Rough Dissertation

Peter E Conn

April 30, 2014

Chapter 1

Background

1.1 Pointer Safety

Playing with raw pointers gives power, but is dangerous. Leads to security holes. Example of buffer overflow attack.

1.2 Literature Review

Give a brief summary of the 25 papers I did in the literature review, talking about their approaches to pointer safety and going into the classifications: fat pointers/ metadata and object-based vs pointer-based approaches.

1.3 CCured Analysis

Explain the CCured system.

1.4 Cyclone Not-Null pointers

Check that Cyclone does classify pointers as being never-null, write about how it chesk for that.

Chapter 2

Implementation

2.1 Working with pointers in LLVM IR

In LLVM IR, variables are created with an `alloca` instruction, which returns a pointer to the allocated variable (which is allocated on the stack). Therefore the line `int a` in C is converted into `%a = alloca i32` in LLVM IR. It should be noted that the variable `%a` is actually the type of `i32*`. This means that in the declaration `int *a` will create in LLVM IR the variable `%a` of type `i32**`.

The `store` instruction takes a piece of data and a pointer and stores the data in the area pointed to. The following C code turns into the following LLVM code:

```
int a;
int b;
a = b;
```

```
%a = alloca i32
%b = alloca i32
%1 = load i32* %b
store i32 %1, i32 %a
```

Here, spaces on the stack for the variables `a` and `b` are created, and pointers to these spaces are stored in the variables `%a` and `%b` respectively. The value stored in the area pointed to by `%b` is loaded, and then stored in the area pointed to by the variable `%a`.

```
int *c;
int d;
...
*c = d;
```

```
%c = alloca i32*
%d = alloca i32
```

```
...
%1 = load i32** %c
%2 = load i32* %d
store i32 %2, i32* %c
```

Here, in the C code we are assigning the value held in `d` to the memory location pointed to by `c` (assume that `c` is set to a valid memory address during the `...` otherwise the code results in undefined behaviour).

In LLVM IR, the variable `%c` is created with type `i32**` and the variable `%d` is created with type `i32*`. The value stored in the memory pointed to by `%c` is loaded into `%1`, so `%1` now contains the address pointed to by `c` in the C code. The value stored in the memory pointed to by `%d` is loaded into `%2` (the value contained within `d` in the C code). Finally the memory pointed to by `%1` is set to the value contained in `%2`.

The last interesting instruction for working with pointers is the GEP instruction. GEP stands for 'Get Element Pointer', and is used for performing pointer arithmetic (it itself does not peform and memory access).

```
int *a, *b, *d;
int c;
a = b + 3;
c = d[3];
```

```
%a = alloca i32*
%b = alloca i32*
%c = alloca i32
%d = alloca i32*

%1 = load i32** %b
%2 = getelementptr %1, i32 0, i32 3
store i32* %2, %a

%3 = load i32** %d
%4 = getelementptr %3, i32 0, i32 3
%5 = load i32* %4
store i32 %5, %c
```

`%1` contains the value of `b` and the getelementptr is used for the pointer arithmetic. The variable `%1` is of type `i32*`. The second parameter of the GEP specifies how many of the size of the type of the first parameter we want to add. In this case, and in most cases it is zero, saying that we don't want to add any of `sizeof(i32*)` to the first parameter. The third parameter specifies how many of the size of the type of the value contained within the first parameter we want to add, and in this case, we want to add `3 * sizeof(i32)`. Therefore the GEP returns:

```
%2 = %1 + 0 * sizeof(i32*) + 3 * sizeof(i32)
```

This address is then stored in the memory pointed to by `%a`.

The second operation can be rewritten as `c = *(d + 3)`, so starts the same as the previous operation, except once the address of `d+3` has been calculated, it is dereferenced to get the value contained there.

2.2 Fat Pointers

While a pointer contains an address to an area in memory, a fat pointer contains an address and additional information. As implemented in Bandage, fat pointers contain three pointers, a value, a base and a bound.

```
int *x = malloc(5*sizeof(int));
x += 3;
```

In bandage, the variable x, of type `i32*` would be turned into a structure of type `{i32*, i32*, i32*}`. Assuming that malloc returned the address `0x1000`, the variable x would contain `{0x1000, 0x1000, 0x1020}` because it contains a pointer that currently points to `0x1000` and whose valid addresses start at `0x1000` inclusive and end at `0x1020` not inclusive. After the `x += 3` instruction, the variable x would contain `{0x1008, 0x1000, 0x1020}`.

Different pointer types give different fat pointer types.

2.2.1 Pointers

2.2.2 Structs

Sizeof Woes

2.2.3 Functions

2.3 Lookup Table

2.4 CCured Analysis

2.5 Non-Null Analysis

Chapter 3

Evaluation

3.1 Microbenchmarks

3.1.1 Fat Pointers

Fat pointers add a storage overhead in the pointer itself, tripling its size to that of three pointers. Fat pointers generate computational overhead on creation, where the base and bound must be set as well as the value and on dereference, where an extra load instruction must be added. Finally fat pointers create computational overhead on bounds checking.

3.1.2 Fat Pointers with CCured Analysis

Safe pointers as Raw or Fat Pointers

When combining the CCured approach with the fat pointer analysis, there is a choice of how to implement pointers designated as SAFE - these are pointers that do not require bounds checks on dereference. These SAFE pointers can either be implemented as a fat pointer but without bounds checking or kept as a raw pointer.

Implementing SAFE pointers as fat pointers but without the bounds checking brings consistency but keeps the overhead of fat pointers, which isn't used when SAFE pointers are kept as raw pointers.