Rough Dissertation

Peter E Conn

May 3, 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

During this chapter, types will be referred to in LLVM notation. An `i32` stands for a 32-bit integer, a `i8` stands for an 8 bit integer (or char) and a pointer to a type is denoted by the type followed by an asterisk.

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 `{0x1012, 0x1000, 0x1020}`.

In order to keep type safety in the LLVM IR, different types of pointers create different types of fat pointers, eg a `i32*` creates an `{i32*, i32*, i32*}` whereas a `i8*` creates a `{i8*, i8*, i8*}`. This will result in a fat pointer class for every pointer type in the program, but because class information does not propagate to the final binary, this shouldn't create any space overhead.

However, this adds some complexity when pointers are cast to different types. Previously, casting from a pointer to a pointer (for example casting from the `i8*` returned from `malloc` to the `i32*` for the integer pointer in the above example would take one bitcast instruction with raw pointers. With fat pointers, a new fat pointer would need to be created, the address of each field would need to be calculated, each field would have to be loaded, bitcast and then stored in the new fat pointer, adding a fair amount of overhead. The current implementation is optimized for this case (detecting whether the result of a `malloc` is going to be bitcast, and if so doesn't create the intermediate fat pointer). **It may be worthwhile seeing how well this optimization does**.

2.2.1 Types of Fat Pointer

During the running of the program, each fat pointer can be classified as one of three types: *HasBounds*, *NoBounds* and *Null*.

The *Null* fat pointers have their value set to `NULL`, and no restrictions on their base or bound. They represent null pointers and throw an error whenever they are dereferenced.

The *NoBounds* fat pointers have their base set to `NULL` and not-`NULL` in their value. They represent fat pointers whose bounds we do not know, for example the result of a external function call. These pointers cannot be bounds checked on dereference, so do not throw an error.

Finally, textitHasBounds pointers have not-`NULL` in all of their fields and represent a pointer whose bounds we know. These pointers are bounds checked on dereference and throw an error if the value is out of bounds.

2.2.2 Pointers

Allocation

The first step in the pointer transformation is to find all `alloca` instructions that create a pointer, and replace them with a fat pointer. Even at this early stage we can add some safety to the program by initialising the fat pointer value to be null.

In terms of performance, we are swapping one allocation instruction allocating a pointer's worth of memory with another, allocating three pointers worth of memory. On testing this was found to produce no performance difference.

When the value was set to `NULL`, an overhead of 1.7

Instruction-based vs Chain-based

The original implementation of bandage moved through the program and transformed it on an instruction-by-instruction basis, gathering all information from the transformation from the instruction being transformed (eg the opcode, its arguments, the types of the arguments and the instruction type).

It was found that this approach, though originally simple, rapidly gained complexity as more sophisticated transformations were needed. Upon integration with the CCured pointer analysis, the implementation was switched to a different, more holistic approach, that of instruction-chains.

An instruction chain is a sequence of instructions, where every instruction is used by the subsequent instruction. Under this implementation, the source is searched for allocation instructions and call instructions - the two instructions that start instruction chains, and their usage was followed.

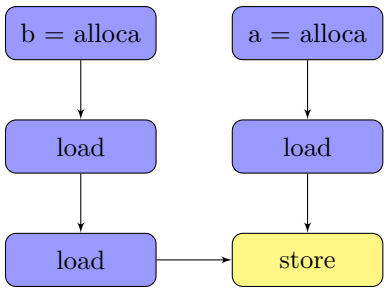
These instruction chains tend to terminate in one of a few common instructions:

- **Store** - A store instruction usually terminates two instruction chains, that of the pointer and that of the value.
- **Call** - A call instruction terminates the instruction chains of all of the parameters that are used in it. Additionally it creates a new instruction chain of its return value (if that return value is used).

- **Return** - Terminates the instruction chain of the return value.
- **Compare** - Similar to a store instruction, this terminates the two instruction chains of its operands.

The following diagram shows the two use chains generated by the following code:

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



Loads

Loads are where most of the work occurs with fat pointers.

First it must be determined whether the load needs be transformed to return the value contained within the fat pointer, or whether the load needs to return the fat pointer. The latter case may arise when the load is to be used as the value in a store (where we store the fat pointer in anther fat pointer) or when the load is loading the parameter for a function call. Checking for these cases is an example where the chain-based view of the data allows simpler checks than the instruction-based view.

If it is determined that the value of the fat pointer must be loaded instead of the fat pointer itself, bounds checking can occur. The value, base and bounds values are loaded out of the fat pointer and have arithmetic performed on them. If the value is less than the base, or greater than or equal to the base, a user defined function is called.

The transformation for the code derefencing the variable `a` is shown below, with the raw pointer code being:

```
%1 = load i32** %a
```

Which will be transformed into:

```
%value_addr = getelementptr %FatPointer* %FP.a, i32 0, i32 0
%value = load i32** %base_addr
%base_addr = getelementptr %FatPointer* %FP.a, i32 0, i32 1
%base = load i32** %base_addr
```

```
%bound_addr = getelementptr %FatPointer* %FP.a, i32 0, i32 2
%bound = load i32** %base_addr
call void @BoundsCheck(i32* value, i32* base, i32* bound)
%1 = load %value
```

The `getelementptr` instructions get the addresses of the fields of the fat pointer. Though the Bandage implementation creates three new `getelementptr` instructions on every dereference, these offsets are constant, and so repeated calls should be removed further down the compilation pipeline.

The cost of fat pointer dereference comes from the additional loads, and the bounds checking.

If bounds checking were ignored, overhead would be introduced because on dereference there are now two loads instead of just one - there is one to get the pointer out of the fat pointer, and one to dereference the pointer. **This would be a nice place for a microbenchmark on the raspberry pi**

Geps

2.2.3 Functions

Functions that are declared in the provided source files are converted into ones compatible with fat pointers.

Functions are duplicated into a function with a modified signature, such that every parameter is replaced by the fat pointer version of that parameter (so a pointer is replaced by a fat pointer, a struct is replaced by a version of the struct that uses fat pointers).

A map from original functions to fat pointer capable functions is kept, and when a call instruction is encountered, the call target is updated to the fat pointer version of the function if it exists.

If no such fat pointer version exists, there may need to be some conversion code around the call. This code will strip any fat pointers that are passed as parameters into raw pointers and, if the function returns a raw pointer, will wrap this into a fat pointer. This, newly created fat pointer will have its base set to `NULL`, making it a *NoBounds* pointer described above.

malloc

When a `malloc` instruction is encountered, the fat pointer being assigned to has its value and base set to the return value, and its bound set to the base plus the argument to the malloc instruction (the size of the area of memory to be allocated).

free

When a `free` instruction is encountered, the argument is followed backwards to find the fat pointer it came from. The value of the fat pointer is set to `NULL`.

Evaluate String Function

2.2.4 Structs

Structures need to be modified to hold fat pointers instead of pointers, eg:

Sizeof Woes

2.3 The Bounds Check Function

2.4 Lookup Table

2.5 CCured Analysis

2.6 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.