Code Reduction

In this phase of the mangrove project, given a false positive bug report from a static code analysis tool (SCAT) and the source code that the bug report is generated for, our goal is to reduce the source code to a small snippet which still causes the SCAT to generate the same false positive bug report. By doing so, our aim is to 1) identify the code patterns and coding behaviors leading such false positive bug reports, and 2) generate a database of such small snippets to be used in the second phase of the project which involve machine learning activities.

1 Initial research and experiments

Preliminary studies for code reduction are done by Matias Marenchino for his master thesis [8] under supervision of Professor Adam Porter. Matias developed a tool named *CodeReducer* which implements two important algorithms for code reduction; delta debugging [13] and iterative reduction debugging. *CodeReducer* integrates these two algorithms with a balanced delimiter based hierarchical code flattening technique [1]. Here these two algorithms will be referred to as DDT (Delta Debugging with Topformflat) and IRDT (Iterative Reduction Debugging with Topformflat).

To evaluate the performance of DDT and IRDT, Matias conducted a set of experiments on c, c++, and java languages. In these experiments *cppcheck* [9] for c/c++ and *FindBugs*[2] [1, 2] for java are used as the subject SCATs. For c and c++ languages, a test suite developed by NIST to evaluate

[1] topformflat is the tool which implements the balanced delimiter based code flattening idea, http://manpages.ubuntu.com/manpages/xenial/man1/topformflat.1.html

[2] *FindBugs* is the most used static analyzer for java

SCATs -namely Juliet [3]- is taken as benchmark which *cppcheck* will run for. Likewise for java, five open source projects are taken as benchmark which *FindBugs* will run for.

Results of his studies suggested that both DDT and IRDT algorithms are promising in reducing the code to small snippet; on average both algorithms achieved 80% and 85% reductions in line of codes (LoC) respectively for c/c++ and java programs.

While these reduction percentages are promising, there was one major problem; for some of the snippets the bug report generated by the subject SCAT was not a false positive but indeed a true positive -real bug- whereas the bug report generated for the original code was a false positive. Another downside with the *CodeReducer* was that, the level parameter to the code flattener was set manually after studying the nature of the flaw in concern.

2 Preserving the falseness of a bug report

Code reduction algorithms like delta debugging [13] uses an interestingness test script which is written by the user. In this interestingness script, user defines some criterion to guide the delta debugging algorithm. Basically, after performing a modification in the code (or in whatever data the delta debugging algorithm is reducing), algorithm runs the interestingness test script to check weather the modification was good or not.

This interestingness script could potentially be one place where we may define a criteria that avoids false positive to true positive (fp→tp) conversion problem. However, after looking many false positive bug report instances, we found out that in many cases finding the reasons of being false positive is too complicated and requires serious manual effort. Hence, automating this examination to become a procedure in delta debugging algorithm is not possible in general.

The alternative approach to take would be, in the reduction algorithm, to perform more careful and intelligent modifications to help avoiding fp→tp conversion. Towards this direction, we reviewed the literature about variations of delta debugging algorithm. One popular variation of delta debugging is the hierarchical delta debugging (HDD) algorithm which takes a tree struc-

tured input data and performs the basic binary search technique on the tree by removing the parts of tree [10]. This algorithm is indeed very similar to what Matias did by integrating delta debugging with topformflat tool [8]. The differences is; HDD cannot be used on source code directly whereas DDT does. Instead, Misherghi and Su run HDD the algorithm on abstract syntax trees (AST) [10].

Unfortunately, the code provided by Misherghi and Su as part of their work [10] is obsolete as of Feb 2016[3]. Therefore, we cannot test HDD algorithm right away, we indeed need to implement the idea which is not trivial. At this point, we did not take this path because we have no reason to believe that HDD is the solution for fp→tp conversion problem. Instead, we decided to look into another more recently introduced code reduction work; *creduce* by Regehr at al [12].

3 Creduce

Creduce is a code reduction tool originally developed to find small programs that cause c/c++ compilers (e.g. gcc) crash [12]. *creduce* does not directly implement delta debugging (nor HDD) algorithm. Instead *creduce* has 150 transformations each of which takes the source code and performs an alteration that, in most cases, reduces resulting source code. Most of these transformations would effect the semantics of the code whereas only a few would do pure syntactic changes; changing variable, parameter, function names, etc.

With creduce, Regehr at al [12] aims at finding small test cases which causes compiler to crash whereas we aim at finding small code snippets which causes SCAT to produce a false positive bug report. Hence, clearly *creduce* can be adopted for our problem. However, as we mentioned earlier, our problem is more challenging because defining criterion in interestingness script to determine falseness of a bug report is not possible whereas failure of a compiler would result in an obvious output that can be easily detect by a simple criteria in the interestingness test script. Nevertheless, we continue exploring

[3] The original HDD code proposed can be found on website of Professor Zhendong Su

creduce as it provides a variety of transformations.

One challenge is that there is no official documentation explaining all of these transformations. By looking at the source code, we found out that 65 of transformations are implemented using clang ASTs. Running *clang_delta* command with *–transformations* flag under the clang_delta directory of *creduce* project will show brief explanations for these 65 clang AST based transformations. Based on these explanations and some experimentation, we separated these transformations into three sets; safe transformations, ineffective transformations, and unsafe transformations. Safe transformations usually perform changes which does not lead to fp→tp conversion problem. Ineffective transformations usually do not perform useful changes and/or they do not necessarily reduce the code. Unsafe transformations perform changes which may lead to fp→tp conversion problem. These three sets will be improved with the none clang based transformations as we experiment with them.

For some of the other transformations naming is helpful. For example, the transformation *remove-unused-function* removes the functions that are not called in any place in that file (we verified the safety of this transformation both by looking at source code and experimentation). Finally, for remaining transformations which do not have any documentation for, we did a series of experiments to find out

Furthermore, there are 17 clex [5] based transformations. A transformation *pass_clex name* removes a chunk of code which matches a specific regular expression defined in the transformation *name*. This *name* is not self-descriptive. One needs to look into the source code of transformation to understand which expressions in input code are being targeted or experiment with them, as we also did so (see Section 3.1).

3.1 Creduce Experiments

As *creduce* only works for c and c++ languages, all experiments explained in this section are performed on c and c++ code. As the subject SCAT, instead of *cppcheck*, we used the clang static analyzer [7] version 3.8 as it employs more modern analysis types and therefore more precise compared

to *cppcheck*.

Reducing Juliet test cases with *creduce* safe transformations. Running the clang static analyzer on entire Juliet test suite code resulted in more than 5000 bug reports. Then for each bug report, we took the source code file the report generated for and reduced it with *creduce* only safe transformations turned on. Here, we like to mention that *creduce* becomes very aggressive with all transformations turn on (the default case). Running *creduce* with the default configuration always leads to fp→tp conversion.

The final step of this experiment, checking the snippets for fp→tp conversion, is manual. Among about 5000 snippets reduced for 5000 bug reports, we randomly selected 170 snippets with their corresponding original source files. We find out that 34 of the bug reports generated for the original source code are true positive -real bug- and the remaining 136 are false positive. After a careful manual examination we verified that there is no fp→tp conversion happened for these 136 bug reports. Furthermore, *creduce* with safe transformations provided about 80% reduction on average. Hence, in this experiment we saw that running *creduce* even with only the safe transformations turned on is very effective in reducing the code while preserving the falseness of the bug report.

Reducing some real programs with *creduce* safe transformations. After the Juliet experiments, we now evaluate the same safe transformations for some real world programs. First, we run the clang static analyzer on all the programs in coreutils [6], all the programs in phylip [11], bzip2, and oggenc[4]. These analysis runs resulted in a total of 192 bug reports; 47 for coreutils, 100 for phylip, 7 for bzip2, and 38 for oggenc. Now, we do not reduce all sources files these bug reports generated for as we did in the previous experiment. Instead, we manually search for some false positive cases and reduced only those source files with *creduce* safe transformations.

Table 1 presents the results of this experiment. First three programs; *yes, who*, and *ls* are from coreutils; next two programs; *dnamove* and *seq* are from phylip and last two programs are standalone software. For these seven data points, although the average reduction is still good, 71.3%, but the problem

[4]For the sake of simplicity during experimentations single file versions of bzip2 and oggenc are taken from; http://people.csail.mit.edu/smcc/projects/single-file-programs

program	original LoC	reduced LoC	reduction percentage	fp→tp
yes	96	67	30	✓
who	617	415	33	✓
ls	3560	419	88	✗
dnamove	1939	84	96	✗
seq	3567	44	99	✓
bzip2	5094	2230	56	✓
oggenc	48322	1342	97	✓

Table 1: Creduce safe transformations

is, in two out of seven cases there is fp→tp conversion problem. Looking into these two cases, *ls* and *dnamove*, we observe that there are global variables initialized in some other initialization functions and then used in the function which owns the line mentioned in the bug report. However, there is no direct call-callee relation in between the initialization function and the function we are focusing on. Therefore, *creduce* removes the initialization function and the bug report becomes a true positive.

```
coreutils/src/ls.c:4579:28: warning: Division by zero
      size_t col = filesno % cols;
```

Figure 1: A false positive warning generated for the program *ls*

For example, the bug report in the Figure 1 is a false positive as the variable *cols* gets initialized to a none-zero value in an initialization function and then used at line 4579. Since there is no direct call-callee relation in between the initialization function and the function that owns the warning line, *creduce* removes that initialization function and therefore the problem becomes real.

Reducing with safe and clex-based transformation. Among the transformation that we could not categorize looking documentations nor source code, there are 17 clex-based ones. Each of these transformations removes a chunk of code which matches a specific regular expression defined

in the transformation. Although one can understand what specific patterns may match to these regular expressions, it is still necessary to experiment with them to see their effect in terms of preserving the falseness. Therefore in the next experiment, in addition to the safe transformations, we first turned the clex-based *rm-tok-pattern-4* transformation on, as we observed that it was among the most effective ones in reduction.

Table 2 presents the results of this experiments. The situation here is got even worse as in six out of seven cases we had fp→tp conversion problem. Hence we added *rm-tok-pattern-4* into the unsafe transformations set.

Furthermore, we repeated this experiment for each of the remaining clex-based transformations by turning one of them on at a time. In each experiment we had at least four fp→tp conversion problem. Consequently, we added all of the clex-based transformation to the unsafe transformations set.

program	original LoC	reduced LoC	reduction percentage	fp→tp
yes	96	70	27	✓
who	617	24	96	✗
ls	3560	433	99	✗
dnamove	1939	16	99	✗
seq	3567	12	99.6	✗
bzip2	5094	43	99.1	✗
oggenc	48322	98	99.8	✗

Table 2: Creduce safe transformations + clex based *rm-tok-pattern-4* transformation

Reducing with remaining transformations. So far, we have covered around 80 transformations. For the remaining 70 transformations, we performed the same experiment we did for clex-based transformations and included these transformations into one of the set we formed earlier based on the result of the experiments. Some transformations which perform syntactic changes are included in the safe set as they indeed reduce the amount of actual text.

Here, we like to mention about one particular transformation; *pass_lines x*. This transformation performs the standard binary search-based delta debug-

ging modifications where x is being the level parameter to the code flattener topformflat. Technically, this is the same thing that *CodeReducer* does. As we already tested this transformation in our initial experiment and got fp→tp conversion problem, we have added *pass_lines x* into the unsafe transformations set.

3.2 Discussions

Here we like to list some important observations based on the results of *creduce* and *CodeReducer* experiments; 1) determining the falseness of a bug report requires human expertise and serious manual effort, 2) finding the smallest piece of code that is responsible for the false positive bug report cannot be automated in a generic way, 3) global variables lead to false positives bug reports from (observed with both *clang static analyzer* and *cppcheck*), and 4) as the code base grows, *creduce* with safe transformations becomes more effective in reducing the code.

The observation about the global variables leading to false positive bug reports is an interesting one. Developers often do not favor using global variables in the programs as it is a bad programming practice. We like to add this consequence of global variables to the reasons to why not to use them.

Lastly, comparing *creduce* and *CoverReducer*, one may prefer *creduce* as it provides some richness and flexibility with the around 150 transformations. Moreover as we menationed earlier, *creduce*'s one particular transformation, *pass_lines*, would do the same job that *CoverReducer* DDT algorithm does.*CoverReducer* however, has the advantages of working for java code and IRDT algorithm.

4 Program Slicing

Program slicing is a technique which finds the set of program statements -a slice- that may effect the values of variables at a certain point (backward slicing). In theory, program slicing should be an effective way of finding the minimum snippet by taking the program location pointed by bug report

as the point of interest then computing the program slice which covers all statements effecting the values at that program location.

In order to test this hypothesis, we looked into frama-c; a SCAT which also has program slicing module [4]. We observed that frama-c first preprocesses the code before doing any kind of analysis to find the slice. Therefore, because of that preprocessing, resulting slice is not similar to the original code at all -i.e. it does not have the control flow of the minimal snippet causing the false warning.

We concluded that due to lacking of a good program slicing tool for c/c++ languages, we are not able to make use of this technique for our reduction problem at this state.

5 Conclusion

In this phase of Mangrove project, we have developed a code reduction tool and to evaluate it's performance we conducted the first set of experiments on three programming languages. Based on promising results of these initial experiments, we then explored other code reduction algorithms and tools. In particular, we have looked into and experimented with *creduce* as it provides more than hundred c/c++ specific transformations. We determined a set of safe transformations which can be very effective in code reduction.

Many modern static code analysis tools, like clang static analyzer [7], perform very powerful types of analyses which help them to reveal many software errors early. While doing that, they also produce many false bug reports as a result of getting tricked by the complex structures in the code. Therefore it is reasonable to expect that determining the causes falseness of a bug report will often be very challenging as it is complex. Likewise because of the very same reason, code reduction to get just the essence of code responsible for false positive bug report is also very challenging.

In a series of experiments, we observed that although the code reduction tools we (developed and) used provide high percentage of reduction, they usually do not find the smallest possible code snippet. In other words, these tools perform the reduction work up to some point. When it is necessary to find the exact solution -i.e. the smallest code snippet-, one should to manual

trimming afterwards which is a lot easier that doing the entire reduction manually. For the purposes of our project, it is necessary to find that small snippet for identifying it, but not not necessary for the machine learning part.

Another important observation is about preserving the falseness of the bug report. Although *CodeReducer* and *creduce* with a subset of transformations turned on reduced the program they work on to a smaller version, there were times when they fail on preserving the falsenes -i.e. fp→tp conversion. We think that there is no sound reduction approach to always guarantee preserving falseness essentially because the problem is undecidable. Otherwise, the SCATs would not generate these false bug reports in first place. Therefore our approximate solution with small fp→tp conversion rates is a good start towards to next phase of mangrove project .

References

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Findbugs - find bugs in java programs. http://findbugs.sourceforge.net, Accessed on 2017-01-20.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[3] T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):0088–90, 2012.

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012. http://frama-c.com, Accessed on 2017-01-20.

[5] R. Godoy. clex: command line file manager which uses the ncurses library. https://launchpad.net/ubuntu/+source/clex, Accessed on 2017-01-20.

[6] B. V. E. B. P. E. Jim Meyering, Padraig Brady and A. Gordon. Coreutils - gnu core utilities. https://www.gnu.org/software/coreutils/coreutils.html, Accessed on 2017-01-20.

[7] Llvm and clang community. Clang static analyzer. https://clang-analyzer.llvm.org, Accessed on 2017-01-20.

[8] M. Marenchino. *Source Code Reduction to Summarize False Positives*. PhD thesis, 2015.

[9] D. Marjamaki. Cppcheck - a tool for static c/c++ code analysis. http://cppcheck.sourceforge.net, Accessed on 2017-01-20.

[10] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[11] D. Plotree and D. Plotgram. Phylip-phylogeny inference package (version 3.2). *cladistics*, 5(163):6, 1989.

[12] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *ACM SIGPLAN Notices*, volume 47, pages 335–346. ACM, 2012.

[13] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.