SPARTA!
Static Program Analysis for Reliable Trusted Apps

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

Version 1.0.1 (24 April 2015)

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 5 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 6 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 `CHECKERFRAMEWORK` environment variable to `.../checker-framework/`

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`

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/`.
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.
   - `CHECKERFRAMEWORK` is the `.../checker-framework/` 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.
   ```
   $ANDROID_HOME/tools/android update project --path .
   ```
3. Copy the file `sparta.jar` generated in Section 2.2 to the directory `libs/` in the main directory of the app. If that folder doesn't exist it must be created.
4. Build the app
   ```
   ant release
   ```
   If the app does not build with the above command, then the SPARTA tool set will not be able to analyze the app. Below are two common compilation issues and solutions.
   - Most Android apps will rely on an auto-generated `R.java` file in the `/gen` directory of the project. This will only be generated if there are no errors in the project. There may be errors in the resources (`.../res` directory) that could cause `R.java` to not be generated.
   - Additionally, if the app depends on an external `.jar` file, it will compile in Eclipse but not with Ant. All dependent `.jar` files must be in the `libs/` directory.
5. Add the SPARTA build targets to the end of the `build.xml` file, just above `</project>`.
   ```
   <property name="checker-framework" value="${env.CHECKERFRAMEWORK}"/>
   <property name="sparta-code" value="${env.SPARTA_CODE}"/>

   <dirname property="checker-framework_dir" file="${checker-framework}/checker-framework" />
   <dirname property="sparta-code_dir" file="${sparta-code}/sparta-code" />

   <import file="${sparta-code_dir}/build.include.xml" />
   <property name="flowPolicy" value="flow-policy"/>
   <property name="componentMap" value="component-map"/>
   ```

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 `CHECKERFRAMEWORK` 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 3.4 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 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 3.5.

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.

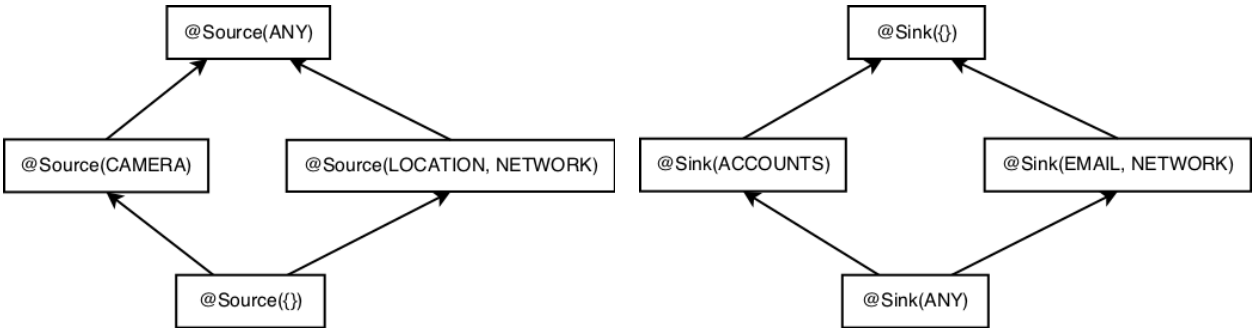

Figure 3.1: Partial qualifier hierarchy for flow source and flow sink type qualifiers, expressed as Java annotations `@Source` and `@Sink`.

3.1.1 Syntax

The source and sink qualifiers are type annotations that can be used on any type use. They have a single attribute, an array of Strings, that represent the permissions. The Strings must start with the string value of a constant in the FlowPermission enum; a class with a static string constant, FlowPermissionString, can be statically imported and used for this purpose. They can be optionally parameterized as shown below:

```
PERMISSION(param1, param1)
```

Some examples:

```
@Source(FILESYSTEM)
@Source(INTERNET+"(uw.edu)")
@Sink({WRITE_SMS, WRITE_CONTACTS})
```

3.1.2 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.

3.2 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 3.5.4.

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.

3.2.1 Receiver polymorphism

Receiver polymorphism restricts the return and/or parameter qualifier types of a method to be exactly the qualifier type of the receiver. For example, the StringBuffer append method uses this sort of polymorphism to restrict they type of strings that can be appended. For example,

```
StringBuffer buffer = (@Source(USER_INPUT) StringBuffer) new StringBuffer();
@Source(READ_SMS) String smsMessage = getSMS();
buffer.append(smsMessage);  // illegal
buffer.append(getUserInput());  // legal
```

Parameters, returns, and receivers are annotated with `@PolySourceR` or `@PolySinkR` to indicate this.

3.3 Comparison to Android permissions

The Information Flow Checker's permissions are 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, an Android program may use any resource mentioned in the manifest in an arbitrary way. Second, the Information Flow Checker refines Android's permissions, as we now discuss.

The Information Flow Checker's permissions are not enforced at run time as Android permissions are (the latter potentially resulting in an exception during execution). Rather, they are statically guaranteed at compile time. Even if an app inherited a permission from another app with the same sharedUserId, the Information Flow Checker will require that permission to be listed in the flow policy.

Restricting existing permissions

Android's INTERNET permission represents all reachable hosts on the Internet, which is too coarse-grained to express the developer's intention. The Information Flow Checker allows this permission to be parameterized with a domain name, as in INTERNET("*.google.com"). Other permissions can be parameterized in a similar style in which the meaning of the optional parameter varies based on the permission it qualifies. For example, a parameter to FILESYSTEM represents a file or directory name or wildcard, whereas the parameter to SEND_SMS represents the number that receives the SMS.

Other permissions that need to be parameterized include CONTACTS, *_EXTERNAL_FILESYSTEM, NFC, *_SMS, and USE_SIP, plus several of those described in Section 3.3, such as USER_INPUT to distinguish sensitive from non-sensitive user input.

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	DISPLAY	CAMERA_SETTINGS
BUNDLE	SPEAKER	CONTENT_PROVIDER
MEDIA	WRITE_CLIPBOARD	DATABASE
PHONE_NUMBER	WRITE_EMAIL	FILESYSTEM
RANDOM	WRITE_LOGS	INTENT
READ_CLIPBOARD		PARCEL
READ_EMAIL		PROCESS_BUILDER
READ_TIME		SECURE_HASH
REFLECTION		SHARED_PREFERENCES
SENSOR		SYSTEM_PROPERTIES
USER_INPUT		

Sinks and sources for additional resources

The Information Flow Checker adds additional sources and sinks to the Android permissions. 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 write to the logs, which a colluding app could potentially read. Table 3.1 lists the additional sources and sinks. We selected and refined these by examining the Android API and Android programs, and it is easy to add additional ones.

3.4 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 $\mathit{flowsource}^* \rightarrow \mathit{flowsink}^*$ 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.

3.4.1 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.

Parameters and the flow policy file

The flow policy file allows for parameterized sources and sinks. Users are allowed to add parameters to sources and sinks, but these are optional. Sources and sinks without parameters default to a parameter of ("*") , a wildcard subsuming all permissions of the given type.

Here are examples of a parameterized sources or sinks, respectively. The flow policy contains:

```
FILESYSTEM("/home/user") -> INTERNET
INTERNET("mydomain.org") -> INTERNET
ACCOUNTS      -> EXTERNAL_STORAGE("/tmp/*"), FILESYSTEM("/usr/bin/output")
ACCELEROMETER -> EXTERNAL_STORAGE("/tmp/*"), FILESYSTEM("/usr/bin/output"), INTERNET("mydomain.org")
```

Here are examples of both sources and sinks parameterized. The flow policy contains:

```
FILESYSTEM("/home/user") -> INTERNET("mydomain.org")
INTERNET("mydomain.org") -> INTERNET("mydomain.org")
```

3.4.2 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. Note that non-sensitive data is always allowed to flow to a sensitive sink. This means that `{} -> ANY` and `ANY -> {}` are always allowed and need not be written in the flow policy.

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.

Both sources and sinks have optional parameters. These parameters must appear right after the permission that is being parameterized, grouped in parentheses. Each parameter string is be wrapped in quotations, and multiple parameter strings for a single permission are separated by commas.

For example, this policy file:

```
MICROPHONE -> INTERNET("mydomain.org", "goodguys.com", "*.google.com")
```

is equivalent to this policy file:

```
MICROPHONE -> INTERNET("mydomain.org")
MICROPHONE -> INTERNET("goodguys.com")
MICROPHONE -> INTERNET("*.google.com")
```

3.4.3 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.

3.5 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 3.5.1). Even for method signatures and fields, a programmer generally writes either `@Source` or `@Sink`, but not both; see Section 3.5.2 and Section 3.5.3.

3.5.1 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.

3.5.2 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)
```

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	`@Source(ANY) @Sink({})`
Method receivers	`@Source(ANY) @Sink({})`
Return types	`@Source({}) @Sink(ANY)`
Fields	`@Source({}) @Sink(ANY)`
`null`	`@Source({}) @Sink(ANY)`
Other literals	`@Source({}) @Sink(ANY)`
Local variables	`@Source(ANY) @Sink({})`
Upper bounds	`@Source(ANY) @Sink({})`
Resource variables	`@Source(ANY) @Sink({})`
All other locations	`@Source({}) @Sink({})`

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.

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.

3.5.3 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 3.5.2.

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)`.

3.5.4 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.

@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

@PolyFlowReceiver

Annotation `@PolyFlowReceiver` expresses that each contained method should be annotated as `@PolySourceR @PolySinkR` for the return type, all parameter types, and the receiver type. (If the method or constructor does not have a receiver, then the annotation is treated as `@PolyFlow`.

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.

3.5.5 Inferring annotations with parameterized permissions

The declaration annotation @InferParameterizedPermission is used to indicate that a method's return type should use a parameterized permission, where the parameter of that permission is derived from the value of a method argument. The annotation indicates which argument(s) and/or receiver should be used, which permission should be parameterized, whether that permission is a source, a sink or both, and what if any separator should be used to concatenate multiple arguments. The value of a method argument is determined using the Constant Value Checker, which is documented in the Checker Framework Manual.

For example,

```
 // stub file:
@InferParameterizedPermission(value=FILESYSTEM, param=1, isA=source)
@Source(FILESYSTEM) FileInputStream(String arg0);

 // source code:
@Source("FILESYSTEM(filename)") FileInputStream fis = new FileInputStream("filename");
```

3.6 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.

3.7 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 6.3.1 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 3.5.)

Any method, constructor, or field not written in the stub files or found in source code is not defaulted normally. Instead, all locations except final fields are default to `@Source(ANY) @Sink(ANY)`. (Final fields are defaulted to `@Source({})`.) This way, if such an API method is used, a type error will occur and alert the user to review and annotate the method. A tool, discussed in Section 6.3.1, issues a warning every time a type declared in a library that does not appear in a stub file is used. This tool also outputs a list of declarations missing from the stub files.

3.8 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.

Chapter 4

Intent Checker

This chapter describes the Intent Checker, which type-checks information flows across communicating components of an Android app.

Android intents are used for communication within an app, among apps, and with the Android system. Intents can be seen as messages exchanged between Android components. Sensitive data can flow in and out of intent objects. Consequently, to detect forbidden flows derived from inter-component communication, the Intent Checker needs to identify the information flow types of the data in an intent.

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

- Information about inter-component communication across apps for the target program. Section 4.1 shows how to compute that information and supply it to the Intent Checker.
- Type qualifiers used to express the data types in an intent. Section 4.2 describes the type qualifiers used by the Intent Checker.

Sending an intent is similar to an ordinary method call, where the data in the intent are the method's arguments. The Intent Checker verifies that for any sending intent method call and its matching receiving intent method declarations, the intent argument of the caller is compatible with that of the corresponding callee.

An Android component can send a message to an Activity by calling the method `startActivity`, a Service by calling the method `startService`, or a BroadcastReceiver by calling `sendBroadcast`. Intents are received in an Activity by the method `setIntent`, in a Service by the method `onBind` or `onStartCommand`, and in a BroadcastReceiver by the method `onReceive`. For simplicity in this manual, we abstract all methods that send an intent as the method call `sendIntent`, and all methods that receive an intent as the method declaration `onReceive`.

4.1 Inter-component communication

The *component map* of an app contains information regarding how the components of this app communicate with each other, and how they communicate with components from other apps. Section 6.3.5 explains how to generate a component map file for an app.

4.1.1 Semantics of a component map

The component map matches `sendIntent` calls to declarations of the `onReceive` methods they implicitly invoke. A `sendIntent` call may be paired with more than one `onReceive` declaration. Each such pair indicates that the two components, possibly from different apps, may communicate through intents. The set of pairs of communicating components is conservative, that is it includes all possible pairs of methods that might communicate.

4.1.2 Syntax of a component map file

Each line of a component map file specifies one intent-sending method in the app and all components in the app that may receive intents that it sends.

For example, the following line

```
com.package.ActivityA.foo() -> com.package.ActivityB, com.package.ActivityC
```

indicates that an intent sent in the method `foo()` from `ActivityA` may be received by the components whose fully-qualified class names are `com.package.ActivityB` and `com.package.ActivityC`.

4.1.3 Using a component map file

It is recommended to name the component map file as `component-map` and to put it in the top level app directory. By doing so the ant target will use it automatically when running:

```
ant check-intent
```

Alternatively it is possible to pass the component map file path as an option to ant:

```
ant -DcomponentMap=mycomponentmapfile check-intent
```

4.2 Intent types

An intent contains a map from string keys to arbitrary values. Consider an intent variable `i`. Data can be added to the map of `i` by the sender component with `i.putExtra("key",val)` and then retrieved by the receiver component with `i.getExtra("key",default)`. An intent type is an approximation to the keys that may be in the intent object at run time and to the type of the values that those keys may map to. The type qualifiers used to represent an intent type are `@IntentMap` and `@Extra`.

4.2.1 Syntax

The type qualifier `@IntentMap` on an intent variable's type indicates the types of the key/value mappings that are permitted to be accessed via `putExtra` and `getExtra` calls. An `@IntentMap` type qualifier contains a set of `@Extra` type qualifiers. An `@Extra` type qualifier contains a key *k*, a source type *source*, and a sink type *sink*. This means that the key *k* maps to a value of source type *source* and sink type *sink*. Consider the declaration below:

```
@IntentMap({@Extra(key = "k1", source = {FILESYSTEM}, sink = {INTERNET}),
            @Extra(key = "k2", source = {FILESYSTEM}, sink = {DISPLAY})})
Intent i = new Intent();
```

The variable `i` is annotated with an `@IntentMap` type containing a set of two `@Extra` types, allowing this variable to be accessed via `i.putExtra("k1",...)`, `i.putExtra("k2",...)`, `i.getExtra("k1")` and `i.getExtra("k2")`. No other keys are allowed to be accessed via `putExtra` or `getExtra` calls.

Each intent variable's type must have only one `@IntentMap` type qualifier. Two different `@Extra` type qualifiers in the same `@IntentMap` may not have the same key *k*.

Every `onReceive` method has an intent formal parameter. Below is an example of an annotated intent formal parameter for the `onReceive` method `setIntent`:

```
@Override
public void setIntent(@IntentMap({
        @Extra(key = "location", source = {ACCESS_FINE_LOCATION }, sink = {})
                }) Intent newIntent) {
    super.setIntent(newIntent);
}
```

4.2.2 Semantics

If variable `i` is declared to have an intent type *T*, then two facts must be true. (1) For any key `k` that is accessed at run time in `i`, it must be listed in *T*. That is, the keys of `i` that are accessed are a subset of *T*'s keys. It is permitted for the run-time value of variable `i` to have fewer keys than those listed by its type. It is also permitted for the run-time value of variable `i` to have more keys than those listed by its type but they may not be accessed. (2) For any key *k* that is a key in the run-time value of `i`, the value mapped by *k* in the value has type mapped by *k* in *T*. This can be more concisely expressed as $\forall k.i[k] <: T[k]$. As permitted by object-oriented typing, the run-time class of `i[k]` may be a subtype of *T*.

As an example, consider the declarations and method calls below:

```
@IntentMap({@Extra(key = "picture", source = {FILESYSTEM}, sink = {INTERNET}),
            @Extra(key = "location", source = {FILESYSTEM}, sink = {DISPLAY})})
Intent i = new Intent();

@Source(FILESYSTEM) @Sink(INTERNET) File getPicture() {...}
@Source(ACCESS_FINE_LOCATION) @Sink(INTERNET) String getLocation() {...}

void fillIntent() {
    i.putExtra("picture", getPicture());        // Legal.
    i.putExtra("someRandomKey", getPicture());  // Violates requirement (1).
    i.putExtra("location", getLocation());      // Violates requirement (2).
    ...
}

void processDataFromIntent() {
    // pic will have source type FILESYSTEM and sink type INTERNET.
    File pic = i.getExtra("picture", null);               // Legal.
    // loc will have source type FILESYSTEM and sink type DISPLAY.
    String loc = i.getStringExtra("location", null);      // Legal.
    Object randomObject = i.getExtra("someRandomKey");    // Violates requirement (1)
    ...
}
```

The type of variable `i` indicates that this object may contain up to two elements in its map which can be accessed, one with key `"picture"`, source type `FILESYSTEM`, and sink type `INTERNET`, and another with key `"location"`, source type `FILESYSTEM`, and sink type `DISPLAY`. This object may contain more elements but they cannot be accessed. The method calls in the method `fillIntent` shows that it is only valid to invoke putExtra if the value passed as argument is a subtype of the declared type for the corresponding key. In the method `processDataFromIntent`, the variables `pic` and `loc` will have their source and sink types inferred from the type of `i`. Trying to access key `"someRandomKey"` violates requirement (1).

Subtyping

Intent type *T1* is a subtype of intent type *T2* if the key set of *T2* is a subset of the key set of *T1* and, for each key *k* in both *T1* and *T2*, *k* is mapped to the exact same type, that is, $T1[k] = T2[k]$.

sendIntent calls and copyable rule

A `sendIntent` call can be viewed as an invocation of one or more `onReceive` methods. A `sendIntent` call type-checks if its intent argument is copyable to the formal parameter of each corresponding `onReceive` methods. Copyable is a subtyping-like relationship with the weaker requirement: `T1[k] <: T2[k]` instead of `T1[k] = T2[k]`. This is sound because the Android system passes a copy of the intent argument to `onReceive`, so aliasing is not a concern.

Overriding and calling onReceive methods

Every `onReceive` method has a parameter of type `Intent`, and this parameter must be annotated with `@IntentMap` and `@Extra`.

The normal Java overriding rules do not apply to declarations of `onReceive`. The type of the formal parameter of `onReceive` is not restricted by the type of the parameter in the overridden declaration. This is allowable because by convention `onReceive` is never called directly but rather is only called by the Android system via a `sendIntent` method call. The Intent Checker prohibits direct calls to `onReceive` methods.

There is a peculiarity for the `onReceive` method in Activities, `setIntent`. Every Activity that calls the method `getIntent` must override both methods `setIntent` and `getIntent`. The return type of `getIntent` must be annotated with the same type as the formal parameter of `setIntent`, so that when the method `getIntent` is called the correct type is returned.

Chapter 5

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?

5.1 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.

5.2 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.

5.3 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 6

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 6.1
3. Run reverse-engineering tools; see Section 6.2
4. Write and check information flow type qualifiers; see Section 6.3

6.1 Write a flow-policy file

Write a flow-policy file. Section 3.4 describes flow policies.

6.1.1 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.

6.1.2 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 6.1.1. 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 6.2.3)
- 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.

6.2 Run reverse-engineering tools

6.2.1 Review constant strings used

Run

```
ant report-strings
```

to get a list of all constant strings used in the the program grouped by category. A file, `found-strings.txt` in the `sparta-out` directory, lists a summary of all strings used in the app grouped by category. Categories are URLs, content URIs, class names, file or path names, SQL statements, messages to the user, and no category. Each string is only assigned to one category. This ant target also reports where in the code the string was found. For example,

```
.../MapActivity.java:41: warning: [found.url]
          intent.setData(Uri.parse("http://darknessmap.com"));
                                   ^
   Possible URL string: http://darknessmap.com
```

shows where a URL string is found in the found.

6.2.2 Review suspicious code and API uses

Run

```
ant report-suspicious
```

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.

6.2.3 Review where permissions are used in the application

Run

```
ant report-permissions
```

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

6.3 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 6.1.

Section 6.3.1 describes how to annotate libraries, Section 6.3.2 and Section 6.3.3 describe how to annotate your own code in a bottom up approach, and Section 6.3.6 describes how to iteratively trace sensitive sources in your application.

6.3.1 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 report-not-reviewed` 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 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 3.5.4 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. 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.

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(READ_PHONE_STATE) String getLine1Number();
    public @Source(READ_PHONE_STATE) 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`).

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.

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

6.3.2 Infer information flow types for the app

Use whole program inference to infer annotations for method signatures and fields. These inferred annotations are written back into source code for use in type checking. In order to use whole program inference, you first must download an install the inference framework. To do so, clone the repository and update to the correct revision using the following commands,

```
hg clone -b remove-scala https://code.google.com/p/checker-framework-inference
hg update -r 4c9ab282f6ba
```

then follow the instructions in README.sparta to install.

Run inference to annotate an app:

```
ant infer
```

The inference framework is not complete, for example, it does not correctly infer annotations for generic types, so you will need to correct annotations. The inference framework will not change annotations that already appear in source code, so you may wish to remove the inferred annotations, add some manual annotations, and then re-run inference.

6.3.3 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.

Review the found forbidden flows in `sparta-out/forbiddenFlows.txt`. Flows found in the app but not list in the flow policy will appear in this file. You may wish to add these flows to the policy.

For easier to read warning messages, use the `check-flow-pp` ant target.

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).

6.3.4 Check for implicit information flow via conditionals

Because the then clause of a conditional only executes if the predicate is true, a write to a sink in the then clause may leak that the predicate evaluated to true. If the predicate contains information from a sensitive source, then the body of a conditional may leak some fact about the sensitive source. This possible information flow is considered implicit.

Run the conditional checker, after the flow checker reports no more warnings.

```
ant check-conditionals
```

This ant target outputs a warning in every location a sensitive sources is used in a predicate. For each warning, you should manually verify that the source is not implicitly leaked to some sink.

6.3.5 Type-check information flow types across communicating components

This section explains how to generate a component map file for an app, how to run the Intent Checker for this app, and how to identify which of the raised warnings are false warnings.

Component map file generation

The component map contains information about inter-component communication of an app. To type-check information flow types across communicating components of an app, a component map file must be generated for that app. The steps to generate a component map file are:

1. Run the `generate-cm` script, located in the `sparta-code` folder. This script receives as argument the path of the target app's root folder, followed by a set of .apk files of apps that might communicate with the target app. It is not necessary to pass the .apk of built-in Android apps as arguments. Below is an example running the script from the `sparta-code` folder:

   `./generate-cm` *`target_app's_root_folder [app1.apk app2.apk ... appN.apk]`*

   Example of a generated component map file:

   ```
   com.VideoActivity.onOptionsItemSelected(MenuItem) -> com.VideoActivity.PictureActivity
   com.VideoActivity.GCMRegistrar.internalRegister(String) -> com.gtalkservice.PushMessage

   #Broadcast Receivers:
   #Inspect the method com.VideoActivity.AboutUs.onDestroy(Bundle)
   #and replace the "BroadcastReceiver registed in..." text below by
   #the fully-qualified name of the BroadcastReceiver registered in
   #that location.
   #Also, remove the "Update Line: " prefix.

   Update line: com.VideoActivity.AboutUs.onDestroy(Bundle) -> BroadcastReceiver
                       registed in com.VideoActivity.DisplayService.onBind(Intent)

   #Intents assigned at run time:
   #Inspect the method com.VideoActivity.AboutUs.onCreate(android.os.Bundle)
   #and replace the RUN_TIME_ASSIGNED text below by
   #the fully-qualified names of the components that might
   #receive an intent sent from com.VideoActivity.AboutUs.onCreate(android.os.Bundle).
   #Also, remove the "Update Line: " prefix.

   Update line: com.VideoActivity.AboutUs.onCreate(android.os.Bundle) -> RUN_TIME_ASSIGNED

   #Intents using URIs:
   com.VideoActivity.DisplayService$3.run() -> com.VideoActivity.MyBringBack
   com.VideoActivity.DisplayService$3.run() -> com.VideoActivity.SoundListActivity

   #No receiver found for these intents:
   com.VideoActivity.AboutUs.onPause(Bundle) -> RECEIVER_NOT_FOUND
   ```

2. After the component map file is generated you may need to refine it if receiver components could not be resolved in some cases. To check if your `component-map` needs a refinement, open it and look for lines starting with: `Update line:`

   If the component map has at least one occurrence of `Update line:`, it must be refined. The component map is divided into comment sections. Below is explained each comment section, and how to update each line when necessary.

- `#Broadcast Receivers:` BroadcastReceivers can be dynamically registered to receive intents. The generated component map shows a method `m` where a BroadcastReceiver is registered. The user must manually inspect `m` and replace `"BroadcastReceiver registered in m"` in the component map file by the fully-qualified names of the BroadcastReceivers registered in that method. The example above suggests that the user must inspect the method `onBind(Intent)` from the class `DisplayService`, and replace `"BroadcastReceiver registered in com.VideoActivity.DisplayService.onBind(Intent)"` by the fully-qualified names of the BroadcastReceivers registered in that method.
- `#Intents assigned at run time:` The receiver component for this intent is assigned at run time and could not be statically resolved. A manual inspection is needed to update these cases. In the example above the `onCreate(Bundle)` method of the class `AboutUs` must be inspected to find out the receiver components of this intent. The `RUN_TIME_ASSIGNED` string in this line should be replaced by the fully-qualified names of the receiver components assigned at run time.
- `#Intents using URIs:` The component map generation does not differentiate intent filters with the same action and categories but different data (URI). You must manually inspect the intent-sending method and discover the receiver components. Every communication that uses URIs will be shown in the component map file so that the you can comment the ones you are certain that are not receiving an intent from that intent-sending method.
- `#No receiver found for these intents:` There are no receiver components for these intent-sending methods in the set of apps provided in the component map generation.

Annotating components from other applications

In case the app being analyzed communicates with components from other apps, these components need to be correctly annotated and you need to add their .java files in the `sparta-code/annotated-intent-receivers`, or in the target app's source code directory. You should stick with the first option if the component you are adding is an Android system component, so it can be used in future analyses.

Running the Intent Checker with the Information Flow Checker

The Intent Checker type-checks information flow types across communicating components and within components. To run the Intent Checker with the Information Flow Checker from the command line use the command `check-intent` instead of `check-flow`:

```
ant [-DflowPolicy=myflowpolicyfile] [-DcomponentMap=mycomponentmapfile] check-intent
```

Common warnings and errors in the Intent Checker

Error: intent.receiver.notfound:

```
tests/intent/ActivityTest.java:118: error: [intent.receiver.notfound]
            startActivity(i);
                          ^
Could not find receivers for the intent sent in method:
tests.ActivityTest.receiverNotFound(). Regenerate the component map file or add
a line in it from this method to one or more receiver components of that intent.
```

This error occurs when type-checking a `sendIntent` call whose method in which it is called is not in the component map file. This happens when the component map file wasn't correctly generated or when it is necessary to do a manual inspection to add the correct receivers of an intent.

To solve this error, regenerate the component map file. If the error persists you need to manually inspect the method stated in the error message, look for intents sent in that method, identify the receiver components, and add add an entry in the component map file from this method to the receiver components. For example:

```
com.package.ActivityA.foo() -> com.package.ActivityB
or
com.package.ActivityA.foo() -> RECEIVER_NOT_FOUND
```

Use the later if the intent is not being received by any component.

Warning: send.intent.missing.key:

```
tests/intent/ActivityTest.java:128: warning: [send.intent.missing.key]
            startActivity(senderIntent1);
                          ^
There is a type mismatch in the intent types of senderIntent1 and
tests.ActivityReceiverStub.setIntent()'s intent parameter.
Key "k5" is either missing from the intent type of senderIntent1, or should not be present in
the intent type of com.package.ActivityReceiver.receiveIntent.setIntent()'s intent parameter.

senderIntent1's intent type:
@IntentMap({@Extra(key="k2", source={ACCESS_FINE_LOCATION}, sink={})}).
tests.ActivityReceiverStub.setIntent()'s intent parameter intent type:
@IntentMap({@Extra(key="k5", source={ACCESS_FINE_LOCATION}, sink={})}).
```

To solve this warning you need to verify how this key is used in the receiver component, to make sure that it makes sense for the receiver to have it in its intent type. If that is the case, you need to add an `@Extra` with this key in the sender's intent type, otherwise remove the `@Extra` with this key from the receiver's intent type.

Warning: send.intent.incompatible.types:

```
tests/intent/ActivityTest.java:107: error: [send.intent.incompatible.types]
            startActivity(senderIntent2);
                          ^
There is a type mismatch in the intent types of senderIntent2 and
tests.ActivityReceiverStub.setIntent()'s intent parameter. The @Extra with key "k5"
in the intent map of senderIntent2 must be a subtype of the @Extra with key "k5" in
the intent map of tests.ActivityReceiverStub.setIntent()'s intent parameter.

senderIntent2's intent type:
@IntentMap({@Extra(key="k5", source={ANY}, sink={})}).
tests.ActivityReceiverStub.setIntent()'s intent parameter intent type:
@IntentMap({@Extra(key="k5", source={ACCESS_FINE_LOCATION}, sink={})}).
```

This warning occurs when an `@Extra` from the sender's intent type has `@Source` and/or `@Sink` types that are not subtypes of the receiver's `@Extra` with same key.

To solve this warning you must first understand what should be the correct intent type of both intent variables by checking how the key mentioned in the warning is used in each component. You will need to either modify the intent type of the sender, or the intent type of the receiver such that the `@Extra` with the key mentioned in the warning has the same type in both intent types. For the example above, you could either modify the `@Source` type of the `@Extra` with key `"k5"` in `senderIntent2`'s intent type to `ACCESS_FINE_LOCATION`, or modify the `@Source` type of the `@Extra` with key `"k5"` in `onStartCommand()'s intent parameter`'s intent type to `ANY`.

Warning: intent.key.notfound:

```
 tests/intent/ConstCheckerTest.java:44: warning: [intent.key.notfound]
         String test1 = i1.getStringExtra("k2");
                                          ^
Invalid access: key "k2" is not in the @IntentMap annotation of i1.
```

This warning occurs when a key `"k"` is not in the intent type of an intent variable `i` and either `i.getExtra("k")` or `i.putExtra("k",...)` is called.

Warning: argument.type.incompatible:

```
 IntentMapBottomTest.java:60: warning: [argument.type.incompatible]
 incompatible types in argument.
         intentMapBottom1.putExtra("RandomKey1", getFile());
                                                 ^
found   : @Sink(FlowPermission.INTERNET) @Source(FlowPermission.FILESYSTEM) String
required: @Sink(FlowPermission.ANY) @Source({}) String
```

This warning occurs when an object is inserted into an intent's map but it does not have the expected `@Source` and/or `@Sink` type according the intent type.

To solve this warning you must first understand the `@Source` and `@Sink` types of the object that is being inserted in the intent and annotate it correctly. After that, you should modify the `@Extra` of the intent type with that key so that the type of `@Extra` will match the object's type. Considering the example above, you could either modify the `@Extra` with key `"RandomKey1"` to have its `@Source` as `FILESYSTEM` and its `@Sink` as `INTERNET`, or annotate the return type of `getFile()` with `@Source()` and `@Sink(ANY)`.

6.3.6 Trace information flows

On execution, the Information Flow Checker creates a file called `forbiddenFlows.txt` in the `sparta-out` 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.

- 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 all flows, select a source from the `forbiddenFlows.txt` to trace. Use flow filtering to display all forbidden flow locations for the selected source.

```
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 6.3.3.

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).

6.3.7 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.

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, install the inference framework see Section **??** for details.

Infer types, `ant infer`. Then, run the Information Flow Checker, `ant check-flow-pp`, if the output is similar to the output shown below, then your setup is correct. (You should get 31 warnings.)

```
Buildfile: .../ContactManager/build.xml
...
check-flow-pp:
Compiling 4 source files to /Users/smillst/Downloads/ContactManager/bin/classes
javac 1.8.0-jsr308-1.8.2
.../ContactAdder.java:64: warning: [forbidden.flow]
    private @Source({}) @Sink({}) ArrayList<@Source({FlowPermission.ANY}) @Sink({FlowPermission.CONTENT_PR
                                            ^
  flow forbidden by flow-policy
  found:  { ANY  ->  CONTENT_PROVIDER }  Integer
  forbidden flows:
    ANY -> CONTENT_PROVIDER
```

7.2 Drafting a flow policy

The Information Flow Checker outputs a file, `sparta-out/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 -> CONTENT_PROVIDER
 CONTENT_PROVIDER -> DISPLAY
 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. Because some of the inserted annotations are too permissive, so of the forbidden flows involve ANY.

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

```
CONTENT_PROVIDER -> DISPLAY
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 17 warnings. The `forbiddenFlows.txt` file should also have changed as shown below.

```
# Flows currently forbidden
ANY -> CONTENT_PROVIDER
```

Since this flow contains ANY, there is nothing to add to the flow policy.

Because USER_INPUT→CONTENT_PROVIDER and CONTENT_PROVIDER→DISPLAY are legal flows, the Information Flow Checker reports a possible transitive flow, USER_INPUT→DISPLAY see Section **??**.

```
warning: FlowPolicy: Found transitive flow
  [USER_INPUT]->[DISPLAY]
  Please add them to the flow policy
```

Add USER_INPUT→DISPLAY to the flow policy.

7.3 Correcting Annotations

Inference sometimes inserts annotations that are correct, but too permissive. It may infer the ANY instead of a more specific permission. Because we did not add any flow with ANY to the flow policy, all uses of these types will issue a forbidden flow warning.

Run the Information Flow Checker and filter for forbidden flows.

```
ant check-flow -DcfOpts=-AmsgFilter=forbidden.flow

.../ContactAdder.java:64: warning: [forbidden.flow]
    private @Source({}) @Sink({}) ArrayList<@Source({FlowPermission.ANY}) @Sink({FlowPermission.CONTENT_PR
                                            ^
  flow forbidden by flow-policy
  found: @Sink(FlowPermission.CONTENT_PROVIDER) @Source(FlowPermission.ANY) Integer
  forbidden flows:
    ANY -> CONTENT_PROVIDER
.../ContactAdder.java:68: warning: [forbidden.flow]
    private @Source({}) @Sink({}) ArrayList<@Source({FlowPermission.ANY}) @Sink({FlowPermission.CONTENT_PR
                                            ^
  flow forbidden by flow-policy
  found: @Sink(FlowPermission.CONTENT_PROVIDER) @Source(FlowPermission.ANY) Integer
  forbidden flows:
    ANY -> CONTENT_PROVIDER
```

Remove `@Source(FlowPermission.ANY)` from both locations.

7.4 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 3.7
   - Is the annotation inferred? Section 3.5.1
   - Is the annotation defaulted? Section 3.5.3
2. Why is the found type not a subtype of the required type? Section 3.1.2
   - 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.

7.4.1 Warning 1

Run the Information Flow Checker again.

```
.../ContactAdder.java:96: warning: [assignment.type.incompatible] incompatible types in assignment.
        mContactPhoneTypes = new ArrayList<Integer>();
                             ^
  found   :  {   ->  ANY } ArrayList< {   ->   }  Integer>
  required:  {   ->   } ArrayList< { USER_INPUT  ->  CONTENT_PROVIDER }  Integer>
```

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 `new ArrayList<Integer>()` is the *found* type and the type of `mContactPhoneTypes` is the *required* type.
   - Was it explicitly annotated in the source code? `mContactPhoneTypes` was, but `new ArrayList<Integer>()` was not.
   - Is it from an API method that was annotated in stub file? No
   - Was it inferred from an assignment? No
   - Was it defaulted? Yes, `new ArrayList<Integer>()`

According to the defaulting rules explained in Section 3.5.3 constructor results are annotated with `@Source({})` `@Sink({})` and type arguments are annotated with `@Source({}) @Sink({})` by default. Notice that the source annotation on `mContactPhoneTypes` has been defaulted based on the flow policy. USER_INPUT is the only source allowed to flow to CONTENT_PROVIDER.

2. **Why is the found type not a subtype of the required type?** The primary qualifiers on `mContactPhoneTypes`, `@Source({}) @Sink({})` is a super type of the primary qualifiers on `new ArrayList<Integer>()`, `@Source({})` `@Sink(ANY)`. The qualifiers on the type arguments must be the same. For more details, see `http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#generics`.
3. **What annotation or annotations would make the found type a subtype of the required?** Because the annotations on `mContactPhoneTypes` were inferred, annotations should be add to `new ArrayList<Integer>()` so that the type argument matches `mContactPhoneTypes`

```
mContactEmailTypes = new ArrayList<@Sink(CONTENT_PROVIDER) Integer>();
```

The source annotation will be defaulted to `@Source(USER_INPUT)` .

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

7.4.2 Warning 2

```
.../ContactAdder.java:101: warning: [assignment.type.incompatible] incompatible types in assignment.
        mContactEmailTypes = new ArrayList<Integer>();
                             ^
  found   :  {   ->  ANY } ArrayList< {   ->   }  Integer>
  required:  {   ->   } ArrayList< { USER_INPUT  ->  CONTENT_PROVIDER }  Integer>
```

This warning is nearly identical to the previous warning and can be corrected the same way.
Run the Information Flow Checker again. Only two errors should be issued.

7.4.3 Warning 3

```
.../ContactAdder.java:262: warning: [argument.type.incompatible] incompatible types in argument.
                    accountTypes);
                    ^
  found   :  {   ->   } AuthenticatorDescription  {   ->  ANY }  []
  required:  {   ->  ANY } AuthenticatorDescription  {   ->  ANY }  []
```

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

1. **Where do the found and required types come from?** The found type is the type the local variable , `accountTypes` . The annotation on the array type (i.e. the annotation before `[]`) was refined after the assignment on line 254. Annotations on the element array types (i.e. the annotation before `AuthenticatorDescription`) are never refined, so this annotation was defaulted.
   The required type is the type of the `dictionary` parameter of the method `getAuthenticatorDescription`. The source annotation were inferred, but the sink annotations were not.
2. **Why is the found type not a subtype of the required type?** The element array types are not equal.
3. **What annotation or annotations would make the found type a subtype of the required?** Either the declaration of `accountTypes` needs its array elements type annotated with `@Sink(ANY)` or the del cation of the `dictionary` needs its array elements type annotated with `@Sink({})` . If the type of `accountTypes` is changed the assignment on line 256 will fail. Because of this and because the `@Sink({})` is less permissive, the method signature should be update.
   The inference tool did not infer any sink annotation for this method, so they should be add to all types.

```
 private static @Source({}) @Sink({}) AuthenticatorDescription getAuthenticatorDescription(@Source({})
@Sink({}) String type, @Source({}) @Sink({}) AuthenticatorDescription @Source({}) @Sink({}) []
dictionary)
```

Run the Information Flow Checker; there should be 1 warning.

7.4.4 Warning 4

```
.../ContactAdder.java:313: warning: [assignment.type.incompatible] incompatible types in assignment.
            mName = name;
                    ^
  found   :  {   ->   }  String
  required:  {   ->  CONTENT_PROVIDER, DISPLAY, WRITE_LOGS }  String
```

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. Both types were inferred.
2. **Why is the found type not a subtype of the required type?** The found type sink does not include and the required sinks is CONTENT_PROVIDER, DISPLAY, WRITE_LOGS .
3. **What annotation or annotations would make the found type a subtype of the required?** Assuming the inferred annotations on the field are correct, the annotation on the parameter should be updated to include the required sinks.

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

7.5 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.