

SPARTA!
Static Program Analysis for Reliable Trusted Apps

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

Version 0.7 (16 Oct 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	10
4.1	Overview	10
4.2	Subtyping	10
4.3	Defaults	11
4.4	Qualifier polymorphism	12
4.5	API specifications	12
4.6	Stricter tests	12
4.7	Miscellaneous	13
5	Rules of engagement	14
6	Notes	16
6.1	JSR 308 and Eclipse	16
7	SPARTA internals	17

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/>.

Updates to this manual will be posted to SPARTA releases 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

File `sparta-code/src/sparta/checkers/flow.astub` provides library annotations. You may need to enhance it, if you find that your application uses APIs that are not yet annotated. (The more APIs get annotated, the less work you will have to do in this step for each new app.) For details, see chapter “Annotating libraries” in the Checker Framework manual.

An example from `flow.astub` is:

```
package android.telephony;

class TelephonyManager {
    public @FlowSources(sparta.checkersquals.FlowSources.FlowSource.PHONE_NUMBER) String getLineNumber();
    public @FlowSources(sparta.checkersquals.FlowSources.FlowSource.IMEI) String getDeviceId();
}
```

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

For every API method used by this app (i.e., those output by `ant reportusage`) that does *not* already appear in file `sparta-code/src/sparta/checkers/flow.astub`, do the following.

- Read the Javadoc
- Decide what flow properties the method has.
- Add the method to the `flow.astub` file, with appropriate flow properties expressed as `@FlowSinks(...)` and `@FlowSources(...)` annotations. It would be unusual for an API method to contain both a `@FlowSources` and a `@FlowSinks` annotation.

If the method has no flow, you can write the `@FlowSources` and `@FlowSinks` annotations on each parameter and the return type, *or* you can write a single `@NoFlow` declaration annotation. This annotation is short-hand for annotating the return type and each parameter as not having any flow. It is important to mark API methods that have no flow, so that the method appears in the `flow.astub` file and the next person looking at that API does not need to perform the same analysis again.

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 10.

Chapter 4

Flow Checker

This chapter gives a brief overview of the Flow Checker, one of the main components of SPARTA. The Flow Checker tracks information flow through your program — it gives a guarantee that there is no information flow beyond what is expressed in type annotations.

The Flow Checker does pluggable type-checking of an information flow type system. It is implemented using the Checker Framework. To better understand pluggable type-checking and the Checker Framework, please consult the Checker Framework manual at <http://types.cs.washington.edu/checker-framework/>.

4.1 Overview

You write the annotation `@sparta.checkersquals.FlowSources` on a variable's type to indicate what sensitive sources can affect the variable's value. You write the annotation `@FlowSinks` to indicate where (part of) the value might be output.

As an example, suppose there is a declaration

```
@FlowSources(LOCATION) double latitude;
```

This indicates that the value of `latitude` might have been derived from location information. It does not guarantee that the value came from authentic GPS data, but only sets a bound on where the information might have come from; this assignment is legal: `latitude = 47.6097`.

Similarly, the type

```
@FlowSinks(NETWORK) String someData;
```

marks that the string `someData` might be output to the network, or a part of it or its length might be output to the network. It is also possible that the data has already been output.

The argument to `@FlowSources` (and `@FlowSinks`) is an enum constant, or a set of them to indicate that a value might combine information from multiple sources (or might flow to multiple locations).

`FlowSources` specifies data sources such as phone number, location, 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.checkersquals`.

4.2 Subtyping

As with standard Java types, the type annotation hierarchy indicates which assignments, method calls, and overriding are legal.

Figure 4.1 shows part of the `@FlowSources` qualifier hierarchy. The top type is `@FlowSources(ANY)`, which is shorthand for listing every source. It would be legal to annotate every variable in a program with `@FlowSources(ANY)`,

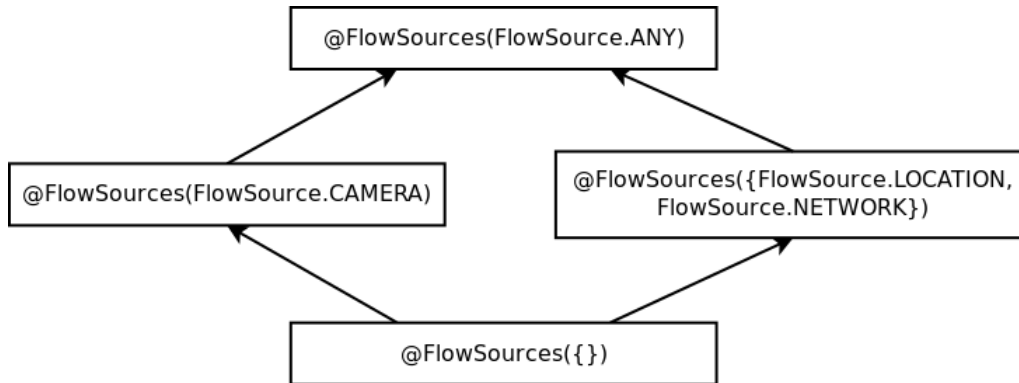


Figure 4.1: Partial qualifier hierarchy for @FlowSources.

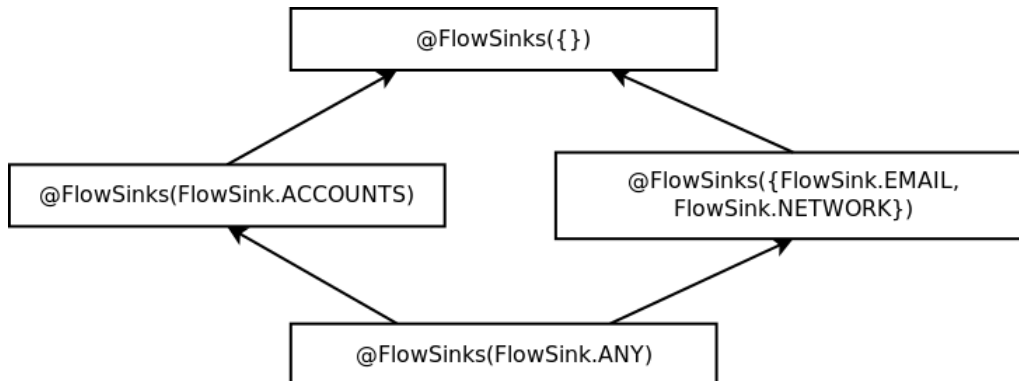


Figure 4.2: Partial qualifier hierarchy for @FlowSinks.

because every variable is derived from some subset of all flow sources. But, such an annotation would be imprecise. @FlowSources() is the bottom type, and may only be applied to variables whose value does not depend on any sensitive source.

Figure 4.2 shows part of the @FlowSinks qualifier hierarchy. The top type is @FlowSinks({}), which indicates that the value is only used locally by the application and never flows to any sensitive sink. The bottom type, @FlowSinks(FlowSink.ANY), is a value that might be output to any location whatsoever; it can be thought of as a completely public value.

Note the different subtyping behavior for sources and sinks.

4.3 Defaults

The default qualifiers are @FlowSinks() and @FlowSources(). This default applies to field types, method parameters and return types, manifest constants (string literals, 1.0, true, 3, 'c', ...), and all other locations except local variables.

This means that this variable declaration raises an error:

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

The reason is that the right-hand side of the assignment has type @FlowSinks(), which is a value that may not be output to any sink. This is not a subtype of the left-hand type @FlowSinks(NETWORK), which is allowed to flow to the network (but to no other sink). If the given string is permitted to flow to the network, then you can suppress the warning by annotating the declaration with @SuppressWarnings("flow").

The `@ConservativeFlow` annotation changes the default for return types; see Section 4.5.

For **local variables**, the default is the top type for each hierarchy, that is, `@FlowSources(ANY) @FlowSinks()`. This applies to the top level of the local variable type, but not to generic type arguments and array elements, if any.

The Checker Framework supports flow-sensitive type refinement. Assignments (such as initializers) are used to refine the type to a more precise one. Thus, in general you do not have to write type annotations on local variables. For details, see section “Automatic type refinement (flow-sensitive type qualifier inference)” in the Checker Framework manual.

4.4 Qualifier polymorphism

To make the type system more expressive, the flow type systems support qualifier polymorphism, via the type qualifiers `@PolyFlowSources` and `@PolyFlowSinks`. See section “Qualifier polymorphism” in the Checker Framework manual.

4.5 API specifications

File `sparta-code/src/sparta/checkers/flow.astub` provides library annotations. You may need to enhance it, if you find that your application uses APIs that are not yet annotated. For details, see section 3.5 of this manual, and also chapter “Annotating libraries” in the Checker Framework manual.

Besides the qualifiers discussed so far, two additional declaration annotations can be used in the stub file: `@NoFlow` and `@ConservativeFlow`. Their effect is to change the default annotations (Section 4.3).

Annotation `@NoFlow` expresses that each method’s return type and all parameter types are `@FlowSources()` `@FlowSinks()`. As with all library annotations, it is trusted rather than checked. Thus, it should be used only if an external analysis has determined that it is correct for the annotated method, class, or package.

Annotation `@ConservativeFlow` expresses that each method should have the most conservative possible annotations: `@FlowSources()` `@FlowSinks(ANY)` on arguments, and `@FlowSources(ANY) @FlowSinks()` on return values. [TODO: what for fields?] This is so conservative that it is sure to cause a type-checking failure whenever the method is used. When the analyst encounters such type-checking errors, the analyst can annotate the methods more appropriately. This is a way of knowing when a program uses a previously-unannotated library.

These annotations can be used on any declaration: a method, a class, or even a whole package. For example, the beginning of `flow.astub` declares that the whole `android` package should use conservative defaults. More specific annotations given in the rest of the file override these defaults.

4.6 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`, where `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 left-hand side.

4.7 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

Rules of engagement

The goal of an application developer is to create a safe, functional application — and to write the documentation and code so that the safety and functionality are immediately obvious. In particular, the code and documentation should be clear and complete, and the system should pass all the tests that the SPARTA toolset performs. If the application developer fails to meet any of these objectives, then the application will be rejected from the app store, and the fault will be with the application developer, not with the app store.

A malicious developer would need to write clear documentation and code, but would attempt to hide malicious behavior in the app nonetheless. If the documentation or code is not clear, or if the malicious behavior is not well-hidden, or if the SPARTA tools do not confirm that the code conforms to the documentation, then the malicious developer has failed in his task.

Note that the malicious developer's goal is more difficult than just writing malicious code, and is even more difficult than writing well-hidden malicious code. The reason is that the SPARTA toolset encourages good coding style: poor style requires more warning suppressions. The SPARTA tools lead a programmer to better, clearer code.

Here are some specific requirements of the app developer:

- Use good style. Code must not be obfuscated. Raw types must not be used. Minimize use of undesirable/un-sound features such as arrays, casts, heterogeneous collections, and reflection. Provide source code. Provide a build file (for Ant, Maven, Android, etc.).
- State the intended information flows in the application. This should be expressed both in precise English and also in a machine-readable format that can be read by the SPARTA tools. The English description should include how the information flows between parts of the application (the paths along which information flows), and the conditions under which it flows (such as only after a particular user action or external trigger). These will eventually be represented in the SPARTA toolset's file format and checked by SPARTA, but they are not yet.
- Annotate the application. Write type qualifiers on variables. This is essentially just a restatement of the information flows above at a lower and more detailed level.
- Type-check the application. Run the type-checker on its source code. Do not take advantage of any of the unsound features of the type-checker. (Those features are supported to reduce the workload of people who are not concerned about an absolute guarantee.) For example, do not skip any portions of the code
- Minimize the number of type-checking failures, and justify each one. A type-checking failure indicates either a bug (i.e., security flaw) in the application, or an instance of subtle code that is beyond the capabilities of the type system. In either case, the app developer's first inclination should be to rewrite the code to be correct and straightforward.

If rewriting the code is impossible, then every remaining warning should be suppressed with a `@SuppressWarnings` annotation. Every `@SuppressWarnings` annotation requires a clear, compelling justification regarding why the code is actually correct and safe (even though the type-checker cannot prove this property), and why the code cannot be rewritten to address the warning. This justification should be written in the source code at the location of the `@SuppressWarnings` annotation.

An excessive number of type-checker warnings, or missing justifications for warning suppressions, is grounds for rejection from the app store.

Chapter 6

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.]

6.1 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 methods (return type annotations go in the same place).

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. The SPARTA tools will interpret them, but Eclipse and other Java compilers will ignore them.

For details, see section “Annotations in comments” in the Checker Framework manual.

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);
```


Chapter 7

SPARTA internals

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

The SPARTA team uses four Mercurial repositories: `sparta-code` for the source code of the SPARTA toolset, `sparta-subjects` for test applications (case studies), `sparta-meetings` for notes about UW team meetings, and `apac-meetings` for notes about DARPA meetings.

To get a copy do:

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

and similarly for the other three repositories.

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

Note that SPARTA as well as the Checker Framework are evolving rapidly. Thus you should periodically get the latest version of the source code (by running `hg fetch`) and rebuild the projects.

After installing your copy, 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] .../sparta-subjects/Sky/src/org/jsharkey/sky/WebserviceHelper.java:308: error: incompatible types.  
[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.checkersquals.FlowSinks(sparta.checkersquals.FlowSinks.FlowSink.NETWORK) @FlowSources String
```

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>

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.