CS615 – Homework Assignment 5

September 20, 2006

The previous homework assignment was largely theoretical. This homework assignment is largely practical.

In class we have covered operational semantics (large- and small-step), denotational semantics, axiomatic semantics (including verification condition generation) and some abstract interpretation. You are now qualified to pull ideas from many of those techniques together and create a non-trivial program analysis.

This analysis will target off-the-shelf C programs. We will use the CIL library to intercept the standard `Makefile` build process of a program and present you with an OCaml intermediate representation of the program. We will use John Kodumal's implementation of Manuvir Das' One-Level Flow alias analysis to aid in reasoning about pointers. We will use the Simplify automated theorem prover of Detlefs, Nelson and Saxe to reason about infeasible paths or otherwise decide questions of logic.

Our analysis will print out a warning if it believes that the argument to a system call must be or may be 0 (zero). This is undecidable in general (by direct reduction with the halting problem). This analysis could be used to find bugs on its own: passing 0 to `free`, `memcpy`, `bzero`, `strcmp`, or the first argument to `printf` is a fatal bug. This analysis will be similar in flavor to a simplified version PREfix or Metal. It is also more-or-less isomorphic to security analyses that check to make sure that you have dropped privileges before calling `exec` (e.g., make sure that along every path that includes `exec` the most recent call to `setuid` had a non-zero argument).

I have provided an introductory analysis. It performs flow-sensitive, path-sensitive, context-insensitive, intraprocedural symbolic execution and uses McCarthy's Select and Update axioms as well as an alias analysis to reason about memory. Unlike previous assignments, for this assignment you *may change anything* in the main file you like.

Initial steps:

1. Familiarize yourself with the source code in `nf.ml` ("null-finder") and the C AST in `cil/src/cil.mli`. I have placed the formal papers on CIL ("make it easy to write C analyses by boiling C down to a small set

of expressions"), PREfix ("pick N paths, do forward symbolic execution to find bugs"), GOLF ("one-level flow alias analysis"), and Simplify ("automated theorem proving") on the homework page if you desire more background.

2. Compile `nf` and run it on `risky/risky.i` (the post-processed form of `risky/risky.c`). It should miss all of the potential false positives in `main()` and find the real bug in `method_with_actual_error()`. The `README.txt` file contains build instructions.

3. Use CIL to intercept `nullhttpd-0.5.1`'s build process and merge all of its source files into the single stand-alone file `http_comb.c`. (Optionally, since `nullhttpd` is bundled with the homework code pack you can skip this step and just use the pre-generated `http_comb.c`.) Run `nf` on that and note that every bug report is a false positive.

Exercise 1: This exercise is open-ended. You must do something to convince me that you have an integrated understanding of the theory and practice of using PL research techniques to analyze programs. More concretely, you must modify `nf.ml` so that it is "better" in a way of your choosing. As a rough estimate, I would expect a `diff` of your modified source to indicate at least 200 changed lines. Then you must write up a formal three- or four-paragraph explanation of what you did and why it was worthwhile. Your explanation should motivate your changes and explain why the problem you tackled is important.

Any of the following could suffice:

- Modify `nf` so that it handles loops in an intelligent manner. For example, you might use a dataflow-style join – if it is to possible reach the loop head knowing $x = 0 \wedge y = 55$ and it is also possible to reach the loop head knowing $x = 5 \wedge y = 55$, you should process the loop in a state where $y = 55$ (or, better yet, $x \geq 0 \wedge y = 55$).

- Modify `nf` so that it handles the heap more precisely. For example, you might introduce an explicit handling of `malloc` (which either returns 0 or a new non-zero address that is distinct from all previous addresses) and `free`.

- Modify `nf` so that it actually uses the computed alias information. Currently the alias information is not used when `decide` is called to evaluate expressions and possibly invoke the theorem prover. This would be a relatively short change, so you should also do something else and/or provide compelling examples to show that the alias analysis really helps.

- Modify `nf` so that its performance and scalability are non-trivially improved. This typically requires more "engineering" than "theory", but getting an analysis to run on millions of lines of code (e.g., the Linux kernel, SQL Server) is very difficult. Your modified version should run significantly faster on `nullhttpd` and other benchmarks of your choosing.

- Modify `nf` to remove (or rank, or otherwise deal with) false positives. You might get started by inspecting all of the false positives reports that `nf` generates on `nullhttpd` and finding principled, reasoned ways to remove them.

- Modify `nf` to accept different safety policies (e.g., a global finite state machine that has state transitions on certain function calls). After inspecting the code you may well have concluded that it is pretty much equivalently difficult to check most safety properties but that some properties are much more likely to yield real bugs and avoid false positives. Notably, the "non-null system call argument" policy is actually a very poor choice. Feel free to use domain-specific knowledge and concentrate your specification on a particular area. A stellar job would involve finding a bug in a real program (even an old version where the bug is known to be there).

- Modify `nf` to be context-sensitive. You might compute the call graph and analyze the functions in reverse dependency order. You might do a full-blown CFL reachability analysis. Or you might just start in `main` and take very long paths through the entire program (note that if you start in `main` it can actually be very difficult to have any sort of decent statement or path coverage).

- Modify `nf` so that it more directly implements a small-valued abstract interpretation like the one that we covered in class (e.g., positive, negative, zero). Then you must show convincingly that this actually achieves something (e.g., faster performance, fewer false positives, more bugs found). Many researchers have gotten quite a bit of mileage out of nominally-simple abstract domains (e.g., locked and unlocked, null and non-null, socket – bind – listen – accept – read, int32 wraparound).

- Modify `nf` so that it computes a "semantic checksum" of methods based on the resulting symbolic states (either their "average" or their "difference" or whatever ends up being useful). For example, see if you can find some way of summarizing procedures so that two hand-written "viruses" appear similar but a "virus" does not appear similar to the example "risky.i" code.

You should submit your modified `lastname-nf.ml` file, an ASCII text file containing your multi-paragraph report, and any other compelling examples or figures that you feel back up your case. Recall that you should demonstrate that you did something useful with respect to this homework's goals of using program analysis techniques either (1) in your research or (2) to understand programs or (3) to find bugs or (4) to verify properties of programs or (5) to make related tools more usable.

Exercise 2: Submit two post-processed C files (e.g., `.i` files or `_comb.c` files) named `lastname-zero.c` and `lastname-one.c`. The first file should have no errors *and* your analysis should not report any false positives on it. The second file should have exactly one error *and* your analysis should report exactly that error. We will have the standard extra-credit shootout based on these. Your program must not loop forever or crash on any input.

If you did something non-standard (e.g., the semantic checksum variant), instead submit two or three files that demonstrate that your modified program behaves as intended.

Exercise 3: Compose a brief (one- or two-paragraph) email to one of the authors of the infrastructure or papers you used in this homework and send the email *to me (not to that person).* In addition, indicate whether you are willing to use your name or whether you would like to be portrayed as an anonymous student in my class. I will check off the fact that you wrote something and potentially forward it along. You can comment on any aspect of your experience with their work — your comments need not be positive. For example, you might ask Saxe why Simplify doesn't handle multiplication, complain to Kodumal or Das that OLF isn't precise enough for C programs, or tell Necula or McPeak that you find CIL's memory lvalue semantics unintuitive. You might even write to Pincus and ask him how he managed to get some part of PREfix to work given all the difficulties you observed when wrestling with C. If you do offer criticism, strive to make it constructive by commenting on what you would have liked to have seen instead or how you might like to see things improved if the time were available. If you absolutely cannot think of anything to say, thank them for making their tools available and let them know that you used them with success. Even minor comments about documentation or a fresh-eyed perspective on usability can be helpful.

The purpose of this non-standard exercise is two-fold.

- First, I have observed multiple instances in this class of a student being unwilling to contact the author of some publicly-available project. While I realize that you don't want to be known as a whiny grad student who didn't bother to read the manual, it's also not worth wasting your time to try to decipher a research prototype when the author is only an email away. I think it would legitimately be good practice for

many of you to correspond with a random researcher. You may not get a response, but the sky won't fall. (In addition, I know the people involved in all of this software and they are all quite friendly.)

- Second, internships are not the only way to build up contacts and networks. It is entirely reasonable to grow a friendship or collaboration with someone over time, starting with a lowly email about research, moving on to chatting a conferences, and eventually working together on new research. You're rarely certain of exactly where you will end up or what you will be working on, so it behooves you to know as many people out there as possible.