Stony Brook University

Light-weight Bounds Checking

A Thesis Presented
by

Ashish Misra

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2010

Stony Brook University

The Graduate School

Ashish Misra

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. R. C. Sekar, Thesis Advisor
Professor, Computer Science

Dr. C. R. Ramakrishnan, Thesis Committee Chair
Associate Professor, Computer Science

Dr. Robert Johnson
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Light-weight Bounds Checking

by

Ashish Misra

Master of Science

in

Computer Science

Stony Brook University

2010

Memory-related errors such as buffer overflows and dangling pointers remain one of the principle reasons for the failure of C programs. Such failures do not always manifest as program crashes but also as incorrect outputs. Well-tested programs do run error free in most cases, but studies have shown that even such programs can crash when presented with unexpected data. Out of bounds array and pointer accesses are an important subclass of memory-related errors. Despite many years of research in bounds-checking, current solutions are mostly deployed as debugging and testing aids. This is because the current techniques for bounds protection are either too performance intensive to be used in production software or are unable to process all valid C programs.

Hence, in this thesis, we present a backwards compatible lightweight bounds checking technique that aims to provide practical protection to C programs that can be deployed in production software. Our technique involves flanking memory objects with guard zones. We generate instrumentation to check that memory references do not access these guard zones. We are able to avoid some of the compatibility problems associated with previous bounds-checking techniques by avoiding pointer arithmetic checks, and instead relying on checks on the values of dereferenced pointers. To obtain good performance, we partition these runtime checks into two parts. The first part is always performed, and is very fast because it does not introduce additional memory dereferences. The more expensive second part is triggered only if the first check succeeds, which is relatively rare. We present an efficient implementation of our technique. Our results show that the technique has a relatively low overhead in CPU intensive benchmarks. Furthermore, by instrumenting real world applications, we prove the practical utility of our approach.

Dedicated to my loving parents and Preetee Didi.

Contents

List of Figures

List of Tables

Acknowledgments

I am eternally gratefully to my advisor, Dr. R.C. Sekar, for his invaluable guidance and support. I thank Dr. C R Ramakrishnan and Dr. Robert Johnson for being on my defense committee, and providing valuable suggestions. I am thankful to everyone in Secure Systems Lab for having made this such a memorable experience. Last but not the least, I would also like to specially thank Bhuvan Mital and Niranjan Hasabnis for their help and support in this project.

This thesis was made possible in part thanks to NSF grants CNS-0627687 and CNS-0831298.

Chapter 1

Introduction

Memory errors are notoriously difficult to debug. Every programmer dreads the day when an unexplained bug suddenly creeps into a large project. Incorrect output, program crashes, non reproducibility of bugs are some of the symptoms of such errors. Todd Austin states in 1994 [2] reports *"nearly all of the seemingly mature programs could be coaxed into dumping core."*. Fast forward to 2006, its the case that *"errors in the use of pointers and array subscript still dominate the results of our tests"* [7]. Recently Microsoft reported finding and fixing 1800 MS Office bugs using a fuzzing botnet. Thus memory corruption is as a big problem today as it ever was.

Programs written in low level programming languages like C and C++ are considered particularly vulnerable to such errors. C, affectionately known as "Portable Assembly language", provides a programmer complete control over a program's address space. Features like raw memory pointers, pointer arithmetic, unchecked bounds etc result in very efficient code. However such power is more than what most programmers can control and memory errors such as dangling pointers, buffer-overflows and segmentation faults are common even in commercial software. Consequently, a dominant goal in security and programming languages research has been the development of techniques to ensure memory safety of C programs.

Unlike type-safe languages like Java and Ocaml, ensuring memory safety of a C program is a difficult problem due to pointer manipulations that are freely allowed without any checks. A plethora of techniques have been developed to mitigate the problem. These differ widely in terms of range of errors detected, accomplishments, performance and backwards compatibility

An obvious solution is to use a "safe" language instead of C, e.g. Java. However given the popularity of C, languages like CCured and Cyclone have been proposed that aim to introduce minimal changes to C while guaranteeing memory safety. Nevertheless, a change in language leaves open the question of retrofitting existing software. Thus any solution involving a change in language and consequently manual effort is less than desirable.

Instrumenting programs for ensuring safety at runtime is currently the favoured approach. Further more such instrumentation techniques have traditionally focussed on tackling widespread spatial errors (e.g. bounds overflow, uninitialized variables) as opposed to temporal errors (dangling pointers, double frees etc.). Ensuring complete memory safety entails maintaining bounds information regarding all objects at runtime. As has been the experience so far, such an instru-

mentation is expensive in terms of runtime performance. Softbound, a backwards compatible bounds checking technique, reports performance overhead of 67% on average for Specint-2000 benchmarks. For a debugging tool, the performance figures suffice.

Debugging and testing cannot prevent all memory errors. Errors usually happen when programs are fed unexpected data. Fuzz testing tools are based on this methodology and usually are successful in detecting hidden memory errors. Unfortunately testing cannot guarantee memory safety. Hence for detection and prevention of memory errors, runtime instrumentation is a must. This instrumentation has a performance penalty. And that too in perfectly safe runs of the program. A performance impact of 67% on an average is simply not acceptable.

Of course, there have been techniques that have better performance. Enforcing bounds allocation [1, 14] holds appeal in terms of performance. But these techniques have limitations while dealing with out-of-bounds pointers that render them incapable of dealing with set of all C programs. Thus the combined issues of low performance overhead and full compatibility are constraints that none of the existing techniques convincingly address. Hence the objective in our research work has been to develop a technique that has performance adequate for incorporation in production software while being capable of dealing with all valid programs.

In this thesis we propose a new technique for bounds checking that guards both ends of unsafe variables with guard zones and introduces runtime instrumentation to ensure a safe memory access. Any access to the guard zone is flagged as an error and the program stopped. Our technique does not guarantee complete memory safety. However, as indicated by pervasiness of buffer-overflow errors earlier, we believe it still represents a potent solution that can be practically deployed.

The rest of the thesis is organized as follows: Chapter 2 details the contemporary approaches to bounds checking in C programs. Chapter 3 discusses the design of our technique. Chapter 4 describes our prototype implementation and discusses some of the difficulties we faced and problems that needed to be solved. Chapter 5 presents our experimental results with various benchmarks. Chapter 6 discusses related work in this field and contrasts our technique with existing ones. Finally Chapter 7 lists out further ways to improve the performance of our system and concludes the thesis

Chapter 2

Background

This chapter reviews some of the current approaches to bounds checking for C programs. The research pertaining to these approaches has been detailed in Chapter 6

2.1 Static Analysis

2.1.1 Methodology

One way of approaching bounds checking is to have a compile-time only analysis phase. Such tools usually trade scalability for precision.

2.1.2 Advantages

1. No runtime instrumentation is introduced. Thus no performance overheads are generated

2.1.3 Disadvantages

1. Both false positives and false negatives are generated.
2. May require programmer annotation for effective operation.

2.2 Object-based Approach

2.2.1 Methodology

The object based approach utilises the principle of intended referent object. In this approach, bounds information for all the validly allocated memory regions in a program are tracked using an independent data-structure. The validity of pointers can be verified by ensuring that they point to an object contained in the data-structure. Bounds checking is performed at runtime for every pointer arithmetic operation. It ensures that after performing pointer arithmetic using a valid pointer, the resultant pointer also points to the same valid object, the intended referent object. Note that if the resulting pointer goes out of bounds, it still cannot be flagged as an error. An error can be flagged only if an out-of-bounds pointer is dereferenced. More-over an out-of-bounds pointer can return in-bounds again by later pointer-arithmetic operations. Hence when a pointer arithmetic operation results in an out-of-bounds pointer, a new data-structure called out-of-bounds object (OOB object) is created, that stores the base and bounds of the intended referent object.

Subsequent pointer arithmetic operations on an out-of-bounds pointer use the information stored in out-of-bounds object to determine the intended referent object of pointer arithmetic operations.

2.2.2 Advantages

1. The representation of pointers and the memory layout of the program remains the same. Thus the instrumented program can interact with uninstrumented code. (Note that this is not always true. There are techniques which change the memory layout of programs and thus hinder interoperability)
2. Since the metadata of validly allocated objects is stored in a central data-structure, by instrumenting dynamic memory allocation routines, metadata for all dynamically allocated objects can be maintained even if the object was allocated by an uninstrumented library. Furthermore, pointers to objects in uninstrumented libraries can be accomodated even if the object is not registered in the central data-structure.

2.2.3 Disadvantages

1. The metadata for objects is stored in a central data-structure which is usually a splay tree. This is often a performance bottleneck resulting in overheads of 6x or more.
2. Out-of-bounds pointers require the special data-structure, OOB (Out Of Bounds) objects. The very nature of OOB objects, restricts the use of out-of-bounds pointers to only pointer copy and pointer arithmetic operations. Any other use (e.g. using the pointer as input to hash function or an index to an array) is not permitted. This limits the compatibility of the technique to only a subset of C programs.

2.3 Pointer-based Approach

2.3.1 Methodology

In the Pointer-based approach, the bounds metadata is maintained on a per pointer basis. This is in contrast to the object-based approach where bounds metadata is maintained centrally and pointers to an object, share the meta-data. Thus while two pointers can point to the same memory object, different bounds information can be maintained for them. This can prevent intra-object overflows. For eg: overflows in arrays that are member fields of a given structure. A typical implementation of this approach is the fat pointer approach. With fat-pointers, the representation of the pointers is modified to include the bounds information. During a pointer dereference operation, the pointer is checked againsts its bounds information to ensure the validity of the dereference.

2.3.2 Advantages

1. Bounds checking only at pointer dereference implies greater source code compatibility with respect to out-of-bounds pointers.
2. Intra-object overflows can be prevented.

2.3.3 Disadvantages

1. Interfacing with uninstrumented libraries requires functions wrappers.
2. Fat pointers change the memory layout of the programmer. Hence source code must be modi-

fied.

2.4 Redzones

2.4.1 Methodology

Many bounds checking tools focus on dynamically allocated heap objects only. The standard versions of heap routines (e.g. malloc, free, calloc etc.) are replaced by routines that produced heap objects with padding at their ends. These paddings are filled with distinctive values and are called redzones. When the heap block is deallocated, the redzones are checked to verify their integrity. If they have been written to, then a warning is issued.

A variant of this approach is used by some commercial Tools like Valgrind and Purify. These tools maintain additional addressability metadata about every byte of memory in addition to the redzones thus expanding the portfolio of the errors they can detect . The memory accesses are checked wrt to this metadata rather than the redzones. The redzones are marked as inaccessible in this meta-data and thus buffer overflow and underflows are prevented. In this case the redzones serve only as a buffer and are not filled with any predetermined byte pattern

2.4.2 Advantages

1. Easiest way to ensure bounds integrity of heap objects without instrumenting the program.
2. Redzones coupled with addressability metadata work well, as exemplified by Purify.

2.4.3 Disadvantages

1. Works only for heap objects.
2. Only write overflows are detected. Read accesses to the redzones are not detected.
3. Errors are reported only when heap blocks are freed and not where the overflow occurs. It must also be noted that not all heap blocks are freed.
4. Errors detected are limited by the size of the redzone. Other heap blocks can be accessed without any redzone ever being violated.

Chapter 3

Design

In this chapter, we review the overall design of our approach *(also referred to as LBC)*. While the technique is conceptually simple, various subtle features of the C language and its standard library make its implementation rather challenging.

3.1 First Look

The key observation in the design of our technique is that one of the key components of the performance overhead in memory-safety techniques, especially among the faster ones, is the additional memory accesses needed to fetch the metadata regarding bounds information. Hence our design places primary emphasis in minimizing additional memory references.

Conventional bounds-checkers protect against out-of-bounds accesses by storing additional meta-data about the pointer in the terms of the object that it is currently pointing to. By instrumenting either pointer arithmetic or pointer dereference, its possible to ensure at runtime that pointers always refer to their intended objects.

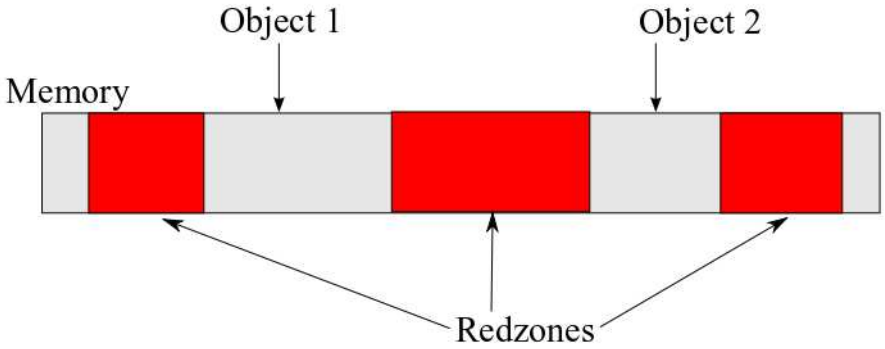


Figure 3.1: Guarding objects using redzones

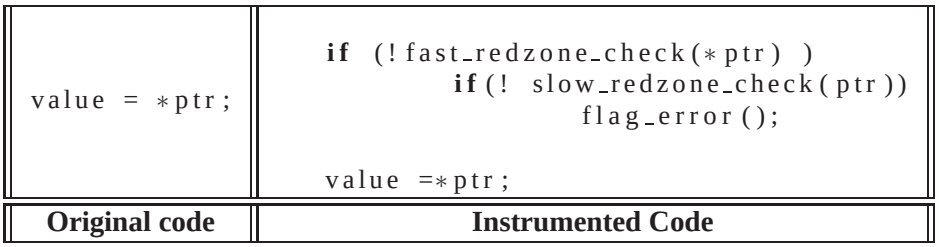


Figure 3.2: Light-weight bounds checking

In LBC, instead of maintaining object bounds, consecutive objects are distinguished from one another, by creating guardzones *(hence forth referred to as red-zones)* before and after objects. The location of these red-zones is maintained in a separate data-structure *(called the red-zone map)* All the red-zones are filled with a pre-specified byte pattern *(called the red-zone value)*. These red-zones represent memory that must not be accessed by a memory safe program. Thus we partition the memory into parts that can be legally accessed and the ones that must never be accessed. Figure 3.1 illustrates this.

Since red-zones do not overlap with any original memory object, red-zones are never accessed by non-pointer lvalues. Hence to ensure memory safety, at runtime, only pointer dereferences need to be checked to ensure that they do not refer to any red-zone.

Our instrumentation comprises of two distinct checks. The first check is called the fast red-zone check. Depending on the outcome of the fast red-zone check, the slow red-zone check is conditionally invoked.

When a pointer is dereferenced, the dereferenced data is checked to see if it matches the red-zone value. This test comprises the fast red-zone test. If it does not, then the program continues with the memory access. Given that the red-zone value is selected randomly apriori, the probability that the dereferenced data has the same value as the red-zone value is low. Thus the probability that the fast red-zone check declares the dereference safe is high.

However, if the fast red-zone check fails, it does not imply that the memory access is unsafe. It only implies that the current memory location has the red-zone value. In such a case, the red-zone map is consulted to validate the pointer dereference. This is the slow red-zone check.Note that the system cannot catch pointers that jump across red-zones. However by selecting the size of red-zone judiciously, we can ensure that the probability of such a case is minimized.

Since the primary objective of our system is to be fast enough to be practically deployable and reasonably effective, case mentioned above is trade-off that we consciously make.

3.2 High Level Architecture

Our technique has both a compile time component and runtime component. We implement LBC using source to source transformation.

The compile-time transformation changes the program in two ways:

- *Object Transformation:* Unsafe memory objects are transformed to incorporate red-zones at

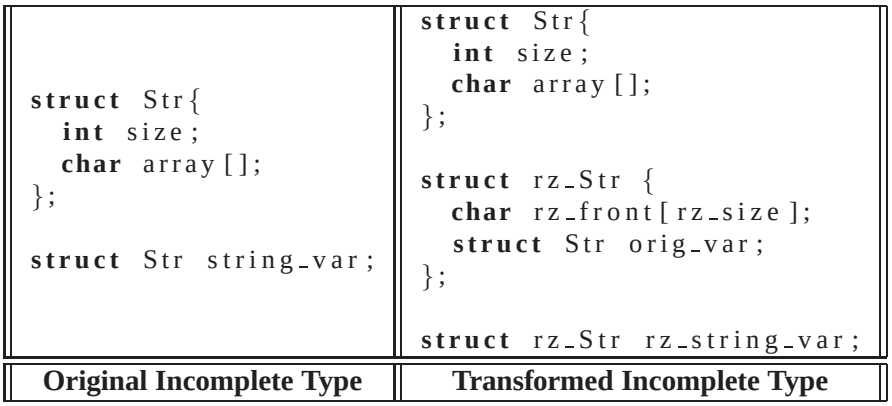


Figure 3.3: Transformation of Incomplete Types

the start and end of the original object.

- *Runtime Instrumentation:* All pointer dereferences are then instrumented to enforce the runtime checks.

The generated code is then linked with our static library and other binary libraries — either instrumented or uninstrumented.

A point to note is that one of the design goals of the LBC system is the ability to be easily incorporated into the build process of a program. This is achieved by doing the transformation on a per translation unit basis.

3.3 Low Level Design

The detailed design involves the following components

- Design of the transformed memory object.
- Size of red-zone
- Initialization of red-zone
- Design of the instrumentation.

3.3.1 Design of transformed memory object

In our transformation, not all objects are transformed. Moreover, of the objects that are transformed, there are differences in how objects are transformed based on the data-type of the object. An object of any complete type can have red-zones both before and after the object. However, for incomplete data types, only the front red-zone is enforced.

Incomplete types

C99 allows a structure to have a unsized array at the end. This basically entails that the program can validly access any memory after the declared structure. Hence given such a structure definition, its not possible to have a rear red-zone for objects defined as such. Refer Figure 3.3

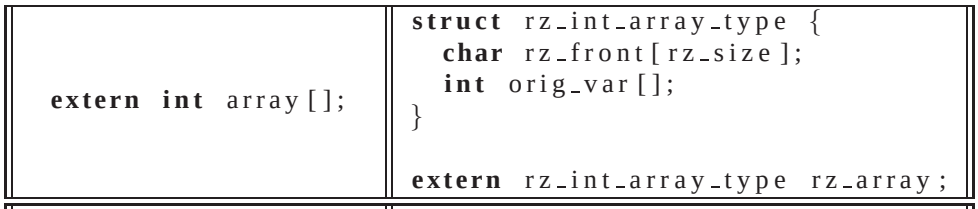


Figure 3.4: Transformation of Global Arrays

Array Declarations

A common C programming idiom is to declare an extern array in the header file without its size information. In such a case, transforming the memory object becomes difficult since the size of the red-zone is based on the data-type of the object. Figure 3.4 gives an illustration of the same.

Such a case to leads to a transformation in which the rear red-zone is absent since it is not permitted to embed an unsized array within a structure. Such an array can only be the last field of a structure. However it must be noted that such a declaration need not be initialized since it is evident that the actual array object will be defined and initialized elsewhere

3.3.2 Size of red-zone

The size of red-zones is important for the safety of the system. It primarily depends on the location of the memory object.

Heap Object

The size of red-zone for the heap objects is influenced by the following factors

1. Must be a multiple of heap chunk alignment (usually 8 bytes).

   The semantics of heap allocation functions dictate that the memory objects are aligned on eight byte boundaries. Hence the red-zone itself must be a multiple of eight bytes to honour this agreement.

2. Type of pointer being assigned to.

   C permits pointers to be cast to pointers to be arbitrary types. Thus its not possible to predict which type of pointer will be used to access the requested memory object. Hence the safest best is the type of the object that is assigned the pointer returned by the heap function.

3. Size of memory object requested.

   Size of red-zone can be set to be proportional to the size of memory object requested. For eg: For a hundred byte memory object, ten bytes can be allocated for the front and rear red-zones

4. Performance constraints. However, because of performance overhead of red-zone initialization and uninitialization routines, the size of red-zone can not be very large. A safe upper limit is the size of largest data-type of the application.

Global Variables

The size of red-zone for global objects is a tricky affair. Since the analyses and instrumentation proceeds on a per translation unit basis, size of data types may not always be able for analysis. This problem can be dealt with by categorizing global objects into the following three groups.

1. *Primitive type global variables*
   For primitive types, the size of the object is fixed and thus the size of red-zone can be proportional to the size of the object.
2. *Array variables of primitive type*
   For arrays, the red-zone size can be fixed proportional to the size of primitive data type.
3. *Global variables of aggregate type (structures, unions) and arrays of aggregate types*
   In case of structures, the type definition seen in one translation unit may not be the type definition seen for the same variable in another translation unit.

   Thus the size of the red-zone cannot depend on the size of the data-type. Hence one straight forward solution is to fix the size of the red-zone. Hence all the offsets now become predictable.

However, it must be pointed out then, that structure assignments would need to be carefully instrumented to ensure that a memory access does not violate red-zone integrity.

Stack variables

For stack variables, as opposed to global objects, the simplifying factor is that data type definition must be complete. Thus for stack objects, the size of red-zone can depend on the type of the memory object.

3.3.3 Initialization of red-zone

There are differences in the creation and initialization of red-zones based on the location of the original memory object.

Stack variables

For stack variables, the red-zones and the red-zone map are initialized on function entry. On function exit, the red-zones are nulled out and the red-zone map updated.

Furthermore, not all stack variables need to be transformed. A simple optimization, first proposed by Jones and Kelly [5] is to transform only those variables on which the address-of operation has been performed.

Global Variables

Global variables are initialized before the start of the main function. Since the lifetime of the global variables matches the lifetime of the program, it does not pay to uninitialize the red-zones and update the red-zone map at the end of the program.

As mentioned earlier, in our technique each translation unit in the program being instrumented, is separately analyzed. Thus the visibility of the instrumentation process is limited to the current

Original Global Variable Declaration	Transformed Global Variable Declaration

`struct global_struct;`
`extern struct global_struct global_var;`
	`struct rz_global_struct;`
	`extern struct rz_global_struct rz_global_var;`
	`extern struct rz_global_struct *rz_global_var_ptr;`

Figure 3.5: Transformation of Global Variables with declared types

translation unit.

Its thus not possible to have the same optimization as in the case of stack variables by limiting the transformation to select global objects. Hence all global variables are transformed to contain red-zones.

Extern Variables

A subtle point in the transformation of global variables involves extern variables. C99 allows a structure to be only declared and not defined.

An extern variable can be declared to have such an declared only type provided its used only with the address-of operator. Note that without data type, its not possible to position the front and rear red-zones.

In such a case, our design incorporates additional meta-data with every global variable. We maintain a pointer that is used instead of &(extern-variable) The code is then instrumented to use this pointer instead the extern variable.

The pointer is initialized at the time of initialization of the global object.

Heap Variables

Heap variables are conceptually the easiest to protect. In fact, the concept of red-zones is most popular among the techniques for heap protection. For eg Mpatrol, CCMalloc etc.

During memory allocation, the size of front and rear red-zones is added to the size of memory object requested. The front and rear red-zones are then filled with the red-zone value and the red-zone map is initialized too. The pointer returned back however points to the beginning of the valid memory region enclosed by the front and the rear red-zones.

During memory deallocation, the red-zone map is updated to reflect the deallocation of the front and rear red-zones. The entire memory object is then deallocated.

Difficulties

However there are again subtle issues when dealing with heap allocation. Memory allocation functions like malloc, calloc do not provide for requesting the memory object to be aligned on user-specified boundaries. Hence such functions can be instrumented simply by increasing requested size to include the size of red-zones.

However for functions like memalign, valloc etc that allow a alignment to be specified, there are arise some issues that need to be dealt with.

Aligned Memory Objects

The semantics of functions like memalign, valloc, pvalloc, dictate that the pointer returned back to the user must be aligned on the specified boundary. Incorporation of a front red-zone in such a case would necessitate the object being aligned on the next boundary. For large alignment requirements, (for eg: on page boundary) this would lead to a lot of wastage of memory.

Hence our current design does not instrument such aligned objects with a front red-zone.

Free Operation The heap object free operation involves a subtle point. As a mentioned earlier, a heap allocated object may/may not incorporate the front red-zone. When a pointer is given to the free function, thus the presence of the front red-zone is unknown.

In such case, the metadata maintained by the red-zone map is relied on to provide the answer.

3.3.4 Design of instrumentation

Emperically it has been observed that number of pointer dereference operations in a program is more than the number of pointer arithmetic operations. This is places our technique at a disadvantage wrt those bounds checking techniques that instrument pointer arithmetic.

Hence it is of vital importance for the instrumentation to be as efficient as possible. While implementing such an instrumentation, the following points need to be observed:

1. The fast red-zone check must not generate extra memory
2. The slow red-zone check validates the correctness of a memory reference. Since the red-zone map is consulted in this case, additional memory references are generated. Hence it must be ensured that the probability of slow red-zone check being invoked is low.
3. Transformation of stack-based memory objects involves red-zone initialization and uninitialization at function entry and exit respectively. This is a significant source of performance overhead and it must be carefully implemented.

3.4 Disadvantages of our design

1. Our technique cannot catch all errors. Pointer dereference operations that lead to a valid pointer accessing memory beyond the red-zone cannot be caught.
2. Instrumenting write operations will lead to additional memory accesses.

Chapter 4

Implementation

We implemented our prototype LBOP system for 32-bit X86 machines. This chapter discusses in detail our prototype implementation, the difficulties we faced and a few points that any similar system would probably need to address.

4.1 Implementation Framework

The LBOP system interacts with the target program at both compile time and runtime. Our prototype is made up of the following components

1. A source to source transformation module implemented using CIL [C Intermediate Language] program analysis tool.
2. Static Libraries
3. Instrumented glibc (for heap object instrumentation)

The system comprises of roughly 2500 line of CIL (Ocaml Code). Static libraries and glibc malloc instrumentation accounts for 1000 lines of C code.

4.2 Source to Source Transformation

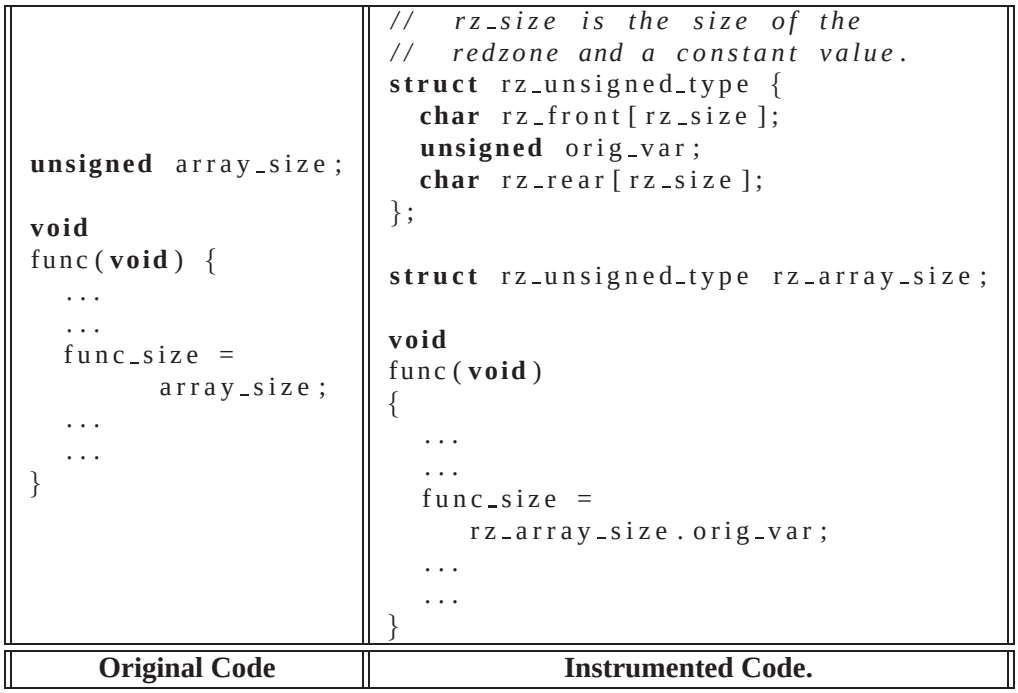
Our source transformation module has been implemented as a module using CIL program analysis infrastructure. CIL provides a high-level tree representation of a C program along with a set of tools that permit analysis and source to source transformation.

CIL implements the Visitor design pattern and provides a visiting engine that scans depth-first the tree structure and provides options to the module at each node.

The module comprises of two passes over the the translation unit under transformation.

4.2.1 Object transformation pass

In the first pass all global variables and unsafe local variables are transformed to incorporate the redzones. Since the redzones need to flank the original memory object, data-type of the transformed memory object is a structure with two char arrays (redzones) and and an object of original data-type as the member fields. The references to these variables are then updated to reflect the



```
unsigned array_size;

void
func(void) {
  ...
  ...
  func_size =
        array_size;
  ...
  ...
}
```

```
//  rz_size is the size of the
//  redzone and a constant value.
struct rz_unsigned_type {
  char rz_front[rz_size];
  unsigned orig_var;
  char rz_rear[rz_size];
};

struct rz_unsigned_type rz_array_size;

void
func(void)
{
  ...
  ...
  func_size =
     rz_array_size.orig_var;
  ...
  ...
}
```

Original Code	Instrumented Code.

Figure 4.1: Transformation of variables and updated reference.

new memory object. An example of the transformation is displayed in the Figure 4.1

Note that transformation of variables themselves is not sufficient. Instrumentation is added to initialize and uninitialize redzones.

Local Variables

For local variables, our current implementation performs this instrumentation at the entry and exit of functions. There is however room for improvement that will be discussed in Chapter 7.

Global Variables

The initialization of global variables is achieved by code in constructor functions created for that purpose. A constructor function is a function with the GCC constructor function attribute and is guaranteed to be executed before the main function is called.

4.2.2 Runtime Instrumentation Pass

In the second pass, all pointer dereferences are instrumented with the runtime checks to ensure memory safety. Figure 4.2 demonstrates the runtime instrumentation. However note that the function call redzone_check(ptr) is only symbolic of the redzone checks. The actual fast and slow redzone checks are different.

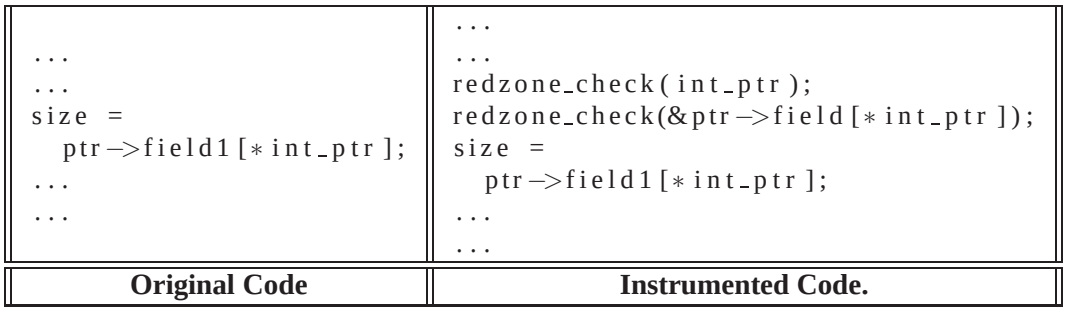


Figure 4.2: Intrumentation of pointer dereferences

4.3 Static Library

The most important component of the library is the implementation of the redzone map (The data structure that maintains the location of the redzones for the entire virtual address space).

4.3.1 Redzone map

In our current implementation, the redzone map has been implemented as a bitmap. Every bit represents a byte of the address-space. Thus theoretically a maximum of 12.5% of additional memory can be consumed by the bitmap.

The redzone map has been organized as a two-level data-structure with the first level as an array of pointers, each pointing to an array representing the address bits. This two level structure abstains from allocating the entire memory in one go. Adopting a single array bitmap for the red-zone bitmap will have the following performance disadvantages:

1. Allocating the entire red-zone bitmap as single array at program startup would dramatically slow down process startup.
2. It would impose large memory overheads uneccessarily

The library contains the subroutines for the maintainance of the redzone map. These routines are invoked by the redzone initialization and uninitialization functions. It also contains a slow-redzone check function that accepts the address to be checked. This address is then looked up in the redzone bitmap.

4.4 Shared library

In our system, the heap-related functions in glibc have been instrumented to introduced redzones around heap objects with related initialization activities. This enables accurate bounds checking of all heap related objects.

4.5 Optimizations

4.5.1 Fast Redzone Check

An efficient of fast redzone check is crucial for the performance of the system. A naive implementation would be to create an inlined function that would accept the pointer to be verified,

```
int
fast_redzone_check(void *ptr)
{
  // redzone_value is
  // the byte pattern
  // with which the
  // redzone is filled.
  return (*((char *)ptr)
          == redzone_value);
}
```

Naive implementation

```
#define                                 \
fast_redzone_check(value, ptr) \
 ({value == redzone_value;})    \
```

Current Implementation

Figure 4.3: Implementation of fast redzone checks

```
//  This is only an illustration.
//  Actual fast and slow redzone
//  checks differ.
if(!fast_redzone_check(ptr))
   slow_redzone_check(ptr);
```

Naive implementation

```
if(!fast_redzone_check(ptr)){
 asm("pushl %%eax\n"
     "pushl %%ecx\n"
     "pushl %%edx\n"
     "call slow_redzone_check\n"
     "popl %%edx\n"
     "popl %%ecx\n"
     "popl %%eax\n"
     : : :
   );
}
```

Current Implementation

Figure 4.4: Inline assembly call to slow red-zone check

dereference it and then compare against the redzone value.

Unfortunately, as we found out, this implementation proves too costly in terms of performance. With such an approach, we experienced overheads of about 70-80% with bzip2 function. The fast-redzone check must necessarily be implemented as a macro that accepts the value that represents pointer being dereferenced by the program.

At the assembly level, such a value will most probably reside in the processor's register. The check above would then compare the redzone value against the register enabling a fast check.

Using an inline function, would require casting the pointer to void pointer type, recasting it back to a character pointer and then dereferencing it and comparing against the redzone value.

As we discovered, this casting of pointers leads the compiler to generate code that compares the redzone value directly against the memory leading to a huge overhead.

Figure 4.3 illustrates the naive approach vs the current implementation. Note that the code does not represent the actual red-zone fast check.

4.5.2 Function call using inline assembly

Implementing the fast redzone check as above still is not sufficient to reduce performance overheads. Hence in our implementation the calls to slow redzone check are embedded in an inline assembly The salient feature of the inline assembly code in the Figure 4.4 is that the list of registers clobbered is empty inspite of a function call being made.

It must be noted that ABI on Unix system specifies that the registers eax, ecx, edx are caller saved registers and the called function is free to use them as per its requirements. If the call to slow redzone check had been implemented as a normal function call, the caller-saved registers are considered dead by the compiler and are reloaded from memory. Since about number of redzone checks inserted were of the order of $10\hat{7}$ this lead to a big hit on performance

Hence the inline assembly code saves and restores the caller saved registers from the stack before and after calling the slow redzone check function. Moreover, inline assembly is never parsed by the compiler. Thus the function call is hidden from the compiler and thus extra memory references can be avoided.

Incidentally, Baggy-Bounds-Check [1] too implements a similar concept to reduce performance overheads.

4.5.3 Array Bounds Check

A very simple optimization is to use a bounds check in preference to a redzone check wherever possible. This is motivated by the optimization capabilities of the compiler which can optimize bounds check in a much better way as compared to the redzone check which involves memory dereference.

Thus where-ever the dereference is guaranteed to be an array indexing operation, an array bounds check is performed as compared to a redzone check.

4.5.4 Redzone map maintenance

For redzones guarding the global variables, the redzone map needs to be initialized only once. However for unsafe stack variables that have been transformed, the functions maintaining the redzone map must be as optimized as possible.

Hence our implementation imposes the following restrictions on the redzones.

1. Size of the redzone must be a multiple of 8 bytes.
2. The redzones must be aligned on 8-byte boundary.

This leads to measurable improvement in performance.

Chapter 5

Evaluation

In this chapter, we present the performance evaluation of our prototype implementation. Section 5.1 evaluates the performance of runtime instrumentation when slow red-zone checks alone are performed.

5.1 Motivation for fast redzone checks

As mentioned earlier, one of the primary components of performance overheads in runtime instrumentation is the additional memory accesses necessary to fetch the bounds meta-data. Our primary focus has been to reduce these additional memory accesses.

The key to performance improvement has the fast red-zone checks that use the already dereferenced data to decide whether the slow red-zone check should invoked or not.

To check the effectiveness of this approach we compared our current performance to the performance of runtime instrumentation with only slow red-zone checks to validate the memory accesses.

The experiments involved compressing and decompressing a 225MB media file by instrumented bzip2, bunzip2 and gzip programs.

The results are as in Figure 5.1. As can be seen slow-checks alone cause slowdowns of upto 8X in case of bzip2 and 3X in case of gzip. However, coupled with fast red-zone check, the performance overheads drop to 25%, 32% and 34% for gzip, bunzip2 and bzip2 programs respectively.

5.2 Performance Evaluation

In this section, we evaluate the performance of our prototype implementation using CPU Intensive SPEC benchmarks, its effectiveness in preventing overflows and measuring the performance of real world software.

We evaluated our prototype's performance using CPU2006 benchmarks on a system with 2.00GHZ Intel Core 2 Duo processor and 3GB RAM running Ubuntu 9.10.

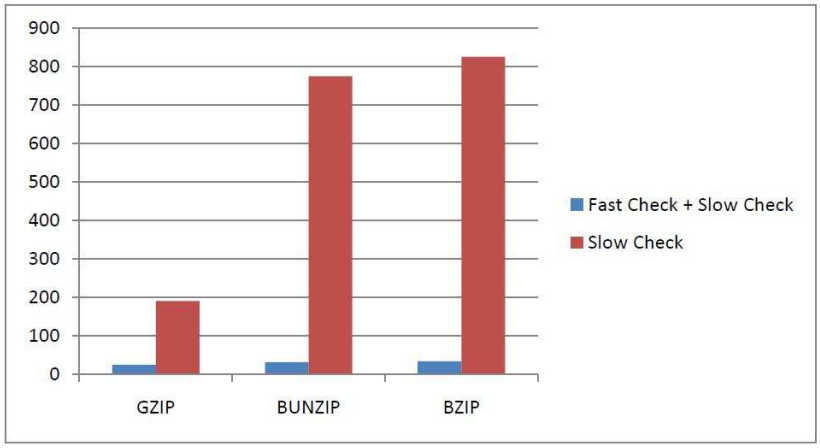


Figure 5.1: Performance evaluation of combined red-zone checks with slow red-zone checks only.

5.2.1 Specint 2006 Benchmark

We chose to use the SpecInt 2006 benchmarks because they better represent the current software performances as compared to SpecInt 2000 benchmarks.

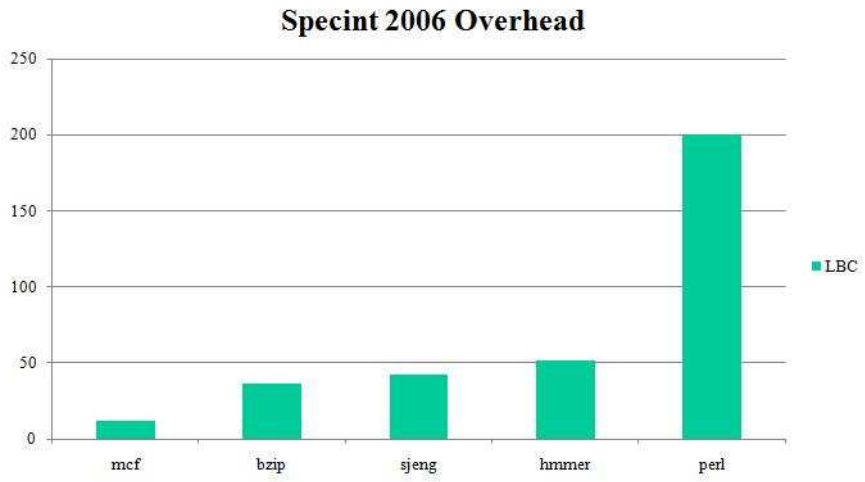


Figure 5.2: Performance evaluation using Specint-2006 benchmarks.

The above Figure 5.2 above shows our runtime overheads. The program mcf shows lowest overheads at 12% while the highest overhead is for the perl benchmark at 232%. The Bzip2 benchmark is reported at 35% runtime overhead.

5.2.2 Comparison with previous work

Since the Bzip2 and Perlbmk benchmarks are common to both SpecInt 2000 and SpecInt 2006 benchmarks, they can be taken to be a reference point for comparison with previous work in this field. Another program from Specint 2000 benchmark, gzip can be easily independently verified for overheads generated.

Our prototype can be compared to previous works: Baggy Bounds Check [1] and WIT. Baggy Bounds has previously reported some of the best performance figures yet. WIT reports even lower performance figures but it instruments only memory writes.

But an important point to note is that WIT clearly states its goal to be prevention of memory exploits. By not instrumenting memory reads, they cannot prevent the most eggregious of memory errors. Our focus, on the contrary, has been to efficiently detect as many memory errors as possible.

The performance overheads can be compared as follows:

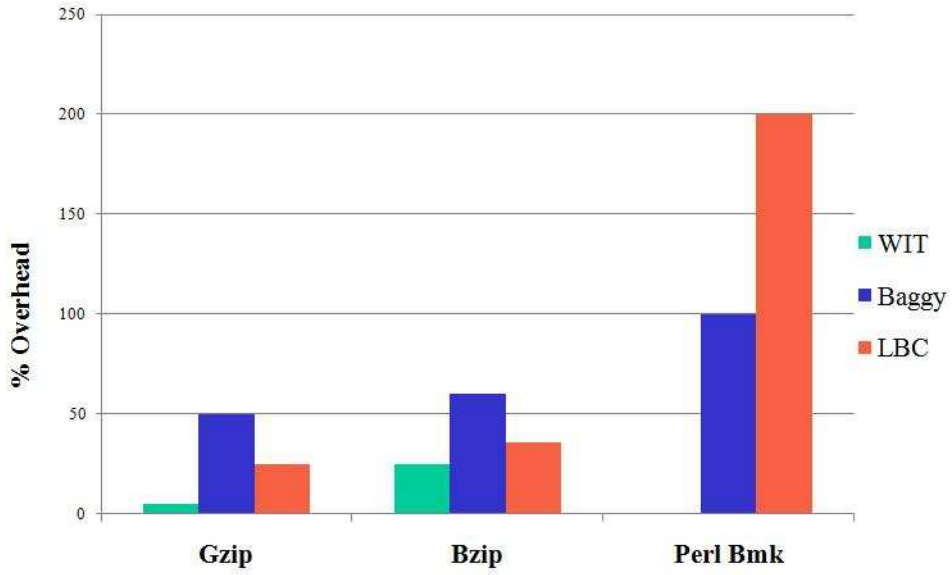
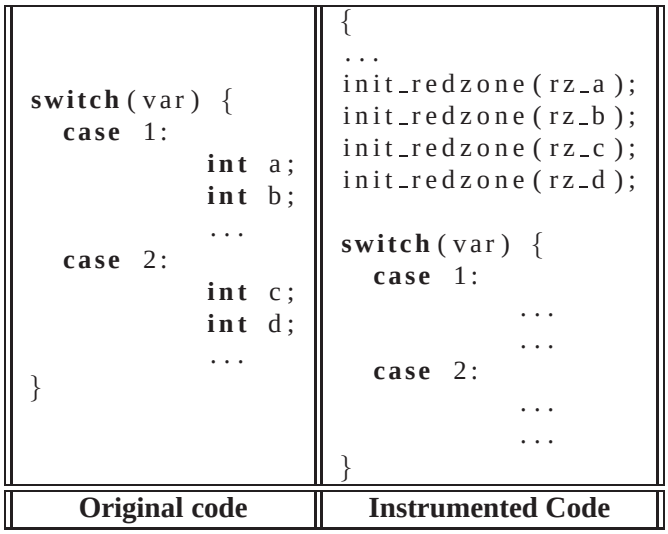


Figure 5.3: Comparison of overheads wrt Baggy Bounds Checking and WIT

Bzip2

Baggy bounds check reports a performance overhead of 60%. WIT quotes an lower performance overhead for Bzip2 at 25%, but then WIT approach only prevents vulnerabilities. It needs to be pointed out that Baggy bounds check paper relies on static analysis based optimizations while our prototype currently employs none.



```
switch(var) {
  case 1:
          int a;
          int b;
          ...
  case 2:
          int c;
          int d;
          ...
}
```

```
{
...
init_redzone(rz_a);
init_redzone(rz_b);
init_redzone(rz_c);
init_redzone(rz_d);

switch(var) {
  case 1:
          ...
          ...
  case 2:
          ...
          ...
}
```

Original code | **Instrumented Code**

Figure 5.4: Perl benchmark code snippet.

Gzip

For gzip too, our technique's 25% overhead compares favourably with Baggy bounds check's 50%. WIT reports 5% overhead.

Perlbmk

However, for Perlbmk, Baggy Bounds Check's performance overhead (100%) is much lesser than our own (232%).The above performance figures necessitated a deeper evaluation of the Perl benchmark. Our analysis indicates that even in case of Perlbmk, the overhead of redzone checks is still 35% while the cost of maintaining the redzone map accounts for the rest.

The Perlbmk has multiple functions which display the structure as in the above Figure 5.4. The salient features are:

1. Extensive use of stack array objects and thus a large number of unsafe stack variables
2. Switch-case structure with variables being declared within the case statements.

However in the instrumented code, all the variables are declared in the outermost block. Thus there is unnecessary definition and initializiation of unsafe variables that would probably not even be allocated in the original code. This removal of block level variable declarations is an ”feature” of CIL.

Other implementations would not suffer the same limitation.However in all probability, there would still remain substantial overheads. These would be mitigated as far as possible, with the help of static analysis techniques discussed in Chapter 7

5.3 Real world Applications

To verify the useability of our approach, we instrumented some real-world applications.

Program	KSLOC
Openssl-0.9.8k	397
Nullhttpd	2
libpng-1.2.5	36

Table 5.1: Real-world applications instrumented and their size

1. OpenSSL toolkit.
2. Libpng. Libpng reported overheads of 30%
3. Nullhttpd. We could trace two array bounds violations that were also reported by Baggy Bounds Check [1]

Chapter 6

Related Work

There has been intense research in Bounds checking over the past few years to the point where current state of the art techniques are nearing deployment to production server but just not completely there yet.

This chapter reviews some of the previous research in this area and how it contrasts with our work.

6.1 Goals of our approach

At this point, it would be useful to review the goals that our approach aims to fulfill:

1. Performance overheads adequate for production systems.
2. Full compatibility with all C programs and full compatibility with uninstrumented libraries.

6.2 Impetus for bounds checking

Patil and Fisher [10] provide good motivation for the need of runtime instrumentation. Miller et al [7] provide an evaluation of random testing of MacOS's applications and provide empirical evidence of the pervasiveness of bounds checking errors.

6.3 Safe Languages

Cyclone [4] is a C-like language that was focussed on minimizing differences with C syntax and semantics while providing memory safety. However the requirement of detailed pointer annotation makes porting C programs to Cyclone programs a non-trivial effort. CCured [9] differentiates between safe and (potentially) unsafe pointers and introduces runtime checks for unsafe operations. However the pointer representation is changed for unsafe pointers and this can create compatibility problems with some external libraries.

Thus the compatibility issues limit the applicability of such languages and hence their usefullness.

6.4 Pointer based Approach

Pointer based approach tracks the bounds metadata for each pointer, e.g using fat pointers. Kendal [6] and Steffen [12] proposed the use of fat pointers with extra fields maintaining the bounds of the current object associated with the pointer. Such an approach can detect intra-object overwrites (an overwrite from one field of a structure into another) but changes the memory layout in a way that induces incompatibility with external libraries. Moreover there are significant runtime overheads associated with this approach. Wei Xu et al [13] proposed techniques that addressed the problem of both spatial and temporal errors by maintaining object related metadata and instrumenting pointer dereferences. However the technique suffered from relatively high overheads and compatiblity issues (library wrappers were needed to interact with uninstrumented libraries). More recently Soft bounds [8] too utilizes a pointer based approach.

6.5 Object based Approach

The problems of fat-pointer approach are resolved by the referent object approach proposed by Jones and Kelly [5]. Their key observation is that pointers go out of bounds on the account of incorrect pointer arithmetic. Thus their implementation instruments both pointer arithmetic and pointer dereferences. Their system maintains backwards compatibility by preserving the pointer representation and at the same time maintaining object bounds in a separate data-structure. Their system unfortunately enforces a strict enforcing of ANSI C standards that breaks 60% of tested programs and also incurrs overheads of upto 12X at runtime [11]. However, to their credit, their object-referent object has proved to be the lynchpin of several techniques proposed later.

CRED [11] improves the referrent object approach by using the OOB [Out-of-Bounds] objects to store meta-data that pointers that overflow or underflow an object bounds. However this technique introduces limitations on the use of pointers to only-copy operations after going out-of-bounds. Thus it is not completely compatible with all valid C programs. For eg: A program that use out-of-bounds pointer (say) as an index would break because of the technique.

Baggy Bounds Checking [1] extends CRED by introducing the notion of maintaining allocation bounds as opposed to object bounds. This insight enables them to achieve good performance figures. However, their compatibility with programs is worse than CRED [11] for it cannot deal with out-of-bounds pointer beyond a very limited address range. This is not an insignificant limitation.

PAriCheck [14], developed concurrently to Baggy bounds [1] checking, also utilizes the principle of referent object checking of pointer arithmetic and enforcement of allocation bounds. Their key insight is in employing a more efficient alternative to pointer arithmetic checking as compared to splay-tree based object bounds approach. Instead they utlize a object-specific 'label' comparison to achieve better performance. In terms of compatibility, they score above Baggy bounds checking, but still suffer from the limitations of CRED [11].

Thus referent object based approaches have so far not resolved compatibility issues.

6.6 Traditional Redzone Techniques

Redzones have traditionally been used for protecting memory objects on the heap (Mpatrol, Purify). This technique has also been used for debugging buffer overflows within the Freebsd kernel.

The methodology of these techniques is usually as follows:

1. Add padding to heap blocks at the ends.
2. Fill the redzones with distinctive values.
3. When the heap block is freed, check the integrity of the redzones.
4. If they have been written to, issue a warning.

Of the above points, the only thing common between our technique and previous approaches is the presence of redzones. The significant differences are as follows:

1. All objects (stack, heap, global) are protected with redzones.
2. Runtime checks are performed at memory access and not just at free.
3. Memory reads into the redzones are detected and flagged as errors, as opposed to just memory writes in case of previous tools

In commercial tools like Valgrind and Purify that utilize redzones, memory accesses are validated against a additional addressability meta-data about every byte of memory. The redzones are marked as inaccessible in this meta-data and thus buffer overflow and underflows are prevented. Thus the redzones serve only as a buffer and are not filled with any predetermined byte pattern as opposed to our technique where the redzones serve as primary defense and redzone map acts as the secondary check. Moreover, the above tools are too heavyweight to be used in production environments and serve strictly as debugging aids.

Chapter 7

Conclusion and Future work

Our current implementation emphasized the development of efficient runtime instrumentation. We do perform some simple optimization in case of stack-based variables by instrumenting only those variables that can accessed in an unsafe manner. The employment of static analysis techniques would enable identification of safe memory operations and thus reduce runtime instrumentation

Furthermore, inspite of the techniques currently employed, handling unsafe stack variables is still an expensive proposition. Efforts need to be directed towards mitigating this problem. In this section, therefore we present some potential directions we would explore in the future.

7.1 Static Analysis

Bounds checking has traditionally relied heavily on static analysis to optimize performance [3]. Runtime checks can be avoided if the validity of a pointer operation can be statically ensured. Furthermore, hoisting a runtime check out of a loop can prove important for a performance-critical loop.

As has been mentioned earlier, a compiler can analyze bounds check more easily as compared to redzone checks. Currently, we naively replace redzone checks with bounds check whenever bounds information is available.

However this optimization can be extended only to unaliased pointer variables whose bounds can be completely tracked. For eg: an instrumentation could loose pointer related bounds information when the pointer is assigned the evaluation of an expression that involves an aliased pointer. Thus source transformation would need to employ either a bounds check or a redzone check depending on the validity of the pointer's bounds information. This decision can be made either at compile time or at runtime.

7.1.1 Runtime selection of instrumentation

One approach would be to associate additional bounds meta-data with every unaliased pointer along with a metadata-validity flag. This flag would be set to invalid when bounds information is no longer reliably available. The runtime instrumentation could then choose to perform either bounds check or redzone check based on the meta-data available at runtime.

7.1.2 Compile-time selection of instrumentation

Another approach is to maintain the bounds metadata for only those pointers for whom it can be assured that bounds information will always be available. This can be implemented by conducting a second pass over the generated runtime instrumentation to ensure that bounds information for a pointer can be tracked. If its the case that bounds information for a given pointer can be lost over even a single code path, then redzone checks are employed for that pointer's dereferences.

The advantage of this approach is that either a bounds check OR redzone check is employed by the runtime instrumentation. But the decision is made at compile time and thus instrumented checks are simplified.

7.2 Instrumentation Optimization

7.2.1 Unsafe Stack Variables

As was demonstrated by the perlbenchmark, the instrumentation of stack variables has a significant effect on the performance of our system. Part of the problem can be attributed to our implementation's use of CIL program analysis infrastructure.

CIL simplifies the code in a function definition by moving variable declarations of every block to the outermost block. While this simplifies the analysis, the generated code consumes additional stack space. Furthermore our current implementation bundles all redzone initialization and uninitialization activity at the function entry and exit. This leads to unnecessary performance overheads.

One solution could be to initialize all the redzones at the first pointer dereference operation. A more sophisticated approach would be initialize redzones of only those objects which can be referenced by the pointer.

7.2.2 Stack frame layout in redzone map

An interesting observation can be made about the redzone bitmap layout for a stack frame for functions without dynamic size stack arrays. It can be easily observed that the bitmap layout would remain the same for every invocation of a given function. This observation could potentially do away with need for expensive redzone map maintenance operations at every function invocation.

7.3 Conclusion

In this thesis, we have thus presented a new lightweight backwards compatible approach for buffer overflow protection in C programs. As opposed to some of the previous techniques, our approach offers a protection mechanism for any C program. It also demonstrates the ability to seamlessy integrate with uninstrumented libraries and modules thus enabling its introduction to any system. Our implementation demonstrates reasonably low performance overheads. Our section on future work details a host of approaches to further reduce the overhead. Furthermore, our emphasis on separate compilation and automatic instrumentation enables an easy integration into a software's build process. These attributes thus make it a potent and readily deployable approach to preventing runtime buffer overflows.

Bibliography

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[3] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *International Conference on Software Engineering (ICSE)*, 2006.

[4] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the Annual USENIX Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[5] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97)*, pages 13–26, Linkoping, Sweden, May 1997.

[6] S. C. Kendall. BCC: Run-time checking for C programs. In *Proceedings of the USENIX Summer Conference*, 1983.

[7] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM.

[8] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *PLDI*, pages 245–258, 2009.

[9] G. C. Necula, S. McPeak, and W. Weimer. Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[10] Harish Patil and Charles N. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *SPE*, 27(1):87–110, 1997.

[11] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium (NDSS)*, February 2004.

[12] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software—Practice & Experience*, 22(4):305–316, 1992.

[13] Wei Xu, Daniel C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *FSE*, California, November 2004.

[14] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *ASIACCS '10: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, New York, NY, USA, 2010. ACM.