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

June 3, 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: 11,853

Signed:

Date:

Abstract

C is one of the most widely used languages, favoured for its low cost abstractions and the control it gives programmers over memory. This control frequently leads to bugs, especially while using pointers, causing unpredictable errors and security vulnerabilities. Therefore much research has been done to providing pointer safety to C programs, however current work 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. I use an approach where the analysis and transformation are kept separate, with a single analysis able to support two drastically different transformations (fat pointers and lookup tables), giving performance overheads comparable to current work. I investigate the effects of these two prevailing methods of ensuring pointer safety on commonly used optimizations passes. Finally, I provide methods for maintaining very high binary compatibility with programs that use already compiled libraries, and evaluates the costs of doing so.

Additionally, I create a taxonomy 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 such as the types of safety provided, completeness and compatibility. Finally an argument is made that such techniques fall on a scale between providing 100% safety and 100% compatibility and places current work 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. For example, Ghostscript, an open source PostScript implementation violates the standard by issuing pointers to the `-1` element of stacks [8].

One method of dealing with such bugs 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.

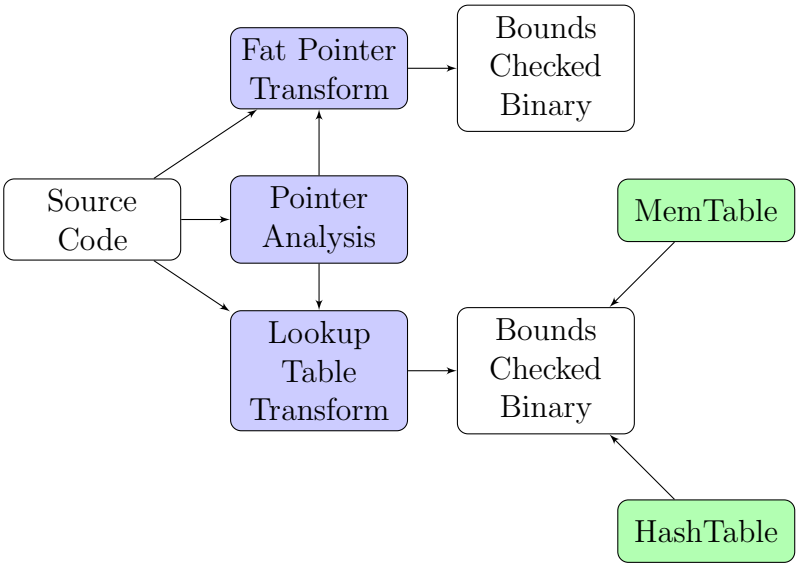


Figure 1.1: Overview of Components of Bandage

I implemented the Bandage system, consisting of the multiple parts shown in Figure 1.1. Implemented for the purposes of this dissertation was the Bandage system, consisting of multiple parts as shown in Figure 1.1. Bandage contains one LLVM analysis pass, two LLVM transformation passes and two C libraries.

The pointer analysis pass uses a CCured-inspired analysis to identify the many pointers not modified using pointer arithmetic during the life of the program. 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. Finally, the two C libraries contain different implementations of the lookup table.

By differentiating the pointer analysis pass from the transformation pass, it 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. Additionally, it allow support

for further backends that were not considered during its design, such as MPX or CHERI.

I use the two different transformation passes 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. This is in contrast to work such as CCured which requires a full recompilation of all included libraries.

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.

1.1 Outline

This dissertation proceeds as follows, with a 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,

the compilation overhead incurred by the analysis and transformations, the effects of the transformation on other optimization passes and the framework for evaluating future pointer-safety systems. 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. Although this is very powerful, it can also be very dangerous and numerous bugs have arisen from programmers not putting careful checks on their memory operations. These bugs can lead to corrupted data, or potentially a security vulnerability.

2.1.1 Example: Buffer Overflow Attack

This first example displays a pointer checking bug that a malicious user can turn into a security vulnerability, by allowing user input to overwrite a variable it should not be able to [24]. 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 [24].

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

```c
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;
    }

    ...
}
```

```c
// 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 [14]. 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).

The bug is contained in the following code [7]:

```
/* 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 [19]. 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 [9].

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

In this section I will review current work in the areas of pointer bounds checking and memory safety and place it in a spectrum with highly compatible on one side and highly complete on the other. This culminates with Figure 5.5 that maps out the design space.

Most pointer-safety systems fall into one of three categories, based on how they track valid areas of memory. In fat pointer systems, information about valid areas of memory is carried alongside the pointer, whereas in lookup table systems it is held seperate and associated with the pointer in some other method. Finally, some systems use a poison based approach, where they signify areas of memory as being invalid and throw an error when an invalid area of memory is accessed.

3.1 Systems requiring code rewrite

Starting on the highly complete but poorly compatible end of the spectrum is Cyclone [10], which is in fact a separate dialect of C. Cyclone allows the programmer to write in a C-like language that is designed to prevent buffer overflows. It does this by imposing restrictions, such as limiting pointer arithmetic and providing extensions, such as a never-null pointer type. While

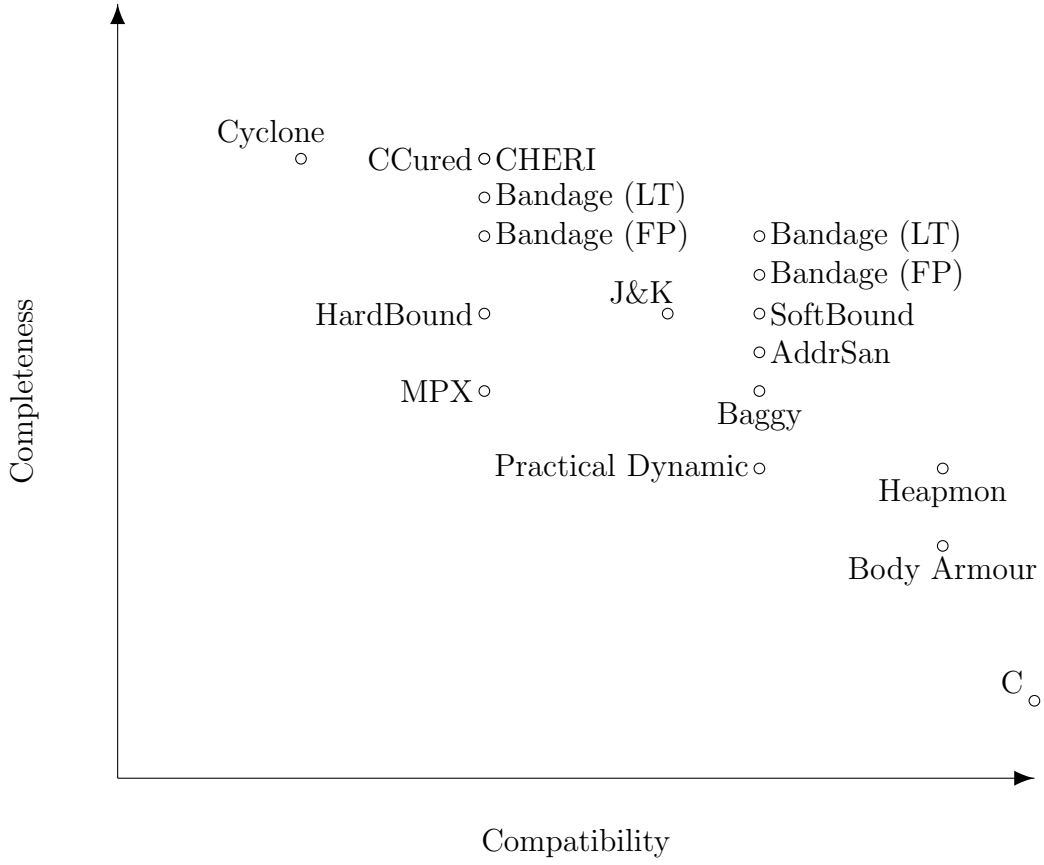


Figure 3.1: The Completeness vs Compatibility Tradeoff in Pointer Safety Systems

some of these extensions are automatic, the programmer must make use of most of them explicitly which required between 0% and 46% of lines of code to be changed in the corpus tested.

Arguably other languages can be included in at this end of the spectrum. For example, Rust [15], which is designed to be memory safe, not permitting null or dangling pointers and placing ownership of memory as a central tenant of the language.

3.2 Systems requiring library recompilation

The next step towards greater compatibility are systems that require the included libraries used by the program to be recompiled. The library recompilation allows such systems to annotate or instrument the libraries, and deal intelligently with pointers returned from and sent to them.

CCured [18] is primarily an analysis that annotates pointers with one of three types: `SAFE` pointers which are only ever assigned to and dereferenced; `SEQ` pointers that have pointer arithmetic performed on them and `DYN` pointers which may have anything done to them including being cast from an integer. Using this analysis, CCured can reduce the number of bounds checks that must be performed, however the analysis must be performed on the entire program, including libraries. Coupled tightly with this analysis is a lookup table implementation of tracking bounds.

The Hardbound [6] proposes hardware pointer checks and introduces the hardware bounded pointer. This is essentially a fat pointer, but the other fields are dealt with transparently by hardware. All registers are upgraded to triples containing a value, base and bound, though the base and bound are set to zero for non-pointers. It is suggested in the paper that Hardbound could be combined with CCured, where hardware bounded pointers are used to keep track of data about `SEQ` and `DYN` pointers.

The Intel Memory Protection Extensions [20, 1] are a set of extensions to the

x86 instruction set designed to check pointer references that may be exploited for security exploits. They require compiler, runtime library and operating system support.

The CHERI system [25], uses a specialized FPGA to track capabilities which are unforgeable pointers that grant access to a linear range of address space. The project uses program knowledge to combine these capabilities with C pointers - the capabilities contain base and length information to that can be used to determine if a pointer is in bounds and uses hardware to perform bounds checks on memory access.

Efficient Detection of all pointer and array access errors [5] 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 variable). 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%.

3.3 Systems requiring program recompilation

The Jones and Kelly bounds checker [11] uses a table based approach, keeping a list of all objects in memory by tracking the uses of `malloc` and `free`. This system holds a unique place in the compatibility to completeness scale, since it works with all programs written in standard compliant C, however an element of its transformation can break a non-standard but commonly used practice.

In standard C, pointers cannot point to an invalid memory address - they

may only point to allocated memory addresses or one past the end of an array. However, most implementations allow pointers to do this, and to maintain their (invalid) value, which may then be transformed to be valid again. For example, the Ghostscript implementation of stacks uses pointers to the `-1` element of an array. The Jones and Kelly system however irrevocably invalidates such pointers. Therefore, the Jones and Kelly bound checker requires the program to be rewritten if it does not adhere to the C standard.

In *A Practical Dynamic Buffer Overflow Detector* [21] 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 [4] 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.

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.

SoftBound [17] 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 Systems requiring no recompilation

Heapmon [22] 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.

Body Armour for Binaries [23] 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.

3.5 Systems implemented in this dissertation

The fat pointer transformation implemented as part of Bandage is not a straight implementation from any research paper, but rather an implementation of the common themes spanning those that take the fat pointer approach. The lookup table approach was more directly inspired by Softbound, using its lookup table structure (though with interchangeable implementations) and optimizations for dealing with local pointers. Both of these transformations were designed with compatibility in mind, requiring a change only in the code for the program, not its libraries.

3.6 Design Space

A summary of the design space is presented in Figure 5.5. At the very bottom end of completeness is the untransformed C language, which offers the highest compatibility but no pointer safety guarantees. Offering the next step up in completeness are systems such as Heapmon and Body Armour for binaries which provide some protection without the need to recompile anything..

The completeness of a system is further increased by giving the system access to the programs source code and the ability to automatically instrument it, giving the myriad of tools forming the central region of the spectrum. As the system becomes more integrated with the developer's toolchain, completeness increases, but the libraries need to be recompiled to work. Finally, with Cyclone, the programmer assists the system in providing the most complete protection.

Chapter 4

Design and Implementation

4.1 Overview

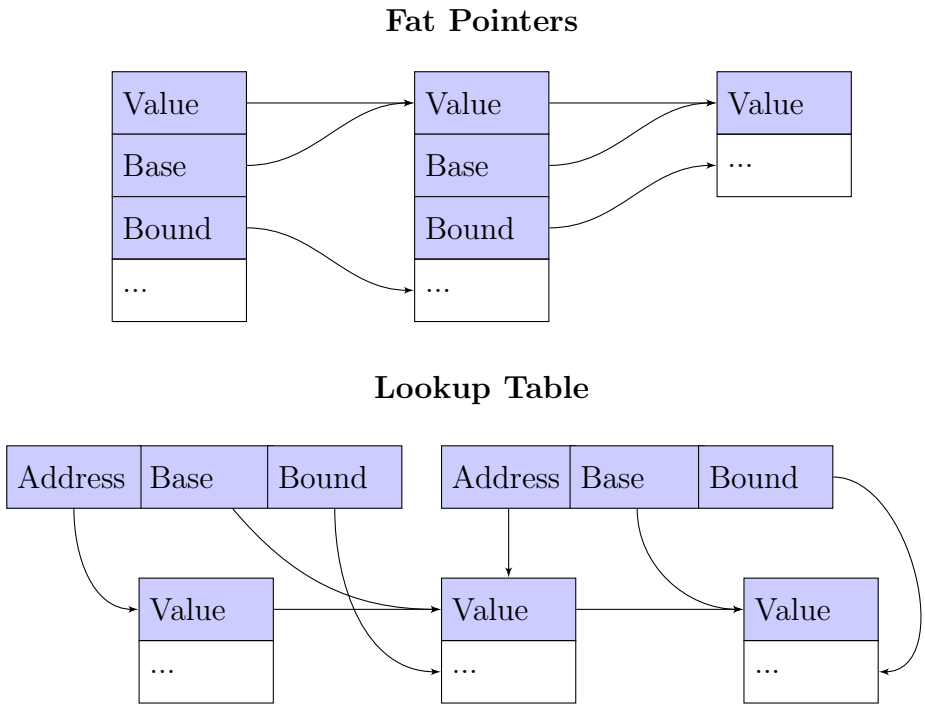


Figure 4.1: Graphical Comparison of Fat Pointer and Lookup Table Methods

This section cover the implementation of the three core sections of Bandage:

the fat pointer and lookup table transformations and the pointer analysis. The two different transformations 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.

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*.

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.

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.

HasBounds 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 Transformation

Allocation

I start the transformation by finding the `alloca` instructions that create a pointer, and replacing them with a fat pointer. Even at this early stage I add some safety to the program by initialising the fat pointer value to be null (a transformation that does not violate the C standard). 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.

Instruction-based vs Chain-based Transformation

My original implementation of the fat-pointer transformation moved through the program and operated on an instruction-by-instruction basis, gathering all information for transforming an instruction from that instruction. I found that this approach, though originally simple rapidly gained complexity as the transformation matured. Upon integration with the pointer analysis, I switched to a different, more holistic approach, that of use-def chains.

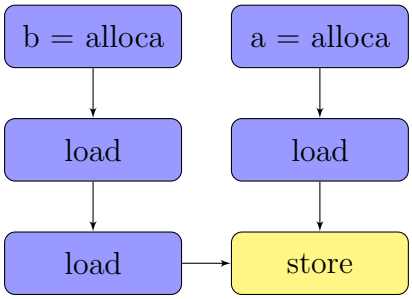


Figure 4.2: Simple example of a use-def chain.

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

These instruction chains terminate in one of a few common instructions. Store and compare instructions terminate the chains leading to both of their operands, the return instruction terminates the chain of its operand and a function call terminates the chain of all of its parameters and initiates one for its return value.

Figure 4.2 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 fat-pointer capable 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, an error is reported.

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.

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 of the function exists, conversion code is inserted 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.

Multiple Source Files

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 declaration but no definition, 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, I created an additional pass, the **Function List** pass. It takes a single LLVM IR file and a filename and outputs all of the functions defined in that IR file.

Now, the process for using Bandage (with either transform) is as follows:

- Run the **Function List** pass over all source files, appending to the same function list file.

- Run the **Transformation** 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 declarations for functions the programmer has no control over (so they can be ignored) and function declarations for functions Bandage can modify.

Bandage transforms all function declarations that match functions in the function list file. This declaration will be transformed identically to the function definition in the file it is defined in so the linker will be able to merge them together after the transformation is complete.

4.2.4 Structs

First, I isolate the structs that can 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` (from `stdio.h`) 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, used and have its body filled in later. Therefore the algorithm to create the fat pointer types consists of:

- Collect a set of all structs.
- Subtract those structs that are used in functions that are declared and not in the function list.
- 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 layer 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 linear order without having to worry about which structs reference other structs.

Sizeof

The clang compiler replaces instances of the `sizeof` operator with constant integers, so it is not present in LLVM IR. The fat pointer versions of structs will always be equal to or larger in size than their raw counter parts. This leads to the undesirable situation where the source program allocates area of memory for 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. Currently the parameter to `sizeof` must be changed manually, but it would be possible to modify clang to add annotations making it possible for Bandage to do this automatically.

4.3 Lookup Table

For comparison with the fat pointer approach, I implemented a lookup table based on the SoftBound system.

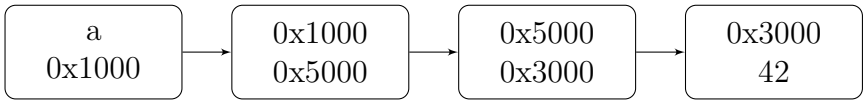
4.3.1 Pointers on the Stack

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 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.

I iterated through all instructions in the program and on finding an `alloca` of a pointer, two new pointers of the same type are created. I store references to them in a map indexed by a reference to the original such that further uses of the original pointer can be used to find its associated variables.



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

Figure 4.3: Example of pointer chain with associated lookup table entries.

4.3.2 Pointers on the Heap

The real difference between the fat pointer and lookup table transformation comes when dealing with pointers on the heap. Here, the base and bound are store in the lookup table, indexed by the address of the pointer. Figure 4.3 shows an example 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, and the lookup table is displayed below.

The local variable `a` points to the address `0x1000`, which points to the address `0x5000` which points to the address `0x3000` which contains `42`.

Consider the following code:

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

The first dereference is simple and requires no lookup as `a` is a local variable and therefore has local variables associated with it to hold 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.

4.3.3 The Table

This table lookup is performed whenever a pointer stored on the heap is dereferenced, and the runtime overhead of this is the major disadvantage of the lookup table approach. I created the lookup table transform to have interchangeable table implementations, allowing for future expansion. 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. Two different implementations of lookup table were implemented to compare the trdae-off between size and lookup cost.

Hash Table

I implemented a lookup table 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. It 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, 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.

Therefore, of the 2^{47} available addresses the program can have, 2^{45} is still used for the program, whereas 3×2^{45} are used for the memtable. The 3×2^{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 3×2^{45}.
- Divide the address by 8 (the entry coverage).
- Multiply the address by 16 (the entry size).
- Add the table start offset 2^{44}.

My primary aim of this scheme was to keep implementation simple and to provide a comparison to investigate runtime overhead. Multiple improvements could be made. First, the table could be shrunk to 2/3 of the available space, not 3/4. Second, a split of 50:50 between the text and stack and heap segments of memory may not be ideal.

I considered some sort of compression for the bounds information, 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 adding greater runtime overhead.

Memtable for 32 bit architectures

I had to create a separate version of the memtable for use on 32-bit architectures. The above implementation assumes a 64-bit architecture. The top 2^{30} addresses are reserved for use by the kernel, leaving 3×2^{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.

4.3.4 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, I modify locally declared functions and those in the function list so that their base and bound are passed in explicitly as additional parameters. Likewise, 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.

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);
```

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, whereas 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 this approach, 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.

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

4.4 Setting pointer bounds

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.

Constant Strings Constant strings are stored in the file text section of the LLVM IR file and have their length encoded in their type. The base and value of the pointer are set to the address of the constant string and the bound is set to the address plus the length of the string.

Address-Of Operator As with setting to constant strings, the base and value are set to the address of the variable, and the type information for the

variable is used to determine the size and therefore the bound.

Pointer To Pointer Assignment On pointer to pointer assignment, the base and bounds of the r-value are transferred to the l-value along with 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).

Pointer Arithmetic The results of pointer arithmetic are derived from one pointer and one integer type. The base and bound of the pointer type are carried from the pointer r-value to the l-value, while the value of the pointer is set to the result of the arithmetic. 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.

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 unnecessary overhead.

4.5 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.

Functions 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 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.

Extern Variables 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.

4.6 Pointer Analysis

My pointer analysis is based on the CCured analysis. In CCured, pointers are classified into three qualifiers: *SAFE* where pointers are only ever assigned

and dereferenced, *SEQ* where pointers have arithmetic performed on them and *DYN* for any other pointers.

4.6.1 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.

Pointer stores are examined, generating the *SetToPointer*, *SetToFunction* and *IsDynamic* constraints. The *IsDynamic* constraint is used when a pointer is cast from an integer or a constant that is not null. Every pointer that has a `GEP` performed on it (that isn't a struct type) has the *PointerArithmetic* constrain created.

4.6.2 Constraint Solving

The constraints used in this analysis are quite simple and can be solved with a linear process.

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 pointers with the *IsArithmetic* constraint are set to the *SEQ* qualifier.

4. The following is repeated until no further changes are made:

   (a) For each *SetToPointer* constraint:

   - If the r-value is *DYN*, set the l-value to *DYN* since the l-value could transitively be set to anything the *DYN* r-value is set to.
   - If the l-value is *DYN*, set the r-value to *DYN* so the l-value can gain bounds information from the r-value.
   - If the l-value is *SEQ* and the r-value is *SAFE*, set the r-value to *SEQ* so the l-value can gain bounds information from the r-value.

4.7 Arrays

C provides no safety for constant size arrays. For completeness, both transformations implement bounds checking on array access. This is made trivial because the type system carries the bounds of arrays and the variable always points to the base. Therefore in the transformation passes the array access can be modified by looking at the type of the pointer operand to the GEP instruction, giving the type of the array and the index operand, giving the requested offset.

Chapter 5

Evaluation

This section starts with the novel framework for evaluating pointer-safety systems. I draw up a list of criteria upon which they can be judged and comment on the patterns across the related work.

In order to place Bandage within the framework, I evaluated the resulting binaries from the tranformations for performance and binary size, showing them to be comparable with the current state of the art. I then benchmarked the transformations themselves to show it runs in a reasonable time and would be capable of scaling to real life projects.

In order to investigate the interactions with optimization passes, I performed further benchmarks in the presence of LLVM's O1 optimizations. Finally, in order to show the security benefit that can be gained from the additional robustness to pointer-based errors that Bandage can provide, I reiterate through the examples from the Background section, commenting on Bandage's successes and short-falls.

5.1 Pointer Safety System Evaluation Framework

The following factors are taken into account when assessing a pointer safety system:

5.1.1 Spatial Safety

Invalid Memory Accesses are important to detect as they are never intentional, definitely bugs are potential security vulnerabilities. Some systems, such as Baggy Bounds checking fail to fulfill this criterion because they only aim to prevent dangerous invalid memory accesses (such as those to a different object).

Incorrect Object Accesses are when a pointer current points to an object that it was not derived from. The Heapmon and Address Sanitizer systems fail to meet this criterion because they only track valid and invalid memory, and do not map between pointers and the memory that is valid for that pointer. Address Sanitizer does use techniques to make the likelihood of this happening very small (by adding poisoned areas around objects).

Checking all variables can be achieved through one of two methods, either by whole program analysis which lowers compatibility or by tracking calls to malloc and free in the compiled binary. The HardBound, SoftBound and Bandage systems fail to provide this because they do not require recompilation of included libraries, and therefore do not have bounds information for pointers from them, unless the entire program is recompiled. Baggy Bounds checking only checks the bounds of strings as an optimization strategy.

5.1.2 Temporal Safety

Free'd Memory Accesses prevents dangling pointers and double frees. I implemented a very simple case of this in the Bandage system, enough to catch simple bugs, however it is not complete, because when a copy of a pointer is freed, the original is not marked as invalid. It is implemented in Cyclone through restrictions on the language and checks at compile time. It is trivial to implement in systems that track valid memory areas, but not their associated pointers, such as Heapmon and Address Sanitizer.

Dead Stack Memory Accesses are a similar case to that above, except the memory is released when the function returns instead of through an explicit function call. As an aspect of temporal pointer safety, it is outside the scope of bandage.

5.1.3 Binary Compatibility

Program Recompilation is required by systems that need to add hooks to the code or change its memory layout. Most systems require program recompilation and there is a limit to the protection that can be achieved without seeing what is going on inside the program. Body Armor for Binaries gets around this reverse engineering the program file to try to reconstruct symbol tables and such.

Whole Program Recompilation requires the program and all of its included libraries to be compiled with the system. This is linked with the **Checking all variables** criteria as a key trade-off between completeness and compatibility, especially with systems that can be run without whole program recompilation, but provide far less security when they do so.

Non-conforming Code Rewrite is required by the Jones and Kelly system which overwrites the results of pointer arithmetic that exceeds the ref-

Safety Guarantee	Bandage	CCured	SoftBound	HardBound	Jones & Kelly	Cyclone	Heapmon	Address San.	Baggy Bounds	MPX
Access to invalid memory							✗		✗	
Access to incorrect object							✗	✗		✗
Checks all variables	✗		✗	✗	✗				✗	✗
Access to free'd memory	*	✗	✗	✗						✗
Access to dead stack memory	✗	✗	✗	✗			✗	✗		✗
Code recompilation	✗	✗	✗	✗	✗	✗		✗	✗	✗
Library recompilation		✗		✗		✗				✗
Code must be standardized					✗					
Code rewrite						✗				

Table 5.1: Safety guarantees for evaluated pointer safety systems. A cross indicates the guarantee is **not** provided.

erent to the constant `-2`. This would be a valid transformation for programs that adhered strictly to the C standard, but it was found that 60% of programs tested relied on undefined behaviour, that this broke [21].

Code Rewrite is the poorest level of compatibility as all programs will require some form of modification to get the system to work with them. Cyclone requires this, due to it being a dialect of C and not a system to be run on C itself.

The evaluated systems are marked according to the previous criteria in Table 5.1.

5.1.4 Runtime Overhead

Runtime Overhead is the current leading method for comparing pointer-safety systems, probably due to its quantitative and easy to measure nature. It is important, and therefore I've included it, however as can be seen in Table 5.2, current systems perform comparably.

5.1.5 Extra Requirements

Hardware Support may be required by the system. This likely signifies that the system will not be very compatible - a full program analysis is likely to be required for full coverage. Additionally the cost of the new hardware must be factored in, making it unrealistic for projects such as those designed for widespread use on personal computers.

Additional Threads could be an issue if the target program is desired to be run on less powerful systems. An additional thread running may require more resources that aren't made explicit in the benchmarks.

I have summarized the extra requirements and runtime overheads imposed by each tool in Table 5.2.

Together these three graphs provide a full picture of the relative strengths and weaknesses of these tools, allowing programmers to make an informed decision over which is the best for the task at hand. A programmer with a non-real time legacy system could start by choosing Heapmon and accept the runtime overhead as an acceptable price for the lack of compilation time overhead. Alternatively a programmer looking to write a secure system from scratch would be well advised to use Cyclone (provided the garbage collector isn't an issue).

5.2 Transformed Code Performance

The following benchmarks were carried out on an 8 core, 64bit, 3.6GHz Intel Xeon CPU, running FreeBSD 10.0. In addition to the benchmarks, Bandage was developed using a suite of 54 tests, including a few complex tests along with unit tests for:

- Array operations
- Pointer operations

System	Runtime Overhead	Extra Requirements
Bandage	Up to 250%	
CCured	Up to 250%	
SoftBound	Up to 350%	
HardBound	Up to 25%	Hardware
Jones & Kelly	Not Reported	
Cyclone	Up to 250%	Garbage Collector
HeapMon	Up to 450%	Extra Thread
Address Sanitizer	Typically 200%	
Baggy Bounds	Up to 230%	
MPX	None Available	Hardware

Table 5.2: Runtime and extra requirements of evaluated pointer safety systems

- Function boundary transitions
- Compatibility with C libraries
- Correct identification of pointers for the analysis pass.

5.2.1 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 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 pointer 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.

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.

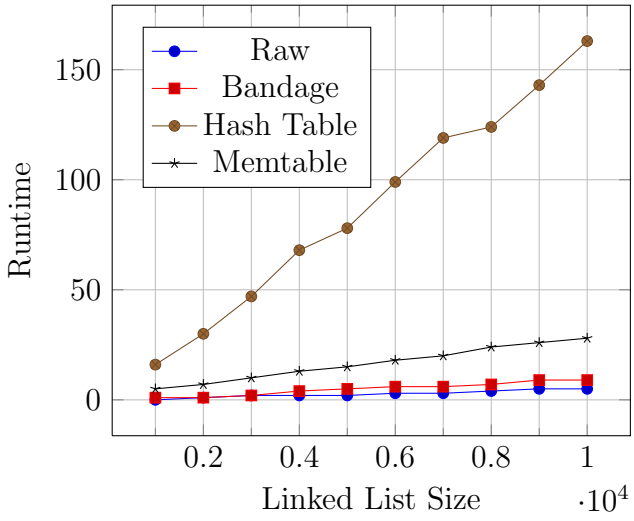


Figure 5.1: Increase in runtime following a Linked List as size increases

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.

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 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 an immediate offset and a load. 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.

Benchmark	Fat Pointer Overhead	Lookup Table Overhead
Bisort	$71\% \pm 1\%$	$164\% \pm 1\%$
Perimeter	$35\% \pm 7\%$	$294\% \pm 11\%$
Power	$46\% \pm 1\%$	$101\% \pm 1\%$
Treeadd	$49\% \pm 4\%$	$253\% \pm 6\%$
Tsp	$95\% \pm 2\%$	$211\% \pm 3\%$

Table 5.3: Fat pointer and lookup table overheads compared to uninstrumented runtime for the Olden benchmarks

5.2.2 Olden Benchmarks

The olden suite of benchmarks are designed to be very pointer operation heavy [13]. Due to the complex nature of the transformation (especially the fat pointer transformation), some of the olden benchmarks are not transformed correctly, and fail to run (`em3d`, `health`, `bh`, `mst`). This is due to issues with the implementation of Bandage, and not due to limits in the system.

Most of the benchmarks center around constructing binary trees (each node contains pointers to two other nodes, and a value), for example the treeadd benchmark constructs a tree and performs a depth-first search to accumulate all the values and the bisort benchmark performs a binary merge at each node to sort the tree. The results of the benchmarks are shown in Table 5.3 and Figure 5.2. These results show performances comparable to those of the state of the art research papers, with the fat pointer implementation not exceeding 200% of the original runtime in any benchmark. The runtimes for the lookup table approach were measured with the memtable implementation, though this still produced a greater overhead (if in part due to the inability to inline the lookup functions as they were included from a different library).

Failing Benchmarks

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

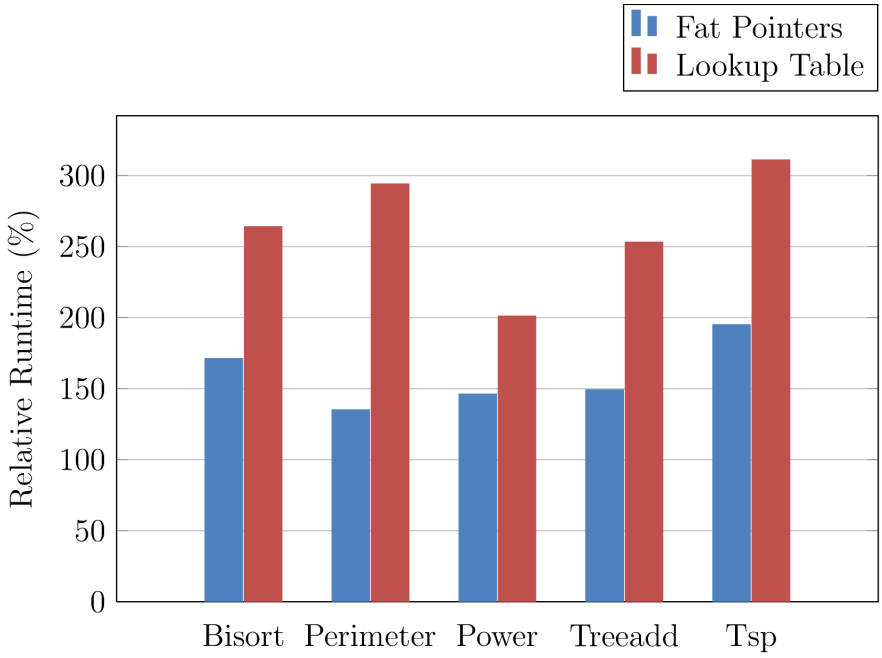


Figure 5.2: Graphical representation of relative runtime for Olden benchmarks

5.3 Compile-time Performance

I timed the compilation for the Olden benchmarks for both transformations. The results, shown in Figure 5.3 show that the fat pointer transformation is usually quicker than the lookup table transformation, but both transformations are always in the same order of magnitude of raw compilation and frequently take less than twice its length.

I measured the resulting binaries, finding that the transformation produced a greater increase in binary size than in compilation time (Figure 5.4). The binary sizes for both of the transformations were both quite similar, indicating that the primary increase in binary size was due to the common operations shared between the transformations, such as calculating bounds and calling the bounds checking functions.

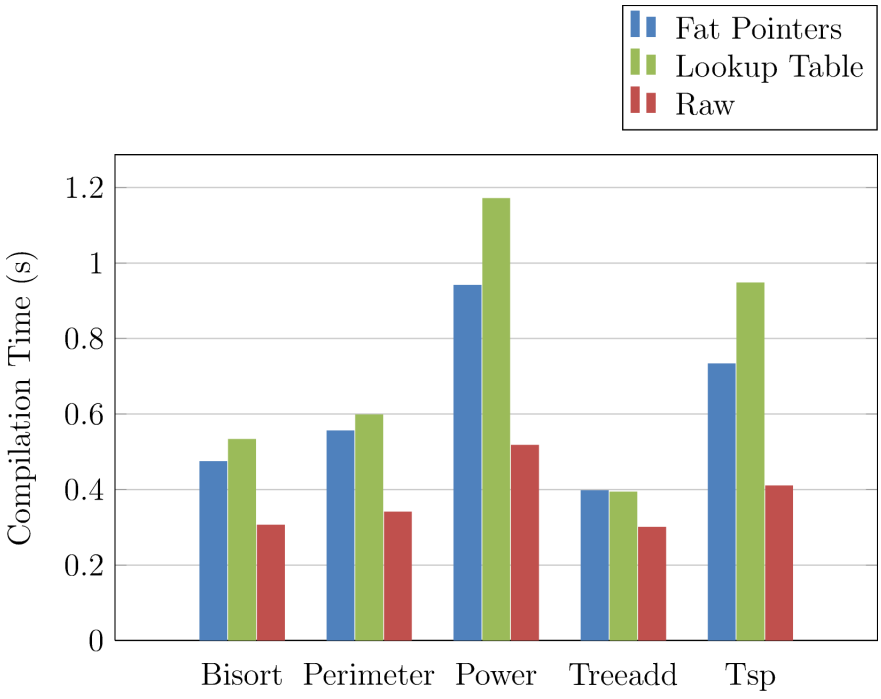


Figure 5.3: Compilation Time for Olden Benchmarks

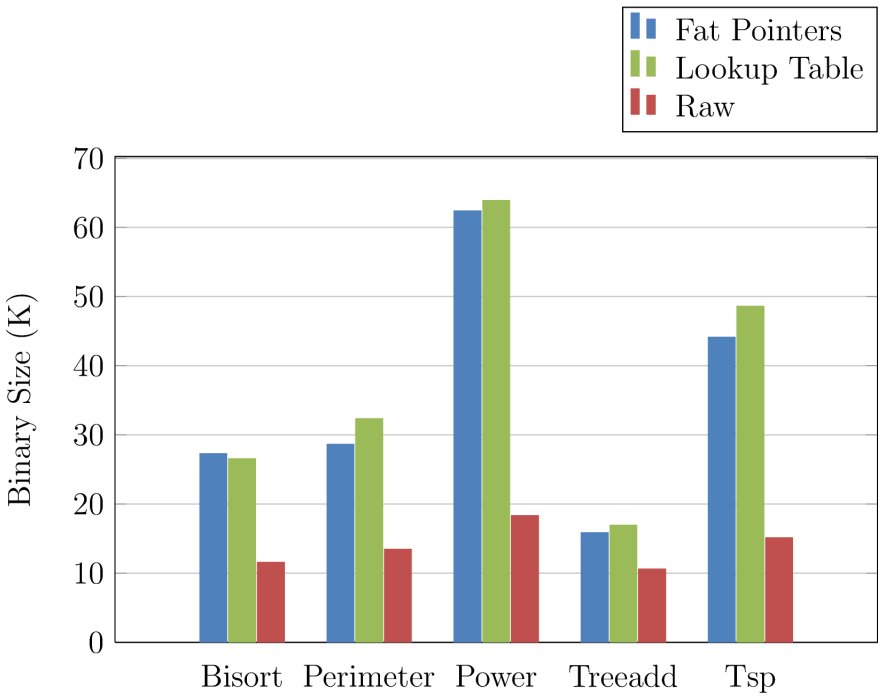


Figure 5.4: Binary Size of Olden Benchmarks

	sroa	loop-rotate	loop-unroll	instcombine
bisort	19.06%	-0.90%	0.34%	5.13%
power	3.01%	18.54%	7.71%	-0.33%
perimeter	17.46%	1.62%	-2.51%	1.26%
treeadd	12.84%	-0.30%	0.93%	-1.37%
tsp	-7.32%	-9.56%	-0.14%	-7.97%

Table 5.4: Performance Differences on the Olden benchmarks with selection optimizations disabled

5.4 Interactions with Optimizations passes

Since many optimizations rely on each other and inter-react, instead of individually enabling optimizations, I individually disabled them. Table 5.4 shows the performance difference when the tsp benchmark (picked as it produced the highest runtime increase itself) was run with the stated optimizations disabled. All of the optimizations used in O1 optimization were run 10 times and only those that produced a difference of greater than 5% are shown in the table.

Since the stated optimizations were omitted, a beneficial optimization would give a negative value in the table - showing that its omission caused the resulting binary to take longer to execute. The table shows that most optimizations give a negative result when applied to the fat pointer code.

5.4.1 Scalar Replacement of Aggregates

Scalar replacement of aggregates is a function based optimization that aims to separate aggregates, such as structs and arrays into simple components which can then be assigned to temporary values. These temporary values can then be the target of further optimizations such as register allocation and copy or constant propegation [16, Chapter 12].

One of the actions of this pass is to replace some member accesses with `extractvalue` and `insertvalue` instructions [12], for example in the follow-

ing code:

```
%1 = alloca %FatPointer
%2 = getelementptr %FatPointer* %Ptr, i32 0, i32 1
store i32* null, i32** %2
%3 = getelementptr %FatPointer* %Ptr, i32 0, i32 2
store i32* null, i32** %3
```

```
%1 = alloca %FatPointer
%2 = insertvalue %FatPointer %Ptr, i32* null, 1
%3 = insertvalue %FatPointer %Ptr, i32* null, 2
```

The `insertvalue` instruction returns a pointer to the updated struct, which is then used in further code. This simplifies the dependency graph, making each subsequent field assignment dependant on the last (so `%3` depends on `%2` which depends on `%1`) instead of all dependant on the `alloca` at the top.

Usually with this pass, the `alloca` instruction at the top is removed entirely, since where originally there was an `alloca`, then a call to `malloc` then an assignment to the allocated variable, there is now only a call. Unfortunately, both of the Bandage transformation passes rely on the `alloca` instructions at the start of each function to gather the variables, so this pass cannot be performed before the transformation. The fat pointer pass transforms variables into their fat pointer versions at this point, and the lookup table pass registers them as local variables and generates their local bounds.

Without this optimization, fat pointers as aggregate objects will be ineligable for many future transformations, which would be very detrimental considering the prevalence of pointers in many C programs. With it, for the purposes of future, function based optimzations a fat pointer can be seen as three separate scalar variables.

5.4.2 Instruction Combination

Instruction combination is a simple worklist driven algorithm that looks for opportunities to simplify the code, for example by replacing multiplications

Olden Benchmark	O0 Overhead	O3 Overhead
Bisort	7.50%	3.44%
Perimeter	-18.62%	-43.49%
Power	19.32%	12.12%
Treeadd	28.75%	17.11%
Tsp	7.29%	28.57%

Table 5.5: Runtime overhead on olden benchmarks when no bounds or null checks are inserted

by powers of two with shift operations. Like with SROA, the primary benefit of this pass is that it allows future optimizations to be more effective.

5.4.3 Overhead due to Cache and Register Churn

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 5.5, 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.

Tsp interfering with Optimization

Looking at the runtime breakdown of the Tsp benchmark gives us the timings in Table 5.6. It can be seen that large discrepancy in optimization speedups arises during the set of O1 optimizations.

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

Table 5.6: Speedup relative to O0 optimization for raw and bandage transformed Tsp benchmark.

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%.

5.5 Conclusion

I have developed a system that by providing multiple and backends for pointer bounds checking and allowing the user to choose how much of their program they want to recompile creates four new points in the design space of pointer-safety systems presented earlier (shown again in Figure 5.5). The lookup table transformation provides slightly more complete safety than the fat pointer transformation, since there are C operations that can strip the bounds information from a fat pointer (for example converting a pointer to

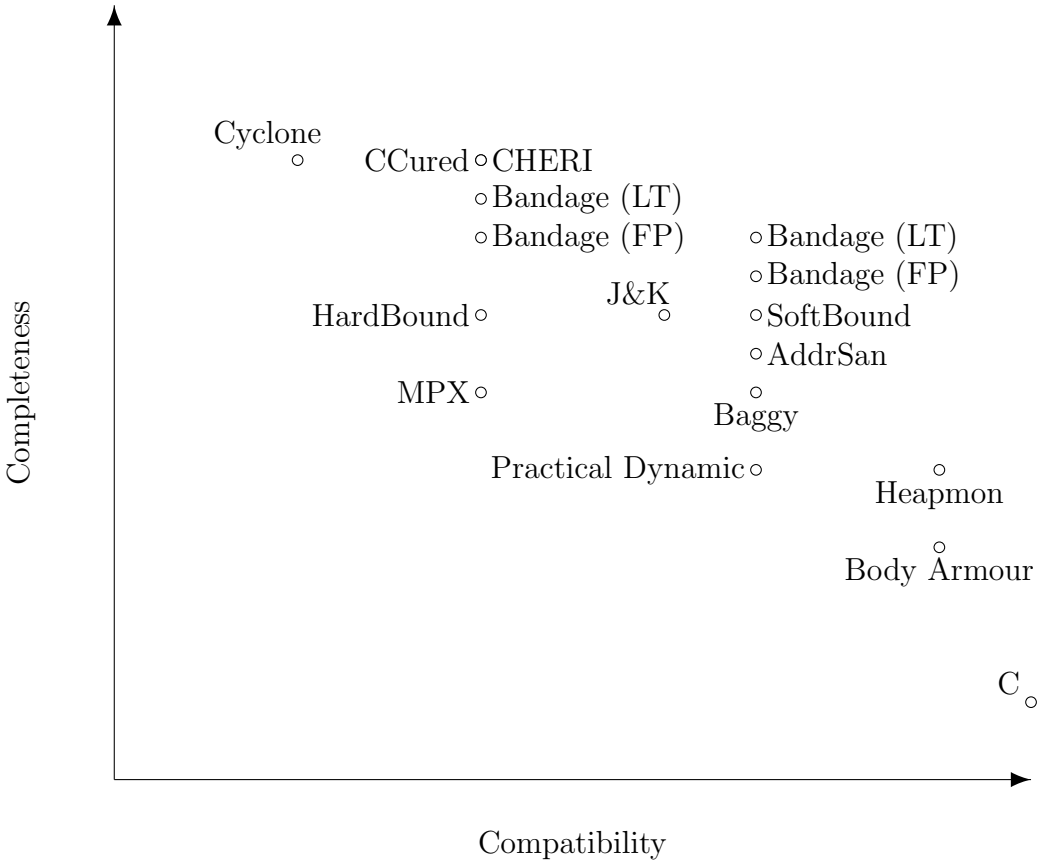


Figure 5.5: The Completeness vs Compatibility Tradeoff in Pointer Safety Systems

an integer and back). Additionally Bandage can be used as a full program analysis, which would provide bounds for pointers received from libraries.

The robustness provided by the Bandage system brings with it security benefits, for example it is capable of stopping the buffer-overflow attack detailed in the Background section:

```
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);
}
```

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.

I used the Olden benchmarks to evaluate the performance overhead incurred by the Bandage transformations. This resulted in overheads comparable to current techniques. However, the Bandage transformation is highly modular, allowing it to be used in future research in the area, with more efficient transformations or more accurate analyses, potentially leading to it being used with lower overhead in the future.

Investigation into further optimizations revealed that the scalar replacement of aggregates pass was highly useful after the fat pointer optimization, as essentially made the value of the fat pointer available to all the optimizations

that a raw pointer could take advantage of.

Chapter 6

Summary and Conclusions

The novel idea of separation of analysis and implementation was developed and implemented paving the way for a reusable ecosystem of components to enforce pointer safety and to identify where it needs to be enforced.

The two prevalent methods of carrying spatial pointer information and tracking its safety were implemented and their effects on commonly used compiler optimizations was investigated. **It was found that...**

By placing the existing pointer safety systems in a common evaluation framework they can be compared and contrasted, allowing the users to make a choice of which system to use or allowing researchers to identify gaps in the design space for future work. This evaluation also identified a core trade-off in this area, that of compatibility vs completeness, with Cyclone taking its place at the completeness end of the spectrum and Heapmon taking its place at the compatibility end.

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.

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] Intel architecture instruction set extensions programming reference. http://download-software.intel.com/sites/default/files/319433-015.pdf.

[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] 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.

[5] 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.

[6] 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.

[7] Eric A. Young, Tim J. Hudson. OpenSSL: Source, Tarballs, 2014. https://www.openssl.org/source/.

[8] GNU Project. Ghostscript Source Code, 2012. ftp://ftp.gnu.org/pub/gnu/ghostscript/.

[9] ISO. INTERNATIONAL STANDARD, Programming Languages - C, 2011. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[10] 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.

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

[12] LLVM API Documentation. SROA.cpp, 2013. `http://llvm.org/docs/doxygen/html/SROA_8cpp_source.html`.

[13] Martin C. Carlisle. Olden benchmark, 1996. `http://www.martincarlisle.com/olden.html`.

[14] Matthew Green. Attack of the week: OpenSSL Heartbleed, 2014. `http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html`.

[15] Mozilla Corporation. The Rust Programming Language, 2013. `http://www.rust-lang.org/`.

[16] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[17] 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.

[18] George C Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[19] Nick Parlante. Pointers and Memory, 2000. `http://cslibrary.stanford.edu/102/PointersAndMemory.pdf`.

[20] RB (Intel). Introduction to Intel Memory Protection Extensions, 2013. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`.

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

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

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

[24] T. Schwarz. Buffer Overflow Attack, 2004. `http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html`.

[25] Jonathan Woodruff, Robert Watson, David Chisnall, Simon Moore, Anderson Jonathan, Davis Brooks, Laurie Ben, Neumann Peter, Norton Robert, and Rot Michael. The CHERI capability model: Revisiting RISC in an age of risk. 2014.