SPARTA!
Static Program Analysis for Reliable Trusted Apps

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

Version 0.9.5 (12 December 2013)

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 and annotates the source code with type qualifiers that express that security property. Then a pluggable type-checker [PAC+08, DDE+11] verifies the type qualifiers, and thus verifies 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, but they 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, the type of an object that contains data that came from the camera and is destined for the network would be annotated with

```
@Source(CAMERA) @Sink(INTERNET)
```

The SPARTA tool set was developed with two different types of users in mind. 1.) Application vendors, who are the original authors of an app that submit the app to an app store for a security review. 2.) App analysts, or verification engineers, who work on behalf of the app store to verify that apps meant specific security properties before they are accepted.

Depending on the corporation between these two parties, they may use the SPARTA tools in two different ways.

- Ideally, the application vendor, who understands the source code, writes information flow annotations such as `@Source` 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 run the SPARTA tools. (Section 2.3).

Checker Framework

- If you are using the released version of SPARTA, follow the installation instructions in the manual: `http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#installation`
- If you are using the development version of SPARTA, follow Section 25.3 from the Checker Framework manual (Building from source.)
- For both versions, as described in the installation instructions, set the `CHECKERS` environment variable to `.../checker-framework/checkers/`

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

- Android API version 15 or later

1. Install the Android SDK to some directory.
2. Set `ANDROID_HOME` to the directory where you installed the Android SDK.
3. Download the `android-15` target by running `$ANDROID_HOME/tools/android`

If using Eclipse, go to `Help` → `Install New Software` and install the Android ADT Plugin (`https://dl-ssl.google.com/android/eclipse`).

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. Build the SPARTA tools by compiling the source code:
   `ant jar`
3. 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. Update the app configuration by running the following command in the main directory of the app.
   ```
   $ANDROID_HOME/tools/android update project --path .
   ```
3. Build the app
   ```
   ant release
   ```
   If the app does not build with the above commend, then the SPARTA tool set will not be able to analyze the app. Below are two common compilation issues are solutions.
   Most Android apps will rely on an auto-generated `R.java` file in the `/gen` directory of the project. This will only be generated if there are no errors in the project. There may be errors in the resources (`.../res` directory) that could cause `R.java` to not be generated.
   Additionally, if the app depends on an external `.jar` file (often located in the `lib/` directory), it will compile in Eclipse but not with Ant. To fix this, in ant.properties, add "`jar.libs.dir=lib`" (or wherever the `.jar` is located).
4. Add the SPARTA build targets to the end of the `build.xml` file, just above `</project>`.
   ```
   <property name="checkers" value="${env.CHECKERS}"/>
   <property name="sparta-code" value="${env.SPARTA_CODE}"/>

   <dirname property="checkers_dir" file="${checkers}" />
   <basename property="checkers_base" file="${checkers}" />
   <dirname property="sparta-code_dir" file="${sparta-code}" />
   <basename property="sparta-code_base" file="${sparta-code}" />

   <import file="${sparta-code_dir}/${sparta-code_base}/build.include.xml" />
   ```

Using Eclipse to analyze apps

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

- Import the projects the app. `Import` → `Existing Android Code Into Workspace`
- Make sure `Project Properties` → `Android` → `Android version #` is checked
- Check that `Project Properties` → `Java Build Path` → `Libraries` → `Android version #` appears
- Add sparta.jar to the apps build path
- Right click on the build.xml file and select `Run as` → `External Tools Configurations...`. In the `Main` tab add `check-flow` to the Arguments box. In the `Environment` tab, add the `CHECKERS` and `SPARTA_CODE` variables.

Chapter 3

Information Flow Checker

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

To use the Information 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 3.1 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 Information 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.

3.1 Flow Policy

A flow policy is a list of all the information flows that are permitted to occur in an application. A flow policy file expresses a flow policy, as a list of `flowsource`* $\rightarrow$ `flowsink`* pairs. Just as the Android manifest lists all the permissions that an app uses, the flow policy file lists the flows among permissions and other sensitive locations.

3.1.1 Semantics of a Flow Policy

The Information 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 Information Flow Checker issues a warning even if all types in program otherwise typecheck.

For example, this variable declaration

```
@Source(CAMERA) @Sink(INTERNET) Video video = ...
```

is illegal unless the the policy file contains:

```
CAMERA -> INTERNET
```

Here is another example. The flow policy file contains:

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

The following variable declarations are permitted:

```
@Source(ACCOUNTS) @Sink(EXTERNAL_STORAGE) Account acc = ...
@Source(ACCELEROMETER, ACCOUNTS)
@Sink(EXTERNAL_STORAGE, FILE_SYSTEM) int accel = ...
```

The following definitions would generate "forbidden flow" errors:

```
@Source(ACCOUNTS) @Sink(@INTERNET) Account acc = ...
@Source({ACCELEROMETER, ACCOUNTS})
@Sink({EXTERNAL_STORAGE, FILESYSTEM, INTERNET})
```

Transitivity and the flow policy file

The flow policy file indicates specific permitted information flows. It may be possible to combine these flows. For example, a policy that permits CAMERA→FILESYSTEM and FILESYSTEM→INTERNET will implicitly allow the flow CAMERA→INTERNET, because the application may record from the camera into a file and then send the contents of the file over the network. The Information Flow Checker forbids such implied flows: the developer is required to write the transitive flow in the flow policy file, which requires the developer to justify its purpose or convince the app store that the flow is not used.

3.1.2 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 -> INTERNET` implies that MICROPHONE data is always allowed to flow to INTERNET. The source or sink must be a member of the enum `sparta.checkers.quals.FlowPermission`. ANY is allowed just as it is in @Source and @Sink, but empty, `{}`, is not allowed.

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

```
MICROPHONE -> INTERNET, LOG, DISPLAY
```

is equivalent to this policy file:

```
MICROPHONE -> INTERNET
MICROPHONE -> LOG
MICROPHONE -> DISPLAY, INTERNET
```

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

3.1.3 Using a flow-policy file

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

`-AflowPolicy=`*`mypolicyfile`*

Or if you are using the `check-flow` ant target, you can pass the option to ant:

`ant -DflowPolicy=`*`mypolicyfile`* `check-flow`

If the flow policy is named `flow-policy` and is located the top level app directory, the ant target will use it automatically.

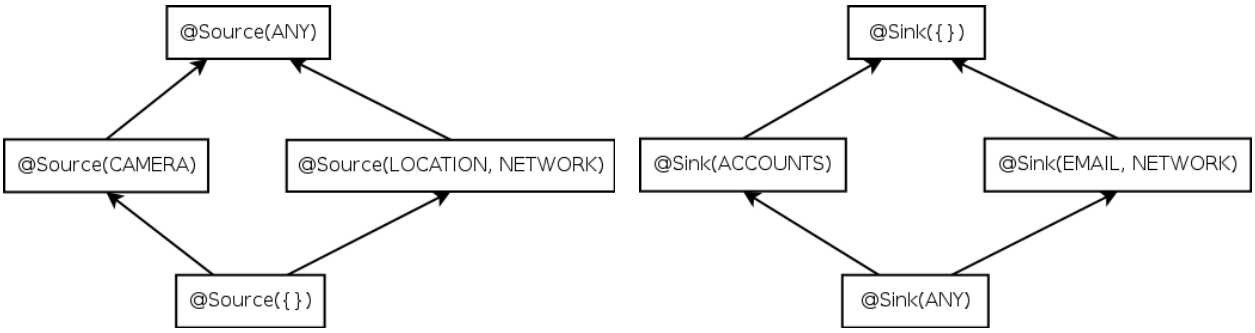

Figure 3.1: Partial qualifier hierarchy for flow source and flow sink type qualifiers, expressed as Java annotations `@Source` and `@Sink`.

3.2 Source and Sink Annotations

The type qualifier `@Source` on a variable's type indicates what sensitive sources might affect the variable's value. The type qualifier `@Sink` indicates where (information computed from) the value might be output. These qualifiers can be used on any occurrence of a type, including in type parameters, object instantiation, and cast types.

As an example, consider the declaration

```
@Source(LOCATION) @Sink(INTERNET) double loc;
```

The type of variable `loc` is `@Source(LOCATION) @Sink(INTERNET) double`. The `@Source(LOCATION)` qualifier indicates that the value of `loc` might have been derived from location information. Similarly, the qualifier `@Sink(INTERNET)` indicates that `loc` might be output to the network. It is also possible that the data has already been output. A programmer typically writes either `@Source` or `@Sink`, but not both, as explained in Section 3.4.

The arguments to `@Source` and `@Sink` are one or more permissions drawn from our enriched permission system (Section 3.3). The rarely-used special constant `ANY` denotes the set of all sources or the set of all sinks.

3.2.1 Subtyping

A type qualifier hierarchy indicates which assignments, method calls, and overridings are legal, according to standard object-oriented typing rules. Figure 3.1 shows parts of the `@Source` and `@Sink` qualifier hierarchies.

`@Source(`*B*`)` is a subtype of `@Source(`*A*`)` iff *B* is a subset of *A*. For example, `@Source(INTERNET)` is a subtype of `@Source({INTERNET, LOCATION})`. This rule reflects the fact that the `@Source` annotation places an upper bound on the set of sensitive sources that were actually used to compute the value. If the type of `x` contains `@Source({INTERNET, LOCATION})`, then the value in `x` might have been derived from both `INTERNET` and `LOCATION` data, or only from `INTERNET`, or only from `LOCATION`, or from no sensitive source at all.

The opposite rule applies for sinks: `@Sink(`*B*`)` is a subtype of `@Sink(`*A*`)` iff *A* is a subset of *B*. The type `@Sink({INTERNET, LOCATION})` indicates that the value is permitted to flow to both `INTERNET` and `FILESYSTEM`. This is a subtype of `@Sink(INTERNET)`, as the latter type provides fewer routes through which the information may be leaked.

3.3 Comparison to Android Permissions

The Information Flow Checker is finer-grained than standard Android manifest permissions in two ways. First, Android permits any flow between any pair of permissions in the manifest — that is, any resource mentioned in the manifest may be used in an arbitrary way. In contrast, the Information Flow Checker enforces the information flows in the flow policy. Second, the Information Flow Checker uses finer-grained permissions than Android does, in particular by adding additional permissions. Such finer-grained analysis is necessary in order to detect Trojans that would look innocuous,

Table 3.1: Additional permissions used by the Information Flow Checker, beyond the built-in 130 Android permissions.

Sources	Sinks	Both source and sink
ACCELEROMETER	CONDITIONAL	CAMERA_SETTINGS
BUNDLE	DISPLAY	CONTENT_PROVIDER
LITERAL	SPEAKER	DATABASE
MEDIA	WRITE_CLIPBOARD	FILESYSTEM
PHONE_NUMBER	WRITE_EMAIL	INTENT
RANDOM	WRITE_LOGS	PARCEL
READ_CLIPBOARD		PROCESS_BUILDER
READ_EMAIL		SECURE_HASH
READ_TIME		SHARED_PREFERENCES
REFLECTION		SQLITE_DATABASE
USER_INPUT		SYSTEM_PROPERTIES

given Android's coarser model. For example, the Information Flow Checker requires a permission to retrieve data from the accelerometer, which can indicate the user's physical activity, and to retrieve the time of day, which can be used as a trigger for malicious behavior.

Our system does not add much complexity: it only adds 28 permissions to Android's standard 130, or 22% more permissions. Table 3.1 lists the additional permissions.

We now discuss two permissions, LITERAL and CONDITIONAL, and empty whose meaning may not be obvious.

Literals

The LITERAL source is used for programmer-written manifest constants, such as `"Hello world!"`. This enables the Information Flow Checker to distinguish information derived from the program source code from other inputs. Manifest literals are used benignly for many purposes, such as configuring default settings. The flow policy shows the ways they are used in the program, and they can be directly examined by the analyst.

Conditionals

The Information Flow Checker treats conditional statements as a flow sink to enable detection of indirect flows that leak private information. For example, without any treatment of conditionals, the following code would be permitted under a flow policy containing LITERAL→INTERNET and USER_INPUT→FILESYSTEM:

```
@Source(USER_INPUT) @Sink(FILESYSTEM)
long creditCard = getCCNumber();
final long MAX_CC_NUM = 9999999999999999;
for (long i = 0 ; i < MAX_CC_NUM ; i++) {
    if (i == creditCard)
        sendToInternet(i);
}
```

To prevent malicious developers from bypassing the type system in this manner, the Information Flow Checker requires the type of a conditional expression to include the sink CONDITIONAL. By default, literals are allowed to flow to a conditional; that is, LITERAL→CONDITIONAL is added to the flow policy by default.

Data containing sensitive information is often passed through conditional statements for benign reasons. For example, an app might verify that a credit card number entered is valid by checking the number of digits.

```
@Source(USER_INPUT) @Sink(FILESYSTEM)
long creditCard = getCCNumber();
final long MAX_CC_NUM = 9999999999999999;
```

```
if (MAX_CC_NUM < creditCard)
    reportTooManyDigits();
```

In this case, the credit card number is not indirectly leaked, but the Information Flow Checker cannot determine this and will report a false positive. The auditor must examine the code nearby to ensure that the conditional is not being used to indirectly leak information.

Empty Sink or Source

Programmers may not use @Source({}) or @Sink({}). 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 sink does not actually affect the output of the program and should therefore be removed.

Note that this does not mean you must specify both a flow source annotation and a flow sink annotation as explained in Section 3.4.

3.4 Inference and defaults

A complete type consists of a `@Source` qualifier, a `@Sink` qualifier, and a Java type. To reduce programmer effort and code clutter, most of the qualifiers are inferred or defaulted.

A programmer need not write qualifiers within method bodies, because such types are inferred by the Information Flow Checker (see Section 3.4.1). Even for method signatures and fields, a programmer generally writes either `@Source` or `@Sink`, but not both; see Section 3.4.2 and Section 3.4.3.

3.4.1 Local variable type inference

A programmer does not write information flow types within method bodies. Rather, local variable types are inferred.

We limit type inference to local variables to ensure that each method can be type-checked in isolation, with a guarantee that the entire program is type-safe if each method has been type-checked. It would be possible to perform a whole-program type inference, but such an approach would not be modular, would be heavier-weight, would not deal well with cooperating or communicating applications, and would provide fewer documentation benefits.

3.4.2 Determining sources from sinks and vice versa

If a type contains only a flow source or only a flow sink, the other qualifier is filled in with the most general one that is consistent with the policy file. If the programmer writes `@Source(`α`)`, the Information Flow Checker defaults this to `@Source(`α`) @Sink(`ω`)` where ω is the set of flow sinks that all sources in α can flow to. Similarly, `@Sink(`ω`)` is defaulted to `@Source(`α`) @Sink(`ω`)` where α is the set of flow sources allowed to flow to all sinks in ω. Defaults are not applied if the programmer writes both a source and a sink qualifier.

For example, suppose the flow policy contains the following:

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

Then these pairs are equivalent:

```
@Source(B,C)   =  @Source(B,C) @Sink(Y)
    @Sink(Y)   =  @Source(A,B,C) @Sink(Y)
```

This mechanism is useful because oftentimes a programmer thinks about a computation in terms of only its sources or only its sinks. The programmer should not have to consider the rest of the program that provides context indicating the other end of the flow.

Table 3.2: Default information-flow qualifiers for unannotated types

Location	Default Flow Type
`@Source(`α`)`	`@Source(`α`) @Sink(`ω`)`, ω is the set of sinks allowed to flow from all sources in α
`@Sink(`ω`)`	`@Source(`α`) @Sink(`ω`)`, α is the set of sources allowed to flow to all sinks in ω
Method parameters	`@Sink(CONDITIONAL)`
Method receivers	`@Sink(CONDITIONAL)`
Return types	`@Source(LITERAL)`
Fields	`@Source(LITERAL)`
`null`	`@Source({}) @Sink(ANY)`
Other literals	`@Source(LITERAL)`
Type arguments	`@Source(LITERAL)`
Local variables	`@Source(ANY) @Sink({})`
Upper bounds	`@Source(ANY) @Sink({})`
Resource variables	`@Source(ANY) @Sink({})`

This defaulting mechanism is essential for annotating libraries. The Information Flow Checker ships with manual annotations for more than 10,000 methods of the Android standard library. 92% of methods use only a `@Source` or `@Sink` annotation but not both. An example is the `File` constructor: a newly-created readable file should be annotated with `@Source(FILESYSTEM)`, but there is no possible `@Sink` annotation that would be correct for all programs. Instead, the `@Sink` annotation is omitted, and our defaulting mechanism provides the correct value based on the application's flow policy.

3.4.3 Defaults for unannotated types

Table 3.2 shows defaults for completely unannotated types. The Information Flow Checker allows a developer to choose a different default for a particular method, class, or package. When the default is only a source or only a sink, the other qualifier is inferred from the policy file as explained in Section 3.4.2.

Most unannotated types (including field types, return types, generic type arguments, and non-`null` literals) are given the qualifier `@Source(LITERAL)`. This is so that simple computation involving manifest literals, but not depending on Android permissions, does not require annotations.

As is standard, the `null` literal is given the bottom type qualifier, which allows it to be assigned to any variable. For the Information Flow Checker, the bottom type qualifier is `Source({}) @Sink(ANY)`.

3.5 Warning Suppression

Sometimes it might be necessary to suppress warnings or errors produced by the Information 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 should 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.

3.6 Annotating Library API methods in stub files

Annotations for API methods are found in the stub files in sparta-code/src/sparta/checkers/flowstubfiles. For details, see section 5.3.1 of this manual, and also chapter "Annotating Libraries" in the Checker Framework Manual. The methods that appear in stub files are defaulted the same way as methods written in apps, including flow policy inference. (See the defaulting section, Section 3.4.)

The Information Flow Checker outputs all of the methods missing from the stub files in a file called `missingAPI.astub` in the current working directory. It also contains a comment the number of times an API method is used by the app.

Any method not written in the stub files or found in source code is not defaulted normally. Instead, its return, receiver, and parameter types are annotated with `@Sources(NOT_REVIEWED) @Sinks(NOT_REVIEWED)`. This way, if such an API method is used, a type error will occur and alert the user to review and annotate the method. It is possible, but dangerous, to ignore these kind of warnings as explained in Section 5.3.3.

3.7 Polymorphism

Information flow type qualifiers interact seamlessly with parametric polymorphism (Java generics). For example, a programmer can declare

```
List<@Source(CONTACTS) @Sink(SMS) String> myList;
```

Here, the elements of `myList` are strings that are obtained from `CONTACTS` and that may flow to `SMS`.

The Information Flow Checker also supports qualifier polymorphism, in which the type qualifiers can change independently of the underlying type. This allows a programmer to write a generic method that can operate on values of any information flow type. For example, if a method is declared as `@PolySource int f(@PolySource int x)`, then it can be called on an `int` with any flow sources, and the result has exactly the same sources as the input. This can be viewed as a declaration and two uses of a type qualifier variable. The implicit type qualifier variables are automatically instantiated by the Information Flow Checker at the point of use.

For brevity, the additional annotation `@PolyFlow` can be written on a class or method declaration to indicate that all contained parameters and return types should be annotated as `@PolySource @PolySink`. `@PolyFlow` does not override explicitly-written annotations as explained in Section 3.8.1.

Parametric polymorphism, qualifier polymorphism, and regular Java types can be used together. The type system combines the qualifier variables and the Java types into a complete qualified type.

See section "Qualifier polymorphism" in the Checker Framework Manual.

3.8 Declaration Annotations to Specify Defaulting

The Information Flow Checker has additional declaration annotations that are shorthand for common annotation patterns on method signatures. They override the usual defaulting of method declarations.

3.8.1 @PolyFlow

Annotation `@PolyFlow` expresses that each contained method should be annotated as `@PolySource @PolySink` for both the return types and all parameters. It should be used to express a relationship between the return type and the parameter types, but not the receiver type.

3.8.2 @PolyFlowReceiver

Annotation `@PolyFlowReceiver` expresses that each contained method should be annotated as `@PolySource @PolySink` for the return type, all parameter types, and the receiver type.

3.8.3 Declaration Annotations in Stub Files

If `@PolyFlow` or `@PolyFlowReceiver` is written on a class or package, then the annotation applies to all contained methods or classes unless those classes or methods are annotated with another declaration annotation.

This change of defaulting happens to library methods that are not written in stub files. For example, the class Integer as been annotated with `@PolyFlowReceiver`, but the toString method is not listed in the stub file. This method is inferred to be annotated with `@PolyFlowReceiver` and therefore its use will not result in a type error involving the `NOT_REVIEWED` FlowPermission.

3.9 Stricter tests

By default, the Information Flow Checker is unsound. After getting the basic checks to pass, the stricter checks should be enabled, by running `ant -Dsound=true check-flow`. 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: "@Sink @Source(ANY) Object"
       may not be casted to the type "@Sink @Source 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.

Enabling stricter checking will also enable the `-Alint=strict-conditional` option to limit allowed sinks to conditionals.

3.10 Guidance on writing annotations

Chapter 7 is a tutorial on how to annotate an existing application. See Chapter 6 for tips about writing information-flow annotations.

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

4.1 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, INTERNET, or something else. Flows that are only implied in the description could be grounds for rejection if the description is too vague.

4.2 Run the Information Flow Checker

Run the Information 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 check-flow
```

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

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. If some `@SuppressWarnings` has no justification comment, that is 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 3.1 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→INTERNET` 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)
- 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 check-permissions
```

to see where the application calls API methods that may require some Android permissions. The `ant check-permissions` 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.

There are three possible types of errors you will see. The first error:

```
MainActivity.java:35:
error: Call require permission(s) [android.permission.SET_WALLPAPER],
but caller only provides []!
                        clearWallpaper();
                                    ^
```

This error means the method requires one or more permissions which the caller does not have. The second error:

```
MediaPlayerActivity.java:218:
error: Call may additionally require permission(s)
[android.Manifest.permission.WAKE_LOCK], but caller only provides []!
Notes: WAKE_LOCK is required if MediaPlayer.setWakeMode has been called first.
                                   stop();
                                       ^
```

This error means the method may or may not require one or more permissions which the caller does not have. An explanation for the current error can be seen on the `Notes`.

```
HelloWorldActivity.java:83: warning: If the constant DeviceAdminReceiver.ACTION_DEVICE_ADMIN_ENABLED
is passed to an intent it will require following permission(s): [android.permission.BIND_DEVICE_ADMIN]!
i.setAction(DeviceAdminReceiver.ACTION_DEVICE_ADMIN_ENABLED);
            ^
```

This error means that the constant used depends on one or more permissions.

You can eliminate the first 2 errors by writing `@RequiredPermissions(android.Manifest.permission.`*`PERMISSION`*`)` or `@MayRequiredPermissions(android.Manifest.permission.`*`WAKE_LOCK`*`)` in front of the method header in the source code, if you would like to propagate the required permission up the call stack. You should use `@MayRequiredPermissions( value=android.Manifest.permission.`*`PERMISSION`*`, notes=java.lang.String)` in case the permission may be required and you should explain the reason on the `notes` argument. However, it is not necessary to eliminate all the errors from RequiredPermissions. The `check-permissions` tool is only a tool to guide your annotation and manual analysis effort.

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` and `@MayRequiredPermissions` annotations.

5.3 Verify information flow security

When the goal is to completely annotate an application it is most effective to write information flow annotations in a bottom up approach: 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. Alternatively, when the goal is to investigate specific information flows, it is more effective to trace and annotate only the flows of interest. Libraries should still be annotated first for all flows types. A bottom up approach can be used as a first pass to annotate large portions of an application while tracing can be then used to find and fix remaining Information Flow Checker warnings. Both approaches use the flow-policy create in Section 5.1.

Section 5.3.1 describes how to annotate libraries, Section 5.3.2 and Section 5.3.3 describe how to annotate your own code in a bottom up approach, and Section 5.3.4 describes how to iteratively trace sensitive sources in your application.

5.3.1 Write information flow types for library APIs

When the Information Flow Checker type-checks your code that calls a library API, the Information 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.)

Run `ant check-flow` to create the `missingAPI.astub` file. For each method in the file do the following.

- 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 Information 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 `@Source(...)` and `@Sink(...)` annotations. It would be unusual for an API method to contain both a `@Source` and a `@Sink` 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 `@Source` annotation), then either added the method to the stub file with no annotations, which is the same as annotating the returns and parameters with `@Source(LITERAL)` or to annotate it with `@PolyFlow` or `@PolyFlowReceiver`, which essentially says that the output can have all the flow sources and sinks of the inputs.. (See Section Section 3.8 for more details.)

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 check-flow` target:

```
ant -Dstubs=path/myAnnoLib.astub check-flow
```

Here is an example from a stub file:

```
package android.telephony;

class TelephonyManager {
    public @Source(FlowPermission.PHONE_NUMBER) String getLine1Number();
    public @Source(FlowPermission.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 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 `Source(...)` or `Sink(...)` 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:

```
@Source(CAMERA)
@Sink(INTERNET) 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 `@Source` and the `@Sink` 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 5.3.3).

5.3.3 Type-check the information flow types in the application

Run the Information Flow Checker:

```
ant check-flow
```

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 `@Sink(CONDITIONAL)` and `@Source(LITERAL)`.
   ```
   @SuppressWarnings("flow") // manually verified to not contain secret data
   @Sink(INTERNET) String url = "http://bazinga.com/";
   ```
   Without warning suppression the assignment raises an error, because string literals have a default annotation of `Source(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 check-flow`. Repeat the process until there are no more errors, or until you find bad code (malicious, or buggy and prone to abuse).

Note: If you want to suppress the `NOT_REVIEWED` warnings you can run the Information Flow Checker in the following ways:

```
ant check-flow-ignorenr
```

or

```
ant -Dignorenr=on check-flow
```

5.3.4 Trace information flows

On execution, the Information Flow Checker creates a file called `forbiddenFlows.txt` in the current working directory. This file contains a summary of all of the information flows in the app that did not have a flow-policy entry when the Information Flow Checker was ran. `forbiddenFlows.txt` is recreated on every execution.

The Information Flow Checker caches the warning for each use of a forbidden flow. This cache of flow warnings can be filtered, called flow-filtering, using the command:

```
ant filter-flows -Dfilter="SOURCE -> SINK"
```

- `SOURCE` and `SINK` should be replaced with the desired sensitive flow permission to search for.
- `SOURCE` and `SINK` can be the exact name of a flow permission or a regular expression.
- The -> is required. However only one of `SOURCE` or `SINK` are required.

To start tracing information flows, begin by running the Information Flow Checker:

```
ant check-flow
```

Next, inspect the `forbiddenFlows.txt` file. Concrete flows are flows that do not have `{}` or `CONDITIONAL` sinks. Each concrete flow should be evaluated:

- If the flow is a desired information flow, add it to the flow-policy file.
- If the flow is a undesired information flow, do not add it to the flow-policy, but record both in the source code and elsewhere that you have found a security flaw.

After evaluating concrete flows, select a source from the `forbiddenFlows.txt` to trace. `LITERAL` sources should be traced last because the unannotated variables have a default source of `LITERAL` which may be revised. Use flow filtering to display all forbidden flow locations for the selected source.

For example, if `forbiddenFlows.txt` contains an entry `CAMERA→{}` this would indicate that a type `@Source(CAMERA)` flows to a type that had no declared sinks, `@Sink({})`. `CAMERA` would be a candidate for tracing. The following command could be used to perform flow-filtering for any flow with a source `CAMERA`.

```
ant command filter-flows -Dflow-filter="CAMERA ->"
```

Flow-filtering displays the source code locations of forbidden flows in the app. Inspect each source location and resolve the forbidden flow by adding annotations or suppressing warnings as described in Section 5.3.3.

Iteratively run the flow-checker, check the `forbiddenFlows.txt` file, and use flow-filtering to trace forbidden flows throughout the app. Eventually the selected source will flow to one or more concrete sinks. Again, determine if these flow should be added to the flow-policy or marked as malicious.

After adding a flow to the flow-policy and rerunning the Information Flow Checker, the flow will no longer appear in the `forbiddenFlows.txt` file. Select another sensitive source file to trace and begin the process again.

Repeat the tracing process until there are no more errors, or until you find bad code (malicious, or buggy and prone to abuse).

5.3.5 Type-check with stricter checking

Once all warnings were resolved, run

```
ant -Dsound=true check-flow
```

Providing the `sound` 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 9.

This option will also use the stricter conditional rule. (`LITERAL` → `CONDITIONAL` rather than the relaxed `ANY` → `CONDITIONAL`)

Chapter 6

Tips for writing information flow annotations

This chapter contains tips for annotating an Android application. The Checker Framework has some general tips, too. See http://types.cs.washington.edu/checker-framework/current/checkers-manual.html#tips-about-writing-annotatio for those tips.

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 Methods

Typically, return types should be annotated with just `@Source` so that `@Sink` can be inferred from the policy file as explained in Section 3.4.2. Similarly, parameters should only be annotated with `@Sink`, so that the `@Source` 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 `@Sink` or `@Source`, or sometimes both.

6.2 Annotating API Methods in Stub Files

6.2.1 Callbacks

The Android API frequently uses callbacks, that the developer implements in an implementing class and then registers through some API. In stub files, these callbacks should be annotated with source information that will be passed when the method is called.

An example annotation of a callback method

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

An example implementation of a callback

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

6.2.2 Methods that Transform Data

Some methods take the arguments passed, transform them, and then return them. These sorts of methods should be annotated with `@PolySource @PloySink` 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 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`.

```
@Source(LOCATION) int i1 = getLocation();
@Source(INTERNET) int i2 = getLocationForNetwork();
@Source({LOCATION,INTERNET}) int min = Math.min(i1,i2);
```

6.3 Common Errors

This section explains some common errors issued by the Information 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 Information 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, the warning below can be corrected by adding `LITERAL -> FILESYSTEM` to the policy file, if this flow is justified and innocuous.

```
NewTest.java:43: error: flow forbidden by flow-policy
        test = new @Sink(FlowPermission.FILESYSTEM)@Source(FlowPermission.LITERAL) TestClass2(fs);
                   ^
  found: @Sink(FlowPermission.FILESYSTEM) @Source(FlowPermission.LITERAL) TestClass2
  forbidden flows:
    LITERAL -> FILESYSTEM
```

6.3.2 Incompatible Types

The most common error type is *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 `@Sink(NOT_REVIEWED) @Source(NOT_REVIEWED)`. 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   : @Sink(CONDITIONAL) @Source(LITERAL) Fragment
   required: @Sink(NOT_REVIEWED) @Source(NOT_REVIEWED) Fragment
```

Incompatible Types If the error is not because of an unannotated API, 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 FlowPermission of the return type.

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

6.3.3 Conditionals

As explained in Section 3.3, any item in a conditional statement must have CONDITIONAL listed as a FlowPermission. If a variable is only annotated with `@Source` 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 @Sink(FlowPermission.INTERNET), the following error will occur. To fix the error, add CONDITIONAL to the flow sink annotation.

```
.../HelloWorld.java:43: warning: sensitive source information found in a conditional.
     if(location.equals(WASHINGTON))
       ^
   found: [ACCESS_FINE_LOCATION]
```

Chapter 7

Tutorial

This chapter demonstrates how to annotate an existing app, ContactManger, where the annotator did not develop the app and the app is assumed to be benign. ContactManger allows the user to view and create contacts that are associate with different accounts.

7.1 Set up

Download the ContactManager app here: `http://types.cs.washington.edu/sparta/tutorial/ContactManager.tgz` Install the Information Flow Checker and set up ContactManger to use the Information Flow Checker. See Section 2.3 and Section 2.2 for instructions. Also add the following imports to both source files:

```
import static sparta.checkers.quals.FlowPermission.*;
import sparta.checkers.quals.*;
```

Run the Information Flow Checker, `ant check-flow`, if the output is similar to the output shown below, then your setup is correct. (You should get 43 warnings.)

```
Buildfile: .../ContactManager/build.xml
...
check-flow:
[jsr308.javac] Compiling 4 source files to .../ContactManager/bin/classes
[jsr308.javac] javac 1.8.0-jsr308-1.7.0
[jsr308.javac] warning: The following options were not recognized by any processor: '[componentMap]'
[jsr308.javac] .../ContactManager/src/com/example/android/contactmanager/ContactAdder.java:75: warning: in
[jsr308.javac]         Log.v(TAG, "Activity State: onCreate()");
[jsr308.javac]               ^
[jsr308.javac]   found   : @Sink(FlowPermission.CONDITIONAL) @Source(FlowPermission.LITERAL) String
[jsr308.javac]   required: @Sink(FlowPermission.WRITE_LOGS) @Source({}) String
```

7.2 Drafting a flow policy

The Information Flow Checker outputs a file, `forbiddenFlows.txt`, that lists all the flows it found in the app that are not allowed by the current flow policy. For this app, `forbiddenFlows.txt` is shown below.

```
# Flows currently forbidden
 ANY -> WRITE_LOGS
 CONTENT_PROVIDER -> {}
 LITERAL -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
 USER_INPUT -> WRITE_LOGS, CONTENT_PROVIDER
```

Because this app does not yet have a flow policy, this file lists all the flows that the Information Flow Checker was able to detect. This is not a complete list of flows in the program, because the Information Flow Checker issued warnings, but it offers a good starting point for the flow policy.

Create a file named `flow-policy` in the ContactManger directory. Copy all the flows that do not flow to or from ANY or {}. These flows should not be copied because they are too permissive. So, for this app the initial flow policy is shown below.

```
 LITERAL -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
 USER_INPUT -> WRITE_LOGS, CONTENT_PROVIDER
```

Run the Information Flow Checker again. (Because you named your flow policy `flow-policy`, the Information Flow Checker will automatically read it.) The Information Flow Checker should now only report 5 warnings. The `forbiddenFlows.txt` file should also have changed as shown below.

```
# Flows currently forbidden
 ANY -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
 CONTENT_PROVIDER -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
```

Using the initial flow policy, the Information Flow Checker was able to find more forbidden flows. Copy the new flows to the flow-policy, so the flow policy should now contain the following flows:

```
 LITERAL -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
 USER_INPUT -> WRITE_LOGS, CONTENT_PROVIDER
 CONTENT_PROVIDER -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
```

Run the Information Flow Checker again. The `forbiddenFlows.txt` file changed again.

```
# Flows currently forbidden
 ANY -> DISPLAY, WRITE_LOGS, CONTENT_PROVIDER
 CONTENT_PROVIDER -> {}
```

Since both of these flows contain either ANY or {}, there is nothing to add to the flow policy. However, the flow policy may not be complete, because the Information Flow Checker issued 5 warnings.

7.3 Adding Annotations

Next, annotate the code to ensure the flow policy correctly represents the flows in the app. One way to annotate an unfamiliar app, is to work through each warning one by one. Correct it by adding annotations, re-running the Information Flow Checker, and then correct the next warning. In general, Information Flow Checker warnings are written as shown below.

```
../SomeClass.java:line number: warning: some types are incompatible
             source code causing warning
                     a caret (^) pointing to the location of the warning.
   found   : Qualified type found by the Information Flow Checker
   required: Qualified type that the Information Flow Checker was expecting.
```

In order to correct a warning and correctly capture the app behavior, answer the following questions for each warning.

1. Why are the found and required annotations those listed in the warning message?
    - Is the annotation explicitly written in the source code?
    - Is the annotation from an API method that was annotated in stub file? Section 3.6
    - Is the annotation inferred? Section 3.4.1

- Is the annotation defaulted? Section 3.4.3

2. Why is the found type not a subtype of the required type? Section 3.2.1
   - Does the found type have more or different found sources than required?
   - Does the found type have less or different found sinks than required?
3. What annotation or annotations correctly capture the behavior of the app at this location? (In other words, what annotation will make the found type a subtype of the required type?)
   - Add only a source or a sink annotation

This tutorial only covers incompatibility warnings; see section Section 6.3 for other warnings and how to handle them.

7.3.1 Warning 1

Run the Information Flow Checker again.

```
.../ContactManager/src/com/example/android/contactmanager/ContactAdder.java:309:
 warning: incompatible types in assignment.
             mName = name;
                     ^
   found   : @Sink(CONDITIONAL) @Source(ANY) String
   required: @Sink({CONDITIONAL, DISPLAY, WRITE_LOGS, CONTENT_PROVIDER})  @Source(LITERAL) String
```

This is an "incompatible types in assignment" error. It means that the type of the left hand side, the found type, is not a subtype of the right hand side, the required type.

1. **Where do the found and required types come from?**
   The type of `name` is they found type and the type of `mName` is shown as *required*.
   - Was it explicitly annotated in the source code? No, we have not added any annotations.
   - Is it from an API method that was annotated in stub file? No, `mName` is a field and `name` is a parameter to this method.
   - Was it inferred from an assignment? No, neither variable was assigned perviously in this method.
   - Was it defaulted? Yes.

   According to the defaulting rules explained in Section 3.4.3 fields, like *mName* are annotated with `@Source(LITERAL)` and the sink annotation is defaulted base on the flow policy. For this app, literals are allowed to flow to DISPLAY, WRITE_LOGS, andCONTENT_PROVIDER, so the sink annotation is defaulted to `@Sink({DISPLAY, WRITE_LOGS, and CONTENT_PROVIDER})`.
   Method parameters, like *name*, are annotated with `@Sink(CONDITIONAL)` by default and the source is defaulted base on what is allowed to flow to conditionals. ANY→CONDTIONAL is added to the flow policy by default, so the default types for fields in this class is `@Sink(CONDITIONAL) @Source(ANY)`.
2. **Why is the found type not a subtype of the required type?** The found type has more sources than the required type. (Remember that ANY is the set of all Flow Permissions). Also, the *found* type has fewer sinks than the required type.
3. **What annotation or annotations would make the found type a subtype of the required?** One of the defaults described above needs to be overridden by annotating the either to the type of `name` or the type of `mName`. So which type should be annotated? Because the current type of `name` allows more flows than the type of `mName`, its annotation should be changed so that it is a sub type of `mName`. The type of `name` should therefore be changed to `@Source(LITERAL)`. The sink annotation will be defaulted base on the flow policy and will match the sink annotation from the required type.
   `public AccountData(@Source(LITERAL) String name, AuthenticatorDescription description)`

This warning could also have been corrected by annotating `mName` with `@Sink(CONDITIONAL)`. In general, you should pick that least permissive one. Literals are only allowed to flow to four sinks, but any source can flow to conditionals, so `@Source(LITERAL)` is less permissive than `@Sink(CONDITIONAL)`.

Run the Information Flow Checker again. Only four errors should be issued.

7.3.2 Warning 2

```
.../ContactManager/src/com/example/android/contactmanager/ContactAdder.java:387:
warning: incompatible types in return.
             return convertView;
                    ^
   found   : @Sink(CONDITIONAL) @Source(ANY) View
   required: @Sink({CONDITIONAL, DISPLAY, WRITE_LOGS, CONTENT_PROVIDER}) @Source(LITERAL) View
```

This is an "incompatible types in return" error. It means that the type of returned by this method, the found type, is not a subtype of the declared return type of this method, the required type.

1. **Where do the found and required types come from?** `convertView` is a parameter that is defaulted to `@Sink(CONDITIONAL)` and the required type is the return type of this method. Because there is not an annotation on the return type of this method, its type is defaulted to `@Source(LITERAL)`.
2. **Why is the found type not a subtype of the required type?** The found type has more sources than the required type. Also, the *found* type has fewer sinks than the required type.
3. **What annotation or annotations would make the found type a subtype of the required?** This warning is similar to the first warning and should also be fixed by annotating the found type with `@Source(LITERAL)`.

   ```
   public View getDropDownView(int position, @Source(LITERAL) View convertView, ViewGroup parent)
   ```

Run the Information Flow Checker again. Only three errors should be issued.

7.3.3 Warning 3

```
.../ContactManager/src/com/example/android/contactmanager/ContactManager.java:74:
warning: incompatible types in argument.
                Log.d(TAG, "mShowInvisibleControl changed: " + isChecked);
                                                               ^
   found   : @Sink(CONDITIONAL) @Source(ANY) String
   required: @Sink(WRITE_LOGS) @Source({LITERAL, USER_INPUT, CONTENT_PROVIDER}) String
```

This is an "incompatible types in argument" error. It means that the type of argument, the found type, is not a subtype of the formal parameter type of the method called, the required type.

1. **Where do the found and required types come from?** The found type is the type of the concatenation of the literal string and `isChecked`. The resulting type of a concatenation is the least upper bounds, LUB, of the two types. The LUB of a flow type is simply the union of sources and the intersection of sinks. The string literal is of type `@Source(LITERAL)`. `isChecked` is a parameter of a methods that overrides an Android API method. The types of overriding methods must match those of the overridden method, so check the stub files for this method. `isChecked` is annotated with `@Source(USER_INPUT)` in the stub file, so it must be annotated the same here.

   ```
   public void onCheckedChanged(CompoundButton buttonView, @Source(USER_INPUT) boolean isChecked)
   ```

Run the Information Flow Checker; there should be 2 warnings.

7.3.4 Warning 4

```
.../ContactManager/src/com/example/android/contactmanager/ContactManager.java:77:
warning: incompatible types in assignment.
                mShowInvisible = isChecked;
                               ^
   found   : @Sink({CONDITIONAL, WRITE_LOGS, CONTENT_PROVIDER}) @Source(USER_INPUT) boolean
   required: @Sink({CONDITIONAL, DISPLAY, WRITE_LOGS, CONTENT_PROVIDER}) @Source(LITERAL) boolean
```

1. **Where do the found and required types come from?** The found type is a parameter that is annotated and the required type is a field that was defaulted.
2. **Why is the found type not a subtype of the required type?** The found source is USER_INPUT and the required source is LITERAL.
3. **What annotation or annotations would make the found type a subtype of the required?** The required type has already been annotated, so it should not be changed; therefore, the found type should be annotated with `@Source(USER_INPUT)`.

   ```
   private @Source(USER_INPUT) boolean mShowInvisible;
   ```

Run the Information Flow Checker; there should be 3 warnings. Two of the warnings are because of the annotation that was just added.

7.3.5 Warning 5

```
.../ContactManager/src/com/example/android/contactmanager/ContactManager.java:63:
warning: incompatible types in assignment.
         mShowInvisible = false;
                        ^
   found   : @Sink({CONDITIONAL,DISPLAY, WRITE_LOGS, CONTENT_PROVIDER}) @Source(LITERAL) boolean
   required: @Sink({CONDITIONAL, WRITE_LOGS, CONTENT_PROVIDER}) @Source(USER_INPUT) boolean
```

1. **Where do the found and required types come from?** The found type is from the boolean literal. The required type is from an annotated field.
2. **Why is the found type not a subtype of the required type?** The found source is LITERAL and the required type is USER_INPUT
3. **What annotation or annotations would make the found type a subtype of the required?** The annotation on the boolean literal cannot be changed, so the annotation of the field must be changed. The annotation is `@Source(USER_INPUT)`, so we can add the LITERAL to the list of sources.

   ```
   private @Source({USER_INPUT, LITERAL}) boolean mShowInvisible;
   ```

Run the Information Flow Checker; there should be 1 warning. The last change fixed two warnings.

7.3.6 Warning 6

```
.../ContactManager/src/com/example/android/contactmanager/ContactManager.java:118:
warning: incompatible types in return.
         return managedQuery(uri, projection, selection, selectionArgs, sortOrder);
                            ^
   found   : @Sink({CONDITIONAL, DISPLAY, WRITE_LOGS, CONTENT_PROVIDER}) @Source(CONTENT_PROVIDER) Cursor
   required: @Sink({CONDITIONAL, DISPLAY, WRITE_LOGS, CONTENT_PROVIDER}) @Source(LITERAL) Cursor
```

1. **Where do the found and required types come from?** The found type is from the return type of an API method that was annotated in a stub file. The required type is the return type of this method, `getContacts()`, which is defaulted.
2. **Why is the found type not a subtype of the required type?** The found source is `@Source(CONTENT_PROVIDER)` and the required source is `@Source(LITERAL`.
3. **What annotation or annotations would make the found type a subtype of the required?** Because the found type is from a stub file annotation, it cannot be changed. So, the return type of this method, `getContacts()` must be annotated with `@Source(CONTENT_PROVIDER`.

```
private @Source(CONTENT_PROVIDER) Cursor getContacts()
```

Run the Information Flow Checker; there should be no warnings.

7.4 Correctly annotated app

Now that the Information Flow Checker no longer reports any warnings, it guarantees that ContactManger only contains the information flows in the flow policy.

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.