CS6843: Jan-May 2016 Assignment-3 Specifications

Problem Statement: Write an LLVM pass to perform array out-of-bounds analysis

---

Accessing an array out of the bound is a run-time exception which results in serious security issues like corrupting the program stack. You may refer to an interesting article [1] to understand the importance of identifying such violations. Programming languages, like Java, handle array out-of-bounds as a language feature by generating run-time exceptions. However in languages like C and C++, it is the responsibility of the programmer to take care of this. You have to implement a flow-sensitive, intra-procedural static analysis (as an LLVM pass) to generate a set of security constraints, which ensure that each array access is safe.

1 Input

```
1  #include <stdlib.h>
2  
3  void fun(int p, int q) {
4    int *array1;
5    int *array2;
6    array1 = (int *) malloc(sizeof(int) * (p + q));
7    array2 = (int *) malloc(sizeof(int) * (p - q));
8    array1[2*p - 1] = array2[2*q + 1];
9    return;
10 }
```

Figure 1: Input C Program

The input C program will have a single function named `fun`, which accepts parameters of type integer. No `main` function will be there. One or more single dimensional arrays (of type `int *`) will be allocated in heap using malloc. An array access will be performed only after it is allocated using `malloc`. There will not be any `structs`. An array access can be a read or a write. Consider the example in Figure 1.

2 Output

You have to generate constraints, **in terms of the function parameters**, for safe array access. For each array access (read or write) we have to generate constraints. For example, at line 8 in Figure 1, for the write to `array1` the constraint for safe access is `2*p - 1 < p + q` and for the read access to `array2` it is `2*q + 1 < p - q`. For simplicity, we will ignore the lower-bound check (like `0 <= 2 * p - 1`).

The format of the constraint should always be as follows.

$$\mathtt{a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n + c_1 < b_1 * x_1 + b_2 * x_2 + ...b_n * x_n + c_2}$$

, where n is the number of parameters to the function, $\mathtt{x_i}$ is the $\mathtt{i^{th}}$ parameter, $\mathtt{a_i}$ and $\mathtt{b_i}$ are the coefficients of $\mathtt{x_i}$, and $\mathtt{c_1}$ and $\mathtt{c_2}$ are constants. For instance, the constraint generated for the write access to `array1` in line 8 of Figure 1 is $\mathtt{2} * \mathtt{p} + \mathtt{0} * \mathtt{q} + -\mathtt{1} < \mathtt{1} * \mathtt{p} + \mathtt{1} * \mathtt{q} + \mathtt{0}$

Note that the expression to the left of $<$ corresponds to the index of the array access. In all the test cases it is guaranteed that both the expressions to the left and right of $<$ will be affine (a linear expression plus a constant).

Your analysis should output a simplified version of the format which is discussed above. In llvm-IR each array access corresponds to a `getelementptr` instruction which essentially performs the base + index operation and stores the result in a temporary, say `arrayidx`. Then the expected output for each array access, printed in different lines is

$$\mathtt{arrayidx}\ \mathtt{a_1}\ \mathtt{a_2}\ \dots\ \mathtt{a_n}\ \mathtt{c_1}\ \mathtt{b_1}\ \mathtt{b_2}\ \dots\ \mathtt{b_n}\ \mathtt{c_2}$$

(Note that a total of `2*n + 2` numbers will follow `arrayidx`).
For example, for the write access `array[2*p - 1]` in Figure 1 the llvm-IR instruction is `%arrayidx9 = getelementptr inbounds i32* %10, i64 %idxprom8` and the expected output is

```
arrayidx9 2 0 -1 1 1 0
```

3 Conditional Statements

```
1  #include <stdlib.h>
2  void fun(int size) {
3    int *array; int k; int x;
4    array = (int *) malloc(sizeof(int) * size);
5    x = 12;
6    if (size <= 10) {
7      if (size == 5) {
8        k = 10;
9      } else {
10       k = 5;
11     }
12   } else {
13     k = x;
14   }
15   array[k] = 10;
16   return;
17 }
```

Figure 2: Input C program with conditionals

The testcases may have conditional statements as shown in Figure 2.

The constraint for the array access at line 15 would be $\mathtt{5} < \mathtt{size} \wedge \mathtt{10} < \mathtt{size} \wedge \mathtt{12} < \mathtt{size}$. For simplicity, to avoid the conjunction ($\wedge$), the expected output here is

```
arrayidx 0 5 1 0
arrayidx 0 10 1 0
arrayidx 0 12 1 0
```

4 Loops

The testcases may have loop statements too as shown in Figure 3.

```
1 #include <stdlib.h>
2 void fun(int size) {
3   int *array;
4   int k;
5   array = (int *) malloc(sizeof(int) * size);
6   k = 1;
7   while (k < 10) {
8     k++;
9   }
10  array[k] = 20;
11  return;
12 }
```

Figure 3: Input C program with loop

The constraint for the array access at line 10 would be $\texttt{1} < \texttt{size} \wedge \texttt{0} < \texttt{0}$. The constraint $0 < 0$ indicates the conservative approximation due to the update of the variable `k` inside the while loop. Here the expected output is as follows

```
arrayidx 0 1 1 0
arrayidx 0 0 0 0
```

Note that in testcases, the array would be allocated outside the conditional and loop statements.

Happy Hacking !!!!

References

[1] Smashing The Stack For Fun And Profit. https://wkr.io/public/ref/alephone1996smashing.pdf