SPARTA!
Static Program Analysis for Reliable Trusted Apps

http://types.cs.washington.edu/sparta/

Version 0.6 (11 Sep 2012)

Do not distribute.

Contents

1 Introduction **3**
1.1 In case of trouble . . . 3

2 Installation **4**
2.1 Compiling the SPARTA code . . . 4

3 Android App Analysis **6**
3.1 Task: set up the tools for the new application . . . 6
3.2 Task: get a basic understanding of the application . . . 7
3.3 Task: see where permissions are used in the application . . . 7
3.4 Task: see what precise APIs are used where in the application . . . 7
3.5 Task: ensure that the used APIs are annotated with flow information . . . 7
3.6 Task: visualize the existing flow in the application . . . 8
3.7 Task: check the flow in the application . . . 8

4 Flow Checker **9**

5 Notes **12**
5.1 JSR 308 and Eclipse . . . 12
5.2 Annotating libraries . . . 13
5.3 Suppressing warnings . . . 13

6 SPARTA internals **14**

Chapter 1

Introduction

SPARTA is a project at the University of Washington funded by the DARPA APAC (Automated Program Analysis for Cybersecurity) program.

SPARTA aims to detect certain types of malware in Android applications, or to verify that the app contains no such malware. SPARTA's verification approach is type-checking: the developer states a security property, annotates the source code with type qualifiers that express that security property, then runs a pluggable type-checker [PAC+08, DDE+11] to verify that the type qualifiers are right (and thus that the program satisfies the security property).

The Checker Framework is a pluggable type-checker that provides the foundation for the SPARTA project. For more information on pluggable type-systems, please consult the Checker Framework manual at `http://types.cs.washington.edu/checker-framework/`.

Future updates to this manual will be posted to the public SPARTA project webpage at `http://www.cs.washington.edu/sparta/release/`.

1.1 In case of trouble

If you have trouble, please email either
`sparta@cs.washington.edu` (developers mailing list) or
`sparta-users@cs.washington.edu` (users mailing list) and we will try to help.

Chapter 2

Installation

This chapter briefly describes how to install the SPARTA project tools.

2.1 Compiling the SPARTA code

1. Required programs:
   - Download and install Java 7. Add `.../jdk1.7.0/bin` to the beginning of the PATH variable and set `JAVA_HOME` to `.../jdk1.7.0`.
   - Ensure you have recent versions of ant and Mercurial (hg) installed; ant version 1.8.2 and Mercurial version 2.0.2 are known to work.
   - Install the Android SDK to some directory. Set `ANDROID_HOME` to that location. Download the `android-16` target.
   - If using Eclipse, go to `Help` → `Install New Software` and install the Android ADT Plugin (`https://dl-ssl.google.com/android/eclipse`) and MercurialEclipse (`http://cbes.javaforge.com/update`).
2. Install the Checker Framework tool set.
   Linux is our development platform, but we know of successful uses on MacOS; Windows is possible, but takes more effort.
   Follow the instructions at:
   `http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#build-source`
   Note: As of 24 August 2012, the annotation-tools test cases produce some error output. This is expected and can be ignored.
3. Install the SPARTA tools
   - Download the SPARTA code from `http://types.cs.washington.edu/sparta/release/`. (Please do not publicize this URL.)
   - Unpack the archive.
   - Google gson is a dependency for the "-json" targets for projects. Get it from `http://code.google.com/p/google-gson/`; create directory `sparta-code/lib` and unzip gson there. Alternatively, set build property `gson.jar`, which defaults to:
     `gson.jar=$basedir/lib/google-gson-2.2.2/gson-2.2.2.jar`
   - Build the `sparta.jar` file.
     In the `sparta-code` directory, run
     `ant jar`
     It is necessary to specify the Android SDK location, by setting the `ANDROID_HOME` environment variable or the `android.home` property. Here is an example of the latter:
     `ant -Dandroid.home=... jar`
     See file `build.properties` for other configuration properties.

See the output of `ant -p` for the build and test targets.
All projects can also be built and tested in Eclipse. Import the annotation-tools, jsr308-langtools, checkers, javaparser, and sparta-code projects into a workspace.

4. Run test cases.
   As a sanity check of the installation, run
   `ant all-tests`
   You should see "`BUILD SUCCESSFUL`" at the end.

Chapter 3

Android App Analysis

This is a brief guide on how to analyze a new Android application and find security vulnerabilities. This document gives a high-level view of the process, including how various tools relate to one another.

For details about how to install and run the code analysis tools, see Chapter 2.

3.1 Task: set up the tools for the new application

Checking applications

It is required to set three environment variables:

- `CHECKERS` pointing to the `.../checker-framework/checkers` directory
- `SPARTA_CODE` pointing to the `.../sparta-code` directory
- `ANDROID_HOME` pointing to the `.../android-sdk` directory

Update the project with your Android settings:

```
$ ant -buildfile $SPARTA_CODE/build.local.xml
```

Alternatively, you can run:

```
$ $ANDROID_HOME/tools/android update project --path . --target android-15
```

Edit the `build.xml` file of the project under analysis to add the SPARTA build targets at the end (right before the "</project>"):

```
<property environment="env"/>
<dirname property="checkers_dir" file="$env.CHECKERS"/>
<basename property="checkers_base" file="$env.CHECKERS"/>
<dirname property="sparta-code_dir" file="$env.SPARTA_CODE"/>
<basename property="sparta-code_base" file="$env.SPARTA_CODE"/>
<import file="$sparta-code_dir/$sparta-code_base/build.include.xml"/>
```

To use Eclipse to look at and build the code, perform these simple steps:

- Using Eclipse, import the projects (this requires the app to have a .project and .classpath file)
  - Make sure `Project Properties` → `Android` → `Android version #` is checked
  - Check that `Project Properties` → `Java Build Path` → `Libraries` → `Android version #` appears
  - Add the sparta-code project to `Project Properties` → `Java Build Path` → `Projects`

- Compile via command line (`ant clean`, `ant flowtest`)
- If it compiles, or the errors are exclusively about annotations, it's working correctly.

Most Android apps will rely on an auto-generated `R.java` file in the `/gen` directory of the project. This will only be generated if there are no errors in the project. There may be errors in the resources (`.../res` directory) that could cause `R.java` to not be generated.

Additionally, if the app depends on an external `.jar` file (often located in the `lib/` directory), it will compile in Eclipse but not with Ant. To fix this, in ant.properties, add "`jar.libs.dir=lib`" (or wherever the `.jar` is located).

3.2 Task: get a basic understanding of the application

Look at the AndroidManifest.xml file and:

- Determine which permissions the app uses. Look for "uses-permission" entries to understand the used permissions.
- Determine the entry points into the source code. (This may also give a hint about the architecture or overall modular structure of the application.) Look for "activity", "intent-filter", "service", "receiver", and "provider" to see the entry points, intent messages it responds to, etc.

3.3 Task: see where permissions are used in the application

Run `ant reqperms` on the project to see a list of the app's methods that use calls that require certain Android permissions. You can use this to gain an understanding of where sensitive information may come from/go to in the application. You can remove the warning by writing `@RequiredPermissions(android.Manifest.permissions.|specific permission|)` in front of the method header in the source code.

Once all methods in the subject application are correctly annotated, `ant reqperms` will not issue any warnings. Grep for `@RequiredPermissions` to find the required permissions in the application.

3.4 Task: see what precise APIs are used where in the application

Run `ant reportusage`, `ant reportusage-json`, or `ant reportusage-text` to get a report of the used APIs. `ant reportusage` only reports about some Android APIs. The reported APIs are specified in two files: `sparta-code/src/sparta/checker` and `reflection.astub` Additional `.astub` files can be passed as `-Astubs=...` argument in the build file.

3.5 Task: ensure that the used APIs are annotated with flow information

This step is necessary until we have good coverage of all APIs. For every API method used by this app (i.e., those output by `ant reportusage`): If the method appears in file `sparta-code/src/sparta/checkers/flow.astub`, you have nothing to do. Read the Javadoc, decide what flow properties the method has. If it has flow properties, add the method to the flow.astub file. Specify the flow properties by writing annotations in the flows.astub file such as `@FlowSinks(FlowSink.NETWORK)` or `@FlowSources(FlowSource.ESP_FROM_ALIENS)`.

Methods that were checked but deemed to not have any flow should be marked with a `@NoFlow` declaration annotation. This annotation is short-hand for annotating the return type and each parameter as not having any flow. By marking methods as having no flow, the next person looking at that API does not need to perform the same analysis again.

We do not expect that any API method will contain both a `@FlowSources` and a `@FlowSinks` annotation. This task is onerous now, but as more APIs get annotated, it will become relatively quick because few APIs will be first used by each new app.

Note: we have not yet added any annotations to the app itself.

3.6 Task: visualize the existing flow in the application

Run `ant flowshow` or `ant flowshow-json` to get a report of the existing flow. For every type use in the application, it indicates the flow sources and sinks for that variable. This is exactly the annotations written in the program, plus possibly some additional annotations that are inferred by the Checker Framework. The `-json` version creates file flowshow.json that can be visualized separately.

This step does not perform type-checking; it only visualizes the flow information written in the program or libraries as annotations, or inferred from those annotations.

The report is interesting exactly when a given type use contains both a `@FlowSource` and a `@FlowSink` annotation. This is an information flow, and the analyst must decide whether it is desirable or undesirable.

For an unannotated program, the report will not be informative: it only contains API annotations that are propagated to local variables. The report will become more informative as you add more and more annotations to the application. So, you can periodically rerun this step. (You don't have to run this step – you can skip it if you prefer.)

3.7 Task: check the flow in the application

Run `ant flowtest` on the application. Eliminate the warnings in one of two ways.

Add annotations to method signatures and fields in the application, as required by the type-checker. This essentially propagates the flow information required by the APIs through the application.

Use `@SuppressWarnings` to indicate safe uses that are safe for reasons that are beyond the capabilities of the type system. Always write a comment that justifies that it is safe.

A prime example is a String literal that should be allowed to be sent over the network. By default, every literal has `@FlowSinks()` (i.e., nothing) and `@FlowSources()`.

```
// Validated String literal, with no query parameters
// that have my credit card number.
@SuppressWarnings("flow") // manually checked
@FlowSinks(FlowSink.NETWORK) String url = "http://bazinga.com/";
```

Without suppression this must raise an error, as some unqualified information may go to the network. By adding the suppression, you assert that it's OK to send that information.

Focus on the most interesting flow sources and try to connect the flow sources and sinks in the application. Instead of trying to completely annotate only the sources or only the sinks, skim over all the reports and use your intuition to decide which parts of the application to focus on. Try to focus on the parts with the (most) connections between sources and sinks.

Most types will only use either a `@FlowSources` or `@FlowSinks` annotation. The goal is to find places where you need both annotations, e.g. to express that information that comes from the camera may go to the network:

```
@FlowSources(FlowSource.CAMERA)
@FlowSinks(@FlowSink.NETWORK) Picture data;
```

Such a type connects sources and sinks and one needs to carefully decide whether this is a desired information flow or not.

- If this is good information flow, then write both the `@FlowSources` and the `@FlowSinks` annotations at the same place. You will not receive any more error messages, but you can find all these places with grep or (better) with `ant flowshow`.
- If this is bad information flow, then either leave it unannotated, or annotate it but record both in the source code and elsewhere that you have found a security flaw.

You can continue to use `ant flowshow` to visualize the annotation progress.

Once all warnings were resolved, run `ant -Dstricter=true flowtest` on the application. Providing the `stricter` option enables additional checks that are required for soundness, but would be disruptive to enable initially. In particular, the tests for casts and array subtyping are stricter. See the discussion in Chapter 4, page 9.

Chapter 4

Flow Checker

This chapter gives a brief overview of the Flow Checker, one of the main components of SPARTA. The Flow Checker consists of a pluggable type-system written for and enforced by the Checker Framework. For more information please consult the Checker Framework manual at `http://types.cs.washington.edu/checker-framework/`.

Overview Each type consists of a flow source and a flow sink qualifier and each qualifier consists of a set of possible source/sink enum constants, the empty set, or the special `ANY` source/sink.

See classes `sparta.checkers.quals.FlowSources.FlowSource` and `sparta.checkers.quals.FlowSources.FlowSource` for the definition of the possible sources/sinks.

As an example, the type `@FlowSources(FlowSource.LOCATION) double` marks that the value might be tainted by location information. Similarly, the type `@FlowSinks(FlowSink.NETWORK) Data` marks that the value might flow to the network.

Subtyping See Figure 4.1 for an example hierarchy of source qualifiers and Figure 4.2 for sink qualifiers.

For flow sources, `FlowSources(FlowSource.ANY)` builds the top of the type hierarchy. Any other flow source is subsumed by it, as it is safe to conservatively claim more possible sources of information. Subsets of source enums build subtypes. The empty set is the bottom type.

For flow sinks, `FlowSinks({})` — that is, the empty set of flow sinks — builds the top of the type hierarchy. A reference that is not allowed to flow anywhere can safely refer to an object that would have been able to flow somewhere. A type with additional sinks builds a subtype. The type `FlowSinks(FlowSinks.ANY)` builds the bottom type.

Note the different subtyping behavior for sources and sinks.

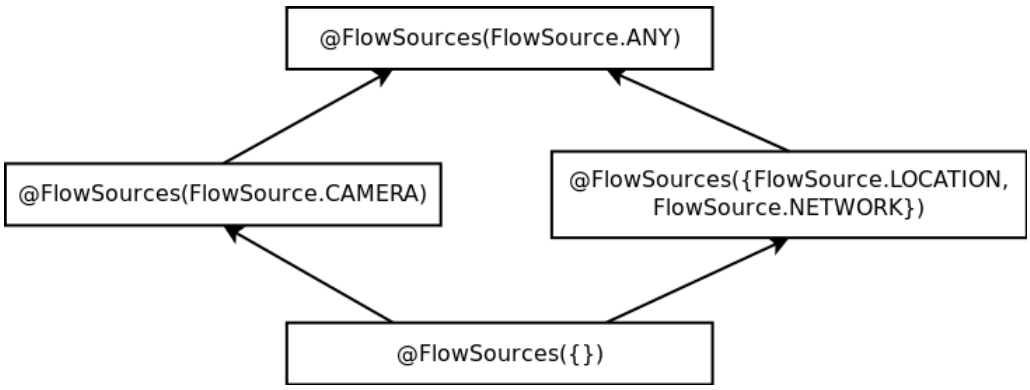

Figure 4.1: Qualifier hierarchy for four possible flow sources.

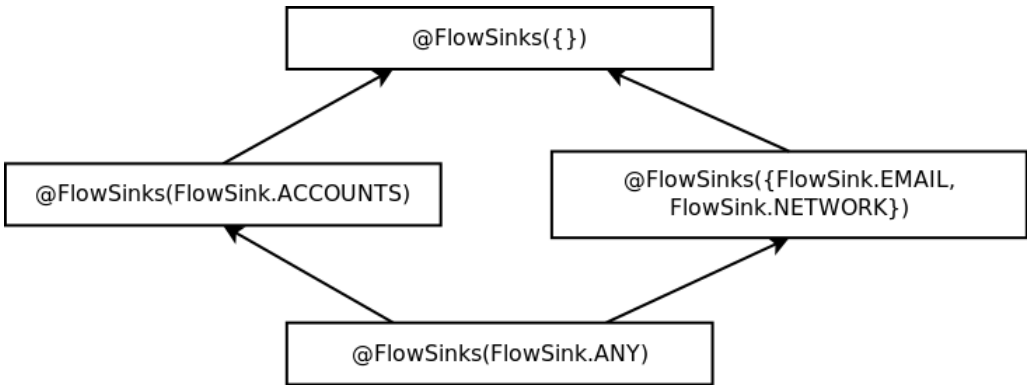

Figure 4.2: Qualifier hierarchy for four possible flow sinks.

Defaults In field and method signature types, the default source type is the empty set and the default sink qualifier is the empty set. The Checker Framework supports flow-sensitive type refinement. The type of local variables is the top type for each hierarchy, that is, any flow source and no flow sinks. The initializer of a variable is used to refine that type to something more concrete.

String literals and all primitive literals (1.0, true, 3, 'c', etc.) have as default `@FlowSources() @FlowSinks()`, that is, no flow sources and no flow sinks. A variable declaration:

```
@FlowSinks(FlowSink.NETWORK) String realString = "asdf";
```

will raise an error, because an unqualified type flows to a sink. After verifying that this is legal, one can suppress the warning, using `@SuppressWarnings("flow")`.

Qualifier polymorphism Two additional type qualifiers exist to express type polymorphism: `@PolyFlowSources` and `@PolyFlowSinks`. These range over their respective type hierarchy and can be used to express type dependencies, for example, that the receiver and parameter types have to correspond. The invocation of a method is used to resolve polymorphic qualifiers to something concrete.

API specifications File `sparta-code/src/sparta/checkers/flow.astub` gives the API specification for the Flow Checker. Besides the qualifiers discussed so far, two additional declaration annotations can be used here: `@NoFlow` and `@ConservativeFlow`. These annotations can be used on any declaration: a method, a class, or even a whole package. For example, in the beginning of `flow.astub` we declare that the whole `android` package should use conservative defaults. More specific annotations given in the rest of the file override these defaults.

Annotation `@NoFlow` expresses that the method has no flow sources and no flow sinks — in the return type and all parameter types. It is used to mark methods/classes/packages that have been analyzed and deemed safe.

Annotation `@ConservativeFlow` uses `FlowSources(FlowSource.ANY)` as the return type of the annotated element. This return type ensures that any use of the method will be tainted as possibly coming from an arbitrary source. The analyst therefore very easily can track down the used API methods, and analyze and annotate the used subset.

Stricter tests The default checks ensure soundness of the flow qualifiers. It ignores two possible sources of unsoundness: covariant array subtyping and downcasts that are not checked at runtime. After getting the basic checks passing, the stricter checks should be enabled. This two-phase approach was chosen to not interfere with the annotation effort too severely and to give two separate phases of the annotation effort.

When strict checks are turned on, a cast `(Object []) x`, were `x` is of type `Object`, will result in a compiler warning:

```
[jsr308.javac] ... warning: "@FlowSinks @FlowSources(FlowSource.ANY) Object"
       may not be casted to the type "@FlowSinks @FlowSources Object"
```

The reason is that statically the component type of the array is simply defaulted. There is no static knowledge about the actual runtime values in the array and important flow could be hidden. The analyst should argue why the downcast is safe.

Note that the main qualifier of a cast is automatically flow-refined by the cast expression.

Stricter checking also enforces invariant array subtyping, which is needed for sound array behavior in the absence of runtime checks. Flow inference automatically refines the type of array creation expressions depending on the LHS.

Miscellaneous Methods like `equals()` and `toString()` that are inherited from `Object` are the most general possible, so that overriding methods can restrict the annotations further. Thus, they return `FlowSource.ANY` and no flow sinks.

Binary operations like string concatenation or integer addition, result in the least upper bound of the two component types.

Chapter 5

Notes

This chapter is currently disorganized notes that will be incorporated into either this manual or the Checker Framework manual.

[Note: `FlowSources(...)` is shorthand noting that the specific flow properties aren't relevant.]

5.1 JSR 308 and Eclipse

JSR 308 extends the Java language to allow annotations in more locations. Java 8 will include support for this extended syntax. The Checker Framework builds on a version of OpenJDK that already supports this syntax. However, existing compilers do not support this syntax and will raise an error. This is an issue in particular if you want to analyze applications in Eclipse.

Eclipse accepts annotations only in the locations supported by Java 1.5, that is, only declaration locations; examples are fields, local variables, parameters, and return types.

Eclipse won't accept annotations in locations that were added in JSR 308; added locations include type arguments, object creation, casts, type parameter bounds, and others. To avoid syntax errors from Eclipse or other Java compilers, you need to put such annotations in comments.

Single qualifiers can use the "annotation-in-comment" syntax `/*@X*/`. Examples:

- Generics: `List</*@FlowSources(...)*/ String>`
- Object creation: `new /*@FlowSinks(...)*/ Object();`
- Casts: `return (/*@FlowSources(...)*/ double) x;`

Note that only a single annotation can be in a comment; for multiple annotations, use multiple comments, as in `/*@X*/ /*@Y*/`.

Also, in Java 8, method receivers can be explicitly given as a `this` parameter. A separate comment syntax can be used to hide method receivers from Eclipse, while still keeping them available to the Checker Framework:

```
public Class(/*>>> @FlowSinks(FlowSink.NETWORK) Class this, */ String paramTwo)
```

When using annotations in comments, Eclipse might warn about unused imports for these annotations. To prevent such warnings, similarly put such imports into comments:

```
/*>>>
import sparta.checkers.quals.*;
*/
```

Sometimes method type argument inference does not interact well with type qualifiers. In such situations, you might need to provide explicit method type arguments, for which the syntax is as follows:

```
Collections.</*@FlowSources(...)*/ Object>sort(l, c);
```

5.2 Annotating libraries

To support adding annotations to libraries, we provide special .astub files that contain the annotations on the signatures of such classes. These files contain Java classes, but provide a few special features:

- they can contain multiple packages and multiple public classes in one file,
- methods do not need to provide implementations,
- the return type of a method does not need to match the real method, e.g. using java.lang.Object is valid for every method.

There are two temporary difference when annotating stub files, rather than source code:

- the receiver is written after the method parameter list, instead of as an explicit first parameter. That is, instead of
  `returntype methodname(@Annotations C this, params);`
  in a stub file one has to write
  `returntype methodname(params) @Annotations;`
- enum constants in annotations need to be fully qualified. For example, one has to write the following
  `@FlowSources(sparta.checkers.quals.FlowSources.FlowSource.ANY)`
  even when the classes are correctly imported.

5.3 Suppressing warnings

Warnings can be suppressed using the standard `@SuppressWarnings` annotation. To suppress warnings from the Flow Checker, use `@SuppressWarnings("flow")`.

The `@SuppressWarnings` annotation cannot be used on statements or expressions, only on declarations. Warnings should be suppressed in the smallest possible scope and should always contain an explanation for why it is safe to ignore the warning.

The general pattern is to introduce a new local variable and suppress the warning on that local variable. The following is legal example for this:

```
@SuppressWarnings("flow") // Explain why suppression is sound.
@FlowSources(...) String a = method();
```

The following is not legal, because the @SuppressWarnings is not on a declaration:

```
@FlowSources(...) String a;
@SuppressWarnings("flow") // Explain why suppression is sound.
a = method();
```

Chapter 6

SPARTA internals

This document contains details that are only relevant for people inside the SPARTA team at UW.

The source code for the SPARTA checkers is in a Mercurial repository at `/projects/swlab1/darpa-apac/sparta-code`.

To get a copy do:

```
$ hg clone ssh://YOURID@SERVERNAME//projects/swlab1/darpa-apac/sparta-code
```

To push your changes to the repository you need to be in sparta group. Contact Werner or Mike to get the permission.

(Along with sparta-code, you may be interested in the sparta-meetings and apac-meetings repositories for general information on the SPARTA project.)

Note that SPARTA as well as the Checker Framework are evolving rapidly. Thus you may need to pull and update with updated source code and rebuild the projects:

```
$ hg fetch
```

Test applications are stored in `/projects/swlab1/darpa-apac/sparta-subjects`

Try to run ant in sparta-subjects/Sky

```
$ ant flowtest
```

If it gives results like this, you're ready to work on annotating!

```
[jsr308.javac] /home/syhan/jsr308/sparta-subjects/Sky/src/org/jsharkey/sky/WebserviceHelper.java:308: erro
[jsr308.javac]             HttpGet request = new HttpGet(String.format(WEBSERVICE_URL, lat, lon, days));
[jsr308.javac]                                                        ^
[jsr308.javac]   found   : @FlowSinks @FlowSources String
[jsr308.javac]   required: @sparta.checkers.quals.FlowSinks(sparta.checkers.quals.FlowSinks.FlowSink.NETWO
```

If you want to add a new application, put it under the `sparta-subjects` directory.

You may need to get Android source code to get sense of what api returns (or gets) what type of data. See `http://source.android.com/source/index.html` You can find the list of all apis from the android source code in `frameworks/base/api/15.txt` - api list for api version 15 (Android 4.0.3) Accessing resource is closely related to android permissions (some of the resources are not protected with permissions though). Android permission list is at: `http://developer.android.com/reference/android/Manifest.permission.html` Hints to add annotations could be permissionmap (which permission is required to call which functions): `http://www.android-permissions.org/permissionmap.html`

Adding Annotations There are two parts in adding annotations. Firstly, we need to annotate the flow stub file (`flow.astub`) in `sparta-code`. It's used for annotating Android APIs. For instance,

```
package android.telephony;

class TelephonyManager
    public @FlowSources(sparta.checkers.quals.FlowSources.FlowSource.PHONE_NUMBER) String getLine1Number()
    public @FlowSources(sparta.checkers.quals.FlowSources.FlowSource.IMEI) String getDeviceId();
```

The above annotates two methods in class `TelephonyManager`. It means that the `getLine1Number` function returns a String which is phone number. For more examples, look into the `flow.astub` file.

FlowSources specifies data sources such as phone number, location, and etc. FlowSinks specifies sinks, such as files, network, and so on. The types of FlowSources and FlowSinks are listed in the `FlowSources.java` and `FlowSinks.java` files in `sparta.checkers.quals`.

The second part is annotating the Android application to match the annotations specified in the stub file. Run flowtest and see if any incompatible types are shown and iteratively add more annotations to the application. The Android API specifications in flow.astub is currently incomplete. As you experience new APIs, extend the specifications.

Bibliography

[DDE+11] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.