CS510 Project Handout: Provenance Tracking Tool

Feb 6, 2017

Project Description

In this project, you are asked to build a provenance tracking tool for Valgrind [2, 3, 4]. Provenance tracking is a process of marking and tracking input data in a program at *runtime* in order to identify all dependencies on those values at certain execution points. This type of dynamic analysis is becoming increasingly popular in the context of software testing, debugging and system security.

Provenance sources provenance sources are initial entry points of information to your program. Depending on the application context, many different types of sources could be defined. It includes standard input (*stdin*), files , and network data, etc. In this project, we choose only the standard input (stdin) and files as provenance sources. You must devise a forward dynamic algorithm (similar to HW2-Q2) and maintain a proper data structure to keep track of data provenance in memory locations and registers. The goal is to trace back all provenance targets to the specified provenance sources.

Provenance Targets We consider program variables or function pointers as provenance targets. Therefore, we are interested in finding their dependency on the specified provenance sources.

Output Format The output trace-file must be a list of log records in the following format:

```
0x####### [DD/CD]: [0x########:value] [all other memory locations and values]
```

- The first column specifies the memory address of a provenance target. You have to repeat the log records to cover all provenance targets.
- The second column shows whether this memory location is dependent on a provenance source due to a data dependency `DD` or control dependency `CD` or both. This field is mandatory for those who work in teams.

- The remaining columns show pairs of memory addresses of provenance sources and their corresponding values. These are the memory addresses that keep the inputs from stdin or files.

For more technical details on Valgrind, please read [4] and refer to the technical documentations at [3]. In addition, the course slides on information flow system (week 5) are particularly useful for this project [1].

Handling Data Dependency

Provenance tracking based on data dependency accounts only for explicit propagation of information.

Example :

```
int a, b, x, y, z;
input(&a);//a = 3
input(&b);//b = 5
x = b * 3;
y = x - a;
z = x + y;
```

In this example, provenance of `x` is {<address of b>:5}; Provenance of `y` and `z` are {<address of b>:5 , <address of a>:3}.

Handling Control Dependency

Provenance tracking based on control dependency accounts for implicit propagation of information in addition to explicit information propagation.

Example :

```
input(&a); a=4
int x, y = 0;
if (a > 5) {
        x = 10;
}
else {
x = 0;
}
print(x);

```

In this example, the value of `a` at line 1 is taken from input (provenance source). Although `a`'s value is not involved in the computation of `x`, it nevertheless affects `x`'s value through control dependency. The outcome of the predicate at line 3 decides whether line 4 or line 7 will be executed next. Therefore, the provenance of `x` is {<address of a>:4}. Conversely, the provenance of variable `y` remains empty.

As mentioned before, a general approach to detect implicit information flow is based on concept of control dependence. When a conditional branching statement *br* decides about the execution of a statement *st*, the values that affect *br*'s outcome may affect the value of the data modified by *st*. Therefore, the *st*'s destination operands must consider the provenance of it's operands. To achieve this result, one approach is to keep track of relevant provenance data at runtime by leveraging statically-computed *postdominance* information.
The second question of HW2 gives another interesting example.

Suggested Steps *(Note: you are NOT required to follow this step-by-step. Since the project is not trivial, we recommend you to start early.):*

1. Familiarize yourself with running programs inside Valgrind by using memcheck, cachegrind etc;
2. Familiarize yourself with how a Valgrind tool works by studying the *none* and *lackey* tools. For example, you need to understand what code you write is executed at instrumentation time and what code you write is executed at program execution time. You can just modify the code of the *none* module to implement your own tool to run inside Valgrind;
3. Familiarize yourself with the Valgrind IR (called *VEX IR*) and its associated header files. Most of this milestone involves understanding the internal representation enough to perform the next steps;
4. Decide how to trace data dependence, and decide what data structure to use (maybe a graph? if so, what data to keep inside each node?) For example, you need to know how to allocate memory for data structures for instrumentation code, and for instrumented code;
5. Perform explicit provenance tracking based on data dependence.
6. Provenance tracking to handle control dependency (optional, if you are doing this project individually);

Further Instructions

- Your tool will take two arguments specifying the input program to analyze and output trace-file. The invocation will be similar to this:
  `valgrind --tool=PTool --target-file==<full_path> --trace-file=<full_path>`
- If your linux kernel supports address space layout randomization, you need to disable it using this command: `echo "0" > /proc/sys/kernel/randomize_va_space` to get similar results at different executions.
- We assume that you have access to the source programs and you can control the compiler options in generating object code that will be executed inside Valgrind.

Grading

Due Date: 24 April 2017, 11:59pm

You can work in a group of two members and both group members will receive the same grade. Expectation of the the deliverables:

- [**1 person**] a provenance tracking tool which handles data dependency
- [**2 persons**] a provenance tracking tool which handles both data dependency and control dependency

Your grade will be mainly based on the correctness of your implementation. We will also use a mixture of face-to-face demonstration, efficient implementation, and your write up of the project to grade.

References

[1] Implementing information flow system on Valgrind. `http://www.cs.purdue.edu/homes/xyzhang/spring17/5-slicing-IFS.updated.pdf`.

[2] Valgrind Mailing List. `http://valgrind.org/support/`.

[3] Valgrind Technical Documentation. `http://valgrind.org/docs/manual/tech-docs.html`.

[4] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.