2.2.1 FlowDroid

FlowDroid is a "*context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android apps*", as described by one of the creators, Bodden [8].

What FlowDroid does differently than other taint analysis tools, is that it precisely models the Android lifecycle. The activity lifecycle in Android creates various entry points, e.g. with the use of asynchronously executing components and callbacks, which have to be taken into consideration when analyzing Android apps. From the app lifecycle information, FlowDroid creates a dummy main method, from which an inter-procedural control flow graph is generated and traversed to follow taint propagation. [9]

An example of this traversal can be seen in Figure 2.1. This depicts the combination of forward and on-demand backwards analysis, where every time a heap object is tainted, a backwards analysis is done to combat aliasing.

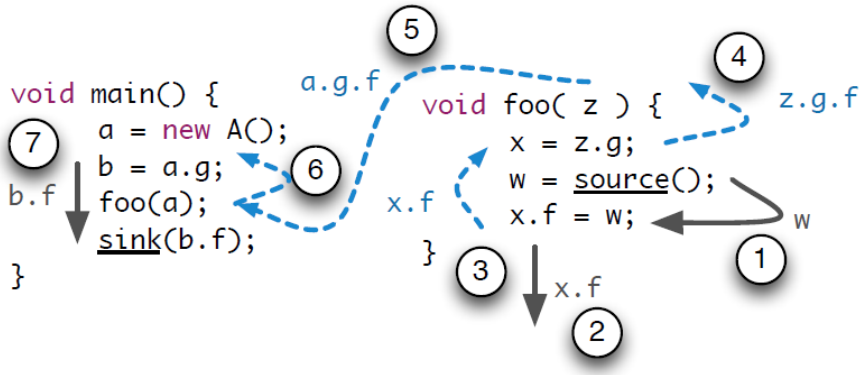

Figure 2.1: *Example of FlowDroid taint analysis in regard to aliasing. [10]*

The purpose of the analysis is to check if there is a connection between a *source* and a *sink*. Sources and sinks are usually defined in one of two different ways. The first is where the source indicates some kind of private data (e.g. a user's location) and the sink publishes this information (e.g. to a webserver). This is the definition used when checking for privacy leaks. However, taint analysis is also used for identifying vulnerabilities coming from unsanitized user input. In this case, the sink would be a vulnerable function, which could be a function for making SQL

calls. The source is then the user input that will be marked as *tainted* until it has been sanitized. Taint analysis is explored more in-depth in Section 3.6.

In order to start the analysis of an Android app with FlowDroid, it needs the app's Android Package Kit (APK) file, the Android software development kit (SDK) and a file with sources and sinks defined. A simple example is seen in Listing 2.1, where the latitude part of the user's location is defined as a source and `openConnection()` is defined as a sink.

```
<android.location.Location: double getLatitude()> -> _SOURCE_

<java.net.URL: java.net.URLConnection openConnection()> -> _SINK_
```

Listing 2.1: *Simple sources and sinks (FlowDroid)*

The result of FlowDroid's analysis is an XML file. The result of analyzing an app with the package name `aexyn.fake.mail.prank` is seen in Listing 2.2. This app was part of the dataset collected in our last semester project [3].

```
<?xml version="1.0" encoding="UTF-8"?>
<DataFlowResults FileFormatVersion="100">
   <Results>
      <Result>
         <Sink Statement="$r5 = virtualinvoke $r4.<java.net.URL:
             ↪ java.net.URLConnection openConnection()>()">
         <AccessPath Value="$r4" Type="java.net.URL"
             ↪ TaintSubFields="true">
            <Fields>
               <Field Value="<java.net.URL: java.lang.String ref>"
                   ↪ Type="java.lang.String" />
            </Fields>
         </AccessPath>
         </Sink>
         <Sources>
            <Source Statement="$d0 = virtualinvoke
                ↪ $r4.<android.location.Location: double getLatitude()>()">
            <AccessPath Value="$d0" Type="double" TaintSubFields="true"
                ↪ />
            </Source>
         </Sources>
      </Result>
   </Results>
</DataFlowResults>
```

Listing 2.2: *FlowDroid results from analyzing the app (XML)*

2.2.2 Gendarme

Where FlowDroid focuses on analyzing Dalvik byte code from the APK file, Gendarme instead performs its analyses on Common Intermediate Language (CIL) code using the Mono.Cecil[1] library to introspect it. Mono.Cecil is a library to generate and inspect CIL programs. The tool is intended to help developers write good code when programming. Gendarme is divided into a number of different categories seen in Table 2.1.

[1] `http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/`

Gendarme categories		
BadPractice	Concurrency	Correctness
Design	Design.Generic	Design.Linq
Exceptions	Interoperability	Maintainability
Naming	Performance	Portability
Security	Security.Cas	Serialization
Smells	Ui	

Table 2.1: *Categories in the Gendarme tool*

One example of a rule from the *BadPractice* category is the rule called *EqualsShouldHandleNullArgRule*. An example that will trigger this rule is seen in Listing 2.3.

```
1 public bool Equals (object obj)
2 {
3     // this could throw a NullReferenceException instead of returning false
4     return ToString ().Equals (obj.ToString ());
5 }
```

Listing 2.3: *Example of bad C# practice (C#)*

An obvious use for the Gerdarme tool is integration into a continuous integration (CI) environment where it is possible to warn developers about any common problems. This tool, however, is not designed for detecting if privacy leaks occur in apps. [11]

2.2.3 Investigating Android Apps

We decided to perform experiments using FlowDroid in order to determine if there exists a problem with Android apps leaking private information, thus possibly violating the GDPR. During the last semester project, we had downloaded a total of 995,009 apps from Google Play Store [3]. The taint analysis in FlowDroid is potentially very time-, processor- and memory intensive. This means that we are not able to scan all the apps, but instead decided to take them one by one alphabetically. In order to maximize the number of apps scanned, it was decided to have a timeout of 120 seconds, meaning that if the analysis is not finished after this time, it will save the results found and move on to the next app. Obviously, this means that we have to expect false negatives, since the analysis does not necessarily finish.

The sources and sinks configuration used in this test is very minimal, in order to reduce the number of false positives. This is done by only having the most obvious privacy leaks, such as submitting location, device ID etc., to a webserver. The full configuration in seen in Appendix A. The result of the test after running for 48 days is seen in Table 2.2.

Apps scanned	Apps flagged	Percentage
66,969	7,823	11.7 %

Table 2.2: *Results of running FlowDroid on 66,969 apps*

The results above only indicate potential data leaks in native Android apps, since no tool for performing taint analysis on Xamarin apps exists. However, we assume that the problem is also present in Xamarin apps.

2.2.4 Summary

FlowDroid analyzes the flow of data through Android apps with the definition of sources and sinks; it performs a so-called static taint analysis. Gendarme is a general analysis tool for code compiled to CIL and analyzes general code smells.

From a test of FlowDroid on 66,969 apps, 11.7% were found to have questionable behaviour, which could be classified as data leakage. Since no analyzer exists for Xamarin, precise numbers for Xamarin apps cannot be given, but we estimate that the number would be similar.

In the next section, the target audience of a analysis tool for Xamarin will be examined.

2.3 Target Audience

There are several potential target audiences for a tool for static analysis of Android apps developed using Xamarin. The choice of audience influences the tool, both in terms of which type of code can be scanned, and the detail and nature of the produced report.

If the tool works on source code, C# in the case of Xamarin apps, the potential audience is reduced, since the source code of apps is not generally accessible for anyone other than the developer, thus restricting the tool to app development. If the compiled CIL code is targeted instead, developers can still make use of the tool at the end of their toolchain, and other users who do not have access to the source code can use it as well.

The choice of audience affects what kind of information is acceptable in the report produced by the tool. Expert users will be able to distinguish between false positives and real positives, due to their understanding of the code. This applies both to the developers of the scanned tool, who have a high knowledge of the source code, and security experts who might not be familiar with the source code, but who still have the expertise to recognize a false positive. The category "security experts" contains researchers, security enthusiasts and other software developers who have not written the program themselves. Both developers and security experts will have the same goal in mind: to find data leakage which can lead to security or privacy issues. The major difference is how the tool is used. Developers will use the tool while developing the app, possibly in the context of CI or a testing flow. Therefore, the analysis tool could be implemented as a plugin for a CI service such as Jenkins[1] or Travis[2]. This would allow the tool to help developers, while being used relatively late in their tool chain, after the code has been compiled.

A security expert may use the tool to research a particular app or set of apps, after they have been developed, without having access to the source code.

Novice users, which are users with limited knowledge about the technical aspects of Xamarin apps and CIL code, will have to rely on the report to a greater extent, unable to recognize a false positive. This could result in the user distrusting an app that is actually secure, due to a false positive being reported by the tool. This distrust could be alleviated by giving the analysis tool in a graphical frontend, maybe as a web application, which explains the found results in depth, similar to the general analysis tool Ostorlab[3].

A way of using the tool, especially on large collection of apps, could be to have a coarse analysis mode, where the tool scans the apps faster, but potentially gives more false positives. Then,

[1] `https://jenkins.io/`
[2] `https://travis-ci.org/`
[3] `https://www.ostorlab.co/`

when some interesting apps are found with the coarse analysis, a finer, but slower analysis can be performed on the selected apps.

2.3.1 Conclusion

We choose to target expert users, including developers and security experts, since it is expected that they will be able to make sane judgments about the results of the scan. To make the tool more widely applicable, we choose to analyze the CIL code instead of targeting the C# code used to write Xamarin apps. The tool should also be able to have a coarse but fast mode, and a fine but slow mode, to make it possible to make a fast analysis on a large number of apps, and then select a few to make a more thorough analysis. This project is focused on the analysis tool itself, and any graphical frontend is out of scope and therefore we do not target novice users. The targeting of novice users could be considered further down the road when the analysis tool have been developed and have matured.

2.4 Problem Statement

Since the introduction of GDPR, which is described in Section 2.1, there is even more focus on handling and securing private data. Apps often deal with private data, e.g. location data, photos and videos, and therefore it is relevant to investigate how private data is handled in these apps.

In the initial problem statement in Section 1.1, the focus on tracking of data through a Xamarin app was chosen. This led to the investigation of tools, which can track data through apps on Section 2.2, where it was discovered that no such tool exists for Xamarin apps. Furthermore, it was discovered that around 11.7% of 66,969 Android apps scanned with FlowDroid were flagged as having potential data leaks, which is documented in Section 2.2.3.

By using FlowDroid and Gendarme, we have developed an idea of what type of tool will have to be developed to analyze Android apps made using Xamarin.

This leads to the following problem statement, which will be the basis of the rest of this report.

> How can a static taint analysis tool be constructed to examine the flow of data in apps developed using Xamarin?

Chapter 3

Theory

This chapter will highlight important domain knowledge and theory that form the basis of this project. First, we examine relevant aspects of the Android and Xamarin platforms in Section 3.1 and Section 3.2, including Android APK files, components and activity lifecycle, and the Xamarin technology stack. This is followed by a summary of CIL, which is the intermediate language Xamarin compiles to, and therefore the language which will be the subject of the analysis. Next, we get into the theoretical foundation of the taint analysis tool: control-flow analysis (Section 3.4), static single assignment (Section 3.5) and the taint analysis itself (Section 3.6). Finally, we present the requirements for a product to solve the problem of performing taint analysis on Xamarin apps, which are based on the problem area analysis in Chapter 2 and the theory in this chapter.

3.1 Android

Android is the most popular operating system for smartphones. [12] Most of the functionality in Android is provided by apps, often downloaded from Google Play Store.

3.1.1 Android APK Files

The apps in Android are packed in so-called APK files. An APK file is a ZIP archive with the APK file extention, thus it can be extracted with programs like unzip[1]. The APK file consists of different folders and files, and the app developer can also define files and folders. The standard folders are `META-INF`, `lib`, `res` and `assets` and the standard files in the APK file is `AndroidManifest.xml` and `classes.dex`. [13] When an app is made with Xamarin, an additional folder called `assemblies` is created. [14]

`META-INF`
: The `META-INF` folder contains the certificate files for the app. These files are used for signing the whole app, which prevents any modifications without having to create a new signing of the app. Android apps can be signed using JAR-signing, which is the old method, and APK Signature Scheme v2, which is the new and most secure way of signing Android apps. [15][16]

`lib`
: If any native binaries are required by the app, they are placed in the `lib` folder. These native binaries are processor specific, and are categorized into application binary interfaces (ABIs) which are divided by CPU and instruction set. The different ABIs are: `armeabi`, `armeabi-v7a`, `arm64-v8a`, `x86` and `x86_64` [17]. For instance, the Mono runtime in Xamarin apps resides in the `lib` folder.

`res`
: Any resources the Android app needs, for instance layout definitions, images or string

[1] `http://infozip.sourceforge.net/UnZip.html`

values, are found in the `res` folder. These resources are accessed with a unique resource ID.

`assets`
: Assets are files, like text, XML, music, video etc., which are needed by the app. The difference between files in the `res` folder and the `assets` folder is that with assets, the app has access to the raw data, where resources are processed by Android's resource system. [18]

`assemblies`
: The `assemblies` folder is not present in a native Android app, but is present in apps made with Xamarin. The `assemblies` folder contains the dynamic-link library (DLL) files which contain all the logic for the Xamarin app.

`classes.dex`
: `classes.dex` is a file in the root of the APK file that contains the source code of the app. In the build process of the app, all the Java classes and Java archive (JAR) libraries are compiled to Dalvik Executable format and stored in the `classes.dex` file. [19]

`AndroidManifest.xml`
: The `AndroidManifest.xml` describes essential information about the app to the Android build tools, Android operating system and Google Play. Android components, permissions, package name and version code among other things are declared in the `AndroidManifest.xml` file. [20]

3.1.2 Android Components

An Android app consists of component, which act as entry points to the app for systems or users, and each serve a specific purpose and use case. There are four different types components:

Activities
: The entry point for a user to interact with an Android app is through activities. An activity represents a single screen or user interface in an app, and different screens in an app are separate activities.

Services
: A more general entry point in an Android app is the service component. A service runs in the background to perform long-running operations, for example playing music in the background when another app is running. The service component does not provide a user interface as the activity does.

Broadcast receivers
: A broadcast receiver enables the delivery of events to an app outside of the regular user flow. For instance, when the app is not running, it will still be possible to deliver broadcasts to it via broadcast receivers. Often, the Android system sends broadcasts, for instance that the battery level is low or that the screen has been turned off. A broadcast is delivered as an intent.

Content providers
: The management of persistent storage in an app is done by content providers. The persistent storage can be an SQLite database, file system etc. Content providers can also give access for other apps to query or modify the data.

All the components in an Android app have to be declared in the `AndroidManifest.xml` file.

In the manifest file, the permissions for the components, specifically regarding inter-process communication (IPC) are also set. [21]

Android IPC

Normally when apps are running in Android, each app is completely separated from each other. To achieve IPC between Android apps, a construct called *intents* can be used. An intent is an abstract description of an operation to be performed, and can be used together with activities, broadcast receivers and services. An intent consists of an action and the data the action should be performed on. [22]

There exists two types of intents: explicit and implicit intents. An explicit intent specifies which component it targets. Implicit intents do not name a specific component and relies on the Android system to find the appropriate component. [23]

3.1.3 The Activity Lifecycle

The activity in Android is an essential part of an app that is created every time an app is used and destroyed when the app is put in the background, killed, crashes etc. The lifecycle of an app consists of six core stages, as seen in Figure 3.1 and is implemented using callbacks. The six callbacks are: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`. [24]

`onCreate()`
: The `onCreate()` callback is the first to be called when the system creates the activity. Therefore, `onCreate()` is mandatory to implement. `onCreate()` handles the startup logic: things which should happen once in the lifetime of the activity, for instance associating the activity with a view model. When the `onCreate()` method finishes executing, the activity is in the *started* state, and `onStart()` and `onResume()` are called in quick succession.

`onStart()`
: `onStart()` is invoked when the app enters the *started* state and makes the activity visible to the user. The activity is prepared to go to the foreground and among other things, the code to maintain the UI is executed. Furthermore, any lifecycle-aware components tied to the activity receives the `ON_START` event. When `onStart()` finishes, the app enters the *resumed* state, where `onResume()` is called.

`onResume()`
: When the activity enters the *resumed* state, the activity comes to the foreground and `onResume()` is invoked. The app stays in the *resumed* state until something happens to take focus away, e.g. if the screen is turned off or a phone call is received. This means that the *resumed* state is where the user interacts with the activity. Lifecycle-aware components receive the `ON_RESUME` event when the *resumed* state is entered.

  When an activity returns to the *resumed* state from *paused*, `onResume()` is called again and therefore `onResume()` should reinitialize components released during `onPause()`.

`onPause()`
: `onResume()` and `onPause()` are closely tied together, since when an interrupting event occurs, the activity enters the *paused* state, and `onPause()` is called. `onPause()` is also used to release system resources, for instance GPS handle, which can affect battery life. An activity can still be visible to the user and in the *paused* state.

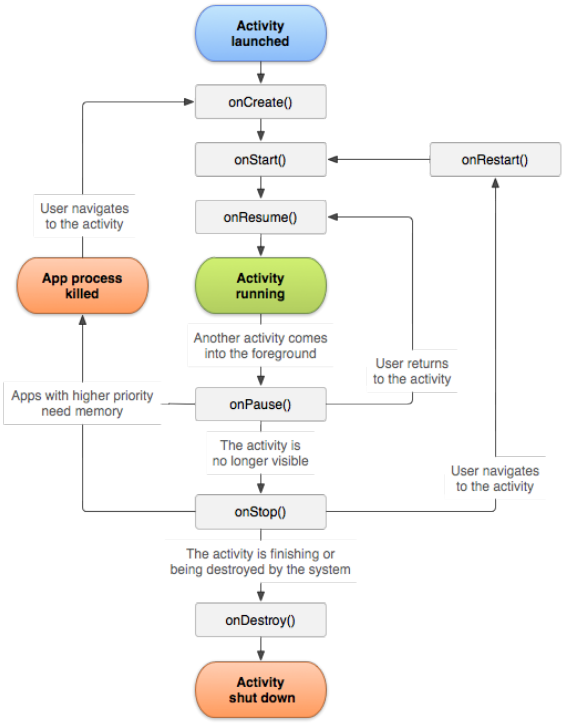

Figure 3.1: *An overview of the activity lifecycle, from [24]*

`onStop()`
: The *stopped* state is entered when the activity is no longer visible to the user, and the `onStop()` callback is invoked. This happens for instance when a new activity that covers the entire screen is opened, or when the activity is finished running and about to terminate. Therefore `onStop()` must release resources that are not needed when the app is not on the screen.

`onDestroy()`
: When the activity is destroyed, the `onDestroy()` callback is invoked. This happens when the activity is finishing or upon a configuration change, for instance if the device is rotated or put in multi-window mode. If `onDestroy()` is called because of a configuration change, `onCreate()` is called immediately when the new activity is created and the lifecycle of the activity restarts. Since `onDestroy()` is the final step in the lifecycle, is should release all resources that have not already been released.

3.2 Xamarin

Xamarin is, as mentioned in Chapter 1, a framework that makes it possible for app developers to code cross-platform apps in C#. Xamarin is build on top of Mono[1], which is an open-source version of the .NET framework. The Android part of Xamarin is called Xamarin.Android, referred to as "Xamarin" from now on, runs side-by-side with the Android Runtime (ART). The C# code gets compiled to CIL code and packed in an APK file. When the app is launched, the CIL code is just-in-time (JIT) compiled to native assembly code, so it can run on the device. Xamarin uses a runtime, which is responsible for handling memory allocation, garbage collection, platform interop etc. [25]

[1] `http://www.mono-project.com/`

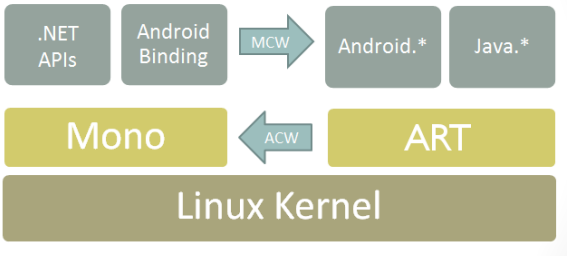

Figure 3.2: *An overview of the architecture of Xamarin [28]*

Xamarin supports different ways of packing the APK file, which is dependent on the way the app is compiled by Xamarin. Xamarin supports JIT compilation as the default setting, where all the assemblies of the app are placed in a folder named `assemblies`, as mentioned in Section 3.1.1, where the assembly files are in plain DLL format [26]. Xamarin also supports experimental ahead-of-time (AOT) compilation, and bundling assemblies together in a platform specific shared object (SO) file [27].

3.2.1 Xamarin Technology Stack

In this section, the Xamarin technology stack is explained, for the purpose of understanding how apps are loaded, and how the apps work together with the Xamarin environment.

Xamarin and ART runs on top of the Linux kernel, which exposes application programming interfaces (APIs) to the underlying systems, e.g. telephony and graphics systems.

Figure 3.2 shows the general architecture of the Xamarin system. Here the kernel, Mono and ART are depicted together with the different methods used to get Mono and Android to run together: managed callable wrappers (MCWs) and Android callable wrappers (ACWs).

Managed Callable Wrappers

To call Java classes from C#, MCWs are used. A managed callable wrapper is a Java Native Interface (JNI) bridge, which is used every time managed code[1] needs to invoke Java code from Android. This is for instance the conversion of types and invoking the underlying Android platform methods. As an example, the whole `Android.*` namespace is used in C# code through MCWs, which can be generated via JAR binding or implemented manually via Java.Interop, which supports invoking specific Java methods using the JNI.

Android Callable Wrappers

For Android to be able to call managed code, ACWs are used. The reason that ACWs are required, is that there is no way of registering classes at runtime with ART, since the JNI class `DefineClass()` is not supported. Xamarin provides a *Java proxy* for when Android needs to execute a virtual or interface method, which is overridden or implemented in managed code. A Java proxy is Java code that has the same base class and Java interface list as the managed type, and implements the same constructors and declared overrides. ACWs are generated for all types that inherit from `Java.Lang.Object` during the build process by `monodroid.exe`. [30]

[1] Managed code is code from programming languages using the Microsoft .NET Framework. [29]

App Startup

When the user starts an activity, service etc., Android checks if a process to host the activity is already running. If not, a new process is created, and the `AndroidManifest.xml` is read to find the attribute `application/android:name`, which references a Java class that inherits from `android.app.Application`, which is loaded and instantiated. Afterwards, all the types specified by the attribute `application/provider/android:name` are instantiated and their `ContentProvider.attachInfo` method is invoked. The Mono run-time automatically creates a run-time provider to the `AndroidManifest.xml` at run time. The Mono run-time is therefore initialized in the method `mono.MonoRuntimeProvider.attachInfo`, which loads the Mono runtime into the process [28]. Afterwards, the `mono.MonoPackageManger.LoadApplication` starts to load the app, set various parts etc. `mono.MonoPackageManger.LoadApplication` also calls `Mono.Android.Runtime.init` which calls into the Xamarin runtime that is packaged in shared object (so) files.

When the runtime is initialized, the `AndroidManifest.xml` is used to start the relevant activities and services. An example is to use the `application/activity/android:name` attribute to determine the name of the activity to load. The method `Class.forName` loads the activity type, but requires a Java type, which triggers the creation of an ACW which invokes the corresponding managed type (here C#). Then, Android invokes `Activity.onCreate(bundle)` which causes the corresponding C# version of `Activity.onCreate(bundle)` to be executed, and the user activity is displayed on the screen.

A diagram of the app startup procedure can be seen in Figure 3.3.

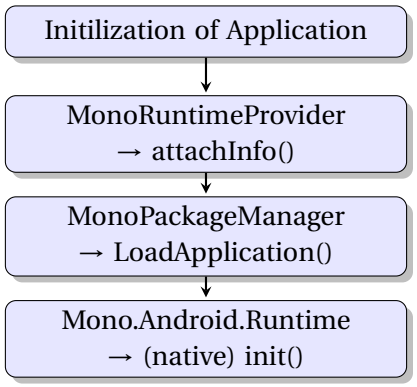

Figure 3.3: *Initialization of Mono runtime [3]*

3.3 Common Intermediate Language

Common Intermediate Language (CIL) is the intermediate representation which all languages following the Common Language Infrastructure (CLI) specification compile to, including Xamarin. This allows for increased portability, as the CIL bytecode can be executed in any CLI compliant run-time system.

CIL is an object-oriented, stack-based assembly language, consisting of a number of directives, attributes and opcodes, that can be combined to make a program. The opcodes are the actual instructions of the language, where directives are used to describe the structure of a program and attributes add information to directives. These opcodes are short mnemonics for the actual opcodes that are used by the compiler, which are byte-sized binary numbers.

Being an object-oriented language, CIL follows principles such as polymorphism. This means, among other things, that methods can be overloaded, and method calls to an overloaded method are resolved using dynamic dispatch at run time. This process can involve looking up the method in the class hierarchy, in order to choose which one should be invoked. This affects the way compilers such as the Xamarin compiler choose to structure their output code, and dynamic dispatch is therefore important to keep in mind when analyzing CIL code.

For more information about CIL in relation to Xamarin, see our report from last semester [3].

3.4 Control Flow Analysis

A control flow analysis (CFA) is an analysis of the flow of control in a program. A flow graph is used to represent the program under analysis, where each node in the flow graph represents a location in the program. The edges in the flow graph represent each possible transition between the program locations.

Fischer, Cytron, and LeBlanc [31] defines a flow graph formally as:

> **Definition 14.1** A flow graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, root)$ is a directed graph: $\mathcal{N}$ is a set (of nodes) and $\mathcal{E}$ is a binary relation on $\mathcal{N}$. The root node is the distinguished entry node of the flow graph: $\forall X \in \mathcal{N}, (X, root) \notin \mathcal{E}$.

A CFA is a static analysis, which means that the analysis is performed on the code without actually executing the program.

We will focus on two types of flow graphs: control flow graph (CFG) and procedure call graph.

3.4.1 Control Flow Graph

To represent the potential execution paths through a procedure, a CFG $\mathcal{G}_{cf}$ is used. Usually a node $n \in \mathcal{N}$ represents a sequence of operations. An edge $e \in \mathcal{E}$ represents a potential execution path through the sequences in the procedure. This makes CFGs useful in intraprocedural analyses, which are analyses within a single procedure.

As mentioned, a node represents a sequence of operations. The definition of a sequence depends on the use case. It can be a single instruction or the whole procedure. It depends on which level of granularity is needed for the analysis, but with finer granularity comes more nodes, which can lead to inefficiency.

Fischer, Cytron, and LeBlanc [31] mentions three popular approaches to choosing a level of granularity of the nodes:

- Associate each node with each statement, since programmers often make procedures which creates meaningful changes to the program state.
- For language-independent optimization, associate each node with each statement or instruction of the intermediate language created by the compiler.
- To alleviate a too fine granularity and improve space efficiency, instructions can be grouped into basic blocks. These basic blocks contain the longest sequence of operations that only have one successor, i.e. do not change the program flow like an `if` or `goto`. An example of a program partitioned into basic blocks can be seen in Figure 3.4, where a new block is made when the `if`-statements are reached. Basic blocks also have some downsides: With sparse flow graphs, very little memory is consumed, minimizing the gain with

using basic blocks even at lower granularities. Furthermore, upon node visit, each basic block has to be "opened up", which often results in almost no time saved. [31]

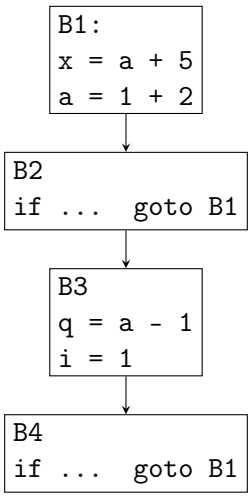

Figure 3.4: *Example of basic blocks.*

3.4.2 Procedure Call Graph

To represent the potential execution between the procedures in a program, a procedure call graph is used. In the procedure call graph, the nodes represent the procedures in the program, and the edges represent the potential procedure calls. Therefore, procedure call graphs are used in interprocedural analysis, which is the analysis of the calling relations between different procedures.

When creating a procedure call graph, dynamic dispatch, which is used in languages such as C# and CIL, has to be taken into consideration. This is the case since virtually dispatched methods are resolved at runtime, thus hard to predict statically. In some cases with higher-order functions or procedure variables, complicated techniques are required to approximate the procedure call graph. Furthermore, a language construct called reflection further complicates the creation of a procedure call graph. Reflection makes it possible to examine, introspect and change behaviour during runtime. An example of this could be calling methods without knowing the method name at compile time, as seen in Listing 3.1, where the method `doSomething` is attempted to be called.

```
1 Method method = foo.getClass().getMethod("doSomething", null);
2 method.invoke(foo, null);
```

Listing 3.1: *Example of reflection [32] (Java)*

An example of a procedure call graph can be seen on Figure 3.5. Here P calls Q and R and Q and R call each other. But this procedure call graph does not show the transfer of control from R to P, when R returns. This would be including in a supergraph, where there could be an edge from R and Q to P, to show the return of control. [31]

```
with invocation i of method m of class c
    set X → all non-abstract subclasses of c
    if c is an interface
        X ∪ all implementing classes
    foreach class x in X
        if x implements method m_x with same signature as m
            m_x is accepted as a candidate
        else
            traverse class hierarchy upwards from x and the first
            compatible method implemented in class extended by x
            is considered a target of i
```

Figure 3.6: *Pseudo code of the dynamic dispatch algorithm used in [33]*

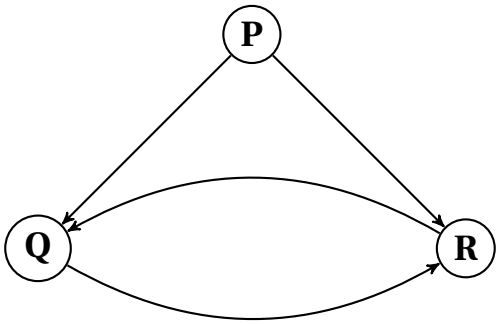

Figure 3.5: *Example of a procedure call graph from [31, p. 559]*

3.4.3 Dynamic Dispatch

As mentioned, dynamic dispatch complicates the generation of a call graph, since the exact methods called are determined at runtime. Poeplau, Fratantonio, Bianchi, *et al.* [33] deals with this problem for Dalvik bytecode by performing a class hierarchy analysis (CHA), where their end-goal is to statically detect dynamic dispatch in Dalvik bytecode from Android apps. Using the CHA, they construct a super control flow graph (sCFG), which is a graph that represents possible invocations between the different methods in the program. This means that an edge in the sCFG is the potential call of a method, which redirects the control flow.

The algorithm they implement creates an over-approximation of the sCFG, which means that there are connections between calls and entry points, which may never be used during runtime. This algorithm assumes that a CHA is performed and that each method has a CFG built.

The algorithm to resolve dynamic dispatch is written as pseudo code in Figure 3.6. With invocation i of method m in class c, all non-abstract subclasses of c are found. These subclasses are found using the class hierarchy. All non-abstract subclasses of class c, together with all implementing classes if c is an interface, are put in the set X. Then, for all x classes in X, if x implements a method m_x, which has the same signature as m, which happens in the case that x overrides m, a connection between i and m_x is made. If no x implements a method with same signature as m, the class hierarchy is traversed upwards starting with class x, to find the first compatible method that is implemented in classes extended by x. This method is connected to i.

3.5 Static Single Assignment

The control flow can take many paths through a program, including loops, jumps and branching. When multiple program paths meet after having branched, a flow-aware analysis must be able to resolve which one of the multiple program states is going to be used going forward. One way of solving this problem is by transforming the program to a form called static single assignment (SSA). This form requires that each variable is defined before it is used, and that each variable is assigned exactly once. This necessitates a transformation of the program, adding new variables to account for any reassignments. This gives the benefit that an assignment such as $a \leftarrow b+1$ is mathematically true: after this assignment, a is mathematically equal to $b+1$ for the rest of the program's execution. This means that a can be substituted for $b+1$ throughout the program, reducing the amount of information the analysis must keep track of at any given point in the program. This section is based on the works of Fischer, Cytron, and LeBlanc [31] and Chong [34].

This transformation process has two phases. First, ϕ *functions* are inserted into the program. These are functions that create the new variables, and they take a number of instances of a single variable as parameters, and return a new instance of the variable to be used going forward. The value of this new variable instance is determined by the flow through the program leading up to the ϕ function. The second phase consists of renaming variables at both the original definition sites, as well as the newly inserted definitions in ϕ functions.

The ϕ functions could be placed anywhere in the program, but they are not necessary at most program points. A ϕ function at a node with only one incoming edge has no effect, since the variable value can simply pass through, and while having more than one incoming edge is necessary, it is not sufficient to determine that a node needs a ϕ function. At least two distinct names need to meet at a node for a ϕ function to be necessary. What we want is an efficient process of inserting ϕ functions only where they are needed, which can be done using dominance.

3.5.1 Dominance

Dominance in a CFG is a set of useful abstractions over flow graphs. The *dominators* of a node are all the nodes that control flow must pass through in order to reach the node. For a control flow graph $\mathcal{G}_f = (\mathcal{N}_f, \mathcal{E}_f, root)$ the following concepts are defined:

- Node Y *dominates* node Z, denoted $Y \underline{\gg} Z$, iff every path in $\mathcal{G}_f$ from $root$ to Z includes node Y. A node always dominates itself.
- Node Y *strictly dominates* Z, denoted $Y \gg Z$, iff $Y \underline{\gg} Z$ and $Y \neq Z$.
- The *immediate dominator* of node Z, denoted $idom(Z)$, is the closest strict dominator of Z:
$$Y = idom(Z) \iff (Y \gg Z \text{ and } \forall X \gg Z,\ X \underline{\gg} Y)$$
- The *dominator tree* for $\mathcal{G}_f$ has nodes $\mathcal{N}_f$; Y is a parent of Z in this tree iff $Y = idom(Z)$.

One method for determining dominators in a flow graph is based on the observation that a node N is dominated by itself and by any node that dominates all of N's predecessors. This observation can be used to create a simple algorithm that finds all dominators of each node in a flow graph.

For the purpose of determining where to place ϕ functions, we need to find *dominance frontiers* for the nodes in the graph. These are the nodes that are just one edge outside of the node's

dominators, formally defined as:

- $DF(X)$ is the set of nodes Z such that X dominates a predecessor Y of Z but does not strictly dominate Z:

$$DF(X) = \{\ Z \mid (\ \exists (Y,Z) \in \mathcal{E}_f\ )(\ X \underline{\geq\!\!\geq} Y \text{ and } X \not\gg Z\ )\}$$

If a definition of a variable happens at a program point corresponding to node X in the program's flow graph, then $DF(X)$ is the set of nodes where this definition will meet other definitions. This is the exact set of nodes where ϕ functions need to be inserted, and we can now proceed to rename every variable definition so that they are all unique.

3.5.2 Renaming

Every variable definition in the program needs to have a unique name, so that the analysis can determine unambiguously which exact value is used at any given point. In order to do this, we need to determine uniquely the definition site that reaches a given use of a variable name, which can be done based on the program's CFG and dominator tree.

In the program, a variable name v may be appear in original uses in the program that existed before the SSA transformation, and uses in ϕ functions inserted into the program. In the first case, the use is reached by the definition of v that most closely dominates that use. In the second case, the use is reached by the definition from an incoming edge that can be found by examining the ϕ function's parameters. The definition of dominance frontiers tells us that if a ϕ function appears at some node Z, then Z is in the dominance frontier of a node X, and X must dominate some predecessor Y of Z. The definition of v that reaches Y is the definition that uses the edge (Y,Z) into the ϕ function at Z. Therefore the renaming algorithm can check the successors of Y for ϕ functions and forward the appropriate name for v.

After each definition site has been renamed, the transformation to SSA form is complete.

3.6 Taint Analysis

Many modern software projects use and produce large amounts of data that flows through the program. Determining where data enters the program and where it ends up can be hard to follow. Flow analysis is the analysis of this flow of data through a program. Taint analysis is a specialization of flow analysis that examines data flow from *sources* to *sinks*. Taint analysis can be static or dynamic; in the rest of this report, we will focus on static taint analysis.

The flow of data through a program is described by Denning and Denning [35] as:

> Information flows from object x to object y, denoted $x \Rightarrow y$, whenever information stored in x is transferred to, or used to derive information transferred to, object y.

Data that flows through the program has a source. This is, as the name suggests, where the data enters the program and can be anything from API calls to user input.

If the source is untrustworthy or of special interest, e.g. if the source loads private data into the program, then the data from that source is *tainted*. To keep track of tainted data, a label or tag is used to identify it, which makes it possible to follow it throughout the program, and see what other data is influenced by the tainted data.

Other data can be influenced by tainted data, in which case the influenced data is also tainted, which is called taint propagation. This taint propagation can be written as: $x \Rightarrow t(y)$, where t is

the taint operator, such that object x taints object y. The taint operator is transitive: If $x \Rightarrow t(y)$ and $y \Rightarrow t(z)$, then $x \Rightarrow t(z)$. [36]

3.6.1 Performing Taint Analysis

If the source code is not available, it is possible to perform static taint analysis on the assembly code of the program. The taint analysis must, for each instruction, identify all the operands and the operand types. In assembly languages, the operand type could be source or destination. For more advanced languages, regular types such as `bool`, `int`, `float` etc., could be something the taint analysis would have to identify. To be able for the taint analysis to track the tainted data, the taint analysis must understand the semantics of each instruction. This means that the taint analysis must understand what each instruction does, and from that make conclusions on the status of tainted data, when the instruction has executed. An example could be the `x86` instruction `PUSH`, where `PUSH EAX` pushes the value of `EAX` on the stack. What must be remembered is that `PUSH` does some implicit things, where the `ESP` register is decremented with the operand size and the stack segment register is changed. [36] Furthermore, some instructions exist in different variants with similar semantics. What is different is often the type of the parameter(s) or the size of the data they perform the operation on.

All these details of the instructions must be taken into consideration when a taint analysis is performed. To combat the complexity and redundancy of assembly languages, an intermediate language is often used. An example of an intermediate language for `x86` is the language Reverse Engineering Intermediate Language (REIL) which only has 17 instructions[37], thus having fewer instruction to handle in the taint analysis.

When performing a taint analysis, a CFG can be used to traverse the program in a controlled manner. For instance, Arzt, Rasthofer, Fritz, *et al.* [9] use an inter-procedural control flow graph (ICFG), which they loop through and track the taints of the program. Using a form of CFG can help ease the traversal of a program to follow the taints, and should be considered important to the analysis of a program.

3.7 Requirements Elicitation

In this section, all the requirements to solve the problem statement are listed. These requirements are extracted from Chapter 2 and Chapter 3 and converted into concrete requirements for an analysis tool.

The requirements will be split into technical and usage requirements.

3.7.1 Technical Requirements

The technical requirements are about features needed for the analysis tool to work. These requirements are regarded as most important, since they directly influence how the analysis tool performs, and its effectiveness.

Parse and extract Xamarin APK files

As mentioned in Section 3.1.1, the Android APK format is a compressed file. This means that it first has to be extracted before the app can be processed.

Create and transform the CIL code to an intermediate format

As mentioned in Section 3.6, it simplifies the process of creating a taint analysis tool if the subject language, in our case CIL is converted to an intermediate language. A simplification could be to group similar instructions together to one.

Convert code to SSA form

As mentioned in Section 3.5, SSA can be used to keep track of the intraprocedural code flow, especially in regards to the confluence of values with the same name, as with for instance `if` statements. Here SSA can keep track of the different values and indicate that two values with the same name meet at this point in the program, and the analysis must determine which value is used going forward. Therefore, SSA form is required for the analysis tool.

Control Flow Analysis

A CFA is required for the analysis to be able to know about both the flow within and between methods, as mentioned in Section 3.6. FlowDroid follows this approach and create an ICFG, indicating that it is a viable solution to the problem.

In Section 3.4, two types of CFA were described: CFG and call graphs. We expect that both are needed to create an analysis of Xamarin apps. Therefore, the analysis tool needs to create both a CFG and call graph, and use that to make inter- and intraprocedural analysis of the apps.

Resolve dynamic dispatch

In Section 3.3, it was documented that CIL is an object oriented language which uses dynamic dispatch to resolve instance method calls at run time. In Section 3.4.3, it is stated that dynamic dispatch is required to be resolved to be able to make a precise analysis of CIL. The consequence of not resolving these instance method calls precisely, is that it will be impossible to know exactly which method is called, and therefore impossible to create a representative call graph. Therefore, it is a requirement for the analysis tool to resolve dynamic dispatch.

Take Android entry points and life cycle into consideration

The Android activity life cycle creates a lot of special entry points in a program with all its callbacks, which is described in Section 3.1.3 and also mentioned in Section 2.2.1. To create an accurate analysis of Xamarin apps, these entry points need to be taken into consideration when creating the CFA, since they create special entry points into the app.

3.7.2 Usage Requirements

The usage requirements are the requirements about how the tool is used by the user. These requirements are not regarded as critical as the technical requirements, since these determine how a user interacts which the tool, and not how the tool itself performs.

Automatic analysis

Automatic analysis means that the user does not need to interact with the analysis tool during the analysis. Therefore, when the user starts a scan, the user should not have to do anything before the scan is done. This also makes it possible to scan more than one app at the time.

Have a coarse, but quick scan mode

When using the analysis tool it could be useful, as mentioned in Section 2.3, to do quick scans over many apps. The result will be less precise, but it can give a hint to which apps to scan more thoroughly. Therefore, with this scan mode, more false positives can be tolerated, given that they are removed in a more thorough scan.

Have a slower, but thorough scan mode

As mentioned in the requirement just above, and in Section 2.3, a thorough scan mode is

needed for a qualitative assessment of an app. With this scan mode, the app is analyzed as thoroughly as possible, with as few false positives as possible. The consequence of the thoroughness is that the scan can be relatively long, hence the need for a quick scan.

It is expected that these requirements are needed to be fulfilled to create a thorough analysis of apps made using Xamarin. In the next chapter, the implementation of the analysis tool will be described.

Chapter 4

Implementation

This chapter details the implementation of the system based on the problem analysis in Chapter 2, the theory in Chapter 3 and the resulting requirements in Section 3.7.

We are going to present Simple CIL (SCIL), Simple CIL Analyzer (SCIL/A) and Flix Analyzer (Flix/A). SCIL is our take on an intermediate language for CIL with the purpose of performing taint analysis. SCIL will be formally described in Section 4.1 and Section 4.2 where its structure and semantics are presented. Section 4.3 presents the flow logic for the language, which is the basis of the taint analysis.

SCIL/A and Flix/A are the tools that perform the analysis on the SCIL language, so in union these tools perform the whole analysis on SCIL. An overview of SCIL/A is given in Section 4.4 where the overall process of analyzing an APK file is shown. The whole process through SCIL/A is then documented, and in Section 4.5 the analyzer is described. In Section 4.6, the part of the analysis which involves tracking the flow through the app is described, which is what Flix/A is responsible for.

The approach in this chapter is heavily inspired by Hansen [38] and occasionally inspired by Wognsen and Karlsen [39].

4.1 Program Structure

This section describes the structure of a program written in SCIL, which is a simplified version of the CIL language.

SCIL has a reduced set of instructions, which will make the taint analysis simpler. The core set of these instructions is listed in Table 4.1.

Instruction	::=	`nop`	No operation
	|	`push` x	Push x on the stack
	|	`pop`	Pop the top of the stack
	|	`dup`	Duplicates the top element
	|	`add`	Adds the two top elements
	|	`stloc` n	Pops and stores the top value in local variable n
	|	`ldloc` n	Pushes value from variable n
	|	`ldarg` x	Load argument number x onto the stack
	|	`brtrue` pc_1	Branch to pc_1 if top at stack is positive
	|	`new` σ	Create new instance of σ
	|	`call` m_t	Calls static method m_t
	|	`callvirt` m_t	Calls instance method m_t

Table 4.1: *Table of the SCIL instructions.*

4.1.1 Notation

A **domain** is defined to be a set with corresponding **access functions** used to modify components of the domain. The **record** notation is used to specify a domain with its access functions:

$$\text{Dom} = (f_1 : \text{Dom}_1) \times \cdots \times (f_n : \text{Dom}_\text{n})$$

Here the domain Dom is defined with access functions $f_i : \text{Dom} \to \text{Dom}_\text{i}$ for $1 \leq i \leq n$. Accessing an element f_i of $d \in \text{Dom}$ is written as $d.f_i$, and updating an element is written $d[f_i \mapsto v]$. This notation is extended from access functions to functions in general: $x.f$ for $f(x)$ and $x_1.f(x_2, \cdots, x_n)$ for $f(x_1, x_2, \cdots, x_n)$.

4.1.2 Type Definitions

These are the types from CIL chosen to be implemented in SCIL. CIL is a strongly typed language, and so is SCIL. Note that while CIL has many types of different sizes, these semantics do not model this, and therefore do not distinguish between these types.

$$\begin{aligned}
\text{Type} &::= \text{ValType} \mid \text{RefType} \\
\text{ValType} &::= \texttt{int} \mid \texttt{float} \mid \texttt{boolean} \\
\text{RefType} &::= \text{ClassName} \mid \text{InterfaceName} \\
\text{ReturnType} &::= \text{Type} \mid \texttt{void} \\
\text{MethodType} &::= \text{Type}^* \ \to \ \text{ReturnType}
\end{aligned}$$

4.1.3 Program Components

A program is defined as a set of namespaces.

$$\text{Program} = \ (namespaces : \mathcal{P}(\text{Namespace}))$$

Each namespace is identified by a name, and contains the set of classes and interfaces defined in it.

$$\begin{aligned}
\text{Namespace} = (&name : \text{NamespaceName}) \times \\
&(classes : \mathcal{P}(\text{Class})) \times \\
&(interfaces : \mathcal{P}(\text{Interface})) \times
\end{aligned}$$

A class is defined by its name, the namespace it belongs to and its base class. The methods and fields defined in a class are accessed through the *methods* and *fields* components, respectively. Additionally, a class may implement a number of interfaces, which are accessed through the *implements* access function.

$$\begin{aligned}\text{Class} = {} & (name : \text{ClassName}) \times \\ & (namespace : \text{Namespace}) \times \\ & (base : \text{Class}_{\perp}) \times \\ & (methods : \mathcal{P}(\text{Method})) \\ & (fields : \mathcal{P}(\text{Field})) \\ & (implements : \mathcal{P}(\text{Interface}))\end{aligned}$$

Interfaces consist of an identifying name, the namespace they belong to and its base interface. They also contain a set of abstract methods that contain no instructions or fields. Finally, they contain a set of references to all the classes that implement the interface. Any class that implements an interface must provide implementations for all the methods in the interface.

$$\begin{aligned}\text{Interface} = {} & (name : \text{InterfaceName}) \times \\ & (namespace : \text{Namespace}) \times \\ & (base : \mathcal{P}(\text{Interface})) \times \\ & (methods : \mathcal{P}(\text{Method})) \times \\ & (fields : \mathcal{P}(\text{Field})) \times \\ & (implementedBy : \mathcal{P}(\text{Class}))\end{aligned}$$

Methods are declared in a class or an interface, and are identified with a method name. They also have a declared return type. The instructions that implement the method are accessed with the function *instructionAt*, which takes a program counter and returns the instruction at that program counter. A method can either be static or an instance method, as indicated by the boolean value accessed through *isStatic*.

$$\begin{aligned}\text{Method} = {} & (class : \text{Class} \cup \text{Interface}) \times \\ & (name : \text{MethodName}) \times \\ & (type : \text{MethodType}) \times \\ & (instructionAt : \text{PC} \to \text{Instruction}) \times \\ & (isStatic : \text{Bool})\end{aligned}$$

Fields are components of classes or interfaces and are identified with a field name and a type. Like methods, they also have a boolean value that indicates whether the field is static.

$$\begin{aligned}\text{Field} = {} & (class : \text{Class} \cup \text{Interface}) \times \\ & (name : \text{FieldName}) \times \\ & (type : \text{Type}) \times \\ & (isStatic : \text{Bool})\end{aligned}$$

4.2 Operational Semantics

In this section, the operational semantics of SCIL are defined. First, the semantic domains are defined and described. Then the semantic rules are defined and split into three categories: the core instructions, object instantiation, and method invocation.

4.2.1 Domain Definitions

Values are either numbers or references to objects. Numbers are simply integers, since modelling the details of different types of numbers is not in our interest. Object references are specific instructions in the program, identified uniquely with a class and an address. Object references can also be null references.

$$\begin{aligned}\text{Val} &= \text{Num} + \text{ObjRef} \\ \text{Num} &= \mathbb{Z} \\ \text{ObjRef} &= \text{Class} + \text{Addr} \cup \{\text{null}\}\end{aligned}$$

The program counter can be any natural number including zero. Programs are assumed to be normalized so the program counter begins at zero at the start of every method, and that the instruction following program counter pc is found at $pc + 1$.

$$\text{PC} = \mathbb{N}_0$$

Addresses are defined as a method and a program counter, and are used to uniquely identify an instruction in a program.

$$\text{Addr} = \text{Method} \times \text{PC}$$

The heap is a map from object references to objects, and an object is defined by its class and the value of its instance fields. The notation $o.f$ is used to refer to an object o with a field $f \in dom(o.fieldValue)$, as a shorthand for $o.fieldValue(f)$.

$$\begin{aligned}\text{Heap} &= \text{ObjRef} \rightarrow \text{Object} \\ \text{Object} &= (class : \text{Class}) \times \\ & \quad (fieldValue : \text{Field} \rightarrow \text{Val})\end{aligned}$$

Static fields are contained in the static heap, uniquely identified by their name, type and the class where they are defined. Therefore the static heap can be implemented simply as a map from fields to values.

$$\text{StatHeap} = \text{Field} \rightarrow \text{Val}$$

Each method has an operand stack, which is implemented as a sequence of values. The left-most element of the stack is the top element, so the stack grows to the left. The length of a stack $S = (v_0 :: \cdots :: v_i :: \cdots :: v_n)$ is written as $|S| = n+1$. The notation $v_0 :: \cdots :: v_n$ is used as shorthand for $[0 \mapsto v_0, \ldots, n \mapsto v_n]$.

$$\text{Stack} = \text{Val}^*$$

Each method has local variables, which are stored in a local heap and indexed with a number. The variable with index zero contains a reference to the object where the method is invoked.

Methods also have a list of the arguments used when invoking the method, which are indexed in the same way.

$$\text{LocVar} = \mathbb{N}_0 \rightarrow \text{Val}$$
$$\text{ArgList} = \mathbb{N}_0 \rightarrow \text{Val}$$

4.2.2 Program Configurations

A stack frame consists of the currently executing method, the program counter, the local variables, the method's arguments and the operand stack

$$\text{Frame} = \text{Method} \times \text{PC} \times \text{LocVar} \times \text{ArgList} \times \text{Stack}$$

The call stack is then a sequence of these stack frames. The call stack uses the same notation as operand stacks.

$$\text{CallStack} = \text{Frame}^*$$

Finally, a semantic configuration of the operational semantics is defined as a heap, a static heap and a call stack.

$$\text{Configuration} = \text{Heap} \times \text{StatHeap} \times \text{CallStack}$$

The semantic rules of SCIL are reductions of the form:

$$P \vdash C \Longrightarrow C'$$

where $P \in \text{Program}$ and $C, C' \in \text{Configuration}$.

As mentioned in Section 3.1.3, Android and Xamarin do not have a regular `main` method, but instead use callbacks which are called by the Android system. Therefore it is required, for an initial configuration, that static fields are initialized in the static heap. Furthermore, Xamarin apps are not expected to terminate, thus no termination state is defined.

4.2.3 Semantic Rules

With the domains and the program configuration defined, we can move on to define the semantic rules of the language. Not all of the semantic rules of SCIL will be discussed here, only the core instructions, instructions for object instantiation and method invocation.

Core Instructions

The core instructions of SCIL are primarily used to manipulate the stack, local heap and program flow.

The `nop` operation is a dummy operation that does not affect the stack frame, except to increment the program counter.

$$\frac{m.instructionAt(pc) = \texttt{nop}}{P \vdash \langle SH, H, \langle m, pc, LV, AL, St\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, St\rangle :: SF\rangle}$$

The `push` instruction pushes a value v to the top of the stack, where the element on the top of the stack is written as $\langle v :: St\rangle$.

$$\frac{m.instructionAt(pc) = \texttt{push}\ v}{P \vdash \langle SH, H, \langle m, pc, LV, AL, St\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, \langle v :: St\rangle\rangle :: SF\rangle}$$

The inverse of `push` is `pop`, which pops the top element of the stack.

$$\frac{m.instructionAt(pc) = \texttt{pop}}{P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v_1 :: St\rangle\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, St\rangle :: SF\rangle}$$

The `dup` instruction duplicates the top element of the stack, here written as v_1.

$$\frac{m.instructionAt(pc) = \texttt{dup}}{P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v_1 :: St\rangle\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, \langle v_1 :: v_1 :: St\rangle\rangle :: SF\rangle}$$

As for arithmetic instructions, SCIL have `add` which do addition on the two top elements, here written as v_1 and v_2, pops them and places the result, v, on top of the stack.

$$\frac{m.instructionAt(pc) = \texttt{add} \qquad v = v_1 + v_2}{P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v_2 :: v_1 :: St\rangle\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, \langle v :: St\rangle\rangle :: SF\rangle}$$

The `stloc` instruction pops the top element and stores it in local variable n.

$$\frac{m.instructionAt(pc) = \texttt{stloc}\ n}{P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v :: St\rangle\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV[n \rightarrowtail v], AL, St\rangle :: SF\rangle}$$

The opposite instruction of `stloc` is `ldloc`, which pushes the element from local variable n to the top of the stack.

$$\frac{m.instructionAt(pc) = \texttt{ldloc}\ n}{P \vdash \langle SH, H, \langle m, pc, LV, AL, St\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, \langle LV(n) :: St\rangle\rangle :: SF\rangle}$$

The instruction `ldarg` loads the value from the argument list with index x. The value from the argument list at index x is then pushed on top of the stack.

$$\frac{m.instructionAt(pc) = \texttt{ldarg}\ x}{P \vdash \langle SH, H, \langle m, pc, LV, AL, St\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc+1, LV, AL, \langle AL(x) :: St\rangle\rangle :: SF\rangle}$$

The `brtrue` instruction is used to change the control flow. `brtrue` branches to pc_t if the top element on the stack, v, is `true`, i.e. if v is not equal to 0. If v is `false`, i.e. equal to 0, it does not branch and the control flow jumps to $pc+1$.

$$\frac{m.instructionAt(pc) = \texttt{brtrue}\ pc_t \qquad pc' = \begin{cases} pc_t, & \text{if } v \neq 0 \\ pc+1, & \text{otherwise} \end{cases}}{P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v :: St\rangle\rangle :: SF\rangle \Longrightarrow \langle SH, H, \langle m, pc', LV, AL, St\rangle :: SF\rangle}$$

Object Instantiation

These instructions handle the instantiation of objects in SCIL, including field manipulation.

To create new instances of a class, the instruction `new` is used. It allocates an unused location on the heap, which is returned and pushed to the top of the stack, which is formalized by the *newObject* function. "Location" is defined as an infinite set of locations on the heap, and written as loc to improve readability.

$$\begin{aligned} &newObject : \text{Class} \times \text{Heap} \rightarrow \text{Location} \times \text{Heap} \\ &newObject(\sigma, H) = (loc, H') \end{aligned}$$

where

$$loc \notin \text{dom}(H) \ \wedge\ o \in \text{Object} \ \wedge\ H' = H[loc \mapsto o] \ \wedge\ o.class = \sigma$$

The new object must be properly initialized, including having all its fields initialized to the correct default value:

$$\forall f \in \sigma.fields : \neg f.isStatic \Rightarrow o.fieldValue(f) = def(f.type)$$

where the default value, *def(t)* can be either 0 or `null`, depending on whether the field is a value type or a reference type:

$$def(t) = \begin{cases} 0 & \text{if } t \in \text{ValType} \\ \texttt{null} & \text{if } t \in \text{RefType} \end{cases}$$

This is then used to create the semantic rule for `new`:

$$\frac{m.instructionAt(pc) = \texttt{new}\ \sigma \qquad \sigma \in \text{Class} \qquad (loc, H') = \text{newObject}(\sigma, \text{H})}{P \vdash \langle SH, H, \langle m, pc, LV, AL, St\rangle :: SF\rangle \Longrightarrow \langle SH, H', \langle m, pc+1, LV, AL, \langle loc :: St\rangle\rangle :: SF\rangle}$$

Method Invocation

The following instructions are used for method calls. As mentioned in Section 4.1.3, methods can either be static methods or instance methods.

To call a static method, the instruction `call` is used. Static methods are resolved at compile time and therefore no method lookup is needed to resolve it.

$$\frac{m.instructionAt(pc) = \texttt{call}\ m_t \qquad n = |m_t|}{\begin{array}{l} P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v_n :: \cdots :: v_1 :: St\rangle\rangle :: SF\rangle \Longrightarrow \\ \quad \langle SH, H, \langle m_t, 0, LV_\varepsilon, \langle AL[0 \mapsto v_n], \ \ldots\ , AL[n \mapsto v_1]\rangle, St_\varepsilon\rangle :: \langle m, pc+1, LV, AL, St\rangle :: SF\rangle \end{array}}$$

If the method is a instance method, the method lookup is resolved at runtime. As mentioned in Section 3.3, this is done by dynamic dispatch, where the method is looked up in the class hierarchy. This method resolving through the class hierarchy is done with the function `methodLookup`:

$$methodLookup(m_t, \sigma) = \begin{cases} \bot & \text{if } \sigma = \bot \\ m_t & \text{if } m_t \in \sigma.methods \wedge \sigma \neq \bot \\ methodLookup(m_t, \sigma.super) & \text{if } m_t \notin \sigma.methods \wedge \sigma \neq \bot \end{cases}$$

This returns the method m_t if it exists in the class σ, otherwise the function is called recursively until the method is found in one of the object's superclasses. If the $super$ function is called on `Object`, which is the superclass of all classes, it returns $\bot$. In this case, $methodLookup$ also returns $\bot$.

The instruction to call instance methods is `callvirt`. This instruction uses `methodLookup` to resolve which instance method is called. This is the biggest difference between `call` and `callvirt`.

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{callvirt}\ m_t \\ o = H(loc) \qquad m_v = methodLookup(m_t, o.class) \\ n = |m_t| \qquad m.isStatic = false \qquad loc \neq \texttt{null} \end{array}}{\begin{array}{l} P \vdash \langle SH, H, \langle m, pc, LV, AL, \langle v_n :: \cdots :: v_1 :: loc :: St \rangle \rangle :: SF \rangle \Longrightarrow \\ \quad \langle SH, H, \langle m_v, 0, LV_\varepsilon, \langle AL[0 \mapsto loc], AL[1 \mapsto v_n], \ \ldots\ , AL[n \mapsto v_1] \rangle, St_\varepsilon \rangle :: \langle m, pc+1, LV, AL, St \rangle :: SF \rangle \end{array}}$$

4.3 Flow Logic

In this section, a flow analysis for SCIL is defined through the definition of flow logic rules for SCIL's instructions. This flow analysis is based on a CFA, as described in Section 3.4.

4.3.1 Preliminary Definitions

In this section we introduce basic definitions and notations used in the following sections. These definitions are standard convention, and are therefore not discussed further.

A **partial order** in a set P is a relation $\sqsubseteq$ on P such that $\sqsubseteq$ is reflexive, anti-symmetric and transitive:

1. $\forall x \in P : x \sqsubseteq x$
2. $\forall x, y \in P : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
3. $\forall x, y, z \in P : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

A **partially ordered set** is a set P equipped with a partial order $\sqsubseteq$. If P has an element $x \in P$ such that $\forall y \in P : x \sqsubseteq y$ then x is called the **least element** of P, denoted $\bot$. The **greatest element** of P is an element $x \in P$ such that $\forall y \in P : y \sqsubseteq x$, denoted $\top$.

Let $(P, \sqsubseteq)$ be a partially ordered set, then $u \in P$ is an **upper bound** for S in P if $\forall x \in S : x \sqsubseteq u$. If $u \sqsubseteq v$ for all upper bounds v of S in P, then u is the **least upper bound** of S in P, denoted $\bigsqcup S$. The binary least upper bound $\bigsqcup\{x, y\}$ is written $x \sqcup y$.

Let $(P, \sqsubseteq)$ be a partially ordered set, then $l \in P$ is a **lower bound** for S in P if $\forall x \in S : l \sqsubseteq x$. If $m \sqsubseteq l$ for all lower bounds l of S in P, then l is the **greatest lower bound** of S in P, denoted $\bigsqcap S$. The binary greatest lower bound $\bigsqcap\{x, y\}$ is written $x \sqcap y$.

Let $(P, \sqsubseteq)$ be a partially ordered set such that $P \neq \emptyset$. If $x \sqcup y$ and $x \sqcup y$ exists for all $x, y \in P$, then P is a **lattice**

Let $(P, \sqsubseteq)$ be a partially ordered set such that $P \neq \emptyset$. If $\bigsqcup S$ and $\bigsqcap S$ exist for all $S \subseteq P$, then $(P, \sqsubseteq)$ is a **complete lattice**.

4.3.2 Abstract Domains

To perform a flow analysis on SCIL, abstract domains have to be defined. These abstract domains are used to support the analysis by representing runtime values. To distinguish abstract domains from the semantic domains, the abstract domains are written with an overline, $\overline{\text{Val}}$. Some abstract domains need additional structure to ensure that the analysis is well-defined, which is granted by having the abstract domain be a complete lattice. The notation $\widehat{\text{Val}}$ is used to represent abstract domains that are also a complete lattices.

In this section we will go through the different abstract domains defined for the analysis.

$\overline{\text{Num}} = \{\texttt{INT}\}$
: To represent a number in the analysis, the abstract domain $\overline{\text{Num}}$ is used. Numbers are not precisely tracked in the analysis, which is why an abstract value is modeled as an integer.

$\overline{\text{ObjRef}} = \text{Class} \uplus \{\texttt{null}\}$
: $\overline{\text{ObjRef}}$ represents the abstraction of object references, which is important for the analysis of object oriented languages, like CIL. From inspecting a number of Xamarin apps, we found that objects are often instantiated only once. Therefore, object references can be abstracted into the class of the object that they reference. The consequence of this is that some precision in the analysis is lost, but since Xamarin apps often use objects with a single instantiation, we deem it sufficient. In some cases, apps use collections of objects, where the difference of each object will be lost, but it was fairly rare in the inspected apps. To simplify the usage of abstract object references, $\sigma \in \text{ObjRef}$, it is written as $(\text{Ref}\ \sigma)$.

$\overline{\text{Val}} = \overline{\text{Num}} + \overline{\text{ObjRef}}$
: $\overline{\text{Val}}$ is a combination of the abstract domain for numbers and values.

$\widehat{\text{Val}} = \mathcal{P}(\overline{\text{Val}})$
: To represent sets of values is $\widehat{\text{Val}}$ defined, which is the powerset of $\overline{\text{Val}}$. As mentioned previously, $\widehat{\text{Val}}$ is a complete lattice.

$\overline{\text{Addr}} = \text{Addr} + (\text{Method} \times \{\text{END}\})$
: $\overline{\text{Addr}}$ is the abstract domain for addresses. It keeps track of the entry and exit points of a method, so that $pc = 0$ represents the entry point and $pc = \text{END}$ represents the exit point.

$\widehat{\text{Obj}} = \text{Field} \to \widehat{\text{Val}}$
: An object's state is the content of its fields and therefore it can be modelled as a mapping from fields to abstract values.

$\widehat{\text{LV}} = \text{Var} \to \widehat{\text{Val}}$
: To keep the local variables simple, they are modelled as a mapping from a variable to values. An alternative approach would be to have an address, such that every instruction would be accessible with an address, thus making the analysis flow sensitive. We choose to use the first and more simple approach, since this extra flow sensitivity is unnecessary.

$\widehat{\text{SH}} = \text{Field} \to \widehat{\text{Val}}$
: The static heap contains the values of static fields. It is modelled as a mapping from static fields to a set of abstract values.

$\widehat{\text{H}} = \overline{\widehat{\text{ObjRef}}} \rightarrow \widehat{\text{Obj}}$
: The heap contains dynamic elements, i.e. objects. These are accessed with an object reference, so the abstract domain for the heap is a mapping from $\overline{\widehat{\text{ObjRef}}}$ to $\widehat{\text{Obj}}$.

$\widehat{\text{AL}} = \text{Num} + \widehat{\text{Val}}$
: The argument list consists of all arguments for a method. These arguments are accessed with a index number, and conventionally the first item in the argument list is `this`, which is the pointer to the object itself.

$\widehat{\text{S}} = \overline{\widehat{\text{Addr}}} \rightarrow (\widehat{\text{Val}})$
: The stack is modelled as a mapping from the abstract domain for addresses to a value.

$\widehat{\text{Analysis}}_{\text{CFA}} = \widehat{\text{SH}} \times \widehat{\text{H}} \times \widehat{\text{LV}} \times \widehat{\text{AL}} \times \widehat{\text{S}}$
: The abstract domain for the control flow analysis can now be defined by combining the domains defined above: the static heap $\widehat{\text{SH}}$, the heap $\widehat{\text{H}}$, the local variables $\widehat{\text{LV}}$, the argument list $\widehat{\text{AL}}$ and the stack $\widehat{\text{S}}$. The results from an analysis of this domain is acceptable when the flow logic judgements are respected.

With the abstract domains defined, the flow logic rules can be specified.

4.3.3 Flow Logic Rules

In this section, some of the flow logic rules for SCIL will be described. Not all rules will be described here; a complete list can be seen in Appendix C.

Flow Logic Notation

The general form of a flow logic rule for SCIL is: $(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{instruction}$, where $(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \in \text{Analysis}_{\text{CFA}}$ and `instruction` is the instruction at the program counter pc in method m. This means that $(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S})$ is accepted as a valid analysis of the instruction in method m at program counter pc.

To bind variables, the notation $A \triangleleft B$: is used, which means that the value of B is bound to A. This notation can be extended to manipulate with abstract stacks: $A_1 :: \cdots :: A_n :: X \triangleleft \widehat{S}(m, pc)$: This notation is read as: The abstract stack at $\widehat{S}(m, pc)$ must contains at least n elements which are bound to variable A_1 through A_n. Any other elements on the stack are bound to X.

`push` Instruction

From the semantics in Section 4.2.3, `push` is defined to push a single argument v on top of the stack. To represent `push` in the analysis, the stack at address (m, pc) is contained in $\widehat{S}(m, pc)$. The `push` instruction creates a new stack with an abstract representation of v, which is denoted as $\beta_{Const}(v)$, which is, as the semantics describe, on top of the stack: $\beta_{Const}(v) :: \widehat{S}(m, pc)$. This is available for the next instruction at $pc + 1$: $\beta_{Const}(v) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc + 1)$. None of the local variables are modified when there is pushed to the stack, thus $\widehat{LV}$ should be available at the next instruction unmodified: $\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc + 1)$. This gives the full flow logic rule for the `push` instuction:

$$
\begin{aligned}
(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{push}\ v \\
\text{iff} \quad \beta_{Const}(v) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1) \\
\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

where

$$
\beta_{Const}(v) = \begin{cases} \{\texttt{INT}\} & \text{if } v \in \text{Num} \\ \{\text{null}\} & \text{if } v = \texttt{null} \end{cases}
$$

`pop` Instruction

The `pop` instruction is the inverse of the `push` instruction, as defined in Section 4.2.3: it removes, or pops, the top element from the stack. Here, a stack is assumed with the following configuration: $X :: Y \lhd \widehat{S}(m, pc)$, where X is at the top of the stack at (m, pc). In the next configuration at $(m, pc+1)$, when the `pop` instruction has executed, X is removed from the stack: $Y \sqsubseteq \widehat{S}(m, pc+1)$. The full flow logic rule for `pop` is the following:

$$
\begin{aligned}
(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{pop} \\
\text{iff} \quad X :: Y \lhd \widehat{S}(m, pc) : \\
Y \sqsubseteq \widehat{S}(m, pc+1) \\
\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`add` Instruction

The `add` instruction takes the top two numbers on the stack, add them together and replaces the two elements with the result. Therefore a stack configuration with two elements expected before the execution of the instruction: $v_1 :: v_2 :: X \lhd \widehat{S}(m, pc)$. In the next configuration, at $(m, pc+1)$, the result is: $\{\texttt{INT}\} :: X \sqsubseteq \widehat{S}(m, pc+1)$ where the two values are removed and the result, where the result of the addition $\{\texttt{INT}\}$ is on top of the stack. The complete flow logic rule is:

$$
\begin{aligned}
(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{add} \\
\text{iff} \quad v_1 :: v_2 :: X \lhd \widehat{S}(m, pc) : \\
\{\texttt{INT}\} :: X \sqsubseteq \widehat{S}(m, pc+1) \\
\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`brtrue` Instruction

The `brtrue` instruction branches to a location pc_t if the top element on the stack is interpreted as `true`. Therefore, it is expected that the stack has at least one element: $B :: X \lhd \widehat{S}(m, pc)$. A new stack is made at the next instruction: $\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$ in the case that the control flow does not branch. In the case that a branch happens, a new stack with the local variables

is created at pc_t: $X \sqsubseteq \widehat{S}(m, pc_t)$. The local variables are copied over in the same manner as the stack to both $(m, pc+1)$ and (m, pc_t). The complete flow logic rule for `brtrue` is:

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{brtrue}\ pc_t \\
&\quad \text{iff} \quad B :: X \lhd \widehat{S}(m, pc) : \\
&\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad X \sqsubseteq \widehat{S}(m, pc_t) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc_t)
\end{aligned}
$$

`stloc` Instruction

According to the semantic rules in Section 4.2.3, `stloc` takes the top element on the stack and saves it in a local variable x. Therefore, a stack state with at least one element on it is expected: $A :: X \lhd \widehat{S}(m, pc)$. Storing the top element, A in this example, in the local variables, is modelled as: $A \sqsubseteq \widehat{LV}(m, pc+1)(x)$. Since x is popped from the stack, the rest of the stack is then copied to the next instruction: $X \sqsubseteq \widehat{S}(m, pc+1)$. Finally, the local variables except x have to be transferred to the next instruction: $\widehat{LV}(m, pc) \sqsubseteq_{\{x\}} \widehat{LV}(m, pc+1)$, where $\sqsubseteq_{\{x\}}$ means that all local variables, except x are transferred. The full flow logic rule is:

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{stloc}\ x \\
&\quad \text{iff} \quad A :: X \lhd \widehat{S}(m, pc) : \\
&\qquad A \sqsubseteq \widehat{LV}(m, pc+1)(x) \\
&\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq_{\{x\}} \widehat{LV}(m, pc+1)
\end{aligned}
$$

`ldloc` Instruction

The `ldloc` instruction is the opposite of `stloc`, where `ldloc` takes a variable x and pushes the value to the top of the stack. First, the value from variable x is put on the stack: $\widehat{LV}(m, pc)(x) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1)$. Then all the local variables are copied to the new instruction, including x since the `ldloc` instruction does not affect the variable that is loaded from: $\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$. The full flow logic rule is:

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{ldloc}\ x \\
&\quad \text{iff} \quad \widehat{LV}(m, pc)(x) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`new` Instruction

The `new` instruction allocates room on the heap for a new instance of a class, which is given as argument σ. `new` returns the reference to the newly created object, which is on top of the stack: $\{(\text{Ref}\ \sigma)\} :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1)$. During the initialization, the fields in the object are set to their default values: $default(\sigma) \sqsubseteq \widehat{H}(\text{Ref}\ \sigma)$. *default* is defined as:

$$\forall f \in fields(\sigma) : default(\sigma)(f) = \beta_{Const}(def(f.type))$$

The *def* function is the same function used in the semantics in Section 4.2, which maps types to their default values. Lastly, the local variables are transferred to the next instruction, since

none were changed: $\widehat{LV}(m,pc) \sqsubseteq \widehat{LV}(m,pc+1)$. The full flow logic rule for `new` is:

$$\begin{array}{l}(\widehat{SH},\widehat{H},\widehat{LV},\widehat{AL},\widehat{S}) \models_{CFA} (m,pc) : \texttt{new}\ \sigma \\ \quad \text{iff} \quad \{(\text{Ref}\ \sigma)\} :: \widehat{S}(m,pc) \sqsubseteq \widehat{S}(m,pc+1) \\ \qquad \mathit{default}(\sigma) \sqsubseteq \widehat{H}(\text{Ref}\ \sigma) \\ \qquad \widehat{LV}(m,pc) \sqsubseteq \widehat{LV}(m,pc+1)\end{array}$$

`call` Instruction

The instruction `call` is used to call a static method. The fact that the method is static means that the method being called is known at compile time. All the arguments for the method are on top of the stack, so the stack has the following configuration: $A_1 :: \cdots :: A_{|m_0|} :: X \lhd \widehat{S}(m,pc) :$, where $|m_0|$ denotes the method arity. Then the arguments are copied to the argument list for method m_0: $A_1 :: \cdots :: A_{|m|} \sqsubseteq \widehat{AL}(m_0,0)[0..|m|-1]$.

The invoked method can either have return type `void` or not `void`. In the case it is `void`, $m_0.\mathit{returnType} = \texttt{void}$, the stack is simply transferred to the next instruction: $X \sqsubseteq \widehat{S}(m,pc+1)$. In the case it returns a value, $m_0.\mathit{returnType} \neq \texttt{void}$, the returned value can be found at the top of the stack of the invoked method m_0 at address (m_0, END), which denotes the end of the method.

The return value must then be copied back to the invoking method at the top of the stack and copied forward to the next instruction: $A \lhd \widehat{S}(m_0,\text{END}) : A :: X \sqsubseteq \widehat{S}(m,pc+1)$. The full flow logic rule for `call` can be seen here:

$$\begin{array}{l}(\widehat{SH},\widehat{H},\widehat{LV},\widehat{AL},\widehat{S}) \models_{CFA} (m,pc) : \texttt{call}\ m_0 \\ \quad \text{iff} \quad A_1 :: \cdots :: A_{|m_0|} :: X \lhd \widehat{S}(m,pc) : \\ \qquad A_1 :: \cdots :: A_{|m|} \sqsubseteq \widehat{AL}(m_0,0)[0..|m|-1] \\ \qquad\qquad m_0.returnType = \texttt{void} \Rightarrow \\ \qquad\qquad\qquad X \sqsubseteq \widehat{S}(m,pc+1) \\ \qquad\qquad m_0.returnType \neq \texttt{void} \Rightarrow \\ \qquad\qquad\qquad A \lhd \widehat{S}(m_0,\text{END}) \\ \qquad\qquad\qquad A :: X \sqsubseteq \widehat{S}(m,pc+1) \\ \qquad\qquad \widehat{LV}(m,pc) \sqsubseteq \widehat{LV}(m,pc+1)\end{array}$$

`callvirt` Instruction

The `callvirt` instruction is similar to `call`, but calls instance methods, which are resolved during runtime. Some of the steps in `callvirt` are the same as `call`, but the difference is in resolving what methods to call.

As with `call`, the parameters are on top of the stack, but here they are followed by a reference to the referenced object, here labeled B: $B :: A_1 :: \cdots :: A_{|m_0|} :: X \lhd \widehat{S}(m,pc) :$. Since the analysis approximates objects, there may be more than one object reference that have to be considered when finding which method to call:

$$\begin{array}{l}\forall(\text{Ref}\ \sigma) \in B : \\ \qquad m_v \lhd methodLookup(m_0,\sigma)\end{array}$$

Here, all the possible methods are bound to m_v for later reference with the "$\triangleleft$" notation.

For each looked up method m_v, the arguments are copied to the argument list at $pc = 0$, so they are available at the first instruction in the method: $A_1 :: \ldots A_{|m_0|} \sqsubseteq \widehat{AL}(m_v, 0)[0..|m_0|]$. Furthermore, the object reference is placed in the local variables of the methods at index 0: $\{(\text{Ref}\,\sigma)\} \sqsubseteq \widehat{LV}(m_v, 0)[0..|m_0|]$

As with `call`, the return type is either `void` or not, and the stack is updated accordingly in the same way.

Finally, all the local variables are copied to the next instruction: $\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$.

The complete flow logic rule for `callvirt` is:

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{callvirt}\ m_0 \\
&\quad \text{iff} \quad B :: A_1 :: \cdots :: A_{|m_0|} :: X \triangleleft \widehat{S}(m, pc) : \\
&\qquad \forall (\text{Ref}\,\sigma) \in B : \\
&\qquad\quad m_v \triangleleft methodLookup(m_0, \sigma) \\
&\qquad\quad A_1 :: \ldots A_{|m_0|} \sqsubseteq \widehat{AL}(m_v, 0)[0..|m_0|-1] \\
&\qquad\quad \{(\text{Ref}\,\sigma)\} \sqsubseteq \widehat{LV}(m_v, 0)[0..|m_0|-1] \\
&\qquad\quad m_0.returnType = \texttt{void} \Rightarrow \\
&\qquad\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad m_0.returnType \neq \texttt{void} \Rightarrow \\
&\qquad\qquad A :: Y \triangleleft \widehat{S}(m_v, \text{END}) : A :: X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

4.4 Analyzer Overview

The analysis consists of multiple steps in order to achieve the desired result. An overview of the steps is seen in Figure 4.1.

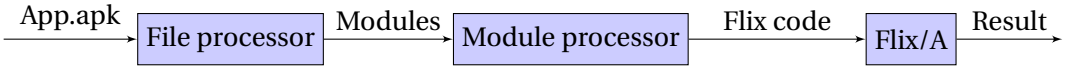

Figure 4.1: *Flow of the analysis*

The analysis takes an APK, which is described in Section 3.1.1, or a managed DLL or executable file (EXE) file as input. The *file processor* then reads the input and loads all the managed DLLs into memory and parses them using Mono.Cecil. The file processor filters APK files such that it only loads files from known directories used for Xamarin. It also supports loading Xamarin modules embedded in bundles, which reside in `libmonodroid_bundle_app.so`.

The next step is the *module processor* which consists of multiple substeps to analyze and generate Flix code. An overview of the steps can be seen in Figure 4.2.

Figure 4.2: *Flow of the module analysis*

The module processor generates a CFG and then uses different kinds of visitors to decorate the created CFG with additional information, which is used by the code generator. Examples of visitors are SSA, rewriters and analyzers.

The last step is the code generation for Flix/A, the execution of Flix with the generated code and returning the result of the analysis.

The different steps will be described in more detail in Section 4.5 and Section 4.6.

4.5 Simple CIL Analyzer

To analyze a program, we first have to transform APK files into Flix code. This is done using Simple CIL Analyzer (SCIL/A) which transforms the APK file into Flix/A facts. SCIL/A also supports direct execution of Flix to simplify the analysis of programs.

SCIL/A accepts an input file or an input path containing multiple managed DLL, APK or EXE files. SCIL/A receives the input file or input path parameter through the command line. It is for example also possible to customize the command line arguments sent to the Flix process.

It is also possible to run SCIL/A with parameters that dictate how thorough the analysis is. More precisely, it is possible to toggle the string analysis of the app. The string analysis is more thorough, but makes the scan slower. The quick mode also disables lookup for asynchronous tasks to further speed up the analysis.

SCIL/A starts by parsing the command line and then begins processing the requested file(s). Dependency injection is then initialized which is used for resolving dependencies in the program, such as for all the applied visitors on the CFG. The next step is the file processor, which receives a file to be analyzed.

4.5.1 File Processor

The first execution step of analyzing an app is the file processor. The file processor receives a file as parameter and returns a list of output Flix files which should be analyzed.

The file processor checks if the file is a ZIP file, since APK files are of ZIP format as mentioned in Section 3.1.1, by looking for a ZIP signature header. If this is found, it processes the file as a ZIP file, if not it will try to process the file as a managed DLL or EXE file.

If the file is a ZIP file, SCIL/A will look for Xamarin bundled assemblies as the first step. This is done since Xamarin supports bundling the assemblies into native code as mentioned in Section 3.2. Microsoft describes this option as “This option keeps the code safe; it protects managed assemblies by embedding them in native binaries” [40].

Bundled assemblies are found by extracting contents of the file `libmonodroid_bundle_app.so`, which is in Executable and Linkable Format (ELF). It has the managed DLL files embedded in the `.rodata` section, and is compressed using GZip.

The next step is to load files in the `assemblies/` directory for Xamarin apps. The files should also have the DLL extension. When a file is found, SCIL/A will load it into memory and load the module using Mono.Cecil.

In order to resolve all modules during the analysis (for example if one method calls another method in another module), we override the `AssemblyResolver` that Mono.Cecil uses. The resolver receives the list of all loaded modules and can therefore return the correct loaded assembly when requested.

When all the modules are loaded, the next step is to call the module processor, which processes one module at a time.

4.5.2 Module Processor

The module processor handles ignoring libraries and analyzing the module. If a module is ignored, the module processor will return `null`. This is done by first generating a control flow graph as described in Section 4.5.4 and then applying all visitors included in SCIL/A to it. The registration of the visitors is done automatically in the dependency injection container, which supports specifying which order the visitors should be executed in.

The last step consists of generating the Flix code, which is done using a visitor not registered automatically that generates the code. When the code is generated, the module processor writes the output to a new file and returns the path to this file.

4.5.3 Flix Executor

In order to start the Flix analysis, SCIL/A contains a Flix executor. SCIL/A embeds all the Flix files into the executable, including the JAR file with the version of Flix we are using.

The embedded files are then extracted to a temporary path, and the Flix process is started with an argument containing the path to all the embedded files and the generated output from the analyzed files.

Flix relies on a temporary folder with the name `target`, which is always in the current working directory. Another temporary directory is created in order to prevent potential concurrency issues when executing multiple Flix processes at a time. Concurrency issues are especially likely when we run our unit tests that are executed in parallel. The unit tests were inconsistent before we decided to create this temporary folder.

4.5.4 Control Flow Graph

SCIL/A implements a control flow graph which is similar to the control flow graph described in Section 3.4. SCIL/A represents the program using the classes `MethodBlock`, `Block` and `Node`, as well as two additional classes: the `Type` class and the `Module` class.

Each node represents one instruction. This is done so that it is possible to override the instruction with another instruction, while keeping the context of the original instruction. The node also contains a block property used for the SSA, which is described in Section 3.5. The node implements an accept method to accept any visitors, which is used for further processing.

The `Block` class contains one or more instances of `Node`. This class is an implementation of the concept referred to as "basic blocks" in Section 3.4. The `Block` class contains (after the optimization and in most cases) the longest sequence of nodes with only one successor. The `Block` class also implements the visitor pattern. The `BaseVisitor` implementation calls the visitor on each of the `Node`s in the block.

The `Method`, `Type` and `Module` classes provide logical grouping blocks for the CIL constructs with the same names.

The control flow graph for a simple method is generated in the following way:

1. Generation of a `Block` for each instruction in the method.
2. For each block, we add all possible normal targets (branching and next instruction).

3. We then add exception targets if any exception handlers have been created. This is done in a naïve way, where it is assumed that all instructions can throw an exception.
4. Next, we remove all blocks that do not have any sources, since they are therefore not called.
5. We then optimize the graph by checking if any of the blocks can be combined with the next block. They can be combined if the first block only has one target and the target is the next block. Another requirement is that the next block should only have one source which is exactly the first block.
6. As the last step, the method calculates all reachable blocks from the start block. If any unreachable block is found, it is removed.

A simple procedure call graph has also been implemented, but is not currently used. This call graph also only supports calls inside the same module.

The method ends with generating a new `Method` object, which is then used for generating a `Type` object and `Module` object.

4.5.5 Method Calls

The method calls are primarily handled in the Flix code, but SCIL/A should emit the method calls in a specific way.

In order to support argument passing, SCIL/A outputs `starg` instructions before each method call.

As the last step of method calling, the `call` or `callvirt` instruction is emitted, which generates a `RET_MethodName` parameter, which is used for returning the result. This is used by the `ret` instruction to transfer the value back to the caller. If the return type is `System.Void` this will not be emitted.

Dynamic Dispatch

Supporting dynamic dispatch in SCIL/A is done by over-approximation to include all possible calls on each method call. This is done concretely by loading arguments from all overridden base methods when the instruction `ldarg` is called. As an example, imagine two classes `A` and `B`, where `B` has `A` as base class, and a method `c(string arg)` which is virtual in `A` and overridden in `B`. The example code can be seen in Listing 4.1.

```
1  class A
2  {
3      public virtual string c(string arg)
4      {
5          return arg;
6      }
7  }
8
9  class B : A
10 {
11     public override string c(string arg)
12     {
13         return arg + arg;
14     }
15 }
```

Listing 4.1: *Example code (C#)*

The example code generates the Flix output as seen in Listing 4.2. The method `B.c()` loads the arguments sent to class `A` and `B` and therefore essentially calls this method for all method calls on `A.c()`, but if `B.c()` is called it will only call `B.c()` and not also `A.c()`.

```
1 // type_AsyncMethods.A<>
2 // method_System.String AsyncMethods.A::c(System.String)<>(System.String
      ↪ arg)
3 LdargStm("st_AsyncMethods.A::c(System.String)_0_0",
      ↪ "AsyncMethods.A::c(System.String)", 1).
4
5
6 // type_AsyncMethods.B<>
7 // method_System.String AsyncMethods.B::c(System.String)<>(System.String
      ↪ arg)
8 LdargStm("st_AsyncMethods.B::c(System.String)_0_0",
      ↪ "AsyncMethods.B::c(System.String)", 1).
9 LdargStm("st_AsyncMethods.B::c(System.String)_0_0",
      ↪ "AsyncMethods.A::c(System.String)", 1).
```

Listing 4.2: *Example output from Flix code (Flix)*

The return statement is updated in the same way, such that the inherited methods return to all the overridden methods.

This way of handling dynamic dispatch highly over-approximates all method calls. This could be improved by analyzing some of the types received, which could make the analysis more accurate.

This approach is only acceptable for taint analysis since we are only interested in how the values flow through the program.

4.5.6 Handling Branching

In order to simplify branching in SCIL/A, all branch instructions are rewritten using the `BranchRewriterVisitor`, which rewrites all the instructions into `brtrue`.

The visitor has the attribute `RegistrerRewriter`, which helps determine the order the visitors are applied. The `BaseVisitor` uses depth-first traversal of the CFG.

A simple example of the rewrite is seen in Listing 4.3. This code rewrites `br` and `br.s` into `brtrue` by loading the constant 1 onto the stack and emitting the instruction `brtrue`.

```
1  switch (node.OpCode.Code)
2  {
3      case Code.Br:
4      case Code.Br_S:
5          // Branch unconditional
6          // Load constant 1
7          newNodes.Add(new Node(node.Instruction, node.Block)
               ↪ {OverrideOpCode = OpCodes.Ldc_I4_1});
8          // Add branch true
9          newNodes.Add(new Node(node.Instruction, node.Block)
               ↪ {OverrideOpCode = OpCodes.Brtrue});
10         break;
```

Listing 4.3: *Rewrites the `br` and `br.s` instruction to `brtrue` (C#)*

Another example is the `beq` and `beq.s` instructions (branch equal)[41] as seen in Listing 4.4. This is rewritten using the instruction `ceq` (compare equal), which pushes 1 to the stack if the two top elements are equal, and 0 otherwise [42].

```
case Code.Beq:
case Code.Beq_S:
    // Branch equal
    // Add ceq (Compare equal - returns 1 if equal - 0 if not equal)
    newNodes.Add(new Node(node.Instruction, node.Block) { OverrideOpCode =
        ↪ OpCodes.Ceq });
    // Add branch true
    newNodes.Add(new Node(node.Instruction, node.Block) { OverrideOpCode =
        ↪ OpCodes.Brtrue });
    break;
```

Listing 4.4: *Rewrites the `beq` and `beq.s` instruction to `brtrue` (C#)*

The Flix code generation for `brtrue` can be seen in Listing 4.5.

```
public bool GenerateCode(Node node, out string outputFlixCode)
{
    switch (node.OpCode.Code)
    {
        case Code.Brtrue: // Branch to target if value is non-zero (true)
        case Code.Brtrue_S:
            outputFlixCode = BrTrue(node);
            return true;
    }

    outputFlixCode = null;
    return false;
}

private string BrTrue(Node node)
{
    if (node.Operand is Instruction branchToInstruction)
    {
        return $"BrtrueStm({node.PopStackNames.First()},
            ↪ {branchToInstruction.Offset}).";
    }
    throw new NotSupportedException();
}
```

Listing 4.5: *Generate code for the `brtrue` instruction (C#)*

4.5.7 Handling Asynchronous Tasks

One of C#'s features is support for Task-based Asynchronous Programming (TAP)[43] using the Task Parallel Library (TPL). This makes it easy for programmers to write asynchronous programs using the C# keywords `async` and `await`. Tasks also enable more efficient use of the system resources by queuing the tasks to a thread pool. [44], [45]

Implementation of Tasks in C# Compiler

In order to support tasks, the compiler does a lot of work, since it is implemented without any special runtime support. The content of this section is based on information from [44], [46], [47]. In order to illustrate the generated code we have some sample code in Listing 4.6. The

example code takes two parameters (path and content) and creates or overwrites the file with the content received in the parameter content. The method then returns the string "This text here is returned from the task".

```
1 static async Task<string> WriteToFile(string path, string content)
2 {
3     await System.IO.File.WriteAllTextAsync(path, content);
4     return "This text here is returned from the task";
5 }
```

Listing 4.6: *Example code which uses async and await (C#)*

The main advantage of making a method like this is the possibility to do other tasks while the I/O request is executing. This is especially the case in highly multithreaded programs, such as a web server, which can handle significantly more requests if the threads can be used for other actions while waiting for the I/O device.

Since TAP is the recommended way to work with concurrent programming in C#[1] and inspection of apps showed that it was widely used, we thought it was a high priority feature to support in SCIL/A. Therefore, in order to support tasks correctly we have to understand in more detail how they work.

The first generated part is the method definition, which does not have any notable changes from a normal synchronous method (other than returning a `Task` with the generic type parameter `string`). The method definition can be seen in Listing 4.7.

```
1 .method private hidebysig static class
    ↪ [System.Runtime]System.Threading.Tasks.Task`1<string>
2     WriteToFile(
3       string path,
4       string content
5     ) cil managed
6   {
```

Listing 4.7: *Task method definition (CIL output from JetBrains dotPeek) (CIL)*

Tasks are implemented using state machines and therefore the first part of the method is initialization of the state machine as seen in Listing 4.8. A new state machine is generated for each async method. In order to use the arguments in the state machine, we have to transfer the arguments to the state machine's fields (path and content).

[1] Quote from Microsoft documentation on TPL: *"Starting with the .NET Framework 4, the TPL is the preferred way to write multi-threaded and parallel code." [48]*

```
1  // Initialize a new instance of an implemented IAsyncStateMachine for this
       ↪ concrete method
2  IL_0000: newobj       instance void
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::.ctor()
3  IL_0005: stloc.0      // V_0
4
5  // Stores path parameter into the state machine
6  IL_0006: ldloc.0      // V_0
7  IL_0007: ldarg.0      // path
8  IL_0008: stfld        string AsyncMethods.Program/'<WriteToFile>d__1'::path
9
10 // Stores content parameter into the state machine (ldarg.1)
11 // Additional code removed
```

Listing 4.8: *Initializes the state machine and sets the two parameters (CIL output from JetBrains dotPeek) (CIL)*

The next part is initializing an `AsyncTaskMethodBuilder`, which is used to build the returned task and set the result or set an exception result in the task. The initialization can be seen in Listing 4.9. The state is also set to -1 which is the initial state.

```
1  // Creates a AsyncTaskMethodBuilder from the state machine which helps
       ↪ support starting and return a assosiated task and sets the builder
       ↪ on the StateMachine struct/class field
2  IL_0014: ldloc.0      // V_0
3  IL_0015: call         valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<!0/*string*/> valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices
       ↪ .AsyncTaskMethodBuilder`1<string>::Create()
4  IL_001a: stfld        valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>t__builder'
5
6  // Sets the start state to -1
7  IL_001f: ldloc.0      // V_0
8  IL_0020: ldc.i4.m1
9  IL_0021: stfld        int32
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>1__state'
```

Listing 4.9: *Initializes the AsyncTaskMethodBuilder (CIL output from JetBrains dotPeek) (CIL)*

The code seen in Listing 4.10 starts the task and then returns the task created using the `AsyncTaskMethodBuilder`. The method is required to start the task according to Microsoft Docs on TAP such that *"consumers may safely assume that the returned task is active"* [49].

```
1  // Loads the AsyncTaskMethodBuilder and stores it into variable V_1
2  IL_0026: ldloc.0      // V_0
3  IL_0027: ldfld        valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder‘1<string>
       ↪ AsyncMethods.Program/’<WriteToFile>d__1’::’<>t__builder’
4  IL_002c: stloc.1      // V_1
5
6  // Load addresses to V_1 (AsyncTaskMethodBuilder) and V_0 (State machine)
       ↪ and start the task ("scheduling it for execution to the current
       ↪ TaskScheduler")
7  IL_002d: ldloca.s     V_1
8  IL_002f: ldloca.s     V_0
9  IL_0031: call         instance void valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder‘1<string>::Start<class
       ↪ AsyncMethods.Program/’<WriteToFile>d__1’>(!!0/*class
       ↪ AsyncMethods.Program/’<WriteToFile>d__1’*/&)
10
11 // Loads the AsyncTaskMethodBuilder and gets the task assosiated with it
       ↪ which is then returned
12 IL_0036: ldloc.0      // V_0
13 IL_0037: ldflda       valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder‘1<string>
       ↪ AsyncMethods.Program/’<WriteToFile>d__1’::’<>t__builder’
14 IL_003c: call         instance class
       ↪ [System.Runtime]System.Threading.Tasks.Task‘1<!0/*string*/>
       ↪ valuetype [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder‘1<string>::get_Task()
15 IL_0041: ret
```

Listing 4.10: *Starts the task - quote is from [50] (CIL output from JetBrains dotPeek) (CIL)*

The method is then implemented using a state machine. Listing 4.11 shows the class definition of a state machine for the `WriteToFile` method defined in Listing 4.6. The class definition implements the `IAsyncStateMachine` interface and has all the arguments as fields, the current state, a task builder and a current task awaiter, which is used each time a task is executed. All of the auto generated fields and the class names have special names with identifiers which are not supported in C# as identifiers to prevent a name overlap [46].

```
.class nested private sealed auto ansi beforefieldinit
    '<WriteToFile>d__1'
      extends [System.Runtime]System.Object
      implements [System.Threading.Tasks]System.Runtime.CompilerServices.
          ↪ IAsyncStateMachine
  {
    // CompilerGenerated attribute removed

    .field public int32 '<>1__state'

    .field public valuetype
        ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
        ↪ AsyncTaskMethodBuilder`1<string> '<>t__builder'

    .field public string path

    .field public string content

    .field private valuetype
        ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
        ↪ '<>u__1'
```

Listing 4.11: *Class definition for an IAsyncStateMachine (CIL output from JetBrains dotPeek) (CIL)*

The most interesting method in the state machine is the `MoveNext()` method. The method definition of `MoveNext()` can be seen in Listing 4.12. The method is called each time a step is completed in the state machine, for example after a asynchronous method call that is awaited. The first part of the method, that loads the current state, is seen in Listing 4.12.

```
.method private final hidebysig virtual newslot instance void
      MoveNext() cil managed
{
    .override method instance void
        ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
        ↪ IAsyncStateMachine::MoveNext()

      // Locals and maxstack removed

      // Loads the current state into V_0
      IL_0000: ldarg.0      // this
      IL_0001: ldfld        int32
          ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>1__state'
      IL_0006: stloc.0      // V_0
```

Listing 4.12: *Method definition for a MoveNext method used in async state machines (CIL output from JetBrains dotPeek) (CIL)*

The method then branches to the current state as seen in Listing 4.13. Since we only have one `await` in this method, we only have three states. The three states are -1 (initial), 0 (I/O task completed - execute last statement) and -2 (exception or result). If we had a method with more `awaits`, we would have had more states [47].

If the state is not 0, we load the arguments into the method variables `IL_00e` and create a `CancellationToken`. We then call the `WriteAllTextAsync` method, which returns a task.

To `await` this task, we call the method `GetAwaiter()` which returns a `TaskAwaiter` used for waiting for the task and call the state machine again when the I/O operation has completed.

The whole block in Listing 4.13 is encapsulated in a try-catch block, which handles all exceptions such that it is possible to set the exception on the `AsyncTaskMethodBuilder`. The catch handler will be presented later in this section as Listing 4.19.

```
1  .try
2  {
3      // If state is equal to 0 - branch to IL_00c -> IL_005d
4      // If not equal to 0 - goto IL_000e
5      IL_0007: ldloc.0      // V_0
6      IL_0008: brfalse.s    IL_000c
7      IL_000a: br.s         IL_000e
8      IL_000c: br.s         IL_005d
9
10     // State not equal to 0 (start state is -1)
11     // Load path and content
12     IL_000e: nop
13     IL_000f: ldarg.0      // this
14     IL_0010: ldfld        string
           ↪ AsyncMethods.Program/'<WriteToFile>d__1'::path
15     IL_0015: ldarg.0      // this
16     IL_0016: ldfld        string
           ↪ AsyncMethods.Program/'<WriteToFile>d__1'::content
17
18     // Initilize a cancellation token and load it onto the stack
19     IL_001b: ldloca.s     V_3
20     IL_001d: initobj
           ↪ [System.Runtime]System.Threading.CancellationToken
21     IL_0023: ldloc.3      // V_3
22
23     // Call System.IO.File.WriteAllTextAsync which returns a task
24     IL_0024: call         class
           ↪ [System.Runtime]System.Threading.Tasks.Task
           ↪ [System.IO.FileSystem]System.IO.File::WriteAllTextAsync(string,
           ↪ string, valuetype
           ↪ [System.Runtime]System.Threading.CancellationToken)
25
26     // Get Awaiter for the IO task and stores it into V_2
27     IL_0029: callvirt     instance valuetype
           ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
           ↪ [System.Runtime]System.Threading.Tasks.Task::GetAwaiter()
28     IL_002e: stloc.2      // V_2
```

Listing 4.13: *Branch to correct state, load arguments and call task (CIL output from JetBrains dotPeek) (CIL)*

Sometimes a method returns a task which is already completed. Examples include caches and implementations of interfaces that return tasks. For performance reasons, the compiler generates a check to see if the task is already completed and can therefore continue execution instantly. The code for the check can be seen in Listing 4.14.

```
1  // Check if the task is completed already and branch to IL_0079 if it is
2  IL_002f: ldloca.s     V_2
3  IL_0031: call         instance bool
       ↪ [System.Runtime]System.Runtime.CompilerServices.
       ↪ TaskAwaiter::get_IsCompleted()
4  IL_0036: brtrue.s     IL_0079
```

Listing 4.14: *Check if the task is already complated (CIL output from JetBrains dotPeek) (CIL)*

Before executing the next part of the code, seen in Listing 4.15, the state is changed to 0, so that the state machine continues at the next state the next time it is called. This code handles registering a callback to the state machine, which is then executed when the I/O request has finished execution. This is done using the method `AwaitUnsafeOnCompleted` which Microsoft Docs states *"Schedules the state machine to proceed to the next action when the specified awaiter completes."* [51].

```
1  // Set current task awaiter in the class
2  IL_0041: ldarg.0      // this
3  IL_0042: ldloc.2      // V_2
4  IL_0043: stfld        valuetype
       ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>u__1'
5
6  // Get AsyncTaskMethodBuilder
7  IL_0048: ldarg.0      // this
8  IL_0049: stloc.s      V_4
9  IL_004b: ldarg.0      // this
10 IL_004c: ldflda       valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>t__builder'
11
12 // Loads the references to the current awaiter and the current
       ↪ AsyncTaskMethodBuilder
13 IL_0051: ldloca.s     V_2
14 IL_0053: ldloca.s     V_4
15
16 // "Schedules the state machine to proceed to the next action when the
       ↪ specified awaiter completes."
17 IL_0055: call         instance void valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>::AwaitUnsafeOnCompleted<valuetype
       ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter, class
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'>(!!0/*valuetype
       ↪ [System.Runtime]System.Runtime.CompilerServices. TaskAwaiter*/&,
       ↪ !!1/*class AsyncMethods.Program/'<WriteToFile>d__1'*/&)
18
19 // Leaves the try catch block and goto the last statement (return from
       ↪ method)
20 IL_005a: nop
21 IL_005b: leave.s      IL_00b8
```

Listing 4.15: *Schedule the state machine to run again after the async operation has completed - quote is from [51] (CIL output from JetBrains dotPeek) (CIL)*

Listing 4.16 is the start of state 0 in the state machine. The code snippet resets the task awaiter field and sets the state to -1. At the end of the code snippet, the method `GetResult()` is called on the task's awaiter, which throws an exception if the I/O request has thrown an exception. Since the `WriteAllTextAsync` is a `void` method, the `GetResult()` method does not return anything and the reason to have it is to unwrap a potential exception.

```
1  // Branch to target if state was 0
2  // Loads current task awaiter (awaiter for the IO call) and stores it into
       ↪ V_2
3  IL_005d: ldarg.0      // this
4  IL_005e: ldfld        valuetype
       ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>u__1'
5  IL_0063: stloc.2      // V_2
6
7  // Reset the task awaiter field
8  IL_0064: ldarg.0      // this
9  IL_0065: ldflda       valuetype
       ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>u__1'
10 IL_006a: initobj
       ↪ [System.Runtime]System.Runtime.CompilerServices.TaskAwaiter
11
12 // Set state to -1
13 IL_0070: ldarg.0      // this
14 IL_0071: ldc.i4.m1
15 IL_0072: dup
16 IL_0073: stloc.0      // V_0
17 IL_0074: stfld        int32
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>1__state'
18
19 // Goto target if the task is completed
20 // Get the result from the current awaiter (the IO call)
21 IL_0079: ldloca.s     V_2
22 IL_007b: call         instance void
       ↪ [System.Runtime]System.Runtime.CompilerServices.
       ↪ TaskAwaiter::GetResult()
23 IL_0080: nop
```

Listing 4.16: *Get result from the executed task and transition to state -1 (result state) (CIL output from JetBrains dotPeek) (CIL)*

Last step is to load the string onto the stack and store it into a variable, which can then be returned. This can be seen in Listing 4.17.

```
1      // Loads a string and stores it into V_1
2      IL_0081: ldstr        "This text here is returned from the task"
3      IL_0086: stloc.1      // V_1
4      IL_0087: leave.s      IL_00a3
5  } // end of .try
```

Listing 4.17: *Load string which is returned as the task result (CIL output from JetBrains dotPeek) (CIL)*

Listing 4.18 shows how the result is set using the `AsyncTaskMethodBuilder` by calling the method `SetResult`. The state is set to -2 which indicates that the state machine has reached its final state.

```
1  // Set state to -2
2  IL_00a3: ldarg.0      // this
3  IL_00a4: ldc.i4.s     -2 // 0xfe
4  IL_00a6: stfld        int32
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>1__state'
5
6  // Get AsyncTaskMethodBuilder and result (V_1) and set the result on the
       ↪ AsyncTaskMethodBuilder
7  IL_00ab: ldarg.0      // this
8  IL_00ac: ldflda       valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>t__builder'
9  IL_00b1: ldloc.1      // V_1
10 IL_00b2: call         instance void valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>::SetResult(!0/*string*/)
11
12 // Return from method
13 IL_00b7: nop
14 IL_00b8: ret
```

Listing 4.18: *Set result on `AsyncTaskMethodBuilder` and return from method (CIL output from JetBrains dotPeek) (CIL)*

In case any exception has occurred during the execution of the task, or other parts of the method that have been transformed into the state machine, the exception has to be handled and returned using the task. This is done by using the method `SetException` on the `AsyncTaskMethodBuilder` as shown in Listing 4.19.

```
1  // If a exception occured
2  // Store the exception into V_5
3  IL_0089: stloc.s      V_5
4
5  // Set state to -2
6  IL_008b: ldarg.0      // this
7  IL_008c: ldc.i4.s     -2 // 0xfe
8  IL_008e: stfld        int32
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>1__state'
9
10 // Get AsyncTaskMethodBuilder and Exception and SetException on the
       ↪ AsyncTaskMethodBuilder and leave to the return statement
11 IL_0093: ldarg.0      // this
12 IL_0094: ldflda       valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>
       ↪ AsyncMethods.Program/'<WriteToFile>d__1'::'<>t__builder'
13 IL_0099: ldloc.s      V_5
14 IL_009b: call         instance void valuetype
       ↪ [System.Threading.Tasks]System.Runtime.CompilerServices.
       ↪ AsyncTaskMethodBuilder`1<string>::SetException(class
       ↪ [System.Runtime]System.Exception)
15 IL_00a0: nop
16 IL_00a1: leave.s      IL_00b8
```

Listing 4.19: *Exception handler for the method and set exception on task (CIL output from JetBrains dotPeek) (CIL)*

Task Implementation in SCIL/A

In order to implement tasks in SCIL/A, we decided to find all method calls (`call`, `callvirt` and `calli`) that return a generic task. The emitted code for this method will be `GetResultStm` used for Flix/A to find the correct task output.

SCIL/A also looks for the `SetResult` method and emits `SetResultStm` such that Flix/A can combine those two methods. However, in order to get the correct name of the task, we have to get the task method which uses the `IAsyncStateMachine`.

This is a very simple way of implementing support for the tasks. It could also have been implemented by simulating the task environment from the method which creates the task to the output from the task. A simple way of doing this would be to create a simulated call to the `MoveNext()` method when the method `Start` is called on a state machine. `GetResult()` and `SetResult` should also be implemented and methods from a task such as `GetAwaiter()` `.GetResult()` or the `Result` property.

4.5.8 Static Single Assignment

In Section 3.5 we described how SSA can help reduce the number of variables in a program. In this section, we will document how SSA has been implemented in SCIL.

The algorithm for transforming the program to SSA form is based on the CFG and is implemented using the visitor pattern. Every node in the CFG has an accept method, that allows visitors to visit and examine each node. The `Visitor` method in the `StaticSingleAssignment-Visitor` class visits all methods in a module, as shown in Listing 4.20.

```
1  public override void Visit(Method method)
2  {
3      // Find dominators in the method (adds to the Domninators list)
4      Dominance.SimpleDominators(method);
5
6      // Find dominance frontiers
7      Dominance.SimpleDominanceFrontiers(method);
8
9      // Find all pushes to the stack
10     var stackPushes = GetStackPushes(method);
11
12     // Compute and insert phi nodes
13     InsertPhis(method, stackPushes);
14
15     // Get all variables
16     var variables = GetVariables(method);
17
18     // Compute and insert phi nodes
19     InsertPhis(method, variables);
20 }
```

Listing 4.20: *The Visitor method in the `StaticSingleAssignmentVisitor` class rewrites the module one method at a time (C#)*

First, we determine where each phi node (a node containing a ϕ function) should be placed. This is found by looking at each node's dominance frontier, as described in Section 3.5.1. Dominators and dominance frontiers are calculated with methods that implement simple algorithms from Fischer, Cytron, and LeBlanc [31], which are invoked in line 4 and 6. These algorithms are easy to implement but can be inefficient compared to slightly more advanced

algorithms described in the same book. Therefore it is possible that performance can be improved by implementing these faster algorithms.

After dominance frontiers have been determined, the visitor stores the nodes where each type of variable use occurs: `push` instructions, variable updates and arguments. This is done in line 10, 16 and 22, respectively. Different overloaded versions of the `InsertPhis` method are invoked for each type of use, with the exception of arguments, which are not currently implemented. The method call for stack pushes happens in line 13, and for variable uses in line 19.< Each `InsertPhis` method transforms the set of nodes into a common signature, which is then handled by a generic insertion method. A segment of this method can be seen in Listing 4.21.

```
1  bool addedPhiNode;
2  do
3  {
4      addedPhiNode = false;
5
6      foreach (var block in method.Blocks)
7      {
8          // Check if we have any variable assignment in this block
9          IEnumerable<KeyValuePair<Node, TKey>> stateUpdatesInBlock =
               ↪ stateUpdates.Where(variable => block.Nodes.Any(node => node
               ↪ == variable.Key));
10
11         // Handle each stack push
12         foreach (var stateUpdate in stateUpdatesInBlock)
13         {
14             // Put a phi node on each dominance frontier
15             foreach (var dominanceFrontier in block.DominanceFrontiers)
16             {
17                 // Get block list
18                 var blockList = addedNodes[dominanceFrontier];
19
20                 // Double check that the phi node does not exists currently
21                 var currentPhiNode = blockList.FirstOrDefault(phiNode =>
22                     compareNodeToKey(phiNode, stateUpdate.Value));
23                 if (currentPhiNode != null)
24                 {
25                     // Detect if the phiNode contains this block as parent
26                     if (!currentPhiNode.Parents.Contains(stateUpdate.Key))
27                     {
28                         currentPhiNode.Parents.Add(stateUpdate.Key);
29                         addedPhiNode = true;
30                     }
31                 }
32                 else
33                 {
34                     currentPhiNode = createTNode(dominanceFrontier, new
                           ↪ List<Node>() { stateUpdate.Key },
35                         stateUpdate.Value);
36                     blockList.Add(currentPhiNode);
37                     addedPhiNode = true;
38                 }
39             }
40         }
41     }
42 } while (addedPhiNode);
```

Listing 4.21: *Main loop of the generic phi insertion method (C#)*

The boolean variable `addedPhiNode` is used to keep track of whether any phi nodes have been placed for the current method. This variable is declared in line 1 and set to `false` in line 4. The do-while loop that makes up most of the method is run until the phi nodes have been added, and `addedPhiNode` is set to `true` in either line 29 or 37.

Inside the do-while loop is a foreach loop that runs through every block in the method. The block's nodes are queried for variable assignments in the statement in line 9. This statement checks for any nodes in the block that matches a key in the set of key-value pairs provided by one of the method arguments. This key is provided by each of the three specific insertion methods and indicates that the given node has received a state update, meaning that a variable has been assigned.

The block's dominance frontier is iterated over, and in line 18 each block in the frontier is used for a lookup in the dictionary `addedNodes`, which contains all blocks in the method. This yields the set of blocks that should be considered for adding phi nodes. The statement in line 21 checks if the phi node already exists, and if it does not, and if the phi node has the current block as a parent, the phi node is added in line 28. If the phi node should be added, but the blocklist does not contain the node, it is created and inserted in line 34-36.

4.6 Flix Analysis

Flix is a functional and logic programming language for calculating fixpoints on lattices. It is based on the Datalog language, which it extends by adding support for user-defined lattices and operations on these. Flix is intended for developing and implementing scalable static analysis tools, in particular points-to analysis and dataflow analysis.

Flix is a domain-specific language for recursive constraints on relations and lattices. A Flix program is a set of facts and a set of constraints for deriving new facts. The computation of such a program determines all possible facts that can be derived from the initial facts and the constraints. [52]

In order to simplify some of the explanations, Listing 4.22 shows a simple Xamarin app which leaks the user's IMEI number by sending it to a web server. This will serve as an example app for the analysis.

```
1  [Activity(Label = "XamarinGetIMEI", MainLauncher = true)]
2  public class MainActivity : Activity
3  {
4      protected override void OnCreate(Bundle savedInstanceState)
5      {
6          var tm = (TelephonyManager)GetSystemService(TelephonyService);
7          string imei = tm.DeviceId;
8
9          HttpWebRequest request =
               ↪ (HttpWebRequest)HttpWebRequest.Create($"URL?imei={imei}");
10         request.Method = "GET";
11         request.GetResponse();
12     }
13
14 }
```

Listing 4.22: *A simple Xamarin app that leaks the user's IMEI number (C#)*

Each instruction from SCIL corresponds to a relation in Flix. In addition to those relations,

there are multiple other relations used for keeping track of tainted values, results, sources and sinks. These are seen in Table 4.2

Relations defined in Flix/A		
TaintListStack	TaintListLocalVar	TaintListArg
TaintListTask	TaintListField	PointerTable
Sources	Sinks	Results

Table 4.2: *Relations defined in Flix/A*

As an example, the contents of `TaintListStack` for the example program shown in Listing 4.22 is seen in Table 4.3. Since `Android.Telephony.TelephonyManager::get_DeviceId()` matches one of the entries in *Sources*, it will be added to the list of tainted values on the stack. This is done with one of the rules for `Call` instruction seen in Listing 4.23. Everything on the right-hand side of :- will be evaluated. This can be either testing the value of a boolean function or if the element exists in the referenced relation. If the body of the rule (everything to the right) evaluates to true, then the head (everything to the left) is inferred.

Name	**Source**	**Type**
RET_*namespace*::get_DeviceId()	*namespace*::get_DeviceId()	System.String
st_*namespace*::OnCreate(*params*)_0_7	*namespace*::get_DeviceId()	System.String
RET_System.String::Format(*params*)	*namespace*::get_DeviceId()	System.String

Table 4.3: *Content of TaintListStack (actual namespaces replaced by "namespace")*

```
1 TaintListStack(ret, name, type) :- CallStm(st, ret, name, type, isTask),
2                                    Sources(name, v).
```

Listing 4.23: *Part of CallStm for detecting when a method call to a source is made (Flix)*

Besides the return value of `get_DeviceId()`, a stack element is also tainted. This happens because the return value is copied to the stack. The return value of `System.Format` is also marked as tainted, since one of its parameters is tainted. The mechanics behind this will be further elaborated in Section 4.6.2.

When the call to `HttpWebRequest.Create` is made, the rule seen in Listing 4.24 is used. It checks if any of the arguments are tained, and the method called is a sink, then adds it to the `Results` relation. The result is seen in Table 4.4.

```
1 Results(source, name, type) :- CallStm(st, ret, name, x, isTask),
2                                Sinks(name, v),
3                                TaintListArg(name, argNo, source, type).
```

Listing 4.24: *Part of CallStm for detecting when a method call to a sink is made (Flix)*

Source	**Sink**	**Type**
namespace::get_DeviceId()	System.Net.WebRequest::Create(System.String)	System.String

Table 4.4: *Flix results of analyzing the example program*

4.6.1 Handling Branching

As described in Section 4.5.6, all branching instructions are rewritten to `brtrue`. Flix/A simply ignores these instructions. In addition to that, the use of SSA, explained in Section 3.5, makes use of ϕ nodes. All logic of creating the ϕ nodes is done in SCIL/A, while Flix/A only contains the rule seen in Listing 4.25.

```
1 // Stack values
2 rel PhiStm(r: Str, x: Str, y: Str)
3
4 // If any of the values are tainted, the result is also tained
5 TaintListStack(r, source, type) :- PhiStm(r, x, y),
6                                   TaintListStack(x, source, type).
7
8 TaintListStack(r, source, type) :- PhiStm(r, x, y),
9                                   TaintListStack(y, source, type).
```

Listing 4.25: *Rule used for ϕ nodes (Flix)*

4.6.2 Handling Method Calls

Method calls have been the largest focus when developing Flix/A. There are two cases to handle: methods created in the user program, and method calls to methods included in `System` as seen in the example with `String.Format`. The rule used for these kind of methods is seen in Listing 4.26.

```
1 TaintListStack(ret, source, type) :- CallStm(st, ret, name, type, isTask),
2                                      anyArgumentTaint(name),
3                                      TaintListArg(name, argNo, source, x).
```

Listing 4.26: *Part of CallStm for detecting when method calls with tainted arguments (Flix)*

In line 1, the right-hand side of `:-` ensures that the fact matches the `CallStm` relation. The next line checks if the name of the method matches a predefined list of methods that have this behaviour, while the last line accesses the `TaintListArg` to check if any of the arguments are tainted. If all these are true, the name of the return value is added to `TaintListStack`.

When a method is called, the behavior of CIL is to call a number of `starg` instructions (equivalent to the number of arguments) before the `call` instruction. This behavior is shown in the example in Listing 4.27.

```
1 StargStm("*ns*::Format(System.String,System.Object)",
      ↪ "st_*ns*::OnCreate(Android.OS.Bundle)_0_6", 0, "System.String").
2 StargStm("*ns*::Format(System.String,System.Object)",
      ↪ "st_*ns*::OnCreate(Android.OS.Bundle)_1_2", 1, "System.Object").
3 CallStm("st_*ns*::OnCreate(Android.OS.Bundle)_0_7",
      ↪ "RET_*ns*::Format(System.String,System.Object)",
      ↪ "System.String::Format(*params*)", "System.String", 0).
```

Listing 4.27: *Flix example when making method call. Namespace replaced by *ns* (Flix)*

The `StargStm` relation and the simple rule used for non-address parameters is seen in Listing 4.28. Since an argument is always loaded from the stack, it is enough to check if that value exists in `TaintListStack`. If that is the case, the argument with its type and number will be

added to `TaintListArg`.

```
1 rel StargStm(a: Str, st: Str, number: Int, type: Str)
2
3 TaintListArg(a, number, source, type) :- StargStm(a, st, number, type),
4                                          TaintListStack(st, source, t).
```

Listing 4.28: *Flix relation for Starg (Flix)*

4.6.3 Handling `out` and `ref` Parameters

When making a call to `TryParse` for example, an `out` parameter should be provided. This is essentially an address to a local variable, in which the parsed value will be stored. This is done using a relation called `PointerTable` that contains the name of the stack position and the name of the local variable it points to. This is added when a `LdlocaStm` is found, as seen in line 3 of Listing 4.29. After the method call, the value is loaded onto the stack using `Ldloc`. The rule seen in line 5-7 shows that the `PointerTable` is used in order to check if the value pointed to is tainted.

```
1 rel PointerTable(st: Str, lv: Str)
2
3 PointerTable(st, lv) :- LdlocaStm(st, lv).
4
5 TaintListStack(st, source, type) :- LdlocStm(st, lv),
6                                     PointerTable(x, lv),
7                                     TaintListStack(x, source, type).
```

Listing 4.29: *Relations used for `ldloca` (Flix)*

The same principle applies to `ref` parameters.

4.6.4 Handling Asynchronous Tasks

Most of the logic regarding asynchronous tasks is handled in SCIL/A, which is explained in Section 4.5.7. Flix/A then takes an argument to `CallStm` and `CallvirtStm` indicating whether this method call is an asynchronous task or not. Different types of tainted values have their own relations, in order to make it easier to keep track of them. Tainted values from tasks a relation called `TaintListTask` which is shown in Listing 4.30

```
1 TaintListTask(ret, source, type) :- CallStm(st, ret, name, type, isTask),
2                                     TaintListStack(x, source, y),
3                                     isTrue(isTask).
```

Listing 4.30: *Rule used for asynchronous tasks (Flix)*

The next step is to support getting and setting the result, which is how it is implemented in SCIL/A. This gets converted to `SetResultStm` and `GetResultStm` whose behaviour is seen in Listing 4.31

```
1 rel GetResultStm(st: Str, name: Str)
2 rel SetResultStm(dest: Str, st: Str, type: Str)
3
4 TaintListStack(st, source, type) :- GetResultStm(st, name),
5                                     TaintListTask(x, source, type).
6
7 TaintListTask(dest, source, type) :- SetResultStm(dest, st, type),
8                                      TaintListStack(x, source, y).
```

Listing 4.31: *Rule used for get/set result (Flix)*

4.6.5 String Analysis

Much of the personal data that is interesting to track in the taint analysis is in the form of strings. These strings can have different forms depending on the type of data. By implementing some type of string analysis, it is possible to recognize certain types of data, in order to determine whether it is of interest to the taint analysis. Our implementation of string analysis is based on the work by Madsen and Andreasen [53].

The analysis is based on the abstract string domain defined by the *character inclusion* lattice CI. This lattice tracks what characters a string *may* and *must* include. The lattice CI is the cartesian product of four sub-lattices: c_{may}, c_{must}, e_{may} and e_{must}. The first two are powerset lattices of characters ordered by subset- and superset inclusion, respectively. The last two are boolean lattices that track whether the concrete set of strings being represented may or must contain the empty string.

This abstract string domain can be used to keep track of strings, without necessarily knowing exactly which characters the concrete strings contain.

For the string analysis, a Flix lattice consisting of a string (used for the name) and a character set, called `Charset`, has been implemented. The character set is seen in Listing 4.32

```
1 pub enum Charset {
2     case Top,
3     case Charset(Str, Str),
4     case Bot
5 }
```

Listing 4.32: *Charset used in string lattice (Flix)*

As seen in line 3, the character set consists of two strings. This is not optimal and should rather be `Set[Char]` instead of `Str`. This solution is currently used because of a bug in Flix which generates compile errors when trying to use a set of characters. The actual code therefore contains a lot of `toSet` and `toString` operations, which converts from a `string` to a `set` and vice versa. However, those have been removed from the following listings in order to increase readability. A character set should consist of two `Set[Char]` - one for *may* and one for *must*.

A lattice consists of a partially ordered set where every two elements have a least upper bound and a greatest lower bound. These need to be defined in Flix in order to define a lattice. The functions for least upper bound, `lub`, and greatest lower bound, `glb`, are seen in Listing 4.33 and Listing 4.34.

```
def lub (e1: Charset, e2: Charset): Charset = match (e1, e2) with {
    case (Bot, x) => x
    case (x, Bot) => x

    case (Charset(may1, must1), Charset(may2, must2)) =>
        Charset(Set.union(may1, may2), toString(must1, must2))

    case _ => Top
}
```

Listing 4.33: *Least Upper Bound definition (Flix)*

```
def glb (e1: Charset, e2: Charset): Charset =  match (e1, e2) with {
    case (Top, x) => x
    case (x, Top) => x

    case (Charset(may1, must1), Charset(may2, must2)) =>
        Charset(Set.intersection(may1, may2), Set.intersection(must1,
            ↪ must2))

    case _ => Bot
}
```

Listing 4.34: *Greatest Lower Bound definition (Flix)*

Since this is a partially ordered set, there also needs to be a way of defining if one element is "smaller" than the other. Therefore, the function `leq`, less than or equal, is defined as seen in Listing 4.35.

```
def leq (e1: Charset, e2: Charset): Bool = match (e1, e2) with {
    case (Bot, _) => true
    case (_, Top) => true

    case (Charset(may1, must1), Charset(may2, must2)) =>
        Set.isSubsetOf(may1, may2) && Set.isSubsetOf(must2, must1)

    case _ => false
}
```

Listing 4.35: *Less than or equal definition (Flix)*

The foundation of the string analysis is now implemented. The first instruction from SCIL that should be supported, is `ldstr` which loads a string onto the stack. The rule and definitions for this is seen in Listing 4.36.

```
StringLattice(r, StringAnalysis.abstract(c)) :- LdstrStm(r, c).

pub def abstract (s: Str): Charset = Charset(s, s)
```

Listing 4.36: *Rule for string analysis with `ldstr` instruction (Flix)*

This converts the concrete string to an abstract character set and adds it to `StringLattice` which contains all abstract strings in the program.

An operation in SCIL that is important to support is `ldloc` and `stloc` where local variables are saved to the stack and vice versa. The rules used for supporting these is seen in Listing 4.37

```
1 StringLattice(lv, charset) :- StlocStm(lv, st),
2                               StringLattice(st, charset).
3
4 StringLattice(st, charset) :- LdlocStm(st, lv),
5                               StringLattice(lv, charset).
```

Listing 4.37: *Rules for `stloc` and `ldloc` (Flix)*

Another important operation that needs to be supported is concatenation of strings, since a string should still be tracked after being concatenated with another string. The implementation for supporting this is seen in Listing 4.38.

```
1 StringLattice(result, StringAnalysis.concat(charset0, charset1)) :-
    ↪ CallStm(result, ret, name, x, isTask),
2       isConcat(name),
3       StargStm(name, st0, 0, type0),
4       StargStm(name, st1, 1, type1),
5       StringLattice(st0, charset0),
6       StringLattice(st1, charset1).
```

Listing 4.38: *String analysis when calling Concat (Flix)*

Chapter 5

Test & Evaluation

In this chapter, the test and evaluation of SCIL/A and Flix/A is performed. First, automated testing of Flix/A is described, where unit tests are created to make sure the wanted results always occur, and regression does not happen. Then SCIL/A and Flix/A are evaluated by analyzing 2,866 apps, in order to find apps that potentially leak data. From this evaluation, three apps are further analyzed, to show examples of results from the analysis.

5.1 Unit Testing

Unit testing is done by writing some small C# programs which are then analyzed with SCIL/A and Flix/A. A list of all unit tests created is seen in Table 5.1 - the names should be self-explanatory.

5.1.1 Testing `out` Parameters

AsyncFileRead	AsyncMethods	Branching
HttpBasicAuthNegative	HttpBasicAuthSimple	MethodCallMultipleParameters
MethodOverloading	MethodOverloadingWithSinks	MultipleMethodCalls
OutParameters	RefParameters1	RefParameters2
SwitchCase	VariableCasting	VirtualMethod
HttpBasicAuthConcat	MethodCalls	NonStaticMethod

Table 5.1: *Unit tests developed*

The execution of the unit tests relies on the same Flix executor that is used in the implementation of the analysis itself, as described in Section 4.5.3. An example of how the tests are structured is seen in Listing 5.1, which shows the program, and Listing 5.2, which shows the test.

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          var userInput = Console.ReadLine();
6
7          var parsedInput = int.TryParse(userInput, out int result);
8
9          Console.WriteLine(result);
10     }
11 }
```

Listing 5.1: *Program used for unit testing out parameters (C#)*

Line 7 of the program shows a `TryParse` of an integer. The result is stored in the out parameter `result` and then written to the console. We except our analysis to detect this, as shown in the `expected` value in the test assertion.

```
[Fact]
public async Task Test1()
{
    var logs = new List<string>();

    var expected = new List<Result>
    {
        new Result
        {
            Source = "System.Console::ReadLine()",
            Sink   = "System.Console::WriteLine(System.Int32)",
            Type   = "System.Int32"
        }
    };

    await Helper.AnalyzeTestProgram("OutParameters", logs);
    var actual = Helper.ParseResults(logs);

    Assert.Equal(expected, actual);
}
```

Listing 5.2: *Unit test for out parameters (C#)*

The other tests of the different operations/instructions are structured the same way.

5.1.2 Testing concatenation in string analysis

We also wanted to test that the results of the string analysis were correct. One of the tests, which is seen in Listing 5.3, is the test of concatenation.

```
static void Main(string[] args)
{
    var url = "url.dk";

    var fullUrl = "http://user:pass@" + url;

    Console.WriteLine(fullUrl);
}
```

Listing 5.3: *Unit test for string analysis (C#)*

The code should raise a flag by adding it to the `SecretStrings` lattice. There is also a test without the URL which should not be flagged.

5.2 Evaluation

After developing the analysis, we have scanned 2,866 Xamarin apps from the pool of downloaded apps in last semester's project [3]. The result was a total of 574 flagged apps (equivalent to 20 %) with 1,561 different flags. The analysis was run with the "quick mode" enabled in order to analyze as many apps as possible before the deadline. As explained in Section 4.5, this means that string analysis is not enabled, and that if a source came from an asynchronous task, it will not be remembered but instead displayed as *UNKNOWN*.

20 % flagged apps seems high - some of this is due to having `WriteLine` as a sink, since this is apparently used many times when the developer is debugging the app. In order to check if the results otherwise seem valid, it was decided to manually inspect three of them.

Analyzed apps	Flagged apps	Percentage flagged	Total flags	Analysis failed
2,866	574	20 %	1,561	190[1]

Table 5.2: *Overview of analysis results*

5.2.1 com.connixt.imarq.fm

One of the flagged apps, called iMarq FM, gives the results seen in Table 5.3

Source	Sink	Type
UNKNOWN	HttpClient::GetAsync(System.String)	System.String
Position::get_Longitude()	HttpClient::GetAsync(System.String)	System.String
Position::get_Latitude()	HttpClient::GetAsync(System.String)	System.String

Table 5.3: *Results from com.connixt.imarq.fm (namespace omitted)*

Besides the unknown source, it is seen that the user's location (latitude and longitude) is actually sent to a web server with the `GetAsync`. This can be done for legitimate reasons so it requires a deeper inspection of the app in order to find out exactly what happens. The code that gave this result is seen in Listing 5.4.

```
1  public void OnLocationChanged(Location location)
2  {
3    this.UploadLocation(location);
4  }
5
6  public async void UploadLocation(Location location)
7  {
8    LocationUpdateBroadcastReceiver broadcastReceiver = this;
9    broadcastReceiver.locationManager.RemoveUpdates((ILocationListener)
       ↪ broadcastReceiver);
10   DateTime now = DateTime.Now;
11   string fromLocationAsync = await
       ↪ Application.Context.GetAddressFromLocationAsync(location.Latitude,
       ↪ location.Longitude);
12   string data = string.Format(broadcastReceiver.xmlRequestDataFormat,
       ↪ (object) location.Latitude, (object) location.Longitude, (object)
       ↪ location.Altitude, (object) now.ToString("yyyy-MM-dd'T'HH:mm:ss"),
       ↪ (object) fromLocationAsync, (object) "Logged from iMarq FM Android
       ↪ app");
13   string str = await
       ↪ broadcastReceiver.client.SendAsync(broadcastReceiver.url,
       ↪ broadcastReceiver.soapAction, data);
14 }
```

Listing 5.4: *Snippet of code from com.connixt.imarq.fm app (C#)*

It is clearly seen that every time the user moves (changes location) the new location is submitted to a web server using a SOAP HTTP client. This type of behaviour should definitely be found with the analysis, and in this example SCIL/A demonstrates that it is capable of just that.

5.2.2 com.concapps.benfit

This app, which is a Dutch app with tips on staying healthy, shows another leak of the user's private data. The third result in Table 5.4 shows that the device id (IMEI number) is submitted to a web server.

Source	Sink	Type
TelephonyManager::get_DeviceId()	Console::WriteLine(System.String)	System.String
TelephonyManager::get_DeviceId()	Console::Write(System.String)	System.String
TelephonyManager::get_DeviceId()	WebRequest::Create(System.String)	System.String

Table 5.4: *Results from* `com.concapps.benfit`

After a closer inspection of the app, we found the code responsible for this flag. A snippet of it is seen in Listing 5.5. A lot of the code has been removed from the listing in order to make it more readable.

```
1 protected override void OnRegistered(Context context, string
      ↪ registrationId)
2 {
3     this.RegistrationId = registrationId;
4     TelephonyManager systemService = this.GetSystemService("phone") as
          ↪ TelephonyManager;
5     string device = systemService.DeviceId != null ?
          ↪ systemService.DeviceId : "UNAVAILABLE";
6
7     webService.RegisterForPushAsync(Helper.PasswordHash(device +
          ↪ registrationId + (object) Configuration.I.AppConfig.AppId),
          ↪ device, registrationId, Configuration.I.AppConfig.AppId,
          ↪ "android");
8 }
```

Listing 5.5: *Snippet of code from* `com.concapps.benfit` *app (C#)*

The code shows that the IMEI number, as well as a registration ID and app ID, is used to generate a hash for the user in order to register an account. This is acceptable since it does not save the IMEI number itself. This is a false positive that is expected and accepted since we expect that expert users will see through these types of false positives.

5.2.3 `com.asus.advantage`

Another interesting app we found was developed by Asus, called ASUS Advantage. This app leaks some private data from the user.

Source	Sink	Type
TelephonyManager::get_DeviceId()	WebRequest::Create(System.String)	System.String

Table 5.5: *Results from* `com.asus.advantage`

The code shown in Listing 5.6 shows that the app sends the device's IMEI number to an ASUS server. It also shows that it is unencrypted.

```
protected override void OnCreate(Bundle bundle)
{
  TelephonyManager systemService = (TelephonyManager)
      ↪ this.GetSystemService("phone");
  this.imei = systemService.DeviceId != null ? systemService.DeviceId :
      ↪ "000000000";
}

// Button click
pwEncode(this.imei, edittextPassword.Text);

public static string pwEncode(string imei, string text)
{
    var url = string.Format(
        ↪ "http://appservice.asus.com/AsusAdvantageWebApi/api/
        ↪ Process/?imei={0}&text={1}", (object) imei, (object)
        ↪ HttpUtility.UrlEncode(text));
    return JsonConvert.DeserializeObject<string>(new
        ↪ StreamReader(WebRequest.Create(url).GetResponse().
        ↪ GetResponseStream(),
        ↪ Encoding.GetEncoding("utf-8")).ReadToEnd().Trim());
     }
}
```

Listing 5.6: *Simplified snippet of code from* `com.asus.advantage` *app (C#)*

To test the app, we decided to run it on one of the group members' phone (with a newer version of the app) and started capturing the packets it was sending. The output from the package capture in Figure 5.1 clearly shows that the IMEI number is sent unencrypted to the server.

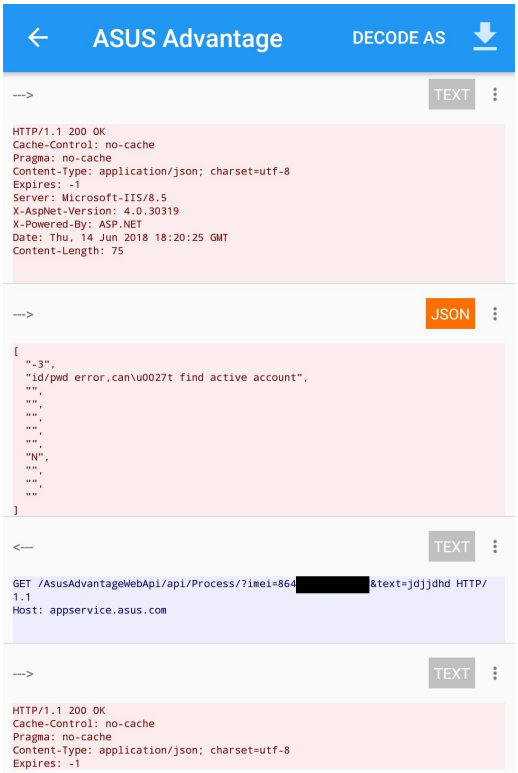

Figure 5.1: *Packet capture from the app* `com.asus.advantage` *using an application called Packet Capture*

Chapter 6

Discussion

6.1 SCIL

The discussion of SCIL focuses mostly on undocumented parts of the definition. Furthermore, the analysis defined in flow logic covers a CFG, which is also discussed.

6.1.1 Undocumented Instructions

The number of instructions in SCIL is greater than the instructions covered in the documentation of the semantics and flow logic, in Section 4.2 and Section 4.3 respectively. 12 instructions have been documented with semantic and flow logic rules, where the number of instructions in SCIL and supported by SCIL/A is 35. All 35 instructions are seen in Appendix D with an accompanying description.

It was chosen not to formally document all the instructions because of time constrains. We chose the instructions that appeared most important, which resulted in the 12 documented instructions. Furthermore, implementing SCIL/A was seen as more important, which resulted in fewer formally documented instructions.

One instruction that was not documented, but should have been, is `ret`, which returns from a method. This instruction was overlooked when the documentation of SCIL was done, thus it was not documented together with the other 12 instructions.

6.1.2 Flow Logic Only Covers CFA

The flow logic rules defined in Section 4.3 is made to cover a CFA. This is done since the CFA is the essential part of analyzing CIL code: to create the relationships between all the different parts of the code and what code is potentially executed. This is then used to analyze the data flow with a taint analysis, but the taint analysis is built on top of the CFA and uses all the relations created between the code, for instance a CFG or call graphs. Therefore we find the essential part to document about the analysis of CIL code the way SCIL/A creates the CFA.

With that said, it would be useful to have a flow logic analysis of how Flix/A performs the taint analysis of the transformed CIL code. In that way the taint analysis could also be formalized.

6.2 SCIL/A

When implementing SCIL/A, not everything was achieved and implemented to our satisfactory. In this section, some of these unsatisfactory parts of SCIL/A are discussed. This is for instance unhandled instructions, reflection, fields, SSA for arguments and exception handling.

6.2.1 Unhandled Instructions and Constructs

Since CIL is a large assembly language it contains a lot of instructions and constructs. Therefore, we decided not to implement all instructions and constructs because of time constraints.

Some of the most important instructions missing are all instructions related to arrays. The `out` and `ref` parameters are also not handled in most cases, as well as support for exception filters.

The unhandled instructions complicate things, since unhandled instructions are still assigned a unique stack name (if the instruction pushes to or pops from the stack) and therefore all further processing of this value is not possible.

Array Support

Array support is one of the more serious missing implementations. Many apps use lists for many data structures, but lists also use arrays internally, and we do not have a specific handling of lists, collections or enumerable in SCIL/A. This essentially makes the analysis ineffective if any parameters are sent to either a list or array.

This limitation will potentially results in false negatives, and is therefore a important feature given the target audience of expert users described in Section 2.3.1. The feature was not implemented due to time constraints.

The `out` and `ref` parameters

The `out` and `ref` parameters are also in many cases not handled. Flix/A handles tainting output parameters with a pointer table to support use cases like `int.TryParse(string s, out int result)`, where we are interested in tainting the value result if the input parameter `s` is tainted. The unhandled use case is if a user-defined method returns a tainted value using the output parameter. Our only support is therefore cases where we know that if an input parameter is tainted, we also need to taint the output parameter. This is therefore only usable for methods inside the framework libraries.

Exception Filter Handling

A feature which exists in CIL is exception filters. Currently, we do not handle this construct when we generate the CFG. The result of this is that some code is removed by the reachability check

Exception filters are used to determine if a catch handler should be called. An advantage of the exception filter is that, when a catch handler is called, the stack is unwound. This is not the case under the execution of the exception filter, which therefore can make debugging easier. [54]

Handling the exception filters has been a low priority since exception filters are only directly supported in C# 6 [54] (and some other languages targeting CIL[55]) and is typically used for filtering of the exceptions received.

6.2.2 Reflection

Reflection has been out of scope for the development of SCIL/A. This can potentially lead to false negatives, since it is possible to call individual methods using reflection.

However, since C# is primarily a statically typed language, it is typically bad practice to call reflection.

Focusing on implementing the reflection methods would have resulted in deprioritizing other features.

6.2.3 Fields

Currently, we only supports one instance of a class, and therefore all field accesses are done to the same field, which could lead to false positives.

We think this is acceptable for now, however in order to remove many of the false positives it should be considered if it is possible to analyze the specific instances of the objects, such that it is possible to get the instance fields instead.

6.2.4 Libraries

The analysis takes a long time for many apps because of the size of a typical app. One contributing factor to the large apps is the libraries included, which therefore makes the analysis slower. In order to decrease the time usage, it was decided to exclude many libraries used in Xamarin apps. The big disadvantage is that some of the libraries could return tainted values or should taint the output value if the input value is tainted.

6.2.5 SSA for Arguments

SSA is currently not supported for arguments. This creates an over-approximation since the argument generates the same name for every usage of the argument, and therefore the arguments will be represented as tainted if they at any point in the function or a method call are tainted. This is therefore considered a low priority problem, given the target audience of expert users described in Section 2.3.1.

False negatives will be worse for the expert users, since they can rule out false positives by manually inspecting the result, but they do not have a result to look at if the tool does not return any problems.

6.2.6 Method Overloading

The way of handling method calling is over-approximation by essentially calling all overloads of a specific called method.

This will give more false positives, but since the target audience as defined in Section 2.3.1, this is considered as a low priority.

This could possibly be improved by analyzing the specific type the method is called on. In this way, it would be possible to find the correct method that should be called.

6.2.7 Method Calling

Another over-approximation issue exists for method calls in SCIL/A. The problem is that every method is assumed to only be called once or the output is always tainted or never tainted.

This is especially a problem since some methods (for example in the runtime) are called a lot. An example is calling a method which returns a tainted string. The method then concatenates the tainted string with another value. Another unrelated method then has a statement where two strings are concatenated. The resulting string is tainted, which is a false positive.

6.2.8 Control Flow Graph

The currently implemented CFG will in specific cases not generate the most optimized “basic blocks”.

Currently, the CFG is optimized by looking at the instructions in order to determine if it is possible to combine the current block with the next block. This is sometimes not the most efficient way of doing it. An example of this is the code shown in Listing 6.1.

```
1 IL_01     brtrue IL_03
2 IL_02     br     IL_04
3 IL_03     br     IL_05
4 IL_04     br     IL_06
5 IL_05     br     IL_06
6 IL_06     ret
```

Listing 6.1: *CIL pseudo code for illustration of problem in CFG (CIL)*

The code in Listing 6.1 will generate a block for every instruction with the current method for generating the CFG. A more optimized version would include `IL_02` and `IL_04` as one block and `IL_03` and `IL_05` as another.

The more optimized version could be created by looking at the targets for each of the blocks, instead of the current approach that goes through the blocks in order. However, it should be kept in mind that special handling must be made for loops since it could otherwise make the CFG follow the loop forever.

The problem does not affect the taint analysis in any way and is therefore not a issue we want to fix. The SSA should also be unaffected.

Exception Handling

Another issue with the CFG is the way exceptions are handled. Currently it is assumed that the exception that is handled can be thrown anywhere the `try` block starts. A example is a method that handles the `DivideByZeroException`, which is also set as a target from the `add` instruction, which can never throw this exception, since the exception is only thrown when any division by zero occurs. The current way of handling this is an over-approximaion, but it is sufficient for the analysis.

6.2.9 Procedural Call Graph

SCIL/A implements a simple procedural call graph that only works with methods inside the currently scanned module, and therefore does not handle many methods. If the procedural call graph was actually used, it would be a big disadvantage.

6.2.10 Tasks

In order to support tasks, some special task methods were created. However, it is possible that the implementation of the task handling is not working in some cases where the compiler generates other code than expected. The registered problem is that sometimes it is not possible to find the specific task which calls the state machine. We did not have time enough to investigate the problem, but it works for many cases that we have tested.

6.2.11 More Specific Context

Another over-approximation is for example if we have a class that is used for settings to another class, which returns a tainted value if a specific property is set.

Currently, our analysis only supports tainting the value when it is set, and therefore we could end up tainting a value which should not be tainted in specific cases.

One example is the class `RequestMessage`, which should only be tainted when it is used by the method `SendAsync`. The `RequestMessage` is not normally used without using `SendAsync`, so it should not be a problem for the analysis to taint the class if any of the properties are set with a tainted value. This will only result in over-approximation.

6.3 Flix/A

When implementing Flix/A, a few things could have been done better. This section will cover the discussion of failed analysis attempts and the slowness of the string analysis.

6.3.1 Failed Analysis Attempts

As seen in Table 5.2, the analysis had failed 190 times. There are multiple reasons for this. The first being that the analysis was run on an always-on desktop computer owned by one of the group members. This computer has multiple users, and since the analysis requires a lot of CPU, one of the other users decided to kill this process once in a while, because it made the computer slow – unknowingly killing our analysis. This accounts for around half of the failed analysis attempts.

The other half is caused by apps taking an incredibly long time to analyze, which we assume is caused by state space explosion. After an app analysis had been running for multiple hours, we sometimes decided to kill it in order to move the analysis forward and analyze as many apps as possible. Once in a while, the analysis also failed because of an exception when trying to resolve the libraries.

6.3.2 Slow String Analysis

The analyses done in order to evaluate the project does not include string analysis but only taint analysis. The reason for this is that the string analysis takes hours for even just a single app, which was not viable, since we wanted quantitative results from many apps, instead of qualitative results for few apps. The implementation chapter mentions a bug in Flix that made it necessary to use `string` instead of `Set[Char]` which leads to a huge computation penalty when converting between the two. This may be what accounts for the long processing time.

Chapter 7

Conclusion

To conclude on the project, we look at the problem statement and the requirements that were created in the beginning of the report. The problem statement from Section 2.4 states the following:

> How can a static taint analysis tool be constructed to examine the flow of data in apps developed using Xamarin?

From the problem statement, it is required that we perform a taint analysis on Android apps made with Xamarin. As stated throughout Chapter 4, the creation of SCIL, SCIL/A, and Flix/A are described, which together makes it possible to perform a taint analysis on CIL code. In Section 5.2, it is shown that the analysis tool is capable of scanning Xamarin apps and detecting when a source flows to a sink. Therefore, the analysis tool fulfills the requirements set by the problem statement.

In Section 3.7, six functional and three usage requirements are set for the analysis tool. To evaluate on these nine requirements, each requirement is assessed to determine whether the analysis tool fulfills it.

First, the technical requirements are evaluated:

Parse and extract Xamarin APK files

In Section 4.5.1, the implementation of the file processor is documented, which is the part of SCIL/A that extracts the APK file. When evaluating the analysis tool in Section 5.2, SCIL/A extracted all 2,866 apps without any issues. Therefore, we conclude that the analysis tool fulfills the requirement for parsing and extracting the relevant data from APK files.

Create and transform the CIL code to an intermediate format

The simplified CIL intermediate language SCIL is presented and defined in Section 4.1 and Section 4.2. SCIL/A, presented in Section 4.5, takes CIL code as input, in the form of an APK file, and outputs SCIL code in the form of Flix facts to Flix/A. In the evaluation documented in Section 5.2, the analysis tool analyzed 2,866 apps, where each app was converted to SCIL by SCIL/A.

In regards to the creation of SCIL, we have created a language that is able to cover a large part of CIL, but not all instructions are covered, as discussed in Section 6.2.1. An example of uncovered instructions are instructions regarding arrays.

The documentation of SCIL does not completely cover the language, as mentioned in Section 6.1.1. Only the instructions that we consider most important are documented, where the complete language should be documented to get a satisfactory semantic definition of the language.

Nevertheless, we believe that we have successfully created the intermediate language SCIL that makes static analysis simpler.

Convert code to SSA form

A requirement was to be able to represent the code in SSA format. In Section 4.5.8, the implementation of SSA in SCIL/A is documented.

We are able to convert all code in a Xamarin app to SSA, except for method arguments, as mentioned in Section 6.2.5. Method arguments are skipped because it was deemed low priority. It was found while investigating Xamarin apps during the development of SCIL/A, and implementing the SSA form, that the overriding of method parameters was not widely used. To combat the few cases where method parameters are overridden, over-approximation is used, meaning that if a method parameter is tainted, all usage of the parameter will continue being tainted in all future cases. This is an over-approximation, but we deem that it is acceptable. Therefore, we believe that we successfully convert SCIL code into SSA format, thus fulfilling the requirement.

Control Flow Analysis & Resolve Dynamic Dispatch

The construction of a CFG and resolving dynamic dispatch is combined together here, since the two requirements are connection.

In Section 4.3, flow logic rules for constructing a CFA for SCIL were defined. Defined by the flow logic rules is also how to resolve dynamic dispatch through a lookup in the class hierarchy, which is used with the `callvirt` instruction, since is makes use of dynamic dispatch.

As with the definition of SCIL, not all aspects of the CFA are covered with flow logic, but we believe that the most important aspects of the CFA are covered. What is missing is completeness: only the important aspects are covered which results in e.g. array and interfaces to not be defined.

In regards to the implementation of the CFA, a CFG is implemented in Section 4.5.4. This CFG is used as the base for the whole transformation of CIL to SCIL, for instance SSA is handled through the CFG by decorating the nodes through visitors.

With the resolving of dynamic dispatch, this is done by over-approximating the possible methods to be called by a `callvirt` instruction. This means that all possible calls are included as targets, thus creating method calls that cannot happen during normal execution of the app. This a coarse over-approximation which results in more methods being analyzed, thus resulting in longer analysis time.

To conclude, we have created a CFA analysis based on a CFG, which is used to scan Xamarin apps with. Therefore, we believe that we have created an acceptable CFA for scanning Xamarin apps.

In regards to the resolving of dynamic dispatch being done by over-approximation, no calls to methods are lost, thus if an instance method is called, we will detect it. Therefore, we believe that we have implemented a naïve way of resolving dynamic dispatch, but it is a functioning one.

Take Android entry point and lifecycle into consideration

The handling of Android entry points and the lifecycle of Android components is something that we do not consider at all. Due to time constraints, it was not considered. This makes our analysis less precise when scanning Xamarin apps, and some code could be skipped by the scanner, since the special Android entry points are not considered. Therefore, we can conclude that this requirement is not fulfilled.

With the functional requirements covered, the usage requirements are evaluated:

Automatic analysis
A requirements was to have an automatic analysis, which could run without any user interaction. When analyzing the apps in Section 5.2 when evaluating the analysis tool, the tool ran non-stop for 14 days, without requiring any input from the user. Therefore, we think that we fulfill the requirement of having an automated analysis.

Have a coarse, but quick scan mode
When starting a scan, it is a requirement to have a coarse, but quick mode. As mentioned in Section 4.5, this is implemented in the way that the quick mode scans without the string analysis and without making asynchronous task lookup. This greatly increases the speed of the analysis, but the analysis is not as thorough. This requirement of having a quick mode is fulfilled.

Have a slower, but thorough scan mode
As with having a quick mode, we also have the thorough mode, which is a scan with the asynchronous task lookup and with the string analysis. This scan is slow compared to the quick mode, but it is the best analysis the tool can deliver. Our analysis also fulfills this requirement.

With the evaluation of the problem statement and the requirements posed for the analysis tool, we believe that we have almost completely achieved what we wanted to do. We created an analysis tool which is in great length able to scan Android apps developed using Xamarin, and get results from the scanner that can be used to identify apps that potentially leak a user's data. A downside of the tool is that it can be slow, but this was not a major focus point, since it was more important to have a usable tool than an efficient one.

Chapter 8

Future Work

8.1 Complete Documentation of SCIL

As mentioned in Section 6.1.1, not all SCIL instructions are formally documented. Therefore, a part of the future work for SCIL would be to fully document the language. This includes having a full structured operational semantics for the language.

Some of the things that are missing completely from the documentation are: interfaces, exception handling and type handling.

8.2 Flow Logic for Taint Analysis

In Section 6.1.2, it was described that the flow logic of SCIL only covers a CFA. While the CFA is considered the essential part of analyzing CIL, the taint analysis could also be formalized in the same way as the CFA. Therefore the creation of flow logic rules for the taint analysis is a part of the future work of this project.

A way of creating the flow logic for the taint analysis could be to extend the current flow logic for CFA, thus transforming it to cover taint analysis. Another way could be to make a separate extension to the current flow logic just for the taint analysis, thus making the flow logic for the taint analysis dependent on the current flow logic for the CFA.

8.3 Features Described in Discussion

Some unimplemented features are problematic as described in the discussion at Chapter 6.

This section will briefly describe how each of the features could be implemented.

8.3.1 Object Reference

One of the current problems with the analysis is that it assumes that there can only be one instance of a class. This could result in over-approximation and false positives.

One way of handling this in a better way could be to assign a unique identifier to an object based on the address where it is initialized. This identifier would then follow the object to all methods and therefore make it possible to assign to fields for each object. This method will increase the time consumption for the analysis, since it has to take this extra identifier into account when analyzing all the other facts.

One problem with this approach would also be that if multiple objects are initialized in a loop, each of the elements would get the same identifier. This is still preferable, since the number of identifiers would be finite with an identifier per instruction that creates an object, instead of potentially infinite if a new identifier was created for each new object.

8.3.2 Method Call Identifier

Like the object identifier, we could also implement some method call identifier, which is also represented by the position where the call is made.

This would help solve the problems described in the discussion in Section 6.2.7, where a result from a method could get tainted for all calls to this method although only one call should have been tainted.

In order to support the identifier, it would need to be copied into every tainted value, such that it is possible to return it at the end.

The change would have the side effect that it would be easier to track a tainted value through the program. Currently, you have to look for the result manually in the program, but if the instruction call was numbered it would be easy to find the concrete call.

8.3.3 Arrays and Lists

Currently there is no support for arrays and lists. The lack of support for arrays and lists can cause a tainted value to not be propagated into a sink if it is added to an array or list.

This should be implemented, however since both arrays and lists can be theoretically unlimited size it would not always be possible to register the individual elements. Therefore it might be the case that the complete array or list should be considered tainted if a value is added.

Other features to consider is the interface `IEnumerable`, which is a iterator that supports reading one element at a time. The `IEnumerable` interface is for example used in `foreach` (C#) statements. Other uses include .NET Language-Integrated Query (LINQ) which for example supports filtering an `IEnumerable` and returning a new `IEnumerable`.

The tainted values should therefore be propagated out into different methods, and this would therefore possibly be a substantial feature to implement.

8.3.4 Aliasing

Currently, loading addresses is not handled generally and only in some specific cases. An example of this is the handling of the `out` and `ref` parameters. This is currently handled by a special pointer list, however the actual methods used for writing to an address are not handled.

An example of a problem in the current implementation is when loading the address for fields, arguments or local variables, where only local variables are handled right now, if they are passed to a `TryParse` method. Address handling should be implemented in Flix/A to do it properly. This should be done by extending the pointer table to include all types of pointers to fields, arguments and local variables.

In order to actually use the references, the instructions used for storing values in addresses should also be generated using the Flix Code Generator. An example of a missing instruction is `stind.ref`, which is used to support `out` and `ref` parameters in methods developed by the user.

8.3.5 Identifying Managed DLL or EXE Files

To support future use cases, it would be beneficial to have a way of detecting if a DLL or EXE file is managed. The could specifically be the case of scanning ZIP or APK files for additional managed files. This would also prevent an author of a malicious app to include a DLL file inside another folder and dynamically load it using freflection, since it could then be detected.

8.3.6 Library Scanning

Since libraries are often not interesting when scanning an app, it was decided to ignore as many libraries as possible, so that the scan is faster.

This has the effect that if a library has a method that returns a tainted value, it will not be detected. We could choose to include the libraries when scanning an app, or to scan each library once and register the tainted values. Each specific file version could be checked with a hash value, the name and the assembly version. When an app is registered using a specific library the analysis loads the taint list and scans the app with this list. This would both be more efficient than scanning with all libraries and more accurate than ignoring the libraries.

8.3.7 Better Handling of Modules

Currently, the modules are processed one at a time. This has a disadvantage that it is not possible for the procedural call graph to reference methods inside other modules.

This was a design choice, but it should be possible to change without major issues. Instead, all modules could be loaded at the same time, so that calls from one module to another can be handled properly.

The analysis could potentially be more accurate by doing that, and it would be easier to get information of the calls to other libraries.

8.4 Type Analyzer

SCIL/A currently implements a simple type analyzer, which tries to analyze the specific type of a variable. The type analyzer does not implement all instructions yet and does not currently support retrieving information from other methods.

Other issues include that the type analyzer currently does not handle all instructions, meaning that in some cases it is unable to get the result of e.g. a specific stack value.

8.4.1 Method Overloading

Currently, the method overloading resolutions use over-approximation. This could be more accurate by improving and using the type analyzer.

An example is when a value is returned from a method which has the return type `A`. Currently, we do not know if the method actually returns an instance of `B` that inherits from `A`, but if we looked at the method, we could see that it can never return an instance of class `A` but always returns an instance of class `B`.

If we know that the concrete type is `B`, we could directly call method `a()` on class `B` instead of over-approximating by calling the method on both class `A` and class `B`.

8.5 Tasks

The way tasks are implemented currently is not perfect, and could be updated to be more precise and handle the callbacks into the state machine as described in Section 4.5.7. The task handling should specifically look into the class `AsyncTaskMethodBuilder` and the task methods such as `Start`, which should simulate the first call to the `MoveNext()` method in the state machine.

The `SetException()` and `SetResult()` methods should be handled as well as the `GetResult()` method on the `TaskAwaiter`.

The updated task handling could take advantage of the type analyzer to get the type of the specific task that the `GetAwaiter()` method is called on, and also the references to other tasks that should be awaited.

8.6 Structured output

A feature which could be an improvement to automating the scanning is to produce structured output from Flix/A. The structured output could for example be in `JSON` or `XML` format.

This will be done such that it would be easier for users of the analyzer to analyze a large number of apps, and read the output in a way that it is searchable.

8.6.1 Output into Database

In order to search many apps, it would be beneficial to build a database or further extend the database developed in last semester's project [3].

If the structured output is created first, it would be easier to implement an output to the database, and it could be implemented in a way where it would be optional, and not required, since some use cases could be to use the scanner only to scan a single app.

8.7 More Features in String Analysis

One possible addition to implement in the future could be more features for the string analysis. The string analysis could for example be extended with http/https checking of urls or high entropy keys, which could be used for api keys or hardcoded passwords.

Other possibilities of extension of the string analysis is detecting if specific strings can contain ip addresses, SMTP server informations, database connection strings or email addresses. There are a lot of other possibilities other that that and the biggest question is the use case needed to be solved.

In order to handle dynamic input from the user the string analysis should also have the possibility to further extend the handling of specific string manipulation methods such as the method `String.Replace(String oldValue, String newValue)` which supports replacing for example one character to the empty string. If this method is called it could therefore be safely assumed that the specified character will not be in the string.

Appendix A

Sources and Sinks Configuration

```
<android.location.Location: double getLatitude()> -> _SOURCE_
<android.location.Location: double getLongitude()> -> _SOURCE_
<android.location.LocationManager: android.location.Location
    ↪ getLastKnownLocation(java.lang.String)> -> _SOURCE_

<android.telephony.TelephonyManager: java.lang.String getDeviceId()>
    ↪ android.permission.READ_PHONE_STATE -> _SOURCE_
<android.telephony.TelephonyManager: java.lang.String getSubscriberId()>
    ↪ android.permission.READ_PHONE_STATE -> _SOURCE_
<android.telephony.TelephonyManager: java.lang.String
    ↪ getSimSerialNumber()> android.permission.READ_PHONE_STATE -> _SOURCE_
<android.telephony.TelephonyManager: java.lang.String getLine1Number()>
    ↪ android.permission.READ_PHONE_STATE -> _SOURCE_

%bundle sinks
<android.media.AudioRecord: int read(short[],int,int)> -> _SOURCE_
<android.media.AudioRecord: int read(byte[],int,int)> -> _SOURCE_
<android.media.AudioRecord: int read(java.nio.ByteBuffer,int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ getInstalledApplications(int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ getInstalledPackages(int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ queryIntentActivities(android.content.Intent,int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ queryIntentServices(android.content.Intent,int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ queryBroadcastReceivers(android.content.Intent,int)> -> _SOURCE_
<android.content.pm.PackageManager: java.util.List
    ↪ queryContentProviders(java.lang.String,int,int)> -> _SOURCE_

<java.io.OutputStream: void write(byte[])> -> _SINK_
<java.io.OutputStream: void write(byte[],int,int)> -> _SINK_
<java.io.OutputStream: void write(int)> -> _SINK_

<java.io.FileOutputStream: void write(byte[])> -> _SINK_
<java.io.FileOutputStream: void write(byte[],int,int)> -> _SINK_
<java.io.FileOutputStream: void write(int)> -> _SINK_

<java.io.Writer: void write(char[])> -> _SINK_
<java.io.Writer: void write(char[],int,int)> -> _SINK_
<java.io.Writer: void write(int)> -> _SINK_
<java.io.Writer: void write(java.lang.String)> -> _SINK_
<java.io.Writer: void write(java.lang.String,int,int)> -> _SINK_
<java.io.Writer: java.io.Writer append(java.lang.CharSequence)> -> _SINK_

<java.io.OutputStreamWriter: java.io.Writer
    ↪ append(java.lang.CharSequence)> -> _SINK_

<java.net.URL: void set(java.lang.String,java.lang.String,int,
    ↪ java.lang.String,java.lang.String)> -> _SINK_
```

```
<java.net.URL: void set(java.lang.String,java.lang.String,int,
    ↪ java.lang.String,java.lang.String,java.lang.String,
    ↪ java.lang.String,java.lang.String)> -> _SINK_

<java.net.URLConnection: void
    ↪ setRequestProperty(java.lang.String,java.lang.String)> -> _SINK_

<android.content.Context: void sendBroadcast(android.content.Intent)> ->
    ↪ _SINK_
<android.content.Context: void
    ↪ sendBroadcast(android.content.Intent,java.lang.String)> -> _SINK_
<android.content.Context: void
    ↪ sendOrderedBroadcast(android.content.Intent,java.lang.String)> ->
    ↪ _SINK_

<android.content.ContextWrapper: void
    ↪ sendOrderedBroadcast(android.content.Intent,java.lang.String)> ->
    ↪ _SINK_

<android.media.MediaRecorder: void setVideoSource(int)> -> _SINK_
<android.media.MediaRecorder: void
    ↪ setPreviewDisplay(android.view.Surface)> -> _SINK_
<android.media.MediaRecorder: void start()> -> _SINK_

<android.telephony.SmsManager: void
    ↪ sendTextMessage(java.lang.String,java.lang.String,java.lang.String,
    ↪ android.app.PendingIntent,android.app.PendingIntent)>
    ↪ android.permission.SEND_SMS -> _SINK_
<android.telephony.SmsManager: void
    ↪ sendDataMessage(java.lang.String,java.lang.String,short,byte[],
    ↪ android.app.PendingIntent,android.app.PendingIntent)>
    ↪ android.permission.SEND_SMS -> _SINK_
<android.telephony.SmsManager: void
    ↪ sendMultipartTextMessage(java.lang.String,java.lang.String,
    ↪ java.util.ArrayList,java.util.ArrayList,java.util.ArrayList)>
    ↪ android.permission.SEND_SMS -> _SINK_
<java.net.Socket: void connect(java.net.SocketAddress)> -> _SINK_
<android.os.Handler: boolean sendMessage(android.os.Message)> -> _SINK_

<android.bluetooth.BluetoothAdapter: java.lang.String getAddress()> ->
    ↪ _SOURCE_
<android.net.wifi.WifiInfo: java.lang.String getMacAddress()> -> _SOURCE_
<java.util.Locale: java.lang.String getCountry()> -> _SOURCE_
<android.net.wifi.WifiInfo: java.lang.String getSSID()> -> _SOURCE_
<android.telephony.gsm.GsmCellLocation: int getCid()> -> _SOURCE_
<android.telephony.gsm.GsmCellLocation: int getLac()> -> _SOURCE_

<android.accounts.AccountManager: android.accounts.Account[]
    ↪ getAccounts()> -> _SOURCE_
<java.util.Calendar: java.util.TimeZone getTimeZone()> -> _SOURCE_
<android.provider.Browser: android.database.Cursor getAllBookmarks()> ->
    ↪ _SOURCE_
<android.provider.Browser: android.database.Cursor getAllVisitedUrls()> ->
    ↪ _SOURCE_

<java.net.URL: java.net.URLConnection openConnection()> -> _SINK_

<org.apache.http.impl.client.DefaultHttpClient:
    ↪ org.apache.http.HttpResponse
    ↪ execute(org.apache.http.client.methods.HttpUriRequest)> -> _SINK_
```

```
74 <org.apache.http.client.HttpClient: org.apache.http.HttpResponse
       ↪ execute(org.apache.http.client.methods.HttpUriRequest)> -> _SINK_
75
76 <android.content.Context: void startActivities(android.content.Intent[])>
       ↪ -> _SINK_
77 <android.content.Context: void
       ↪ startActivities(android.content.Intent[],android.os.Bundle)> ->
       ↪ _SINK_
78 <android.content.Context: android.content.ComponentName
       ↪ startService(android.content.Intent)> -> _SINK_
79 <android.content.Context: boolean bindService(android.content.Intent,
       ↪ android.content.ServiceConnection,int)> -> _SINK_
80 <android.content.Context: void sendBroadcast(android.content.Intent)> ->
       ↪ _SINK_
81 <android.content.Context: void
       ↪ sendBroadcast(android.content.Intent,java.lang.String)> -> _SINK_
82
83 <java.lang.ProcessBuilder: java.lang.Process start()> -> _SINK_
```

Listing A.1: *Sources and Sinks configuration (FlowDroid)*

Appendix B

ZIP File

This appendix serves as an overview to show what is included in the ZIP file that is uploaded digitally to Digital Exam along with the report.

- **SCIL.zip**
  Includes the source code for SCIL/A and Flix/A.

  Flix/A can be found in `SCIL/Analysis`.

  The source code can also be found on GitHub via the following link:

  `https://github.com/sahb1239/SCIL/tree/final`. The final version has the tag "final" and the commit ID: `d1d2000528dc5f83cee49629a0040e457b07480b`.

  Remember to initialize the submodule in order to get Flix/A working.

  The submodule can be found here:

  `https://github.com/mclc/Shared-Project-SW10-deis2037/tree/final`. The final version of Flix/A has the tag "final" and the commit id: `150e1dad797a80051be51858d1137c7c373c57f5`.

Appendix C

Flow Logic Rules

`nop` instruction

$$(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{nop}$$
$$\text{iff} \quad true$$

`push` instruction

$$(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{push}\ v$$
$$\text{iff} \quad \beta_{Const}(v) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1)$$
$$\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$$

where

$$\beta_{Const}(v) = \begin{cases} \{\texttt{INT}\} & \text{if } v \in \text{Num} \\ \{\text{null}\} & \text{if } v = \texttt{null} \end{cases}$$

`pop` instruction

$$(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{pop}$$
$$\text{iff} \quad X :: Y \lhd \widehat{S}(m, pc) :$$
$$Y \sqsubseteq \widehat{S}(m, pc+1)$$
$$\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$$

`dup` instruction

$$(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{dup}$$
$$\text{iff} \quad A :: \widehat{S} \lhd \widehat{S}(m, pc)$$
$$A :: A :: S \sqsubseteq \widehat{S}(m, pc+1)$$
$$\widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)$$

`add` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{add} \\
&\quad \text{iff} \quad v_1 :: v_2 :: X \lhd \widehat{S}(m, pc) : \\
&\qquad \{\texttt{INT}\} :: X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`brtrue` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{brtrue}\ pc_t \\
&\quad \text{iff} \quad B :: X \lhd \widehat{S}(m, pc) : \\
&\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad X \sqsubseteq \widehat{S}(m, pc_t) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc_t)
\end{aligned}
$$

`stloc` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{stloc}\ x \\
&\quad \text{iff} \quad A :: X \lhd \widehat{S}(m, pc) : \\
&\qquad A \sqsubseteq \widehat{LV}(m, pc+1)(x) \\
&\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq_{\{x\}} \widehat{LV}(m, pc+1)
\end{aligned}
$$

`ldloc` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{ldloc}\ x \\
&\quad \text{iff} \quad \widehat{LV}(m, pc)(x) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`ldarg` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{ldarg}\ x \\
&\quad \text{iff} \quad \widehat{AL}(m, pc)(x) :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \widehat{AL}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`new` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{new}\ \sigma \\
&\quad \text{iff} \quad \{(\text{Ref}\ \sigma)\} :: \widehat{S}(m, pc) \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad \mathit{default}(\sigma) \sqsubseteq \widehat{H}(\text{Ref}\ \sigma) \\
&\qquad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`call` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{call}\ m_0 \\
&\quad \text{iff} \quad A_1 :: \cdots :: A_{|m_0|} :: X \lhd \widehat{S}(m, pc) : \\
&\qquad A_1 :: \cdots :: A_{|m|} \sqsubseteq \widehat{AL}(m_0, 0)[0..|m|-1] \\
&\qquad\quad m_0.returnType = \texttt{void} \Rightarrow \\
&\qquad\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad m_0.returnType \neq \texttt{void} \Rightarrow \\
&\qquad\qquad A \lhd \widehat{S}(m_0, \text{END}) \\
&\qquad\qquad A :: X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

`callvirt` instruction

$$
\begin{aligned}
&(\widehat{SH}, \widehat{H}, \widehat{LV}, \widehat{AL}, \widehat{S}) \models_{CFA} (m, pc) : \texttt{callvirt}\ m_0 \\
&\quad \text{iff} \quad B :: A_1 :: \cdots :: A_{|m_0|} :: X \lhd \widehat{S}(m, pc) : \\
&\qquad \forall (\text{Ref}\ \sigma) \in B : \\
&\qquad\quad m_v \lhd methodLookup(m_0, \sigma) \\
&\qquad\quad A_1 :: \ldots A_{|m_0|} \sqsubseteq \widehat{AL}(m_v, 0)[0..|m_0|-1] \\
&\qquad\quad \{(\text{Ref}\ \sigma)\} \sqsubseteq \widehat{LV}(m_v, 0)[0..|m_0|-1] \\
&\qquad\quad m_0.returnType = \texttt{void} \Rightarrow \\
&\qquad\qquad X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad m_0.returnType \neq \texttt{void} \Rightarrow \\
&\qquad\qquad A :: Y \lhd \widehat{S}(m_v, \text{END}) : A :: X \sqsubseteq \widehat{S}(m, pc+1) \\
&\qquad\quad \widehat{LV}(m, pc) \sqsubseteq \widehat{LV}(m, pc+1)
\end{aligned}
$$

Appendix D

All SCIL Instructions

Instruction:	Description:
	Arithmetic
`add`	Add two numbers
`sub`	Subtract two numbers
`mul`	Multiply two numbers
`div`	Divide two numbers
`rem`	Get division remainder
`clt`	Compare less than
`cge`	Compare greater than
`ceq`	Compare equal
`and`	Bitwise and
`or`	Bitwise or
`xor`	Bitwise xor
	Branching
`brtrue`	Branches when top of stack is `true`
	Arguments
`ldarg`	Load argument onto the stack
`ldarga`	Get address of argument
`starg`	Store value to argument
	Field
`ldfld`	Push value from field to the stack
`ldflda`	Push the address of field to the stack
`ldftn`	Push a pointer to method referenced by method to the stack
`ldsfld`	Push the value of static field to the stack
`ldsflda`	Push the address of static field to the stack
`stfld`	Replaces the value of field with value
`stsfld`	Replaces the value of static field with value
	Local Variables
`stloc`	Pop a value and store it in local variable
`ldloc`	Load value from local variable onto the stack
`ldloca`	Load address of local variable onto the stack
	Constants
`ldc`	Load constant of type `long`, `double` or `string`
`ldstr`	Load constant of type `string`
	Methods
`call`	Call a static method
`callvirt`	Call a virtual method
`ret`	Return from a metod
`SetResult`	Used for async tasks to set the result of the async process
`GetResult`	Used for async task to get the result of async process
	Miscellaneous

`pop`	Pops a value from the stack
`dup`	Duplicates a value on the stack
`nop`	No operation

Table D.1: *All SCIL instructions*

Bibliography

[1] European Parliament, Council of the European Union, European Commission. (Apr. 2016). EU Regulation 2016/679, [Online]. Available: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en (visited on 02/08/2018).

[2] S. Guthrie, *Microsoft to acquire Xamarin and empower more developers to build apps on any device*. [Online]. Available: https://blogs.microsoft.com/blog/2016/02/24/microsoft-to-acquire-xamarin-and-empower-more-developers-to-build-apps-on-any-device/ (visited on 02/13/2018).

[3] M. Christensen, S. Bjergmark, T. Nielsen, and S. Larsen, "Scanning for Vulnerabilities in Xamarin Apps", Distributed, Embedded and Intelligent Systems, Department Of Computer Science, Aalborg University, Jan. 2018.

[4] S. Inc., *Number of available applications in the Google Play Store from December 2009 to December 2017*. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (visited on 06/14/2018).

[5] P. Ferrara and F. Spoto, "Static Analysis for GDPR Compliance", 2018.

[6] European Parliament, Council of the European Union, "DIRECTIVE 95/46/EC OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data", *Official Journal*, vol. OJ L 119, 1995. [Online]. Available: http://eur-lex.europa.eu/eli/dir/1995/46/oj (visited on 02/09/2018).

[7] A. Cavoukian, "Privacy by design: The 7 foundational principles", Jan. 2011. [Online]. Available: https://www.ipc.on.ca/wp-content/uploads/Resources/pbd-implement-7found-principles.pdf (visited on 02/14/2018).

[8] E. Bodden, *FlowDroid – Taint Analysis*. [Online]. Available: https://blogs.uni-paderborn.de/sse/tools/flowdroid/ (visited on 02/09/2018).

[9] S. Arzt, S. Rasthofer, C. Fritz, *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps", *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014, ISSN: 0362-1340. DOI: 10.1145/2666356.2594299. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594299.

[10] C. Fritz, S. Arzt, S. Rasthofer, *et al.*, "Highly Precise Taint Analysis for Android Applications", no. TUD-CS-2013-0113, 2013.

[11] M. Project, *Gendarme*. [Online]. Available: http://www.mono-project.com/docs/tools+libraries/tools/gendarme/ (visited on 06/04/2018).

[12] Statista, *What operating system does your smartphone use?* [Online]. Available: https://www.statista.com/statistics/716053/most-popular-smartphone-operating-systems-in-us/ (visited on 05/25/2018).

[13] Ryantzj, *Android Application/Package APK Structure Part 1*. [Online]. Available: http://www.ryantzj.com/android-applicationpackage-apk-structure-part-1.html (visited on 05/22/2018).

[14] A. Pedley, *How Xamarin.Android AOT Works*. [Online]. Available: https://xamarinhelp.com/xamarin-android-aot-works/ (visited on 05/22/2018).

[15] Android, *APK Signature Scheme v2*. [Online]. Available: https://source.android.com/security/apksigning/v2 (visited on 05/22/2018).

[16] Oracle, *JAR File Specification*. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Signed_JAR_File (visited on 05/22/2018).

[17] A. O. S. Project, *ABI Management*. [Online]. Available: https://developer.android.com/ndk/guides/abis.html (visited on 05/22/2018).

[18] X. Inc., *Using Android Assets*. [Online]. Available: https://developer.xamarin.com/guides/android/application_fundamentals/resources_in_android/part_6_-_using_android_assets/ (visited on 01/03/2018).

[19] D. Griffiths, *How Android Apps are Built and Run*. [Online]. Available: https://github.com/dogriffiths/HeadFirstAndroid/wiki/How-Android-Apps-are-Built-and-Run (visited on 05/22/2018).

[20] A. O. S. Project, *Application Fundamentals*. [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro (visited on 05/22/2018).

[21] ——, *Application Fundamentals*. [Online]. Available: https://developer.android.com/guide/components/fundamentals.html (visited on 05/25/2018).

[22] ——, *Intent*. [Online]. Available: https://developer.android.com/reference/android/content/Intent.html (visited on 05/25/2018).

[23] ——, *Intents and Intent Filters*. [Online]. Available: https://developer.android.com/guide/components/intents-filters.html (visited on 05/25/2018).

[24] ——, *Understand the Activity Lifecycle*. [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle (visited on 05/25/2018).

[25] Xamarin, *Introduction to Mobile Development*. [Online]. Available: https://developer.xamarin.com/guides/cross-platform/getting_started/introduction_to_mobile_development/ (visited on 02/23/2018).

[26] Microsoft, *Part 1 – Understanding the Xamarin Mobile Platform*. [Online]. Available: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform (visited on 06/11/2018).

[27] ——, *Preparing an Application for Release*. [Online]. Available: https://docs.microsoft.com/en-gb/xamarin/android/deploy-test/release-prep/?tabs=vswin (visited on 06/11/2018).

[28] Xamarin, *Architecture*. [Online]. Available: https://developer.xamarin.com/guides/android/under_the_hood/architecture/ (visited on 02/21/2018).

[29] Microsoft, *What Is Managed Code?* [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx (visited on 02/20/2018).

[30] Xamarin, *Android Callable Wrappers*. [Online]. Available: https://developer.xamarin.com/guides/android/advanced_topics/java_integration_overview/android_callable_wrappers/ (visited on 02/21/2018).

[31] C. N. Fischer, R. K. Cytron, and R. J. LeBlanc, *Crafting A Compiler*, 1st. USA: Addison-Wesley Publishing Company, 2009, ISBN: 0136067050, 9780136067054.

[32] M. Sheppard, *What is reflection and why is it useful?* [Online]. Available: https://stackoverflow.com/a/37632 (visited on 05/14/2018).

[33] S. Poeplau, Y. Fratantonio, A. Bianchi, *et al.*, *Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications*, Jan. 2014.

[34] S. Chong, *Static single assignment form (and dominators, post-dominators, dominance frontiers...)* University Lecture, 2010. [Online]. Available: https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec04-SSA.pdf (visited on 05/16/2018).

[35] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow", *Commun. ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977, ISSN: 0001-0782. DOI: 10.1145/359636.359712. [Online]. Available: http://doi.acm.org/10.1145/359636.359712.

[36] W. Le, *Taint analysis*. [Online]. Available: http://web.cs.iastate.edu/~weile/cs513x/2018spring/taintanalysis.pdf (visited on 05/17/2018).

[37] zynamics BinNavi, *Reil - the reverse engineering intermediate language*. [Online]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm (visited on 05/19/2018).

[38] R. Hansen, "Flow logic for language-based safety and security", Jun. 2018.

[39] E. R. Wognsen and H. S. Karlsen, "Static Analysis of Dalvik Bytecode and Reflection in Android", Distributed, Embedded and Intelligent Systems, Department Of Computer Science, Aalborg University, Jun. 2012.

[40] Microsoft, *Bundle Assemblies into Native Code*. [Online]. Available: https://docs.microsoft.com/en-us/xamarin/android/deploy-test/release-prep/?tabs=vswin#bundle-assemblies-into-native-code (visited on 06/11/2018).

[41] ——, *OpCodes.Beq Field*. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.beq(v=vs.110).aspx (visited on 06/11/2018).

[42] ——, *OpCodes.Ceq Field*. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.ceq(v=vs.110).aspx (visited on 06/11/2018).

[43] ——, *Task-based Asynchronous Pattern (TAP)*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap (visited on 06/11/2018).

[44] ——, *Task-based Asynchronous Programming*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming (visited on 06/11/2018).

[45] ——, *Asynchronous programming*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/async (visited on 06/11/2018).

[46] S. T. Microsoft, *Dissecting the async methods in C#*. [Online]. Available: https://blogs.msdn.microsoft.com/seteplia/2017/11/30/dissecting-the-async-methods-in-c/ (visited on 06/11/2018).

[47] D. Yan, *Understanding C# async / await (1) Compilation*. [Online]. Available: https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-1-compilation (visited on 06/11/2018).

[48] Microsoft, *Task Parallel Library (TPL)*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (visited on 06/11/2018).

[49] ——, *Task-based Asynchronous Pattern (TAP) - Task Status*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap#task-status (visited on 06/11/2018).

[50] ——, *Task.Start Method ()*. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd270682(v=vs.110).aspx (visited on 06/11/2018).

[51] ——, *AsyncTaskMethodBuilder.AwaitUnsafeOnCompleted<TAwaiter, TStateMachine> Method (TAwaiter, TStateMachine)*. [Online]. Available: https://msdn.microsoft.com/en-us/library/hh472363(v=vs.110).aspx (visited on 06/11/2018).

[52] Flix, *The flix programming language*. [Online]. Available: http://flix.github.io/doc/ (visited on 02/22/2018).

[53] M. Madsen and E. Andreasen, "String Analysis for Dynamic Field Access", in *Compiler Construction (CC)*, 2014.

[54] D. Yan, *C# 6.0 Exception Filter and when Keyword*. [Online]. Available: https://weblogs.asp.net/dixin/c-6-0-exception-filter-and-when-keyword (visited on 06/13/2018).

[55] Damien_The_Unbeliever, *How are CIL 'fault' clauses different from 'catch' clauses in C#?* [Online]. Available: https://stackoverflow.com/a/11988377 (visited on 06/13/2018).

List of Figures

List of Tables

Listings