# Dangless: Safe Dangling Pointer Errors

Gábor Kozár

January 19, 2019

# Contents

Abstract

Manual memory management required in programming languages like C and C++ has its advantages, but comes at a cost in complexity, frequently leading to bugs and security vulnerabilities. One such example is temporal memory errors, whereby an object or memory region is accessed after it has been deallocated. The pointer through which this access occurs is said to be dangling.

Our solution, Dangless, protects against such bugs by ensuring that any references through dangling pointers are caught immediately. This is done by maintaining a unique virtual alias for each individual allocation. We do this efficiently by running the process in a light-weight virtual environment, where the allocator can directly modify the page tables.

We have evaluated performance on the SPEC2006 benchmarking suite, and on a subset of the benchmarks have found a geometric mean of 3.5% runtime performance overhead and 406% memory overhead. This makes this solution very efficient in performance - comparable to other state-of-the-art solutions - but the high memory overhead limits its usability in practice.

# Chapter 1

# Introduction

## 1.1 Motivation

Developing software is difficult. Developing software that is bug-free is all but impossible. Programming languages like C and C++ (the so-called "low-level" programming languages [1]) offer a great deal of control to the programmer, allowing them to write small and efficient computer programs. However, they also require the programmer to take a great deal of care with their development: the same level of control that allows extremely efficient software to be built also places a large burden on the developer, as making mistakes has steeper consequences than in high-level, safer programming languages (such as Java or C#).

Typically, low-level programming languages require the programmers to manage memory manually, while higher-level languages generally include a garbage collector (GC) that frees the programmer from this burden. Managing memory manually means that objects whose lifetime is dynamic (not tied to a particular program scope) have to be *allocated* as well as *deallocated* (freed) explicitly.

In C, such memory allocation typically occurs using the `malloc()` or `calloc()` functions. These allocate a region inside the *heap memory* of the application, reserving it for use, and returning a pointer (typically, an untyped `void *`) to it. On the x86 and x86-64 architectures, which we will mainly concern ourselves with in this thesis, a pointer is just a linear memory

[1] "low-level" is traditionally used to indicate that the level of abstraction used by these programming languages is relatively close to that of the hardware

address: a number representing the index of the first byte of the pointed region in the main memory. This makes pointer arithmetic, such as accessing `numbers[4]` in a `int *numbers` very easy and efficient to perform: just load `sizeof(int)` bytes from the memory address `(uintptr_t)numbers + 4 * sizeof(int)`. Conversely, after we are done with using a given memory region, we can and should deallocate it using the `free()` function. This marks the memory region as no longer in use, and potentially reusable – a characteristic that forms the basis of this thesis.

In C++, memory allocation typically happens using the `new` or `new[]` operators, and deallocation using the `delete` or `delete[]` operators. However, these behave exactly like C's `malloc()` and `free()` in all ways that are important from a memory management point of view. I should note that in modern C++, the use of such memory management is discouraged and generally unnecessary since *smart pointers* – wrappers around pointers that automate the lifecycle management of the pointed memory region, conceptually similarly to a garbage collector – were introduced. However, a lot of applications are still being developed and maintained that do not make use of such features.

Manual memory management is easy to make mistakes with. Allocating but not freeing a memory region even after it's no longer used is called a *memory leak* and it increases the memory usage of the application, often in an unbounded manner, potentially until a crash occurs due to insufficient memory. Attempting to deallocate an object twice – *double free* – causes the memory allocator to attempt to access accounting data stored typically alongside the user data in the since-freed region, often leading to seemingly nonsensical behaviour or a crash, given that the region may have been reused. Accessing an offset that falls outside the memory region reserved for the object – *out-of-bounds access*, sometimes also called *buffer overflow* or *buffer underflow* – can lead to reading unexpected data or overwriting an unrelated object, again often causing hard-to-understand bugs and crashes. One example for this would be attempting to write to `numbers[5]` when only enough space to hold 5 elements was allocated, e.g. using `int *numbers = malloc(5 * sizeof(int))` (recall that in C and related languages, indexes start at 0, so the first item is located at index 0, the second at index 1, and so on). Finally, accessing a memory region that has been deallocated – *use after free* – is similarly problematic. This generally occurs when a pointer is not cleaned up along with the object it referenced, leaving it dangling; often also called a *dangling pointer*.

Besides the instability and general mayhem that such memory errors routinely cause, they can also leave the application vulnerable to attacks, for instance by enabling unchecked reads or writes to sensitive data, sometimes even allowing an attacker to hijack the control flow of the application, execute almost arbitrary instructions, and in essence take over the application.

It should be clear by now why modern, "high-level" programming languages restrict or completely prohibit the direct use of pointers, often by making it impossible and unnecessary to manage memory manually. In such languages, the programmer can perform allocations only by creating objects (preventing another class of bugs relating to the use of uninitialized memory), and leaving their deallocation to the runtime environment, commonly its component called the garbage collector (GC). The GC will periodically analyse the memory of the application, and upon finding objects that are no longer referenced, marks them for reclaiming, and eventually deallocating them automatically. This, of course, comes at a cost in performance, often one that's unpredictable as the GC is controlled by the runtime environment as opposed to the user code. (It's worth noting that this scheme doesn't protect against all possibilities of memory errors; for instance, leaks are still both possible and common.)

A notable exception is the Rust programming language, which, while does allow pointers, heavily restricts how they can be used, preventing any code that could potentially be unsafe. It does so using static (compile-time) checking using their so-called *borrow checker*. However, realizing that in doing so it also disallows some valid uses of pointers, it also provides an escape hatch, allowing code sections to be marked as *unsafe* and go unchecked. (For example, it's not possible to implement a linked list in safe Rust, and even the built-in `vec` type is written using unsafe code.) Another programming language that follows a similar pattern is C#: normally used as a high-level, managed language employing a GC, it also allows pointers to be used directly in code marked as unsafe [2].

Still, applications written in languages like C or (older) C++ with no safe alternatives to pointers have been written and are being maintained, and these applications remain affected by memory errors. Significant amount of

[2]Usage of raw pointers in an otherwise managed environment comes with caveats; for instance, the memory is often compacted after GC passes with the surviving objects moved next to each other to reduce fragmentation. Such relocation is not possible if there are raw pointers in play; therefore, programmers are required to mark the pointers they use as *pinned* using the `fixed()` construct.

research has been and continues to be conducted in this topic, as such applications are often high-value targets for attackers: operating system kernels, device drivers, web servers, anti-virus programs are commonly developed using these technologies.

This thesis is focused specifically on dangling pointer errors, a class of memory issues defending against which has traditionally been difficult and inefficient. **TODO: structure**

**TODO: Talk about virtual memory in general?**

## 1.2 Dangling pointers

A *dangling pointer* is a pointer which outlives the memory region it references. Subsequent accesses to the pointer usually lead to unwanted, confusing behaviour.

In the very best-case scenario, the memory access fails, and the application is killed by the operating system kernel; for example on Unix systems by sending it a signal like `SIGSEGV`, leading to the well-known "Segmentation fault" error and a groan from the programmer. This is useful (and often highly underrated by programmers), because it clearly indicates a bug, and the responsible memory address is readily available, greatly helping with debugging.

Unfortunately, in the majority of cases in practice, the memory access will not fail. The reason for this is that most modern architectures in widespread use (such as x86 and ARM) handle memory on the granularity of *pages*, where a single page is usually 4096 bytes (4 kilobytes). From the point of view of the hardware, and so the kernel, a page is either in use or is not; pages are treated as a unit and are never split up. Of course, typical memory allocations tend to be significantly smaller than this, and it would be wasteful to dedicate an entire page of memory to just hold for instance 200 bytes of user data.

Therefore, all memory allocator implementations used in practice do split up pages, and will readily place two allocations on the same page, typically even directly next to each other (not counting any meta-data). After the deallocation of one of the objects, the page as a whole still remains in use, and so the hardware will not fault on subsequent accesses to it, regardless of the offset; see Figure 1.1. Notably, even if a memory page holds no live objects, it's often still not returned to the system; the memory allocator

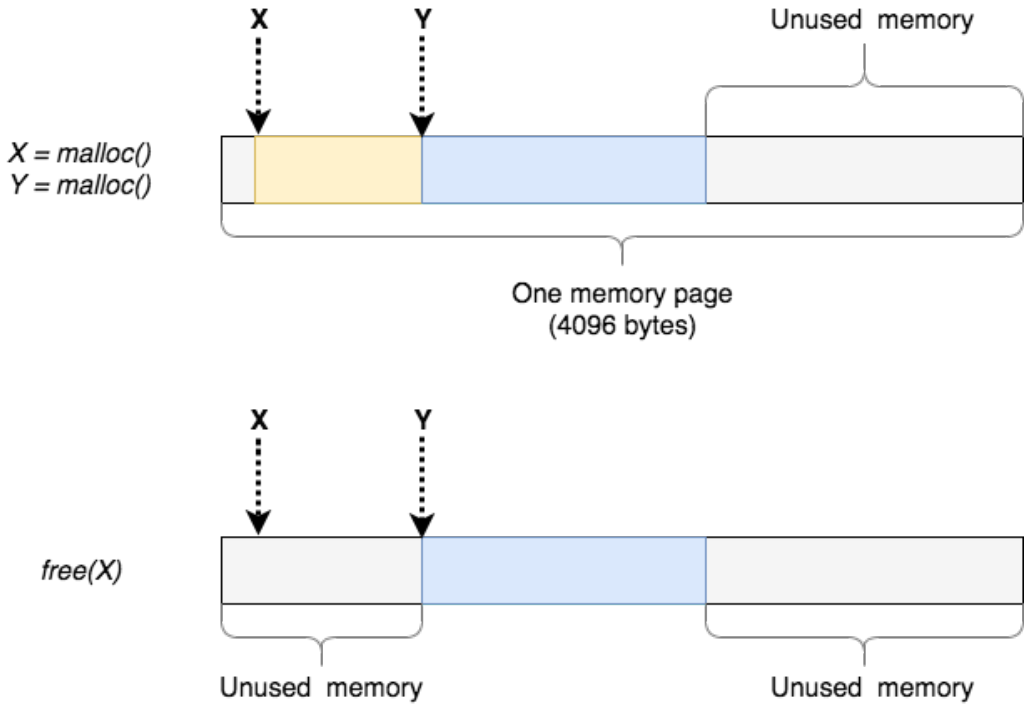


Figure 1.1: Memory layout of two small allocations. $X$ and $Y$ are pointers, referencing their corresponding memory regions. $X$ becomes dangling

retains it as an optimization, expecting more allocations in the future. (This is because the memory allocators being discussed run in user-space, so in order to gain access to memory pages to use, they have to perform a system call such as `mmap()` or `brk()`, which is costly.)

If a page is known to be unused by the hardware and kernel, then accessing it will trigger a page fault in the kernel, which will generally terminate the application, leading to the best-case scenario described earlier. This is a far more manageable problem than the alternative, because the error is clear, even if in practice it's often difficult to discover the underlying reason, given that time may have passed between the deallocation and attempted access, and so the code executing at the time of access may not have any relation to the code that was responsible for the deallocation. Furthermore, this scenario doesn't generally pose a security problem, as a crashed application is difficult to exploit. Therefore, I will generally ignore this scenario for the remainder of this thesis.

The effect of an unchecked access through a dangling pointer depends on whether or not the referenced memory region has been reused since the time of deallocation. If it hasn't, the data read often will still be valid, and execution may continue without anyone the wiser, masking the bug until a modification in the code or one of the libraries leads to a change in the memory allocation pattern. Otherwise, the data read or overwritten will almost always be a source of unexpected behaviour. One typical case is *type confusion*: the value read or written will be treated as a different type than the value stored there. A string value can be for instance accessed as an integer, causing the characters that make up the string to be interpreted as bytes in an integer, essentially leading to the same behaviour as this code snippet:

```
1 char *s = "foobar";
2 int i = *(int *)s;
```

This code compiles and runs successfully. What will be the value of `i`? Of course, we are deep into undefined behaviour territory here, meaning that the programming language promises nothing. In practice, on x86-64 architectures where the `int` C type is 4 bytes long (`sizeof(int)== 4`), the result will typically be `1651470182`, or `0x626f6f66` in hexadecimal. This makes sense: the string `"foobar"` (including the null terminator) is represented by the byte sequence `0x66 0x6f 0x6f 0x62 0x61 0x72 0x00`. Interpreting it as an `int` means reading the first 4 bytes (`0x66 0x6f 0x6f 0x62`) and assembling it into a multi-byte integer according to the endianness of the processor. My laptop has an Intel CPU in it, which is little endian, meaning that the bytes of an integral type are stored as least significant byte first (this is `0x62`), followed by bytes of increasing significance; simply put, bytes are interpreted in "reverse order".

## 1.3 Prior art

## 1.4 Background

# Chapter 2

# Evaluation

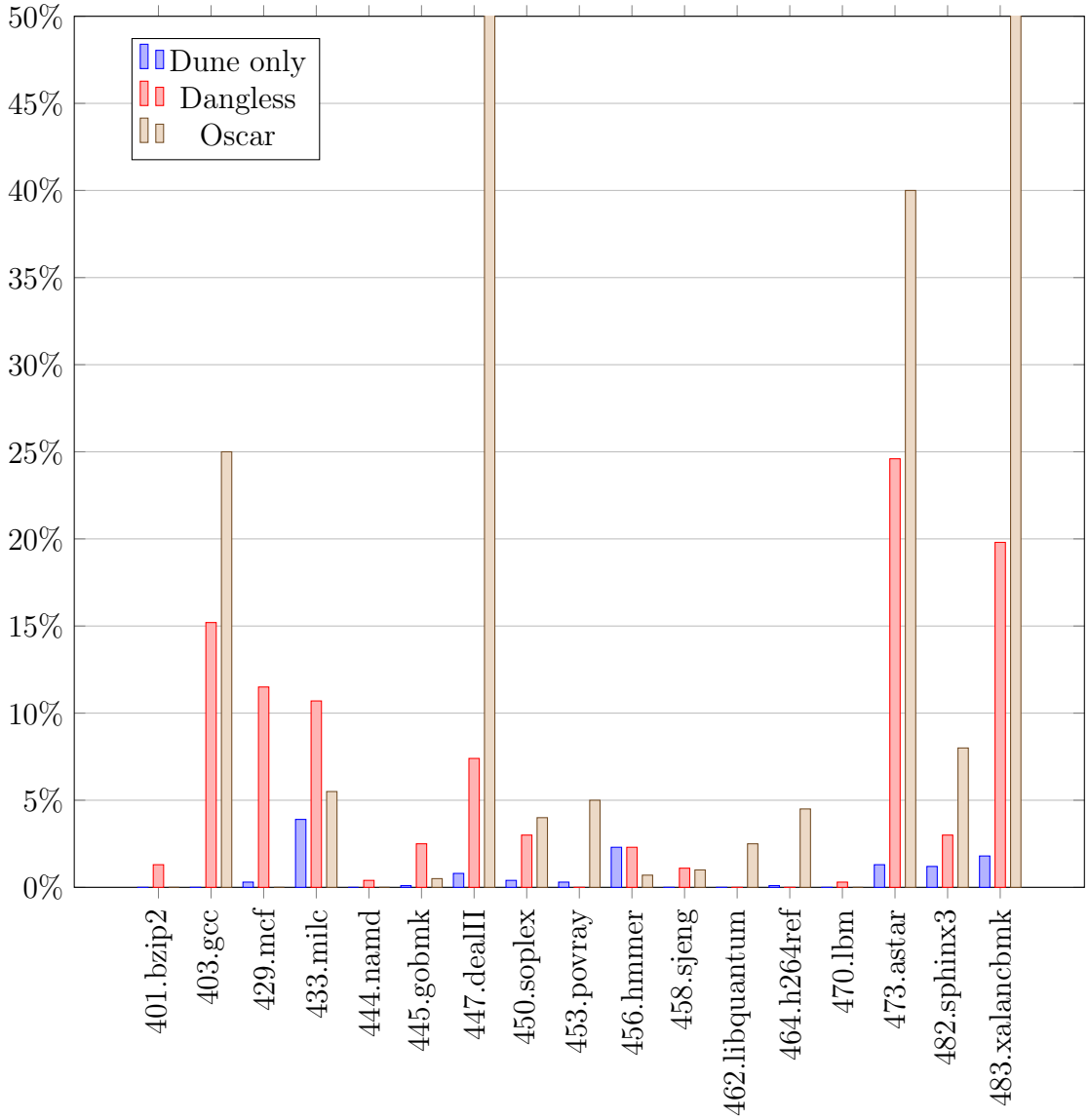


Figure 2.1: Dune vs Dangless vs Oscar: performance overhead

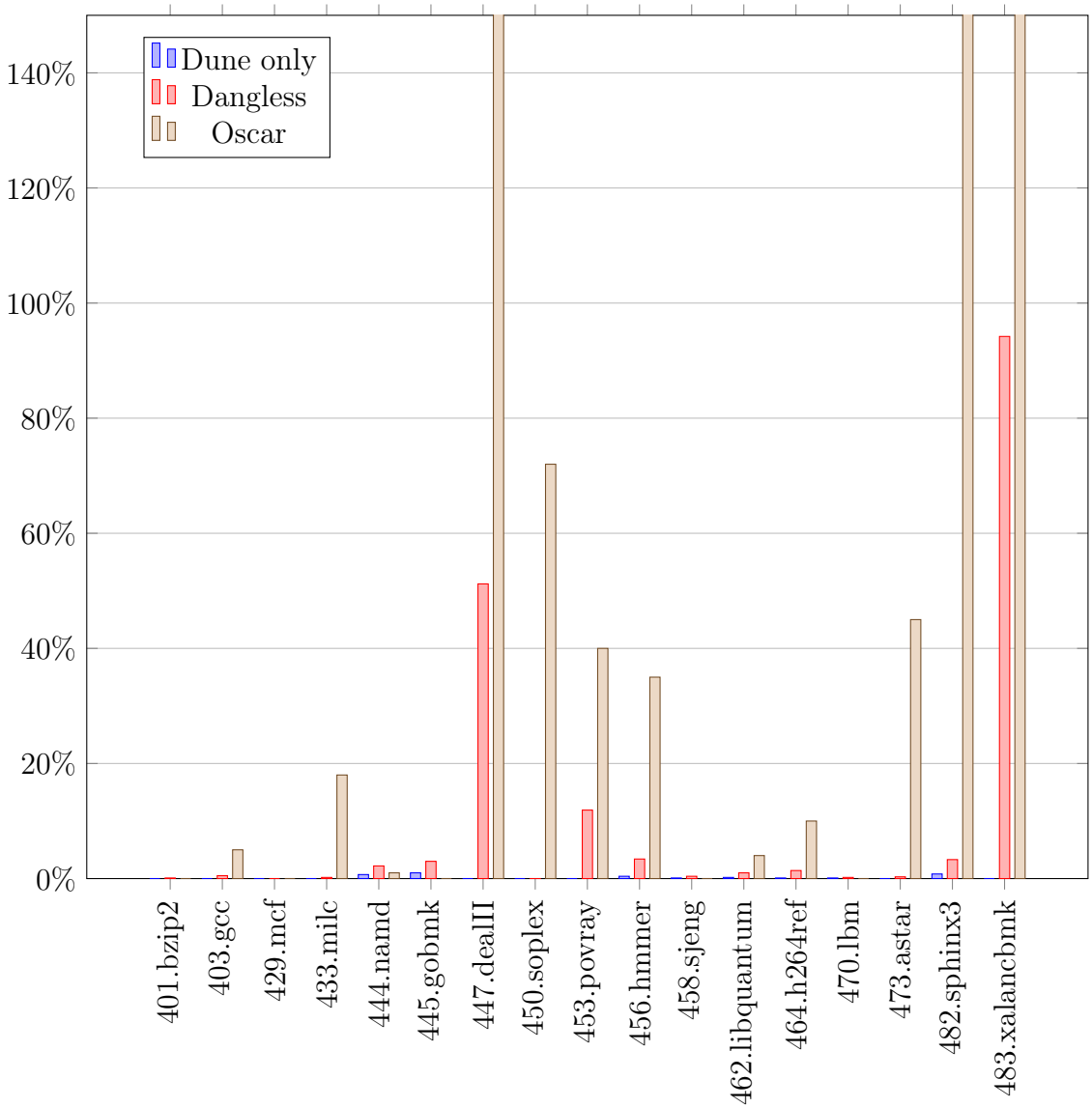


Figure 2.2: Dune vs Dangless vs Oscar: memory overhead

# Chapter 3

# The memory allocation design

## 3.1 Physical and virtual memory

## 3.2 Virtual aliasing

# Chapter 4

# Implementation

## 4.1 Virtualization

## 4.2 Physical memory allocation

## 4.3 Virtual memory allocation

## 4.4 Virtual aliases and `vmcall`s

# Chapter 5

# Evaluation

## 5.1 Overview

## 5.2 Performance evaluation: SPEC 2006

## 5.3 Summary of contributions

## 5.4 Limitations and future work

## 5.5 Conclusion