Fast, Compatible, Complete Memory Safety For C Programs

by

Eli Bristol Davis

S.B., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science
at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Eli Bristol Davis, MMXVI. All rights reserved.
The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature redacted

Author … ………………………………………..
Department of Electrical Engineering and Computer Science

Signature redacted

Certified by…. ……………………………………
Martin C. Rinard (Stelios Sidiroglou-Douskos)
Professor
Thesis Supervisor

Signature redacted

Certified by… ……………………………………
Stelios Sidiroglou-Douskos
Research Scientist
Thesis Supervisor

Signature redacted

Accepted by….. ……………………………………
Christopher J. Terman
Chairman, Department Committee on Graduate Theses
Fast, Compatible, Complete Memory Safety For C Programs

by

Eli Bristol Davis

Submitted to the Department of Electrical Engineering and Computer Science on September 1, 2016, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science

Abstract

The lack of memory safety in C/C++ programs is one of, if not the, most persistent and costly sources of program exploits. Attacks based on memory corruption can range from the reading of private data to a complete hostile takeover of a process. While many solutions to this problem have been proposed, it is as of yet unsolved—as old memory corruption attacks are rendered obsolete, new attacks continually spring up.

This lack of success is largely due to the trade-offs that memory safety solutions make between completeness, compatibility, and overhead. There no a single solution with all three properties, and a solution must have all three in order to once-and-for-all solve the lack of memory safety in C programs: If a solution is incomplete, attackers will find a workaround. Unless it is backwards compatible and low-overhead, it will not be deployed in production.

My goal for this thesis was to take an existing system which is close to having all three properties, and add the missing property. I chose to work with SoftboundCETS an LLVM pass which is already complete and backwards compatible, but has high runtime overhead.

In this thesis, I take SoftboundCETS and heavily optimize its runtimes, reducing its total overhead by half. I split the original pass into two separate passes (one to mark which instructions were to be instrumented and the second to do the actual instrumentation) and then insert several optimization passes between them.

I test my results on selected benchmarks from SPEC2000 and SPEC2006, and create a virtual machine image which allows my results to be reliably reproduced. Lastly, I propose a number of further optimizations which would allow Softbound-CETS to achieve low enough overhead to be used in a mid-performance production system.

Thesis Supervisor: Martin C. Rinard (Stelios Sidiroglou-Douskos)
Title: Professor

Thesis Supervisor: Stelios Sidiroglou-Douskos
Acknowledgments

I would like to thank Professor Martin Rinard for sparking my interest in compilers and for giving me the opportunity to work on this project. Many thanks to Dr. Stelios Sidiroglou-Douskos for endless ideas, sanity checking, and advice. Thanks also to Fan Long and Anthony Eden for answering entirely too many questions about LLVM.

Further gratitude to Professor Gerald Sussman, my undergraduate advisor, for advice, support, and a fascinating window into the world of watchmaking. Thanks to my parents, Laura and Karyn, for encouragement, love, and support.

And I would be remiss if I left out Jenny Ramseyer, my lovely girlfriend. Thanks for the late-night advice, \LaTeX{} help, and paper editing.

And of course, thanks to my MIT friends for collapsing IHTFP into “I Have Truly Found Paradise”.

3.3 Restructuring of SoftboundCETS ............................................ 47
3.4 Redundant Spatial Check Removal ........................................... 48
   3.4.1 Dataflow Analysis ....................................................... 50
   3.4.2 Block-Level Analysis and Removal ................................. 54
   3.4.3 End Result .............................................................. 54
3.5 Redundant Temporal Check Removal ......................................... 54
   3.5.1 Dataflow Analysis ....................................................... 55
   3.5.2 Block-Level Analysis And Removal ................................. 56
   3.5.3 End Result .............................................................. 56
3.6 Constant Array Check Reduction ............................................. 57
   3.6.1 End Result .............................................................. 58
3.7 Loop Hoisting ....................................................................... 59
   3.7.1 Process ........................................................................ 60
3.8 Shadow Stack Removal ............................................................ 71
   3.8.1 Soft Shadow Stack Removal ............................................ 73
   3.8.2 Hard Shadow Stack Removal ......................................... 75
   3.8.3 End Result .............................................................. 76
3.9 Global Variable Hoist ............................................................. 76
3.10 Optimization Effectiveness ...................................................... 78

4 Future Work ........................................................................ 84
4.1 New Optimizations .................................................................. 86
   4.1.1 Constant Loop Check Removal ........................................ 86
   4.1.2 Manual Check Analysis ............................................... 86
   4.1.3 Array Prescanning Hoist ............................................. 87
   4.1.4 Check Batching .......................................................... 89
4.2 Source Code Annotations ........................................................ 91
   4.2.1 Variable Range Annotation ......................................... 91
   4.2.2 Trusted Annotation .................................................... 92
4.3 Enhancements of Existing Optimizations .................................... 92
4.3.1 Constant Array Check Reduction Soundness ........................................ 92
4.3.2 Loop Hoisting Size ................................................................. 93
4.3.3 Loop Splitting ................................................................. 93
4.4 Updating To The New Version of SoftboundCETS .......................... 94
## List of Figures

2-1 A program to set the 3rd element of an array, in C and Python. .... 16
2-2 A program that always prints 5 in Python and probably prints 5 in C. 18
2-3 Example of an unsafe, unchecked typecast. p points at an integer-sized slow of memory, but struct point is two ints long—the write to p->y writes to illegal memory ................... 22
2-4 A contrived example of a sub-object overflow [Nag12] .................. 26
2-5 How arbitrary casts can corrupt metadata. The line p->z = 10 will set the base information of the struct to 10, even though such a write would be allowed. .............................. 27
2-6 A Softbounds spatial check ................................. 32
2-7 An example of direct access. No metadata load is needed, as the base and bound of i are known at compile time. ................... 32
2-8 An example of a metadata load ................................. 33
2-9 Shadow Stack loads and stores in action .............................. 34
2-10 An example of a metadata store/copy. The store is done because *ptr, itself an int pointer, is being assigned to point to a new location... 35
2-11 A CETS temporal check ................................. 36
2-12 The effects of just Softbound versus the effects of Softbound+CETS 37

3-1 Simple example of a PHI node in a loop, with annotated pseudocode for the IR .................................................. 46
3-2 Simple example of a Get Element Pointer instruction .................. 47
3-3 Flowchart of compilation of two files .............................. 49
3-4 A program which doubles one entry of an array, ripe for redundant spatial check removal ............................................ 51
3-5 Skeleton code that displays scopes. B is a child of A, and C is a child of B .......................................................... 55
3-6 A simple C program with a struct access, perfect for Constant Array Check Reduction .................................................. 59
3-7 A simple example of a loop hoist that LLVM will do. The assignment to k is moved from the loop body to the loop header. .............. 60
3-8 A loop with an and statement in its conditional ....................... 65
3-9 A simple example of a the generated LLVM IR for a C loop, before and after loop optimizations ........................................ 71
3-10 A LLVM IR loop, before and after loop hoisting .................... 72
3-11 A simple example of a function call using SoftboundCETS’ shadow stack. 75
3-12 After inlining, our example from Figure 3-11 looks like this. For brevity, I have left all softbound functions unlinlined, although in a real run all non-shadow-stack functions including dereference checks would be inlined. The yellow highlighted lines will be removed. .............. 76
3-13 Once we’ve actually run shadow stack removal, our example from Figures 3-11 / 3-12 looks like this. Note all the shadow stack loads and stores are gone .......................................................... 77
3-14 A simple example of generated LLVM IR involving global variables 78
3-15 Summary of Optimization Gain Per Benchmark .................... 83

4-1 Summary of Optimization Gain Per Benchmark With Projected Gain 85
4-2 A loop of fixed size, allowing the spatial checks to be entirely removed 87
4-3 A simple case where a pointer is already checked by the programmer 88
4-4 A loop with checks indirectly based on the loop induction variable. These are doable, but require prescanning arrays. .................. 90
Imagine that there are two possible buffer sizes here: 32 or 64 characters. In the original case, all possible buffer sizes will be fine. However, in the batched case, a buffer of size 32 will cause a false positive.
List of Tables

2.1 Comparison of C memory safety solutions .................... 23
Chapter 1

Introduction

The lack of memory safety in C/C++ programs is one of, if not the, most persistent and costly sources of program exploits. Memory corruption attacks date back to at least 1988 [Str03], and have been a continual threat for the past 28 years [PSY09, DKA+14, Berol].

Many solutions to C’s memory problems have been proposed [SPWS13], but none of them (and no combination of them) have prevented new memory corruption attacks from appearing. These solutions fall into two broad categories: prevention via memory safety and mitigation after memory corruption. They have been unsuccessful for different reasons. Mitigation does not work, because once a memory address has been corrupted, it’s a snowball effect—one corrupted address can generally be leveraged into another one, until eventually the protection mechanism is either itself corrupted or else sidestepped. Prevention could in theory work, but no prevention system has yet eliminated memory corruption attacks, largely due to the fact that no proposed prevention solution has been able to meet all three necessary criteria for true, practical, memory safety: completeness, compatibility, and overhead:

If a solution is incomplete (meaning that it it theoretically possible for an attacker to get around it), as stated above attackers will find a way around it [EFG+15, ELO+15]. This has born out in every implemented solution to date. Early memory corruption attacks such as the Morris Worm [Str03] relied on injecting code to be executed into the stack. This has since been rendered much more difficult, but, of
course, memory corruption attacks still continue to be found in the wild. Besides non-executable stacks, stack canaries and relocatable code have found their way into major production environments [SPWS13], and while they make some attacks harder, new attacks continue to surface. The only way to prevent continuing memory corruption attacks is to have complete memory safety in C. While measures that give partial memory safety make it more difficult for attackers, they do not render memory corruption attacks impossible, and as attackers have gotten steadily more sophisticated and targets more valuable, “more difficult” has not translated into “fewer successful attacks”. Similarly, measures which attempt to thwart attacks after initial memory corruption has already taken place are ultimately futile—once memory corruption is achieved, such measures can invariably be, themselves, corrupted [SPWS13]. Complete memory safety is needed. However, in order for a solution to stop memory corruption attacks in C, more than completeness is necessary.

Even when a solution is complete, it is useless if not widely deployed. And there are two major hurdles that any memory safety solution must overcome in order to be deployed: compatibility and overhead. A solution is compatible if it can easily be applied to an existing codebase. If a solution requires a large amount of code rewriting, it will never be applied to production code—if a company cares enough about security to rewrite their code to make it memory-safe, they will likely just rewrite it in a language like Java or Rust which has memory safety built in. Further, a solution will not be deployed if it generates unacceptable overhead—memory safety options which cause 15% or greater overhead are rarely used by default [Pay12], and unless a memory safety solution is widely deployed, it does nothing to stop the use of memory corruption attacks.

In this thesis, I look at a number of existing memory safety systems before ultimately selecting one (SoftboundCETS) for improvement. SoftboundCETS is already complete and compatible, so my goal was to reduce its overhead to, hopefully, less than 20%. I first split the existing SoftboundCETS prototype (implemented as an LLVM pass) into two separate passes (one for identifying changes to be made and a second to perform said changes) and inserted a number of optimization passes between
them.

In total, I reduced the overhead of SoftboundCETS by half with passive optimizations that require no developer time to use. I also proposed further optimizations, which, when combined with minimal annotations, will reduce the overhead of SoftboundCETS to 20%-30%, enough to allow deployment in mid-performance production systems.
Chapter 2

Background

The C programming language is notorious for its lack of memory safety. This flaw has lead to numerous [PSY09, DKA+14, Ber01] high-profile security exploits over the years. In this section, I explain the ideas behind memory safety, both spatial and temporal, and look at the difficulties associated with automatically adding memory safety to C. I then take a survey of existing memory safety techniques, noting their strengths and weaknesses, before ultimately deciding on SoftboundCETS for further optimization.

2.1 Computer Memory

All modern computers use a Von Neumann architecture. In a Von Neumann architecture, computer memory is, at its most basic level, an array of cells. Each cell has an address and can hold a value. Computer instructions are stored in a large region of memory. Program Data is using is stored in a different segment of memory. And the stack for the currently running functions is stored in yet another memory segment. See Figure 2-1 for a sample program with two possible generated stacks. Of special note is the cell labeled return address. When the current function finishes, the program will look at that cell to find the address of the next instruction to execute. Also, see the region labeled buff. This is a buffer which has been created directly on the stack. The first element of the buffer is stored in buff[0] and it is indexed up
from there.

(a) In Python

```python
def setThird(buff, value):
    buff[3] = value
```

(b) In C

```c
void setThird(int* buff, int value){
    buff[3] = value;
}
```

(c) The stack of 2-1b when `buff = [1,2,3,4]` and `address = 102324` at the start of the function call. Three cells up from `buff[0]` will be overwritten with 102324, which, in this case, will set `buff[3] = 102324`

(d) The stack of 2-1b when `buff = [2]` and `address = 102324` at the start of the function call. Three cells up from `buff[0]` will be overwritten with 102324. In this case, that’s the return address, which means after `setThird` returns, the program will continue execution reading whatever is at address 102324

Figure 2-1: A program to set the 3rd element of an array, in C and Python.

### 2.1.1 Spatial Memory Safety

Most modern languages (Java, Python, Ruby, Haskell) do not allow users to directly access memory by address. This allows them to guarantee memory safety—in these languages, it is impossible to access undefined memory. However, some older
languages—most notably C, do allow direct memory access. And so, unlike most modern languages, C does not make any guarantees about memory safety.

In both the Python and C examples below, the user assigns to the 6th element of array A. In the Python example, the interpreter looks at the array buff, looks up its second element, and returns the second element. In the C example, however, the outputted assembly will find the address of buff, add one to it, and return whatever is in that memory address.

In most cases, these functions will behave in the same way. If these functions are called with buff as their respective languages’ equivalent of the list [1,2,3,4], and value as 102324, they will both change the list to be [1,2,3,102324]. However, if these functions are called on the “list” [1,2], they will have radically different results. In C, the outputted assembly will look up address of buff, and write 102324 to whatever happens to be three cells past it, overwriting whatever was stored there. In Python, on the other hand, the interpreter will throw an IndexError— it doesn’t just see buff as an address to index off of. Instead, it checks to make sure that the base (lowest allocated element) of buff is less than or equal to the requested index, and the bound (highest allocated element) is greater than or equal to the requested index. Python guarantees the buffer is of a sufficient size before it gets the requested element.

In the best case, the C program writes to memory which has already been used, and will not be read from until reassigned. However, far worse things can happen. If we look back at Figure 2-1d, we can see that, in this particular case, the memory cell three cells past buff is the return pointer. Thus, when the function returns, the next instruction to be executed will be at address 102324. This will probably crash the program. Or, worse, execute code that the developer never meant to have executed. Variations on this theme have lead to numerous security vulnerabilities over the years [PB04].
2.1.2 Temporal Memory Safety

Dereferencing off the end of an existing buffer is not the only way invalid memory can be accessed. C also has a very primitive understanding of how memory should be freed—in most cases, memory must be explicitly freed by the programmer. The only way that memory is naturally freed is when a local variable goes out of scope. And a local variable’s memory is always freed when it goes out of scope.

In the two example programs in Figure 2-2, we will see an idea of what kind of trouble this can cause. In the Python example, i is created, and assigned so both i and j refer to the same list. The list is then updated so its first element is 5, and that element is printed.

```
1 def main():
2     i = [0]
3     if(True):
4         j = [0]
5         i = j
6         j[0] = 5
7         print(i[0])
```

(a) In Python

```
1 int main(){
2     int* i;
3     if(1){
4         int j;
5         i = &j;
6         j = 5;
7 }         printf("%d", *i);
8 }
```

(b) In C

Figure 2-2: A program that always prints 5 in Python and probably prints 5 in C.

In the C example, a very similar thing happens: i is created, then assigned so both i and j refer to the same memory location. The value 5 is assigned to that location, and the contents are printed.

In both languages, j is potentially freed when we leave the if statement. However, Python is smart enough to realize that there is still something referring to that memory (i) and not deallocate it—Python keeps track of a reference count for each object.
When a new reference (anything using the object) is created, the count increases, and when a reference is deleted or removed, the count decreases. If the reference count ever hits zero, the object is immediately removed. Furthermore, Python will periodically stop computation and check all objects for cycles: if object A refers to object B, and object B refers to object A, but nothing else refers to either one, both need to be garbage collected. This has the advantage of providing temporal memory safety, but causes problems with programs that need to always run in real-time; garbage collection can take processing power away from the program it’s collecting from, causing delays. Since i is still referring to the memory allocated for j, it will not be deallocated.

Because C prioritizes speed over safety, it does not have a garbage collector. There are other methods for providing temporal safety that do not cause program delays, but they were largely invented after C. If we look back at Figure 2-2, we see that C has nothing to determine that j is still in use. In C, when a local variable goes out of scope, its memory is deallocated. And so, in the C program, i will be referring to a piece of memory which is invalid. In this particular test case, it is likely to be fine—the memory that i refers to won’t to be re-assigned to anything else in the time between its freeing and subsequent use by i. But it’s always possible that i could be pointing at a different value, and if i was referred to much later in the program, it almost certainly would be. Because i is pointing at deallocated memory, it could be pointing anywhere, including possibly to the return address of a stack frame somewhere. And if i is pointing to the return address of a stack frame and is written to, the same problems arise as with spatial memory safety.

2.2 C and Memory Safety

With all of the problems that come from lack of memory safety in C, it seems odd that the C standard hasn’t been updated to allow some sort of (possibly optional) memory safety. After all, Java is only 10%-20% slower than C, and it provides full memory safety along with a cross-platform VM. Modern computers are fast enough that an
(optional) 20% runtime overhead should be easily worth it for complete memory safely. Unfortunately, unlike Java, C has some properties that make it extremely difficult and expensive to enforce full memory safety: Raw pointers, arbitrary unchecked typecasts, a visible memory structure, and sub-object overflows.

Here, I explain the above problems, and why most memory-safe languages don’t have these problems, using Java as an example language.

2.2.1 Raw Pointers

Raw pointers cause a couple of problems with memory safety.

First and most obviously, the lack of bounds data for pointers is what causes most spatial memory errors—it’s impossible to check that a pointer is within the bounds of its memory if those bounds are not known. This is surmountable, but it’s not the only issue that pointers cause. In Java, pointers are not programmer-accessible. The only way to access an array element is through the array object, which knows its own base and bounds.

Which brings us to the second problem with raw pointers: pointers are, semantically speaking, the same thing as arrays. This equivalence means that there is no way to tell the difference between a pointer intended to access a single element of an array or struct and a pointer intended to access the whole thing. Once again, Java does not have this problem, as pointers are not directly manipulable—in Java, a reference to an entire array is a completely different entity than an array element, and it’s easy to tell the difference between them. The distinction means that a reference to an array doesn’t need to track its bounds information, and an array doesn’t need to be checked every time it is accessed, which is can be extremely expensive. In C, because the two are indistinguishable, this optimization cannot be made, as there’s no way to tell what’s an array and what’s an access to an array.
2.2.2 Internal Arrays And Pointers to Object Middles

In C, pointers can point to the middle of objects. This would not normally be an issue, except that objects (structs) can have internal arrays or internal structs. This means that there is no easy way to distinguish between a pointer to the middle of an object or a pointer to a sub-object, if they share an address.

This problem is, to some extent, impossible to solve—there is no deterministic way to, with complete accuracy, tell if a pointer is to the middle of an object (and thus should be able to access the whole object) or to a sub-object. Memory safety solutions that try to deal with internal arrays will usually have a manual flag which, if enabled, will narrow bounds to sub-objects and if not will not.

2.2.3 Arbitrary Unchecked Typecasts

Arbitrary unchecked typecasts cause additional difficulties. Most notably, a compiler cannot assume that a pointer to an object of a given type actually points to valid memory large enough to hold that type. As an example, see Figure 2-3. p points to a section of memory only one word long, but in line 9, we access the second word of p, which is not allocated. This means that all accesses to pointed-at structs or classes must be checked. Contrast this to Java, which is type-safe. In Java, casting an int* to a point* would be impossible. In Java, class members are always there, and arrays of finite length can be checked at compile time, removing yet more overhead.

Later in this paper (§ 3.6), I will touch on an optimization that I wrote that was unsafe because of this lack of type safety.

2.2.4 Visible Memory Layout

In C, largely because of pointers, the memory layout of structs/the stack/etc. is transparent to the programmer. Although the exact layout is not defined in the C spec, compilers have a convention for laying out memory, and this layout is relied on in many large programs, and, more importantly, in many compiled libraries.

The reliance on exact memory layout is bad for performance of memory safe C;
typedef struct {
    int x;
    int y;
} point;

int main() {
    int a;
    int* p = &a;
    ((point*)p)->y = 10;
}

Figure 2-3: Example of an unsafe, unchecked typecast. `p` points at an integer-sized slow of memory, but struct `point` is two ints long—the write to `p->y` writes to illegal memory.

Any attempt to make C memory safe needs to store some sort of metadata tracking the size of buffers. Because this metadata will be accessed at the same time the buffer is accessed, to maximize performance [NZMZ09], the metadata should be stored inline with with buffer. But doing this breaks certain programs that rely on memory regions being laid out predictably, so metadata must be stored disjointly. This further hurts performance.

### 2.3 Previous Works

Despite these challenges, there have been many attempts to retrofit memory safety onto C. As mentioned in § 1, these attempts juggle trade-offs between three main categories: backwards compatibility, completeness, and overhead. And, as I will explore below, there has not been a single C memory safety solution which fulfills all three categories (is fully backwards compatible, provides complete spatial and temporal memory safety, and has less than 15% overhead). See Table 2.1 for a comparison of various memory safety solutions.

My goal was to take a complete, compatible memory safety solution, and make it fast enough for commercial applications. The idea was, because they are research projects, most of the C memory solutions available are relatively unoptimized. I chose SoftboundCETS [NZMZ10] because it is fast, backwards compatible, and memory safe.
<table>
<thead>
<tr>
<th>Memory Safety Approach</th>
<th>Compatible?</th>
<th>Complete?</th>
<th>Overhead</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valgrind memcheck [SN05]</td>
<td>✓</td>
<td>×</td>
<td>16.7x[NMZ09]</td>
<td>Tripwire</td>
</tr>
<tr>
<td>Address Sanitizer [SBPV12]</td>
<td>✓</td>
<td>×</td>
<td>2x[Nag12]</td>
<td>Tripwire</td>
</tr>
<tr>
<td>Cyclone [JMG+02]</td>
<td>×</td>
<td>✓</td>
<td>1-2x[JMG+02]</td>
<td>Dialect</td>
</tr>
<tr>
<td>Mudflap store [Eig03]</td>
<td>✓</td>
<td>×</td>
<td>7.7x[NMZ09]</td>
<td>Object-based</td>
</tr>
<tr>
<td>Mudflap full [Eig03]</td>
<td>✓</td>
<td>×</td>
<td>24.1x[NMZ09]</td>
<td>Object-based</td>
</tr>
<tr>
<td>Jones and Kelly [JK97]</td>
<td>✓</td>
<td>×</td>
<td>12.1x[NMZ09]</td>
<td>Object-based</td>
</tr>
<tr>
<td>CCured [NMW02]</td>
<td>×</td>
<td>✓</td>
<td>1.3-2x[Nag12]</td>
<td>Pointer-based</td>
</tr>
<tr>
<td>MSCC [XDS04]</td>
<td>×</td>
<td>✓</td>
<td>2x[Nag12]</td>
<td>Pointer-based</td>
</tr>
<tr>
<td>MemSafe [SB13]</td>
<td>✓</td>
<td>✓</td>
<td>2x[Nag12]</td>
<td>Pointer-based</td>
</tr>
<tr>
<td>SoftboundCETS [NZMZ10]</td>
<td>✓</td>
<td>✓</td>
<td>1.1-3x</td>
<td>Pointer-based</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of C memory safety solutions

Before I dive into the specifics of SoftboundCETS, I will look in depth at previous attempts at C memory safety and why I chose not to use them. They can be divided into four main categories: language dialects, tripwire approaches, object-based approaches, and pointer-based approaches.

### 2.3.1 Language Dialects

Language dialects are memory safety solutions which do away with the idea of backwards compatibility and substantially change C semantics in order to achieve memory safety. While this approach can give full safety with relatively low overhead, I was interested in retrofitting memory safety onto existing C programs, which language dialects do not do. Thus, I avoided solutions which dramatically change the language. Cyclone is the most notable dialect solution. Cyclone guarantees safety by a combination of growable regions (heap memory is automatically deallocated when the region it’s allocated in is exited), and a fat pointer spatial safety scheme. However, as it makes no attempt at backwards compatibility, Cyclone was not suitable for my project.
2.3.2 Tripwire Approaches

In a tripwire approach, there is an extra, unwritable, block of memory (the \textit{guard block}) allocated after each object which, if written to, will throw an exception. This prevents many spatial errors (if a program writes contiguously past the end of a buffer, this will be caught), and some temporal errors, but does nothing to stop a program from writing to element 51 of a ten-element buffer if the guard block is 40 cells long.

Valgrind Memcheck and Address Sanitizer are two examples of tripwire approaches, and are among the more commonly used memory safety tools. Early tripwire approaches were mostly used for debugging (Valgrind Memcheck), but modern versions (Address Sanitizer) are fast enough for low performance deployment on production systems (2x overhead). However, more problematically for tripwire approaches, they are not (and cannot be) complete. Firstly, because the guard blocks are finite size and a sufficiently clever attacker can dodge around them, and secondly, because they do nothing to prevent overflows within objects—padding each sub-object with a tripwire would cause much the same problems that fat pointers (§ 2.3.4) have, and so has never been done. And, finally, tripwire approaches do nothing for temporal memory safety.

I chose not to use a tripwire approach because of their lack of completeness.

2.3.3 Object-Based Approaches

In an object-based approach, each allocated object keeps track of its base and bounds. This is generally accomplished by modifying the memory allocator for the program to, when an object is created, also create an entry in a separate table for its base and bounds. Then, when a pointer is manipulated, it is checked to make sure it doesn’t leave the bounds of its current object.

The object-based approach has two main advantages. First, object-based approaches don’t have to worry about changing the memory layout of their program, because object metadata is stored in a separately-allocated table. This means that programs that rely on structs having a specific padding won’t break, and the metadata
won’t accidentally be overwritten by arbitrary typecasts.

Second, if the memory allocator has been modified to create object metadata whenever an object is created, all object creations will create metadata, even when said object is being created by an uninstrumented third-party library. This means that object-based approaches can be freely used with closed-source libraries—external function calls don’t need to be instrumented for the instrumented code to be protected. This compatibility is a huge advantage, but object-based approaches are far from perfect: they have both a solved-but-large-source-of-complexity complication (out-of-bounds pointers), and a yet-to-be-solved disadvantage: incompleteness.

Out-of-bounds pointers are a problem because, according to the C spec, out-of-bounds pointers are allowed, as long as they are not dereferenced. And in a simple object-based model where all pointer manipulations must be in-bound, this mismatch will cause a lot of false positives. Out-of-bounds pointers are dealt with by either adding padding to objects or by using special out-of-bounds proxy objects for the pointers as long as they’re out of bounds, so this problem is possible to work around.

The yet-to-be solved problem is incompleteness. The main way that object-based approaches fail is in the realm of structs with sub-buffers. See Figure 2-4 for an example [Nag12]. In this case, the problem is that a pointer to id and a pointer to bank_account are indistinguishable in code—they both point to the first element of bank_account. If ptr is assumed to point to just the id field, no pointers to bank_account will be able to access the account_balance field (clearly wrong), and if ptr is assumed to point to the entire struct, then the strcpy(ptr, “overflow.”) will overwrite the account_balance field with “.”. Also, like tripwire approaches, no object-based approaches handle temporal memory safety, although there’s no reason why they couldn’t in theory. It might be theoretically possible for an object-based approach to surmount these problems, but no implementation to date does so [Nag12].

Specific object-based approaches are Mudflap [Eig03], Jones and Kelly [JK97], and Baggy Bounds [ACCH09]. Of the three of them, Mudflap and Jones and Kelly were immediately off the table for further work due to their overheads (7.7x and 12.1x respectively), and their lack of completeness. Baggy Bounds is more interesting—its
Figure 2-4: A contrived example of a sub-object overflow [Nagl12]

overhead ranges from 0% to 130% [ACCH09], making it one of the faster approaches. It is also largely compatible with existing C programs. Noninstrumented libraries can be used without any issues, and the only way an existing C program will break is if it relies on a custom memory allocator (as Baggy Bounds uses a custom allocator, overwriting the default allocator).

Unfortunately, Baggy Bounds has a number of issues which make it unsuitable for further optimization. Firstly, as mentioned above, it is incomplete. Baggy Bounds does not find sub-object overflows [ACCH09]. Second, the Baggy Bounds implementation is for the Microsoft Phoenix compiler, which is not easily available. And, lastly, Baggy Bounds is already heavily optimized—further optimizations would be extremely challenging. For these reasons, I decided not to use an object-based approach.

### 2.3.4 Pointer-Based Methods

In contrast to object-based methods, pointer-based methods assign each pointer its own base and bounds. Most pointer-based methods use fat pointers: each pointer is replaced with a struct that contains `pointer_address/base/bound`. When the pointer is modified, its `pointer_address` is changed, and when the pointer is dereferenced, it is checked to make sure `pointer_address` is between `base` and `bound`. This nicely alleviates the two problems with object-based methods: out-of-bounds pointers which are not dereferenced do not cause issues, because pointers’ addresses are not verified until they are dereferenced. And because each pointer has its own base and bounds information, in Figure 2-4, it’s possible to have one pointer point to
id, while another points to the entire bank_account object.

Also, because information is tracked with individual pointers, it is entirely possible to use this information to provide temporal safety as well. Below, when I talk about individual pointer-based methods, I will also specify how they handle temporal safety.

However, fat pointers have their own problems. Because pointer size is different, this changes program memory in programmer visible ways. And as mentioned in § 2.2.4, this causes large code incompatibilities with both compiling existing code and allowing instrumented code to access non-instrumented libraries—namely, a struct passed into a library will often expect its fields to be at a specific offset, which the fat pointers will change. This can be remedied by copying structs containing pointers to remove the extra girth of the pointer for the library call, but deep copying all structs passed to library calls creates unacceptable overhead.

Besides deep copies, because fat pointers contain their metadata inline, structs containing fat pointers are subject to metadata corruption in the presence of arbitrary casts. See Figure 2-5 for an example. To deal with this, fat-pointer based schemes either disallow arbitrary typecasts (which breaks compatibility) or they use much more complex casting mechanics, which adds overhead.

Besides deep copies, because fat pointers contain their metadata inline, structs containing fat pointers are subject to metadata corruption in the presence of arbitrary casts. See Figure 2-5 for an example. To deal with this, fat-pointer based schemes either disallow arbitrary typecasts (which breaks compatibility) or they use much more complex casting mechanics, which adds overhead.

```c
typedef struct{
  int size;
  char* buff;
  int hash;
} str;

typedef struct{
  int x;
  int y;
  int z;
  int t;
} pos;

void badCast(str* s)
{
  pos* p = (pos*) s;
  p->z = 10;
}
```

(a) The layout of str in memory with fat pointers. Note that cells 2 and 3 of the struct contain metadata.

(b) The layout of pos in memory with fat pointers. Note that cells 2 and 3 are z and t, respectively.

Figure 2-5: How arbitrary casts can corrupt metadata. The line p->z = 10 will set the base information of the struct to 10, even though such a write would be allowed.
That being said, the pointer-based approach allows much more completeness than object-based memory safety. And, without fat pointers, backwards compatibility would be much easier. And sure enough, there are several pointer-based approaches for C memory safety which, instead of using fat pointers, have disjoint metadata—they store the metadata for each pointer in a separate datastructure. This eliminates fat pointers, allowing much better compatibility. I looked at several different pointer-based techniques before ultimately settling on SoftboundCETS.

**CCured**

First was CCured [NMW02], which uses fat pointers. CCured implements a garbage collector for C and moves objects which are accessed via pointer from the stack to the heap, providing complete temporal safety. Additionally, it has the advantage of extremely low overhead (30%-100%), while providing complete spatial safety in the presence of arbitrary typecasts. However, the way it achieves this makes unsuited for my purposes:

CCured uses whole-program analysis to break pointers into three classifications: SAFE, SEQ, and WILD pointers. SAFE pointers are known by the compiler to always be either valid or null; they are simply checked for null before dereferencing. SEQ pointers are involved in pointer arithmetic but not in casts. They carry the bounds of the arrays they are supposed to point to, and are checked to be within those bounds when dereferenced. WILD pointers, on the other hand, can be involved in typecasts. Besides base and bound information, they carry struct layout tags which are used to make sure typecasts are not overwriting their metadata. While this works, WILD pointers cause a large amount of additional overhead—CCured’s average overhead on unmodified programs with WILD pointers is approximately 2x. If the source code is manually edited to remove WILD pointers, runtime drops down to an average of 1.3x original.

This means that CCured has two large problems. First, in order to get their 30% overheads, program rewriting is needed. And as discussed in § 1, in order for a solution to be deployed, it needs to be deployed easily. And large-to-moderate
amounts of program rewriting is a large enough barrier for usage that it is unlikely to see widespread adoption. And, more problematically, CCured needs whole-program analysis in order to be sure pointers are not used in typecasts and thus are not WILD. And this provides an even larger barrier to adoption—large projects are often compiled as separate files which are linked together at the last moment, something that is impossible to do while using CCured.

Also, due to the use of a garbage collector, CCured is unsuitable for certain tasks that C is otherwise excellent for, most notably real-time systems such as operating system kernels or video games.

For the above reasons, I chose not to use CCured.

**MSCC**

MSCC [XDS04] solves several of CCured’s problems. It uses disjoint metadata, and does not rely on a garbage collector. Other than this, however, it works similarly to other pointer-based approaches—when a pointer is created, it is assigned base and bound metadata. When it is dereferenced, it is checked to make sure it is within that range.

MSCC gives temporal safety through capabilities. Each memory block has a unique capability, which is stored in a global capability store (GCS). Each capability is stored as an int, in an address in the GCS. When a block is deallocated, its stored capability is changed.

When a pointer to a given block is created, part of its metadata is a pointer to its block’s capability, along with a copy of that capability’s initial value. When a pointer is accessed, its capability pointer is followed, and if it has the same value as its stored capability value, the pointer is temporally sound.

With 2x overhead, it has comparable runtime as to the best solutions I considered.

Unfortunately, MSCC suffers from its own issues. First and foremost, because each block of memory is one capability, it does not differentiate between sub-buffers in structs, giving it some of the same problems as the object-based approach. Also, MSCC has difficulty with arbitrary typecasts. While they do support most typecasts,
they cannot handle casts from pointers to integers and back[XS16], and it’s not clear if their runtime numbers are for programs with arbitrary typecasts or without. And, lastly, it requires a small amount of code rewriting to function[XS16]:

```c
void *p = malloc(1024);
...
char **q = (char **)p;
```

must be changed to

```c
char **p = (char **) malloc(1024);
...
char **q = p;
```

Also, like CCured, MSCC relies on whole-program analysis, which makes compilation of large projects difficult. And, MSCC’s prototype was developed as a stand-alone compiler, which makes widespread adoption unlikely—a modification for an existing compiler (GGC or Clang) is much more likely to be integrated into an existing project than it is for a new project to gain a large market share.

For these reasons, I didn’t use MSCC.

MemSafe

MemSafe [SB13], like MSCC, avoids fat pointers by using disjoint metadata, and does not rely on a garbage collector. Like other pointer-based approaches, MemSafe gives each pointer a base and a bound, which are checked when the pointer is dereferenced. Each pointer also has an ID field, which is used for temporal safety:

Whenever a pointer is freed (or goes out of scope), the object it points at has its metadata edited to have a size of -1. Thus, if any pointer to the object is later dereferenced, it will throw a spatial error. However, if that object has an internal pointer, the internal pointer’s value’s size would not be similarly reduced. So, each pointer also keeps an ID field, which indicates its pointed object’s outermost object. When a pointer to a sub-object is dereferenced, its ID’d parent object is also checked.

Performing this ID lookup is much more expensive than a normal check, however, so MemSafe relies on full-program analysis to determine which pointers can point
to internal sub-objects. MemSafe also uses full-program analysis for a number of other optimizations which, collectively, allow MemSafe to get an overhead of 130% on average for SPEC2000 and SPEC2006 benchmarks [SB13].

Given all of this, MemSafe is a reasonable target for further optimization. It has the drawback of requiring full-program analysis, but other than that, is an excellent candidate. However, I found a better optimization candidate in SoftboundCETS.

2.3.5 SoftboundCETS

SoftboundCETS is a pointer-based scheme for C memory safety. It is actually the fusion of two distinct systems: Softbounds[NMZ09], which provides spatial memory safety, and CETS[NMZ10], which provides temporal memory safety. Both programs are compiler passes for LLVM 3.4—they take unannotated C code and generate a binary which is spatially and temporally memory-safe, a bit slower, and otherwise runs exactly the same as the original program. SoftboundCETS is a fusion of the two of them, providing both spatial and temporal memory safety. In this section, I will first go over how exactly Softbound and CETS each work in theory, and then talk about the state of the combined implementation which I am using.

Softbounds

Softbounds is a pointer-based memory safety scheme which uses disjoint metadata, and does not provide any temporal safety on its own. Fundamentally, it works the same way as any other pointer-based disjoint metadata scheme—every time a pointer is dereferenced, it is checked as in Figure 2-6—the pointer, its base, bound, and size are passed in to a function which makes sure that the pointer is within bounds. There is a small amount of overhead from the actual check, but more overhead tends to come from the collection of the base and bound.

There are three ways to collect the metadata for a spatial check: shadow stack loading, metadata loading, and direct access.
```c
void spatial_check(ptr, base, bound, size) {
  if ((ptr < base) || (ptr+size > bound)) {
    abort();
  }
}
```

**Figure 2-6:** A Softbounds spatial check

**Direct Access** Sometimes, the base and bound of a memory access are determinable statically at compile time. This is usually for local variables. As an example, see 2-7. From an overhead perspective, direct access is definitely the best, as it introduces no overhead over the check itself. However, most of the time, one of the other two methods are used:

```c
#include<stdio.h>
typedef struct {
  int x;
  int y;
} point;

int main(){
  volatile point i;
  i.x = 10;
}
```

**Figure 2-7:** An example of direct access. No metadata load is needed, as the base and bound of i are known at compile time.
**Metadata Loading** Metadata loading is done whenever the base and bounds of the pointer to be checked are not immediately known but the identity of the pointer is. This is done with globals as well as locals defined sufficiently far away from the check. Figure 2-8 is a simple example of a place where metadata loads are needed. We can see that i is declared as a global, and so the metadata for i will need to be loaded from its external repository.

```c
#include<stdio.h>
int* i;

int main(){
  printf("%d", *i);
}
```

Figure 2-8: An example of a metadata load

**Shadow Stack Loading** Shadow stack loading, as shown in Figure 2-9, is performed for pointers which are function parameters—when it’s unknown which pointer will be checked. When a call to a function with a pointer parameter is found, Soft.bounds pushes the base and bound of the pointer onto the shadow stack (Figure 2-9b). When the function executes, the first thing it does is load the base and bound of the pointer from the shadow stack (Figure 2-9c). That loaded base and bound are then passed in as the base and bound for 2-6. Size is easily calculated by the compiler from type information of ptr.

The above mechanisms are, collectively, responsible for between, usually, 80% to
1. **void increment(int* i){**
2. *i += 1;
3. **}
4. **int main(){**
5. **int i = 4;**
6. increment(&i);
7. **}**

(a) Original C code

1. **define i32 @softboundcets_pseudo_main() {**
2. **entry:**
3. %0 = call i8* @__softboundcets_get_global_lock()
4. %i = alloca i32, align 4
5. %i1 = bitcast i32* %i to i32*
6. %bitcast = bitcast i32* %i to i8*
7. %mtmp = getelementptr i32* %i1, i64 1
8. %bitcast2 = bitcast i32* %mtmp to i8*
9. call void @__softboundcets_allocate_shadow_stack_space(i32 2)
10. call void @__softboundcets_store_base_shadow_stack(i8* %bitcast, i32 1)
11. call void @__softboundcets_store_bound_shadow_stack(i8* %bitcast2, i32 1)
12. call void @increment(i32* %i1) #1
13. call void @__softboundcets_deallocate_shadow_stack_space()
14. ret i32 0

(b) Generated IR for main

1. **define void @increment(i32* nocapture %i) #0 {**
2. **entry:**
3. %0 = call i8* @__softboundcets_load_base_shadow_stack(i32 1)
4. %1 = call i8* @__softboundcets_load_bound_shadow_stack(i32 1)
5. %2 = call i8* @__softboundcets_get_global_lock()
6. %bitcast = bitcast i32* %i to i8*
7. call void @__softboundcets.spatial.dereference_check(i8* %0, i8* %1, i8* %bitcast, i64 ptrtoint (i32* getelementptr (i32* null, i32 1) to i64))
8. call void @__softboundcets.temporal.load.dereference_check(i8* %3, i64 %2, i8* %0, i8* %1)
9. %3 = load i32* %i, align 4, !tbaa !1
10. %add = add nsw i32 %3, 1
11. store i32 %add, i32* %i, align 4, !tbaa !1
12. ret void

(c) Generated IR for inc

Figure 2-9: Shadow Stack loads and stores in action

100% of the overhead, dependent on which program is being run. Generally speaking, of that 80%–100%, 30% of it are the actual spatial checks, while the rest is the metadata loads and shadow stack loads/stores. Between metadata loads and shadow stack loads/stores, the relative amounts of overhead are highly variable—metadata loads are much cheaper (8 instructions), but also more common. A complete shadow stack store/load for base and bound is 36 instructions, so the exact overhead distribution depends how many function calls vs. nonlocal accesses there are.
Metadata Copying  The remaining 20% (more, in a few benchmarks) is metadata copying. Whenever a pointer \( n \) is created off of another pointer \( m \), \( n \)'s metadata is copied from \( m \). See Figure 2-10 for an example. These metadata copies are 16 instructions each, and in most of my test benchmarks, were almost negligible. However, in benchmarks that have lots of pointer assignments within loops (most commonly from linked list traversal), metadata copies can add up to a large amount of overhead.

```c
#include<stdio.h>
void inc_ptr (int** ptr){
    *ptr = *ptr + 1;
}
```

```c
define void @inc_ptr(i32** nocapture %ptr) #0 {
entry:
%base.alloca = alloca i8*
%bound.alloca = alloca i8*
%0 = call i8* @__softboundcets_load_base_shadow_stack(i32 1)
%1 = call i8* @__softboundcets_load_bound_shadow_stack(i32 1)
%2 = call i8* @__softboundcets_get_global_lock()
%bitcast0 = bitcast i32** %ptr to i8*
call @__softboundcetsSpatialLoadDereferenceCheck(i8* %0, i8* %1, i8* %bitcast0, i64 ptrtoint (i1** getelementptr (i1** null, i32 1) to i64))
%3 = load i32** %ptr, align 8, !tbaa !1
%bitcast = bitcast i32** %ptr to i8*
call void @__softboundcetsMetadataLoad(i8* %bitcast, i8** %base.alloca, i8** %bound.alloca)
%base.load = load i8** %base.alloca
%bound.load = load i8** %bound.alloca
%add.ptr = getelementptr inbounds i32* %3, i64 1
store i32* %add.ptr, i32** %ptr, align 8, !tbaa !1
%bitcast1 = bitcast i32** %ptr to i8*
call void @__softboundcetsMetadataStore(i8* %bitcast1, i8* %base.load, i8* %bound.load)
ret void
```

Figure 2-10: An example of a metadata store/copy. The store is done because \(*\text{ptr},\) itself an int pointer, is being assigned to point to a new location.

CETS

CETS guaranteed full temporal safety, assuming spatial safety. In general, it works very similarly to Softbounds—every time a pointer is dereferenced, it is checked. However, instead of the check in Figure 2-6, the check inserted by CETS is Figure 2-11. In the temporal check, there are only two fields: \( \text{ptr.key} \), and \( \text{ptr.lock_addr} \).

The CETS method for temporal safety is similar to that of MSCC. Whenever a block
is created, a unique key is associated with said block, and stored in disjointly. When a pointer to a block is created, it is given both the value of the key and a pointer to the disjoint key store. When a pointer is dereferenced, as in Figure 2-11, the value of the key is compared to the value stored in the disjoint key store.

```c
void temporal_check(int ptr_key, int* ptr_lock_addr){
  if (ptr_key != *ptr_lock_addr) {
    abort();
  }
}
```

Figure 2-11: A CETS temporal check

**Metadata Acquisition**  
CETS uses exactly the same metadata propagation methods as Softbounds—when a pointer is a function argument, its temporal data is propagated on the shadow stack along with its spatial data. When a pointer is looked up, its temporal information is loaded along with its spatial information. In fact, with CETS, Softbounds hardly changes—see Figure 2-12 for the difference in the case of a metadata load.

**SoftboundCETS Implementation**

SoftboundCETS is implemented as a single LLVM 3.4 pass with several phases. The first phase, which I will call boilerplate, performs transformations and analysis necessary for the later phases to run. This includes initializing the functions inserted into later passes (Creating the function prototypes for spatial_check, temporal_check, etc.) and figuring out which functions to transform (there’s an option to blacklist certain functions).

The second phase, broken up into two functions, is gathering of base/bound/key/lock metadata. This is where SoftboundCETS inserts the code that calculates metadata. Here, the calls to load_metadata and shadow_stack_load are inserted. However, at this point, the actual checks are not. Instead, the values which will be returned by these calls are stored in a global map, metadata_associations.
Figure 2-12: The effects of just Softbound versus the effects of Softbound+CETS
The last phase is check insertion. In this phase, `spatial_check` and `temporal_check` are inserted into the code. Their arguments are the metadata values pulled from `metadata_associations`, and here, SoftboundCETS performs extremely simple local optimizations to make sure that the same address is not checked twice in the same block. This phase is responsible for most of the overhead—while metadata loads actually cause a larger fraction of the overhead, LLVM is smart enough to optimize them out if the check relying on them is removed.

While SoftboundCETS is theoretically backwards compatible and able to compile any C program, the prototype implementation I’m using falls short of this goal—it does not handle vectorization, so getting it to compile some programs amounts to messing with compiler flags until no vectorization is going on. Also, some programs (GCC, for instance) simply won’t compile with it, regardless of compiler optimizations.

Despite that, it does seem to work for most of the programs I tried. Of the benchmarks I ran it with, SoftboundCETS gave an average overhead of 172%, or 2.72x. This is a bit above the 2x claimed in the SoftboundCETS paper. Some of this can be explained by the fact that I used a somewhat different benchmark set than the SoftboundCETS authors, but there are several specific benchmarks that we both did were I got substantially higher overhead than they did. When I talked to Dr. Nagarakatte [Nag16], I was told that differences of ±50% of the total overhead was normal and I shouldn’t worry about it. As he did not have access to the VM which they used to get their results, I was unable to see if what my differences result from. I hypothesize that they are due to some or all of the following factors:

- I’m running on a different version of Ubuntu than they were.
- I had my benchmark compilation scripts configured differently than they did (They did not have access to the original compilation scripts when I asked).
- The original SoftboundCETS results were done with the pre-alpha version of SoftboundCETS which ran on LLVM 2.5, while the version I am using is LLVM 3.4. The different LLVM versions might be the cause.
• The pre-alpha version of SoftboundCETS had some differences in how it actually modifies the code (it doesn’t use a shadow stack).
Chapter 3

Optimizations

As mentioned in § 2.3.5, SoftboundCETS is a compiler pass for LLVM 3.4. It provides complete spatial and temporal memory safety (although sub-object overflows are detected with an optional flag—some valid C programs rely on pointers to internal struct fields accessing other internal struct fields), with an average overhead of 100%. My goal was to bring this overhead down to 15%. In order to identify candidate optimizations, I broke this goal into three steps: benchmark gathering and setup, overhead analysis, and optimization implementation.

3.1 Benchmarking

In the paper where SoftboundCETS is introduced, it is evaluated on a mix of SPEC2006, SPEC2000, and SPEC1995 benchmarks. In order to maintain consistency, I tried to recreate their original benchmark set, with the goal of further optimizing the runtime of their benchmarks. Unfortunately, I was not able to get a copy of SPEC1995, and the SoftboundCETS prototype I had access to was unable to compile several of their benchmarks. The benchmarks I ended up using were bzip, sjeng, lbm, gobmk, hammer, h264ref, sphinx3, equake, and ammp. I was unable to get a copy of compress and jpeg, as they were part of SPEC1995, and I was unable to get crafty, mesa, art, libquant, or vpr to compile with SoftboundCETS (even with O0), due to assertion failures thrown by SoftboundCETS. I added milc, as it was easy to compile with
SoftboundCETS and I wanted more datapoints.

Once I had a set of benchmarks compiling, my first task was to determine overhead and analyze where exactly the SoftboundCETS overhead was coming from.

I determined the instruction-level overhead by using Valgrind’s Callgrind functionality to count the number of assembly instructions for each benchmark with and without SoftboundCETS.

In order to determine where exactly the overhead from each benchmark was coming from, I tagged each LLVM IR instruction in SoftboundCETS with a piece of metadata marking it as `Overhead`. I then had a separate pass which added a dump statement after every `Overhead` instruction and dumped a loop hierarchy of instructions. And finally, an MIT scheme program which aggregated the dumped instructions and loop hierarchy and output a coherent cost model for each loop and function. This model is not incredibly precise, as it is at the LLVM IR level granularity, but it was enough to identify which loops and which LLVM IR instructions were particularly costly.

Given my analysis and benchmarks, I present a list of passive compiler optimizations which will, in theory, reduce the overhead of SoftboundCETS by roughly 75%. Unfortunately, 75% of 172% is 43%, which is still more overhead than would be acceptable by industry standards. Thus, I further propose a list of simple annotations which would allow overhead to drop below the 15%, on average.

For optimizations, I propose:

- **Redundant Check Removal**: SoftboundCETS removes checks to the same address within the same basic block, but does not do this cross-basic-block. Removing redundant checks across different basic blocks will provide small speedups in almost every benchmark.

- **Constant Array Check Removal**: If an array’s size is known at compile time, and an access into said array is constant, the check of that access can be removed—safety can be determined at compile time.

- **Loop Check Hoisting**: Instead of running checks within a loop, if the loop
bounds can be determined, it’s entirely possible to run all of the checks before the loop runs. This can give dramatic speedup in benchmarks that rely on large loops.

- **Shadow Stack Removal**: If a function is inlined, its shadow stack loads and stores can be turned into direct access. See § 2.3.5 for a comparison of different metadata collection techniques.

- **Global Variable Hoist**: LLVM does not hoist global variables in circumstances where it would be safe, sans concurrency issues. Because Softbound-CETS does not provide memory safety in the presence of concurrency (and thus programs compiled with SoftboundCETS can be assumed to be serial), hoisting globals is safe. This allows further Loop Check Hoisting

- **Constant Loop Check Removal** (Future work): If a loop has constant bounds, it is possible to altogether remove checks which rely on the loop induction variable (if they are accessing arrays of constant size), as the range of said variable can be determined statically. This provides a significant performance improvement for milc.

- **Manual Check Analysis** (Future work): The idea for this optimization is that in many places, programmers have already inserted null checks. If a pointer a.) is not involved in pointer arithmetic and b.) cannot be null at a given point in code, because of programmer-inserted checks, checks on it can be removed. This optimization is not always safe, but could be sound if some sort of CCured-like type-analysis could be done on local variables to ensure they are not used in pointer arithmetic. This is highly situational, but allows the removal of many checks to linked list traversal functions. This would reduce ammp overhead by almost half.

- **Array Prescanning Hoist** (Future work): Several benchmarks (most notably sjeng) have loops where the loop induction variable is not be used as an index, but instead is used as an index into a different array, and that returned value is
used as an index for the other array accesses in the loop. It’s possible to hoist these loops by scanning the array the final index will come from to make sure that all possible values are in bounds.

- **Check Batching** (Future work): Many benchmarks access the same buffer repeatedly in different places. Check Batching would, instead of checking all of the accesses to a given buffer, only check the highest and lowest access. If they are both in bounds, all accesses between them will be too.

The annotations I propose are:

- **Variable Range Annotation**: Many loops cannot be hoisted because their range cannot be determined. Variable Range Annotation would assert that a given variable stays in specified bounds. This would both allow previously impossible loop hoists and allow the removal of checks of static arrays larger than the annotated bounds.

- **Trusted Blocks**: By marking certain blocks of code as trusted, all checks can be removed from them. This leaves metadata stores as the only major source of overhead in trusted blocks.

Unfortunately, I only had time to implement approximately half of my proposed optimizations (Redundant Check Removal, Constant Array Check Removal, Loop Check Hoisting, Shadow Stack Removal, and Global Variable Hoist), and none of the proposed annotations. The yet-to-be implemented improvements are saved for future work, while in this chapter, I will go over the completed optimizations in detail: how they work, why they work, and how much speedup each one gives.

According to my evaluation, the original SoftboundCETS pass gave an average of 192% overhead. With my optimizations, there is an average of 99.51% overhead, a nearly 50% speed up. According to the CETS paper [NZMZ10], SoftboundCETS should have a 116% overhead, but I have a somewhat different set of benchmarks (I added one that they did not use, and several benchmarks which they claim should work I was unable to run with SoftboundCETS). If my speedup holds with their
benchmark set, the overhead should be around 58%. Considering that alternatives
clock in much higher, as seen in Table 2.1, this is fantastic—The only alternative
explored by Santosh et. al. which has comparable overhead is CCured [NMW02],
which requires large-scale code modifications. There are other alternatives (Baggy
Bounds checking [ACCH09]), which might be faster, but are incomplete—they do
not catch all spatial violations, and as far as I can tell, there are no comparable pro-
grams which have a remotely similar overhead that catch both spatial and temporal
violations, while retaining backwards compatibility.

The only alternatives which is both comparable and complete is MemSafe. Un-
fortunately, MemSafe relies on who-program analysis [SB13] which makes compila-
tion difficult in certain circumstances. SoftboundCETS, and my optimizations, have
no such restriction—while whole-program analysis can decrease overhead slightly (it
makes Shadow Stack removal work better), it is unnecessary both for safety and for
the majority of my performance speedup.

But before I talk about the technical details of each optimization, I will explain
the language that they operate on:

3.2 LLVM Intermediate Representation

LLVM (Low Level Virtual Machine) [LA04] is a compiler backend, which, coupled
with Clang [Lat08] (the C frontend for LLVM), is one of the most popular C compi-
lers, along with GCC and Visual C. Like all compilers, LLVM/Clang consists of a
number of passes, which incrementally change and improve code until it eventually
spits out optimized machine code. Unlike most compilers, LLVM only ever uses one
intermediate representation (IR) between passes, and said IR can easily be dumped
and re-read back in. This means that it is easy to work with—a programmer who
wants to hack on LLVM only needs to learn one new language, not several, and does
not need to hook their work into the depths of LLVM. This makes LLVM very popular
for compiler research projects.

LLVM’s IR is mostly a straightforward assembly-with-additions—it has load,
store, basic arithmetic, jumps, etc. However, it has a few nonstandard types of instructions which can cause some confusion. As these additions feature in my work, I explain them below.

### 3.2.1 PHI nodes

LLVM IR is Single Static Assignment (SSA) [LA04]. SSA means that each variable is defined exactly once, and once defined cannot be changed. In order to allow, for example ternary assignment, PHI nodes are a necessary component: A PHI node is a variable that has its value dependent on which block it was reached from. SSA also makes certain things (loops, most commonly) hard to represent efficiently without PHI nodes. See Figure 3-1 for an example: At the start of the loop (line 8 of Figure 3-1b), \( i = 0 \). But, each loop iteration, \( i \) changes value. The way that is handled without a PHI node is to re-load \( i \) for each iteration of the loop (lines 13 and 16). This works, but most LLVM loop optimizations will see \( %0 \) (the value checked for whether or not the loop continues) and \( %1 \) (the value incremented once a loop) as two different values, making loop optimizations impossible. A PHI node handily solves this problem, as in Figure 3-1c. Here (line 7), \( \%i.02 \) has the value \( %0 \) if control comes from the basic block \( \%entry \), and has the value \( %inc \) if control comes from the basic block \( \%for.body \). In this case, the optimizer can trace what exactly is going on, allowing for faster, more concise code.

### 3.2.2 Get Element Pointer

Get Element Pointer (GEP for short) is an instruction that calculates a memory address. It notably does not actually access said address. See Figure 3-2 for an example. As you can see from the C code, all that is happening here is that 5 is being stored into \( j[5] \). However, the LLVM code shown is slightly more complex—the GEP instruction is \( \%arrayidx = getelementptr inbounds [10 x i32]*\%j, i32 0, i64 5 \)

What the \( i32 \ 0 \) does is gets the address of the 0th element of \( \%j \). This makes
void phi_node()
{
    int j;
    for(int i = 0; i < 1000; i++){
        j = 10;
    }
}

(a) C code for a simple loop. Note how i has different values at the start of the loop depending which iteration of the loop we are in.

(b) Without the PHI node, %i is loaded twice—once as %0 and once as %1. Since most LLVM passes don’t track associations between loads, this makes further optimizations almost impossible. Note that this code is largely unoptimized—PHI nodes are one of the first things added by optimizations.

c) With the PHI node, %i isn’t actually needed at all—the loop variable is %i.02, which starts at 0 when coming from entry, and otherwise is %inc when coming from for.body

Figure 3-1: Simple example of a PHI node in a loop, with annotated pseudocode for the IR
sense if you look closely at the type information for %j above. %j is not of type [10 x i32], it is of type [10 x i32]*. Thus, the i32 0 is to look at the 0th element of that pointer. Once the 0th element is being looked at, the type is now [10 x i32], and we get the 6th element there with the i64 5.

Note that when I say “look at” I really mean “calculate the address of”—the GEP by itself does not access any memory, it merely calculates the address. It’s the next instruction, store i32 5, i32* %arrayidx, align 4 that stores the value 2 into the memory address indicated by %arrayidx

```
void gep()
{
volatile int j[10];
j[5] = 2;
}
```

(a) C for a simple array access. This will generate a GEP.

```
define void @gep() #0 {
entry:
  %j = alloca [10 x i32], align 16
%arrayidx = getelementptr inbounds [10 x i32]* %j, i32 0, i64 2
store volatile i32 5, i32* %arrayidx, align 4
ret void
}
```

(b) Generated LLVM IR

Figure 3-2: Simple example of a Get Element Pointer instruction

### 3.3 Restructuring of SoftboundCETS

The first modification I made to SoftboundCETS was not an optimization, but some structural changes which enabled almost all of my other optimizations. In its original form, SoftboundCETS is one pass with several components, which can largely be divided into two categories: Analysis Phase and Modification Phase. In the analysis stages, SoftboundCETS figures out which instructions need checking (it does some simple optimizations). Modification Phase (the steps where the code is actually rewritten to add checks) is quite complex, largely because each check needs metadata which has to be loaded, which is stored in a table in-memory.
This means that adding additional checks after the SoftboundCETS pass runs is extremely difficult. Since many of my passes both add and remove checks, they really need to run after Analysis Phase (so I know what instructions are going to be checked) and before Modification Phase (so if I want to add a new check, say, for loop hoisting, I can have SoftboundCETS do it instead of trying to reverse-engineer which metadata corresponds to which buffer).

Thus, I split SoftboundCETS. This involved breaking up some function bodies, but what I ended up with was two passes: SoftboundCETSMark (Analysis Phase) and SoftboundCETSPass (Modification Phase). SoftboundCETSMark runs the portions of SoftboundCETS responsible for finding which instructions are to have checks added. However, instead of adding the checks, it adds metadata to the instruction indicating that it will be checked. This is the only modification to the IR that SoftBoundCETSMark performs. SoftboundCETSPass does all of the other modifications to the IR, and, in general behaves very similarly to the original SoftboundCETS, except it inserts checks wherever an instruction has “insert check here” metadata. See Figure 3-3 for a flowchart of compilation involving two files. Each sequence up to linking is a separate file, while the sequence afterward is the linked files.

This splitting is extremely useful—what was once a single monolithic pass (which would have to get larger and larger as more optimizations were added) is now two smaller passes. And anyone who wants to write an optimization pass can easily slip another between the run of SoftboundCETSMark and SoftboundCETSPass.

### 3.4 Redundant Spatial Check Removal

After splitting SoftboundCETS, I began adding optimization passes. Unless otherwise noted, these all run between SoftboundCETSMark and SoftboundCETSPass.

Redundant Spatial Check Removal is actually one of two closely related passes which form a core part of my optimizations: Redundant Spatial Check Removal, and Redundant Temporal Check Removal.

The majority of SoftboundCETS overhead comes not from actually running its
Figure 3-3: Flowchart of compilation of two files
checks, but from loading the metadata (the base, bound, key, and lock) necessary for running of said checks. Thus, one of my optimization goals was to remove as many metadata loads as possible. And in order to do so, checks that rely on them must first be removed safely.

At a high level, this pass works to remove twice-performed spatial checks. Given a code snippet like Figure 3-4a, LLVM will generate IR that looks like 3-4c. For ease of viewing, I’ve translated it into pseudo-C 3-4b. SoftboundCETS will, by default, add a check to each of these accesses—both the store and the load. However, if the load is safe, the store is too—the second check is unnecessary. This pass removes that second, unnecessary check.

By far the bulk of the work in this pass is in its dataflow analysis—determining which spatial checks are redundant. After dataflow analysis, there is a block-level analysis and removal step.

Redundant Spatial Check Removal is done in between marking and insertion. At the time it’s run, instructions that will have checks marked as such in the LLVM IR metadata, but the checks are not actually inserted yet. This allows easy removal of checks—simply remove the metadata that says "Insert a spatial check here".

### 3.4.1 Dataflow Analysis

The analysis for Redundant Spatial Check Removal is a straightforward forward dataflow analysis [Rin16]. Its GEN, IN and OUT set entries consist of a buffer followed by a single index (or list of indexes for multidimensional buffers) which are checked in the block (and are not killed within the block). If a buffer is accessed in multiple places, it may have multiple entries in a given basic block’s GEN set with different indices. Its KILL set entries are single variables whose values are changed in the basic block.

Its GEN function does the following:

- For each instruction in a basic block:
  - If it is a load, look at the loaded value:
#include<stdlib.h>

void doubleEntry(int i, int* arr){
    <metadata load>
    <spatial check>
    <temporal check>
    <metadata load>
    <spatial check>
    <temporal check>
    <metadata load>
    <spatial check>
    <temporal check>
    arr[i] = arr[i] + arr[i];
}

(a) Original Source Code. There are three sets of checks, one for each array access. One array access (and thus check) will be removed by LLVM, but the other two are there to stay

void addEntries(int i, int* arr1){
    <metadata load>
    <spatial check>
    <temporal check>
    int t1 = arr[i];
    int t2 = t1 + t1;
    <metadata load>
    <spatial check>
    <temporal check>
    arr[i] = t3;
}

(b) Expanded source code. This is more or less what LLVM will generate, expressed in C-syntax. Note that there are only two checks now

define void @doubleEntry(i32 %i, i32* nocapture %arr) #0 {
entry:
    %idxprom = sext i32 %i to i64
    %arrayidx = getelementptr inbounds i32* %arr, i64 %idxprom
    %0 = load i32* %arrayidx, align 4, !tbaa !l
    %add = shl nsw i32 %0, 1
    store i32 %add, i32* %arrayidx, align 4, !tbaa !l
    ret void
}

(c) The generated LLVM IR. Note the load of array and the store of the array.

Figure 3-4: A program which doubles one entry of an array, ripe for redundant spatial check removal
* If the loaded value is a Get Element Pointer (GEP) (See § 3.2.2 for details on GEP), create a GEN entry with all the GEP’s indices.

* If it is not a GEP, we are accessing a pointer at location 0, so create a GEN entry for the loaded pointer with with index 0.

- If it is a store, do the same as above, for the value being stored (not the variable being stored to). If a value is being stored, it will be checked. Then remove anything from the GEN set that has the variable being stored to as an index—if a variable is being stored to, its value may change, so anything relying on its value will not be known at the end of the block.

- If it is a function call, and the function body is available, find all globals and parameters in the function which are written to, and remove anything that relies on them. If the function body is unavailable, kill any GEN entry that relies on a global or a variable being passed to the function.

Pseudocode can be found in Algorithm 1.

Its KILL function is:

• For each instruction in a basic block:

  - If it is a store, add the variable being written to the KILL set
  
  - If it is a function call, and the function body is available, find all globals and parameters in the function which are written to, and add them to the KILL set. If not, just add all globals, and all parameters to the function to the KILL set.

Pseudocode can be found in Algorithm 2

Its MERGE INPUTS function is simple. A given buffer location must be checked in all possible predecessors in order to know it’s safe, so MERGE INPUTS is simply the intersection of the OUT sets of all predecessors.

Its TRANSFER function is also simple. If a variable has been killed, anything relying on it can no longer be known safe, so we remove all IN entries which have an
Parameters: a basic block $B$
Returns : $GEN$, a set of GEN entries

$GEN \leftarrow \{\varnothing\}$

for Instruction $I$ in $B$ do
  if $I$ is a load then
    $v \leftarrow getLoadedValue(I)$
    if $v$ is a Get Element Pointer then
      $G \leftarrow v.getOperands()$
      $GEN \leftarrow \{GEN, G\}$
    else
      $GEN \leftarrow \{GEN, [v, 0]\}$
    continue
  if $I$ is a store then
    $l \leftarrow getStoredValue(I)$
    $r \leftarrow getStoredToValue(I)$
    if $l$ is a Get Element Pointer then
      $G \leftarrow [l.getOperands()]
      $GEN \leftarrow \{GEN, G\}$
    else
      $GEN \leftarrow \{GEN, [l, 0]\}$
    for $G \in GEN$ do
      if $r \in G$ then
        $GEN \leftarrow GEN - \{G \in GEN \mid r \in G\}$
    continue
  if $I$ is a function call then
    $V \leftarrow getWrittenParametersAndGlobals(I.getFunction())$
    $GEN \leftarrow GEN - \{G \in GEN \mid V \cap G \neq \varnothing\}$
end

return $GEN$

Algorithm 1: Redundant Spatial Check Removal $GEN$ algorithm.

Parameters: a basic block $B$
Returns : $GEN$, a set of GEN entries

$KILL \leftarrow \{\varnothing\}$

for Instruction $I$ in $B$ do
  if $I$ is a store then
    $r \leftarrow getStoredToValue(I)$
    $KILL \leftarrow KILL, r$
    continue
  if $I$ is a function call then
    $V \leftarrow getWrittenParametersAndGlobals(I.getFunction())$
    $KILL \leftarrow KILL \cup V$
end

return $KILL$

Algorithm 2: Redundant Spatial Check Removal $KILL$ algorithm.

index that is in the $KILL$ set or whose buffer is in the $KILL$ set. We take the result of the above, add $GEN$ to it, and we get the $OUT$ set.
After the dataflow analysis is complete, each basic block has a IN set which lists all entries in all buffers which we are sure are checked. Once we have that, we proceed to the basic block level analysis and removal of redundant instructions.

3.4.2 Block-Level Analysis and Removal

The block-level analysis and removal runs almost identically to the dataflow GEN function, with some small modifications.

First, we make a copy of IN, which we’ll call CHECKED. Now, we perform dataflow GEN, using CHECKED instead of IN. And, before anything else, whenever we encounter a load or a store that is marked for a SoftboundCETS check, we remove the check if the address being stored to or loaded from is in CHECKED.

3.4.3 End Result

At the end of this pass, already-performed spatial checks are gone. This, by itself, only reduces overhead by 5%. However, it allows the optimizer to remove the portions of metadata loads which the spatial checks relied on. For testing purposes, I combined this with Redundant Temporal Check Removal to get overhead removal numbers.

3.5 Redundant Temporal Check Removal

This pass is the close cousin of Redundant Spatial Check Removal (RSCR). Like RSCR, it relies on a forward dataflow analysis, but due to the nature of temporal safety, it is rather more complex—instead of one GEN, IN, and OUT set for each basic block, it keeps track of one GEN, IN, and OUT set for each scope, for each basic block. Each basic block still only has a single KILL set.

The way Redundant Temporal Check Removal works is that it assigns each basic block to a scope. (This requires access to LLVM’s debug info, which is relatively easy to get. It just mandates that programs be compiled with the -g flag). Scopes are arranged in a tree hierarchy. See Figure 3-5 for an example—Scope A is the parent...
of B, which is the parent of Scope C.

The extra complexity in Redundant Temporal Check Removal comes from the fact that memory can go out of scope—if a memory cell is allocated in scope B, it will be a temporal error for it to be accessed in a scope which is not either B or enclosed within B.

Thus, each basic block keeps a IN, and OUT for all scopes, including its own scope. When a memory cell is GEN’d it is only GEN’d for its native scope and its native scope’s descendants.

As in RSCR, the bulk of the work for Redundant Temporal Check Removal is done in a dataflow analysis step, with a second step for block-by-block analysis and check removal.

```c
int main(){
  //Scope A
  while(1){
    //Scope B
    if(true){
      //Scope C
    }
    //Scope B
  }
  //Scope A
}
```

Figure 3-5: Skeleton code that displays scopes. B is a child of A, and C is a child of B

### 3.5.1 Dataflow Analysis

As stated above, Redundant Temporal Check Removal is like RSCR, except each basic block has a IN and OUT set for each scope. Each basic block’s scope’s set’s entries are, as in RSCR, a buffer followed by one or more indices.

Its KILL set entries are also the same as RSCR: a single variable which has been invalidated.

Redundant Temporal Check Removal and Redundant Spatial Check Removal have exactly the same GEN and KILL functions. Specifically, since free is a function which
the compiler can’t find a body for, any buffer which is freed will be killed as a matter of course.

MERGE INPUTS function is also quite similar to RSCR’s, except this time it is done per-scope. For a given block, \( B \), for each scope, \( S \), \( B \)’s IN set for \( S \) is the intersection of the OUT sets for \( B \)’s predecessors for scope \( S \).

The TRANSFER function is the main place where the different IN and OUT for each scope come into play. We still remove all IN entries which have an index that is in the KILL set, but now, instead of indiscriminately adding all GEN entries to all scopes’ IN sets, we only add the GEN entries to the scopes’ IN sets where the scopes the same as that of the basic block we are transferring from, or else one that scope’s descendants Thus, in Figure 3-5, if we GEN a variable in scope \( B \), it will be added to the IN set in the if(true) block, but not to the IN set of the basic block after the end of the while loop which is scope \( A \).

### 3.5.2 Block-Level Analysis And Removal

Once we have our dataflow analysis, each block only looks at the IN set for its scope. Block-Level Analysis then proceeds identically to that of RSCR.

### 3.5.3 End Result

After this pass, already-performed temporal checks will have been removed, allowing the LLVM optimizer to remove the metadata loads they have been relying on. With Redundant Temporal and Spatial Check Removals together, the average runtime overhead is reduced from 192% to 172%, removing about 11% of the total SoftboundCETS overhead. Several of the benchmarks get no speedup, while the largest speedup goes to himmer. Its overhead is reduced from 400% to 300% a 25% decrease.
3.6 Constant Array Check Reduction

Constant Array Check Reduction is necessary mostly because of how LLVM deals with struct member accesses. Specifically, given code like Figure 3-6a, LLVM will generate Figure 3-6b. As you can see, to store to a struct member, LLVM first looks up the address of that member using a GEP instruction. Specifically, if we look at the second GEP in Figure 3-6b, we see \( \%y = \text{getelementptr inbounds} \ %\text{struct.point} \* \ %p, \ i32 \ 0, \ i32 \ 1 \)

What this is doing is: starting at the address of \( \%p \), it first looks up the 0th element of \( \%p \). \( \%p \) is a pointer, an unlike in C, LLVM requires that pointers be explicitly indexed—\texttt{load (GEP \%p, 0)} in LLVM is equivalent to \( \ast \%p \) in C. Thus, the \( i32 \ 0 \) is getting the address of the struct, and \( i32 \ 1 \) is getting the address of the 1th (second) element of the struct. \( \%x \) is then set to the address of the 1th element of the 0th element of the pointer.

When SoftboundCETS sees the store to \( \%x \), it looks for the GEP that determines \( \%x \), and checks it. Which means in this particular example, there will be three sets of checks—one for \( p.x \), one for \( p.y \), and one for \( p.z \).

**Parameters:** a function \( F \)

**Returns:** \( F \), with constant array checks reduced

**for Instruction I in F do**

**if I is a marked load or store then**

\( \text{buff} \leftarrow \text{getBufferBeingAccessed}(I) \)

\( \text{idxs} \leftarrow \text{getIndexesBeingUsedToAccess(buff)} \)

**if All idxs are constant then**

\( \text{size} \leftarrow \text{getSizeOfBuffUnindexed(buff)} \)

\( \text{farthestToCheck} \leftarrow 0 \)

**for \( \langle idx, i \rangle \) in \( \langle idxs, \text{range(len(idxs))} \rangle \) do**

**if idx < size then**

\( \text{size} \leftarrow \text{getSizeOfBuffAtIndex(buff, idx)} \)

**else**

\( \text{farthestToCheck} \leftarrow i \)

**end**

un-mark \( I \)

create \( I' \), a load of \( buff \), using \( idxs \) up to \( \text{farthestToCheck} \)

mark \( I' \) for checking

**end**

**Algorithm 3:** Constant Array Check Reduction algorithm.
However, these checks are unnecessary – given the type information for \%p, we know that it has a field for each \texttt{x}, \texttt{y}, and \texttt{z}. Thus, all of these checks can, instead of checking \texttt{p.x}, \texttt{p.y}, and \texttt{p.z}, we can simply check for \%p. See Algorithm 3 for the algorithm used.

This by itself changes nothing; we still have three sets of checks. However, as all three checks are now looking at the same thing, Redundant Spatial and Temporal Check Removals will remove two of the checks.

Unfortunately, as alluded to in § 2.2, this optimization doesn’t actually work on pointers to structs—because of arbitrary typecasts, it is entirely possible for a pointer to point to a block of memory too small for its type. The correct version of this optimization (which is actually much more general, and would probably give more speedup) would group all accesses to a given buffer where the buffer has not changed, and check the minimum and maximum index accessed, but I have not yet had time to implement this. See § 4.1.4 for more details on how this will work.

Luckily, Constant Array Check Reduction does work for arrays—because a pointer cannot be cast to an array, if LLVM can find that a value points to an array allocation of a specific size, this kind of optimization is, in fact, safe.

### 3.6.1 End Result

Constant Array Check Reduction removes, on average, 22% overhead—the average runtime overhead (when combined with Redundant Check Removal) goes from 172% to 148%. This reduction is 24% overhead, or 12.5% of the total SoftboundCETS overhead. Constant Array Check Reduction (safely) removes 167% (from 277% to 111%, a 60% speedup) overhead from milc; milc stores its points as arrays of length 3, and accesses then by hardcoded index. Constant Array Check reduction removes all checks on these accesses. It also provides substantial speedup for mcf, while doing very little for most other benchmarks.
typedef struct {
int x;
int y;
int z;
} point;

void zeroPoint(point* p){
    // <spatial/temporal check of p->x>
p->x = 10;
    // <spatial/temporal check of p->y>
p->y = 12;
    // <spatial/temporal check of p->z>
p->z = -3;
}

(a) The original C code

(b) The generated LLVM IR

Figure 3-6: A simple C program with a struct access, perfect for Constant Array Check Reduction

3.7 Loop Hoisting

Loop hoisting is a standard compiler optimization, and LLVM already does a great deal of loop hoisting in general.

Loop hoisting is simply the process of removing code from a loop and making it run either before the loop or after. See Figure 3-7 for an example of a simple loop hoist. LLVM is good at hoisting loop invariant code; as in Figure 3-7 if a value of a load or store doesn’t change in a loop, LLVM will generally move it to the top or bottom of the loop.

This pass does something slightly different—it moves entire spatial/temporal checks (and their associated metadata loads) from a loop body to outside the loop, even if
the accesses they are checking are not loop invariant.

```
int main()
{
    int k;
    for(int i = 0; i < 1000; i++)
    {
        k = 1000;
        printf("%d", i);
    }
}
```

(a) Original code before the loop hoist.

```
int main()
{
    int k = 1000;
    for(int i = 0; i < 1000; i++)
    {
        printf("%d", i);
    }
}
```

(b) Code after the loop hoist

Figure 3-7: A simple example of a loop hoist that LLVM will do. The assignment to 
k is moved from the loop body to the loop header.

### 3.7.1 Process

Loop Hoisting makes heavy use of the LLVM Scalar Evolution library (SCEV) [SCE16].
SCEV is a library built for loop modifications that supports several different operations. First, and most importantly, it can, given a variable which changes predictably in a loop, create an expression which has the variable’s start value and how much it increments each loop iteration. Second, it can perform operations on these values: addition, multiplication, division, as well as max and min. And, lastly, SCEV can generate code to compute these expressions and insert it into the LLVM IR.

Given the SCEV library, there are still two main challenges to solve regarding loop hoisting:

First, we need to, for each loop, figure out the maximum number of loop iterations (The max tripcount of the loop). The SCEV library contains a function to do this, but it only succeeds on extremely simple loops (As far as I can determine, only constant loops). My implementation works on loops with more complex end conditions, by providing only an upper bound—while the SCEV library will give the maximum
tripcount the loop can actually take during runtime, I am slightly more relaxed. I simply get an upper bound on that number (it's possible that my tripcount is actually impossible for the loop to ever hit). However, in practice, I have not found anywhere where this approach gives false positives in loop hoisting.

Second, we need to figure out, for each buffer accessed in the loop, what the minimum and maximum indices accessed are, and check if they are in bounds. This cannot, in general, be computed at compile time (variable values aren't known). Thus, we need to insert code to, at run time, calculate the largest and smallest access to each buffer, across all loop iterations, and then check to make sure that both of them are in the bounds of the buffer.

Calculating loop tripcounts (number of loop iterations) turns out to be the most complex one; at the point that SoftboundCETS runs on a program, the original loop structure is lost. The LLVM IR consists of instructions in basic blocks, which have successors and predecessors. There is no loop primitive. Loops are objects in LLVM, and there is an analysis pass which gathers loop information, but neither of these have anything like a loop tripcount, and while the SCEV library in theory has one, it is limited to extremely simple loops.

My approach for finding the maximum loop tripcount was something like the following:

1. Figure out the conditions on variables, which, if true, means the loop will exit next iteration. This takes the form of some sort of boolean expression along the lines of \((a > 4 \vee (\neg(a > 4) \land (b > 2)))\)

2. Simplify said condition into a simplified OR statement.

3. If any of the OR clauses have a single expression, \(E\), we know the loop will terminate one iteration after that expression is true.

4. Get the SCEV expression of the variable in each \(E\), and calculate how many loop iterations it will take for it to be true. Take the minimum. This is the maximum tripcount.
Of special note here is that this maximum tripcount is an upper bound. It is entirely possible that one of the other clauses of simplified OR (we ignore clauses with an AND in them) will be satisfied first and the loop will terminate early. This means that, in theory, this method for loop hoisting could cause false positives (could cause SoftboundCETS to find a memory error when there isn’t one). The solution to this would be to do loop splitting, as described in § 4.3.3. However, in practice, I have not found this to be an issue, which is why loop splitting has not yet been implemented.

These steps are rather complex in themselves, so here I will break down how each of them works:

**Exit Condition Extraction**

At its most basic level, what I did for Exit Condition Extraction was to create a single multi-clause \(\lor\) expression which, if it was satisfied, the loop would exit.

For said \(\lor\) expression, each clause is derived from a single exit block (a block which, after it finished, would exit the loop). What we ultimately want each clause to be is the precondition necessary to get to one exit block. Deriving the preconditions for a given exit block is a recursive process that runs something like this:

To find the precondition for a given block, \(B\), we make a \(\lor\) statement from the preconditions on the block’s predecessors. For each predecessor, \(D\), we take its precondition, \(P_D\) (computed recursively). If \(B\) is \(D\)’s only successor, or if \(D\) has a true-false branching and \(B\) is the true branch, then add \(P_D\) as a clause of the \(\lor\) statement. If \(D\) has a true-false branching and \(B\) is the false branch, add \(\neg P_D\) as a clause of the \(\lor\) statement.

The precondition for the loop header block is just \texttt{true}, and if a block has more than two branches, the preconditions of its successors is just \texttt{T} (Top: we know we don’t know what it is). There is additional bookkeeping needed to make sure that we don’t infinite loop: we only visit a given node once—if we hit it again and don’t have a precondition, we return \texttt{T}, since we have no idea what it should be.

The algorithm for the above can be found in Algorithm 4:
Parameters: a basic block B, a basic block H, the loop head
Returns : the conditions that must be true in order for B to execute

Function getPredicate(B) =
begin
  args ← map((getConditionForPredToNodePath node), [B.preds()])
  OR ← new OrExp (args)
  return OR
end

Function getConditionForPredToNodePath(node, pred) =
begin
  if node dominates pred then
    return FalseExp
  if node is pred’s false branch then
    if pred == H then
      return pred.getFalseCondition()
    else
      P ← getPrecondition(pred)
      return new AndExp (P, pred.getFalseCondition())
  if node is pred’s true branch then
    repeat above but for true branch
end

Algorithm 4: Algorithm for loop exit condition extraction.

While this, in theory, works on everything, it generates unacceptably large expressions in many cases—as we will see in the next section, simplification is worst-case exponential. So in order to be able to find tripcounts for as many loops as possible, we want our initial expression to be as small as possible. As generated statement size is roughly linear with the number of basic blocks, loops that are too large are unhoistable. In many cases, one of the worst causes of larger initial expressions is nested loops, and, unlike large if-statement chains, nested loops are relatively easy to optimize away. Thus, whenever a block is visited that is the sole exit block of a sub-loop, we assume the sub-loop will exit eventually, and take as its precondition the precondition of its sub-loop’s header. We completely skip the middle of the loop.

A similar optimization could be done for large branching if-statements. If there are possible conditions that all end in the same spot (before the loop exit), it should be possible to go immediately from the loop exit to the top of the if-statement. See § 4.3.2 for details.
Exit Condition Simplification

At this point, we have a large, rather unwieldy boolean expression tree, with the root an OR node, and the leaves conditions on program variables. We want to extract specific conditions on specific variables which will cause the loop to exit, so that we can get an upper bound on the number of times the loop can run. Unfortunately, due to the way LLVM generates basic blocks, and the above algorithm, very simple relationships between these variables are not obvious.

For example, look at Figure 3-8a. From the original C code, it's clear that, if either \( i \geq a \) or \( j \leq 80 \), the loop will terminate. However, if we look at the CFG generated by LLVM in Figure 3-8b, and run the Exit Condition Extraction algorithm, we can see that we get \( \neg(i < a) \lor ((i < a) \land \neg(j > 80)) \). Unfortunately, in the final tripcount analysis, we cannot use clauses that have \( \land \) in them. We need conditions on specific variables which, if true, will mean the loop has ended. As each condition requires non-trivial calculation, we quickly run into the point where the overhead added from the calculation is more than the overhead saved from hoisting if we work with clauses containing an \( \land \).

Since we cannot use clauses in our \( \lor \) statement that have \( \land \) in them, we miss out on the \( \neg(j > 80) \) condition for loop ending. What we want is the statement \( \neg(i < a) \lor \neg(j > 80) \) Thus, we need a boolean simplifier.

In order to get a boolean simplifier, I first looked at open source boolean simplifier libraries for C++. Unfortunately, I did not find anything that allows the specification of an arbitrary output format—I specifically needed my boolean clause in the form where the root node was an OR, the leaf nodes were atoms, and there were as few atoms as possible, which did not seem to be any of the standard formats. Also, all of the boolean solvers I found were embedded in symbolic algebra systems, which were huge and clunky and extremely complex. Luckily, I had a boolean simplifier rules system which I was using for 6.945 [Sus16], the class I was TAing this term. So I ported it from MIT scheme into C++.

My port is messy (It turns out that scheme is much, much easier for this kind
void and_loop (){
  int* buff;
  int a, j;
  for (int i = 0; i < a && j > 80; i++)
  {
    j++;
    buff[i] = 7;
  }
}

(a) Original C code

(b) CFG of the LLVM IR for Figure 3-8a

Figure 3-8: A loop with an and statement in its conditional
of thing than C++), but it works. Given more time, I would have made it more extensible, but as it is now, it is hardcoded for the specific boolean class I was using.

Basically, what my port does is: Given a boolean expression, which can be an AND, OR, NOT, PRIMITIVE (contains an LLVM value), TRUE, FALSE, or TOP, it attempts to match it to a list of transformations (rules), repeatedly applying the first rule that fits until no rule fits. The transformations are of the form \(<\text{pattern}>\rightarrow<\text{result}>\), where pattern has two added types from the base expression: variables and segments. (Denoted in future as \((? <\text{name}>)\) and \((?? <\text{name}>)\), respectively. A variable matches any single expression, but must be constant across all uses. A segment matches any number of expressions (including zero), and, similarly, if referenced multiple times, must refer to the same thing each time.

A result is in terms of the same types as a pattern, but where the pattern attempts to match its variables and segments against a given boolean expression, the result takes the assignments from the pattern and outputs a new expression. Thus, as an example, when applying the rule \([\neg(\neg(\ ? a))] \rightarrow [(\ ? a)]\) to the expression \(\neg(\neg(i > 4 \land j == 10))\), \((\ ? a)\) matches \(i > 4 \land j == 10\), and the output is the value of \((\ ? a) = i > 4 \land j == 10\).

As this is an exponential process, I have a first step in the rule system which counts the number of clauses it is checking and immediately gives up if it is more than 150 (the experimental limit I found for when compilation began to take an unacceptable amount of time). Also, rules must be chosen carefully to avoid the possibility of an infinite cycle of “simplifications”.

In the original Scheme program, all rules were simply lists, which meant that \(\land\) and \(\lor\) were, themselves just symbols. This allowed a much more broadly applicable matcher, as it works on arbitrary symbols. However, as I already had the AND/OR/etc... classes being generated, I opted to just match on them. If I were to redo this, I would have the matcher just work on lists.

The rules I ended up using were the following:

**Remove redundant clauses**
\((?? a) \land (? x) \land (?? b) \land (? x) \land (?? c)\) \rightarrow ((?? a) \land (?? x) \land (?? b) \land (?? c))

\((?? a) \lor (? x) \lor (?? b) \lor (? x) \lor (?? c)\) \rightarrow ((?? a) \lor (? x) \lor (?? b) \lor (?? c))

**And simplification**
\((T \land (? s)) \rightarrow T\)
\((F \land (? s)) \rightarrow F\)
\((T \land (? s)) \rightarrow (? s)\)

**Or simplification**
\((T \lor (? s)) \rightarrow T\)
\((F \lor (? s)) \rightarrow (? s)\)
\((T \lor (? s)) \rightarrow T\)

**Negate simplification**
\(\neg T \rightarrow T\)
\(\neg (\neg (? v)) \rightarrow (? v)\)
\(\neg T \rightarrow F\)
\(\neg F \rightarrow T\)

**A and NOT B simplification**
\((?? a) \lor (? y) \lor (?? b) \lor ((? x) \land \neg (? y)) \lor (?? c)) \rightarrow ((?? a) \lor (?? b) \lor (?? c) \lor (? x) \lor (? y))\)
\((?? a) \lor \neg (? y) \lor (?? b) ((? y) \land \neg (? x)) \lor (? c)) \rightarrow ((?? a) \lor (?? b) \lor (?? c) \lor (? x) \lor (? y))\)

This, along with enforcing an ordering \((F < T < \land/\lor/\neg, \land/\lor/\neg \text{ sorted by how many layers deep they ran})\) on the terms of an \(\land\) and \(\lor\) statement, and merging \(\land\) and \(\lor\) (so \((\lor (? a)(?? b)) \rightarrow (\lor (? a)(?? b))\) ) seems to be enough to get everything which I was generating into a nice \(\lor\) statement with as few repeats as possible.

I didn’t do sort and merge as rules because sort is extremely slow as a rule, and merge was easy to do as part of the boolean expression creation process.
Getting the Tripcount

Once we have a simplified boolean expression, getting an expression for the upper bound on the tripcount of the loop is relatively easy.

First, we filter the clauses, looking for clauses of the form `<condition on variable>` or `¬<condition on variable>`. If a statement is a not-statement, we simply change the operation of the conditional. The other possibilities are an `∧` statement or and `∨` statement, and we know it’s not a `∨` statement because we merge `∨` statements together. If a clause is an `∧` statement, we can’t use it—we want single conditions on variables which are guaranteed to make the loop exit.

Once we have a list of conditions of the form `<LHS OP RHS>`, for each condition, we get the SCEVs for `LHS` and `RHS`. If exactly one of them is an AddRecExpr (an expression which starts at a specific value, `b`, and increments by a specific value, `i` each loop iteration), then we do some simple math based on what `OP` is to get the number of times the loop can repeat before the equality or inequality is satisfied—if `OP` is `==` (and `i=1`), the number of repetitions for the loop to be satisfied is just `E - S`, where `S` is the starting value, and `E` is the value it needs to equal. If `OP` is `<` or `>`, then the repeat count is `(E - S)/i`, assuming the loop will terminate. `OP` is never anything else, thanks to compiler simplifications.

Once we have SCEVs for each condition, we use the SCEV library to get a SCEV which is the minimum of all of them. This is the SCEV expression of the loop tripcount.

Performing the hoist

Once we have a maximum tripcount for each loop, the rest of our job is relatively simple. We go through the loop, and, for each buffer which is checked, use the SCEV library to get a SCEV representing both the minimum access to the buffer, and the maximum access to the buffer. Because numerical overflow is a possibility, we create an expression which takes the min/max of all accesses and all accesses multiplied by the loop tripcount.
Given this SCEV, we use an LLVM builtin function (SCEVExpander::expandCodeFor) to generate code which will calculate it at runtime and insert it into the loop pre-header. However, before we can, we have to make an important safety check—we can only do this if all variables that appear in the SCEV are defined outside the loop. (The SCEV takes into account if the variables change predictably, and if they change unpredictably, we can’t get a SCEV anyway, so the point is moot). Specifically, this means that check hoisting is highly dependent on loop induction variables being hoisted/collapsed into PHI nodes before it runs. Unoptimized code cannot be loop hoisted, because the value incremented each loop iteration will be loaded and stored back into its original variable, as will the value being checked as the loop condition. And LLVM cannot track relationships between different loads of the same variable.

For an example of what I’m talking about, look at Figure 3-9. From a look at the C code, it’s clear that i is the induction variable. However, the unoptimized LLVM code that will be generated is shown in Figure 3-9b. If we look at for.cond, the block where the loop exit condition actually lives, we can see that the exit condition is \( \%\text{cmp} = \text{icmp slt} \ %1, 1000 \) — it’s in terms of \( \%1 \), a new load of \( \%i \). If we take the SCEV of \( \%\text{cmp} \), we get an expression in terms of \( \%1 \). Because this is a new load of \( \%i \), the SCEV library knows nothing about its recurrence — I believe this is because it would be impractical to trace all accesses and assignments to \( \%i \) across the entire function. Thus, we can’t hoist anything above where \( \%1 \) is defined. As \( \%1 \) is defined in the loop body, this makes hoisting impossible.

However, after the standard suite of LLVM loop optimizations runs, we end up with Figure 3-9c. The loop exit condition is now in for.body, and is in terms of \( \%3 \). However, \( \%3 \) is not a new load of \( \%i \), but rather a cast of \( \%\text{indvars}.\text{iv.next} \) (See line 21 of Figure 3-9c. And \( \%\text{indvars}.\text{iv.next} \) is just \( \%\text{indvars} + 1 \). And \( \%\text{indvars} \) is a PHI node (See § 3.2.1 for more on PHI nodes) which starts at \( \%2 \), which is a cast of \( \%1 \). \( \%1 \) is a new load of \( \%i \), so the SCEV stops tracing there. So the SCEV ends up being in terms of Figure 3-9c’s \( \%1 \), which is defined in entry, far above the loop header. Thus, Figure 3-9c is hoistable. While this is not normally a large concern (code is usually optimized with the standard LLVM O3 suite before hitting my optimizations),
it makes debugging loop hoisting difficult, as debuggers are often not great at dealing with optimized code).

Unfortunately for our purposes, loops that rely on global variables tend not be be optimized in such a manner. See § 3.9 for more details on dealing with global variable hoists.

```c
int main()
{
    int j[1000];
    int k;
    // i is the loop induction variable
    for(int i = k; i < 1000; i++)
    {
        j[i] = 0;
    }
}
```

(a) Original C code

```assembly
; @k = external global i32
; Function Attrs: nounwind uwtable
define i32 @main() #0 {
    entry:
        %retval = alloca i32, align 4
        %j = alloca [1000 x i32], align 16
        %i = alloca i32, align 4
        store i32 0, i32* %retval
        %0 = load i32* @k, align 4
        store i32 %0, i32* %i, align 4
        br label %for.cond
    for.cond: ; preds = %for.inc, %entry
        %cmp = icmp slt i32 %1, 1000
        br il %cmp, label %for.body, label %for.end
    for.body: ; preds = %for.cond
        %2 = load i32* %i, align 4
        %idxprom = sext i32 %2 to i64
        %arrayidx = getelementptr inbounds [1000 x i32]* %j, i32 0, i64 %idxprom
        store volatile i32 0, i32* %arrayidx, align 4
        br label %for.inc
    for.inc: ; preds = %for.body
        %3 = load i32* %i, align 4
        %inc = add nsw i32 %3, 1
        store i32 %inc, i32* %i, align 4
        br label %for.cond
    for.end: ; preds = %for.cond
        %4 = load i32* %retval
        ret i32 %4
}
```

(b) Before LLVM's loop optimizations have run
Figure 3-9: A simple example of a the generated LLVM IR for a C loop, before and after loop optimizations

End Result

Loop hoisting, if performed after Constant Array Check Reduction, reduces the average overhead from 148% to 114%, removing 18% of the original SoftboundCETS overhead. It provides the most speedup to hmmer and ibm (149% and 115% respectively). However, due to fact that calculating loop tripcounts requires some runtime overhead, it adds a few tenths of a percent overhead to several tests. See Figure 3-10 for an example of the added code.

3.8 Shadow Stack Removal

Unlike the other optimizations in this project, Shadow Stack Removal (SSR) is largely a matter of tweaking built-in LLVM functions to get what we want. There is a
(a) The same code from 3-9 after loop optimizations have been run and it has been marked for checks. Note the !SoftBoundsOpt on line 9

(b) After hoisting. Yellow Lines have been added. Note that there is no longer a !SoftBoundsOpt in the loop body (for.body). However, the loop header (for.body.lr.ph) is much larger—it has code to calculate the minimum and maximum reference to %j, as well as too checks to make sure that the maximum and minimum are within bounds.

Figure 3-10: A LLVM IR loop, before and after loop hoisting

hard version (gets better results, but dramatically increases compilation time) and a soft version of Shadow Stack Removal (less speedup, but minimal compilation time increase). They are the similar, except the hard version has a few more steps, and renders compile times prohibitively slow due to massive code size increase (>10x). In
future, it would be nice to write a compromise between the two, but I have not had time to do this.

Also, unlike most other optimizations in this project, SSR runs after linking. While it could theoretically do some good before, it works much better if run after linked inlining happens—it removes overhead from inlined functions, so the more inlined functions, the better its results.

As explained in § 2.3.5, SoftboundCETS uses a special stack (the "shadow stack") to pass necessary pointer metadata for the parameters of functions. If a function is inlined, this isn’t necessary—inlining can replace all uses of the parameter with the value it’s assigned to. Unfortunately, LLVM is not smart enough to remove the shadow stack functions which are passing the metadata around—they write to global variables. LLVM treats all global variables as volatile. Thus, we want to manually remove these unneeded shadow stack operations.

The basic layout of a function call with shadow stack operations is something like Figure 3-11

3.8.1 Soft Shadow Stack Removal

First, we mark all SoftboundCETS functions pertaining to loading and storing from the shadow stack (loads and store to/from the shadow stack for base, bounds, key, and lock) with noinline.

Next, we run LLVM’s 03\textsuperscript{1} pass. This, among other things, inlines functions which would normally be inlined (but not the shadow stack functions—they’re marked noinline).

Note that this is after linking, so cross-file inlining is done. We now have a file where everything except the shadow stack functions are as inlined as they are going to get. Our example from Figure 3-11 now looks like 3-12. As you can see, the shadow stack operations here are pointless—values are written to the shadow stack and then immediately read off again. And because the shadow stack internals use a global variable to keep track of the top of the stack, LLVM will not optimize this away on

\textsuperscript{1}Optimization Level Three

73
its own, even if the shadow stack functions are inlined.

From here, we iterate through each function body. Whenever we come to a shadow 
stack store operation, \( S \), we iterate through the basic block it’s a part of. If we find 
another non-shadow-stack function call or a call to deallocate shadow stack space, 
we abort—the function where \( S \)’s corresponding load, \( L \), is was not inlined. This 
case looks like Figure 3-11b; instead of running into a corresponding load for our 
stores, the first thing we run into is just the call to increment. If, however, in- 
lining has happened (as in Figure 3-12) and we find a corresponding load to the 
store, then everywhere the loaded value is used, we can simply use the stored value. 
We can then remove the shadow stack loads and stores. This gives us Figure 3-13.

```c
void increment(int* i){
    *i += 1;
}

int main(){
    int i = 4;
    increment(&i);
}
```

(a) Original C code

```c
define i32 @softboundcets_pseudo_main() {
    entry:
        %lock_alloca = alloca i8*
        %key_alloca = alloca i64
    call void @__softboundcets_stack_memory_allocation(i8** %lock_alloca , i64* %key_alloca)
    %lock.load = load i8** %lock_alloca
    %key.load = load i64* %key_alloca
    %0 = call i8* @__softboundcets_get_global_lock()
    %i = alloc i32, align 4
    %il = bitcast i32* %i to i32*
    %bitcast2 = bitcast i32* %tmp to i8*
    call void @__softboundcets_allocate_shadow_stack_space(i32 2)
    call void @__softboundcets_store_base_shadow_stack(i8* %bitcast2 , i32 1)
    call void @__softboundcets_store_bound_shadow_stack(i8* %bitcast2 , i32 1)
    call void @__softboundcets_store_key_shadow_stack(i64 %key.load , i32 1)
    call void @__softboundcets_store_lock_shadow_stack(i8* %lock.load , i32 1)
    call void @increment(i32* %i) #1
    call void @__softboundcets_deallocate_shadow_stack_space()
    call void @__softboundcets_stack_memory_deallocation(i64 %key.load)
    ret i32 0
}
```

(b) The generated IR code for `main()`. This is the caller’s perspective: shadow stack space is 
allocated, values are stored to the shadow stack, the function is called, and then the shadow 
stack space is freed.
The generated IR code for `increment()`. This is the callee’s perspective: base, bound, key, and lock are loaded off the shadow stack and then used for checks.

Figure 3-11: A simple example of a function call using SoftboundCETS’ shadow stack.

In theory, it would be possible to remove the allocate_shadow_stack_space and deallocate_shadow_stack_space, but in most non-trivial examples, this gets much harder, as allocate and deallocate will be on opposite sides of the inlined function, which could be massive/have arbitrary control flow in the middle. It would be possible to do by tagging each allocation/deallocation pair as a pair before inlining, but it’s complex enough that it should probably be a separate pass, which I have yet to write. Also, the overhead from shadow stack allocation/deallocation is relatively small compared to the overhead from shadow stack loading/storing.

### 3.8.2 Hard Shadow Stack Removal

Hard shadow stack removal works identically to Soft Shadow Stack Removal, except it forces all functions to be inlined as much as possible. This gives roughly 2x the speedup of Soft Shadow Stack Removal, but it dramatically blows up code size, and increases compile time of some tests to >8 hours. Ideally, I would write an inline pass which inlines more than LLVM default but less than Hard Shadow Stack Removal is currently doing, but I have yet to do so.
Figure 3-12: After inlining, our example from Figure 3-11 looks like this. For brevity, I have left all softbound functions uninlined, although in a real run all non-shadow-stack functions including dereference checks would be inlined. The yellow highlighted lines will be removed.

### 3.8.3 End Result

Shadow stack removal is highly variable. For most applications, it does little-to-nothing. However, there are benchmarks (milc and equake) where it makes a 10%-20% difference.

### 3.9 Global Variable Hoist

Global variable hoist is a simple optimization, which is, in theory, unsafe. As discussed in Loop Hoisting, Loop Hoisting needs all variables which the exit condition relies on to be either PHI nodes or to be hoisted out of the loop. Unfortunately, LLVM
Figure 3-13: Once we’ve actually run shadow stack removal, our example from Figures 3-11 / 3-12 looks like this. Note all the shadow stack loads and stores are gone

often does not hoist global variables when it would absolutely be able to. I suspect it is because, in a concurrent program, hoisting global variables is not safe. As an example of what I mean, see Figure 3-14a. When generating optimized code, it gives Figure 3-14b. Since the comparison is between %2 (ultimately dependent on nothing – it’s just 0, plus 1 for each iteration) and %3 (a load within the loop body), nothing can be hoisted—we cannot hoist if the would-be-hoisted SCEV is reliant on a load in the loop body.

Thus, global variable hoist comes in. It looks through all loads in a loop, and, if there are no writes to the loaded variable within the loop, hoists it. This gives an average of 5% speedup, by allowing additional loop hoists.
(a) Original C code

```c
int 1;
int main(){
    int j[1000];
    for (int i = 0; i < 1; i++){
        j[i] = 0;
    }
}
```

(b) The generated LLVM IR. Note that %2 is a new load of @l, which makes hoisting impossible.

Figure 3-14: A simple example of generated LLVM IR involving global variables

3.10 Optimization Effectiveness

Here, I will briefly go over each benchmark: What it does, how much overhead it started with, and how much overhead my implemented optimizations knock off.

**bzip**

Bzip is a compression program. My compilation of it started with very little overhead (7%), and that overhead is largely unremovable.

**sjeng**

Sjeng is a chess-playing program. It has 113% overhead unoptimized, and is
relatively difficult to optimize—most of its overhead is from loops where the loop induction variable is not be used as an index. Rather, the loop induction variable is used as an index into a different array, and that returned value is used as an index. It is a fantastic candidate for Array Prescanning (see § 4.1.3). Redundant Check Removal took off 3%, while Constant Array Check Reduction and Loop Hoisting took 10% off each. The Constant Array Check Reduction is from struct accesses, and the loop hoisting comes from removing the few loop induction variable loads that show up in sjeng’s many loops. The final optimized overhead is 92%.

**Ibm**

Ibm simulates incompressible fluids. It starts with 120% overhead, but all of this overhead is contained within one large but simple loop, and 99% of that is removed by loop hoisting. When optimized, it has no overhead.

**milc**

Milc is a quantum chromodynamics simulation. When run with SoftboundCETS, it starts with 277% overhead. A large amount of this overhead is both within relatively simple loops and is from constant accesses of arrays of constant size (milc uses arrays of three elements to indicate points). Of its 227% overhead, 167% is removed by Const Array Check Reduction. A further 20% is removed by loop hoisting.

Also, milc is one of the few benchmarks to make heavy use of function calls within its inner loops. Soft Shadow Stack Removal removes 10% of its overhead, while hard shadow stack removal (with its forced inlining) removes closer to 20%, but increases compilation time to several hours. Because some of milc’s loops are compared against global variables, an additional 10% overhead is removed by Global Variable Hoisting. This gives a total optimized overhead of 72%.

**equake**

Equake simulates earthquakes. Unoptimized, equake has 150% overhead. It has a fair amount of repeated accesses to the same array locations, so 30% is removed by Redundant Check Removal. It has no constant array checks, and gets a 10% speedup.
from loop hoisting, plus an additional 5% speedup when global variables are hoisted. Shadow Stack Removal gives 6% speedup, for a total optimized overhead of 97%

sphinx
Sphinx (technically sphinx3) is a speech recognition program from CMU. It starts with 183% overhead. 80% overhead is removed by loop hoisting—sphinx has several easily hoisted loops where much of its overhead happens in the non-optimized case. Constant Array Access and Redundant Check Removal remove approximately 9% each, for a low of 85%. Global Variable Hoisting actually increases its overhead by a percentage point, as the hoists cause overhead and do not allow any extra loop hoists.

ammp
Ammp runs molecular dynamics on specific biological macromolecules. It clocks in at 267% overhead, and is relatively unaffected by my optimizations—even with all of them, it is still at 254% overhead. This is due to the fact that most of ammp’s overhead comes from iterating over linked lists, which none of my current optimizations work well with. Luckily, these iterations are on local variables and already have programmer-inserted checks, so Manual Check Analysis would be able to dramatically reduce this overhead.

hmmer
Hmmer is the benchmark that I looked at with the most unoptimized overhead at 400% unoptimized. It is a gene sequence search, and it relies on a large amount of repeated accesses to the same addresses—105% overhead is removed by Redundant Check Removal. Const Array Checks does relatively little (10%), but loop hoisting removes another 150% overhead, for an optimized overhead of 137%. Unfortunately, the rest of hmmer overhead is from checks which are probably necessary; the only way to get overhead much lower is to start marking portions of hmmer as trusted.

gobmk
Gobmk is a go-playing AI. It was one of the hardest benchmarks to compile, and I ended up having to blacklist two of its functions to get it to run with Softbound-CETS. Go only has 53% overhead, but my optimizations don’t make a large change in its runtime—they take it from 53% to 52%. The overhead is mostly spread around the program in hard-to-optimize snippets, so the only way to decrease it further is large amounts of trusted code sections.

Art

Art is a neural net used to recognize thermal images. It starts at 65% overhead unoptimized, and Constant Array Check Reduction, Loop Hoisting, and Redundant Check Removal each remove a few percent overhead. Art makes use of function calls in its inner loops, however, and so Shadow Stack Removal gets rid of 10% overhead. And said loops are largely predicated on global variables, so while Loop Hoisting did relatively little, Loop Hoisting with Global Variable Hoist removes an additional 10%, for a total overhead of 37%.

Mcf

Mcf schedules vehicle pick up/drop off. Unoptimized, it has 300% overhead, a large portion of of which is in loops with exit conditions based on one struct pointer being equal to another. As loop hoisting cannot determine the bounds on these loops, hoisting does nothing. Constant Array Check Reduction and Redundant Check Removal each remove approximately 50% overhead, and so the total optimized overhead is 195%.

Overall Figure 3-15 shows optimization effectivenesses per benchmark. As is shown, there is substantial speedup for roughly half of the benchmarks, while the other half are largely unchanged. This has mostly to do with how the benchmarks use pointers and arrays. In the benchmarks which access the same array repeatedly in a predictable way, my optimizations were extremely effective. In the benchmarks with
odd loop patterns or lots of pointers being assigned to other pointers, (which causes lots of overhead from metadata copies), my optimizations were much less effective.
Optimization Gain Per Benchmark

- Spatial/Temporal Check Removal
- Const Array Checks
- Loop Hoisting
- Shadow Stack Removal
- Global Variable Hoist
- Unremoved
Chapter 4

Future Work

As explained in § 3.10, my optimizations were extremely effective for some benchmarks. However, for others, they were much less so. I have compiled a list of potential optimizations for the future which would further reduce the overhead of the (particularly unoptimized) benchmarks, and, given my restructuring of SoftboundCETS, will be relatively easy to add. However, even given these, I suspect that the goal of $<15\%$ overhead on all benchmarks is impossible. Thus, I have a further list of potential source code annotations, which, while requiring nonzero programmer effort, would allow overheads to average in the $<15\%$ range. See 4-1 for a graph of the projected results.

Aside from new optimizations, there are a few places where old optimizations could be improved, most notable loop hoisting.

And, lastly, the prototype SoftboundCETS system I am using is not particularly robust—there are many programs that SoftboundCETS simply fails to compile or else cannot be compiled with the full LLVM optimization suite. Luckily, the authors of SoftboundCETS assure me that a new version of the SoftboundCETS prototype will be out eventually which will fix many of the problems in the current prototype.
Optimization Gain Per Benchmark

- Spatial/Temporal Check Removal
- Global Variable Hoist
- Variable Range Annotation
- Better Loop Hoisting
- Const Array Checks
- Unremoved
- Trusted Block Annotation
- Array Prescanning hoist
- Loop Hoisting
- Unremovable
- Constant Loop Removal
- Previously manually checked analysis

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Overhead</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlc</td>
<td>380</td>
<td>360</td>
</tr>
<tr>
<td>sjeng</td>
<td>340</td>
<td>320</td>
</tr>
<tr>
<td>bzip</td>
<td>300</td>
<td>280</td>
</tr>
<tr>
<td>lbm</td>
<td>260</td>
<td>240</td>
</tr>
<tr>
<td>equake</td>
<td>220</td>
<td>200</td>
</tr>
<tr>
<td>sphinx</td>
<td>200</td>
<td>180</td>
</tr>
<tr>
<td>ammp</td>
<td>160</td>
<td>140</td>
</tr>
<tr>
<td>hammmer</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>gobmk</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>art</td>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>mcf</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Mean: 120
Mean no annotations: 100
Mean with annotations: 140
4.1 New Optimizations

4.1.1 Constant Loop Check Removal

I implemented loop hoisting. This allows repeated checks within a loop body to be only performed once. However, in some cases, it's actually possible to remove the loop checks all together if the loop bounds can be determined exactly. See Figure 4-2 for an example of a loop where this can be done. There are relatively few places where such loops appear in practice, and most of the overhead is removed by hoisting, so while this optimization would give small amounts of speedup, its effectiveness is likely to be limited.

4.1.2 Manual Check Analysis

In many places (most notably in the benchmarks ammp and mcf), the original programmers included checks that would detect if a spatial violation was occurring (these generally take the form of if (x == NULL) break;, and were most common for linked list traversals). In this kind of case, ideally SoftboundCETS would figure out that there was no way for a spatial violation to occur and not add any additional checks. In practice, detecting such a thing would be extremely difficult, but possible. I imagine it would work similarly to finding loop exit conditions:

For a given check, look at its basic block. Like in Loop Hoisting (see § 3.7.1 for how this works), figure out which conditions must be true in order for the check to execute. If any of these preclude the necessity of the check, remove the check.

Manual check analysis also requires a CCured-like type inference—programmer-inserted checks test for null pointers, so in a pointer first has to be shown to not be involved in pointer arithmetic. However, this is relatively easy to determine at the local level, and all instances where manual check analysis would be useful that I discovered (ammp and mcf) would be for local variables, so this type inference would not need global analysis. And in certain benchmarks, this optimization could yield a substantial speedup. See Figure 4-3 for an example.
```c
int main(){
    int j[1000];
    for(int i = 0; i < 1000; i++){
        // spatial/temporal check
        j[i] = 0;
    }
}
```

(a) Original C code with the check annotated. There are multiple ways the overhead of this can be reduced

```c
int main(){
    int j[1000];
    for(int i = 0; i < 1000; i++){
        // spatial/temporal check 0, 1000
        j[i] = 0;
    }
}
```

(b) The same code, with the check hoisted. This is what my current Loop Hoisting pass will do. Now, instead of 1000 checks, there is one. However, in some cases (many small, repeated loops), this may not be enough

```c
int main(){
    int j[1000];
    for(int i = 0; i < 1000; i++){
        // temporal check j
        j[i] = 0;
    }
}
```

(c) The same code, after the check is removed. Because we know the array size (1000), and can figure out the bounds on i, we can show statically that there will never be a spatial violation, so all spatial checks can be removed. In this particular example, the temporal check could be eliminated as well, but that would be a different optimization.

```c
int main(){
    int j[1000];
    for(int i = 0; i < 1000; i++){
        // temporal check j
        j[i] = 0;
    }
}
```

(d) Now spatial checks have been removed and the temporal check has been hoisted. This is the end result you would get if Constant Loop Check Removal were added to the existing system.

Figure 4-2: A loop of fixed size, allowing the spatial checks to be entirely removed

### 4.1.3 Array Prescanning Hoist

There are a large number of loops in the benchmarks I looked at that did something along the lines of the idiom shown in Figure 4-4a. Unfortunately, while the first check
while (node->pred) {
  tmp = node->sibling;
  if (tmp) {
    node = tmp;
    break;
  } else {
    node = node->pred;
  }
}

(a) Source code snippet from mcf (mcfutil.c). node->pred cannot be null, so checking it on line 8 is not necessary.

Figure 4-3: A simple case where a pointer is already checked by the programmer can be hoisted out of the loop, all the following ones cannot—the value of idx varies unpredictably across loop iterations, so my loop hoisting pass is unable to figure out the minimum and maximum values for the hoist.

However, the minimum and maximum values are there—since idxs does not change during the loop, we have all possible idx values at loop start. Thus, the checks at lines 5 and 7 are possible to hoist:

We would start by running loop hoist as normal. But when we hit idx being used as an index into a buffer, we would calculate its minimum and maximum values very differently than in normal loop hoisting. In normal loop hoisting, we take the minimum of the minimum of all expressions using it (for a given buffer) and the minimum of those same expressions times the maximum number of loop iterations (without simplification because of the possibility of overhead). The same thing is done to find its maximum value, except taking maxes instead of mins.

Array Prescanning Hoist takes this one step further. In our example in Figure 4-4a, we first find the minimum and maximum values for i, using the loop hoisting method from above. We then walk through the buffer idxs. idxs’ min/max would be the minimum/maximum of all values in the buffer between i_min and i_max. This allows several benchmarks (most notably sjeng) to have substantially better loop hoisting.

There is a major hazard of this kind of optimization: it is entirely possible for it
to add substantial overhead instead of removing any. If the array which needs to be
scanned is large enough and the number of checks avoided small enough, calculating
the minimum value of the array may be more expensive than any savings from
removing the checks.

4.1.4 Check Batching

Check batching is already done as part of loop hoisting, but only within hoisted
loops. What Loop hoisting does for the actual hoist is that it calculates the minimum
and maximum accesses of each buffer in the loop, checks those, and removes the
intermediate checks.

Check batching takes that idea and applies it universally, regardless of loops.
Basically, the way it works (spatially) is the following:

For each buffer, find the contiguous basic blocks where it is unchanged. Find
the minimum and maximum access to the buffer. Check those, while removing all
intermediate checks.

A similar optimization can be done for temporal checks.

This has a few advantages. First, it should give a relatively large amount of
speedup in places where an array is being repeatedly accessed. And since fields of a
struct being accessed are transformed into the same thing in the LLVM IR, it would
render Constant Array Check Reduction obsolete, which is excellent, as Constant
Array Check Reduction is not sound for use on pointers to structs.

Check Batching’s disadvantage is that, if implemented naively, check batching
can theoretically cause false positives. See Figure 4-5 for an example. The main way
this happens is if there are conditionals related to the size of the buffer which control
which part of the buffer is accessed—batching might take the maximum of all possible
accesses, even though a small buffer might never see a large access. There are two
potential solutions to this: Either Check Batching could be restricted to within basic
blocks, or else code splitting could be employed—if the batched check fails, the code
is run with non-batched checks active. Code splitting has some problems as far as
binary size and caching goes, so an alternate solution would be to live-modify the
void move(int* buff, char** words, int* idxs){
    for(int i = 0; i < 1000; i++){
        int idx = idxs[i];
        buff[idx] = 10;
        words[idx] = "hello";
        ...
    }
}

(a) Original C code with the check annotated. There are three sets of checks, only one of which is based on the loop induction variable.

void move(int* buff, char** words, int* idxs){
    for(int i = 0; i < 1000; i++){
        int idx = idxs[i];
        buff[idx] = 10;
        words[idx] = "hello";
        ...
    }
}

(b) After running normal loop hoisting. The check based on i (the loop induction variable) has been hoisted, but the other two have not

void move(int* buff, char** words, int* idxs){
    idx_min = min(idxs[0], idxs[1000])
    idx_max = max(idxs[0], idxs[1000])
    buff[idx_min], buff[idx_max]
    for(int i = 0; i < 1000; i++){
        int idx = idxs[i];
        buff[idx] = 10;
        words[idx] = "hello";
        ...
    }
}

(c) With Array Prescan Loop Hoisting, the other two checks can be hoisted as well.

Figure 4-4: A loop with checks indirectly based on the loop induction variable. These are doable, but require prescanning arrays.

binary to un-batch the check if the batched check fails. This would get extremely complicated, though, so a simpler solution is probably preferable.
```c
void null_terminate(char* buff, size_t size){
  if(size == 64){
    buff[63] = '\0';
  } else {
    buff[31] = '\0';
  }
}
```

(a) Before Batching

```c
void null_terminate(char* buff, size_t size){
  if(size == 64){
    buff[63] = '\0';
  } else {
    buff[31] = '\0';
  }
}
```

(b) After Batching

Figure 4-5: Imagine that there are two possible buffer sizes here: 32 or 64 characters. In the original case, all possible buffer sizes will be fine. However, in the batched case, a buffer of size 32 will cause a false positive.

### 4.2 Source Code Annotations

So far everything I have implemented or proposed is automatic—all the user needs to do is run the compiler with the correct flags. However, automatic optimizations can only get so far. If we want to get to even lower overhead, we need to add some compiler hints. The two types of annotations that would be most useful are Variable Range Annotation and Trusted

#### 4.2.1 Variable Range Annotation

This annotation would simply consist of a range on a variable (probably an integer) which the variable is guaranteed to stay between for its entire lifetime which is smaller than the bounds of any buffer the variable is being used to index into. This means that any given buffer only needs to be checked once for all accesses from this variable. This is extremely helpful for loops which cannot normally be hoisted for whatever
reason (usually because their bounds are irregular) or in a place where a variable is repeatedly used to index into a variable, changed, used to index in again, etc.

Note that this could potentially be done automatically at runtime. The way it would work is something like the following: Track the values that a given variable has for a substantial period of time. Then automatically add this annotation. If the variable is ever found to not be in its “annotated” range, fall back to the original checks.

4.2.2 Trusted Annotation

The Trusted annotation would simply allow a block of code to be marked as trusted. Trusted code would still need metadata copies, but would be able to eliminate all checks and the metadata loads they rely on. SoftboundCETS already supports blacklisting of individual functions, but many times I found a single loop within a larger function to be a large irremovable bottleneck. Manually checking a specific loop and just marking it as trusted is the ultimate resort of those who want C, security, and low overhead.

4.3 Enhancements of Existing Optimizations

There are several places where existing optimizations could be further improved. Constant Array Check Reduction as it currently exists is not sound in the presence of arbitrary casts, and Loop Hoisting is both somewhat limited in the size of loops it can hoist and has the potential to cause false positives.

4.3.1 Constant Array Check Reduction Soundness

As mentioned in § 3.6, Constant Array Check Reduction (CACR) can lead to false negatives in the presence of arbitrary typecasts. This is not to say it’s completely useless, however. Only pointers can be cast. Thus, if CACR is being run on a struct and not a pointer to a struct or else a pointer which it can verify has not been cast,
it can still be used.

Alternately, instead of all of the extra analysis that making CACR sound would entail, once Check Batching it active, it can be entirely removed—Check Batching does almost exactly the same thing, more broadly, and is sound.

### 4.3.2 Loop Hoisting Size

As mentioned in §3.7.1, Loop Hoisting currently assumes that the precondition of the sole exit block of a sub-loop is the precondition of the header of the loop. This dramatically reduces the load on the simplifier.

However, there are some loops which have many, many basic blocks which are not part of subloops. Instead, they are due to internal loop control structure (Loops with if nests which don’t effect the flow of the loop). These too could be short-circuited the way subloops are. The main task is to find groups of basic blocks with common start and end points. If our precondition hunt hits the end of one of these groups, we can just short-circuit to the top.

### 4.3.3 Loop Splitting

There are several ways that loop hoisting can cause false positives, in a similar vein as check batching—it’s possible for internal control structures in the loop to make sure that a violation never occurs, even though a hoisted check might see a violation. The solution to this is loop splitting—if a hoisted check is failed, instead of aborting the program, the loop executes with the normal runtime checks in place. Once again, it would be possible to live-patch the application to un-hoist the loop, but that would be extremely complex and probably not worth it. The only reason normal loop splitting has not been implemented yet is that I haven’t run into a need for it; none of the benchmarks I’ve used have had false positives from loop hoisting.
4.4 Updating To The New Version of SoftboundCETS

As mentioned above, the current prototype of SoftboundCETS I am using is not particularly robust. It successfully compiles 50% of the SPEC2006 C benchmark suite, and less of SPEC2000. A new version of SoftboundCETS is coming out eventually, which will hopefully fix many of these compatibility issues. Unfortunately, this update will cause issues with my optimizations—The current version of SoftboundCETS (and thus my optimizations) is for LLVM 3.4. The new version will be for LLVM 3.9. And LLVM versions are not backwards compatible with each other. Thus, my optimizations will require a substantial re-write in order to be usable with the new system. Also, the new system will once again have to be split into a marking pass and a modification pass. That being said, large chunks of the LLVM API are the same, so while substantial modifications will need to be made, the whole thing will not need to be rewritten for the new version. My hope is that the new version will be stable enough that, coupled with low overhead from the optimizations presented in this paper, SoftboundCETS will be able to give effortless memory safety for C programs in the real world.
Bibliography


