17-355/665/819
Interval Analysis Project Report

Jake Zych (Andrew ID `jzych`)

1 Summary

An interval data flow analysis was implemented using the Soot framework to detect out of bounds array accesses in Java code. The goal of this project was to apply the analysis on real world software to detect bugs in order to evaluate the efficacy of such an analysis compared to testing or other static analysis tools. The implementation makes use of the widening operator to allow for termination when analyzing code with loops. Furthermore, it also considers the set of constants in the program to more intelligently apply the widening operator resulting in more precise results. Testing on real world software proved to be more difficult than anticipated, highlighting some of the drawbacks of data flow analyses using the Soot framework. However, the implementation still produced precise and valid results when tested on real world software demonstrating the value of using data flow analysis as a bug catching tool.

2 Background

2.1 Mathematical Definitions

We can define the lattice for Interval analysis as follows:

$$L = \mathbb{Z}_\infty \times \mathbb{Z}_\infty$$

where $\mathbb{Z}_\infty$ is the set of all integers including $-\infty$ and ∞.

In order to define $\top$, we can use the interval $[-\infty, \infty]$ which essentially means the variable can take on any possible value. We define $\bot$ in a similar way, but we flip the range to be $[\infty, -\infty]$. This acts as a "dummy" interval because every value for the lower end of this interval will be less than ∞ and every value for the upper end of this interval will be greater than $-\infty$.

We can define the join operation as follows:

$$[l_1, h_1] \sqcup [l_2, h_2] = [min_\infty(l_1, l_2), max_\infty(h_1, h_2)]$$

Here, the min and max operations are augmented to handle ∞ and $-\infty$. In the implementation, these are represented as the maximum and minimum values a 32 bit integer can hold.

Moreover, an example of a flow function for interval analysis is as follows:

$$f[x = y + z](\sigma) = \sigma[x \to [l, h]]$$

where l is the sum of y and z's lower bound values and h is the sum of y and z's upper bound values. As with the join operation, we must augment all arithmetic operations to be able to handle ∞ and $-\infty$. Furthermore, in the implementation, we must handle things like multiplying or dividing by -1 flipping the lower and upper bound in the interval.

If either y or z is $\perp$, the flow function should keep σ the same.

2.2 Widening Operator

Recall that for an analysis to be guaranteed to terminate, it must have a finite height lattice and its flow functions must be monotonic. With the lattice we have defined above, its possible to have an infinite ascending chain. For example, consider a while loop that increments a value until it reaches a certain threshold.

```
x = 0;
while (x != y) {
    x += 1;
}
```

A while statement is implemented internally using if statements and goto statements. When analyzing a program intraprocedurally, we re-process any statements whenever a new input value is detected. We can see from this example that the chain of lattice values for x would be $[0, 0], [0, 1], [0, 2], ...$ which will ascend infinitely, preventing termination of the analysis.

In order to shorten the length of this chain to be finite, we can define a widening operator to be used before processing the head of a loop:

$$W(\perp, l_{current}) = l_{current}$$

$$W(l_{previous}, l_{current}) = [min_W(l_1, l_2), max_W(h_1, h_2)]$$

where $min_W(l_1, l_2) = l_1$ if $l_1 \leq l_2$ otherwise $-\infty$ and $max_W(h_1, h_2) = h_1$ if $h_1 \geq h_2$ otherwise ∞. Despite it being safe to apply the widening operator before processing each statement in a program, doing so would result in a loss of termination with little to no actual benefit. It is important to apply the widening operator whenever a statement is processed that is likely to be processed again with a different input i.e. a loop guard.

2.3 Constant Truncation

As mentioned above, a limitation of the widening operator is that it sacrifices precision for the guarantee of termination. We can, however, improve the widening operator to give more precise results by inferring information about what the possible values a variable can take are. Rather than setting the lower bound to $-\infty$ when $l_2 > l_1$, we can instead set it to the largest constant less than l_2. Similarly for the upper bound, we can set it to be the smallest value greater than h_2. Below is the newly defined widening operator.

$$W(l_{previous}, l_{current}) = [min_K(l_1, l_2), max_K(h_1, h_2)]$$

where $min_K(l_1, l_2) = l_1$ if $l_1 \leq l_2$ otherwise $max(\{k | k \leq l_2\})$ and $max_K(h_1, h_2) = h_1$ if $h_1 \geq h_2$ otherwise $min(\{k | k \geq h_2\})$.

This gives us the advantage of potentially terminating the infinite chain at a constant k in the program which may be the difference in an array access being considered safe or a warning.

2.4 Why Interval Analysis?

Other data flow analyses such as Sign Analysis and Constant Propagation can be used in order to detect guaranteed and potential out of bounds array accesses. However, each of these comes with their own set of limitations which make them less suited candidates for performing such an analysis.

Sign Analysis contains no numeric information about each variable at different program locations. Rather, it keeps track of the sign mapping variables to lattice values which are either Pos, Neg, Zero, $\top$, or $\bot$. Sign analysis does not map well to the problem space. The only time it is guaranteed that an array access is safe is when the variable maps to Zero and the array has a non-zero length. On the contrary, the only time an error is guaranteed is when the variable maps to Neg. If a variable is Pos or $\top$, it is unclear as to whether or not it is unsafe to use this variable to access an array.

Constant Propagation maps to the problem space much better than Sign Analysis does. By keeping track of the exact integer value of each variable, more precise information is available at each program point to determine whether or not an access is safe. One can now check if the lattice value of the variable is within the exact range of indices safe to access. This type of analysis can, however, provide less precise results. If a variable x can map to both constant c_1 and constant c_2, it's lattice value would be $\top$. With range analysis, assuming $c_1 < c_2$, we would have a more precise lattice

value of $[c_1, c_2]$ implying that x can be any of the values between c_1 and c_2 exclusive. Note that if the values variables can hold in a program have a wide range of possibilities, this will limit the precision of Interval Analysis.

3 Implementation

This analysis was implemented in Java using the Soot Framework. The link to the repository containing the implementation is: `https://github.com/jakezych/range-analysis`

This analysis is interprocedural and uses a k-string-cutoff context where k is set to 3. Based on the testing done, this value for k was found to be the best trade-off between precision and performance. Larger values of k may provide more precise results at the cost of a longer run-time. Depending on the type of code being analyzed, this may be worth changing. For example, a program where recursive calls are being made from the same line would benefit more from increasing k. Increasing k when testing other programs that do not have many nested function calls may result in slower performance with a lack of improvement in the precision of the analysis. Below are some of the important implementation ideas/challenges:

3.1 Range

Now that lattice values are intervals, we introduce a new class to encode this. The reason for this is because there are many different operations done on Ranges. Using a native datatype like an integer array would work, but creating a separate class that holds data and is responsible for performing operations on ranges improves cohesion. Below is code snippet of the method that combines ranges together based on the type of arithmetic operation.

```java
public static Range combine(Range r1, Range r2, Operator op) {
    switch (op) {
        case ADD -> {
            return new Range(sentinelAdd(r1.low, r2.low), sentinelAdd(r1.high, r2.high));
        }
        case SUB -> {
            return new Range(sentinelSub(r1.low, r2.low), sentinelSub(r1.high, r2.high));
        }
        case MUL -> {
            // Multiplying by negative flips the range
            int low = sentinelMul(r1.low, r2.low);
            int high = sentinelMul(r1.high, r2.high);
            return new Range(min(low, high), max(low, high));
        }
        case DIV -> {
            // Dividing by negative flips the range
            int low = sentinelDiv(r1.low, r2.low);
            int high = sentinelDiv(r1.high, r2.high);
            return new Range(min(low, high), max(low, high));
        }
    }
    // return Top
    return new Range(Integer.MIN_VALUE, Integer.MAX_VALUE);
}
```

Each operation is augmented to handle $-\infty$ and ∞ which are implemented as the maximum and minimum values a 32 bit integer can hold in Java.

3.2 Extracting Constants

In order to improve the precision of the widening operator, we use the set of all constants in the function to widen intervals to those values instead of jumping immediately to $-\infty$ or ∞. In order to implement this, we instrument the intraprocedural analysis to pre-process the control flow graph in search of all of the constants in the program. These are added to a set which is later referenced by the widening operator to determine whether we can widen a range to one of the constants. Below is the snippet of code from the constructor for the intraprocedural analysis that finds all of the constants:

```java
// Preprocess all constants to gather the max value any variable is set to.
for (ValueBox box : graph.getBody().getUseAndDefBoxes()) {
    if (box instanceof ImmediateBox) {
        ImmediateBox constantBox = (ImmediateBox) box;
        String constantExpr = constantBox.getValue().toString();
        if (isNumeric(constantExpr)) {
            constants.add(Integer.parseInt(constantExpr));
        }
    }
}
```

One limitation of this implementation is that it only works for integer constants. We use a helper function that attempts to pattern match the right hand side string of any expression to a number which is then parsed as an integer. A limitation of this is that we cannot use other numeric data

types such as doubles or floats. This is partly because we define our intervals in our analysis as integers since integers must be used to access arrays.

3.3 Applying the Widening Operator

The widening operator itself presents with two implementation challenges. Firstly, we must instrument the analysis to keep track of the previous lattice value for every variable at each program point. We model this using a HashMap that maps line numbers to the previous Sigma input value at that point.

```
// Used for the widening operator, keeps track of the previous sigma_i value at each instruction point
private Map<Integer, Sigma> previousSigma = new HashMap<>();
```

This is updated every time we process the head of a loop. It is unnecessary to update it in other places because widening is only applied at the head (or guard) of each loop in the function.

The other implementation challenge, which is much more difficult than the previous one is actually applying the widening operator at the start of each loop. Soot does not differentiate between traditional if statements and loop guards since loops are implemented as if statements and goto statements. Thus, we must do further preprocessing of the input control flow graph to determine which line numbers correspond to if statements that are actually the heads of loops. I used Soot's LoopFinder to identify these lines and save their line numbers into a set that can be referenced whenever processing an if statement in flowThrough.

```
LoopFinder loopFinder = new LoopFinder();

// Find all loop guards
Collection<Loop> loops = loopFinder.getLoops(graph);
for (Loop l : loops) {
    loopHeads.add(l.getHead());
}
```

When processing if statements, we can reference whether or not this line corresponds to the head of a loop. If so, we can apply the widening operator. We can also extract further information with this design as well. If a variable has a range that satisfies the conditional, we can map that variable to the exact value that satisfies the conditional, improving precision. This does, however, have the limitation of working best when the loop guard is an equality statement. For example, when the conditional is $x \leq 0$ and x can range from $[-\infty, 10]$, the most precise range we can give for x after processing the conditional is $[-\infty, 0]$.

```
if (guard.getSymbol().trim().equals("==")) {
    // if n is within range, assign it
    if (lRange.getLow() <= intExpr.value && intExpr.value <= lRange.getHigh()) {
        Range value = evaluateExpression(inValue, l, rhs, outValue);
        outValue.map.put(l, value);
    }
}
```

3.4 Reporting Warnings

In order to successfully report warnings in this type of analysis, we must reference the length of the array being accessed to know whether or not the range of values for the index is valid. Once the length is known, we can issue an error if the range of the index is completely disjoint from the range of possible values an index is allowed to take and a warning if the range of the index contains both valid and invalid index values.

This appears to be a point of poor extensibility for this analysis. The Soot framework is a static analysis tool, meaning we cannot dynamically reference the length of the array when reporting warnings. This proves to be a challenge as there are many different ways to initialize an array. My implementation searches for ”newarray” statements (from the jimple code) and pulls the length from the input to ”newarray”. However, one can also declare an already initialized array in Java as follows:

$$\textbf{int}\,[\,]\quad r = \{1,2,3,4,5\};$$

Furthermore, arrays can be passed in from other functions or referenced as fields which would make it very difficult to determine the length of the array in the analysis. The most straightforward solution to this issue is to instrument all of the code being tested with dummy array declarations matching the format the analysis code expects. An example is provided below comparing the original source code (left) to the instrumented code (right:

```
// Exchange Function
public static void exchange(int i, int j){
    int t=a[i];
    a[i]=a[j];
    a[j]=t;
    }
```

```
// Exchange Function
public static int exchange(int a[], int i, int j){
    int []ignore = new int[5]; // instrumented for reportWarnings
    int t=a[i];
    a[i]=a[j];
    a[j]=t;
    // instrumented for reportWarnings
    int x = ignore[i]; // OKAY
    int y = ignore[j]; // OKAY
    return 0;
}
```

This requires programmer effort to make separate array declarations for every array in the program, hardcoding its length to a constant. Such a solution does not scale well unfortunately. The other solution would be to add code to the analysis to handle more situations such as referencing fields or passing arrays as parameters which requires a significant amount of effort for the programmer.

However, this may be worthwhile to do if the size of the target code being analyzed is sufficiently large enough.

4 Results and Discussion

The implementation was tested on a variety of simple, hand-made programs with little to no real value, and a few pieces of real world software: a heap sort implementation and a linear interpolation algorithm used by a popular MineCraft mod WorldEdit. No bugs were found in the real world software. While Interval Analysis does have strong potential in being able to effectively catch array accessing bugs, it suffers from poor extensibility which is the reason why more real world software was not tested on and why it'd be challenging to expose bugs in a piece of real world software that the analysis is not specifically written to handle. Analyzing many different types of programs proves to be difficult because the analysis uses Soot Units which provide specific information about the line of code being executed. For example, even though the logic for checking out of bounds indexing is very similar for standard arrays and ArrayLists, one would need to add code that specifically checks and handles .get() calls and ArrayList declarations. If there are other data structures that are also being indexed into, these would need to be handled as well. Furthermore, if a program uses other types such as floats and doubles, these would also need to be explicitly handled in the analysis. Thus, it is preferable for a programmer to use such a tool when they are working with a specific domain and codebase. Applying the same analysis code to many different codebases will likely require a lot of effort to either instrument the target code to match what the analysis code expects or write more analysis code to handle new cases. The benefits of using such an analysis are not moot. While fuzzing and testing may expose the same kinds of indexing bugs, it does not provide the programmer with the exact location the issue is happening. It is on the programmer to determine what the exact cause of the bug is. When contrasted with Interval Data Flow Analysis, not only is the exact line number pointed out, one can also generate log files of the values that each variable can hold at every program point which can assist in debugging. Moreover, warnings are issued which can lead to further bug fixes.

5 Links and References

- `https://cmu-program-analysis.github.io/2022/resources/program-analysis.pdf`
- `https://github.com/jakezych/range-analysis`
- `https://www.sable.mcgill.ca/soot/doc/index.html`
- `https://github.com/farhankhwaja/HeapSort/blob/master/HeapSort.java`

- https://github.com/EngineHub/WorldEdit/blob/master/worldedit-core/src/main/java/com/sk89q/worldedit/math/interpolation/LinearInterpolatio
  java