Fun With BAP!
Due March 12.

Startdate 2015-03-10.16:39

1 Overview

The goal of this assignment is to:

- Gain experience using BAP, including writing unit tests, debugging, and using the new plugin system.
- Build a function for computing the *call strings* for a binary program. Call strings are used when performing a context-sensitive analysis.
- Become familiar with test driven development.

New!:

- The current plugin system doesn't allow you to pass parameters. In your library, please allow for a k parameter. However, in the actual *plugin*, you should hard-code a value. Please use $k = 3$. (2015-03-10.16:39)
- We currently do not have access to the PLT to determine whether a call is external or not. Also, I did not specify what happens on indirect calls (or indirect jumps). For now, if you have a function that has no calls, then it is terminal. If you have a function that makes an unresolved call, then also label it terminal. A few notes:
  - LLVM may say something is not a call, but there is a jump to another symbol. We consider such jumps "calls". (2015-03-10.16:39)
  - After you have labeled terminals as above, feel free to play around with better labels, such as which nodes are indirect calls and such. However, please only consider this after everything else is working, and consider it an optional exercise. (2015-03-10.16:39)
  - We now include a revision/stardate in the assignment! (2015-03-10.16:39)

2 Call Strings

In this assignment we will focus on call strings, which are one of the ways to create a context-sensitive program analysis. The following description is based on the Dragon book 2nd edition, chapter 12.1.3.

Recall that a context sensitive analysis distinguishes between different calling contexts in a program. A *calling context* for a location is defined by the contents of the entire call stack up to that point. The string of call sites on the stack is referred to the *call string.*

In security, like optimization, we want to distinguish between calling contexts. For example, some calls to `strcpy` may be safe, and some not. We can use the calling context to distinguish between safe and unsafe call chains.

The first thing we have done is label each call site with a unique integer, herein denoted by c_i. Consider the following program, slightly modified from example 12.6 in the Dragon book:

Listing 1: Example 1

```
     void main() {
       int t1, t2, t3;
c1   t1 = g(0);
c2   t2 = g(243);
c3   t3 = g(255);
     }

     int g(int v) {
c4    return f(v);
     }

     int f(int v) {
       return v+1;
     }
```

Consider the function `f`. `f` is called from `g`, which in turn could be called from three different locations in `main`. Therefore, the call strings for `f` are: (c_1, c_4), (c_2, c_4), (c_3, c_4). If we want to analyze the behavior of `f`, we potentially have to analyze each calling context separately. For example, each call string will return a different number: in the context (c_1, c_4) `f` is passed 0, and in the context (c_2, c_4) `f` is passed 243.

When a program has recursive (or mutually recursive) functions, then the call stack is unbounded. The direct result is the number of call strings is potentially infinite for a real program. For example, consider the slightly changed program, where all changes are to `g`:

Listing 2: Example 1

```
     void main() {
       int t1, t2, t3;
c1   t1 = g(0);
c2   t2 = g(243);
c3   t3 = g(255);
     }

     int g(int v) {
        if(v > 1) {
c4    return g(v-1);
        else
c5    return f(v);
     }

     int f(int v) {
       return v+1;
     }
```

Now `g` has a recursive call to itself, so possible call strings to `f` may (statically) have any number of recursive steps. Just considering calls starting at c_1, we statically would derive call strings: (c_1, c_5), (c_1, c_4, c_5), (c_1, c_4, c_4, c_5), and so on.

We don't want to enumerate a potentially infinite number of calls, so we must bound call strings. There are at least two possibilities:

- Limit the call string size to a finite length k. We call the result a k-context sensitive analysis. Suppose we have chosen a limit k, and we already have a k-length call string $(c_1, \ldots, c_k)$, and we have a call c. We remove the first element c_1, and create a new call string $(c_2, \ldots, c_k, c)$. We ensure termination by stopping if we end up with a k-length call string where all call sites are the same, e.g., (c, c, c, c) would cause termination in a 4-length call string. Note that a context-insensitive analysis is simply the case where $k = 0$.

Suppose there are $M - 1$ call sites. Then one representation for this approach is to assign an a unique number between 1 and $M - 1$ to each call site. Then a k-length call string is a k-digit number base M. This number can be encoded in decimal using the normal method. The result is a number between zero and M^k, where zero denotes the empty call string ϵ.

- Instead of choosing a fixed value of k, we consider all *acyclic* call strings. For call strings with recursion we collapse all recursive cycles. For example, in the recursive example we create the call string (c_1, c_4^*, c_5), where c_4^* denotes zero or more recursive calls.

2.1 Call String Forests

Suppose function `foo` has call string $(c_1, c_2, ..., c_n)$, where c_n is the function calling `foo`. For clarity, we call c_n the callee, and `foo` the caller.

We say a parameterized name is the name of a function parameterized by each caller of the function. For example, `g` in Listing 2 has the parameterized names `g:c1`, `g:c2`, `g:c3`, and `g:c4`.

We define a call string tree as:

- N: the set of parameterized names, which are nodes in the tree.
- $r \in N$: The root of the tree. By default, we let this be `main` when known, else the start address of the executable.
- $L \in \{\mathtt{T}, \mathtt{R}, \mathtt{E}\}$: a set of *leaf* node labels where:
  - `T(x)` is a leaf node label that means parameterized name `x` is a "terminal" in the call chain, i.e., `x` makes no further calls.
  - `R(x,y)` is a leaf node label that means "recursive", and stands for parameterized name `x` making a recursive (or mutually recursive) call to `y`.
  - `E(x)` is a leaf node label that means "external", meaning `x` is a call to an external library. For example, a leaf node `strcpy:cx` is labeled `E(strcpy:cx)`.

A call string tree can be extended to a *call string forest* where we have a set of rooted trees (instead of just a single tree).

Suppose in Listing 2 we let the root be `main`. Then the call tree is given in Figure 1.

As a second example, suppose we had the following function:

```
void main(int argc, char *argv[])
{
  char buf[0xf];
c1  strcpy(buf, argv[1]);
}
```

In this example we would label the `strcpy` node with `E(strcpy:c1)` since `strcpy` is an external function.

Note a call tree is a graphical depiction of the evolution of calls. It allows us to easily find terminal nodes (which should be easier to prove are safe), as well as understand the calling context for potentially unsafe functions (which we can tackle one by one).

3 Your Assignment

You can work in pairs. However, please do not try to "divide in conquer" so that one person learns how to set up the environment, and the other has no idea. Please spend time working side-by-side doing "pair programming".

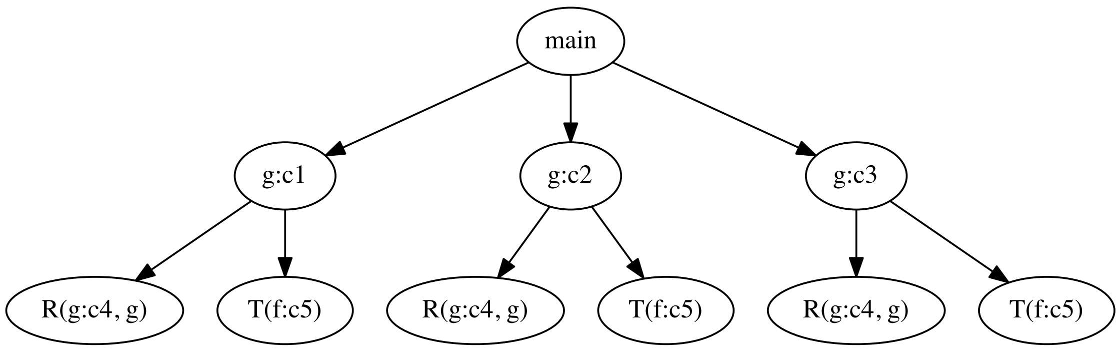

Figure 1: The call string graph for Example 1

3.1 Test Driven Development

Your assignment is to produce a plugin for computing the call strings of a program. We are going to follow the `Test Driven Development` (TDD) process when creating this plugin. A TDD process consists of three steps: Red, Green, Refactor.

One complication is writing tests within the BAP plugin architecture. We'll adapt TDD to cover end-to-end functional tests.

1. Red Step: The first step is called the Red step, and it occurs before you write any code. You start off by defining a *functional test case* of the desired behavior. A functional test case is sometimes called an *acceptance test*, or an *end-to-end test.* A functional test consists of calling your plugin with a small example, and asserting the correct output value. Use OUnit2 for the test.

You check in your test case *before* you write any code. This may seem strange at first blush; why would you check in a test case with no code to test? The reason is you are really specifying an acceptance criteria: when you have implemented the function at least correct enough to compute one value right. (Some may state that creating a test case first lowers productivity. Writing new lines of code should not be equated with productivity: writing *working* lines of code seems much closer, which is what we are doing here.)

For this step you should check in your own binary with Makefile that exhibits mutual recursion, self recursion, terminal nodes, and external library calls. Then, write out a document that describes (via a set of lists) the expected acyclic call string and k=5 call string. Please include unreachable functions as well (and make sure they are not optimized out during compilation).

2. Green Step: After you've checked in your example, you enter the Green step, where you write your BAP plugin pass. The goal is to write code that passes your test case.

Here we do not have an automated testing infrastructure a priori, which is against TDD philosophy. We'll make do as follows. After you decide the output format, write a program called `functional_callstrings_test` that can check your manually created output from step 1 against the real output from the plugin. The program should output success or failure. The program can be in the language of your choice: python, ocaml, whatever.

3. Refactor Step: In this step, you refactor to make better code. This is a step often forgotten, but very important.

To refactor properly, one partner explains the code verbally to the other partner. When issues or questions come up during the discussion, note them down on a list.

For example, perhaps you wrote a function that was very specialized and didn't handle recursion very well, and suppose you also had a warning during compilation for a missing `match` statement. Then you would have:

- Handle recursive functions.
- Fix missing match case.

Everything on the list become new things to refactor. After you refactor each one, rerun `functional_callstring_test` to make sure it still passes.

Everything on your list becomes a new test case to write. You go back to step 1: you write a recursive test case for the red step, implement it in the green step, and then refactor the code.

You should follow standard programming and ocaml best practices. For example, functions shouldn't be very big usually. If they are, refactor. You shouldn't repeat yourself in code. If you do, refactor. You should use `ocp-indent` for indentation. If you didn't, install it and refactor. Your code should be, in principle, mergable with bap on git (this includes meeting the Ivan standard).

During refactoring, you also create the documentation to your plugin.

Warning: You may be tempted to not write documentation that explains what your plugin is doing and call it done. For example:

```
module Interval =  struct
  type t = | Interval of int * int
           | Empty

  (* creates an interval *)
  let create low high =
    if high < low then Empty else Interval (low,high)
end;;
```

While the above case is trivial, it exemplifies comments that just explain what the code does. This is not right! Include in your documentation how you expect the function/interface/module to be used. Give example calls. Think creatively how you can explain to the user *how to use your function*, not just what it does. For example, in the above the documentation may say something about expecting the low to be less than high, and if not, returned Empty instead of throwing an exception, which may not be what the caller expected.

3.2 The Actual Plugin

Your plugin should implement the following behaviors:

- Given a program and an integer k, return a table m where m maps from a function to a k-sensitive call string.
- Given a program, return a table m where m maps from a function to the acyclic call string.
- Given a program and a root r, generate a call string tree.
- Given a call string table and a root r, generate a call string tree.

When I use the term "map" above, I'm talking functionality, not necessarily ocaml "map". For example, it may be based on BAP tables, or something else. You should not have to install any additional dependencies not already in BAP. For example, an experienced programmer would potentially think about data structures that don't need to keep callstrings for every point in memory if they are not needed. I make the following requirements, though:

- The table should implement at least fold and lookup. You don't need to implement everything in the "map" interface, and the use of functors in this assignment is discouraged.
- The table should be able to be stored to disk, and read back in.

- You should be able to pretty-print the table to stdout for human consumption. Suppose a callsite c_i is in function f at address a. The printout of a call site internally numbered c_i should be $f : a$, i.e., output the surrounding function name and the address of the call. We heavily encourage dot.

3.3 Turn in

You should turn in your git repo, which should contain:

- Well documented source.
- Your test cases with `functional_callstrings_tests`
- Your Plugin.
- A report on how the plugin works on the TA2 test suit. Include the amount of time it takes to run your plugin on each of the TA2 test items as a separate file, and in the report include min, max, and average time. Also note any behavior such as non-termination, running out of memory, etc. You can experiment with different k (and hopefully the acyclic version will terminate).

3.4 Some Relevant Tests

Here are some relevant small examples to test. Please compile and try them out (along with synthetic examples you come up with yourself).

Listing 3: Simple strcpy

```
#include <string.h>


/* This function calls strcpy on user input. It is a leaf.*/
int main(int argc, char *argv[])
{
  char buf[0xf];
  strcpy(buf, argv[1]);
}
```

Listing 4: Buffer Overflow - no strcpy

```
#include <stdio.h>

int foo(char *ptr){
  char buf[0xf];
  int i;

  for(i = 0; ptr[i] != 0; i++)
     buf[i] = ptr[i];
  return i;
}

int main(int argc, char *argv[])
{

  return foo(argv[1]);
}
```

Listing 5: Recursion recursed

```c
#include <stdio.h>
#include <string.h>

int is_odd(unsigned int n);
int is_even(unsigned int n);

int is_even(unsigned int n) {
    if (n == 0)
      return 1;
    else
      return is_odd(n - 1);
}

int is_odd(unsigned int n) {
    if (n == 0)
      return 0;
    else
      return is_even(n - 1);
}

void foo(char *ptr)
{
  int even;
  even = 0;
  even = is_even(strlen(ptr));
  printf("strlen(ptr) is even: %s\n", (even == 0) ? "False" : "True");
}

int main(int argc, char *argv[])
{
  char *ptr = argv[1];
  foo(ptr);
}
```