UNIVERSITY OF CALIFORNIA, SAN DIEGO

Project Midpoint Report– Fall 2015

Predefined Project - Part 1 and Static Race Detection using RELAY

CSE 231: Advanced Compilers

Submitted by:

Fnu Anand A53081794

Siddhant Arya A53079389

Suneeth Subbiah A53055580

1 Overview

Based on the concepts mentioned in the RELAY paper (Voung, Jhala and Lerner FSE 2007) in [1], we will be applying data/control flow analysis on the Linux kernel and implement the techniques and heuristics mentioned in the paper to detect the possibility of race conditions.

The paper accomplishes this by analyzing functions in isolation to compute summaries using relative lockets that capture the behavior of the function independent of the calling context, and then composing the summaries to determine whether races exist. The analysis reveals a large number of possible race conditions which are then categorised and filtered to eliminate false positives and locate definite race conditions in the kernel.

As part of the future work, we wish to use the LLVM Toolchain to implement static analysis for race detection as described in the paper. As of the Midpoint Duedate i.e. Nov 3 2015, we have implemented part 1 of the predefined project to get more comfortable with the functionality offered by LLVM as discussed with Prof Sorin Lerner. It has been done so as to gain hands-on experience of using the LLVM Compiler Infrastructure and to write dynamic and static program analyses analyses

In the following sections of the report, we will be highlighting the salient features of our implementation of the Pre-Defined Implementation Project. The project contains three sections, namely Static Instruction count, Dynamic Instruction Count and Branch Bias Calculation.

2 Static instruction count

In this section, we were asked to write a pass that counts the number of static instructions in a program.

We extended the ModulePass and iterated successively over modules, functions, basic blocks and instructions. For each instruction type, we insert and update the count in a STL map and print the statistics at the end.

3 Dynamic Instruction Count

To get count for instructions dynamically, we need to update our counter in our instrumentation module corresponding to each executed instruction. Since a basic block gurantees sequential execution, we get a instruction count in each basic block at compile time and insert a call to our update_map function at the end of each basic block.

The arguments for each basic block are instruction name and count within the block. We managed to avoid name mangling issues by declaring our helper functions as extern.

4 Branch Bias

Initially we wanted to implement this pass by evaluating the branch condition before each br statement and then calling our helper module with the enclosing function's name and the value of the condition. However, we cannot know the value of the condition at compile

time - a fact which sounds trivial in retrospect but we lost a few hours on it. Ideally we would like to insert a statement which evaluates the condition, calls our module counter at run time and update the condition in the branch statement with this value to avoid reevaluation of the expression. We have gone with a different approach. Essentially we insert a call to our module at the beginning of the true block and a call before all branch statements. Their ratio should give us the branch bias. However we are not sure if the basic block for the true branch is called exclusively by the branch statment or can be called from the other locations as well - in which case our bias would be inaccurate.

5 Going further

This assignment gave us a better idea about the possibilities of the LLVM framework and we will next see how to utilize it to implement static rac

References

[1] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. *RELAY: static race detection on millions of lines of code.* ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07). ACM, New York, NY, USA, 205-214. DOI=http://dx.doi.org/10.1145/1287624.1287654