Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

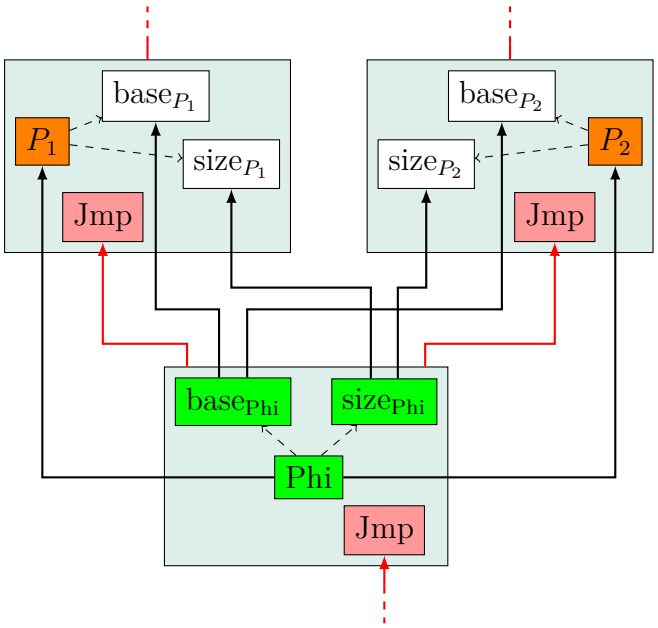
Lehrstuhl Prof. Dr.-Ing. Snelting

Protection of Heap-allocated Memory using Low Fat Pointers in libFirm

Bachelorarbeit von

Achim Kriso

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Andreas Fried
Abgabedatum:	24. März 2020

Zusammenfassung

Memory safety during program execution is a notoriously difficult problem. It is responsible for numerous critical security vulnerabilities in commonly used software. Heap buffer overflows are a common subset of memory safety errors and are further explored. We attempt to mitigate this problem by using Low Fat Pointers to detect pointers leaving their allocation and catch potential errors before they occur. This thesis implements this technique in libFirm as an optional compiler phase. We were able to detect known out of bounds pointers in the SPEC2006 benchmark and find undefined behaviour within in libFirm. The performance and memory usage of our implementation is comparable with the existing implementation for LLVM.

Es ist bekanntlich ein schwieriges Problem, den Programmspeicher sicher zu verwenden. Speicherfehler sind die Ursache für viele kritische Sicherheitslücken in alltäglich verwendeter Software. Überläufe von heap-allokierten Buffern sind eine häufig auftretende Art an Speicherfehlern und werden in dieser Arbeit behandelt. Wir versuchen dieses Problem zu minimieren, indem wir Low Fat Pointer verwenden um Zeiger zu entdecken, die ihre Allokation verlassen haben und somit potentielle Fehler zu aufzuspüren, bevor sie auftreten. Dafür implementieren wir diese Technik in libFirm als eine optionale Kompilierphase. Wir waren damit in der Lage, bekannte Speicherfehler in der SPEC2006 benchmark zu erkennen und haben undefiniertes Verhalten in libFirm gefunden. Die Performanz und der Speicherverbrauch unsere implementation ist vergleichbar mit der existierenden Implementation für LLVM.

Contents

1 Introduction

Memory safety is a very common cause for security vulnerabilities[1]. An important subset of memory safety are heap buffer overflows, which are the focus of this thesis. In most commonly used operating systems the memory of a process has both a *heap* and a *stack*. The stack keeps track of local variables and function calls. The heap serves as memory which can be allocated and freed by the programmer at will and therefore be used for arbitrary amounts of time. Usage of heap memory carries its own risk in programming languages like C, which rely on the programmer to allocate and free from the heap. It is an easy mistake to accidentally read, or write outside the allocated area. This can either crash the program, if the memory was not mapped by the operating system, or silently corrupt data which is used by the program in other places. Errors like this can be exceedingly difficult to find.

To solve this problem we implement an additional compiler phase in the libFirm compiler framework. This phase inserts checks into the libFirm IR to verify memory accesses on the heap.

1.1 Outline

We explain the basics required to understand the content of this thesis, by starting with a summary of a compiler's architecture and related concepts. We continue with introducing libFirm and explaining its relevant concepts. From section 2.4 we present an overview of memory error detection technique, as well as several implementation of similar existing tools and how they use these techniques. In section 2.5 we describe the key concepts presented by [2] regarding Low Fat Pointer. Starting from chapter 3 we discuss how we integrated Low Fat Pointers in libFirm to build a working heap memory error detector. We especially focus on how to manipulate libFirm main datastructure, the firm graph. In chapter 4 we benchmark our memory error detector and compare the results against the results of [2]. Finally, in chapter 5 we discuss the results and possible improvements to our current implementation.

2 Basics and Related Works

2.1 Compiler

A compiler's purpose is to translate programs written in a higher level language into a lower level language, usually assembly or machine code. Modern compilers consist of several parts. The front end analyses the input and transforms it into an intermediate representation. This includes parsing the input into an *abstract syntax tree* (AST), which is further analysed to check for the correct types and perform preliminary optimizations usually specific to the input language. The AST is transformed into an *intermediate representation* (IR) by the front end and passed to the middle end. The intermediate representation is a language and hardware agnostic representation of a program used by the middle end to apply general optimizations. Finally the resulting IR is passed to the back end where hardware specific optimizations are performed and the final assembly created. An IR allows both language front ends as well as hardware back ends to only target the IR. With n language implementations targeting m hardware architectures, only n front ends and m back ends are required. Implementing the same without using an IR would require $n \cdot m$ different compilers.

2.2 Static Single Assignment Form (SSA)

Intermediate representations which are in static single assignment form or SSA [3] have the property that every variable can only be assigned once. This constraint allows for easier reasoning and subsequent optimization of the program code. For example, it is trivial to find the definition of a variable in SSA while it is much harder to do the same in non-SSA programs since the definition of a variable can change. To satisfy this constraint, the basic block of a variable assignment needs to dominate [4] all blocks which use that variable. This is a problem for non-linear control flow like if-conditions or loops, which change a variable in their body.

As an example, a control flow graph is shown in Figure 2.1. Depending on the value of `x1` it executes the left or right path which both change the variable. After the conditional it would be impossible to get the resulting value of both paths in a new variable. To solve this issue SSA uses a *phi* function which takes two or more variables and returns the correct variable depending which path the control flow has taken.

Figure 2.2a is a simple C-like program which contains a loop and prints all numbers from 0 to 8. The left version is not in SSA-form because it reassigns the variable `i` in the body of the loop. After transforming it into Figure 2.2b, phi is used to decide if

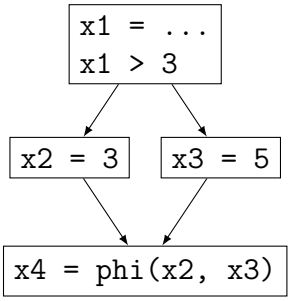


Figure 2.1: Controlflow graph in SSA which requires a phi.

```
void main() {
  int i = 0;
  while (true) {
    if (i >= 9) {
      break;
    }
    print(i);
    i = i + 1;
  }
}
```

(a) Program not in SSA-form.

```
void main() {
  int i1 = 0;
  while (true) {
    i2 = phi(i1, i3)
    if (i2 >= 9) {
      break;
    }
    print(i);
    i3 = i2 + 1;
  }
}
```

(b) Program in SSA-form.

Figure 2.2: Simple program before and after SSA-transformation.

`i` is 0 if the loop was just entered, or the value of the last iteration. The phi function allows every program to be represented in SSA-form.

2.3 libFirm

libFirm[5] is a compiler framework developed since 1996 at KIT. It is a C library and can be used in compilers as the middle end and back end. Most notably libFirm uses a directed graph as intermediate representation called firm.

2.3.1 Firm

Firm is the only intermediate representation used in the entire compilation process of libFirm and gets transformed in each compilation phase. It is a graph and based on SSA which is inherent to its structure, where every variable is represented as a node.

Every function gets its own graph. The nodes of the graph are either values, basic blocks or describe the control flow. Values are usually the result of an instruction or constant values. There are, however, more unusal values:

- **Memory** is treated as a value. Firm uses it to define the order of all memory accesses. If a node requires memory, it returns it as well to define the order of memory accesses where necessary. Usually there is only one memory value which is passed along the path from the start to the end of a function.
- **Tuple** is a collection of values. Their individual elements can be separated using projection nodes. They, for example, appear in load nodes, which return the memory value and the loaded value.

The edges in firm describe the relation between the nodes. There are four types of edges [6]:

- **Data Dependencies** (black): These are the most common edges and describe what data a node depends on.
- **Reversed Control Flow Edges** (red): Reversed Control Flow Edges describe the order of basic blocks. They can only start at basic blocks and point to control flow nodes (like jump nodes).
- **Memory Dependencies** (blue): Since the memory value is passed from one node to another to define the order of memory accesses, memory dependency edges are used to describe this dependency. Usually every used memory value originates from the one given at the beginning of the function. There is an exception to this rule which is used in subsection 3.2.2 and explained in more detail there.
- **Keep** (violet): If code is not reachable via the above dependencies but reachable by the control-flow (like infinite loops) keep edges are used.

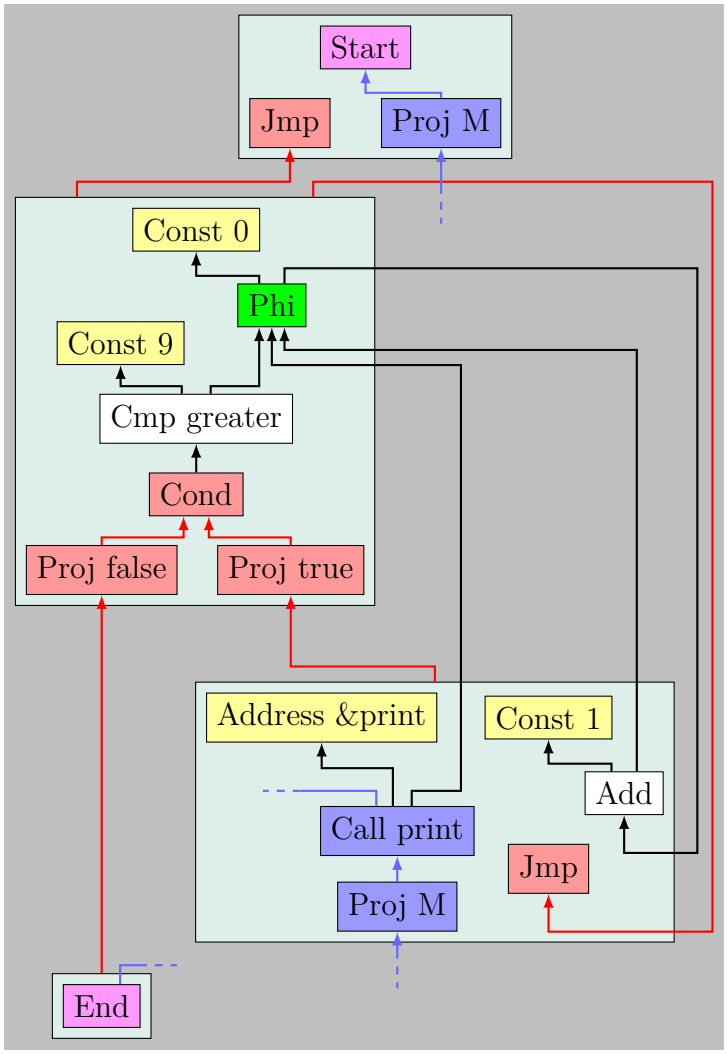


Figure 2.3: A moderately complex firm graph, demonstrating most important concepts required by this thesis.

Unlike the entire graph, the subgraph inside a basic block forms a directed acyclic graph. As a matter of fact, only phi-nodes are able to introduce cycles into the firm graph, in order to form loops.

Figure 2.3 is a simplified[1] firm graph of Figure 2.2. It starts at the pink start-node which marks the beginning of the function and returns a tuple containing the memory value and another tuple of all function arguments. Following the control flow jmp-node, we reach the second highest basic block which contains the if-condition inside the loop. The phi-node represents `i2` in Figure 2.2b and is the left constant zero or the result of the add-node in the basic block directly below this one. The result of the phi-node is compared against a constant nine with the cmp-node.

[1]There is another phi-node for the memory value, which is left out because is only increases the complexity and is not important for this thesis.

Depending on the result, a different control flow path is taken with the use of the cond-, proj false and proj true nodes. If the result is true, we reach the third basic block. This block calls the print function with the call-node. Since a function can access memory, every call-node requires a memory value and therefore has a memory dependency. Additionally, every call-node needs the address where the function is located in memory, which is usually a constant address. Here, it is the address-node `&print`. Finally, depending on the function signature, the call-node needs zero or more additional arguments. Because the `print` function only takes one argument there is one more data dependency from the call-node to the phi which represents the loop variable and is supposed to be printed. The current basic block additionally increases the loop variable with the add-node which takes the last value from the phi-node and adds a constant one value. Once this block is finished the control flow jumps back to the basic block above. This continues until the condition is false and the end-node is reached which represents the end of the function and takes the memory value.

2.4 Related Works

Detection of memory errors has been actively researched and there are multiple implementations of different approaches. They generally use a combination of three techniques. There is shadow memory to keep track of metadata, instrumentation to manipulate the programs behaviour, and a custom allocator to maintain the invariants expected by the error detection scheme.

2.4.1 Shadow Memory

Shadow memory is a dedicated area of memory which stores metadata about every piece of data of the running program. Every segment of memory in the address space gets associated with a segment of memory in the shadow space. Therefore tools that utilize shadow memory usually incur a high memory footprint. This mapping of an address to shadow memory is done either by calculation or by using a lookup table. The latter requires a pointer indirection which costs significant performance, so the former is generally preferred. The data stored in the shadow space can be used to check boundaries or similar.

As an example, we could use shadow memory to tell if a byte is allocated when a corresponding bit in shadow memory is set. We map 8 byte chunks of arbitrary memory to a byte of shadow memory where each bit corresponds to a byte in the chunk. Given a pointer p we can calculate the corresponding shadow byte using $b = \frac{p}{8} + o$ where o is some offset to ensure that the shadow memory doesn't start at `NULL`. Assuming the 8 byte chunks are aligned we can use $p \,\%\, 8$ to find the bit which corresponds to p.

2.4.2 Instrumentation

A common approach to check if a memory error occurred is to verify every memory access (pointer dereference). To achieve this behaviour, instrumentation code is inserted into the program before the memory access to do this check. If such a check fails during program execution, it can be reported and the program terminated. One differentiates between compile-time and binary instrumentation, where binary instrumentation is applied on a binary executable, while compile-time instrumentations is additional code inserted by the compiler into the compiled program.

2.4.3 Custom Memory Allocators

Customizing the memory allocator allows invariants about memory to be upheld. These invariants are used to verify memory accesses. For example such invariants can be memory allocations at certain positions in the address space or redzones around an allocation.

2.4.4 Implementations

AddressSanitizer [7] is a popular tool for detecting memory errors, included in the LLVM compiler framework. It leverages all three of the above mentioned techniques. It uses shadow memory which is between $\frac{1}{8}$ to $\frac{1}{128}$ the size of used memory. This shadow memory encodes whether a byte is currently allocated. The shadow memory is created and maintained by a custom allocator. Finally, compile-time instrumentation is used to verify memory accesses as described in subsection 2.4.2.

Memcheck[8] is a memory checker implemented on the Valgrind framework[9]. Valgrind analyzes a binary and constructs its own processor independent IR upon which several program analysis tools are built. Memcheck can detect several classes of memory errors, including out of bounds memory accesses, incorrect freeing of memory, using uninitialized values and memory leaks. Memcheck instruments the IR with the necessary checks and finally executed. It uses shadow memory to track which bit values are undefined. Using Memcheck a program runs 20-30 times slower than without.

Dr. Memory[10] is another tool find memory errors. It uses shadow memory to categorize every byte into unadressable, uninitialized and defined. This information can then be used to detect out of bounds accesses and reading of uninitalized memory. Using a system similar to a tracing GC, it can detect memory leaks based on reachability. It is built upon DynamoRIO[11], a code manipulation system.

2.5 Low Fat Pointers

Fat Pointers are pointers which carry metadata about their corresponding memory allocation and therefore take up more memory (hence the name) than normal native pointers to store their metadata.

```
struct {
    void *ptr;
    void *base;
    size_t size;
}
```

They can be represented using a struct with the original pointer `ptr` and two additional members for the `base` and `size` of its allocation. `base` points to the first byte of the allocation and `size` stores the amount of bytes allocated. Pointer arithmetic just changes `ptr`. Everytime such a Fat Pointer is dereferenced it is possible to check if `ptr` within the allocation bounds by evaluating if $\texttt{base} \leq \texttt{ptr} < \texttt{base} + \texttt{size}$ is true. If this check fails, we know that `ptr` has left its allocation.

Low Fat Pointers were first proposed in [12] and are like Fat Pointers in that they remember their bounds metadata, but have the same size as native pointers. They encode their metadata within the pointer where they utilize the first few most significant bits to store their metadata. Both Fat Pointers and Low Fat Pointers break *binary compatibility*. If a program is compiled with Fat Pointers but is linking a library which does not support them, both will likely malfunction. Since the data layout of a Fat Pointer and a native pointer differ, both sides need to use the same pointer type, otherwise the calling convention is broken when passing pointers across API boundaries. When using the original Low Fat Pointer encoding, the calling convention holds but their respective usage is not the same. For example it is generally not possible to just dereference a Low Fat Pointer because the metadata in the MSB changes the value of the pointer.

Low Fat Pointers as presented in [2] use a different encoding which leverages the large virtual address space in 64-bit systems to encode their metadata in their value, the address they point to. This value can be used to look up the pointer's corresponding allocation size inside a table. Additionally, this encoding requires all allocations to be aligned to a multiple of their size. This is required to allow for the calculation of the allocation base using only the pointer and its size. Low Fat Pointers using this encoding are ordinary pointers with all the properties of native pointers and therefore solve the binary compatibility problem. They can be used by software compiled without consideration for Low Fat Pointers.

2.5.1 Low Fat Pointer Memory Allocator

The entire virtual address space is divided into *regions* of equal size. The size is chosen as some 2^n. We follow the original paper in our choice of $n = 32$ (4GB). Each region is aligned to its size, meaning that the starting address of a region is a multiple of 2^n. A subset of all regions is chosen to serve as the heap. All other regions are not mapped by the allocator and are free to be used by other allocators, e.g. the standard libc allocator. To implement Low Fat Pointers there are two requirements.

1. Every region only allocates objects with a certain size. If no region exists for an allocation size, the allocation is padded until it does. Each region serves as

a *bucket* or *sub-heap* of an allocation size.

2. All objects must be aligned to their size. (The position has to be a multiple of their size.)

Again we follow [2] in choosing the allocation sizes of

$$sizes = [16B, 32B, 48B, 64B, \ldots, 8KB, 16KB, 32KB, 64KB, \ldots, 1GB]$$

using a total of 530 regions. Finally, each region has an *index*, starting at index #0 to #65535[2] in the case of a 4GB region size. We can calculate the index of a region which contains a given pointer by shifting the pointer by n bits to the right.

2.5.2 Reconstructing Metadata

To check whether a pointer p is within the bounds of its allocation, we need two values: the allocation's $base_p$ and $size_p$. p is within bounds if $base_p \leq p < base_p + size_p$.

First, we determine the region index of p by shifting it 32-times to the right. $size_p$ is reconstructed with the help of a lookup table, indexed by the region index. The lookup table stores the allocation size of every region.

Finally, $base_p$ can be calculated by using the following calculation:

$$base_p = (p/size_p) * size_p$$

or the equivalent but more efficient

$$base_p = p - (p \,\%\, size_p)$$

where /, $*$ and % represent the 64-bit unsigned integer division, multiplication and modulo operations[3].

This reconstruction procedure assumes that p is within bounds. If p has left its allocation, the calculated metadata will be the metadata of the allocation which contains the pointer. It is up to the Low Fat Error detector to ensure that it only calculates the metadata of a pointer which upholds this precondition. We only need to consider this problem when performing pointer arithmetic and it is further elaborated in subsection 2.5.5.

Calculating the metadata is a fairly expensive operation since it requires a division operation which requires up to 123 clock cycles [14]. It should be only used when absolutely necessary.

[2]Modern 64-bit systems, only have a 48-bit address space [13], which lowers the amount of regions significantly.

[3]libFirm will optimize the first calculation into the second.

2.5.3 Binary Compatibility

Maintaining binary compatibility is a goal of Low Fat pointers. Therefore it is vital to ensure the program still works if it is passed non-Low-Fat Pointers by external functions. This is achieved by giving every pointer that is not a Low Fat Pointer the base of `NULL` and size of `UINT64_MAX`. Any bound check will pass with this metadata. Naturally all safety guarantees get erased for such pointers.

For the implementation this implies that the lookup table has the value `UINT64_MAX` for every unused region. In this case the base is:

$$base_p = p - (p \,\%\, \texttt{UINT64\_MAX}) = p - p = \texttt{NULL}$$

2.5.4 Metadata Propagation

When discussing how to determine the metadata, we need to differentiate between two types. There is *implicit metadata* and *concrete metadata*. Implicit metadata is the metadata that is stored implicitely within a Low Fat Pointer but we don't have direct access to. Concrete metadata is metadata we have direct access to, like if it is stored inside a variable. To check if a pointer is within correct bounds, a pointer needs to have its concrete metadata associated with it, which it can use during a bound check. This metadata is stored in two variables (or nodes in the case of a firm graph) and needs to be determined for every pointer used in a bounds check. In most cases it would be possible to fallback to use the fairly expensive calculation of the metadata using the formula in subsection 2.5.2. Pointer arithmetic presents the exception to the rule because addition and subtraction are the only operations which can change the value of the pointer, possibly moving it into the range of a different allocation.

To avoid the calculation there are other cases where we can infer the metadata based on its creation:

If p is the result of a phi, we can use the metadata of its inputs to determine the metadata of p. p needs to inherit the metadata of the input which was returned by phi. We create a new phi for both the base and size metadata, which take the corresponding metadata of phis inputs. These new phis will result in the correct concrete metadata for p.

Constant pointers like the null pointer can be hardcoded with the `UINT64_MAX` as size and `NULL` as base.

If p is the result of an allocation, the function signature of the allocation function usually can tell us the size metadata of the newly allocated pointer. The returned pointer is pointing at the beginning of the allocation, therefore representing the base.

In all other cases the metadata is calculated as described in subsection 2.5.2.

2.5.5 Bounds Checking

Naively, one might only check if a pointer is valid if it is dereferenced. This ensures that every memory access is valid under the assumption that the concrete metadata

values currently associated with this pointer are valid. As described in subsection 2.5.2, calculating a pointer's metadata after performing pointer arithmetic can lead to a wrong result. If the metadata of the original pointer is available this is not an issue since we can just inherit it. This is impossible if the result of pointer arithmetic is passed to a function as an argument. The function does not have access to the metadata of the original pointer and can only calculate it. To ensure that it always calculates the correct metadata, the pointer must be bound checked before it is passed to the function. Everytime a pointer *escapes* the function context like this, it needs to be checked if it is still within its bounds. Other cases of such escapes include if a pointer is returned by a function or if the pointer is stored in memory.

This implies the invariant that pointers entering a new function context have to be within bounds. Therefore, memory accesses or escapes in this new function do not need to be bound checked if the pointer does not change. Since pointers can only change because of pointer arithmetic we only need to insert bound checks if such a change occured.

There is a minor optimization regarding bound checking: Instead of calculating $base_p \leq p$ and $p < base_p + size_p$, we can use the overflow semantics of binary numbers to only check $p - base_p < size_p$. If p is less than $base_p$ then the subtraction underflows resulting in a larger unsigned value than $size_p$.

2.5.6 Tradeoffs

While Low Fat Pointers are an elegant technique to remember the meta-information of a pointer, this section discusses their drawbacks.

There is only a very limited set of sizes that can be allocated. Different sizes need to be padded and the pointers forget the exact size which they were supposed to be allocated as. This results in out-of-bounds accesses not being detected if they are within the additional padding. This behaviour, however, cannot cause memory corruption since only the padding is read or written.

However, if Low Fat Pointers are only used for testing, they might not detect out of bounds accesses which can corrupt memory if the program is shipped without Low Fat Pointer support. Another possible solution to this problem is to first profile all allocation sizes and then configure the allocation sizes used by the custom allocator correspondingly.

Regions can overflow if too many allocations of the same size are made. In this case the allocator defers to the standard libc allocator. Pointers which don't originate from the low fat memory allocator will pass all bound checks because their metadata has the base of `NULL` and size of `UINT64_MAX`.

With our configuration a program would need to allocate more than 4GB of a single size of allocations. Such programs are deemed rare enough to not explore a more dynamic approach. One can always change the region size if it is necessary.

3 Design and Implementation

The implementation consists of two parts: 1.) The memory allocator for assuring the positioning invariants of allocations. 2.) The additional compiler phase which inserts compile-time instrumentation to calculate pointer metadata and check whether they are within the correct bounds.

3.1 Memory Allocator

To implement the memory allocator as described in subsection 2.5.1, we use freelists. Since each region serves as a heap for a certain allocation size, each region has a freelist to keep track of available memory. An advantage of a fixed set of possible allocation sizes is that there is no need to iterate over the freelist since all elements are the same size and we can just take the first element and do not have to consider memory fragmentation.

As each region is 4GB in size, we do not initialize the entire freelist in the beginning, to avoid physically allocating this much memory in the beginning. Instead, we split each region into two parts. The first part of a region is the freelist which contains the currently available segments of memory which can be allocated. The second part directly follows after the freelist and is unallocated memory which takes up the remainder of the region and is therefore called *remainder*. If the freelist is exhausted it grows into the remainder for additional memory to allocate.

The allocator is initialized on the first invocation of the allocator. It reserves and maps the memory for all used regions using `mmap` and initializes all freelists. The mapped memory is read and write. We follow [2] by using the `NO_RESERVE` flag to not reserve physical memory.

To implement an allocator which can be a drop-in replacement for the standard libc allocator we need to implement ten functions [15].

To support multiple threads using the allocator we give a unique mutex to each region. Multiple threads can allocate in parallel as long as their allocation sizes are different.

The use of this allocator is only sensible when the entire low fat sanitizing scheme is used including the required instrumentation. Therefore, we statically, link the allocator and specifically call the allocation functions of our custom allocator which is further explained in subsection 3.2.1. External libraries which are not compiled with the low fat sanitizer continue to use their standard allocator.

3.2 Instrumentation

Within libFirm we insert additional nodes and edges to the firm graph with the help of an additional compiler phase. The phase is called on the firm graph of every function. It analyzes the graph and subsequently transforms it. The phase consists of four phases itself:

1. All allocation function calls from section 3.1 are changed to reference our custom allocator.

2. The lookup table for the different allocation sizes is initialized if it does not already exist.

3. The function graph is traversed and all positions which require a bound check are remembered.

4. All bound checks are inserted and the metadata calculation inserted where necessary.

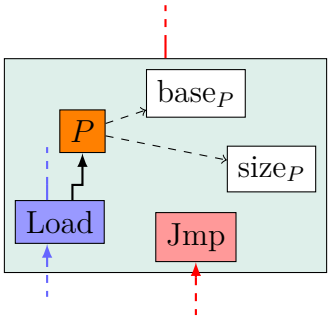
3.2.1 Calling Memory Allocator

Since the memory allocator is linked statically the allocation functions have been renamed to not cause linking issues when the program uses external libraries which use the normal allocator. Therefore, we need to change all allocation function calls in the graph to call our allocation functions. We traverse the graph and check if the current node is a call-node with one of the ten allocation functions. In this case we change the linking name and continue traversing the graph.

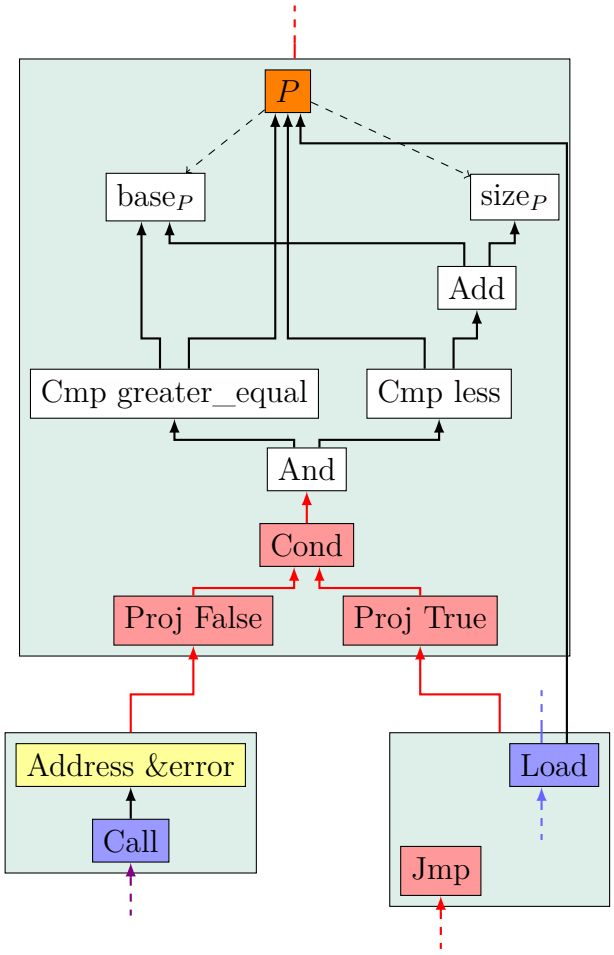
3.2.2 Lookup Table

The lookup table in the final binary is a standard constant array in .RODATA. It is present in the allocator as well as during the metadata calculation instrumentation. We tell the linker to treat those as the same array to avoid duplication in the final binary.

Using the lookup table requires a load-node in the firm graph. Such nodes need the memory value. While it would be possible to use the normal memory value it is quite cumbersome because we need to choose a place where we can insert the load node along the path of memory nodes. Instead we can use an independent memory value called no_mem which allows us to insert memory accesses anywhere which use this no_mem instead of the normal memory value. It is up to us to ensure that such memory accesses do not conflict with others. Because the lookup table is readonly there is no dependency between the lookup table and any other memory operation and therefore we can use no_mem for these lookups.



(a) Graph before inserted bound check.



(b) Graph after inserting bound check.

Figure 3.1: The left graph shows the dereferencing of a pointer P with known metadata. It does not use the faster check explained in subsection 2.5.5. After inserting a bound check between P and the load-node we get the right graph. The cmp greater_equal node determines if the base of the allocation is greater or equal than P. The cmp less node determines if the end of the allocation is less than P. If both are true, P is within bounds and we can load its value. Otherwise, some error function is called.

3.2.3 Inserting Bound Checks

This section elaborates on the procedure for inserting bound checks into the firm graph. We begin by searching for all positions in the firm graph where a bound check is required. Obviously memory accesses need to be checked to ensure that the pointer is within the correct bounds. Moreover, as explained in subsection 2.5.5, every time a pointer leaves a function and therefore the graph that is currently analyzed, the pointer might need to be checked as well. Traversing the firm graph does a recursive DFS starting from the end-node. It is important to not mutate the graph while traversing it, because it can invalidate local references held by the DFS.

Every pointer node in the graph needs to keep track of its metadata where the metadata consists of the base value and size value. Both those values are nodes in the graph to allow calculations using them during execution. To keep track of the metadata of a node we use a hashmap which maps a pointer node to its corresponding metadata nodes. If a pointer node does not appear in the hashmap, its metadata has not been determined yet.

Figure 3.1 shows the firm graph before and after inserting a bound check before a load-node. Every time a bound check is inserted we find the associated metadata of the pointer in the hashmap and instrument the graph to check if the pointer is larger or equal than its base and smaller than its base plus its size. In this case, the control flow continues as it would normally, otherwise we call an error function provided by the runtime. We split the current basic block and move the pointer node, its metadata nodes and every node they depend on in the new basic block above. Since the metadata nodes are independent from the pointer node we explicitely need to move them, because their relation is not described in the firm graph.

If a check fails we print an error message detailing which pointer was out of bounds, its metadata and type of error, i.e what kind of memory access or type of function context escape. Additionally we print the location of the error in the original code, by embedding its corresponding filename and line number inside the binary and passing it to the error function. Whether the program exits if it encounters an error is a compile time flag. If the program doesn't stop subsequent errors can be incorrect because the metadata calculation is no longer correct once a pointer is out of bounds.

Checking if a pointer is within bounds is a comparatively expensive operation because it contains a conditional jump instruction. Therefore, we should avoid inserting bound checks whenever possible. We can omit a bound check if we know that the pointer does not originate from our allocator. This applies if the pointer is a constant like the `NULL` pointer or an address from the binary. If a pointer has not been changed by pointer arithmetic we do not require a bound check as well.

3.2.4 Finding Metadata

The metadata of a pointer depends how the pointer was created. Due to firm being in SSA-form each pointer is immutable and has exactly one definition which is used to determine its corresponding metadata. Once this metadata is found, other pointers

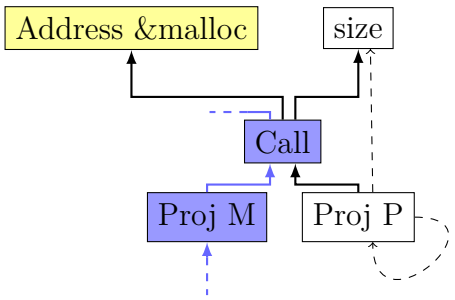


Figure 3.2: The call-node returns both the memory and a pointer as a tuple. Both values are extracted using projection nodes[1]. The pointer projection node has itself as the base because `malloc` (and the other allocation functions) return the start of the allocated area. The size node is an argument of `malloc` and can be used as the size metadata for the returned pointer.

can depend on it without the need to consider that it might change.

Every possible pointer node has to be able to find its metadata based only on itself or its inputs. In total, there are six cases to consider:

If the pointer is the result of an allocation function, we assume that those functions are working correctly and use their function arguments and return value to infer the correct metadata. We consider all functions listed in [15]. `malloc`, `valloc` and `pvalloc` all return the base address of the allocation and are given the size as the first argument. Figure 3.2 shows how the firm graph looks for `malloc`. `realloc`, `aligned_alloc` and `memalign` behave very similar to the previous three functions except that their size is the second argument. `calloc`, while also returning the base address allocates a size of the multiplication of both arguments, to check for overflows. We create a mult-node which performs this multiplication and represents the size. If the the allocation call fails `NULL` is returned. In this case the inferred metadata is incorrect. For memory accesses this is not an issue, since a memory access at `NULL` is undefined behaviour and out of scope of this thesis. However if the pointer escapes its current function context it will report an error aswell. This behaviour is uncommon but if required it is possible to disable inferring the metadata of allocation functions and fall back to calculating the metadata of the returned pointer instead.

When the pointer is a result of pointer arithmetic the pointer inherits the metadata of the original pointer. Figure 3.3 illustrates this in the firm graph. The orange P node is some pointer with the metadata being the base and size nodes. An add-node is using this P to calculate a new pointer but its metadata remain the same nodes.

Member-nodes are used to get pointers of members of some struct and effectively only perform pointer arithmetic on the original pointer. They have to account for

[1]There is actually another projection node between the call node and Proj P which was left out for simplicity.

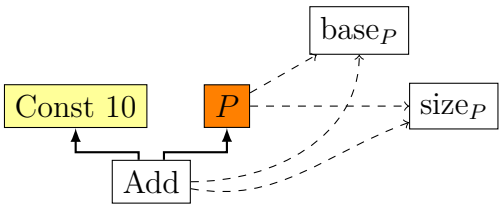


Figure 3.3: Metadata propagation of an add-node doing pointer arithmetic. P is some node with a pointer as an result. P has the nodes base and size as metadata. The add-node calculates a new pointer which inherits the base and size nodes from P.

memory layout considerations which is the reason why they are their own node type. Since they effectively just represent pointer arithmetic we can apply the same logic used for add-nodes regarding their metadata.

If a phi node has a pointer value all its inputs have to be pointers and therefore have associated metadata. To create the concrete metadata of a phi we insert two additional phi-nodes into the firm graph which represent the base and size and the phi-node. Both of these new phis get the original phis inputs as metadata correspondingly. Figure 3.4 shows a general example if phi has two inputs.

The metadata propagation of phis for loops is non-trivial. The metadata nodes of such a phi depends on the metadata of its input. One of these metadata nodes, however, depend on the metadata nodes of the phi, creating a cyclic dependency. In order to solve this problem, dummy metadata nodes are created for the phi before determining the metadata nodes of its inputs. Once the inputs metadata is found, we can create the correct metadata for the phi using the newly found input metadata and swap it with the dummy.

If a pointer is a constant value, either `NULL` or by type casting an integer, we provide zero safety guarantees for such a pointer since it is completely seperate to heap allocations. Such pointer get the base `NULL` and size `UINT64_MAX`.

If none of the above cases apply to a pointer node, the only option remaining is to reconstruct the pointers metadata with its position using the procedure outlined in subsection 2.5.2. Figure 3.5 shows the firm graph of the metadata calculation of some pointer P. P is converted to an unsigned 64bit value to perform the calculation. It is shifted to the right by 32 bits to get the region index. Because each element in the lookup value is 64 bit in size we need to multiply the index by 8 bytes to get the correct memory offset into the lookup table. To get the actual address we add the address of the lookup table, called sizes, to the previously calculated memory offset. The offset needs to be signed to perform pointer arithmetic in the firm graph. The loaded value is the size metadata node for P. With the size node the base is calculated by subtracting the remainder of P divided by the size from P. It is again necessary to convert the unsigned loaded value to a signed value to perform the subtraction.

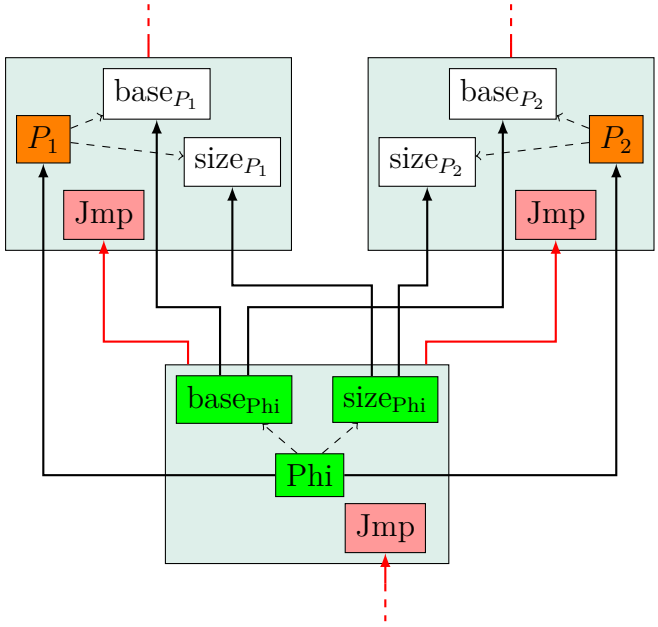


Figure 3.4: Metadata propagation of a phi-node with two pointer inputs. Both input pointers P_n have corresponding metadata nodes base$_n$ and size$_n$. The metadata of the phi are two new phi-nodes, one for the base and the other for the size. These new phi-nodes take their corresponding metadata nodes from both P_n.

3.3 Problem

It is not undefined behaviour in C for a pointer to be one byte past its allocation bounds as long as it is not dereferenced. Our current scheme, however, would detect it as an error if such a pointer leaves the function context. While we could ignore the error in this case, it can cause false positives as the program will calculate invalid metadata for this pointer if they are needed as the program continues. This problem is not discussed in [2]. We only found one option to circumvent the problem by increasing every allocation by one byte. Pointers which would ordinarily lie one byte past their allocation are now included and therefore no longer trigger an error. We pay with increased memory usage and less precise allocations which increase the likelyhood of false negatives.

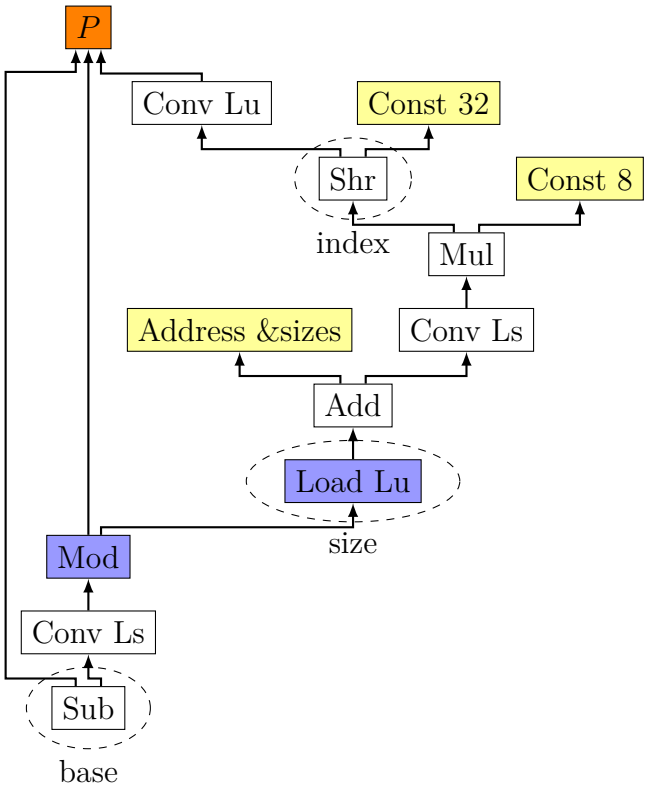


Figure 3.5: Firm graph of the metadata calculation of pointer P. P is shifted by 32 to calculate the region index. The region index is used to look up the size in the $sizes$ lookup table. Finally the base is calculated by subtracting the remainder of P divided by its size. Projections nodes are left out for simplicity.

4 Evaluation

We benchmark using the SPEC 2006 benchmark suite[16]. The suite was compiled using cparser with our modified libFirm version with the x86_64 back end. The benchmarks are executed on a system with an Intel Core i5-8500 clocked at 3.9Ghz using Ubuntu 18.04.4. Because no Fortran or C++ front end exist for libFirm, only C benchmarks are tested. We compare our results with the results of [2] where applicable. Because they sum all timings and memory usage respectively in their evaluation[1], we do the same in ours to be able to compare our results with theirs.

4.1 Performance

Figure 4.1 shows the results of the SPEC2006 benchmark suite. Here, the libFirm column is the uninstrumented version of the program compiled with libFirm. libFirm + LowFatPtr is the instrumented version using our implementation. The last column LLVM + LowFatPtr are the results presented by [2].

Except for two outliers `libquantum` and `h264ref`, we are fairly close to the ratio of [2]. Exluding those outliers we get a total ratio of 2.10x compared to their 2.06x. We are not certain of why `libquantum` and `h264ref` have such bad performance compared to them. However, considering that we are using different compiler frameworks, it could be possible that libFirm has more difficulties optimizing our instrumented code compared to LLVM.

4.2 Memory Usage

Like [2], we measure the maximum resident set size. SPEC2006 doesn't support measuring this metric, so we observe what programs are executed as part of the benchmark. We repeat their execution and use the GNU `time` utility to measure the memory usage. The different benchmark programs are executed multiple times with different arguments. It appears that [2] only took the execution with the largest memory footprint into account for their result, as those match the most with our baseline results. We follow their decision in order to compare our results with theirs. Figure 4.2 shows our results in the same layout as in Figure 4.1.

While the memory usage of [2] virtually stays the same, we have an increase of around 33%. There are several possible reasons for our higher memory usage: 1) We

[1] It appears that they follow AddressSanitizer's paper[7] which did the same in their evaluation of memory usage. Their performance evaluation, however, shows an average.

Benchmark	libFirm base	libFirm + LowFatPtr base	LowFatPtr ratio	LLVM + LowFatPtr ratio
perlbench	222	804	3.62x	2.23x
bzip2	339	711	2.10x	2.16x
gcc	181	654	3.61x	3.27x
mcf	175	220	1.26x	1.61x
gobmk	360	767	2.13x	1.58x
hmmer	558	1030	1.85x	2.62x
sjeng	390	773	1.98x	1.39x
libquantum	249	1973	7.92x	4.25x
h264ref	359	3075	8.58x	3.55x
milc	323	615	1.90x	1.92x
lbm	166	223	1.34x	1.81x
sphinx3	577	1136	1.97x	2.57x
total	3899	11981	3.07x	2.47x

Figure 4.1: Performance result of the SPEC2006 benchmark suite. (in s)

Benchmark	libFirm base	libFirm + LowFatPtr base	LowFatPtr ratio	LLVM + LowFatPtr ratio
perlbench	567	705	1.24x	0.99x
bzip2	868	874	1.01x	0.99x
gcc	831	1514	1.82x	0.99x
mcf	859	1718	2.00x	1.00x
gobmk	231	353	1.53x	1.00x
hmmer	27	50	1.85x	1.19x
sjeng	180	182	1.01x	1.00x
libquantum	100	166	1.66x	1.00x
h264ref	64	71	1.11x	1.00x
milc	673	714	1.06x	1.01x
lbm	420	421	1.00x	1.00x
sphinx3	400	509	1.27x	1.00x
total	5220	6959	1.33x	0.99x

Figure 4.2: Memory usage of the SPEC2006 benchmark suite. (in Mb)

Benchmark	min	max	median	allocations
`perlbench`	1b	8.48Mb	14b	353.15M
`bzip2`	16b	231Mb	94.88Kb	140
`gcc`	1b	253.93Mb	16b	28.43M
`gobmk`	1b	7.46Mb	16b	620.05K
`hmmer`	1b	3.7Mb	9b	2.47M
`libquantum`	8b	32Mb	8b	141
`h264ref`	4b	2.79Mb	16b	177.77K
`milc`	16b	199.5Mb	692b	6.51K
`sphinx3`	1b	2.51Mb	8b	13.90M

Figure 4.3: Unnecessary allocation padding of allocations performed by the SPEC2006 programs.

are missing an optimization where the actual size of the lookup table is reduced by mapping the entries of unused regions (which are all `UINT64_MAX`) onto the same page. 2) Our error function takes several values to be able to print a useful error message. This includes a string which contains the filename and a line number. While there are usually only comparatively few files, every inserted bound check has potentially a different line number which is 8 bytes each. 3) Finally, we always have to add one byte to the actual allocation size as described in section 3.3. This often causes the allocator to use the next larger allocation size as can be seen in section 4.3.

4.3 Precision

We analyze the precision of our memory error detectory by measuring the precision of the performed allocations. The closer the size of the performed allocation is to the requested size by the program, the more likely we detect an out of bounds pointer. Therefore, for every allocation we measure the difference between the actual allocation size and the requested allocation size. We refer to this extra unwanted memory as padding. Figure 4.3 shows the results for the SPEC2006 benchmark suite. The columns from left to right show the minimal amount padding added to each allocation, the maximum padding which is added to atleast one allocation, the median padding of all allocation and the number of allocations performed. We have left out `mcf`, `sjeng` and `lbm` because they each perform less than five, but very large allocations and distort the results. The range of the median is usually between 8 bytes to around half a kilobyte. `bzip2` performs relatively similar to `mcf`, `sjeng` and `lbm` by requesting few large allocations. The very large max allocation differences are explained by the content of the lookup table. As it can be seen in subsection 2.5.1 the table starts at 16 bytes and every subsequent size is incremented by 16 bytes until we reach 8Kb. At this point the size is doubled and therefore the sizes increase

exponentially. If allocations are performed in this size range the difference between actually allocated and requested sizes increase exponentially as well and are the cause for such high peak values. Another noteworthy observation is the common median value of 16 bytes. As a matter of fact, 7.6% of all allocations have a difference of 16 bytes. This is explained by the fact that it is common for programs to allocate multiple of 16 bytes. Because we need to add an additional byte to the allocation size (section 3.3), allocations which are a multiple of 16 bytes are moved into the next allocation size which is usually 16 bytes larger. The most common difference is 14 bytes, which is only first because `perlbench` performs the most allocations with significant margin and its allocations seem to favour a difference of 14 bytes.

4.4 Compilation Speed

Since the lookup table is fairly large (65536 entries each 8 byte), it takes libFirm a significant amount of time to do its integrity checks during compilation in debug mode, even for very small programs. Those checks are omitted when compiling an optimized cparser and libFirm. In this case, with our additional phase we are around 3x slower. Since compilation speed hasn't been a focus of this work, it can presumably be improved significantly.

4.5 Results

We were able to reproduce the out of bounds pointer in the `perlbench` and `gcc` benchmarks, which were described in [2].

We found one instance of undefined behaviour within libFirm using our memory error detector where an out-of-bounds pointer is written to memory. Because cparser and libFirm are tested with LLVMs AddressSanitizer as part of its continous integration, it was expected that no illegal memory accesses would be found. AddressSanitizer was not able to find this case of undefined behaviour because it is only checking for correct memory read and writes. While our detector, as part of its metadata propagation technique, has to check a pointer everytime it leaves the function context, here, writing a pointer into memory.

4.6 Other

The implementation of this memory error detector required only changes in the libFirm library and none in cparser except to run the instrumentation phase. Since libFirm is language agnostic it is possible for the memory error detector to used with any front end for libFirm.

5 Conclusion

In this thesis we have designed and implemented a Low Fat Pointer encoding to detect heap buffer overflows in libFirm. We explored how the required metadata can be inferred and propagated within the firm graph and what transformations need to be performed. Our evaluation shows that it is comparable to [2] implementation and is capable of detecting heap errors.

5.1 Future Work

Based on this work there are several things that can be improved.

Obviously we would like to see further optimizations to our implementation, like reducing the performance and memory overhead. It is possible to optimize the metadata calculation by using fixed-point arithmetic instead of integer division, as was described in [2]. There is possibly a better solution to the problem described in section 3.3 which could improve our precision as well. Alternatively one could store the exact size of every allocation to achieve perfect precision.

Currently our implementation is limited to the heap. It is possible to extend this technique to also check pointer pointing to the stack as presented in [17].

Additionally one could implement additional memory checks, like detecting double frees or if memory was accessed after it was freed. The former should be possible by only changing the allocator. Detecting if an allocation has been freed when accessing memory using Low Fat Pointer presents a harder challenge. One might use shadow memory to map every possible allocation to a bit.

Bibliography

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=%22buffer+overflow%22.

[2] G. Duck and R. Yap, "Heap bounds protection with low fat pointers," pp. 132–142, 03 2016.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, Oct. 1991.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[5] G. Lindenmaier, "libFIRM – a library for compiler optimization research implementing FIRM," Tech. Rep. 2002-5, Sept. 2002.

[6] "Libfirm edge types." https://pp.ipd.kit.edu/firm/Edge_Types.html.

[7] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 309–318, USENIX, 2012.

[8] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision.," pp. 17–30, 01 2005.

[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), p. 89–100, Association for Computing Machinery, 2007.

[10] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (USA), p. 213–223, IEEE Computer Society, 2011.

[11] D. Bruening and D. Lane, "Efcient, transparent, and comprehensive runtime code manipulation," 01 2004.

[12] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, G. Bioworks, and A. Dehon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. ccs," 2013.

[13] R. Intel, "Intel® 64 and ia-32 architectures software developer's manual," vol. 2, 2016.

[14] R. Intel, "Intel 64 and ia-32 architectures optimization reference manual," *Intel Corporation, Sept*, 2014.

[15] "Replacing `malloc`." `https://www.gnu.org/software/libc/manual/html_node/Replacing-malloc.html#Replacing-malloc`.

[16] "SPEC CPU® 2006." `https://www.spec.org/cpu2006/`.

[17] G. Duck, R. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," 01 2017.

Erklärung

Hiermit erkläre ich, Achim Kriso, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift