# A Safety Enhancing Source Translation for C

Peter E. Conn
Trinity Hall

UNIVERSITY OF CAMBRIDGE

*A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Engineering in Advanced Computer Science*

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: pc424@cl.cam.ac.uk

May 28, 2014

# Declaration

I Peter E. Conn of Trinity Hall, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: xx,xxx

**Signed**:

**Date**:

# Abstract

Current research into the area of providing pointer safety to C programs suffers from a few systematic flaws. First, the analysis performed by such systems is inextricably linked to the transformation required to provide pointer safety. Second, they are studied in isolation and not in combination with commonly used compiler optimizations or as part of a widespread compiler tool chain. Finally, the majority of these approaches sacrifice binary compatibility, requiring the entire libraries a program uses to be recompiled as well or even the program to be rewritten.

This dissertation aims to provide solutions to these problems. It uses an approach where the analysis and transformation are kept separate, with a single analysis working with two drastically different transformations. It investigates the effects of the two prevailing methods of ensuring pointer safety (fat pointers and lookup tables) on commonly used optimizations passes. It provides methods for maintaining very high binary compatibility with programs that use already compiled libraries, and evaluates the costs of doing so.

Additionally, a taxonomy is created for classifying and evaluating current methods for providing pointer safety. It goes beyond the currently quantitative measure of performance overhead imposed by a pointer safety system and includes qualitative measures including the types of safety provided, completeness and compatibility. Finally an argument is made that such techniques tend to fall on a spectrum between providing 100% safety and 100% compatibility and places current techniques on that spectrum.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Pointer errors account for many bugs in C, ranging from simple off-by-one errors where the programmer writes one past the end of the array to malicious buffer overflow attacks, where an attacker inputs data designed to manipulate the code into writing past the end of the array and over some other important data.

C allows many practices that make such errors easy to make, for example a pointer can legally point one past the end of an array (although dereferencing it is undefined). A pointer frequently starts pointing to its valid area of memory and then is modified (through pointer arithmetic) to point to an invalid area of memory. This pointer is modified again, bringing it back to point to the valid area and used (though used in practice, this is not allowed in the standard).

One method of dealing with such attacks is to keep track of data about the valid area of memory that a pointer has access to, and consult this data on pointer dereference. In this dissertation, the data used to determine the validity of a pointer is the address of the start of the area of allocated memory (the base) and the address one past its end (the bound). Furthermore, the base and bound data must be associated with the pointer in some way.

Implemented for the purposes of this dissertation was the Bandage system,

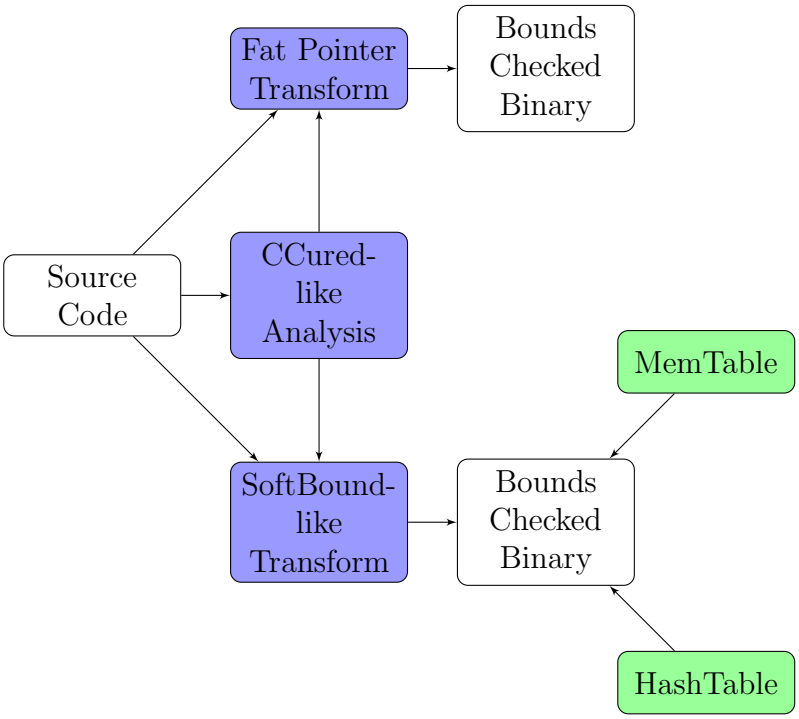



Figure 1.1: Overview of Components of Bandage

consisting of multiple parts as shown in Figure 1.1. Bandage consists of three LLVM passes and two C libraries. The two transformation LLVM passes implement different methods of associating data about pointers with the pointers themselves. With fat pointers, pointers themselves are replaced by structures that contain the pointer value and their bounds information. With a lookup table approach, the address of the pointer is used as the key to a lookup table containing the bounds information. The two C libraries contain different implementations of the lookup table.

The CCured-like analysis is used to identify the many pointers not modified using pointer arithmetic during the life of the program, and can be used by the transformation passes to prevent bounds checking there.

The implementation of Bandage is the first example of the separation of analysis and transformation and serves as a good example of its benefits. By having the CCured-like analysis being an independent LLVM pass is can

be combined with different transformations to achieve different trade-offs in terms of safety, runtime and binary compatibility, allowing a much larger range of potential solutions to a specific need.

The two different transformation passes are used to investigate the interplay with common optimisation passes - to determine which optimization passes do the most to mitigate the overhead generated by them and which passes they impede.

The entire Bandage system is designed to provide high binary compatibility by drawing a boundary around the source files it can transform and ensuring that the instrumentation is stripped when it passes through this boundary. This allows programs using Bandage to also use libraries that haven't been modified, but also restricts the safety completeness it can offer, as pointers that come in through this boundary do not contain sufficient information for full safety.

Finally, Bandage will be used as the primary example in the classification of pointer safety systems, providing a reference point to compare other systems with and a in depth study to highlight some subtleties in the taxonomy.

This dissertation continues with brief coverage of the background information required: the types of pointer safety, examples of vulnerabilities caused by lack of it and an overview of how pointers are dealt with in LLVM IR. A related work section follows, enumerating previous efforts to provide pointer safety, drawing together common themes between them and highlighting the position of this work.

The design and implementation section contains details of the two transformation passes (the fat pointer and the metadata) and the one analysis pass (the CCured-like analysis). The evaluation section covers the runtime penalties, tested on handcrafted microbenchmarks and the olden benchmark suite, alongside performance security focussed correctness tests. Everything is wrapped up in the conclusion and a few example source programs are annotated in the appendices to give a concrete view of the transformations performed.

# Chapter 2

# Background

## 2.1 Pointer Safety

One of the primary advantages of C is that it gives the programmer a model that is very close to hardware, allowing access to essentially arbitrary areas of memory. Though this is very powerful, it can also be very dangerous and numerous security vulnerabilities have arisen from programmers not putting careful checks on their memory operations.

### 2.1.1 Example: Buffer Overflow Attack

In a buffer overflow attack, the attacker makes use of the memory layout of the program to overwrite a variable that they should not have access to.

```
#include <stdio.h>
#include <string.h>

int main(void){
    char user[16];
    char pass[16];
    int userid = 0;
```

```
    printf("Username: ");
    gets(user);
    printf("Password: ");
    gets(pass);

    if(PasswordCorrect(user, pass))
        userid = GetUserId(user);

    if(userid == 0){
        printf("Invalid username or password.");
        return;
    }

    ...
}
```

```
// In stdio
char *gets(char *buf)
{
    int c;
    char *s;

    for (s = buf; (c = getchar()) != '\n';)
        if (c == EOF)
            if (s == buf)
                return (NULL);
            else
                break;
        else
            *s++ = c;
    *s = 0;
    return (buf);
```

```
}
```

In this example, the main function simulates a login, the user is asked to enter a username and password, these are checked against each other, the user id is fetched and if `userid` is still 0, they are kicked out as they have failed to log in. The problem arises from the implementation of `gets` (a simplified version taken from Apple's libc - the original version printed a warning notifying the user that `gets` is unsafe).

The variable `s` is set to point to the first byte of `buf`, and for each character read in through `getchar` that isn't a newline or end-of-file, the memory location pointed to by `s` is set to that character and `s` is incremented. The problem here is that there is no bounds checking performed on `s`.

The attack occurs as follows:

- The attacker provides a password that is longer than 16 characters.
- `gets` is called, with `pass` being passed in as the argument.
- The for loop continues until a newline in countered, in this case continuing to write past the memory allocated for `pass`.
- `userid` is positioned in memory after the end of `pass`, so when `gets` writes past the end of `pass`, it overwrites the value of `userid`.
- `userid`, which was originally set to zero, and assumed by the programmer to be unchanged unless the password and username match, has been checked to a non-zero value.
- The attacker is given a valid `userid`, and can even set the userid to be whatever they want to be by varying the 17th character of the input password.

### 2.1.2 Example: Heartbleed

A more topical example is the Heartbleed vulnerability found in the OpenSSL library - an open source implementation of the SSL and TLS protocols. The vulnerability, found in April 2014 is the opposite of a buffer overflow. Where, in a buffer overflow the attacker can write past the allocated memory to overwrite other values, the heartbleed bug allows the attacker to read past the memory allocated for the legitimate information and read values stored in RAM (such as passwords and private keys).

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
...
unsigned char *buffer, *bp;
int r;

buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```

The variable `p` points to a `ssl3_record_st`, the incoming message. The first byte of the message is read (the type of the record) and is stored in `bhtype`, and then the `n2s` macro takes two bytes from `p` and stores them in the variable `payload`, which is designed to contain the length of the message. Finally, `pl` is set to contain the rest of the received message. Note that the received message is received from a potentially untrusted source.

A buffer is created and has `1 + 2 + payload + padding` allocated for it.

The response message is created in `bp` and its first byte (its record type) being set to `TLS1_HB_RESPONSE`, and with the macro `s2n` copying two bytes from `payload` into `bp`. Finally, `memcpy` is called, which copies `payload` bytes of `pl` into `bp`. This was originally intended to copy the rest of the heartbeat message back into the response but is the line that introduces the vulnerability.

The bug comes into play when the variable `payload`, originally extracted from the adversary originating message specifies a size bigger than the rest of the package. `memcpy` will copy past the end of the legitimate data stored in `pl` and start copying bytes into `bp` of other variables and essentially whatever happens to be sitting in RAM nearby `pl` at the time. Finally `bp` is sent back to the adversary.

### 2.1.3 Example: Returning a Pointer to the Stack

This final example shows a nasty bug, one that potentially could be used for a security vulnerability, but since the code normally does not function correctly it hasn't been a major exploit. This bug stems from the programmers misunderstanding of the lifetime of variables on the stack.

```
Triangle *CreateTriangle(Point *a, Point *b, Point *c){
    Triangle T;
    T.a = a;
        T.b = b;
    T.c = c;
    return &T;
}
```

The triangle `T` is created as a local variable, on the stack and modified. A reference to it is then returned from the function. However, when the function exits, the local variables lifespan ends, resulting in the returned pointer pointing to deallocated space that will probably be overwritten on the next function call.

### 2.1.4 Spatial and Temporal Pointer Safety

There are two types of safety that can be provided when dealing with pointers - temporal and spatial. Temporal safety ensures that the pointer points to a valid area of memory at the time that it is dereferenced, while spatial safety just ensures that the pointer points to a valid area of memory at a certain time. In the examples provided above, the first two were violations of spatial pointer safety - in the first data was written beyond the end of the object it was meant to write to and in the second, data was read beyond the end of the object it was meant to write to. The last example was a violation of temporal pointer safety - a pointer is returned an object that is past the end of its lifetime.

The methods explored in this dissertation provide spatial safety, but not complete temporal safety. Consider the following code:

```
int *a=malloc(sizeof(int));
int *b=a;

free(a);

printf("%d\n", *a);
printf("%d\n", *b);
```

The methods implemented in this dissertation will catch the memory access error introduced by the first `printf` statement, but not by the second. This is because the information about pointer validity is associated with the pointer, and so when `a` is freed the information associated with it indicates that it does not point to valid memory, but the information associated with `b` is not likewise updated.‘

Therefore the techniques implemented in this dissertation do not provide complete pointer safety, but provide an additional layer of robustness.

## 2.2 Allowances in the C standard

The C99 standard defines loose rules for pointer related operations, which allow the transformations performed by Bandage to result in a program that adhers to it [1].

By omission in section `6.2.6.1` of the standard, the representation of the pointer type in unspecified. This allows pointers to be represented by more informative data structures that just the address they point to, for example a fat pointer representation.

The C standard places the following notable restrictions on pointer types (unless otherwise stated, these are contained in section `6.3.2.3`).

- A pointer to any object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer.
- An integer constant expression with the value 0, or such an expression cast to type `void *`, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function. Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.
- An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

These restrictions place `void` pointers as the lowest common denominator, and state that NULL must be equivalent for all pointer types.

However, it also provides a number of allowances that pointer safety schemes can use:

- The value of a pointer becomes indeterminate when the object it points

to (or just past) reaches the end of its lifetime (6.2.4.2).

- When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. ... If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated. (6.5.6)

- If an invalid value has been assigned to a pointer, the behaviour of the * pointer is undefined. Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. (6.5.3.2).

The first point would allow creation of a simple temporal pointer safety scheme where the pointer parameter passed to the `free` function is set to null after the function returns, while adhering to the C standard.

Bandage makes use of the latter points, allowing undefined behaviour in the case of pointers pointing past the ends of arrays (or objects) being dereferenced. However in practice, it is common for programmers to manipulate pointers such that they point out of the bounds of arrays (2 past the end, or before the start), and this is expected to be defined behaviour. Bandage is conservative and only causes undefined behaviour for dereferences to these pointers.

## 2.3 Working with Pointers in LLVM IR

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.

In LLVM IR, space on the stack is set aside with the `alloca` instruction, which returns a pointer to the allocated memory ??. 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;
int *b;
int c;
int *d;



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.

# Chapter 3

# Related Work

Since C is such a popular language and pointer based bugs are both widespread and highly exploitable, there have been many efforts to introduce pointer safety to C, and many different approaches.

LLVM's address sanitizer [2, 3] can be considered state of the art and is capable of detecting out-of-bounds accesses on the heap, stack and for globals, use-after-free, and some use-after return bugs.

It does this by creating a copy of memory, called shadow memory, where 1 byte of shadow memory maps to 8 bytes of real memory. This takes advantage of the fact that `malloc` is guaranteed to return an 8-byte aligned segment of memory, therefore a value of 0 in shadow memory means the corresponding main memory is valid, a negative value means the corresponding main memory is invalid and a positive value of $n$ means the first $n$ bytes are valid and the rest are invalid.

The `malloc` and `free` functions are modified so as to mark the shadowed areas of memory as valid and poisoned respectively. Additionally, `malloc` is modified so that the memory surrounding that allocated to the program is poisoned to prevent overflows.

However, address sanitizer provides no mapping between the valid areas of memory and variables. It would be possible for pointer arithmetic to be used

to to still cause a buffer overflow into another variables' valid area of memory, though it must jump over the poisoned area.

Hardbound criticizes prior work, saying that it either introduces high overhead, isn't complete or introduces incompatibility [4]. In contrast it proposes to shift the bounds checking to hardware, and akin to SoftBound stores the bounds information separately from the pointer.

## 3.1 The Jones and Kelly System

An example of the table based approach is the Jones and Kelly approach [5]. At runtime an ordered list of objects in memory is maintained by tracking the uses of `malloc` and `free`. It makes use of the fact that every valid pointer-valued expression in C derives its results from exactly one original storage object. During pointer arithmetic, the referent object (the object pointed to) is identified in the object list using the operand pointer. The bounds information is retrieved from the object list and used to check if the result is in bounds.

The Jones and Kelly bounds checker uses a strict interpretation of the C standard, where pointers cannot point to invalid memory areas. Therefore out of bounds pointers are marked as such in a non-recoverable way (they are set equal to `-2`). To account for the standard allowed practice of generating a pointer pointing one past the end of an array, the bounds checker increases all arrays by 1 element.

In *A Practical Dynamic Buffer Overflow Detector* [6] it was found that 60% of programs tested did not adhere to the C-standard assumed in the Jones and Kelly approach and were therefore broken by the tactic of signifying illegal pointers by setting them to `-2`.

To combat this, they created a new approach, where the creation of an out of bounds pointer would result in the creation of an Out Of Bounds object created on the heap which contains the address of the pointer and the referent

object originally pointed to. These Out Of Bounds objects are stored in a hash table. Therefore on dereference, both the object list and the out of bounds hash table may be consulted to determine the validity of the pointer. In order to reduce the overhead from these two lookups, only strings are bounds checked on the rationale that they are the tool used in buffer overflow attacks.

*Baggy Bounds Checking* [7] is an alternate optimization of the Jones and Kelly system, based on reducing the lookup time. On a memory allocation, the size of the object is padded to the next power of two, enabling the size of the allocated memory to be stored more compactly as $lg_2(\text{size})$ taking the size of a single byte. Due to the lower memory overhead of a entry, a constant sized array is used instead of an object list. This allows a quick and constant time address calculation to be performed. Alternate methods are used for dealing with pointers pointing past the end of arrays as adding one element to an array could double the size it could take up.

This approach does not prevent out of bounds accesses as the size associated with the pointer (the allocated bounds) is larger than the size of the object (the object bounds), so it is still possible to exceed the bounds of the object. However it prevents dangerous overflows, since a pointer cannot access memory of an object that it was not created for.

## 3.2 Other Systems

*Efficient Detection of all pointer and array access errors* [8] uses a more complex fat pointer representation than just {value, base, bound} to extend their coverage beyond just spatial safety to include temporal safety.

The first additional field is the storage class enumeration, ranging over the values of Heap, Global and Local. This allows detection of erroneous deallocations (such as attempting to free a local varaible). The second field, a capability is more interesting. On memory allocation, a unique capability is created and stored in a capability table. The capability is deallocated once

the memory is freed (either through free or returning from a function). The presence of the capability referenced by a fat pointer is checked on pointer dereference. This was found to produce an overhead of between 130-450%.

*Cyclone* [9] takes a different route from source code analysis, being a dialect of C that allows the programmer to program in a C-like language that prevents buffer overflows, memory managements and format string attacks. It does this by imposing restrictions on C, such as limiting pointer arithmetic, disallowing unsafe casts and forbidding jumps into scopes. Additionally it provides extensions such as a never-null pointer type and fat pointer for arithmetic.

While some of these extensions are automatic, the programmer must make use of most of them explicitly, so existing programs must be converted to Cyclone. When tested, Cylcone produce overheads of between 0 and 250% while the translation required between 0 and 46% of the lines of code to be changed.

On the other end of the spectrum are tools that don't even need access to the source code of the program.

*Body Armour for Binaries* [10] targets a very specific type of attack - buffer overflows into non-control data, without requiring the source code or symbol table of the program. It essentially does this by reverse engineering the binary to extract information about the data structures that need protecting and rewriting it to contain checks on pointer dereference.

*Heapmon* [11] deploys a helper thread that stores two bits for every word on the heap to keep track of whether or not the area is allocated and whether or not the area is initialized. Memory leaks are therefore detected by looking for areas of allocated memory left over after the program exits.

To detect overflows, memory allocation is modified to leave unallocated areas between objects, so writing to that area will trigger an error. Heapmon can only deal with memory on the heap (not the stack or globals) and works at a word granularity, so errors of less than 3 bytes may not be detected.

## 3.3 SoftBound

One of the two main systems implemented in this dissertation is SoftBound [12]. It is a compile-time transformation that stores information about the valid area of memory associated with a pointer separately from the pointer.

By storing information separately from the pointer, memory layout doesn't change, enabling binary compatibility and reducing implementation effort, however it does require a search for suitable bounds information on pointer dereference. Additionally the paper contains a proof that spatial integrity is provided by checking the bounds of pointers on a store or load.

## 3.4 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 $T(a) = int$ ref $SAFE$.

### 3.4.1 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, $Q(b)! = SAFE$.

Assignment is slightly more complicated, the assignment `a = b` generates the following contraints if both a and 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 `a = b`, where $b$ is an integer and $b$ is a pointer. In this case, the only constraint generated is $Q(e)! = SAFE$.

Similar constraints are generated on a cast.

After the source code is iterated through and all of the constraints are gen-

erated, a constraint solver is run and the qualifiers for all variables.

## 3.5 Relation to Work Done

During the course of this project, two different LLVM transformation passes were developed. The first uses a fat pointer representation to keep track of the base and bounds information for allocated memory areas, whereas the second keeps this data separate from the pointers themselves.

Therefore, both approaches described in this dissertation use the same method for ensuring temporal pointer safety - checking of base and bounds. However the way they store the bounds information is different.

This should result in similar strengths and failings in terms of capability (what violations are caught), but different trade-offs in terms of performance.

# Chapter 4

# Design and Implementation

## 4.1 Overview

**Fat Pointers**

| Address | Value |
|---|---|
| 0x00000080 | 0x00000100 |
| 0x00000088 | 0x00000100 |
| 0x00000090 | 0x00000118 |
| ..... | ..... |
| 0x00000100 | 0x00004000 |
| 0x00000108 | 0x00004000 |
| 0x00000110 | 0x00004004 |
| ..... | ..... |
| 0x00004000 | 0x00000005 |

**Lookup Table**

| Address | Value |
|---|---|
| 0x00000080 | 0x00000100 |
| ..... | ..... |
| 0x00000100 | 0x00004000 |
| ..... | ..... |
| 0x00004000 | 0x00000005 |

| Address | Base | Bound |
|---|---|---|
| 0x00000080 | 0x00000100 | 0x00000118 |
| 0x00000100 | 0x00004000 | 0x00004004 |

Figure 4.1: Graphical Comparison of Fat Pointer and Lookup Table Methods

This section cover the implementation of the three core sections of Bandage: the fat pointer implementation, the SoftBound-like implementation and the CCured-like analysis. The two different implementation methods for tracking pointer bounds are contrasted in Figure 4.1. With fat pointers the bounds information associated with a pointer is stored alongside it in memory, while with the lookup table, the bounds information is stored separately and indexed by the address of the pointer.

## 4.2 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**.

### 4.2.1 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.

### 4.2.2 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, though could increase cache pressure (as the number of pointers that can fit on the cache are divided by three).

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

### 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-like pointer analysis, the implementation was switched to a different, more holistic approach, that of instruction-chains.

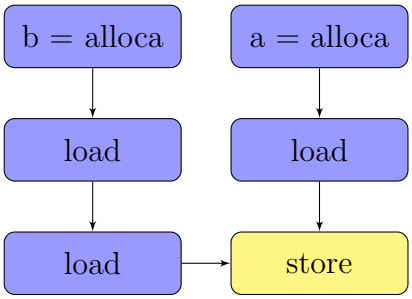
An instruction chain is a sequence of instructions, where every instruction is used by the subsequent instruction. Under this implementation, the source is searched for allocation instructions and call instructions - the two instructions that start instruction chains, and their usage was followed.

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

### 4.2.3 Functions

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.

### Multiple Source File Projects

Since LLVM module passes run on a single *.c source file, issues arise when using functions declared in other source files that the user has still written. When Bandage encounters a function definition but no declaration, it assumes that the function is external to the project, and is not to be modified. This works well for cases where the included function is part of a library that the programmer cannot modify, such as `malloc` or `printf`, however also is triggered when the function declaration is in another *.c file that the programmer can modify.

Therefore, an additional pass was created, the **Function List** pass. This pass, takes a single llvm IR file and a filename and outputs all of the functions declared in that IR file.

Now, the process for using Bandage is as follows:

- Run the **Function List** pass over all source files, appending to the same function list file.
- Run the **Bandage** pass over all source files, with the function list file as a parameter.

This gives the Module pass information about other modules that it would not normally have access to, and allows it to distinguish between function definitions for functions the programmer has no control over (so they can be ignored) and function definitions the programmer has control over.

On finding a function definition that matches a function in the function list file, Bandage transforms the function definition. This function definition will be identically transformed to the function declaration in the file it is declared in, and so the linker will be able to merge them together after the Bandage transformation is complete.

#### 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

### 4.2.4 Structs

First, it is nessecary to isolate the structs that can be modified - those that are defined and used only in the code that Bandage can modify. For example, Bandage should be able to recognise that `FILE` struct, included from `stdio.h` should not be modified.

To accomplish this, Bandage collects all of the structs used in the input file, and subtracts from this set those structs that are used in functions (as a parameter or return type) that are declared, but not defined. So for example, `FILE` would originally be flagged for modification, but upon discovery of the function declaration for `fopen`, it would be removed from the list. In order to play nicely with projects that consist of multiple source files, the function file used to determine which functions can be modified is used again.

Once the safe to modify structs are isolated, two modifications must be carried out on them:

- Pointers within the struct must be modified to fat pointers of the same object.
- Structs within the struct must be changed to structs that themselves are modified. This includes correctly modifying self-referential structs, such as a linked list.

The latter point is made simple due to an upgrade to the LLVM type system introduced in LLVM 3.0, type completion. This allows a type to be specified with no body, be used and have its body filled in later.

Therefore the algorithm to create the fat pointer types consists of:

- Collect a set of all struct.
- Subtract those structs that are used in functions that are defined externally.
- Create an empty type, for each struct in the set.
- For each struct in the set:
  - For each element in the struct construct a new element with a type such that:
    - For every layer of pointer indirection, there is now a fat pointer.
    - For every nesting of static arrays, there is a static array of the same dimension, but potentially of a new type.
    - If the base type is a struct which is in the set, change it to the (potentially empty) fat pointer capable struct.
  - Fill the previously empty type.

In this way, all structs can be modified in any plain linear order without having to worry about which structs reference other structs.

#### Sizeof Woes

The Bandage pass works on LLVM IR an intermediate stage created by the clang compiler. Unfortunately the clang compiler replaces instances of the `sizeof` operator with constant integers in this process.

The fat pointer versions of structs will always be equal to or larger in size than their raw counter parts, as pointers take up three times their previous size. This leads to the undesirable situation where the source program allocates an area of memory for a struct using the common `MyStruct *S = malloc(sizeof(MyStruct))` idiom. The LLVM IR will contain a constant integer (indistinguishable from other all other constant integers) representing the size of the original struct, though the actual struct used will be the fat pointer version, which requires more memory. This can lead to memory access violations.

## 4.3 Softbound-like Lookup Table

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.

### 4.3.1 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;
```

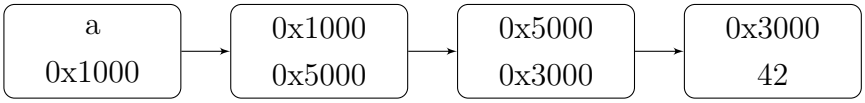
This is essentially the same as the fat pointer approach, except the data is stored as multiple variables instead of all in one structure. However, whereas replacing all uses of the pointer with a fat pointer then requires modifications in its further uses, this approach breaks nothing further down the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

This is implemented by iterating through all instructions in the program and acting on `alloca`s 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.

### 4.3.2 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 table must map from the address of the pointer to the base and bounds of that pointer.

Bandage was created with an exchangable backend, to allow multiple table implementations. The source program can load one of multiple provided headers to implement different types of lookup table, and the correct functions are called for table setup, teardown, lookup and storage.

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

### Toy lookup table

The first implementation is a toy example, consisting of a constant sized array that stores bounds that it is given and is linearly searched for lookup. If an attempt is made to add an entry to a filled table, the 'not-most-recently-used' entry is overwritten.

This table was partially created for debugging purposes, but also to examine the idea of imperfect bounds storing. Since most programs will have incomplete bounds information from calls to external functions (such as `gets`), the ideal of 100% bounds coverage is practically not achievable. Therefore it is interesting to investigate the performance advantage gained from relaxing the constraint that all internally that all internally declared pointers must have their bounds fully tracked.

### 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 a single pointer to a pair of pointers, each area of memory mapped from is 8 bytes to 16 bytes. The size of the table is very large, as it consumes space for every field that it could contain, however value lookup consists only of pointer arithmetic and derefence so is contant time, as is value insertion. However, the size of the table means it cannot be entirely contained within the cache, so variable access times may arise due to cache effects.

Current implementations of x86-64 do not allow programs to use the full range of addresses available to them, so for a pointer of 8 bytes/ 64 bits, only the least significant 48 bits will be used in address translation. Additionally, the last 16 bits are a sign extension of the 62nd bit, resulting in addresses falling into two categories, those that begin with 16 '1's and those that begin with 16 '0's.

The upper set of these addresses (those starting with '1') are used by the kernel on most operating systems, with the lower half being used by the application. Therefore, the memory table only needs to be able to deal with the lower set - a range of addresses extending from 0 to $2^{47}$.

However, the memtable needs to be contained within this address space as well, with a ratio of 2 bytes of storage required for every 1 byte the table is capable of mapping. To keep the arithmetic tidy, the available memory was partitioned into 4, with 3 quarters being devoted to the table and 1 quarter being used for application memory.

Finally, this application memory needs to have available space at both the top and the bottom. This is because the text and initialized data read from the program file are stored at the bottom of the available address space, and the stack and the heap are store at the top end of the address space.

**This will have a diagram with it.** Therefore, of the $2^{47}$ available addresses

the program can have, $2^{45}$ is still used for the program, whereas $3 \times 2^{45}$ are used for the memtable. The $3 \times 2^{45}$ area allocated for the table is positioned at offset $2^{44}$, effectively bisecting the application memory into that used for text and that used for the stack and heap.

Therefore, to find the bounds information associated with the pointer $x$:

- If $x$ is greater than $2^{44}$ (it is in the heap and stack region), subtract $3 \times 2^{45}$.
- Divide the address by 8 (the entry coverage).
- Multiply the address by 16 (the entry size).
- Add the table start offset $2^{44}$.

One of the primary aims of this scheme was to keep implementation simple, and its purpose was to provide a comparison for runtimes. Multiple improvements could be made. Firstly, the table could be shrunk to 2/3 of the available space, not 3/4. Secondly, a split of 50:50 between the text and stack and heap segments of memory may not be ideal.

The idea of some sort of compression for the bounds information was considered, for example storing the base and bound as 32-bit offsets from the value of the pointer instead of 64-bit absolute offsets, however this would require a table update on each piece of pointer arithmetic and would go against the design of SoftBound.

### Memtable for 32 bit architectures

The above implementation assumes a 64-bit architecture. In the evaluation section, benchmarks are run on a Raspberry Pi (to examine how well parallelism masks bounds checking). For this purpose, the memtable was modified to run on 32-bit architectures, which modified the arithmetic behind the memtable, but not the core concept.

The top $2^{30}$ addresses are reserved for use by the kernel, leaving $3 \times 2^{30}$

addresses for use by the application, again with the stack and heap on the top, and text and data at the bottom. This time, $2^{31}$ bytes are consumed by the memtable, with $2^{29}$ bytes left above for the stack and heap, and $2^{29}$ bytes left below for the text and data.

### 4.3.3 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.

## 4.4 The Bounds Check Function

Two bounds check functions can be used, one in practice and one for debugging to allow greater analysis of the program.

The first bounds check function is equivalent to the following code:

```
void BoundsCheck(void *value, void *base, void *bound){
    if(value == NULL){
```

```
        printf("Null Pointer Dereference");
        return;
    } else if(base == NULL){
        printf("No Bounds Set");
        return;
    } else if(value < base || value >= bound){
        printf("Out of Bounds Pointer Dereference");
        return;
    }
}
```

This bounds check function allows differentiation between the cases of a null pointer dereference and an out of bounds dereference and additionally allows the programmer to analyse their program at runtime to determine how many pointer dereferences cannot be checked due to lack of bounds information - for example for pointers that are returned from externally defined functions.

The optimized bounds and non-debug bounds check function is as follows:

```
void BoundsCheck(void *value, void *base, void *bound){
    if((base != NULL) & ((value < base) | (value >= bound)))
        OnError();
}
```

`OnError` is a function provided by the programmer. This optimized version is designed to be inlined and to provide as few jumps as possible (to prevent pipeline stalls).

## 4.5 Setting pointer bounds

The two most common actions that cause a change in pointer bounds are detailed in the functions section above - `malloc` and `free`. For the former, the base and bound of the set pointer are set to the return value and the

return value plus the parameter respectively. For the latter, the value is set to null.

The following sections cover the other common situations in which bounds information for a pointer is changed.

### 4.5.1 Setting to constant string

LLVM IR displays a distinct pattern for the following code:

```
char *username="george";
```

The constant string is stored in the file text section as a `private unnamed_addr constant [Y x i8]` (where `Y` is the length of the string, including null terminator). The assignment is a `store` instruction with the value being a GEP operator (which is different from a GEP instruction) to the start of the memory allocated to the string.

Since setting to a constant string is a static operation - the string and its length are known at compile time, setting the bounds is also a static operation. Bandage detects the type of the string (the constant-sized character array) and sets the bound equal to the base plus the size.

### 4.5.2 Setting to the address

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

For this situation a more general, run-time method is used to detemine the size of the r-value. A GEP is created to the type of the r-value, with its first operand being a null pointer (so the address calculation starts at the offset zero), and with its other operand specifying the second object of that type.

This causes a calcuation, starting from offset zero, calculating the address of the second object of that type in a non-existant array. This will return the

size of the r-value. As before, on non-ARM processors the GEP will be be reduced to a constant in further stages of compilation.

### 4.5.3 Setting to another pointer

Setting a pointer equal to another pointer is simple, the bounds information of the r-value is set to that of the l-value, as is the value of the pointer. Occasionally, pointer casts may be need to be inserted for typing (as the bounds are typed equal to the pointer, so if the pointer is cast, so must the bounds).

### 4.5.4 Setting to pointer arithmetic

The bounds information is modified as above, though this time the value of the r-value fat pointer has arithmetic performed on it (in the form of GEP instructions) before being stored in the value of the l-value fat pointer.

At this point, no bounds checks are performed, it is perfectly legal for a pointer to have arithmetic performed on it that leads it to go out of bounds and later to have further arithmetic that brings it back into bounds.

### 4.5.5 Setting to NULL

Finally, pointers are frequently set to be equal to the null constant. In this case, the bounds information is not changed, only the value. The reason for this being that a null pointer is flagged by bounds checking before the bounds information is touched and therefore two additional assignments would add unessecary overhead.

## 4.6 Binary Compatibility

There are three different entities that Bandage must pay attention to in regards of binary compatibility. These are functions, structs and external variables.

For functions, Bandage follows a rule of "if I can see its definition, I am allowed to modify it". For any function definition it creates a transformed duplicate while retaining the original. This allows a transformed library to be used with untransformed code as the original functions remain. It would be a useful extra step to modify the untransformed version of the function to be a wrapper to the transformed version, inserting null bounds and allowing it to be internally bounds checked, however the current method means that the performance overheads of bounds checking are not imposed on programs that were not created with bounds checking in mind.

For any functions that Bandage cannot see the definition of, it leaves untouched, unless that function is named in the function file explained earlier.

Structs are a bit different, since they are fully defined in every source file they are used in. To determine which structs it is allowed to modify, Bandage looks at the functions again. If a struct is used in a function that is defined, but not declared (taking into account the functions in the function file), it is counted as external and not transformed.

This can lead to a case where the same struct is used internally to a project and also in the libraries included in that project but the struct never crosses the internal/ external boundary. In this case, two versions of the struct will exist in the final binary, the transformed one used internally and the raw one used externally.

Finally, extern variables must be taken into account. External variables are transformed in a deterministic manner, allowing the linker to link the transformed versions. Bandage's fat pointer implementation does not allow the mixing of transformed and untransformed external variables, but its lookup table implementation does, however such a mix is probably indicative of bad

coding practices.

## 4.7 CCured Analysis

In the CCured paper, pointers are classified into three qualifiers: *SAFE*, *SEQ* and *DYN*. Pointers are represented by a tuple, containing the original allocation of the pointer (the `alloca` instruction) and the number of dereferences applied, this way in the analysis, a multi-level pointer is split up so that each level can be classified differently.

### 4.7.1 Constraint Collection

The CCured-like analysis implemented contains four different constraints:

- *PointerArithmetic*, which applies to one pointer.
- *SetToPointer*, which contains the l-value and r-value pointer.
- *SetToFunction*, which contains the l-value pointer and the function.
- *IsDynamic*, which applies to one pointer.

The instructions are iterated through, collecting all pointer allocations. Each function is examined and if the return value is a pointer, the pointer returned is associated with the function. Similarly the function's parameters are associated with the function and their position in the parameter list.

Every store to a pointer is then examined, with the following actions:

- If the r-value is a pointer, a *SetToPointer* constraint is created.
- If the r-value is a function, a *SetToFunction* constraint is created.
- If the r-value is a cast from an integer, a *IsDynamic* constraint is created.

- If the r-value is a constant that isn't null, a *IsDynamic* constraint is created.

Every pointer that has a `GEP` performed on it (that isn't a struct type) has the *PointerArithmetic* constrain created.

### 4.7.2 Constraint Solving

The constraints used in the CCured-like analysis are quite simple and can be solved with a linear process. A map of pointers (the allocation and dereference level pair) to types is created and the following sequence is followed:

1. All pointers that have the *IsDynamic* constraint are set to the *DYN* qualifier.
2. All *SetToFunction* constraints are transformed to *SetToPointer* constraints with the r-value being set to the pointer associated with that function.
3. All *SetToPointer* constraints where the r-value and the l-value are of different types have both l-value and r-value set to *DYN* qualifier (this constraint is specified in the CCured-paper, though it seems a bit strong to my, surely only the l-value need be set to *DYN*).
4. All pointers with the *IsArithmetic* constraint are set to the *SEQ* qualifier.
5. The following is repeated until no further changes are made:
   (a) For each *SetToPointer* constraint:
      - If either l-value or r-value are *DYN*, set both to *DYN*.
      - If the l-value is *SEQ* and the r-value is *SAFE*, set the r-value to *SEQ*
   (b) For each pointer (allocation and level pair) that is *DYN*, set all pointers from the same allocation with a greater level to also be

*DYN*.

**I should go over the CCured-analysis part in the Literature Review section and clearly state the constraints. Then when detailing this, I can refer to which constraints which part of the algorithm are fulfilling**

## 4.8 Arrays

C provides no safety for constant size arrays. For completeness, both the Bandage and SoftBound passes implement bounds checking on array access.

This is made almost trivial because the type system carries the bounds of arrays, and therefore in the transformation pass the array access can be modified by looking at the type of the pointer operand to the GEP instruction (used to calculate the address of the requested array entry).

# Chapter 5

# Evaluation

## 5.1 Performance

### 5.1.1 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.**

### Following a Linked List

This benchmark was created to highlight the difference between the fat pointer and the lookup table approach, since with SoftBound no table lookup occurs for local variables. A linked chain is created and then followed. These tests were repeated with a linked lists of different lengths to investigate how each approach scales with the number of pointers that it needs to keep track of.

Figure 5.1 shows the results. The first and most obvious result is that, now that the pointers are stored on the heap and therefore table lookups are

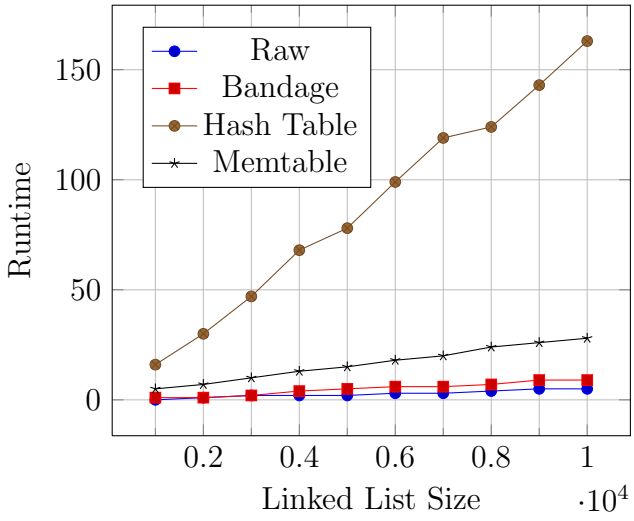



Figure 5.1: Increase in runtime following a Linked List as size increases

required for pointer bounds the table lookup approaches are slower than the fat pointer approach.

The hash table performs the worst with a `32x` runtime increase at the largest list size. This is not surprising considering that each lookup requires a look through each bucket for the matching element. The interesting result is that the runtime seems to increase linearly with the list size, implying a close to $O(1)$ overhead for hashtable lookup.

With the longest linked list length, the MemTable takes five times as long as with no bounds checking and three times longer than bandage. The MemTable lookup consists of pointer arithmetic and an access to the `mmap`ed area, whereas a fat pointer lookup consists of a `GEP` to the object. A potential reason the MemTable approach takes longer than fat pointers is that an iteration for the fat pointer contains a load which fetches both the next pointer value and its bound at the same time, and they are store contiguously in memory. An iteration with MemTables consists of a lookup to find the next pointer, and then a lookup in the table to find the bounds for that pointer, requiring two lookups to two areas of memory that are likely very far apart.

However, since the linked list was allocated in order, each element is likely

arranged sequentially in memory. Therefore, since the MemTable uses the pointer's address in memory as the index to that pointer's information, the bounds information associated with each linked list item will also be arranged sequentially in memory, and quite close together, giving very good spatial locality for the caches to take advantage of.

A final cause of the slowdown could be that since the table lookup functions are compiled separately and linked with the code that uses them it prevents them from being inlined, resulting in more jumping around. linear

### 5.1.2 Olden Benchmarks

The olden suite of benchmarks are designed to be very pointer operation heavy.

#### Treeadd

The treeadd benchmark constructs a binary tree where each node contains a value in addition to two children (all values are set to 1). A depth-first search is then performed, accumulating the value at each node.

The implementation of CCured-like analysis counts all member pointers as a non-SAFE type (since the actions on the pointer and therefore the CCured type will be different for each instance), meaning that the tree traversal doesn't benefit from CCured-analysis.

Under Bandage, the tree construction stage, dominated by memory allocations took a 42% performance hit and the tree traversal stage took a 77% performance hit.

#### Bisort

The bisort benchmark constructs a binary tree with each node containing a random value. The tree is then sorted by performing a binary merge at each

node, working up to the root node.

Under Bandage, tree construction displayed little overhead with a 3% slowdown, though the sorting caused a larger overhead of 165%, resulting in an overall runtime increase of 83%.

### Mst

### Perimeter

-8% 23% -3%

### Power

### Tsp

The tsp benchmark constructs a 2d tree of nodes and proceeds to solve the travelling salesman problem. For nodes close together, it uses the closest pointer heuristic.

Under Bandage, tree construction introduced no overhead but application of the travelling salesman problem increased runtime by 103%.

### Other Benchmarks

Due to the complex nature of the transformation (especially the fat pointer transformation), some of the olden benchmarks are not transformed correctly.

One of reasons that the benchmarks fail to run is Type Coercion. In some circumstances, the LLVM IR produced by clang coerces the types of parameters to functions. For example, the em3d benchmark contains the following function:

```
typedef struct node_t{
    double value;
```

```
} node_t;

typedef struct graph_t{
    node_t *e;
    node_t *h;
} graph_t;

void print_graph(graph_t graph){...}
```

Instead of the struct being passed to the function as a whole struct, it is split into its two members and the function in IR actually takes two `node_t *`s as its parameters, which are then put into a anonymous type of `{node_t *, node_t *}` which is finally bitcast into a `graph_t`.

### 5.1.3 No Checks

Bandage was modified to not insert bounds or null checks on pointer dereference, and the benchmarks were run again. This provided a measurement of the overhead introduced by using fat pointers (and therefore dealing with the extra loads, pointer wrapping and stripping and cache overhead) and by propegating the bounds information. The overheads introduced here allow evaluation of the bounds checking strategy separate from those overheads and also provides a theoretical lower bound on what an optimization to the checking strategy can achieve.

The runtime overheads are shown in Table 5.1, for comparison when the code was run without optimization and when the code was run with O3. It is gratifying to see that in most of the cases, O3 optimization reduces the gap between uninstrumented and instrumented code.

| Olden Benchmark | O0 Overhead | O3 Overhead |
|---|---|---|
| Bisort | 7.50% | 3.44% |
| Mst | xxxxxx% | xxxxxx% |
| Perimeter | -18.62% | -43.49% |
| Power | xxxxxx% | xxxxxx% |
| Treeadd | 28.75% | 17.11% |
| Tsp | 7.29% | 28.57% |

Table 5.1: Runtime overhead on olden benchmarks when no bounds or null checks are inserted

| Optimization Level | Raw | Bandage |
|---|---|---|
| O1 | 20.51% | -2.33% |
| O2 | 26.92% | 13.95% |
| O3 | 26.92% | 13.95% |

Table 5.2: Speedup relative to O0 optimization for raw and bandage transformed Tsp benchmark.

### Tsp interfering with Optimization

Looking at the runtime breakdown of the Tsp benchmark gives us the timings in Table 5.2. It can be seen that large discrepancy in optimization speedups arises during the set of O1 optimizations.

In order to investigate this further, O1 optimization was repeated multiple times on the uninstrumented code, each time with a specific optimization disabled, to determine which was responsible for the 20% speedup. Each of the resulting binaries displayed speedups of 17% or more, suggesting that no single optimization was responsible for the speedup. It was found however, when the `instcombine` optimization was omitted, the speedup for the unoptimized binary was 25%, indicating that it actually slows the resulting program.

The same process was repeated on the instrumented IR. It was found that with each optimization pass specified manually, a speedup of 25\% is achieved - similar to that of the raw binary, but when specified through the O1 flag, the speedup is negligable.

However, this process also identified the O1 optimizations that were the most

responsible for was responsible for the speedup. Without the `instcombine` or the `sroa` optimizations, the speedup dropped to `1%`. Additionally, the absence of the `early-cse` optimization caused the speedup to drop to `17%`.

## 5.2 Interactions with Optimizations passes

It was seen in the previous section that there were three O1 optimizations that produced significant changes in the runtime of the instrumented code.

## 5.3 Security

### 5.3.1 Buffer Overflow Attack

```
int main(void){
    char pass[16];
    int userid = 0;

    gets(pass);
    ...
}
```

```
// In stdio
char *gets(char *buf)
{
    int c;
    char *s;
    for (s = buf; (c = getchar()) != '\n';)
        if (c == EOF)
            if (s == buf)
                return (NULL);
            else
```

```
                break;
        else
            *s++ = c;
    *s = 0;
    return (buf);
}
```

Bandage is capable of stopping the buffer overflow described in the Background section. The `pass` array has its bounds information contained in its type in LLVM IR (as `[16xi8]`). As it is passed into `gets` as a parameter it is converted into a function whose value is set to the address of the array. During the assignment, the bounds of the fat pointer are set correctly. Then as `s` is set to `buf`, the bounds information is transferred to `s` and consulted on every dereference of `s`. Therefore once the value of `s` exceeds its base and bounds (that of the original variable `pass`) an error is triggered.

### 5.3.2 Heartbleed

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
...
unsigned char *buffer, *bp;
int r;

buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
...
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
```

```
memcpy(bp, pl, payload);
```

### 5.3.3 Returning a Local Variable

```
int *ReturnLocalVariable(){
    int x = 5;
    return &x;
}
int main(){
    int *a = ReturnLocalVariable();
    *a = 12;
    return 0;
}
```

Bandage, a system for enforcing spatial pointer safety cannot detect what is wrong with this example of a temporal pointer violation. The expression `&x` is turned into a fat pointer, with value and base pointing to the address of `x` and the bound pointing past its end. After return, it is still a perfectly valid fat pointer as the value is within the base and bound. Therefore on dereference, no error is triggered.

# Chapter 6

# Summary and Conclusions

## 6.1 Software Engineering Practices

The use of a strong test suite was invaluable during the implementation, and the only thing I would change if I did this again would be making it even stronger.

In terms of implementation costs, implementing Bandage took considerably more effort and was far more bug prone than implementing SoftBound. This is in large part due to Bandage changing the representation of pointers - making the change from raw pointers to fat pointers required a considerable amount of code to be written in one piece - if the entire change did not work, the source programs would break. In the SoftBound implementation however, the implementation could be broken down into small pieces (eg providing bounds for local variables, then passing them through functions), and at each stage the source programs would compile, though the checking may not be complete.

Although this does provide a nice guarantee with Bandage that if the program compiles, bounds checking is more likely to be thorough, it creates an all-or-nothing situation in the presence of cases that haven't been implemented where Bandage either breaks the code or performs well, while Softbound can

provide bounds checking for that that has been implemented.

Initially, only the Bandage system was to be implemented. Therefore keeping the Bandage transformation pass separate from the CCured-like analysis pass proved beneficial as it could be easily reused to provide the same information to the SoftBound transformation.

Unexpected setbacks were discovered in trying to run the Softbound transformation on the Raspberry Pi, as both uthash and memtables required modification to run on 32-bit systems.

# Bibliography


[1] ISO. INTERNATIONAL STANDARD, Programming Languages - C, 2011.

[2] Llvm address sanitization doc. *http://clang.llvm.org/docs/AddressSanitizer.html*.

[3] Llvm address sanitizer algorithm. *https://code.google.com/p/address-sanitizer/wiki/AddressSanitizerAlgorithm*.

[4] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. *ACM Sigplan Notices*, 43(3):103–114, 2008.

[5] Richard WM Jones and Paul HJ Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, pages 13–26, 1997.

[6] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[7] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[8] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notices*, 29(6):290–301, 1994.

[9] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[10] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the USENIX Security Symposium*, 2012.

[11] Rithin Kumar Shetty. Heapmon: a low overhead, automatic, and programmable memory bug detector. 2005.

[12] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *ACM Sigplan Notices*, volume 44, pages 245–258. ACM, 2009.