Kenji Nicholson
Lauren Martini
UW Net ID: kenjilee, lmartini

Flow Control
An Overflow Detection Addition to the Checker Framework

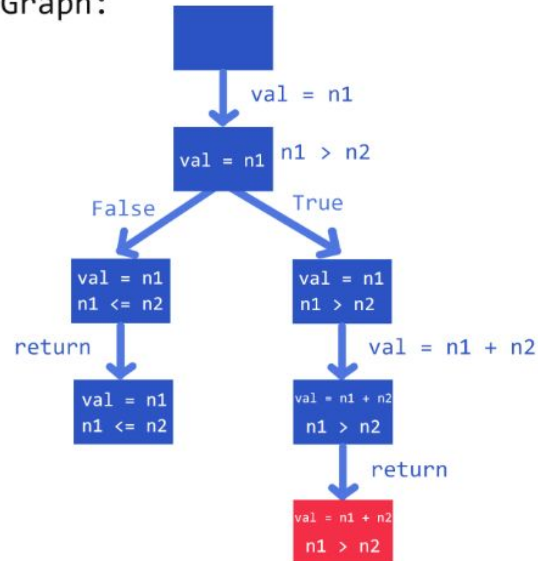
Problem: Integer overflow occurs when integer arithmetic does not agree with ideal arithmetic. For many languages, the integer computation of 2,147,483,647 + 1 results in the negative number -2,147,483,648. This results in the unsoundness of many programs that rely on integer computation. For example, the Index Checker that is currently used to detect out of bounds errors incorrectly assumes that 2,147,483,647 + 1 results in a positive number. We plan on creating a tool that can detect whenever an integer might overflow to help make programs like Index Checker sound when it comes to overflow errors.

Motivation: Creating a tool that detects where integer overflow may occur solves this problem by allowing programs reliant on integer computation to extend their guarantees to cases where integer overflow occurs. For example, the Index Checker would be able to detect when adding to an integer marked with @Positive results in a resultant integer that is now negative, no longer matching the @Positive tag, which would cause an error to be thrown. This error could be modified to be more informative, which would also solve the problem of uninformative index-out-of-bounds error messages that result from integer overflow. This is an important problem to solve, because overflow errors are extremely common, so this is a large gap in the capabilities of the index checker that needs to be filled.

Approach: This tool would likely be designed and developed in one of two ways: 1) as an improved version of the existing index checker included in the Checker framework that includes checks for overflow, or 2) as a separate overflow checker that you could run in addition to the index checker that would catch possible overflow errors separately. In either case, this will be a static analysis tool written in Java, based on the source code provided from the Checker Framework. To do this, our tool would construct a graph of all possible program states (with the vertices keeping track of the program point and the current state and the edges representing a transition between states.) Then, in order to find illegal states in the graph, this tool will either introduce new tags, such as a tag that indicates whether it is valid for the sign of a given integer to change, or modify the usage and checking of existing tags, such as @Positive, in order to correct possible overflow errors. For instance, the @Positive tag could be modified so that the checker checks whether the sign of the integer changes due to overflow. Shown below is a diagram of a graph placed next to corresponding sample code that exemplifies when an illegal state has been reached.

Code:

```
public int sumIfFirstGreater(int n1, int n2, int val) {
    val = n1;
    if(n1 > n2) {
        val = n1 + n2;
    }
    return val;
}
```



Example of a graph of program states with one illegal state (colored in red).

Alternative Approaches: Current alternative approaches are either unsound or have a high rate of false positives.

- One such alternative approach would be IOC: An Integer Overflow Checker for C/C++. This checker is dynamic and modifies the Clang checker, and claims to have no false-positives. However, this method is unsound (due to being dynamic) and does not find all undefined overflow behaviors. Our approach resolves this because its a static checker. However, like other static checkers, ours will fall victim to false positives.
- Another alternative, IntScope uses symbolic execution during static analysis to detect overflow errors. Symbolic execution means replacing data from the code with symbols that each have a set of expressions. After this, the symbols are translated into a control flow graph like above. However, IntScope runs into several false-positives when faced with things like: missing input constraints, lacking global information, and imprecise symbolic execution. Our approach will aim to reduce these false-positives, if possible. An additional benefit to our approach is that it will be an extension of the Checker Framework, which means it will be able to check several other issues on top of integer overflow.

Challenges/Risks: One major challenge will be avoiding false positives. Some programs that have previously attempted to develop a type system for preventing overflow errors experienced high false positive rates. A major risk is that if we decide to develop a modified version of the existing index checker, we will need to preserve the function of the existing index checker. Any tag modifications that we make should not interfere with the original purpose of the modified tags. To mitigate this, we will most likely create new tags rather than modifying existing ones, and follow the checker's method of generating a control flow graph.

References:

[1] Li, Peng, et al. *IOC: An Integer Overflow Checker for C/C++*. embed.cs.utah.edu/ioc/.

[2] Wang, Tielei, et al. *IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution*. web.cse.ohio-state.edu/~lin.3021/file/IntScope_NDSS09.pdf.

This project took approximately 10 hours to complete.