**The Critique of Random Testing for C and C++ Compilers with YARPGen**

This paper is talking about a refined idea of bug finding C and C++ compiler. Random testing is widely used in C compilers as well as in other testing area. But from the paper Exploiting the Saturation Effect in Automatic Random Testing of Android Applications, when using random testing method, it is difficult for the tester to know when the testing process has reached a point where no further code coverage or fault detection can be achieved. And there are still many bugs emerging even if the program had reached the saturation point based on the exploitation of the saturation effect, like the well-known compiler Csmith. So, this paper had found another point of finding bugs for the compiler – generation policies. Generation policy of C or C++ should be known by everyone grasp C and C++. But building a logic that could fix the code like changing some characters in order to let the code work based on generation policies is definitely a new and good idea.

The second section talks about Random Differential Testing and Undefined Behavior, Unspecified Behavior, and Implementation-Defined Behavior, two backgrounds. Although random differential testing has its problems, YARPGen is still using this method. Since C and C++ is an unsafe programming language, undefined behavior is the most serious problem that would be solved in the latter part.

So, in the third section, the first part is to avoid undefined behavior. Csmith uses the method: Generate code without static constraints, inserting dynamic checks as needed to guard against undefined behaviors. But the YARPGen rejects this method in the paper because it is losing expressiveness. I think that generating code in this method is simpler than interleaving analysis and code generation in order to reliably and efficiently generate code that is free of UB, which is used by YARPGen. This paper concentrates expressiveness other than complexity.

The second part is generating code. The code generation consists of creating the environments, statements and expressions and it is generated up-down, like generating a tree. The YARPGen replaces the operation bottom-up when it meets with the unsafe condition. The unsafe condition is something exceeds the INT\_MIN and INT\_MAX. The idea itself is quite clear, but choosing which operation is still unclear to me. Just like a / b when b == 0, it is a straightforward unsafe operation, but I am not sure why it is replaced a \* b, not a or other operations. By guaranteeing all subtrees to be UB-free, then using the bottom-up function, then the whole expression is UB-free. The advantage of the bottom-up checking method is that it is easier to modify subtrees while not affecting the UB status of the whole expression.

The third part is Generation Policies. In this paper, it said empirically, when using the random testing, the test cases with different operations would seem to be alike. So, the YARPGen uses some generation policies to rewrite the expression. (x | c1) ^ (x | c2) and (x & c1) ^ (x & c2) looks alike, but when it rewrites them into (x & c3) ^ c3 where c3 = c1 ^ c2 and (x & (c1^c2)), the expression looks quite different then. And these expressions also become shorter. The context of these policies consists additive, bitwise, logical, multiplicative, bitwise-shift and additive-multiplicative. When rewriting a code, the YARPGen chooses one of them above and restrict the expression into that format. The idea of maintaining a buffer of previously used constants in this method is a good idea. It makes it easier to check and use these constants in expressions in some format, although it would cause some more memory cost. Parameter shuffling is another great idea in this paper. As the problem mentioned before, random testing would make the test cases looks alike. Parameter shuffling, similar to Swarm testing, shuffles all 23 parameters into different ones in random probabilities, which could avoid this problem to some extent.

The fourth part is Automation. From the paper, we could know that the YARPGen uses 2 compliers to execute the code and do the comparison. It uses the third complier to deal with the crashed code, and put it into the C-reduce as well as the wrong code found in the comparison process. Then these codes would go to a bug classifier and finally get a bug report. Using two compilers for comparison decreases the chances of bugs. A wrong code would have a 1% not be found in an independent compiler, but the chances would drop to 0.01% for 2 compilers. Bug classifier is used to prevent frequently-occurring bugs by collecting its frequency.

The remaining part of the third section describes the remaining features of the YARPGen. To solve the problem of changing the result of an FP computation, YARPGen stores these FP variable into a file, then perform the random testing. This method partially solves this problem. but it also has more memory cost. Vectors are easy to deal with. And for the loop. YARPGen prevents the UB behavior by testing the first iteration and ensure the rest of the loop on these observed variables do not have UB. By disabling mixed variables, this kind method could be feasible as I listed some loop iterations, and it all works. There are still many limitations of YARPGen, that it cannot deal with operations and types, and I hope these limitations could be lifted in the future.

The fourth section is evaluation. From the result, we could know that YARPGen could detect bugs of already tested production-grade compilers. One factor is that YARPGen uses fuzzer to detect bugs. Fuzzer could find Partial bug fixes and some neglected bugs. But I think it is the advantage of Fuzzer itself, not the YARPGen. Another factor is that YARPGen could find the Sanitizer bugs. YARPGen is described to avoid meeting sanitizers from the third section. It is its unique advantage to find more bugs.

As for the impact of Generation Policies, this paper both consider the direct and indirect impact of GP, but the result seems that GP does not show a clear improvement. In the direct impact considering the crash bugs, in Table 4, the null hypothesis: GP is better at 95% is rejected. And for the indirect impact considering whether GP or no GP gets more optimization counters, the GP performs even worse in Table 5 in the number of LLVM counters. In Table 6, geometric mean of counters of GP is 20% more than that of no GP and for GCC, it is 40% more. The ratio is clear, but considering all these data, the advantage of GP on LLVM is not so convincing, The remaining parts are the related work and conclusion, which is unnecessary to discuss here.

To conclude, the YARPGen get a decent result on finding bugs on LLVM and GCC. In addition to the effort of fuzzer, generating the UB-free code and avoiding saturation using GP are the advantage of YARPGen. The GP changes many codes, although I still some examples in GP still looks alike, and it performs well on GCC compiler. This paper apparently shows a great idea of compiler, which is not seen in other papers.