SPARTA!
Static Program Analysis for Reliable Trusted Apps

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

Version 0.9.1 (1 May 2013)

Do not distribute.

Contents

Chapter 1

Introduction

SPARTA is a research project at the University of Washington funded by the DARPA Automated Program Analysis for Cybersecurity (APAC) 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 the type qualifiers (and thus to verify that the program satisfies the security property).

You can find the latest version of this manual in the `sparta-code` version control repository, in directory `sparta-code/docs`. Alternately, you can find it in a SPARTA release at `http://types.cs.washington.edu/sparta/release/`, though that may not be as up-to-date.

1.1 Overview: malware detection and prevention tools

The SPARTA toolset contains two types of tools: reverse engineering tools to find potentially dangerous code in an Android app, and a tool to statically verify information flow properties.

The reverse engineering tools to find potentially dangerous code can be run on arbitrary unannotated Android source code. Those tools give no guarantees, however; they just direct the analyst's attention to suspicious locations in the source code.

By contrast, the tools to statically verify information flow require a person to write the information flow properties of the program, primarily as source code annotations. For instance, an object that contains data that came from the camera and is destined for the network would be annotated with

```
@FlowSources(CAMERA) @FlowSinks(NETWORK)
```

There are two different scenarios in which the SPARTA tools might be used.

- Ideally, the application vendor, who understands the source code, writes information flow annotations such as `@FlowSources` in the source code, iterating until the static information flow tool issues no warnings.
  In this case, the analyst merely re-runs the static information flow tool to confirm the vendor's work. This shows that there are no undesired information flows in the program.
  Chapter 4 explains how to use the SPARTA tools for this scenario.
- If the application vendor delivers an unannotated program, then the analyst must understand the program well enough to annotate it and then annotate it.
  In this case, it is most efficient to first run the reverse engineering tools to detect suspicious code. Those tools might reveal unacceptable code: either malware or code that the vendor should rewrite in a clearer or safer way. If the suspicious code detection tools do not reveal problems so severe that the app should be rejected, then they help to guide the next step. The analyst writes information flow annotations and runs the information flow tool until either the analyst has found a vulnerability or the lack of tool warnings indicates there is no vulnerability.
  Chapter 5 explains how to use the SPARTA tools for this scenario.

1.2 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 and app setup

This chapter describes how to install the SPARTA tools (Section 2.2) and how to prepare an Android App to have the SPARTA tools run on it (Section 2.3).

2.1 Requirements

Java 7

- `.../jdk1.7.0/bin` must be on your path.
- `JAVA_HOME` should be set to `.../jdk1.7.0`.

Ant

- Ant version 1.8.2 or later

Android SDK

- Install the Android SDK to some directory.
- Set `ANDROID_HOME` to the directory where you installed the Android SDK.
- Download the `android-15` 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`).

Checker Framework

- Follow the installation instructions in the manual: `http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#installation`
- As described in the installation instructions, set the `CHECKERS` environment variable to `.../checker-framework/checkers/`

2.2 Install SPARTA

1. Obtain the source code for the SPARTA tools, either from its version control repository or from a packaged release.
   - To obtain from the version control repository, run
     ```
     hg clone https://dada.cs.washington.edu/hgweb/sparta-code
     ```
     using the credentials you have been given.

- If you do not have access to the source code repository, then download the SPARTA release from `http://types.cs.washington.edu/sparta/release/`. (Please do not publicize this URL.)
  Then, unpack the archive.

2. Update the Android development environment, by running the following:

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

   Alternatively, you can run:

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

   Rationale: When working with Android, a developer must "update a project" to properly set the path to the Android SDK. For more details about updating an Android project, see `http://developer.android.com/tools/projects/projects-cmdline.html#UpdatingAProject`.
3. Build the SPARTA tools by compiling the source code:
   `ant jar`
4. As a sanity check of the installation, run
   `ant all-tests`
   You should see "`BUILD SUCCESSFUL`" at the end.

2.3 Android App Setup

This section explains how to set up an Android application for analysis with the SPARTA tools.

1. Ensure the following environment variables are set.
   - `CHECKERS` is the `.../checker-framework/checkers` directory
   - `SPARTA_CODE` is the `.../sparta-code` directory
   - `ANDROID_HOME` is the `.../android-sdk` directory
2. If your Android project does not have a build.xml file, update the project.

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

3. Add the SPARTA build targets to the end of the `build.xml` file, just above `</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" optional="true"/>
   ```

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).

Chapter 3

Flow Checker

This chapter describes the Flow Checker, a type-checker that tracks information flow through your program.

To use the Flow Checker, a programmer must supply two types of information:

- A flow policy that expresses what information flows the program is allowed to have. For example, a program might be allowed to send location information to the network, but not allowed to access contacts nor to send SMS messages. The flow policy is primarily derived from the program's user documentation. Section 3.2 describes how to write a flow policy.
- Type qualifiers written on (some of) the variables in the program. The type qualifiers indicate where the variable's value came from and where it might go to.

When you run the Flow Checker, it verifies that the annotations in the program are consistent with what the program's code does, and that the annotations are consistent with the flow policy. This gives a guarantee that the program has no information flow beyond what is expressed in the flow policy and type annotations.

The Flow Checker does pluggable type-checking of an information flow type system. It is implemented using the Checker Framework. This chapter is logically a chapter of the Checker Framework Manual (`http://types.cs.washington.edu/checker-framework/current/checkers-manual.pdf`). Therefore, in order to understand this chapter, you should first read chapters 1–2 of the Checker Framework Manual, and you should at least skim chapters 18–21 (generics through libraries) and 24–25 (FAQ and troubleshooting).

After you read this chapter, see Chapter 6 for tips about writing information-flow annotations.

3.1 The information-flow type system

You write the annotation `@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(FlowSource.LOCATION) @FlowSinks(FlowSink.NETWORK) double latitude;
```

The `@FlowSources(FlowSource.LOCATION)` annotation 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 annotation `@FlowSinks(FlowSInk.NETWORK)` marks that the string `latitude` 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

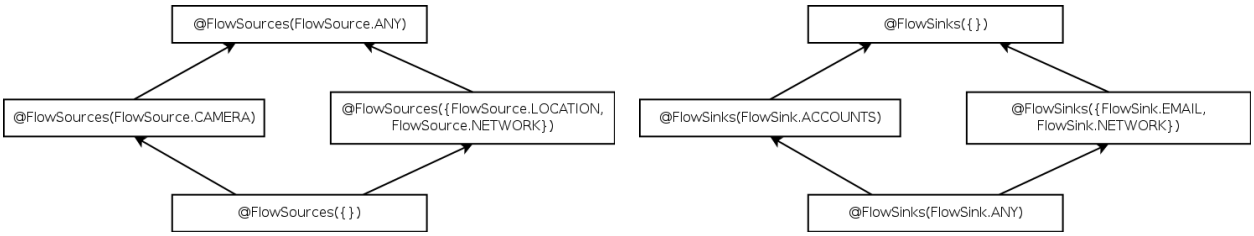

Figure 3.1: Partial qualifier hierarchy for `@FlowSources` and `@FlowSinks`.

`sparta.checkers.quals`. Most of them correspond to Android permissions (see `http://developer.android.com/reference/android/Manifest.permission.html`), and others are new "permissions" that provide finer-grained control over the behavior of an application.

3.1.1 Subtyping

As with standard Java types, the type annotation hierarchy indicates which assignments, method calls, and overriding are legal. Figure 3.1 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)`, because every variable is derived from some subset of all flow sources. But, such an annotation would be imprecise. `@FlowSources({})` or `@FlowSources()` is the bottom type, and may only be applied to variables whose value does not depend on any sensitive source.

Figure 3.1 shows part of the `@FlowSinks` qualifier hierarchy. The top type is `@FlowSinks({})` or `@FlowSinks()`, which indicates that the value is only used locally by the application and never flows to any sensitive sink. The bottom type, `@FlowSinks(FlowSinks.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.

3.1.2 Conditionals

The Flow Checker considers conditional expressions to be a flow sink. If a variable will be used in a conditional, then it must have a flow sink of CONDITIONAL. By default, any source is allowed flow through a conditional. That is to say that ANY $\rightarrow$ CONDITIONAL is added to the flow policy by default.

Conditionals are treated this way because they can leak information. For example, the given a flow policy of USER_INPUT $\rightarrow$ FILESYSTEM, the following type checks.

```
@FlowSources(FlowSource.USER_INPUT) @FlowSinks(FlowSink.FILESYSTEM)
int creditCard = getCCNumber();
final int MAX_CC_NUM = 9999999999999999;
for (int i = 0 ; i < MAX_CC_NUM ; i++){
   if (i == creditCard)
        sendToInternet(i);
}
```

To catch this sort of information leak, pass `-Alint=strict-conditional` to change the default from ANY $\rightarrow$ CONDITIONAL to LITERAL $\rightarrow$ CONDITIONAL.

3.1.3 Empty Flow Sinks or Flow Sources

Programmers should not use @FlowSources({}) or @FlowSinks({}) for any types except in stub files. These types are only needed for top/bottom types which are used in the default types: the `null` literal uses the bottom type in order for

it to be assignable everywhere; local variables use the top type which will be refined by flow sensitivity. Every value should either flow from a literal or from some sensitive source. Likewise, every value must flow to a sensitive sink or to a conditional expression. Any variable that does not have a flow source or a flow sink does not actually affect the output of the program and should therefore be removed.

This may seem overly strict, but a variable without a flow source or flow sink that does affect the output of the program comes from an abuse of the type system. Most likely a variable with no source or sink would come from an improperly suppressed warning. Therefore it is necessary to not allow flows from and to nowhere. (The flow policy ensures this; see section Section 3.2)

Note that this does not mean you must specify both a flow source annotation and a flow sink annotation as explained in Section 3.3.

3.2 Flow Policy

A flow policy is a list of all the flows that are permitted to occur in an application. A flow-policy file is a list of ⟨flow source, flow sink⟩ pairs. If a flow is not listed in the flow policy, then it is forbidden to occur. If no flow policy is specified, then no flows are permitted.

3.2.1 Semantics of a Flow Policy

The Flow Checker guarantees that there is no information flow except for what is explicitly permitted by the policy file. If a user writes a type that is not permitted by the policy file, then the flow checker issues a warning even if all types in program otherwise typecheck.

For example, this variable declaration

```
@FlowSource(FlowSource.CAMERA) @FlowSink(NETWORK) Video video = ...
```

is illegal unless the the policy file contains:

```
CAMERA -> NETWORK
```

Here is another example. The flow policy file contains:

```
ACCOUNTS      -> EXTERNAL_STORAGE, FILESYSTEM
ACCELEROMETER -> EXTERNAL_STORAGE, FILESYSTEM, NETWORK
```

The following variable declarations are permitted:

```
@FlowSources(FlowSource.ACCOUNTS) @FlowSinks(FlowSink.EXTERNAL_STORAGE) Account acc = ...
@FlowSources(FlowSource.ACCELEROMETER, FlowSource.ACCOUNTS)
@FlowSinks(FlowSink.EXTERNAL_STORAGE, FlowSink.FILE_SYSTEM) int accel = ...
```

The following definitions would generate "forbidden flow" errors:

```
@FlowSources(FlowSource.ACCOUNTS) @FlowSinks(@FlowSink.NETWORK) Account acc = ...
@FlowSources({FlowSource.ACCELEROMETER, FlowSource.ACCOUNTS})
@FlowSinks({FlowSink.EXTERNAL_STORAGE, FlowSink.FILESYSTEM, FlowSink.NETWORK})
```

3.2.2 Syntax of a Flow Policy File

Each line of a policy file specifies a permitted flow from a source to one or more sinks. For example, `MICROPHONE -> NETWORK` implies that MICROPHONE data is always allowed to flow to NETWORK. The source must be a member of the enum `sparta.checkers.quals.FlowSources.FlowSource` and the sink must be a member of the enum `sparta.checkers.quals.FlowSinks.FlowSink`. The source and sink names should not be preceded by the name

of the enumeration which contains them. ANY is allowed just as it is in @FlowSources and @FlowSinks, but empty, {}, is not allowed.

Multiple sources or sinks can appear on the same line if they are separated by commas. For example, this policy file:

```
MICROPHONE -> NETWORK, LOG, DISPLAY
```

is equivalent to this policy file:

```
MICROPHONE -> NETWORK
MICROPHONE -> LOG
MICROPHONE -> DISPLAY, NETWORK
```

The policy file may contain blank lines and comments that begin with a number sign (#) character.

3.2.3 Using a flow-policy file

To use a flow-policy file when invoking the Flow Checker from the command line pass it the option:

`-AflowPolicy=`*`mypolicyfile`*

Or if you are using the `flowtest` ant targets, you can pass the option to ant:

`ant -DflowPolicy=`*`mypolicyfile`* `flowtest`

Remember, not specifying a flow policy file is equivalent to not allowing any flows.

3.3 Default Types

Location	Default Flow Type
Fields	`@FlowSources(FlowSource.LITERAL)`
Method parameters	`@FlowSources(FlowSource.LITERAL)`
Return values	`@FlowSources(FlowSource.LITERAL)`
`null`	`@FlowSources({}) @FlowSinks(FlowSink.ANY)`
Literals	`@FlowSources(FlowSource.LITERAL) @FlowSinks(FlowSink.CONDITIONAL)`
Local variables	`@FlowSources(FlowSource.ANY) @FlowSinks({})`
`@FlowSources(`α`)`	`@FlowSources(`α`) @FlowSinks(`ω`)`, ω is the set of sinks allowed to flow from all sources in α
`@FlowSinks(`ω`)`	`@FlowSources(`α`) @FlowSinks(`ω`)`, α is the set of sources allowed to flow to all sinks in ω

Table 3.1: Default types

To reduce the number of annotations needed, default types are used. Table 3.1 shows the default types used by the flow checker.

The defaults are not applied if the programmer uses annotations. For example, the parameter below is of type `@FlowSources(FlowSource.LOCATION) @FlowSinks(FlowSink.NETWORK)` rather than `@FlowSources(FlowSource.LITERAL)`. Note that `@FlowSources` means `@FlowSources({})` and `@FlowSinks` means `@FlowSinks({})`.

```
public void sendToInternet(
    @FlowSources(FlowSource.LOCATION) @FlowSinks(FlowSink.NETWORK) String message){...}
```

The value `null` has the bottom type (`@FlowSinks(FlowSink.ANY) @FlowSources`), so that it can be assigned to any type. For local variables, the default 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.

If the programmer specifies only flow sources, the flow sink is defaulted to be the sinks that the all the specified flow sources can flow to. This is to say that it is the intersection of the set of sinks each source can flow to. In other words, if a type is annotated with `@FlowSources(`α`)`, where α is a set of sources, then the flow sinks are the set ω where ω is the intersection of sinks B where A $\rightarrow$ B and A is a flow source in α. For example, if the flow policy contains the following:

```
A -> X,Y
B -> Y
C -> Y
```

then these are equivalent:

```
@FlowSources(A)                 ==   @FlowSources(A) @FlowSinks({X, Y})
@FlowSources(B)                 ==   @FlowSources(B) @FlowSinks(Y)
@FlowSources({B,C})             ==   @FlowSources({B,C}) @FlowSinks(Y)
@FlowSources(A) @FlowSinks(Y)   ==   @FlowSources(A) @FlowSinks(Y)
@FlowSources({A,B})             ==   @FlowSources({A,B}) @FlowSinks(Y)
```

Similarly, if the programmer only specifies flow sinks, the flow sources are defaulted to be the sources that are allowed to flow to all the specified sinks. In other words, if a type is annotated with `@FlowSinks(`ω`)`, where ω is a set of sinks, then the flow sources are the set α where α is the intersection of sources A where A $\rightarrow$ B and B is a flow sink in ω. For example, using the same policy file as above, the following are equivalent:

```
@FlowSinks(X)                   ==   @FlowSources(A) @FlowSinks(X)
@FlowSinks(Y)                   ==   @FlowSources({A,B,C}) @FlowSinks(Y)
```

3.4 Warning Suppression

Sometimes it might be necessary to suppress warnings or errors produced by the Flow Checker. This can be done by using the `@SuppressWarnings("flow")` annotation on a variable, method, or (rarely) class declaration. Because this annotation can be used to subvert the Flow Checker, its use is considered suspicious. Anytime a warning or error is suppressed, you must write a brief comment justifying the suppression. `@SuppressWarnings("flow")` should only be used if there is no way to annotate the code so that an error or warning does not occur. Most programs should not suppress warnings.

3.5 API specifications

Files in `sparta-code/src/sparta/checkers/flowstubfiles` provides library annotations. You may need to enhance them, if you find that your application uses APIs that are not yet annotated. For details, see section 5.3.1 of this manual, and also chapter "Annotating Libraries" in the Checker Framework Manual.

3.6 Qualifier polymorphism: @PolyFlowSources and @PolyFlowSinks

Two additional type annotations can be used to annotate polymorphic methods: `@PolyFlowSources`, `@PolyFlowSinks`.

To make the type system more expressive, the flow type system supports qualifier polymorphism, via the type qualifiers `@PolyFlowSources` and `@PolyFlowSinks`. These are mostly used when annotating APIs when the specific flow sources or flow sinks are not known or can vary. See section "Qualifier polymorphism" in the Checker Framework Manual.

3.7 Additional Annotations

Three additional declaration annotations can be used to annotate APIs: `@DefaultFlow`, `@ConservativeFlow`, and `@PolyFlow`. They change the default annotations (Section 3.3).

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

3.7.1 @DefaultFlow

An element annotated with `@DefaultFlow` expresses that an enclosed method's return type and all parameter types are `@FlowSources(FlowSource.LITERAL) @FlowSinks(FlowSink.CONDITIONAL)`. 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.

3.7.2 @ConservativeFlow

Annotation `@ConservativeFlow` expresses that each contained method should have the most conservative possible annotations: `@FlowSources() @FlowSinks(ANY)` on arguments, and `@FlowSources(ANY) @FlowSinks()` on return values. 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.

3.7.3 @PolyFlow

Annotation `@PolyFlow` expresses that each contained method should be annotated as `@PolyFlowSource @PolyFlowSink` for both the return types and all parameters.

3.8 Stricter tests

By default, the flow checker is unsound. After getting the basic checks to pass, the stricter checks should be enabled, by running `ant -Dsound=true flowtest`. This two-phase approach was chosen to reduce the annotation effortand to give two separate phases of the annotation effort. The sound checking enforces invariant array subtyping and type safety in downcasts.

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 there is not way for the type-checker to verify the component type of the array. 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.

3.9 Miscellaneous

Some methods that are intended to be overridden by subclasses, such as `Object`'s `equals()` and `toString()`, are given the most general possible return type, `@FlowSources(FlowSource.ANY) @FlowSinks()`. This permits overriding methods to give them a more specific return type, which might depend on fields of the overriding class as well as on the types of the arguments. In fact, an overriding method must give a more specific return type, since `@FlowSinks` prevents the value from being used.

Most binary operations, such as string concatenation and integer addition, produce a result whose type is the least upper bound of the two operand types.

Chapter 4

How to Analyze an Annotated App

If you are presented with an annotated app, you can confirm the work of the person who did the annotation by answering affirmatively three questions.

1. Does the Flow Checker produce any errors or warnings?
2. Does the flow-policy file match the application description?
3. Does the justification for each `@SuppressWarnings` make sense?

4.1 Run the Flow Checker

Run the Flow Checker (Chapter 3) to ensure that there is no data flow in the application beyond what is expressed in the given flow policy:

```
ant -DflowPolicy=myflowpolicy flowtest
```

If the Flow Checker produces any errors or warnings, then the app has not been properly annotated and should be rejected.

4.2 Review the Flow Policy

Does the flow-policy file match the application description? There should not be any flows that are not explained in the description. These flows may be explicitly stated, such as "encrypt and sign messages, send them via your preferred email app." Or a flow may only be implied, such as "This Application allows the user to share pics with their contacts." In the first example, you would expect an EMAIL sink to appear somewhere in the policy file. In the second, "share" could mean a you would see a Flow Sink of EMAIL, SMS, NETWORK, or something else. Flows that are only implied in the description could be grounds for rejection if the description is too vague.

4.3 Review `@SuppressWarnings` Justifications

Does the justification for every `@SuppressWarnings` make sense? Search for every instance of `@SuppressWarnings("flow")` and read the justification comment. Compare the justification to the actual code and determine if it make sense and should be allowed. No justification comment could be grounds for rejection.

Chapter 5

How to Analyze an Unannotated App

If you are presented with an unannotated app and wish to confirm that it contains no malware, then you need to perform three tasks:

- Look for obvious malware.
- Run reverse-engineering tools to understand the application.
- Write and check information-flow type qualifiers to ensure that the program has no undesired information flow.

More specifically, the recommended workflow is:

1. Set up the app for analysis by the SPARTA tools; see Section 2.3
2. Write the flow policy; see Section 5.1
3. Run reverse-engineering tools; see Section 5.2
4. Write and check information flow type qualifiers; see Section 5.3

5.1 Write a flow-policy file

Write a flow-policy file. Section 3.2 describes flow policies.

5.1.1 Read the app description

Read the App description and user documentation, looking for clues about the permissions , sensitive sources, and sinks and how information flows between them. For example, if this is a map app, does the description say anything about sending your location data over the network? If so, then you should add `LOCATION→NETWORK` to the flow-policy file. Where else does the description say LOCATION data can go?

Theoretically, you should be able to write a complete Flow Policy from the description if the description is well-written and the app does not contain malware. In practice, you will have to add flows to the policy file as you more fully annotate the app, but you should ensure that they are reasonable and make note of what additional flows you had to add.

5.1.2 Read the manifest file

Look at the `AndroidManifest.xml` file and:

- Determine which permissions the app uses — the "uses-permission" entries in the manifest file.
  (If you are short on time, you could start with reading the manifest file rather than first reading the app description as recommended in Section 5.1.1. But determining the permissions from the documentation will be more effective in finding problems in either the documentation or the code.)

- Compare the used permissions with the description of the application and determine whether or not they are well justified. If an application uses certain permissions that are not justified in the description, this indicates suspicious code. (To determine where these permissions are used in the application, see 5.2.2)
  (Permissions are not one-to-one with sinks and sources, but the name of the permission should give a clue about which sources and sinks are involved.) For example, if the SEND_SMS permission is requested, then SMS should be listed as a flow sink somewhere in the policy file. If this is not the case, do not add it to the policy file, but pay close attention to how this permission is used in the code.
- 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.

5.2 Run reverse-engineering tools

5.2.1 Review suspicious code and API uses

Run

```
ant reportsuspicious
```

to get a list of the most suspicious code locations. The code may be innocuous, but a human should examine it.

This target reports

- uses of potentially dangerous APIs, including reflection, randomness, thread spawning, and the ACTION VIEW intent.
  The file `sparta-code/src/sparta/checkers/suspicious.astub` contains the classes and methods that are considered suspicious.
  The following example from the `suspicious.astub` file reports all calls of the `invoke` method and, additionally, all constructor calls of the class `java.util.Random`:

```
package java.lang.reflect;
class Method {
    @ReportCall
    public Object invoke(Object obj, Object [] objs) {}
}

package java.util;
@ReportCreation
class Random {}
```

- suspicious String patterns (e.g., hard-coded URIs and IP and MAC addresses) in `.java` and `strings.xml` files. The searched-for patterns appear in the script `sparta-code/suspicious.pl`.

If you learn of additional suspicious API uses or String patterns, please inform the SPARTA developers so they can add them to the `suspicious.astub` or `suspicious.pl` file.

5.2.2 Review where permissions are used in the application

Run

```
ant reqperms
```

to see where the application calls API methods that require an Android permission.

The `ant reqperms` tool will help you gain an understanding of how your app first obtains information from a sensitive source, or how your app finally sends information to a sensitive sink. This may help you decide what parts of the app to further investigate, or where to start your annotation work.

The command produces output like the following:

```
error: Call additionally requires permissions [android.permission.INTERNET],
   but caller only provides []!
```

You can eliminate the error by writing `@RequiredPermissions(android.Manifest.permission.`*`PERMISSION`*`)` in front of the method header in the source code, if you would like to propagate the required permission up the call stack.

Once all methods in the subject application are correctly annotated with `@RequiredPermissions`, then `ant reqperms` will not produce any output. Then, you can search for `@RequiredPermissions` to find the required permissions in the application.

Any permission that is required should already be listed in the `AndroidManifest.xml` file.

The permissions required by the Android API appear in file `src/sparta/checkers/permission.astub`, expressed as `@RequiredPermissions` annotations.

5.2.3 Review what APIs are used where in the application

Run

```
ant reportapiusage
```

to get a report of Android and other APIs used in the application. These APIs are not suspicious in general (see 5.2.1 for reports about suspicious API use). However, they do help the analyst to better understand the structure of the code not just with respect to its standard module structure, but in terms of how it interacts with Android interfaces.

The `ant reportapiusage` target only reports about APIs that appear in the file: `sparta-code/src/sparta/checkers/apiusage.astub`. Additional `.astub` files can be passed using the `-Astubs=...` argument in the build file.

The following example from `apiusage.astub` causes `ant reportapiusage` to report the use of all entities in the package `com.android`:

```
@ReportUse
package com.android;
```

As you work, enhance the `apiusage.astub` to add the entities that are most crucial to understand the behavior of an application. Contact the SPARTA developers to tell them of additions that should appear in the file.

5.3 Verify information flow security

It is most effective to write information flow annotations from the bottom up: first annotate libraries your code uses, then your packages and classes that use those libraries, and so forth up to the entry points of your application. Section 5.3.1 describes how to annotate libraries, and the remainder of this section describes how to annotate your own code.

5.3.1 Write information flow types for library APIs

When the Flow Checker type-checks your code that calls a library API, the Flow Checker needs to know the effect of that call. Stub files in `sparta-code/src/sparta/checkers/flowstubfiles/` provide that information. You may need to enhance those stub files, if they do not yet contain information about the library APIs that your application uses. (Over time, the stub files will become more complete, and you will have to do less work in this step for each new app.)

For every API method used by this app (including all those output by `ant reportapiusage`), do the following:

- If the API method already appears in some stub file, there is nothing to do; continue to the next API method.
- Read the Javadoc.
- Decide what flow properties the method has. Take care with this step, because your decision will be trusted, not checked. If you make a mistake, the Flow Checker's results will not be sound.

- Add the method to the stub file that corresponds to the class package, with appropriate flow properties expressed as `@FlowSources(...)` and `@FlowSinks(...)` annotations. It would be unusual for an API method to contain both a `@FlowSources` and a `@FlowSinks` annotation.
  If the method does is not directly related to information flow (its inputs and outputs could be anything and are not required to have a specific `@FlowSources` annotation), then annotate its parameters and return type with `@PolyFlowSources @PloyFlowSinks`, which essentially says that the output can have all the flow sources and sinks of the inputs.

Important: After changing or adding stub files, run `ant jar` to rebuild `sparta.jar`.

The stub files can include any third-party library that is not compiled along with your application. You can add a new `.astub` file to the `flowstubfiles/` directory. Alternately, you can put a new `.astub` file elsewhere and then pass this file to the `ant flowtest` target:

`ant -Dstubs=`*`path/myAnnoLib.astub`* `flowtest`

Here is an example from a stub file:

```
package android.telephony;

class TelephonyManager {
    public @FlowSources(FlowSource.PHONE_NUMBER) String getLine1Number();
    public @FlowSources(FlowSource.IMEI) String getDeviceId();
}
```

The above annotates two methods in class `TelephonyManager`. It indicates that the `getLine1Number` function returns a String that is a phone number. For more examples, look into the stub files. Also, see the "Annotating Libraries" chapter in the Checker Framework Manual (`http://types.cs.washington.edu/checker-framework/current/checkers-manual.pdf`).

When creating a new stub file, see the section "Creating a stub file" in the Checker Framework Manual to learn how to create an initial file and prevent a great deal of repetitive cut-and-paste editing.

It is usually a good idea to annotate an entire API class at a time, rather than to just annotate the specific methods that your app uses. Annotating an entire class lets you think about it once, which takes less time in the long run. It also reduces confusion for people who will later wonder whether a particular method was intended to be unannotated or had not yet been annotated.

Note: at the end of this step, you have not yet added any annotations to the app itself, only to libraries.

5.3.2 Task: visualize the existing flow in the application

You do not have to run this step — you can skip it if you prefer.

Run `ant flowshow` 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.

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.

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.

5.3.3 Write information flow types for the app

Write information flow annotations for your application, in the same way as you did for the libraries. Read the documentation, decide on the types, and write those. A fast and effective way to do this is to grep the source code for words related to information flow properties, such as "camera" or "network". These words might appear in documentation

or in source code. Wherever the words appear, you may be able to write information flow type qualifiers. This approach is both faster and less error-prone than iteratively running the type-checker and fixing the errors that it reports one-by-one.

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.

Once you have written as many type qualifiers as possible, proceed to type-checking (Section 5.3.4).

5.3.4 Type-check the information flow types in the application

Run

```
ant flowtest
```

Eliminate each warning in one of two ways.

1. 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.
2. 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 the code is safe, and why the type system cannot recognize that fact.
   An 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()`.
   ```
   @SuppressWarnings("flow") // manually verified to not contain secret data
   @FlowSinks(FlowSink.NETWORK) String url = "http://bazinga.com/";
   ```
   Without warning suppression the assignment raises an error, because string literals are assumed to be annotated with `FlowSources(FlowSource.LITERAL)`. By adding the suppression, you assert that it's OK to send this string to the network.

After you have corrected some of the errors, re-run `ant flowtest`. Repeat the process until there are no more errors, or until you find bad code (malicious, or buggy and prone to abuse).

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

5.3.5 Type-check with stricter checking

Once all warnings were resolved, run

```
ant -Dstricter=true flowtest
```

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 3, page 8.

This option will also use the stricter conditional rule. (LITERAL $\rightarrow$ CONDITIONAL rather than the relaxed ANY $\rightarrow$ CONDITIONAL)

Chapter 6

Tips for writing information flow annotations

This chapter contains tips for annotating an Android application.

In general, only fields and methods signatures in your own code and in libraries need to be annotated. Usually method bodies do not need to be annotated.

6.1 Annotating Application Methods

Typically, return types should be annotated with just FlowSources so that the FlowSinks can be inferred from the policy file as explained in Section 3.3. Similarly, parameters should only be annotated with FlowSinks, so that the FlowSources can be inferred from the policy file. Local variables should not have to be annotated, because their types can be inferred. Fields must be annotated with FlowSinks or FlowSources, or sometimes both.

6.2 Annotating APIs in Stub Files

6.2.1 Methods with Sources

If you read the Javadoc and the method returns an object from a certain source, then you should annotate the return type with that flow source and with `@FlowSinks(FlowSink.ANY)`. (If `@FlowSinks` is omitted, then it is assumed to be empty which means that the object returned cannot be assigned to a variable with a sink.)

The `getParameters` method is an example of a method that returns an object with a source.

```
package android.hardware;
class Camera {
    @FlowSources(FlowSource.CAMERA_SETTINGS) @FlowSinks(FlowSink.ANY)
        Camera.Parameters getParameters ();
    void setParameters (@FlowSources(FlowSource.ANY) @FlowSinks(CAMERA_SETTINGS)
         Camera.Parameters params);
}
```

```
Camera.Parameters parameters
     = mCamera.getParameters();
// Change some parameters
// If the policy file contains: CAMERA_SETTINGS->CAMERA_SETTINGS
// Then the following statement will not give an error:
mCamera.setParameters(parameters);
```

6.2.2 Methods with Sinks

If you read the Javadoc for an API method an it sends some parameters to a FlowSink, then the parameters should be annotated with that flow sink and ANY flow source.

Example annotation

```
package android.database.sqlite;
class SQLiteDatabase{
  public void execSQL (@FlowSinks(FlowSink.DATABASE) @FlowSource(FlowSource.ANY) String sql);
}
```

Use of the API

```
@FlowSources(SMS,LITERAL)  String getSMSQuery(){..}
String mes = getSMSQuery();
//if SMS,LITERAL->DATABASE is in flow policy
//Then the following statement will not give an error
db.execSQL(mes);
```

6.2.3 Callbacks

The Android API frequently uses callbacks. These are methods that the developer must implement and register. In stub files, these callbacks should be annotated with source information and FlowSink.ANY.

An example annotation of a callback method

```
package android.hardware;
class Camera$PictureCallback{
  //data: a byte array of the picture data
   void onPictureTaken
      (@FlowSource(CAMERA) @FlowSinks(FlowSink.ANY) byte[] data,
            @FlowSource(CAMERA) @FlowSinks(FlowSink.ANY)  Camera camera);
}
```

An example implementation of a callback

```
public void onPictureTaken
(@FlowSource(CAMERA) byte[] data, @FlowSource(CAMERA) Camera camera){
   //If CAMERA->FILE_SYSTEM is in policy file
   //Then the following statement will not give an error
    writeToFile(data);
}
```

6.2.4 Methods that Transform Data

Some methods take the arguments passed, transform them, and then return them. These sorts of methods should be annotated with `@PolyFlowSources @PloyFlowSinks` to preserve the flow information. The declaration annotation `@PolyFlow` can be used instead of annotating all the parameters and return types. See Section 3.7.3 for more information

Math.min(...) is a good example of these kinds of methods.

```
package java.lang;
class Math{
   @PolyFlow
   int min(int i1, int i2);
}
```

Example use of `@PolyFlow`.

```
@FlowSources(FlowSource.LOCATION) int i1 = getLocation();
@FlowSources(FlowSource.NETWORK) int i2 = getLocationForNetwork();
@FlowSources({FlowSource.LOCATION,FlowSource.NETWORK)}) int min = Math.min(i1,i2);
```

6.3 Common Errors

This section explains some common errors issued by the Flow Checker, and gives advice about correcting the errors.

Also see the Checker Framework Manual (`http://types.cs.washington.edu/checker-framework/current/checkers-manual.pdf`), which contains information about pluggable type-checking in general. Many of your errors may not be specific to the Flow Checker and are likely to be answered in the Checker Framework Manual.

If you encounter a problem you cannot solve, contact the SPARTA developers (Section 1.2).

6.3.1 Forbidden Flow

Every source-sink pair in your code must be listed in the flow policy or else a *forbidden flow* error will occur. To correct a forbidden flow error, add the forbidden flow to the policy file.

For example, fix the error below by adding `LITERAL -> FILESYSTEM` to the policy file.

```
NewTest.java:43: error: flow forbidden by flow-policy
        test = new @FlowSinks(FlowSink.FILESYSTEM)@FlowSources(FlowSource.LITERAL) TestClass2(fs);
                   ^
  found: @FlowSinks(FlowSink.FILESYSTEM) @FlowSources(FlowSource.LITERAL) TestClass2
  forbidden flows:
    LITERAL -> FILESYSTEM
```

6.3.2 Incompatible Types

The most common errors are *incompatible types*. They can be in arguments, assignment, return, etc.

Conservative Typing APIs that have not been annotated have been typed so conservatively that they will always produce incompatible types errors where the required is `@FlowSinks(ANY) @FlowSources()` or `@FlowSinks() @FlowSources(ANY)`. These errors can be fixed by annotating the API method; Section 5.3.1 explains how to annotate APIs. Below is an example of this sort of error.

```
HelloWorld.java:84: error: incompatible types in argument
                 .replace(R.id.container, fragment)
                                          ^
   found   : @FlowSinks(CONDITIONAL) @FlowSources(LITERAL) Fragment
   required: @FlowSinks(ANY) @FlowSources({}) Fragment
```

Incompatible Types If the incompatible types error is not from conservative defaulting, then the error must be fixed by adding or removing annotations in the application. For example, the error below can be fixed by adding ACCELEROMETER to the FlowSource of the return type.

```
HelloWorld.java:49: error: incompatible types in return.
       return x;
              ^
   found   : @FlowSinks(CONDITIONAL) @FlowSources({LITERAL, ACCELEROMETER}) int
   required: @FlowSinks(CONDITIONAL) @FlowSources(LITERAL) int
```

6.3.3 Conditionals

As explained in Section 3.1.2, any item in a conditional statement must have CONDITIONAL listed as a FlowSink. If a variable is only annotated with FlowSources and strict conditionals are not used, then CONDITIONAL is added as a flow sink by default.

For example, if input is a parameter in a method and is annotated with @FlowSinks(FlowSink.NETWORK), the following error will occur. To fix the error, add CONDITIONAL to the flow sink annotation.

```
HelloWorld.java:48: error: Conditions are not allowed to depend on flow information.
         if(i1 > 2){
            ^
```

Chapter 7

Requirements of the app developer (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/unsound 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 8

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 https://dada.cs.washington.edu/hgweb/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. Also, if you are going to push changes, please add a `.hgrc` file to your home directory on the server. The `.hgrc` file should contain:

```
[trusted]
users = wmdietl
groups = sparta
```

This allows emails to be sent when you push changes.

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.checkers.quals.FlowSinks(sparta.checkers.quals.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). The 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.