MEng Project Report

Finding What C++ Lost:
Tracking Referent Objects with GCC

Alexander Lamaison
Department of Computing
Imperial College London

<alexander.lamaison03@imperial.ac.uk>

Supervisor: Paul H. J. Kelly <phjk@doc.ic.ac.uk>

16th June 2007

Abstract

Pointer arithmetic in C and C++, though powerful, is inherently dangerous and a constant source of program flaws. Technologies exist to reduce these risks, one of which is Mudflap: an extension to GCC 4 that performs runtime bounds-checking. This has an number of weaknesses, notably its inability to detect the case of pointers that stray between objects allocated adjacently. We have created MIRO, an improved version which uses the stricter approach of referent object tracking. As well as increasing the effectiveness of the checking, this allows us to provide more useful debugging information when violations occur. It requires no changes to existing C and C++ programs including non-compliant, but common, calculations involving invalid pointer values. Code compiled with MIRO can be mixed freely with unchecked code combining safety with performance and reducing the threshold to adoption.

Acknowledgements

I would like to thank, firstly, my supervisor and mentor Paul Kelly for guiding me through this project as well as the last four years of study and for always listening to my ramblings without complaint. Steven Lovegrove, my friend, in particular for helping me track down the source of the C++ STL problem but for help with many other problems too. Also John Wilander, Mariam Kamkar, Michael Zhivich, Tim Leek and Richard Lippmann for kindly donating the source code to their testbeds that were so invaluable throughout this project.

Contents

1 Introduction

In any other branch of engineering, a system designed with no internal safeguards, yet which time and time again had been shown to cause serious failures, would be considered grossly negligent and would have long been outlawed. However, for some reason this is accepted as standard practice in computing [15].

Programming languages suitable for writing low-level system software must allow direct access to memory. This is achieved with *pointers*, which store memory addresses, or a similar mechanism. C and C++, currently the second and third[1] most popular programming languages respectively [1], give programmers full access to the power of pointer arithmetic without any checks on how that power is used. It is left up to the programmer to ensure this is done correctly. Herein lies the problem: programmers make mistakes. Frequently.[2]

Other languages take a variety of approaches to provide an increased level of safety (Section 2.2). Java, for instance, does away with pointers entirely in favour of references but loses the ability to create system software into the bargain. This has resulted in the bizarre situation where higher-level languages for run-of-the mill programs are relatively safe but the languages in which almost all critical software is written have no protection at all.

There are two principle effects of this lack of protection. Firstly, it can be very hard to identify the source of problems. Errors when accessing memory can damage unrelated data and symptoms only manifest themselves when this data is used. Secondly, and arguably more importantly, there is nothing to prevent errors being exploited by a malicious third party especially as these are the typical languages used for software which is exposed to the outside world.

Several attempts have been made to solve this problem (Chapter 3) but their limitations prevent common use.

Cyclone is a restricted dialect of C that prevents unsafe use of the language [12]. It requires 5-15% of existing code to be changed (Section 3.2.2).

CCured uses *fat pointers*: a modified pointer representation that holds extra information about the legal bounds [4]. The modified pointers make it hard to interact with pre-compiled code such as system libraries. Either the entire system must be recompiled to use fat pointers or wrappers must be written to convert into and out of fat pointer form. These complications limit its appeal (Section 3.2.2).

ProPolice rearranges the way buffers are placed in memory and inserts dummy data in order to detect when a buffer has overflowed [10]. It is aimed at security and its suitability for general program checking is limited (Section 3.2.1).

[1]Java is first.

[2]During 2005, 5,990 security vulnerabilities were reported to the CERT Coordination Center [2].

Jones and Kelly created an extension for the GNU C Compiler (GCC) that tracks all allocated blocks of memory [21]. It halts execution if the program tries to use a pointer that changed which block it referred to as a result of pointer arithmetic or that does not refer to an allocated block at all. This is known as the *referent object* approach. While this interpretation of C is strictly according to the standard, in practice a large number of programs use invalid pointer values in normal operation. This limits the real-world effectiveness of the extension (Section 3.2.4).

We set out to improve this situation in a way that was suitable for mainstream use. This meant adding support for bounds-checking to the latest version of the GNU Compiler Collection [16] (GCC 4) specifically focussing on C++ as this had received less attention than C. Our aims included:

- Improved program stability
- Improved program security
- Easier debugging to reduce burden on programmer

Our solution builds upon two of the most promising approaches that avoid some of the limitations outlined above:

Mudflap is part of GCC 4 and supports C++ in addition to C (although it has some problems in that regard: see Section 5.7). Similarly to Jones and Kelly, it maintains a database of allocated regions of memory at runtime. If a program tries to use a pointer that does not refer to one of these allocated regions, a violation is reported. Unlike Jones and Kelly, it does not check pointer arithmetic—only dereferencing—making it unable to detect a pointer that strays into the wrong allocated block (Section 3.2.3).

CRED is an update to the Jones and Kelly extension that deals with the problem of programs operating on invalid pointers by creating an alternative pointer representation used only for invalid ones. This allows the referent object approach to work with real-world programs (Section 3.2.4). Like Jones and Kelly, CRED supports C alone.

Achievements

We extended Mudflap to use the *referent object* approach by Jones and Kelly shown to be successful for C [21] and included the later improvements to that approach by Ruwase and Lam with CRED [28] which greatly increased its practical effectiveness (Chapter 5).

Mudflap checks when a pointer is dereferenced that it refers to an allocated region of memory. The referent object approach has stricter requirements: a pointer can never change which allocated region it refers to as a result of pointer arithmetic. We added checks to enforce this. In addition we incorporated the CRED improvements which require almost all operations involving pointers to be intercepted to convert invalid pointers in and out of their alternate representation.

The result of our efforts is MIRO (Mudflap Improved with Referent Objects), a bounds-checker that contributes a number of improvements to the state-of-the-art:

- Bounds-checking of C code: checks ensure that a pointer can never be dereferenced with a value outside of an allocated region. Furthermore, a pointer cannot be dereferenced if it has strayed into another allocated region. Mudflap is unable to detect this latter case and is one of the main strengths of using the referent object approach.

- Bounds-checking of C++ code: we used the language-independent middle-end of GCC 4 making the referent object approach available for C++ code for the first time. Problems that Mudflap had when using the C++ Standard Template Library have also been eliminated (see Section 5.7).

- Fine-grained bounds-checking: a bounds violation is detected even if a pointer strays one byte outside of its allocated region. This level of accuracy is not possible with approaches such as ProPolice and, in the case where two regions are allocated contiguously, not with Mudflap either.

- Informative error messages for easy debugging: in addition to reporting the location of attempts to dereference invalid pointers as done by Mudflap, we are able to provide the location at which the pointer first became invalid. If these two locations are far apart, this can greatly speed up debugging.

- Backwards-compatibility with existing software: no changes are needed to existing code to support bounds-checking. Any valid C or C++ will work. This has great advantages for practical use over approaches such as Cyclone. Some invalid code can also be correctly checked. Most notably, real-world programs often use invalid pointers to calculate valid ones. We can correctly check such code by storing the invalid pointer separately and intercepting its use in calculations.

- Ability to mix with unchecked, precompiled code: parts of a program can be selectively compiled with bounds-checking while code which is considered reliable, such as libraries, can be left unchecked. This provides the best-of-both-worlds: safety checks for new code leaving mature code with high performance. This is not possible with several other solutions especially those based on the fat-pointer approach.

This report presents a background of the existing work in the field, the design towards which we were working, the details of our implementation and an evaluation of our solution's effectiveness.

2 The Problem

Pointers store memory addresses allowing direct access to whatever is stored at that address. They are a very useful programming tool giving programmers much flexibility when accessing memory; something of critical importance if a language is to be used for low-level system software. In C and C++ there are effectively no restrictions on how the pointers can be used and it is this freedom that makes development hard and programs vulnerable.

2.1 Sources of error with pointers

Arrays in C and C++ are synonymous with pointers. Although they use a more natural syntax, this is simply masking a *pointer arithmetic* operation using the address of the array's first element as the base pointer (see Pointer arithmetic).

An array is declared to be of a certain fixed size and this amount of memory is allocated to it. However, the language imposes no restrictions on subsequent array accesses which can refer to memory far beyond the allocated region. This is demonstrated in the following example.

```
int main()
{
    int array[2];   // Create an array with two elements
    array[2] = 0;   // ERROR: array[2] is not part of the array
}
```

A two-element array has been declared and the appropriate amount of memory (eight bytes) is allocated. The array is accessed with an index which is outside the boundary of the array but, as the language does not enforce these boundaries, this is not prevented. Depending on the exact location of the allocated memory, the assignment may well have overwritten critical data. It is likely that this would only manifest itself later when the memory in question is read, making diagnosis of the problem extremely difficult.

Pointer arithmetic can be used to increment, decrement, compare and even arbitrarily assign a value to a pointer. This has huge implications for the safety of pointers in C and C++. If a pointer can be assigned an arbitrary value then it is only the correctness of the programmer's logic that ensures it refers to something sensible.

This example is equivalent to the previous but uses pointer arithmetic to access the array. C and C++ do not restrict a pointer to make it always point at the object[1] it first referred to.

[1]Unless otherwise stated, we use the notion of an object as "the fundamental unit of memory allocation" as defined in Jones and Kelly [21]. This can be a simple variable, an array, a structure or a region of memory allocated on the heap in a single `malloc`/`new`-style operation.

```
int main()
{
    int array[2];           // Create an array with two elements
    int* pointer = &array;  // Put address of array in a pointer
    *(pointer + 2) = 0;     // ERROR: this address is past end of array
}
```

Once again, the location assigned to is not part of the array and memory belonging to another object has been overwritten possibly causing serious problems later.

Null pointers are a special pointer value indicating that a pointer does not point to an object. While they are often used intentionally, they can occur accidentally if, for instance, an object was freed from memory and a subsequent attempt was made to use it or if the object was not initialised in the first place.

```
int main()
{
    list<int>* list = NULL; // Create a pointer to a list
    list->clear();          // ERROR: we used pointer before initialising list
}
```

Unlike the previous problems this error will usually cause the program to abort at runtime and therefore there is less potential for damage. However, the abort will generally take the form of a *segmentation fault* and therefore provides little or no information on where the problem occurred to help with debugging.

2.2 Solutions

Other programming languages take a variety of approaches to tackle these problems. Java, for instance, does away with pointers in favour of references. These are a form of *opaque pointer* that make it impossible to retrieve or change the raw memory address being pointed to. Only the referenced object can be accessed. Pointer arithmetic is not possible and the associated problems—along with the flexibility—are eliminated. Pascal takes a compromise approach by making the notion of arrays and pointers distinct unlike in C and C++ where they are synonymous. Arrays can only be accessed via the traditional `array[index]` operator and the index is checked against the stored size of the array. Explicit pointers are not checked giving freedom similar to C and C++.

Another safety measure is *bounds-checking*. A limited definition of bounds-checking is ensuring that an array access does not go beyond the end of the array. The following example shows the equivalent Java code for the array bounds example given earlier, and the resultant output.

```
public static void main(String[] args)
{
    int[] array = new int[2];
    array[2] = 0;
}
```

```
>java Array
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException: 2
  at Array.main(Array.java:4)
```

The error is trapped at runtime and the program prints an informative message explaining the type of problem encountered along with the exact position in the source code, before halting. By way of contrast, the equivalent C++ program gives no indication that anything untoward may have happened:

```
int main(int argc, char** argv)
{
    int array[2];
    array[2] = 0;
}
```

```
>./Array
```

The only symptoms of such a problem may be general program instability, due to overwriting unrelated data, or security breaches as outside entities take advantage of the program flaws to launch *buffer overflow* attacks [6].

Although not immediately apparent as one, dereferencing a null—or otherwise invalid—pointer is a form of bounds error. Pointers are generally meant to point to an object and thus dereferencing an invalid pointer can be considered an error caused by a pointer that has strayed outside the bounds of its intended object. A broader definition of bounds-checking, therefore, should include catching attempts to dereference an invalid pointer.

The following example demonstrates how a Java program deals with a null pointer problem equivalent to the C++ version outlined earlier.

```
public static void main(String[] args)
{
    List list = null;
    list.clear();
}
```

```
>java Nullp
Exception in thread "main"
  java.lang.NullPointerException
  at Nullp.main(Nullp.java:6)
```

As with array bounds errors, the program indicates the type of error and the location in the source code that caused it and then terminates.

```
int main(int argc, char** argv)
{
    list<int>* list = NULL;
    list->clear();
}
```

```
>./Nullp
Segmentation fault
```

Unlike with the array bounds violation, this time the C++ program terminates with a segmentation fault. This is caused by attempting to access a region of memory which the program does not own (namely, the address `0x00000000`). However, the error message produced is wholly uninformative making debugging a large program a difficult task.

2.3 Aim

Our broad aim is to increase the safety of C and C++ to improve program stability and security and ease the burden on the programmer by making debugging easier. Pointers make these the powerful language that they are and therefore we do not want to follow the route of only allowing references. Instead we will focus on ensuring that the pointers are used correctly.

3 Background

Much work has been done on the topic of improving pointer safety. While most has focussed on C rather than C++, many of the principles are the same. The approaches can be divided into two types which we explore in the rest of this chapter.

3.1 Static analysis

Source code can be analysed by tools that detect unsafe use of pointers and bring these to the attention of the programmer. This is done automatically and without the need for test cases [35]. The program is not running at this point and the detected problems must be fixed before it is, however, the large number of false positives make this difficult [32]. The problems detected at this stage can be considered *expected problems*.

Static analysers also give false negatives [32] allowing *unexpected problems* through the testing process and any effects of these are unconstrained in the running program.

3.2 Dynamic detection

Other approaches are concerned with detecting pointer safety problems at *runtime*. If such solutions are used to check the quality of a product (to find *expected problems*) then they must usually be run in conjunction with a test suite. When *unexpected problems*, inevitably, slip through testing this approach has the advantage that the program will generally terminate before any damage such as buffer overflows can occur and, in addition, can sometimes return detailed information about the source of the failure.

A drawback of dynamic detection is the effect on runtime performance. With static analysis, the checking is performed during product testing but not thereafter. Dynamic methods must check pointer operations as they are used, thereby reducing the performance of the program, often drastically [35, 28].

A number of methods have been developed to allow dynamic detection of unsafe pointer use. Wilander and Kamkar [33] and Zhivich, Leek and Lippmann [35] have studied some of them in depth and we will explore their findings along with those of others below.

3.2.1 Canary values

Using a buffer overflow to modify an address stored on the stack, know as *stack-smashing*, is one of the most common software attacks. Generally, all the memory between the buffer and the attack target must be overwritten and, as the attacker will not know the contents of this memory beforehand, there will be a difference after the attack [33]. This can be exploited by putting known dummy data values around sensitive areas of memory and checking if they have changed before the sensitive data is used. If they have, we know there has been a buffer overflow attack. These dummy values are called *canary values*.

This approach is mainly aimed at security, specifically halting in-progress attacks [33]. This type of protection is limited to the stack and is not *fine-grained*[1] [35].

StackGuard by Immunix [5] focusses on the stored return address of a function as the target of an attack. An attack on this address would have to begin by overflowing a local variable and proceed to overwrite everything further up the stack until it reaches the return address. StackGuard places a canary value just before the return address and checks it against the expected value before allowing the function to return.

ProPolice included in GCC 4.1, takes a similar approach to that of StackGuard but with one main improvement. It rearranges the local variables so that `char`-type buffers—vulnerable to overflows due to user input—are together at the top of the stack just under the canary value [33]. This prevents them from being used to attack any other local variables, which the canary value cannot protect, while the canary value continues to guard the stored addresses.

3.2.2 Fat pointers

These are a form of pointer that hold values for the upper and lower bounds of the *referent object*[2] in addition to the current address. Whenever an operation such as pointer arithmetic is performed the resultant address is checked against the stored bounds and, if not within this permitted range, an error is raised. This gives fine-grained protection.

Explicitly recording the referent object's bounds in this manner allows fat pointer based solutions to correctly detect the case where a pointer references a valid but incorrect region of memory, in other words, an object that is not its referent object.

The biggest drawback of this method is that it changes the *ABI*[3] of the language [8, 30, 28]. This prevents the code being linked with *uninstrumented* libraries, including the standard system libraries, without wrappers to convert the fat pointer representation to and from normal pointers.

Cyclone is a dialect of C designed to restrict many of the unsafe ways in which C programs can be written. Pointer arithmetic is only allowed on pointers explicitly declared as fat:

```
int array[2];
int *@fat p = array;
*(p++) = 0;  /* array[1] = 0 */
```

The fat pointer stores the bounds of `array` and the compiler inserts bounds checks before any fat pointer is dereferenced. At runtime, these catch any buffer overflows. Thin (normal) pointers are allowed in Cyclone code but cannot be subject to pointer arithmetic. The disadvantage of Cyclone in practical use is that 5-15% of existing code must be changed to port a program from C [12].

[1] Fine-grained bounds-checking was defined in Zhivich, Leek and Lippmann as the ability to "detect small (off-by-one) overflows" [35].

[2] This is the unit of allocated memory, such as a local variable or an array, which the pointer is supposed to refer to (see Section 3.2.4).

[3] The Application Binary Interface is the interface that allows separately compiled code to interact.

Bounded Pointers added bounds-checking for C and C++ to GCC [23, 22]. It uses a three-field pointer representation holding the address and the bounds against which pointer uses are checked. Due to the non-standard ABI, it must use an instrumented version of the standard library and any external code must be recompiled with bounded pointers [8].

As the extension does not maintain a database of allocated objects—each pointer is simply checked against the locally stored bounds—it cannot provide any information for debugging such as where the bounds violation originated in the code [8].

This project has been abandoned.

CCured uses a combination of static analysis and runtime detection to ensure pointer safety [4]. Where the information obtained statically is insufficient to make safety guarantees CCured uses three types of fat pointer, `SAFE`, `SEQ` and `WILD`, with different properties. CCured uses inference rules to determine which pointer representation should be used.

The inference rules are not always able to correctly determine the representation that should be used for a particular pointer, requiring the programmer to rewrite 1–2% of the code [35]. The programmer must also write wrappers to interface with uninstrumented libraries, although wrappers are provided for part of the standard C library.

3.2.3 Allocation maps

An alternative to explicitly matching a pointer to a particular allocated object is to just record which regions of memory have been allocated and flag an error if a pointer passes outside one of these. This can be done by intercepting memory allocations and deallocations to maintain a map of the memory. Memory accesses are also intercepted and the region to be addressed is checked against the map.

The advantage of such a technique over explicitly linking a pointer to an object with the fat pointer approach is that the pointer representation is unchanged removing the ABI compatibility problems. The disadvantage is that it is not possible to detect a problem where a pointer is accidentally incremented past the end of an object to point to another valid object allocated adjacent to the first (see Mudflap below for details) [30].

Mudflap introduced in GCC 4.0 [26], maintains an *object tree* that records information about all valid objects [34]. When a pointer is dereferenced the address is checked against these objects and, if it is not found to be within one, an error is raised. This is achieved by adding calls to the `libmudflap` library into the GCC intermediate representation of a program [8].

While the information stored about the objects allows for verbose error message—useful for debugging—it is not able to determine which object a particular pointer is actually supposed to refer to (see referent objects, Section 3.2.4). In effect it is just a sparse representation of the maps maintained by tools such as Purify (see below). This can make error detection a bit hit-and-miss as referencing an incorrect but valid object or a buffer overflow that overwrites another valid object will not be detected [34]. An example of this (taken from the GCC mailing-list [9]) is shown in Listing 3.1. As arrays `a` and `b` are allocated adjacently on the stack, and therefore form one contiguous block of valid memory, Mudflap cannot detect the violation that occurs half-way through the loop as pointer `i` strays from `b` into `a`. See Section 5.3 for more details.

Valgrind is a popular tool for testing various aspects of a program's execution by deconstructing the compiled binary and running it in a simulator [29]. Its Memcheck tool intercepts calls to `malloc`, `free`, `new` and `delete` hereby maintaining a record of which memory has been

Listing 3.1: Illegally traversing adjacent buffers

```
int main()
{
    int a[1024];
    int b[1024];

    for (int* i = &b[0]; i < &a[1024]; i++)
        *i = 0;
}
```

allocated. It also augments the program instructions with code to keep track of memory accesses. In this way it can validate every heap memory access against the recorded regions and detect if an attempt was made to access invalid memory.

The penalty for this style of simulated execution is a slowdown factor of 25–50 times [35].

Purify is a commercial tool for verifying the memory accesses of a compiled program and works in a similar way to Valgrind [13, 25]. It inserts a function call before ever memory access which maintains a map of the state of each byte of memory. Storing the three possible states of *unallocated*, *allocated-uninitialised* and *allocated-initialised* requires two bits meaning at least a 25% memory overhead.

While able to catch many bugs, Purify, like Valgrind and Mudflap, cannot catch cases where incorrect pointer arithmetic results in a pointer straying into the region of memory allocated to another valid object [21].

3.2.4 Referent objects

A compromise between fat pointers and allocation maps can be found with the realisation that if

(1) every pointer points at a particular valid object

(2) pointer operations are only permitted if the result still points at the same object as (1)

then the relationship between a pointer and its referent object can be maintained without storing it in the pointer: it can be looked up in a separate database when needed. As long as a record of every allocated object is maintained in the database, the address held in a pointer must be found to refer to one of them.

These principles are the basis for several bounds-checking solutions that maintain a database of all the allocated objects and track pointer operations to ensure not only that they do not produce pointers to unallocated regions of memory but also that existing valid pointers are not used to produce pointers to regions of memory belonging to other objects. Two of these are explored below.

Jones and Kelly developed the referent object concept [21] and created a patch for the GNU C Compiler that used it to perform bounds-checking [20, 16]. The major advantage of this method over the earlier fat pointer methods was its ability to link seamlessly with unchecked libraries due to an unchanged ABI. This greatly simplifies its use and means that mature code can be left unchecked for efficiency while new code is checked for safety.

Listing 3.2: Deriving an in-bounds pointer from an out-of-bounds one

```
int main()
{
    int array[2];
    int* out_p = array + 5 * sizeof(int); // 1st new pointer is out-of-bounds
    int* in_p  = out_p - 4 * sizeof(int); // 2nd new pointer is back in-bounds
}
```

The interpretation of what constitutes a valid pointer follows the ANSI C standard very strictly. A pointer can point one byte beyond the bounds of its referent object as this address is often used to determine when a loop has reached the end of an array. Although this would appear to break the rules given earlier, all objects are padded by one byte to comply with these. Any other out-of-bounds addresses—specifically, the case where an out-of-bounds pointer is used to calculate an in-bounds pointer (see Listing 3.2)—are not permitted [28]. While strictly correct, the effect of this was to break compatibility with much existing software. Ruwase and Lam found that 60% of the programs they tested failed because of this reason.

CRED Ruwase and Lam, coincidentally with Suffield who took a similar approach, improved upon the Jones and Kelly bounds-checker by adhering less strictly to the standard for out-of-bounds pointers [28, 30]. Instead of preventing the use of out-of-bounds pointers, they developed an alternate representation that allows them to be used in address calculations but raises an error if they are dereferenced. Unlike for in-bounds pointers, the alternate representation does explicitly store which referent object is associated with it but only while the pointer is out-of-bounds. Ruwase and Lam thought it extremely unlikely that correct code would pass an out-of-bounds pointer to a seperately compiled library.

Security vulnerabilities arise when outside input overflows the bounds of a buffer meant to hold it. Ruwase and Lam added the facility to limit the bounds checking to character buffers thus achieving a significant optimisation while still catching all pointer uses that could lead to exploits.

Zhivich, Leek and Lippmann showed CRED to be particularly effective at detecting bounds errors (90% detection rate) [33] while Ruwase and Lam's own analysis showed that CRED could successfully detect all twenty buffer overflow attacks presented in Wilander and Kamkar which, at the time, no other tool could do.

3.3 Summary

With the exception of canary values—which, due to their security focus, are aimed at protecting the target of a buffer overflow rather than preventing it at the source—all the approaches covered provide ways of tracking which memory has been allocated and attempt to restrict access to these areas. These have been summarised below.

Fat pointers Intended referent is held for every pointer (as bounds).

- Pointers hold own bounds
- Checked against these whenever dereferenced

They offer fine-grained bounds-checking that is able to cope well with adjacent blocks of valid memory. The changed ABI makes their use with uninstrumented code and system libraries cumbersome. The lack of a global object database can lead to rather limited debugging output.

Referent objects Intended referent not held for every pointer individually. Objects held globally.

- Database of objects holds bounds
- Bounds looked up at every pointer calculation or dereference
- Checked against bounds of referent object

The bounds-checking is fine-grained and able to distinguish adjacent valid objects from each other. Mixing checked and unchecked code is easy and encouraged as it allows mature code to run fast and new code to run safely. Excellent debugging information is available due to the global object database and the ability to associate a pointer with an intended referent.

Allocation maps Intended referent not held at all.

- Map of valid regions holds bounds
- Bounds looked up at every pointer dereference
- Checked against bounds of all objects

Though fine-grained, the bounds-checking is unable to distinguish between adjacent valid regions of memory leading to imperfect detection. The ABI is unchanged allowing instrumented and uninstrumented code to be mixed freely.

3.3.1 Current state-of-the-art

Compatibility Mudflap is integrated into the language independent *middle-end* of the GCC 4 release. This gives it the potential for excellent compatibility with both C and C++, though problems have been reported with programs using C++ objects such as vectors [17] (see Section 5.7).

Detection Zhivich, Leek and Lippmann found CRED to have an excellent error detection rate, able to prevent around 90% of overflows tried [35]. Ruwase and Lam showed that it could detect all 20 overflow tests developed by Wilander and Kamkar [28, 33]. No other tool was capable of this.

Performance ProPolice was shown to have a very low performance overhead of 0%–20% when benchmarked by Zhivich, Leek and Lippmann with the next lowest score (held by CRED) being 40%–2930%. However, the level of checking offered by ProPolice is very limited—stack-smashing protection. Mudflap claims an overhead of 25%–350% on some standard benchmarks but this has not been subject to a comprehensive analysis and comparison.

Intra-object bounds-checking A limitation of all the tools is that bounds-checking is restricted to *allocation level* objects; those allocated explicitly on the heap or implicitly on the stack. These objects may be instances of a `struct` or `class` which can contain member variables and other objects but currently this type information is not taken into account and they are treated simply as byte buffers. While a bounds error will be detected if one of the member variables overflows past the end of the containing object, this gives considerable leeway for bounds errors to go undetected especially if the container is large.

Pool Allocation optimisation Dinakar and Adve applied the memory partitioning technique, Automatic Pool Allocation, to achieve dramatic performance improvements—down to an average of 12%—with CRED [7]. Their technique divides the object tree into several smaller trees based on object type. This greatly reduces the time needed to look up a pointer's referent object.

3.3.2 Where we aimed to go from here

Tools existed to perform bounds-checking for C and C++; tools that detected fine-grained bounds errors even with adjacent objects; tools that allowed mixed checked and unchecked code; tools that worked correctly with non-standards-compliant pointer use and tools that provided excellent debugging information to assist the programmer in their task. No tool, however, did all of these things.

We aimed to create a tool that combined these features and also offered intra-object bounds-checking which no other tool could do. While we were broadly successful in achieving the first aim, we did not manage to implement the second. The next chapter outlines the plan we made for achieving our goals and the later chapters document and evaluate our progress.

4 Plan

We aimed to develop as a proof of concept a method by which bounds errors due to incorrect pointer usage could be prevented and reported in C and C++ programs. We focussed primarily on suitability for use during development. As such there were a number of desirable features.

- Fine-grained bounds-checking: off-by-one
- Informative error messages for easy debugging
- Backwards-compatibility with existing software
  - All necessary information can be inferred from existing code
  - Allows non-standards-compliant OOB pointer use
  - Works with C as well as C++
- Ability to mix with unchecked, precompiled code
- Detection of intra-object bounds errors (see Section 3.3.1)

As CRED and Mudflap had the best feature sets, we chose them as the basis for further development. CRED began as an extension to GCC 3 and, as was most appropriate at the time, operates in the language-specific *front-end*. Although CRED continued in this manner with GCC 4, we felt that it would be most appropriate to move the functionality to the language-independent *middle-end*, especially as we are aiming this at both C and C++. Mudflap is implemented in the middle-end and its code is likely to be a good guide as it performs broadly similar tasks.

Efficiency and performance were not project goals as they did not further the proof-of-concept and would greatly complicate development (see Section 5.9).

In order to divide the work into manageable chunks and to facilitate progress tracking, we defined a set of milestones. Ass GCC turned out to be far more complex and harder to work with than expected we achieved the first three milestones. The fourth one is likely to be the hardest to complete as it will require treating objects—currently dealt with at allocation level—fundamentally differently from all other bounds-checkers.

4.1 Milestones

Each milestone has a list of goals to be achieved before it was considered complete. This helped to focus the development and prevented a build-up of unsolved problems. Where appropriate, non-goals were also been defined to limit the scope of a particular milestone, helping development and testing to progress incrementally. Lastly, the tests for each milestone are listed. Some of them already existed and others were created as part of the milestone.

Milestone 1: Basic C bounds-checker

Reimplement basic bounds-checking in the Jones and Kelly referent object style. Rather than modifying the language front-end as in Jones and Kelly, the code should be moved to the language-independent middle-end.

Although in theory language-independent, the focus is on getting this to work correctly with simple C programs. Allowing the use of out-of-bounds (OOB) pointers to calculate in-bounds ones (as required by many real-world programs and implemented in CRED) is not necessary.

Goals

- Bounds-checking simple examples written in C
- Implementation in GCC 4 middle-end

Non-goals

- OOB pointer calculations
- C++ programs

Tests

- Toy examples of basic bounds errors
- NDSS'03 buffer overflow attack tests in C by Wilander and Kamkar [33]
- Test cases and models of real-world vulnerabilities in C programs by Zhivich, Leek and Lippmann [35]

Milestone 2: Advanced C bounds-checker

Add support for correct handling of calculations using OOB pointers, as done in CRED [28] and Suffield [30], allowing correct bounds-checking for complex C programs.

Goals

- Bounds-checking complex C code including real-world programs
- Support for calculations using OOB pointers

Non-goals

- C++ programs

Tests

- All the tests used in the previous milestone
- Toy examples of complex bounds errors including code using off-by-one addresses and in-bounds pointers derived from OOB pointers
- Selection of real-world programs written in C: compile and run their testsuites

Milestone 3: C++ bounds-checker

Move the focus to compiling and detecting bounds errors in C++ programs. Although using language-independence of the middle-end, care should be taken to properly handle any differences between C and C++. At the same time we wish to retain the ability to compile and bounds-check C code therefore regression testing is crucial.

Goals

- Bounds-checking of complex C++ code
- Bounds-checking of complex C code

Non-goals

- Intra-object bounds-checking

Tests

- All the tests used in the previous milestones (ensures backwards compatibility with C)
- C++ versions of tests used in previous milestones
- Toy examples of complex bounds errors in C++
- Selection of real-world programs written in C++: compile and run their testsuites

Milestone 4: Intra-object bounds-checking

A method should be developed to apply a similar level of fine-grained bounds-checking to members of objects as is applied to the allocation level objects (see section 3.3.1). It should be possible to disable this level of bounds-checking using compiler flags.

Goals

- Fine-grained bounds-checking of objects within an allocation level object

Tests

- All the tests used in the previous milestones (prevents regression)
- Toy examples that test cases where a member variable overflows without breaching bounds of containing object.

Milestone 5: Leverage GCC 4 optimisations

Make use of the advanced optimisation framework in GCC 4 to reduce the overhead caused by bounds-checking. For instance, if a pointer is used repeatedly, say in a loop, and neither the pointer value nor the object it references are changed then it should be possible to hoist the code that performs the bounds-checking to a point just before the loop.

Goals

- Take advantage of optimisation facilities in GCC 4 to reduce bounds-checking overhead

Tests

- Time how long is needed to run the testsuites of some real-world applications

Milestone 6: Application of Pool Allocation theory

Apply the Automatic Pool Allocation techniques (see section 3.3.1), used to considerably reduce the runtime overhead of CRED, to our solution. This would help to improve its suitability for use in deployed code.

Goals

- Incorporate the Pool Allocation technique into the bounds-checker

Tests

- The same benchmarks as the previous milestone
- The set of nine benchmarks and three real-world application used in the Automatic Pool Allocation extension to CRED

4.2 Testing

Several analyses of existing bounds-checking solutions have been performed, some of which included testsuites. The most appropriate for our needs are those by Wilander and Kamkar [33] and Zhivich, Leek and Lippmann [35] who have both kindly provided us with their source code. We used these heavily to assess our bounds-checking capabilities.

Wilander and Kamkar in their analysis of buffer overflow attacks, found two attack *techniques*, two attack *locations* and four attack *targets*. Combining these in all practical combinations they created a comprehensive testbed of twenty attacks, eighteen of which we have the source code for.

Although the primary focus of these tests is security, they are still useful tools to test general bounds-checking ability.

Zhivich, Leek and Lippmann present an extended set of tests in two parts. Firstly is a set of 55 test cases that test the "ability to detect small and large overflows in different memory regions". Secondly they created models of fourteen historic vulnerabilities that existed in the real-world programs `bind`, `sendmail` and `wu-ftpd`.

These tests are an excellent indicator of how effectively a bounds-checker is working and allows us to focus on successfully detecting one type of bounds error at a time.

5 Implementation

The following chapter describes how we went about adding the CRED functionality to the existing Mudflap bounds-checker. We begin with an introduction to GCC 4; the most popular open-source compiler for C and C++. Following that, we briefly describe how Mudflap integrated into this, how we based our work on it and what changes were made. We explain some of the challenges posed by the increased requirements of referent object checking and invalid pointer handling.

5.1 The GNU Compiler Collection

GCC began as a C compiler but has evolved to support a large number of languages and platforms. The latest version of GCC, version 4, is vast[1] and complex. Compiling it takes about 3 hours on a 2 GHz processor. Coping with this complexity was the major difficulty of the development effort.

GCC 4 is divided into three functional parts: the language-specific front-ends, the language-independent middle-end and the platform-specific back-ends [3]. Each language supported by GCC has its own front-end which parses the code and generates an intermediate representation of the program. The middle-end processes this representation through a chosen sequence of optimisation passes. The back-ends then produce machine code suitable for the target platform.

Mudflap is implemented as an optimisation pass in the middle-end where it can manipulate the internal representation of a program with the welcome side effect that changes made here are language-independent; hence its support for C++. CRED is implemented in the C front-end as it was developed before the middle-end was introduced to GCC. Rather than following this approach, which would have required us to create different versions for C and C++, we added our work to GCC as an optimisation pass based on Mudflap. As a result almost no code has been taken from CRED—only its method—while our pass retains a sufficiently similar structure to Mudflap that it would be feasible to merge them in the future.

5.1.1 GIMPLE: The intermediate representation

The representation of the code in the middle-end is as a tree form called GIMPLE. The simplified nature of the *gimplified* program makes it ideal for analysing pointer use and for inserting checks in the right places. One guarantee of the GIMPLE form is that statements have a maximum of three operands. For instance, the following statement:

```
p = a + b[i] + q
```

[1]The source code for GCC 4.1.1 contains 30,000 files totalling 250 megabytes.

is split into three simpler statements, using temporaries:[2]

```
T1 = b[i];
T2 = T1 + a;
p = T2 + q;
```

Assuming that p and q are pointers, a and i are integers and b is an integer array, the statement assigns to p the result of incrementing pointer q by an integer offset. This is a typical example of the types of dangerous operation bounds-checking is required for. The gimplified form of the statement is much easier to manipulate for bounds-checking than the previous one. Each line can be processed separately without needing to pick complex statements apart.

A check is inserted before the array access on the first line to make sure that the index lies within the array's registered bounds. The last line is replaced with a check that ensures the addition of the offset does not change the referent object of pointer q and performs the arithmetic if the check succeeds.

5.2 Getting started

Faced with the complicated Mudflap code, we had the choice of spending a long time studying it to understand how it worked or learning as we went. We opted for the latter. First we added a simple pass to GCC [3]. It consisted of a function to be executed when the -fbounds-checking flag was passed to GCC and a struct describing the properties of the pass:

```
static unsigned int tree_bounds(void)
{
  fprintf(stderr, "Hello world! I'm a bounds-checker\n");
  return 0;
}

struct tree_opt_pass pass_bounds =
{
  "bounds",                     /* name */
  tree_bounds,                  /* execute */
  PROP_gimple_any,              /* properties_required */
      ...
};
```

As we began to understand how pieces of Mudflap worked, we extended this pass to perform the necessary bounds-checking.

5.3 Existing Mudflap functionality

The Mudflap implementation is in two halves as shown in Figure 5.1: the compiler pass that instruments the code by inserting the necessary checks and the runtime library that the instrumented code calls into. The checking is performed in the library using a database of allocated regions that have been registered with it. The majority of these allocated regions are from two sources: local variables and heap allocation.

The local variables that need to be registered are those whose address is used, either explicitly or implicitly. The others are ignored as, without taking their address, they can never be

[2]This is only an attempt at describing the GIMPLE representation in a C-like way. GIMPLE code is a tree.

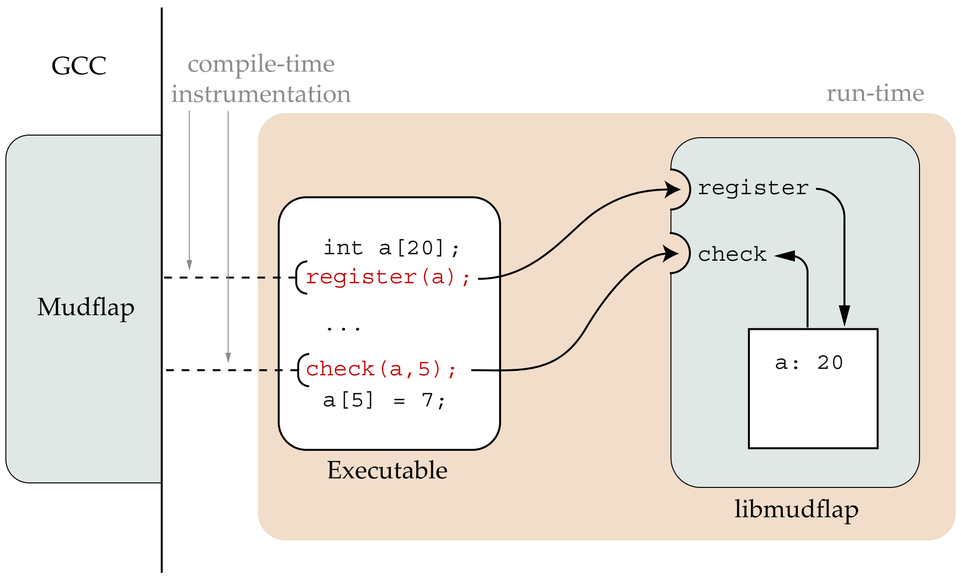

Figure 5.1: The structure of Mudflap in two parts

meaningfully accesses via a pointer. At the start of each function, the Mudflap compiler passes insert a call to the library that registers these variables:

```
void f(int n)
{
  int a[4];  int i=3;  int *p;

  __mf_register (&a, 16, ... );
  __mf_register (&i, 4, ... );

  p = &i;     /* Address of i used explicitly */
  a[n] = 7;   /* Address of a used implicitly */
}
```

As a local variable should not be accessed once a function has returned, Mudflap also inserts a call to the library that unregisters the local variables at the end of the function (not shown).

Memory allocated on the heap is treated differently. These allocations are not local to the function. Pointers to the allocated regions can be passed around and used throughout the program in places entirely divorced from where they were originally allocated[3]. To register and unregister these regions Mudflap provides modified versions of the usual heap allocation functions: `malloc\free` for C and `new\delete` for C++:

```
void * malloc(size_t c)
{
  void *result = __real_malloc (c);
  if (result) __mf_register (result, c, ... );
  return result;
}
```

These make the call into the library to register the region when it is allocated and the corresponding call to unregister when it is freed.

[3]This is, in fact, the main cause of memory leaks.

When a pointer is dereferenced, either explicitly or by array access, Mudflap inserts a call to the library that checks whether the intended target region is within a registered region of memory. If not, a bounds violation is reported.

```
void f(int n)
{
  int a[4];  int i = 3;  int *p = &i;

  __mf_check (p, 4, ... );
  *p = 0;    /* Explicitly dereference */

  __mf_check (&a, n, ... );
  a[n] = 7;  /* Dereference via array operator */
}
```

Mudflap supports a relatively simple form of checking, akin to that of Valgrind (see section 3.2.3). If the target region is found to lie completely within **any** registered region, the check is successful. It takes no account of which region the pointer was originally associated with. This can allow bounds errors to go undetected if a pointer strays from one allocated region into another contiguously allocated one.

5.4 Referent objects: Strengthening the invariant

Making an explicit association between a pointer and an allocated region, while solving the problem of straying into other regions, requires a mechanism such as fat pointers (see section 3.2.2) that causes problems when linking with unchecked code. The referent object approach tackles this by preventing a pointer from leaving the region it refers to in the first place. In order to enforce this strengthened invariant, we added pointer arithmetic checks to the existing Mudflap functionality and changed how the dereference check worked.

A legal pointer arithmetic operation must have exactly one pointer operand and an integer offset resulting in a pointer value[4]. We refer to this operand as the *base pointer*. Our compiler pass replaces the pointer arithmetic operation with a call to the library that first checks that the result of the arithmetic points to the same region as the base pointer and then returns this result if the check is successful:

```
int a[4];  int *p = &a[0];  int *q

~~q = p + 1;~~
q = __bounds_arith (p, p + 1, ... );  /* base pointer: p */

*q = 0;  /* a[1] = 0 */
```

The call to the library **replaces** the arithmetic rather than just verifying it due to the out-of-bounds pointer handling explained in section 5.5.

When Mudflap checks an impending pointer dereference (section 5.3) it verifies that the entire target region, from the pointer value for a length of sizeof(type), is contained in registered regions. We take a slightly different approach. Firstly, using the pointer we lookup the corresponding registered region (referent object) in the database. Having found it, we verify that the end of the target region lies within the bounds of this same referent object. If this is

[4]Note that this does not prevent the subtraction of two pointers which is another type of operation entirely—pointer difference—and whose result is an integer.

the case we can be sure that the entire target region lies within the referent object. As we have a stronger invariant than Mudflap we do not need to check any other registered regions. If either the lookup or the bounds verification fails we halt the execution and report a bounds violation. For example, the following incorrect program:

```
int main()
{
  int a[2];
  a[2] = 0;
  return 0;
}
```

produces this violation when compiled with bounds-checking and executed:

```
>./viol
*******
bounds violation (check/write): time=1181254695.211243 ptr=0xbfeef6d0 size=4
pc=0xb7f72124
location='viol.c:4 (main)'
error='Start of access is out of bounds'
```

The referent object lookup should never fail due to our invariant that pointer arithmetic resulting in an invalid pointer is not allowed, however due to out-of-bounds pointer calculations (Section 5.5) and foreign pointers from unchecked code (Section 5.6) this cannot be assumed.

5.4.1 Practical problem: finding the pointer operand

Implementing pointer arithmetic checking was complicated greatly by that way GCC represents pointer arithmetic internally. Pointer addition is represented by the PLUS_EXPR GIMPLE node in the same way as any other numeric addition. The restriction on such addition is that both operands must have the same type. Remembering that pointer arithmetic has one base pointer operand and an integer offset, GCC coerces both values to pointer type for the addition:

```
q = p + i;
```

is internally transformed as follows:

```
int i = 3;
unsigned int T1 = (unsigned int) i;
unsigned int T2 = T1 * 4;
int *T3 = (int *) T2;
int *q = T3 + p;
```

By the time the arithmetic takes place, on the last line, both the operands have become pointers. The effect of this is to make it impossible to reliably decide which of the operands is the base pointer and which the offset.

As a temporary solution, we have developed a heuristic that finds the correct operand in most cases. Starting from the arithmetic statement, it looks back for statements that assign to either of the operands: the dominators. If one of these assigns to the operand from a non-pointer type, the other operand must be the base pointer as a real pointer cannot be derived from a non-pointer (ignoring the possibility of casts).

This method cannot trace back further than the current basic block but this should be sufficient for cases of internal pointer coercion. In the cases where this method fails, the heuristic falls back to user internal node properties to try to decide. With more intelligent code it may have been possible to make this method correct in all cases but as time was limited and, as it did not further the proof-of-concept, this was not a priority.

A development branch of GCC, pointer_plus, is devoted to creating a dedicated representation for pointer addition that keeps the two operands separate [18].

5.5 Out-of-bounds pointers

The ANSI C standard states that a pointer must always point at a valid object (allocated region), NULL or one byte past the end of an array [19, p.37]. In practice much real-world software does not adhere to this restriction (see section 3.2.4) and invalid (out-of-bounds) pointer are used in the calculation of valid ones. Although not standards-compliant we don't think that these should be considered bounds violations as long as the invalid pointers are not dereferenced. In order to allow these calculations to work correctly while maintaining our invariant that a pointer cannot become invalid, we replace a pointer that goes out-of-bounds with an alternate representation and allow the program to continue rather than reporting a violation. Although the alternate representation breaks the ABI compatibility necessary for use with unchecked code (section 5.6), Ruwase and Lam did not find a single case of an out-of-bounds pointer being passed to unchecked code in the millions of lines of code tested for CRED [28, §2.2].

Our alternate representation holds the invalid pointer value, the value the pointer held before arithmetic caused it to go out-of-bounds and the location in the source code where this arithmetic took place for use in debug messages:

```
struct __bounds_oob_ptr
{
  uintptr_t ptr;              /* Current pointer value */
  uintptr_t last_valid;
  const char* arith_location;
};
```

When arithmetic takes a pointer out-of-bounds, we instantiate one of these out-of-bounds pointer objects (OOB) and replace the value of the pointer with its address. The old value is not lost as it stored in the OOB.

If, when checking a pointer that is to be dereferenced (see section 5.4), the referent object lookup fails we check to see whether the pointer is in fact pointing to an OOB. If so, we report a bounds violation. The information collected in the OOB allows us to produce more informative error messages than Mudflap is capable of. To illustrate, the following code:

```
int main()
{
  int a[2];  int *p = a;

  p = p + 2;
  *p = 0;

  return 0;
}
```

produces an error message pinpointing not only which use of the pointer caused the violation but also where the pointer first went out-of-bounds and, for stack variables, what the referent object was called:

```
>./viol
*******
bounds violation caused by earlier pointer arithmetic:
        The arithmetic took place at viol.c:5 (main)
        The violation occured at viol.c:6 (main)
        Before going out-of-bounds the pointer referred to viol.c:3 (main) a
        time=1181317406.431297 ptr=0xbfaf12cc size=4 pc=0xb7f57124
```

When pointer arithmetic is intercepted a similar referent object lookup takes place. If this fails we again check if the pointer points to an OOB and either update the OOB to reflect the new out-of-bounds value or, in the case that the pointer arithmetic brought it back within bounds, replace the OOB with a standard pointer value.

5.5.1 Practical problem: Other pointer operations

One consequence of the using OOBs is that all operations that use pointers now need to be intercepted in order to work correctly. This includes pointer comparison, difference, address-of and casts in addition to the existing dereference and pointer arithmetic interception.

As an example of why this is needed, consider the following typical example of a loop traversal using a pointer:

```
int a[100];  int *p
for (p = a; p < &a[100]; p++)
    *p = 0;
```

The right-hand-side of the loop bound condition, `&a[100]`, is an out-of-bounds address[5] and an OOB will have been created for it, replacing its value with the OOBs address. This causes a problem with the comparison as we do not want to compare p to the OOBs address, rather to the out-of-bounds address stored in the OOB. For this reason, we replace pointer comparisons with a library call that retrieves the address from the OOB, if one exists, and performs the comparison.

5.6 Foreign pointers

One of the main advantages of the referent object approach is that, due to the unchanged pointer representation, it is possible to link bounds-checked code with unchecked code. Pointers passed to checked code from unchecked code, which we call *foreign pointers*, are unlikely to refer to objects that have been registered in our database yet they must not produce bounds errors when used. We deal with this using detection-by-failure: any pointer that fails a referent object lookup and does not point to an OOB is considered to be foreign and used as-is, unchecked.

5.7 C++ Standard Template Library

Mudflap has great problems with C++ code that makes use of the Standard Template Library. Even very simple programs produce multiple violations. The following example taken from the GCC bug report on the issue [17] reports seven violations:

[5]The common use of such addresses in loop bounds is the reason that ANSI C permits pointers to be one byte out-of-bounds.

```
#include <vector>

int main() {
  std::vector<int> v;
  v.push_back(1);
}
```

At first, we encountered the same errors but we tracked it down to the way the Mudflap code registers a function. Parameters and local variable declarations whose address was ever used (i.e. a pointer to them was created) were registered but the return value was not. This proves to be a problem in return-by-value cases, common in C++, where the object is returned directly without being stored in an intermediate variable. For example the following STL code:

```
iterator begin() {
  return iterator (this->_M_impl._M_start);
}
```

is transformed in the intermediate representation:

```
iterator begin()
{
  comp_ctor (&<retval>, &this->_M_impl._M_start);
  return <retval>;
}
```

where `comp_ctor()` is the constructor of `iterator`. Inside `comp_ctor()` the `iterator` constructs itself using its `this` pointer which is actually `<retval>` but this has not been registered by `begin()`.

Once we fixed our code so that return values were properly registered, C++ STL code did not cause false violations. We applied the same fix to Mudflap and it appears to have fixed the problem there too. A patch has been submitted and we are awaiting feedback.

5.8 Memory alignment

While, in general, our bounds-checking is fine-grained (see Chapter 3) in some cases this is not so.

Due to the benefits of aligned memory access `structs` are padded such that, given an array of them, all elements are aligned [27]. As shown in Figure 5.2, the following `struct` is padded by 2 bytes on a 32-bit architecture:

```
struct bob {
    int i;
    short j;
};
```

otherwise member `i` would be unaligned in every second element of an array of these.

When we register an object in the database we use the result given by `sizeof()` to determine its bounds. As this value includes the padding, any bounds errors that remain within the padded region are not detected.

5.9 Casualties of time constraints

Mudflap uses a lookup cache to speed up the common case of searching the database of registered memory [8, p.59]. One unfortunate effect of this is that it both requires and produces

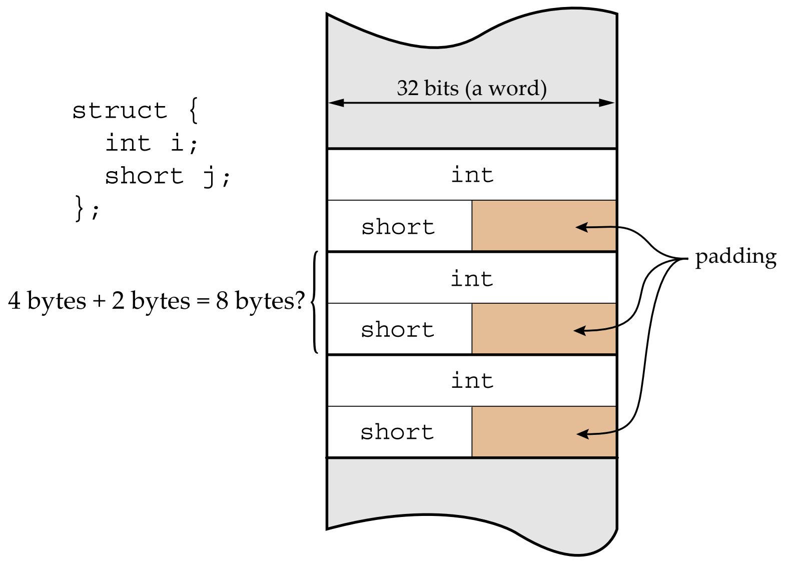

Figure 5.2: Struct padding to force word-alignment of members

vastly more complicated code. Due to limited and in the interest of furthering the proof-of-concept we decided early-on that this would not be included in OOPS. The drastic effect this has on performance can be seen in Section 6.2. Other such casualties include threading support and the abundance of runtime options available in Mudflap. Due to the similar structure of our code, these should not be hard to add back as time permits. A solution to the problem of intra-object bounds-checking (see Section 7.1) would also solve this problem.

6 Evaluation

6.1 Bounds-checking ability

We used two testbeds, designed expressly for testing the effectiveness of bounds-checkers, to assess the effectiveness of MIRO. The first, a set of tests kindly made available to us by the authors Wilander and Kamkar (see Section 4.2), test the ability to prevent eighteen types of buffer overflow attack [33]. This suite is aimed at the security applications of bounds-checking. The second testbed, by Zhivich, Leek and Lippmann (see Section 4.2), demonstrates our capacity for detection when faced with real-world examples. They took fourteen historic buffer overflows in the popular `bind`, `sendmail` and `wu-ftpd` programs and created small models, suitable for testing, from them. They also created models of each with the overflows patched [35]. Correct bounds-checking should report a violation on the unpatched models but not do so for the patched ones.

During development we created an extensive set of testcases exploring aspects of the C and C++ languages and how we should respond to correct and incorrect code with bounds-checking enabled. Many of these were written in response to problems encountered and solved to prevent regression. This testsuite was absolutely crucial to the successful development of MIRO as we guided our development to make tests pass.

Real-world programs are the ultimate test of bounds-checking effectiveness, however, getting useful results can be difficult. They can easily test the basic, but challenging, ability to successfully build a program; most problems in this regard come from linking (see Section 6.1.5). Testing whether bounds errors are caught successfully is harder as it is difficult to know whether a reported bounds violation is truly a violation or a false positive and, even more so, whether a clean run truly indicates the absence of bounds violations or is caused by flaws in the checking. The models of real-world violations (section 6.1.2) are perhaps the best approach to this and go quite some way towards demonstrating the ability of MIRO to detect real-world flaws accurately.

6.1.1 Wilander and Kamkar

These tests demonstrate eighteen different ways of exploiting buffer overflows by injecting code that tries to launch a new shell. Such a shell would have the same permissions as an exploited program and, were this to have root access, a malicious exploit could gain complete control of the system.

Both Mudflap and MIRO halted all possible exploits, producing a violation message indicating its location. Our tests showed that GCC alone was able to prevent six of the exploits in unchecked code producing an executable that could either continue unharmed or that caused a segmentation fault. Six of the exploits were not possible on our test platform[1]. See Table 6.1

[1]Ubuntu Linux 6.10, kernel 2.6.7-11.

Type	Target	GCC	Mudflap	MIRO
Buffer overflow on the stack all the way to the target	Param function pointer	FAIL sh	PASS v	PASS v
	Param longjmp buffer	PASS seg	PASS v	PASS v
	Return address	FAIL sh	PASS v	PASS v
	Old base pointer	PASS p	PASS v	PASS v
	Function pointer	FAIL seg	PASS v	PASS v
	Longjmp buffer	PASS seg	PASS v	PASS v
Buffer overflow on heap/BSS all the way to the target	Function pointer	np	np	np
	Longjmp buffer	np	np	np
Buffer overflow of pointer on stack and then pointing to target	Param function pointer	FAIL sh	PASS v	PASS v
	Param longjmp buffer	PASS seg	PASS v	PASS v
	Return address	FAIL sh	PASS v	PASS v
	Old base pointer	PASS p	PASS v	PASS v
	Function pointer	FAIL sh	PASS v	PASS v
	Longjmp buffer	PASS seg	PASS v	PASS v
Buffer overflow of pointer on heap/BSS and then pointing to target	Return address	np	np	np
	Old base pointer	np	np	np
	Function pointer	np	np	np
	Longjmp buffer	np	np	np

	sh	`/bin/sh` launched	seg	Segmentation fault
Key:	p	Prevented and execution continued	v	Violation reported
	np	Exploit not possible		

Table 6.1: Results of testing GCC alone, GCC with Mudflap and GCC with MIRO using the Wilander and Kamkar testbed

for the results.

6.1.2 Zhivich, Leek and Lippmann

This set of tests that model real-world buffer overflows provides a good indication of a solution's effectiveness when faced with real-world problems. Unlike the previous security-related tests where segmentation faults prevented a vulnerability being exploited here we do not consider this to be a successful result. One of our goals focus for this project was to create a bounds-checker that would ease debugging by catching bounds violations and reporting them. A segmentation fault provides no report or other debugging information and cannot be considered a bounds-checking success in the general case.

Uninstrumented code compiled with GCC failed on all the unpatched cases by either causing a segmentation fault or, more seriously, allowing the buffer to overflow but continuing with execution. This is what we expected. In tests reported by Zhivich, Leek and Lippmann they found that uninstrumented code passed six of the tests but we saw no evidence of this [35, p.6].

MIRO halted execution and reported a bounds violation at the correct location for all the unpatched tests. When checking whether the tests had halted correctly, we found that the extra information included in our reports—the location of the original arithmetic that caused the overflow—made this much easier to do than had we just had the Mudflap reports available. The patched tests executed correctly without triggering any false positives.

In contrast Mudflap, which we had expected to pass all the tests as well, failed several. Two buffer overflows in sendmail went undetected: tests s3 and s5 (see Table 6.2). Both buffers were allocated adjacent to others on the stack and due to the simple form of checking employed by Mudflap (see Sections 5.3, 3.2.3 and 6.1.3) the pointer could wander unhindered from one allocated region into another. In the unpatched s3, for example, the function `mime_fromqp()` can overflow buffer `obuf` on specific inputs:

```
void mime7to8(...)
{
    register char *p;
    u_char *obp;
    char buf[MAXLINE];
    u_char obuf[MAXLINE];
    ...
    if (mime_fromqp((u_char *) buf, &obuf, 0, MAXLINE) == 0)
    ...
}

int mime_fromqp(u_char *infile, u_char **outfile, int state, int maxlen)
{
    ... *(*outfile)++ = c1; ...
}
```

As the `buf` buffer is allocated adjacent to `obuf`, this overflow is not detected by Mudflap. At all times the `outfile` pointer references validly allocated memory that has been registered with `libmudflap` but `outfile` does not always reference the **right** bit of allocated memory. This shows the strength of the referent object approach employed by MIRO which prevents pointer arithmetic that would result in switching which region is referenced (see Section 5.4).

Mudflap fails several tests due to false positives: s2, s7 (both patched and unpatched) and f3. These are caused by use of system libraries which pass a foreign pointer into the programs. For example, in s2 the program calls the C library function `getpwent()` which returns a pointer to the desired `struct` but as this object was created by the unchecked external library it is foreign:

```
register struct passwd *pw;
pw = getpwent();
buildfname(pw->pw_gecos, pw->pw_name, buf);
```

When the program tries to use a member of the `struct` Mudflap reports a violation:

```
*******
mudflap violation 1 (check/read): time=1181492607.449880 ptr=0xb7ec3950 size=4
pc=0xb7ece31d location='recipient-ok.c:309 (finduser)'
      /usr/local/lib/libmudflap.so.0(__mf_check+0x3d) [0xb7ece31d]
      ./ge-ok(finduser+0x87f) [0x804aca2]
      ./ge-ok(recipient+0x496) [0x80492de]
```

Our solution is able to use detection-by-failure to tell that the pointer is foreign (see Section 5.6) and allows the operation to proceed unhindered.

6.1.3 Straying pointers

A known shortcoming of other bounds-checkers including Mudflap (see Sections 5.3 and 3.2.3) is that they cannot catch a pointer that incorrectly strays from one block of memory into an-

Program	Test	GCC		Mudflap		MIRO	
		unpatched	patched	unpatched	patched	unpatched	patched
sendmail	s1	FAIL seg	PASS nf	PASS v	PASS n	PASS v	PASS nf
	s2	FAIL seg	PASS nf	PASS v	FAIL fp	PASS v	PASS nf
	s3	FAIL x	PASS nf	FAIL x	PASS nf	PASS v	PASS nf
	s4	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	s5	FAIL x	PASS nf	FAIL x [a]	PASS nf	PASS v	PASS nf
	s6	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	s7	FAIL seg	PASS nf	FAIL fp [b]	FAIL fp	PASS v	PASS nf
bind	b1	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b2	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b3	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	b4	FAIL seg	PASS nf	PASS v	PASS nf	PASS v	PASS nf
wu-ftpd	f1	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	f2	FAIL x	PASS nf	PASS v	PASS nf	PASS v	PASS nf
	f3	FAIL x	PASS nf	PASS v	FAIL fp	PASS v	PASS nf

[a] Although Mudflap did report a violation on this test, it was too late. The adjacent buffer had already been overwritten. Mudflap caught the overflow once the pointer left the entire block of contiguously allocated variables.

[b] Mudflap reported a violation on this test but at the wrong place in the code. Both the patched and unpatched version reported the same false positive caused by a foreign pointer from a system library.

Key:				
	seg	Segmentation fault	x	Exploited - uncaught buffer overflow
	v	Violation reported	nf	No failure
	fp	False positive		

Table 6.2: Results of testing GCC alone, GCC with Mudflap and GCC with our bounds-checker on patched and unpatched models of real-world bounds errors

other valid adjacent block. The following example taken from the GCC mailing-list discussion of the issue [9] demonstrates the problem:

```
int a[1024];
int b[1024];

int *i;
for (i = &b[0]  ; i < &a[1024] ; i++)
     *i = 0;
```

Pointer i is initialised to the array b but the loop terminates at the end of array a. Both arrays are valid registered objects but it is very unlikely that correct code should traverse both arrays in this way. However, this does not cause a violation when compiled with Mudflap.

The referent object approach has a stronger method of checking (see Section 5.4) that ensures pointer i cannot change its referent object—it does this half way through the loop. Our referent object bounds-checker catches this mistake in the right place and prints an error message explaining where this happened in the code and what the pointer was supposed to be referring to.

6.1.4 C++ support

C++ code caused Mudflap to report many false violation message, most noticeable when code made use of the C++ STL (see Section 5.7). The following simple program would generate seven violation reports [17]:

```
#include <vector>

int main() {
  std::vector<int> v;
  v.push_back(1);
}
```

6.1.5 Real-world programs

gzip is a common file compression tool written in C [11]. It compiles and runs correctly with bounds-checking. Files can be compressed to form a valid archive which can be uncompressed, either with the same instrumented `gzip` or another tool, producing files identical to the original. Analysis of the library trace indicates that the pointer operand to an arithmetic operation is mis-guessed (see Section 5.4.1) only once. All other operations are correctly checked. Performance was very slow when compared to uninstrumented code (see Section 6.2.1).

Mesa 3D is a software OpenGL implementation written in C [24]. It compiled and ran successfully with bounds-checking but the example programs ran very slowly. This gave us a good opportunity to test mixing checked and unchecked code. We recompiled the Mesa libraries without bounds-checking and linked the checked example programs to them. This resulted in a significant increase in rendering speed: 2,007 draws/s compared to 0.01 draws/s. Only one program, an example of particle effects, still ran slowly as this was dependent on calculations performed in the program rather than the library.

When we introduced a bounds error into the example program and recompiled it a bounds violation was reported at the correct location.

Firefox is a popular web browser written in C. Unfortunately it could not be built with bounds-checking due to a compatibility issue between code instrumented with calls to our library and the XPIDL tool used in its build process.

Doxygen is a documentation generation tool written in C++ [14]. We were able to compile it with bounds-checking and when we ran it on its test documentation it reported bounds errors. After checking the code manually the violations detected are genuine. We reported them to the mailing list and are awaiting feedback.

6.1.6 Custom testsuite

We created over 250 tests during development, split across C and C++. Most had a version for both the failure case—where a bounds violation should be reported—and for correct code which should execute normally. During testing each is run first without optimisation and then with `-O2` and `-O3`. Optimisation typically produced more failures than without as GCC makes some significant changed to the representation of a program, particularly in the case of loops.

The GCC test framework treats each test as consisting of three sub-tests: compilation, execution and output. At time of writing, 1906 of 1938 sub-tests pass:

File size (KB)	Time taken (s)		
	GCC	Mudflap	MIRO
2.4	0.00	0.07	0.44
12.8	0.00	0.11	2.04
126.3	0.05	0.09	22.08
357.7	0.07	0.27	76.89
904.9	0.14	0.56	219.59
1433.6	0.23	0.78	401.59
6451.2	1.12	3.46	1843.57
26316.8	1.49	6.11	5783.78

Table 6.3: Time taken to compress files of varying sizes using `gzip`

```
                === libbounds Summary ===

# of expected passes            1906
# of unexpected failures        32
```

The only test failures not caused by optimisation are caused by our inability to catch overflows in the `struct` alignment padding region (see Section 5.8).

6.2 Benchmarks

6.2.1 gzip

We used eight files with widely varying sizes to tests the performance of `gzip-1.2.4` compiled with bounds-checking. We also verified that the contents of the resultant archive matched the original files exactly when uncompressed with a separate tool.

As shown in Table 6.3, `gzip` took considerably longer to compress when instrumented with MIRO than when using Mudflap or when uninstrumented: on average about 400 and 2,000 times slower respectively. Further investigation revealed this to be less of a problem than it might appear.

Firstly, the increase in time with respect to increasing file size is O(N) meaning there is no fundamental problem of complexity. Figure 6.1 shows how the ratio of file size to time taken varies with increasing file size. We can see that this is relatively constant for MIRO. The graphs for GCC and Mudflap vary considerably but this is likely to be caused by the difficulty of timing the tiny execution times for smaller file sizes accurately.

Secondly, looking at the execution trace, the vast majority calls to our library were to check an impending dereference or to check pointer arithmetic, not the far more expensive out-of-bounds pointer handling. This means that we are likely to regain much of Mudflap's performance if we were to implement the caching ommitted from our code (see Section 5.9) which accounts for much of its performance [8, p.59].

6.2.2 benchcpplinux

We ran a selection of tests from the `benchcpplinux` C++ benchmark suite [31] bounds-checked with our solution and with Mudflap as well as without checking. The results, given in Table 6.4, show that our bounds-checker is much slower that Mudflap or unchecked code. This matches the results of the `gzip` benchmark except that the Mudflap score is much closer to

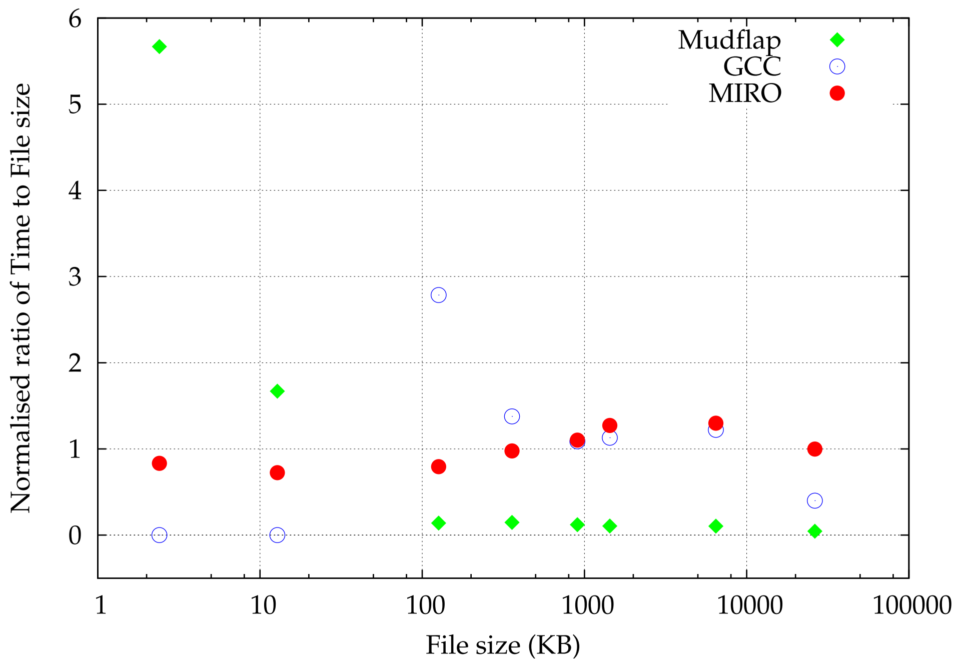

Figure 6.1: Computational complexity of gzip with increasing filesize: The results from Table 6.3 were used to calculate the ratio of time taken to file size. This was normalised by dividing each column of results by its average

Test	Score (bigger is better)		
	GCC	Mudflap	Our solution
dhrystone	536301	1458	760
inducvar-ho	970904	31508	1113
codemotion	252062	21558	457
deadcode	6765006	39961	20480
whetstone	562884	317617	2430
stringops	57	114	73

Table 6.4: Results of the `benchcpplinux` test suite

ours than to unchecked code. Mudflap reported thousands of bounds violations during this benchmark and it is likely that the overhead of printing these report on the terminal slowed it down significantly. The results remained very consistent over numerous runs.

7 Conclusion

Before this project there were two choices available: powerful referent object bounds-checking but limited to C or support for C++ with weaker methods. We have combined the best of both along with some other improvements to create OOPS: an extension to GCC 4—one of the most popular compilers in the world.

Without any changes to existing source code, a program written in C or C++ and compiled with OOPS is subject to fine-grained bounds-checking at runtime. Memory cannot be accidentally overwritten nor can vulnerabilities be exploited by malicious third-parties. Even the difficult case of accidentally using memory belonging to an adjacent object is handled correctly. Programs that, as is common, do not comply with the standards and use invalid pointer values in calculations are nevertheless bounds-checked in a safe without affecting their operation.

Life for the developer has been made easier. Debugging messages are more informative due to the extra information available with referent object tracking. Complex code, including use of the popular C++ Standard Template Library now works correctly and freely mixing checked and unchecked code is not only possible; it is encouraged. In this way a developer can bounds-check new code leaving mature code and system libraries unchecked, thereby maintaining all the performance these provide.

OOPS has a number of limitations compared to existing solutions. Most noticeable among them is performance. At the moment OOPS run significantly slower than Mudflap although it seems likely that including features such as lookup caching with improve this dramatically. There is still much scope for improvement as quite some difference remains between the facilities provided by OOPS and the level of support provided by a high-level language such as Java.

7.1 Further work

Improve stored information for heap objects Currently we store a lot of debugging information when we register a stack object such as its name and location in the source code. Unfortunately, due to the limits of using a wrapper to `malloc` etc., this information is not available for objects allocated on the heap. We could create an additional wrapper that had more parameters through which we could pass extra information. When a program is being instrumented we replace all calls to memory allocation functions with calls to our augmented wrapper which registers the information in the database. Useful information for heap objects could include the name and location of the pointer to which they were first assigned. Another approach, if replacing the calls is too problematic, could be to insert a separate call immediately after an allocation that registers the information although this would have thread-safety implications.

Detecting memory leaks As we maintain a database of all allocated objects and delete entries from it when an object is freed, any entries remaining when the program terminates will be memory leaks: objects that were allocated but never freed. We could inform the user and, if extra information is available for heap objects (see above), pinpoint exactly where the object was originally allocated and to which pointer.

Intra-object bounds-checking A limitation of our bounds-checking when compared to that of Java is that we are unable to detect violation that occur within an object. While this was a problem with C it is more serious for C++ where classes are heavily used. Implementing this would require support for registering nested objects: treating an array member a `struct` or `class` as a separate object. As multiple registered object could now have overlapping addresses, each object's entry would have to tagged with it's type so that they can be distinguished from each other. This is likely to be difficult across separately compiled files.

Lookup caching Mudflap employs a complex lookup cache which significantly reduces the overhead of searching the database of objects—the main performance bottleneck (see Section 5.9. This was not incorporated into MIRO due to its complexity but may be the most beneficial and urgent future improvement.

References

[1] **TIOBE Software BV**. TIOBE programming community index, December 2006. URL http://www.tiobe.com/tiobe_index/index.htm.

[2] **CERT Coordination Center**. Statistics 1988-2006. Website. Accessed January 2007, URL http://www.cert.org/stats/cert_stats.html.

[3] **Zbigniew Chamski**, **Albert Cohen**, **Sebastian Pop**, **Georges Silber** and **Ayal Zaks**. HiPEAC tutorial on middle-end and back-end program manipulation in GCC. In *HiPEAC GCC Tutorial*. May 2006 [Presentation], URL http://www.hipeac.net/gcc-tutorial.

[4] **Jeremy Condit**, **Matthew Harren**, **Scott McPeak**, **George C. Necula**, and **Westley Weimer**. CCured in the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, San Diego, California, USA. ISBN 1-58113-662-5, 2003 pp. 232–244. doi:http://doi.acm.org/10.1145/781131.781157. URL http://www.cs.berkeley.edu/~jcondit/ccured-pldi-2003.pdf.

[5] **Crispin Cowan**. Stackguard. Website. Not accessible January 2007, URL http://immunix.org/stackguard.html.

[6] **Crispin Cowan**, **Perry Wagle**, **Calton Pu**, **Steve Beattie** and **Jonathan Walpole**. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition*, volume 2. ISBN 0-7695-0490-6, January 2000 pp. 119–129. URL http://ieeexplore.ieee.org/iel5/6658/17794/00821514.pdf?isnumber=17794&arnumber=821514.

[7] **Dinakar Dhurjati** and **Vikram Adve**. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*. ACM Press, New York, NY, USA. ISBN 1-59593-375-1, 2006 pp. 162–171. doi:http://doi.acm.org/10.1145/1134285.1134309.

[8] **Frank Ch. Eigler**. Mudflap: Pointer use checking for C/C++. In *GCC Developers Summit*. Red Hat Inc., Ottawa, Ontario, Canada, May 2003 pp. 57–70. URL http://gcc.fyxm.net/summit/2003/mudflap.pdf.

[9] **Frank Ch. Eigler** and **Chris Scott**. Re: mudflap problem. Mailing-list, January 2005. From the GCC mailing-list, URL http://gcc.gnu.org/ml/gcc/2005-01/msg01655.html.

[10] **Hiroaki Etoh**. ProPolice. Website. Accessed January 2007, URL http://www.research.ibm.com/trl/projects/security/ssp.

[11] **Jean loup Gailly** and **Mark Adler**. gzip. Website. Accessed June 2007, URL http://www.gzip.org.

[12] **Dan Grossman**, **Michael Hicks**, **Trevor Jim** and **Greg Morrisett**. Cyclone: a type-safe dialect of C. In *C/C++ Users Journal* volume 23(2005)(1).

[13] **Reed Hastings** and **Bob Joyce**. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the Winter USENIX Conference*. Pure Software Inc., USENIX, San Francisco, California, USA, 1992 pp. 125–138.

[14] **Dimitri van Heesch**. Doxygen. Website. Accessed June 2007, URL `http://www.stack.nl/~dimitri/doxygen`.

[15] **Charles Antony Richard Hoare**. The emperor's old clothes. In *Commun. ACM* volume 24(1981)(2):pp. 75–83. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/358549.358561.

[16] **Free Software Foundation Inc.** GNU compiler collection. Website, 1987–2007. Accessed January 2007, URL `http://gcc.gnu.org`.

[17] **Free Software Foundation Inc.** Bug 19319. Bug report, January 2005. Accessed January 2007, URL `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=19319`.

[18] **Free Software Foundation Inc.** GCC subversion repository guide. Website, 2007. Accessed June 2007, URL `http://gcc.gnu.org/svn.html`.

[19] **American National Standards Institute**. *Rationale for the ANSI C Programming Language*. Silicon Press, Summit, NJ, USA, 1990. ISBN 096153365X.

[20] **Richard W. M. Jones** and **Paul H. J. Kelly**. Bounds checking for C. Website, July 1995. Accessed January 2007, URL `http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html`.

[21] **Richard W. M. Jones** and **Paul H. J. Kelly**. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging* (editors **M. Kamkar** and **D. Byers**), volume 2 (1997), No. 009 of *Linköping Electronic Articles in Computer and Information Science*. Linköping University Electronic Press, Linköping, Sweden. ISSN 1401-9841, May 1997 pp. 13–26. URL `http://www.ep.liu.se/ea/cis/1997/009/02/`.

[22] **Greg McGary**. Bounded pointers. Website. Original URL `http://gcc.gnu.org/projects/bp/main.html` no longer holds this project. Accessed via mirror in January 2007, URL `http://gnu.teleglobe.net/software/gcc/projects/bp/main.html`.

[23] **Greg McGary**. Technical specification for bounded pointers in GCC, April 1998. In the GCC mailing list archives, URL `http://gcc.gnu.org/ml/gcc/1998-05/msg00073.html`.

[24] **Brian Paul**. The Mesa 3D graphics library. Website. Accessed June 2007, URL `http://www.mesa3d.org`.

[25] **IBM Rational**. Purify. Website. Accessed January 2007, URL `http://www-306.ibm.com/software/awdtools/purify`.

[26] **René Rebe**. Test flight: GCC 4.1 features and benchmarks. In *Linux Magazine* (2006)(68):p. 78. URL `http://www.linuxpromagazine.com/issue/68/GCC_4.1_Features_Benchmarks.pdf`.

[27] **Jonathan Rentzsch**. Data alignment: Straighten up and fly right. Website. URL `http://www-128.ibm.com/developerworks/library/pa-dalign/#N10232`.

[28] **Olatunji Ruwase** and **Monica S. Lam**. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*. February 2004 pp. 159–169. URL `http://suif.stanford.edu/papers/tunji04.pdf`.

[29] **Julian Seward**. Valgrind memcheck, 2000–2006. URL `http://valgrind.org`.

[30] **Andrew Suffield**. *Bounds Checking for C and C++*. BEng dissertation, Imperial College London, 2003. URL `http://www.doc.ic.ac.uk/teaching/projects/Distinguished03/AndrewSuffield.pdf`.

[31] **Frederic Trouche**. Compbenchmarks. Website. Accessed January 2007, URL `http://compbench.sourceforge.net`.

[32] **John Wilander** and **Mariam Kamkar**. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*. Karlstad, Sweden, November 2002 pp. 68–84. URL `http://www.ida.liu.se/~johwi/research_publications/paper_nordsec2002_wilander_kamkar.pdf`.

[33] **John Wilander** and **Mariam Kamkar**. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*. San Diego, California, USA, February 2003 pp. 149–162. URL `http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_wilander_kamkar.pdf`.

[34] **Michael Zhivich**. *Detecting Buffer Overflows Using Testcase Synthesis and Code Instrumentation*. Master's thesis, Massachusetts Institute of Technology, 2005. URL `http://hdl.handle.net/1721.1/32112`.

[35] **Michael Zhivich**, **Tim Leek** and **Richard Lippmann**. Dynamic buffer overflow detection. In *Workshop on the Evaluation of Software Defect Detection Tools*. Chicago, Illinois, USA, June 2005 URL `http://www.cs.umd.edu/~pugh/BugWorkshop05/papers/61-zhivich.pdf`.

A Logbook

Week of 22nd January

This was mainly spent familiarising myself with the GCC source. I tried to add an optimisation pass to the middle-end of GCC but I have not been able to compiler it successfully. GCC takes a very long time to compile as I have been performing a full rebuild each time as I am not sure where my build problems lie.

Week of 29th January

I have successfully added the new pass to the middle-end. The problems I faced last week were due to my unfamiliarity with the GNU autotools. I was basing my changes on the existing code for Mudflap but was not changing things in all the right places. Also I was not aware that GCC could no be executed from its build directory. It must be installed first.

The optimisation is enabled with the `-fbounds-checking` flag.

Week of 5th February

The simple pass from last week (which just printed a greeting message) has been modified to call two functions from and external library when a local variable is declared. These register the call by printing the properties of this variable.

This involved manipulating the intermediate representation trees to include function calls. This manipulation is done as an optimisation.

Adding the external library to GCC was not immediately successful as I did not run `automake` on the `Makefile.am` in the `gcc/libbounds` directory. This prevented it from being built or installed so when bounds checked programs were compiled they could not be linked. When run they could also not find the `.so` file.

Week of 12th February

The library calls added last week, which occur when a local variable is declared, have been expanded to record the details of the variable as an object in a splay-tree rather than simply printing them. To aid testing the newly created object is retrieved from the splay-tree before its details are printed.

A second, later, pass has been added which adds a third library call which checks the validity of an array access against the registered properties of the array. If invalid a violation message is displayed.

This week I finally succeeded in running the Mudflap and bounds-checker testsuites. When adding the bounds-checker library I had edited the top-level `autoconf` file, `configure.in`, and rebuilt `configure` from this using `autoconf` v2.60. It appears, however, that the top-level configure must be rebuilt using `autoconf` v2.13 for the testcases to work.

Week of 5th March

The array bounds-checking added last week worked in the Mudflap style: it checks that the region being accessed lies in a registered block of memory. In order to begin to adopt the Jones & Kelly method, I strengthened the condition of valid access such that not only must the region of memory be validly registered, it must also be within the array who's name is being used to access it. E.g. `array[3]` must lie within the registered bounds of `array` and no other object.

Similarly I added checking for accessing arrays via pointers. Initially this was just done the Mudflap way: the start and end of the access were checked to ensure they were located inside a region of allocated memory but nothing more. This gives very limited checking for pointers. For example an access of the form:

```
int a[10];  int *p = a;  p[3] = 0;
```

only checks that the region of memory between `&p[3]` up to `&p[4]` is valid. Similarly for

```
int a[10];  int *p = a;  p += 3;  *p = 0;
```

This is because the GCC GIMPLE middle-end `p[3] = 0` into two statements: `p = p + 3` and `*p = 0`. Mudflap does not intercept pointer arithmetic and therefore cannot check that the `p + 3` operation is somehow invalid. This is not the case for plain array accesses, `a[3]`, which are not split into separate instructions and, therefore, the entire region from `&a[0]` to `&a[4]` can be checked for validity.

For this reason, the Jones and Kelly method ensures that a pointer does not stray out of its 'referent' object by checking pointer arithmetic. Most of this weeks has been spent trying to implement this effectively in the GCC middle-end.

it has been very tricky working with the tree-manipulation functions in order to insert the new checks. The functions provided for this have little documentation and often have unexpected side-effects. For example, the two separate functions for adding access checks and arithmetic checks had a tendency to 'eat' more of the tree than they were supposed to causing statements to go missing-in-the-wash. Another problem was working out how to add a block of statements to the beginning of a basic-block.

There are still a number of issues which prevent compiling real-world programs. A large number of them relate to the fact that out-of-bounds pointers are considered erroneous and we do not wait till we try to dereference one before flagging and error. This has a number of effects such as preventing the correct operation of `for`-loops and causing bounds-checking on, as-yet, uninitialised pointers to fail as it is unable to locate a referent object.

Another problem is initialising a pointer in the following way: `p = &a[6]`. This is not yet supported but is commonly used.

One problem encountered at the beginning of the weeks was that the splay-tree can only look-up based on the low-address of the object. When trying to find the object that a particular pointer lies within, a range query is required. This meant writing a lookup function to search the tree which may not be efficient.

We have effectively **achieved Milestone 1** as the current problems are caused by not having an OOB pointer representation.

Week of 12th March

I added out-of-bounds objects (OOBs) this week. They are not implemented completely by any means as they vastly increase the complexity of what the bounds-checker has to do. Whereas, before, we only needed to intercept dereferencing (through pointers and array accesses) and pointer arithmetic, we must now intercept any operation that makes sense to perform on an out-of-bounds pointer. For example, if a loop has a condition:

```
while( p < &[10] )
```

the 2nd operand, `&a[10]`, is likely to be out-of-bounds by one. In order for this to work correctly the conditional must be intercepted so that for any of the pointer operands that are actually OOBs, the correct value is pulled out of the OOB and used in the comparison rather than the value of the pointer which is just the address of the OOB.

This work has not been done yet as other major problems were discovered which needed solving first. So far, only correct OOB handling for pointer arithmetic and dereferencing has been achieved.

While working on the OOB handling it became apparent that the current pointer checking logic had a fundamental flaw. During early development I had mainly been concerned with statements of the form:

```
p = p + i;
```

I had been taking `p` as the referent object pointer and using this to check that `p + i` lay within the same R.O. The problem lay in the fact that I had been using the **LEFT**-hand `p` for this. Once statements of the form:

```
p = q + i;
```

were appearing I assumed that checking that `q + i` had the same R.O. as `p` would still be OK. However, with a little thought it is clear that this is nonsense; the old value of `p` could be completely unrelated; perhaps we are just reusing the pointer.

This has not been fixed yet and is the first task for next week.

Week of 19th March

In trying to solve the problem discovered last week, we came up against a real problem. For the statement:

```
p = q + i;
```

we need to check that `q + i` remains in the same referent object a `q`. In the GCC middle-end `q + i` is represented by a `PLUS_EXPR` tree node. However **BOTH** the operands to this node are represented as pointers—GCC has coerced `i` into pointer form in an earlier statement. This makes it very hard (impossible) to tell which operand was actually the pointer and which was the simply the value being added to it and therefore we cannot work out which operand to use as the R.O. pointer.

GCC 4 has a representation called SSA that guarantees that no variable is ever assigned to more than once and provides handy ways to discover the use-definition relationships. This would enable us to look up the definition history of each operand and whichever was derived at some earlier point from a non-pointer would have to be the non-pointer operand.

While this sounded promising, most of this week was spent trying to turn SSA on. It turned out that in order to receive a tree in SSA form we had to move our pass to within the SSA optimisations and not merely set the flag noting that we require SSA. This had the undesirable side-effect of only allowing bounds-checking when optimisations were turned on (`-O/-O2/-O3`). Furthermore, as the SSA tree is different from the standard GIMPLE representation, it would have required rewriting much of the existing code.

After a week of trying I decided that it was not worth the effort as I am short on time and this does nothing to further the proof-of-concept. It is simple a work-around for a current GCC deficiency. As a stop-gap I have created a function that uses some educated guessing (such as whether the operand is artificial) to decide which is the pointer. This works a lot of the time.

Possible future solutions for this include using the 'pointer_plus' development branch of GCC which has an alternative node representation or looking at the LHS of every `MODIFY_EXPR` in a function to see which one defines the operand.

Week of 14th May

This week I added interception of pointer comparisons. This was needed as it is possible (in the case of loops, quite likely) for a pointer that is out-of-bounds to be used in a comparison. This requires the real pointer value to be retrieved from the OOB and that value used in the comparison instead of the actual value of the pointer which is simply the address of the OOB. There are now only a couple of tests that fail for any reason other than the pointer-operand-guessing hack.

Most of the week was spent trying to work out what on earth the code was doing. After a two-month break for exams I had almost completely forgotten.

Week of 21st May

Testing bounds-checking of real-world programs began in earnest this week. Before this could be done I had to add support for structs and built a suite of testcases to test this functionality. In doing so I found that if a struct is smaller than an integral multiple of a word size (i.e. if it leaves some free space at the end of a word) then any bounds violations in that remaining space are not detected. I will look into this next week.

The real-world testing began with trying to use the SPEC2000 benchmark but as this was not able to run successfully I pulled the first test program out of it, namely `gzip-1.2.4`, and began testing with that.

After fixing a bug where all function parameters were being assigned blank OOBs, it compiled and ran without problems. The files it produced could be uncompressed with another tool and their contents passed a binary diff test. The process is, however, extremely slow. I calculated that it was roughly 1000 times slower (100,000%) with bounds-checking enabled. Optimisation did not seem to affect this result. After looking through a trace I found that almost all the activity is checking arithmetic or dereference. Therefore the slowdown can be almost completely attributed to removing the R.O. caching that Mudflap used.

Other improvements this week include intercepting pointer casts allowing pointer difference to work.

Week of 28th May

The testing has continued this week with the focus on C++. Programs making use of the Standard Template Library required several changes in order for bounds-checking to work.

Firstly, the hack that decided which operand in a pointer addition was the actual pointer failed far too often with STL code to be useful. I updated this so that it searches back through the function looking for statements that the operands are derived from (their dominators). If one of the operands is found to be derived from a non-pointer value, the other operand must be a real pointer. This is still a hack but it appears to work quite well (some problems in `-O3`).

Secondly, the storage allocated for holding return values was not beeing registered. This was a problem in `struct/class` return-by-value cases as the method returning the `class` would often try to verify that where it had been told to put the return value was valid but as it had not been registered this would cause foreign pointers to be reported in the trace. This is only in the where the return value's address is taken. For example, code of the form:

```
Iterator begin() {
    return Iterator(args);
}
```

is transformed into the following in the intermediate representation:

```
Iterator begin() {
    Iterator(&<retval>, args);
    return <retval>;
}
```

When the `Iterator()` constructor executes, the '`this`' pointer that it uses to construct itself is, in fact, `<retval>`. As it uses this pointer, it is checked as normal, however as `<retval>` has not been registered, this check fails to find an R.O.

Another problem that was fixed this week was a flaw in the OOB logic. Updating and OOB after arithmetic is not as easy as it first seemed: the value of the pointer that is passed in is the value of the OOB's address with the arithmetic applied to it. To find the real value to update the OOB with, the OOB's address must be subtracted from the pointer to find the value of the arithmetic that was applied. This must then be used to update the stored pointer value in the OOB.

Testing with real world software has not been terribly successful this week. Firefox, Apache and Firebird DBMS all fail to build successfully, mostly due to problems to do with linking. The lack of support for threads with `libbounds` is unlikely to be helping the situation.