

# Rough Dissertation

Peter E Conn

May 10, 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

CCured designates pointer as one of three types: *SAFE*, *SEQ* and *DYNAMIC*. **Explain what each of these mean** While the CCured language itself has pointers explicitly marked as one of these types, its main goal is to allow the use of CCured with existing unmodified C programs and it uses a type inference algorithm to do so.

Every pointer is annotated with a \*qualifier variable\*, which is one of the three above types. We'll use  $Q(a)$  to mean the qualifier variable of  $a$  and  $T(a)$  to mean the type pointed to by  $a$ . These can be recursive, with the type pointed to by  $a$  being a pointer itself, for example  $a$  could be of type *int ref SAFE ref SAFE*, in which case  $T(a) = \text{int ref SAFE}$ .

#### 1.3.1 CCured Constraint Collection

There are three operations that generate constraints in a C program, these are arithmetic, casting and assignment.

Arithmetic is the simplest one of these, and says that if a pointer has arithmetic performed on it, it cannot be *SAFE*.

```

int *a;
int *b;

a = b + 4;

```

Two constraints are generated here, one from the arithmetic and one from the assignment. The arithmetic constraint marks the pointer that has arithmetic performed on it, so in this case,  $Q(b)! = \text{SAFE}$ .

Assignment is slightly more complicated, the assignment  $\mathbf{a} = \mathbf{b}$  generates the following constraints if both  $\mathbf{a}$  and  $\mathbf{b}$  are pointers:

```

Q(a) = Q(b) = DYNQ
\ / (
    (Q(a) = Q(b) = SAFE
    \ / Q(a) = Q(b) = SEQ
    \ / Q(a) = SAFE /\ Q(b) = SEQ)
\ / T(a) = T(b)
)

```

The first line is a provision that a *DYNAMIC* pointer can be set to a *DYNAMIC* pointer regardless of the types.

The remaining lines allow the matching up of qualifiers given if the types pointed to are equal. A *SAFE* pointer can be set to a *SAFE* pointer, a *SEQ* pointer can be set to a *SEQ* pointer and a *SAFE* pointer can be set to a *SEQ* pointer. In the last case, a bounds check is performed to ensure the safe pointer is set to a value within the bounds of the sequential pointer. After the assignment, the safe pointer contains no further bounds information.

One final case exists for the assignment of  $\mathbf{a} = \mathbf{b}$ , where  $\mathbf{b}$  is an integer and  $\mathbf{a}$  is a pointer. In this case, the only constraint generated is  $Q(e)! = \text{SAFE}$ .

Similar constraints are generated on a cast.

After the source code is iterated through and all of the constraints are generated, a constraint solver is run and the qualifiers for all variables.

## 1.4 Cyclone Not-Null pointers

In the CCured system, even *SAFE* pointers must undergo null checks on dereference. The Cyclone dialect of C contains the notion of a not-NULL pointer - one that can never hold the value NULL.

## 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 perform 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:

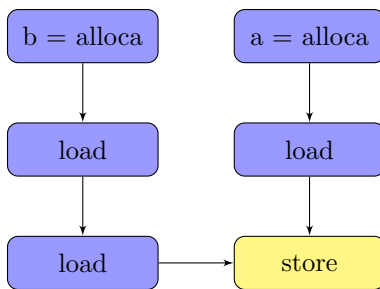
- **Store** - A store instruction usually terminates two instruction chains, that of the pointer and that of the value.
- **Call** - A call instruction terminates the instruction chains of all of the parameters that are used in it. Additionally it creates a new instruction chain of its return value (if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

For comparison between a fat pointer and meta-data approaches, the SoftBound system was implemented. SoftBound was chosen because, apart from differences in where the bounds information is stored, it is very similar to the fat pointer approach as it associates each pointer with a base and bound.

#### 2.4.1 Pointers on the Stack

The SoftBound approach for local pointers with only one layer of indirection is very simple, two additional pointers are created alongside the original one to hold the base and the bound.

```
// Before
int *ptr;
// After
int *ptr;
int *ptr_base;
int *ptr_bound;
```

This is essentially the same as the fat pointer approach, except the data is stored as multiple variables instead of all in one structure. However, whereas replacing all uses of the pointer with a fat pointer then requires modifications in its further uses, this approach breaks nothing further down the line.

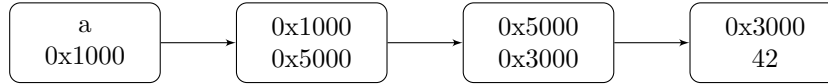
This is implemented by iterating through all instructions in the program and acting on `allocas` of pointer types. On finding such an `alloca`, two new pointers of the same type are created, and references of them are stored in a map, indexed by the reference of the original, such that further uses of the original pointer can be used to find its associated variables.

#### 2.4.2 Pointers on the Heap

With the fat pointer approach, when a pointer is allocated on the heap its base and bounds are allocated alongside it, in the fat pointer structure. SoftBound

uses a table data structure to map the address of a pointer in memory to the base and bound for that pointer.

The base and bound are retrieved from the lookup table, indexed by the address of the pointer on every load. For example, consider the case shown below, where each box contains the name of a variable, if it is local or its address if it is not, and its value below it.



The local variable **a** points to the address **0x1000**, which points to the address **0x5000** which points to the address **0x3000** which contains **42**. This would be accompanied by the following lookup table:

Address	Base	Bound
0x1000	0x5000	0x5008
0x5000	0x3000	0x3004

Consider the following code:

```
***a=12;
```

The first dereference is simple and requires no lookup as **a** is a local variable and therefore has other local variables associated with it containing its base and bound. The second dereference, loading the value from the pointer stored at **0x1000**, is accompanied by a table lookup using the address of the pointer. This lookup finds the base and bounds and performs the bounds check.

The third and final dereference follows the same pattern, loading the value at **0x3000** from the pointer which it itself stored at **0x5000**, using the address of the pointer (**0x5000**) as a key to the lookup table and getting the correct bounds.

### Table lookup

This table lookup is performed whenever a pointer stored on the heap is dereferenced, and this is what introduces the major disadvantage of the lookup table approach. Such a hash table must map from the address of the pointer to the base and bounds of that pointer.

Two different implementations of lookup table were implemented to compare the trade-off between size and lookup cost.

### Hash Table

The lookup table was implemented using **uthash**, a hash table implementation written in C and released under the BSD revised license.

A hash table uses a hashing function to split its contents into a series of buckets, each of which contains a list of elements. On access, the index is hashed to find the correct bucket and the contents of the bucket are scanned linearly to find the correct item. In order for this scan to run in constant time, the number of items in the bucket must be bounded.

In `uthash`, the limit of items in each bucket is set to 10, and once this is exceeded, the number of buckets is doubled and items are redistributed into new buckets. This means that lookup time has a constant bound (though it still a linear lookup, which would take longer than an array access), though adding items to the hash table can either be constant time, or can cause the buckets to be resized, an operation linear in the size of the hash table.

The advantage of a hash table is that it is quite small, with its size depending on the number of entries it contains.

### Memtable

In contrast to the hash table is `memtable`, which maps from areas of memory. It essentially creates a linear array of entries, with each entry covering a range of addresses. Since we are mapping from pointers, each area of memory mapped from is 8 bytes.

In this case, the size of the hash table is large, as it consumes space for every field that it could contain, however value lookup consists of pointer arithmetic and an array access so is constant time, as is value insertion.

### 2.4.3 Functions

Since pointers are copied in function calls, the address of the pointer cannot be used to carry information about the pointer's base and bound across the function boundary. Therefore, locally defined functions are modified so that their base and bound are passed in explicitly as additional parameters.

As with the fat pointer approach, this required duplicating all provided functions to those with modified signatures. Special care must be taken because there is no longer a one-to-one mapping between the parameters in the original function and those in the modified function.

```
// Before
int Func(int *x);
// After
int Func(int *x, int *x_base, int *x_bound);
```

Additionally, the base and bounds of the return value must be transferred across the function call boundary. This involves creating a structure that contains the pointer value, base and bound and returning this instead.

There are two key differences between this returned structure and a fat pointer. The first is in the purpose, a fat pointer will be returned from a function and then used as a fat pointer. The return structure will be returned from the function and immediately have its information stripped from it into local variables, it is purely a vehicle for returning multiple values.

The second is in how these structures deal with nested pointers. A multi-level pointer, when transformed into a fat pointer, has each layer of indirection turned into a structure, eg:

```
// Before
int **a;
// After
struct FatPointer.1{int *value, *base, *bound;};
struct FatPointer.2{struct FatPointer.2 *value, *base, *bound;};
struct FatPointer.2 a;
```

This is because a multi-level fat pointer must contain all of the bounds information with it. On the other hand, with `SoftBound` we only care about the pointer we are passing, so:

```
// Before
int **a;
// After
struct ReturnPointer{int **value, **base, **bound};
struct ReturnPointer a;
```

We only need the bounds of the immediate pointer since the bounds of anything that it points to will be stored in the lookup table.

In terms of implementation, the duplicated functions returned the 'pointer return' object and code was added at each call site to extract the value, base and bound and to associate the value with the base and bound.

#### 2.4.4 Setting pointer bounds

### 2.5 CCured Analysis

### 2.6 Not-Null Analysis

Whereas the previous analysis is concerned with spatial safety (overrunning the end of the valid space allocated to a pointer), Cyclone's Not-Null analysis is concerned with temporal safety.

In the CCured analysis, *SAFE* pointers are allowed to be null, and a null check is still inserted before dereference. As an optimisation, a simplified form of not-null analysis was implemented, in order to remove some of the redundant null checks.

#### Manual Null Checks

Frequently the programmer will check for null themselves, such as this example from the olden benchmark `TreeAdd`:

```
int TreeAdd(tree_t *t){
    if(t == NULL)
        return 0;
    int leftval = TreeAdd(t->left);
```

```
int rightval = TreeAdd(t->right);
int value = t->val;
return leftval + rightval + value;
}
```

Here, obviously adding manual null checks to the dereferences of `t` would be pointless.

Bandage searches for compare instructions, comparing pointers to null, and follows them to check for a jump instruction. If a jump instruction is found, the basic block jumped to on a successful null check (where the pointer is not null) is marked as not requiring null checks for that pointer. The pointer, if not *SAFE* will still undergo bounds checks.





## Chapter 3

# Evaluation

### 3.1 Performance

#### 3.1.1 Microbenchmarks

A selection of tailored micro-benchmarks were created to investigate the effects of the transformation on individual parts of code.

##### Safe Pointer Dereference

```
// Setup
int x;
int y=malloc(sizeof(int));
// Benchmarked Code
x=*y;
```

As `y` is recognised as a safe pointer, no bounds checking will be carried out, however at the moment the variable is not marked as `not-NULL`, therefore a null check is performed. This null check incurs an overhead of 4.6%.

If there null check were omitted, the only overhead of the fat pointer approach would be the load required to retrieve the fat pointer value. This benchmark was run without the null check, and the load was found to incur an overhead of 0.9%.

##### Unsafe Pointer Dereference

```
// Setup
int x;
int y=malloc(sizeof(int));
x=y[0];
// Benchmark Code
x=*y;
```

By using array addressing on the pointer, the CCured analysis detects `y` as a pointer that has arithmetic done on it, and is therefore not *SAFE*. Therefore the pointer dereference will contain the full bounds check, which was found to incur an overhead of 52%.

The bounds check function used was complex, first it checked if the value was null, then if the base was null (signifying a pointer of type `NoBounds`), and finally if the value were within the base and bound. This could be simplified.

### Pointer Allocation

```
// Setup
void Fun1(){
void Fun2(){int *a,*b,...,*j;}
```

Many calls were made to `Fun1` and to `Fun2` and the difference in execution time was measured. This benchmark needed to be done this way because memory used by an allocation is not free until the scope it is allocated in is left, therefore if the allocation were performed in a loop, the stack would run out of space.

This was found to produce no measurable difference.

### Pointer Assignment

```
// Setup
int *a,*b;
// Benchmark Code
a=b;
```

There is no bounds checking on this code as no pointers are dereferenced, its purpose is to observe the overhead of copying three pointers instead of one.

This was found to produce no measurable difference.

### Cache Contention

Since fat pointers are three times as large as raw pointers, they cause increased cache usage. For this benchmark, bounds checks were disabled.

**Unfortunately, on zenith this wasn't found to produce any performance difference.**

## 3.2 Misc

### 3.2.1 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.

### 3.2.2 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.