Programming Assignment 8: Taint Analysis

Course "Static Program Analysis" @Nanjing University
Assignments Designed by Tian Tan and Yue Li

1 Assignment Objectives

- Implement a taint analysis for Java.

Welcome to the last programming assignment of this course! ヾ(o◕∀◕)ﾉ

In this assignment, you will implement a taint analysis for Java based on the context-sensitive pointer analysis (that you implemented in Assignment 6). In addition, we will teach you a technique, called *taint transfer*, to detect more security vulnerabilities in practice. We have provided a configurable taint analysis framework in Tai-e so that you can conveniently configure sources, sinks, and how taint can be transferred in the program.

Similar to Assignment 7, this assignment is also open. You need to figure out how taint analysis interacts with pointer analysis and how to implement taint transfers by yourself.

2 Implementing Taint Analysis

2.1 Scope

In this section, we define the taint analysis that you need to implement in this assignment. Same as the taint analysis introduced in Lecture 13, in this assignment, we consider calls to the specific methods (typically data source APIs) as taint sources, which return tainted data (also called *taint objects* in the analysis); and certain arguments of specific methods are treated as taint sinks. For better precision, you will implement a *context-sensitive* taint analysis following the rules below to handle sources and sinks (modified based on the rules given in page 76 of slides for Lecture 13):

Kind	**Statement**	**Rule (under context c)**
Call (source)	*l*: `r = x.k(a1,…,an)`	$\frac{c:l \to c^t:m \in CG \quad \langle m,u \rangle \in Sources}{[\,]:t_l^u \in pt(c:r)}$
Call (sink)	*l*: `r = x.k(a1,…,an)`	$\frac{c:l \to c^t:m \in CG \quad \langle m,i \rangle \in Sinks \quad [\,]:t_j^u \in pt(c:ai)}{\langle j,l,i \rangle \in TaintFlows}$

Here, $\mathit{Sources}$ is a set of pairs, denoted as $\langle m, u\rangle$, where m is the signature of the source method, and u is the type of the returned taint object. We use t_l^u to denote a taint object, where u is type of the object, and l is the call site where the object is created. For simplicity, you just need to use *empty* context as the heap contexts of taint objects.

Taint Transfer. Although taint analysis and pointer analysis are similar as they both track data flow in the program, they have a subtle difference. Compared to object, taint is a more abstract concept. Taint is associated with the *contents* of the data[1], so that it can be *transferred* among objects, and such phenomenon is called *taint transfer*. Below we use an example to illustrate this concept.

```
1 String taint = getSecret(); // source
2 StringBuilder sb = new StringBuilder();
3 sb.append("abc");
4 sb.append(taint); // taint is transferred to sb
5 sb.append("xyz");
6 String s = sb.toString(); // taint is transferred to s
7 leak(s); // sink
```

Figure 1. An example of taint transfer.

Suppose that we consider `getSecret()` and `leak()` as source and sink, respectively. In this example, the code at line 1 retrieves secret data (in form of a string) and stores it in variable `taint`, and the secret data will finally flow to sink (line 7) via two taint transfers:

1) The method call to `append()` at line 4 appends the contents of `taint` to `sb`, so the `StringBuilder` pointed to by `sb` contains the secret data and should also be treated as tainted data; in other words, `append()` at line 4 transfers taint from `taint` to `sb`.
2) The method call to `toString()` at line 6 converts the `StringBuilder` to a `String`, which contains the same contents of the `StringBuilder`, so the `String` includes the secret data; in other words, `toString()` transfers taint from `sb` to `s`.

Such patterns are common in real code, and if we cannot handle them properly, many security vulnerabilities would be missed. The reason is that regular taint analysis is unaware of the semantics of the APIs in the program, e.g., methods `append()` and `toString()` can transfer the contents (together with sensitive data) among different objects as shown in the above example, thus the taint analysis would fail to propagate taints without handling these methods properly.

To address this problem, we need to tell taint analysis which *methods* can trigger taint transfers and how they transfer taints. In this assignment, we consider three common patterns of taint transfers when a (taint-transfer-relevant) *method* `foo` is called:

---

[1] Neville Grech and Yannis Smaragdakis, "*P/Taint: Unified Points-to and Taint Analysis*". OOPSLA'17.

1) Base-to-result 2) Arg-to-base

```
result = base.foo(a0, a1, ..., an);
```

3) Arg-to-result

1) **Base-to-result**: if the receiver object (pointed to by `base`) is tainted, then the return value of the method call (pointed to by `result`) should also be tainted, e.g., `StringBuilder.toString()`.
2) **Arg-to-base**: if a specified argument is tainted, then the receiver object (pointed to by `base`) should also be tainted, e.g., `StringBuilder.append(String)`.
3) **Arg-to-result**: if a specified argument is tainted, then the return value of the method call (pointed to by `result`) should also be tainted, e.g., `String.concat(String)`.

Note that static methods will *not* cause base-to-result and arg-to-base transfers as they do not have base variables. Besides, some methods may cause multiple taint transfers, e.g., `String.concat(String)` triggers not only arg-to-result but also base-to-result transfers, as its result contains the contents of both argument and receiver object.

Handling Taint Transfer. The essence of taint transfer is that the method calls to some methods will trigger propagation of taints from specific variables to some other variables of the call sites. We call source of taint transfer *from-variable*, and target of taint transfer *to-variable*, e.g., for a base-to-result transfer, the base variable of the call site is from-variable, and the LHS variable of the call site is to-variable.

We define another input of taint analysis, called *TaintTranfers*, which is a set of four-element tuples, denoted as $\langle m, \mathit{from}, \mathit{to}, u \rangle$, where m indicates the method that triggers taint transfer, from the from variable to the to variable, and u is the type of the transferred taint object (pointed to by to). Specifically,

- m is a signature of the method that triggers taint transfer, and
- from is either an integer value (starting from 0) when it represents an argument, or the string “base” when it represents a base variable, and
- to is either the string “base” when it represents a base variable, or the string “result” when it represents an LHS variable of the call site, and
- u is the type of the transferred taint object. As a taint transfer may change the type of the taint object (e.g., `StringBuilder.toString()` transfers a taint object of type `StringBuilder` to a taint object of type `String`), then we need u to tell the taint analysis what the type of the transferred taint object is. It would be particularly useful when the type of the transferred object (pointed to by to) differs from the type of the taint object pointed to by from.

Based on *TaintTranfers*, we define the rules to handle the three patterns of taint transfers as follows:

Kind	Statement	Rule (under context c)
Call (base-to-result)	`l: r = x.k(a1,…,an)`	$\dfrac{c{:}l \rightarrow c^t{:}m \in CG \quad \langle m, \text{"base"}, \text{"result"}, u\rangle \in TaintTranfers \quad [\,]{:}t_j^{u'} \in pt(c{:}x)}{[\,]{:}t_j^{u} \in pt(c{:}r)}$
Call (arg-to-base)	`l: r = x.k(a1,…,an)`	$\dfrac{c{:}l \rightarrow c^t{:}m \in CG \quad \langle m, i, \text{"base"}, u\rangle \in TaintTranfers \quad [\,]{:}t_j^{u'} \in pt(c{:}ai)}{[\,]{:}t_j^{u} \in pt(c{:}x)}$
Call (arg-to-result)	`l: r = x.k(a1,…,an)`	$\dfrac{c{:}l \rightarrow c^t{:}m \in CG \quad \langle m, i, \text{"result"}, u\rangle \in TaintTranfers \quad [\,]{:}t_j^{u'} \in pt(c{:}ai)}{[\,]{:}t_j^{u} \in pt(c{:}r)}$

Configuration for Taint Analysis. To make the taint analysis flexible, we design a configurable taint analysis which allows you to configure sources, sinks and taint transfers in one YAML[2] file. As an example, you could read `src/test/resources/pta/taint/taint-config.yml` in this assignment package.

The format of a source entry is:

```
{ method: <METHOD_SIGNATURE>, type: <TYPE_NAME> }
```

where

- `<METHOD_SIGNATURE>` is the signature of the source method
- `<TYPE_NAME>` is the name of the type of taint object returned from the call to the source method

In pointer analysis, each object has a type, so do taint objects. We need to specify the types of the taint objects in the configuration, as the taint analysis should create a taint object of this type when handling the calls to the source method.

The format of a sink entry is:

```
{ method: <METHOD_SIGNATURE>, index: <INDEX> }
```

where

- `<METHOD_SIGNATURE>` is the signature of the sink method
- `<INDEX>` is the index of the sensitive argument, starting from 0 (typically, only arguments are considered as sinks)

[2] https://yaml.org/

The format of a taint transfer entry is:

```
{ method: <METHOD_SIGNATURE>, from: <INDEX>, to: <INDEX>,
                    type: <TYPE_NAME> }
```

where the four elements exactly correspond to the ones in *TaintTranfers* defined above.

2.2 Tai-e Classes You Need to Know

The classes related to context-sensitive pointer analysis have been introduced in Assignment 6. Below we introduce the classes that are specific to taint analysis.

- `pascal.taie.analysis.pta.plugin.taint.Source`
  This class represents sources.

- `pascal.taie.analysis.pta.plugin.taint.Sink`
  This class represents sinks.

- `pascal.taie.analysis.pta.plugin.taint.TaintTransfer`
  This class represents taint transfers. In this class, we use integers to identify from- and to-variables. Specifically, if the value of the integer is 0 or larger number, it represents the corresponding argument of a call site (of the method specified in the `TaintTransfer`); if the value is -1, it represents the base variables of a call site; if the value is -2, it represents the LHS variables of a call site.

- `pascal.taie.analysis.pta.plugin.taint.TaintConfig`
  This class represents configuration of taint analysis. It provides APIs to parse configuration file and obtain the sources, sinks, and taint transfers specified in the configuration.

- `pascal.taie.analysis.pta.plugin.taint.TaintManager`
  This class manages taint objects in taint analysis.

- `pascal.taie.analysis.pta.plugin.taint.TaintFlow`
  This class represents the detected taint flows (described by the call sites of the taint source and sink), i.e., the result of taint analysis.

- `pascal.taie.analysis.pta.plugin.taint.TaintAnalysiss`
  This class implements taint analysis. It is incomplete, and you need to finish it as explained in Section 2.3. (Note that the class name is `TaintAnalysiss` in this assignment as `TaintAnalysis` has already been used in the non-assignment-version of taint analysis in Tai-e :-/)

2.3 Your Task [Important!]

In this assignment, you need to finish the methods of two classes listed below:

`pascal.taie.analysis.pta.cs.Solver`:

- `void addReachable(CSMethod)`
- `void addPFGEdge(Pointer,Pointer)`
- `void analyze()`
- `PointsToSet propagate(Pointer,PointsToSet)`
- `void processCall(CSVar,CSObj)`

`pascal.taie.analysis.pta.plugin.taint.TaintAnalysiss`

- `Set<TaintFlow> collectTaintFlows()`: returns a set that contains all taint flows detected by the taint analysis. You could implement the rule to handle sink (given in Section 2.1) in this method.

The five methods of `Solver` to be finished are the same as in Assignment 6, but this time, you need to add some code to some of these methods for supporting taint analysis. Do *not* directly replace `Solver.java` by your implementation of Assignment 6, as the skeleton file `Solver.java` in this assignment contains some code related to taint analysis.

In this assignment, you may need to read points-to results to help develop and debug taint analysis. Other context sensitivity variants add context information to points-to results, which may increase reading difficulty. Thus, we choose `CISelector` (context insensitivity) as the default context selector to ease the development and debugging. After you finish `TaintAnalysiss` and `Solver`, you could try other context sensitivity variants as explained in Section 3 to observe the precision differences of taint analysis under different context-sensitivity variants.

As for `TaintAnalysiss`, in addition to `collectTaintFlows()`, you also need to implement the logics to handle sources and taint transfers in this class. Again, this assignment is open, and thus you need to resolve the implementation details by yourself, including how to design and implement your APIs of `TaintAnalysiss`.

Hints: 1) In the constructor of `TaintAnalysiss`, we have provided the code to parse configuration file and store the result in field `config`, so that you could directly use it. Besides, we initialize a `TaintManager` and store it in field `manager`, and you could use it to manage taint objects. If your implementation of `TaintAnalysiss` requires initialization work, you could also do it in the constructor.

2) In this assignment, pointer and taint analyses depend on each other. Both `Solver` and `TaintAnalysiss` hold a reference to each other, i.e., field `taintAnalysis` in

`Solver` and field `solver` in `TaintAnalysiss`. You need to figure out how to use the references to implement the interactions between two analyses. You can add APIs and fields to *both* classes if necessary.

3 Run and Test Your Implementation

You can run the analyses as described in *Tai-e Manual for Assignments*. In this assignment, Tai-e performs context-sensitive pointer analysis and taint analysis together for the program, and outputs the points-to sets of all kinds of pointers and detected taint flows:

```
Points-to sets of all variables

Points-to sets of all static fields

Points-to sets of all instance fields

Points-to sets of all array indexes

Detected 0 taint flow(s):
```

Points-to sets are empty and none of taint flows are detected as you have not finished the analyses yet. After you implement the analyses, the output should be like (points-to results are omitted):

```
Detected 4 taint flow(s):
TaintFlow{<SimpleTaint: void main(java.lang.String[])>[0@L4] temp$0 = invokestatic <SourceSink: java.lang.String source()>(); ->
 <SimpleTaint: void main(java.lang.String[])>[2@L5] invokestatic <SourceSink: void sink(java.lang.String)>(s1);/0}
TaintFlow{<SimpleTaint: void main(java.lang.String[])>[0@L4] temp$0 = invokestatic <SourceSink: java.lang.String source()>(); ->
 <SimpleTaint: void main(java.lang.String[])>[16@L11] invokestatic <SourceSink: void sink(java.lang.String,int)>(s3, %intconst0);/0}
TaintFlow{<SimpleTaint: void main(java.lang.String[])>[3@L7] temp$1 = invokestatic <SourceSink: java.lang.String source()>(); ->
 <SimpleTaint: void main(java.lang.String[])>[5@L8] invokestatic <SourceSink: void sink(java.lang.String)>(s2);/0}
TaintFlow{<SimpleTaint: void main(java.lang.String[])>[3@L7] temp$1 = invokestatic <SourceSink: java.lang.String source()>(); ->
 <SimpleTaint: void main(java.lang.String[])>[16@L11] invokestatic <SourceSink: void sink(java.lang.String,int)>(s3, %intconst0);/0}
```

In addition, Tai-e outputs the IRs for the classes of the program it analyzes to folder `output/`. The IRs are stored as .tir files which can be opened by general text editors.

We provide class `pascal.taie.analysis.pta.TaintTest` as the test drivers for taint analysis, and you could use it to test your implementation as described in *Tai-e Manual for Assignments*. Note that in this assignment, we only compare detected taint flows with the ones given in the expected files. Other analysis results, e.g., points-to sets, are ignored. Some test cases require context-sensitive pointer analysis, thus, to pass all test cases, you need to copy the six context selectors from Assignment 6.

We encourage you to analyze some test cases (e.g., `TaintInList.java`) with other context sensitivity variants (as explained in Assignment 6), e.g., context insensitivity and 2-object sensitivity, and observe how the precision of pointer analysis affects the precision of taint analysis.

4 General Requirements

- In this assignment, your only goal is correctness. Efficiency is not your concern.
- **DO NOT** distribute the assignment package to any others.
- Last but not least, do **NOT** plagiarize. The work must be all your own!

5 Submission of Assignment

Your submission should be a zip file, which contains your implementation of

- `TaintAnalysiss.java`
- `Solver.java`

The naming convention your submission is: <STUDENT_ID>-<NAME>-A8.zip
Please submit your assignment to 教学立方.

6 Grading

The points will be allocated for correctness. We will use your submission to analyze the given test files from the `src/test/resources/` directory, as well as other tests of our own, and compare your output to that of our solution.

Good luck!