Dangless: Safe Dangling Pointer Errors

Gábor Kozár

October 29, 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 Memory errors

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.

Making mistakes with manual memory management is very easy. 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 re-used. 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.

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 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 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

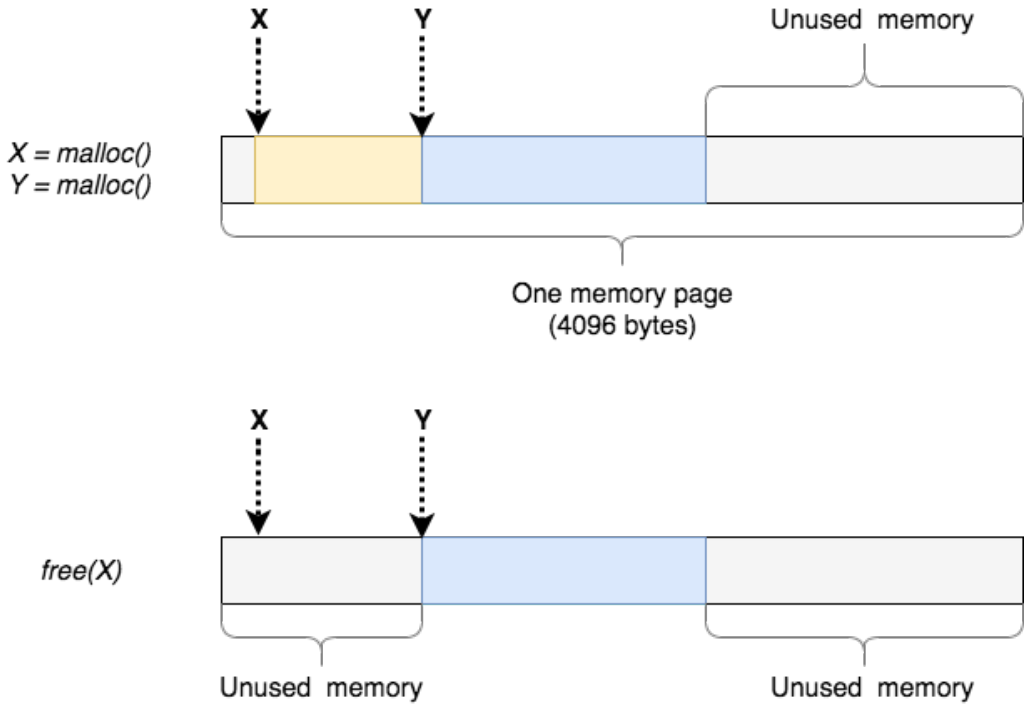

Figure 1.1: Memory layout of two small allocations. X and Y are pointers, referencing their corresponding memory regions. X becomes dangling

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

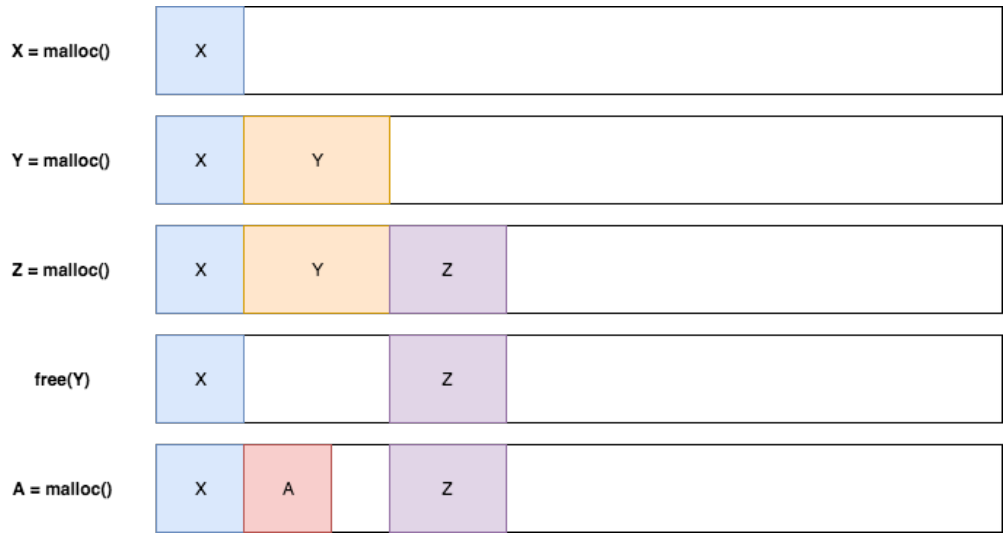

Figure 1.2: (Simplified) Memory layout view when using traditional malloc(): memory is re-used

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 – at least until a modification in the code or one of the libraries leads to a change in the memory allocation pattern. Otherwise, the dangling pointer will point inside another allocation, and the data read or overwritten will almost always be a source of unexpected behaviour: see Figure 1.2.

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".

Of course, type confusion doesn't have to occur in order for invalid behaviour to occur. For instance, overwriting an Unix file descriptor with the number of characters in a text will typically result in an invalid file descriptor; or consider a buffer's length overwritten by the age of the user; or an integer representing the next free index in an array overwritten by the length of a file in bytes. Once memory corruption occurs, sanity flees.

1.3 An example

Let's look at a less trivial example. This is a simplistic codebase, written in C++ of an in-memory messaging system. Each `User` has an inbox and outbox, `Mailbox` objects, which wrap an `std::vector<Message *>`. `Message` objects are allocated on the heap, referenced by plain pointers, allowing the sender and recipient mailboxes to just both retain a pointer to the same message – a memory optimization. Each message object keeps track of who has deleted it, and when both the sender and receiver have done so, the message can be safely deallocated.

```
1  struct Message {
2      const std::string mContent;
3
4      bool mHasSenderDeleted = false;
5      bool mHasRecipientDeleted = false;
6
7      explicit Message(std::string content)
8          :   mContent{std::move(content)}
9      {}
10
11     void OnDeleted() {
12         if (mHasSenderDeleted && mHasRecipientDeleted)
13             delete this;
14     }
```

```
15 };
16
17 struct Mailbox {
18     std::vector<Message*> mMessages;
19
20     void AddMessage(Message* msg) {
21         mMessages.push_back(msg);
22     }
23
24     void DeleteMessage(Message* msg) {
25         mMessages.erase(std::find(mMessages.begin(), ↩
              mMessages.end(), msg));
26         msg->OnDeleted();
27     }
28 };
29
30 struct User {
31     Mailbox mInbox;
32     Mailbox mOutbox;
33
34     void SendMessage(User& recipient, std::string ↩
          content) {
35         Message* msg = new Message{std::move(content)};
36
37         mOutbox.AddMessage(msg);
38         recipient.mInbox.AddMessage(msg);
39     }
40
41     void DeleteReceivedMessage(Message* msg) {
42         msg->mHasRecipientDeleted = true;
43         mInbox.DeleteMessage(msg);
44     }
45
46     void DeleteSentMessage(Message* msg) {
47         msg->mHasSenderDeleted = true;
48         mOutbox.DeleteMessage(msg);
49     }
50 };
```

The noteworthy lines have been highlighted. While this design is error-prone, as we will see, it does work correctly and does not – in its current

form – represent a vulnerability.

However, code evolves over time as bugs are fixed and new features are added, often by a different developer than the original authors. Sometimes these programmers understand the codebase less, or have less experience with programming or the technologies used, and can easily make mistakes. Especially with a language like C and C++, mistakes are extremely easy to make, and sometimes hard to notice, let alone debug.

Consider now that another programmer comes along and has to implement a feature to allow forwarding messages. His deadline is in an hour, perhaps there is a presentation scheduled with a big client, and this feature was simply forgotten about until now. This programmer adds a simple function as a quick hack to get message forwarding to work, and schedules some time for next month to revisit the feature and implement it properly. This function is added:

```
1  struct User {
2  // ...
3
4      void ForwardMessage(User& recipient, Message* ↩
           msg) {
5          // TODO: do this properly later
6          recipient.mInbox.AddMessage(msg);
7      }
8
9  // ...
10 };
```

He didn't understand how the simplistic reference counting of the `Message` objects work, and a quick test showed that this feature seems to work reasonably well. His attention was quickly drawn away by another tasks and this code won't be revisited for a while.

The problem shows itself when a message that was forwarded gets destroyed. While the code correctly ensures that the message is removed from the both the sender and the recipient's mailbox before it can be destroyed, but any potential forwardees were not taken into account. Consider now the following chain of events:

```
1  Message* funnyMessage = bob.SendMessage(alice, "Hey, ↩
       look at this funny gif: <image>");
2  bob.DeleteSentMessage(funnyMessage);
3
```

```
4 alice.SendMessage(cecile, "Haha, look at this funny ↩
     gif!");
5 alice.ForwardMessage(cecile, funnyMessage);
6
7 alice.SendMessage(bob, "HAHA that's pretty awesome");
8 alice.DeleteReceivedMessage(funnyMessage);
```

Bob sends a message to Alice, who forwards it to Cecile. Both Bob and Alice delete the message, causing the object to be destroyed, while Cecile's inbox still retains a pointer to it: a dangling pointer! What will happen if Cecile looks at her inbox? The application will attempt to dereference the dangling pointer, with unpredictable results.

What if there's another message, containing some sensitive information, is sent directly afterwards, potentially between completely unrelated users?

```
1 bob.SendMessage(daniel, "My PIN code is 6666");
```

Depending on the memory allocator, it's entirely possible that the memory referenced by `funnyMessage` before is now reused for the new, secret message. In this case, Cecile's inbox now contains a message not intended for her, containing sensitive information.

```
1 for (const Message* msg : cecile.mInbox.mMessages) {
2     std::cerr << msg->mContent << "\n";
3 }
```

The following output is produced when compiled with a recent version of GCC (regardless of optimizations or other options) and run:

```
Haha, look at this funny gif!
My PIN code is 6666
```

This is an example of how dangling pointer errors can pose a security vulnerability, even without the active efforts of an attacker.

It's worth noting that using a correct implementation of reference-counting, such as with the standard `std::shared_ptr` (since C++11), this problem could have been avoided. However, while smart pointers go a long way towards making dynamic memory safer and more convenient to use, they do have limitations even in the current C++ version (C++17 as of the time of writing). For instance, it's common to use plain pointers to represent a non-owning, optional reference to memory owned by another object, such as an `std::unique_ptr`, enabling dangling pointer errors to occur.

1.3.1 Dangling pointers and late binding in C++

Dangling pointers are an even more severe vulnerability in C++ because it supports the object-oriented programming paradigm, and therefore allows programmers to define classes, virtual methods, and express inheritance. In order to support virtual methods being overridden in derived classes, the C++ compiler creates a data structure called a `vtable` for each class that contains virtual methods, whether defined in the class or inherited from a base class. This `vtable` is essentially a look-up table, containing function pointers to the class' own implementations of the virtual methods. Furthermore, an additional `vptr` pointer field is added to any objects of such classes, which points to the correct `vtable`. This pointer value persists even through derived-to-base casts, to allow the program to behave as expected. For instance:

```
1 class Base {
2 public:
3     virtual ~Base() = default;
4
5     virtual std::string getTypeName() const {
6         return "Base";
7     }
8 };
```

This defines a class `Base` with a virtual method `getTypeName()`, the default implementation of which returns the string `"Base"`. Since the class contains virtual methods, the C++ compiler emits a `vtable` for it with 2 entries: one for the destructor, and one for `getTypeName()`, both of them pointing to `Base`'s own implementations. When we call the `Base()` constructor and instantiate an object of the class, it will contain a hidden field (usually as the first field, placed before any user-defined ones): the `vptr` that points to Base's own `vtable`. (The destructor has to be virtual, otherwise objects may not be correctly destroyed.)

```
1 class Derived : public Base {
2     std::string getTypeName() const override {
3         return "Derived";
4     }
5 };
```

We have now defined a second class which inherits from `Base` and overrides `getTypeName()` with its own implementation, one that returns `"Derived"`.

(The compiler also automatically emits a trivial destructor that overrides the base class'.) The `vtable` emitted for `Derived` has the same structure as that of `Base`, but contains the addresses to `Derived`'s virtual method implementations. Similarly, when a `Derived` object is constructed, the object contains a `vptr` hidden field pointing to its `vtable`.

```
1 void print(const Base& obj) {
2     std::cout << obj.getTypeName() << '\n';
3 }
4
5 int main() {
6     print(Base{}); // prints "Base"
7     print(Derived{}); // prints "Derived"
8     return 0;
9 }
```

Note how the global function `print()` accepts a `const Base&`: this function has no idea about any derived of `Base`. Yet when it calls the `getTypeName()` method on it, the actual function executed is different between the two calls. The reason is that when a virtual method is called, the C++ compiler will emit code that dereferences the `vptr` field, searches the referenced `vtable` for the entry corresponding to the method being called, and performs an indirect function call to the address contained in the entry. This is how late binding function calls are implemented in C++.

Now we know enough to understand why dangling pointers are particularly dangerous in applications written in C++: should the attacker be able to construct its own fake `vtable` in memory, as well as get the application to perform a virtual method call on an object whose `vptr` field it managed to overwrite with one pointing to the fake `vtable`, the attacker can hijack the control flow of the application. This can be tricky to do, as the `vptr` is typically at the same offset in all objects, which makes it more difficult to overwrite with attacker-controlled data.

It's also worth noting that although rare, objects of classes that inherit from multiple base classes (since C++ allows multiple inheritance) have multiple `vptr` fields at different offsets. This potentially makes the job of the attacker easier. Furthermore, access through dangling pointers are not the only way to perform such an attack: a buffer overflow or underflow vulnerability in the object can also enable the attacker to hijack the `vptr` field.

Defenses against such attacks do exist: examples include SafeDispatch [24], work by Gawlik et al [23], or the research of Bounov et al [20].

Chapter 2

Background

2.1 Virtual memory

Virtual memory is an abstraction over the physical memory available to the hardware. It's an abstraction that is typically transparent to both the applications and developers, meaning that they do not have to be aware of it, while enjoying the significant benefits. This is enabled by the hardware and operating system kernel working together in the background.

From a security and stability point of view, the biggest benefit that virtual memory provides is address space isolation: each process executes as if it was the only one running, with all of the memory visible to it belonging either to itself or the kernel. This means that a malicious or misbehaving application cannot directly access the memory of any other process, to either deliberately or due to a programming error expose secrets of the other application (such as passwords or private keys) or destabilize it by corrupting its memory.

An additional security feature is the ability to specify permission flags on individual memory pages: they can be independently made readable, writeable, and executable. For instance, all memory containing application data can be marked as readable, writeable, but not executable, while the memory pages hosting the application code can be made readable, executable, but not writeable, limiting the capabilities of attackers.

Furthermore, virtual memory allows the kernel to optimize physical memory usage by:

- Compressing or swapping out (writing to hard disk) rarely used memory pages (regions) to reduce memory usage

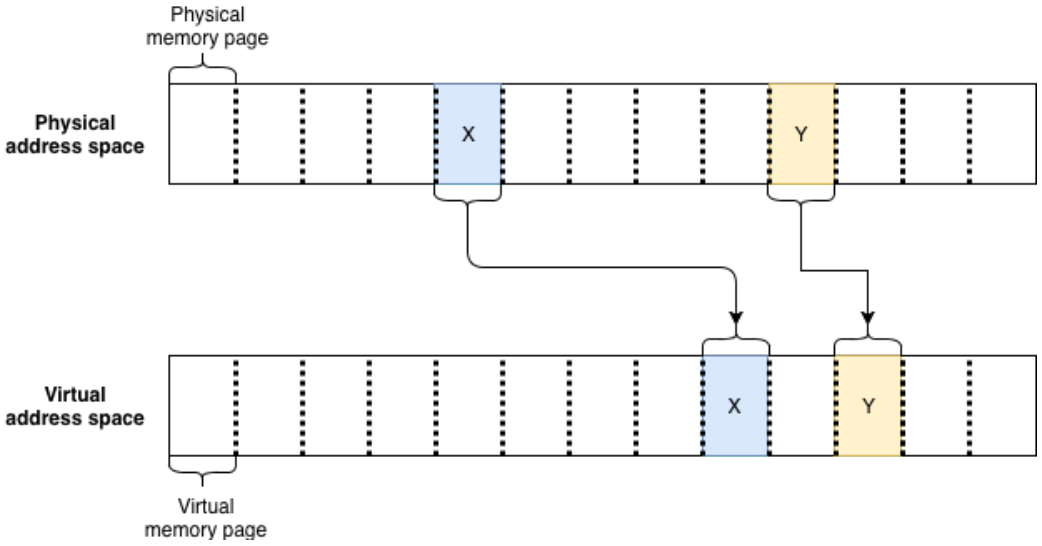

Figure 2.1: Mapping two physical memory pages X and Y to virtual memory

- De-duplicating identical memory pages, such as those resulting from commonly used static or shared libraries
- Lazily allocating memory pages requested by the application

Virtual memory works by creating an artificial (virtual) address space for each process, and then mapping the appropriate regions of it to the backing physical memory. A pointer will reference a location in virtual memory, and upon access, is resolved (typically by the hardware) into a physical memory address. The granularity of the mapping is referred to as a memory page, and is typically 4096 bytes (4 kilobytes) in size. (See Figure 2.1.)

This mapping is encoded in a data structure called the *page table*. This is built up and managed by the kernel: as the application allocates and frees memory, virtual memory mappings have be created and destroyed. The representation of the page table varies depending on the architecture, but on x86-64, it can be represented as a tree, with each node an array of 512 page table entries of 8 bytes each making up a 4096 byte page table page. The root of this tree is where all virtual memory address resolution begins, and it identifies the address space. The leaf nodes are the physical memory pages that contain the application's own data.

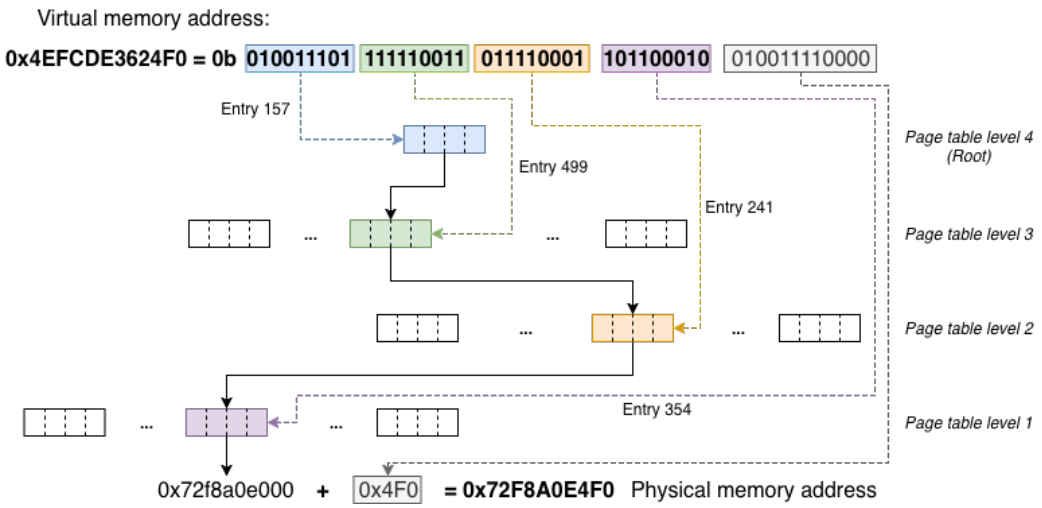

Figure 2.2: Translating a virtual memory address to physical using the page tables

The bits of the virtual memory address identify the page table entry to follow during address resolution. For each level of page tables, 9 bits are required to encode an index into the array of 512 entries. Each entry contains the physical memory address of the next page to traverse during the address resolution, as well as a series of bits that represent the different access permissions, such as writeable and executable. Finally, the least-significant 12 bits are used to address into the application's physical page (which is 4096 bytes) itself and so require no translation. (See Figure 2.2.)

On x86-64, there are currently 4 levels of page tables, using $4*9+12 = 48$ out of the 64 available bits in the memory addresses, and limiting the size of the address space to 2^{48} bytes or 256 terabytes. (The size of addressable space per page table level, in reverse resolution order being: $512 * 4$ kilobytes $= 2$ megabytes; $512 * 2$ megabytes $= 1$ gigabyte; 512 gigabytes; 256 terabytes.)

It's important to realize that it is possible to map a physical page into multiple virtual pages, as well as to have unmapped virtual pages. Attempting to access a virtual page (by dereferencing a pointer to it) that is not mapped – i.e. not backed by a physical memory page – will cause a *page fault* and execution to trap inside the kernel. The kernel then can decide what to do – for instance if it determines that the memory access was in error (an *access violation*), it passes the fault on to the process which usually

terminates it. On Linux this is done by raising the `SIGSEGV` signal (segmentation violation or segmentation fault) in the process, normally aborting its execution.

Other types of access violation, such as attempting to write a non-writeable page – a page on which writing was disallowed by setting the corresponding bit in its page table entry to 0 – or attempting to execute a non-executable page – a page which has its no-execute bit set, a new addition in the x86-64 architecture over x86-32 – will also trigger a page fault in the kernel the same way.

This mechanism also allows the kernel to perform memory optimizations. These are important, because (physical) memory is often a scarce resource. For example, a common scenario is that multiple running processes use the same shared library. The shared library is a single binary file on disk that is loaded into memory by multiple processes, meaning that naively the same data would be loaded into memory multiple times, taking up precious resources for no gain. The kernel can instead load the shared library into physical memory only once and then map this region into the virtual address space of each user application. Other de-duplication opportunities include static libraries that are commonly linked into applications (such as the C standard library), or the same binary executing in multiple processes.

In addition to memory de-duplication, the kernel can also choose to compress or even swap out rarely used memory. In these cases, the kernel marks the relevant page table entries as invalid, causing the hardware to trigger a page fault in the kernel if they are accessed. Upon access, instead of sending a signal to the process, the kernel restores the data by decompressing it or reading it from disk, after which it will resume the process which can continue without even being aware of what happened. Modern kernels include a large number of similar optimizations. **TODO: Could probably find some citations for this**

2.2 Prior art

Memory errors are a well-understood problem and a significant amount of research has already been conducted in this field. Numerous solutions have been proposed, often using similar ideas and offering alternative approaches, optimizations, or other incremental improvements.

Dhurjati et al. [22] proposed a solution for type-safe memory re-use in

2003: although this doesn't prevent all dangling pointer errors, but ensures that type confusion does no occur, limiting the attack vector.

Cling [18] builds on top of the same ideas and is a memory allocator that similarly limits, but does not prevent, the threat surface posed by dangling pointer errors by limiting memory re-use to objects of the same type. It does so by noting down the caller function of each memory allocation, assigning a unique memory pool to each, under the assumption that one call site always performs allocations of the same type.

—

[21]

Some work [20, 17, 9, 8] provides no guarantees, only heuristics; cannot actually detect a dangling pointer after it's been reallocated

SafeC: global capabilities store, fat pointer with a capability value, free removes capability, check before each memory access; good overhead on allocation-heavy software, but increases physical memory usage by 1.6-4x. Improvements by FisherPatil[15] and Xu et al [19].

Purify, Vanguard: for debugging

Electric Fence [16], PageHeap [13] - they use page protections, both are only useful as debugging tools, greatly increasing physical memory usage (single object per physical memory page)

DSN paper: relies on page protections, but does remappings same as Dangless; but also builds upon a compiler transformation called Automatic Pool Allocation to reclaim virtual memory but needs source code for this; does stuff with system calls though so adds an extra system call per allocation and deallocation; increases TLB misses; also increases physical memory usage by 8 bytes per allocation (to record the canonical address)

—

Cling: infers type information at runtime by exmining the call stack ?

Oscar

2.3 Unikernels and Rumprun

The idea of unikernels stems from the observation [25] that in many cloud deployment scenarios, a virtual machine is typically used to host only a single application, such as a web server or database server. These virtual machines all run a commodity operating system – typically a Linux distribution, or less often, Windows – that is built to support multiple users and many arbitrary

applications running concurrently on various hardware. This adds up to a significant amount of effectively dead code in these scenarios. A tiny library operating system with only the necessary modules enabled and tied directly together with the application would result in a much more efficient image.

Given the widespread use of cloud technologies and the prevalence of commodity cloud providers since 2013, unikernels and similar technologies became a topic of research and development, yielding solutions such as MirageOS, IncludeOS, Rump kernels, HaLVM, and OSv [16]. Notably, similar research has also resulted in technologies such as Docker [3].

One approach to building unikernels is basing them on applications developed for commodity operating systems, and then through a mix of manual and automated methods adapt them to be able to run with a unikernel runtime. This approach is taken by Rump kernels [15], which provides various drivers and operating system modules (based on NetBSD) as well as a near-complete POSIX system call interface, allowing many existing applications to be run without any modifications [14].

When the development of Dangless began, it was initially supporting both Rumpkernels and Dune. However, over time, maintaining both has become a major burden, and focus shifted to only supporting Dune. One of the reasons for this was entirely practical: development of Dangless on Rumpkernels was more difficult, due to the complete virtual machine isolation which is required to run them, making debugging and data collection (statistics, logging) more difficult.

Furthermore, Rumpkernels would have also needed custom modifications (similarly to how Dune ended up requiring them), for instance in order to make it possible for Dangless to provide the symbols for `malloc()` and co. while Rumpkernels also define the same strong symbols.

Regardless, I still believe that unikernels are a promising field, because it's focused on providing environments for efficiently hosting long-running, publicly exposed services such as web servers, which are particularly relevant for security research, and by extension for Dangless itself.

2.4 Dune: light-weight process virtualization

Dune [4] is a technology developed to enable the development of Linux applications that can run on an unmodified Linux kernel while having the ability to directly and safely (in isolation from the rest of the system) access hard-

ware features normally reserved for the kernel (ring 0) code [19]. Importantly, while getting all the benefits of having direct access to privileged hardware features, the application still has access to the Linux host operating system's interface (system calls) and features. This means that the same process that can, for instance, directly manipulate its own interrupt descriptor table, can also call a normal `fopen()` function (or `open()` Linux system call) and it'll behave as expected: the system call will pass through to the host kernel.

This is achieved using hardware-assisted virtualization (Intel VT-x) on the process level, rather than on the more common machine level. Dune consists of a kernel module `dune.ko` for x86-64 Linux that initializes the virtual environment and mediates between the Dune-mode process and the host kernel, as well as a user-level library called `libdune` for setting up and managing the virtualized (guest) hardware. The two components communicate via the `ioctl()` system call on the `/dev/dune` special device that is exposed by the kernel module. Finally, `libdune/dune.h` exposes a number of functions to help the application manage the now-accessible hardware features.

An application wishing to use Dune has to statically link to `libdune.a`, and call `dune_init()` and `dune_enter()` to enter Dune mode. For this to succeed, the Dune kernel mode has to be already loaded. That done, the application keeps running as before: file descriptors remain valid, system calls continue to work, and so on, except privileged hardware features also become available. This opens up drastically more efficient methods of implementing some applications, such as those utilizing garbage collection, process migration, and sandboxing untrusted code. It also enables Dangless to work efficiently.

2.4.1 Patching Dune

Dangless was built using ix-project's fork of Dune [5], because when the Dangless project started that was more maintained and supported more recent kernel versions. However, since then, work appears to have resumed on the original Dune repository [6], and they now claim to be supporting Linux kernel versions 4.x.

I have patched Dune with a couple of modifications to allow Dangless to do its work. These are available in the `vendor` directory in the Dangless source code.

`dune-ix-guestppages.patch`: this maps the guest system's pagetable

pages into its own virtual memory, allowing Dangless to modify them. The code for this already existed in the original implementation of Dune, but was removed at some point from the ix-project's fork.

`dune-ix-nosigterm.patch`: for unclear reasons, upon exiting, Dune was raising the `SIGTERM` signal, causing the Dune process to appear to have crashed even when it exited normally. This patch disables this behaviour, without appearing to affect anything else.

`dune-ix-vmcallhooks.patch`: this is a significant patch that allows a pre- and post-hook to be registered for `vmcall`-s. That is, any time a system call is not handled inside Dune, and is about to be forwarded to the host kernel, the pre-hook, if set, is invoked with the system call number, arguments, and return address, allowing the hook to inspect and modify them. Similarly, once the `vmcall` returns, but before the normal code execution resumes, the post-hook is invoked, with the system call result passed to it as argument. This is critical for Dangless, and is explained in detail by Section 3.4.

Finally, Dangless contains a work-around for an issue in Dune which appears with allocation-heavy code. `glibc`'s default `malloc()` implementation relies on the `brk()` system call to perform small memory allocations (in my tests, up to 9000 bytes). After a sufficient number of `brk()` calls, the resulting address crosses the 4 GB boundary, i.e. the memory address `0x100000000`. However, this area doesn't appear to be mapped by Dune in the embedded page table, causing an EPT violation error on access. In the Dangless source code, `testapps/memstress` can be used to trigger this bug:

```
$ cd build/testapps/memstress
$ ./memstress 1500000 9000
```

To work around this, Dangless registers a post-vmcall hook, and if it detects a `brk()` system call with a parameter above the 4 GB mark, it overwrites the system call's return code to make it appear to have failed. `glibc`'s memory allocator will then fall back to using `mmap()` even for these smaller allocations, which will continue to work.

2.4.2 Memory layout

Dune offers two memory layouts, specified when calling `dune_init()`: precise mappings and full mappings. With precise mappings, Dune will consult

`/proc/self/maps` for the memory regions of the so-far ordinary Linux process and map each region found there into the guest environment. With the version of Dune I was using for development, this did not appear to work correctly, and so I used full mappings for Dangless.

In full mappings, Dune creates the following mappings in the guest virtual memory:

- Identity-mapping for the first 4 GB of memory: this is where the executable code and data lives, so this is also where the default `malloc()` implementation places small memory allocations (allocated via `brk()`)
- The stack memory, max 1 GB
- The mmap memory region, max 63 GB
- The static `VDSO` and `VVAR` regions
- Any mappings needed for the executable and libraries (code and data)

Furthermore, the Dune kernel module creates the embedded page table (EPT), the part of the Intel VT-x virtualization technology that is used for translating host virtual addresses (HVA) to guest physical addresses (GPA) and vica versa. The EPT is set up and is kept in sync with the process memory automatically, meaning that mappings are created on-demand on EPT faults, within the limits of the layout defined by Dune and described above.

Dangless makes extensive use of the knowledge of the guest memory layout, for instance for translating physical addresses (e.g. of pagetable pages) to virtual addresses.

Chapter 3

Dangless – Implementation

3.1 Overview

Dangless is a drop-in replacement memory allocator that provides a custom implementation of the standard C memory management functions `malloc()`, `calloc()`, `realloc()`, `free()` and a few others. It aims to solve the problems that dangling pointers lead to by guaranteeing that any access through a dangling pointer will fail, and the application will terminate. It relies on the underlying memory allocator to perform the actual allocation and deallocation. In principle, because Dangless makes no assumptions on the nature or behaviour of the underlying allocator – referred to as "system allocator" by Dangless –, it should be usable on top of even non-standard implementations such as Google tcmalloc [11].

Catching dangling pointer accesses is ensured by *permanently* marking memory regions as no longer in use upon deallocation (e.g. a call to `free()`). Therefore, during the lifetime of the application, in principle, no other memory will be allocated in such a way that it would visibly alias a previously used location: see Figure 3.1.

Of course, the physical memory available is very limited even on modern systems, and not re-using is hopeless. The trick then, is to leave the management of physical memory to the system allocator, and change how the physical allocations are mapped to virtual memory: the address space that user applications interact with.

Since we want to rely on the system allocator to efficiently manage physical memory, instead of hijacking the pointer it returns following a successful

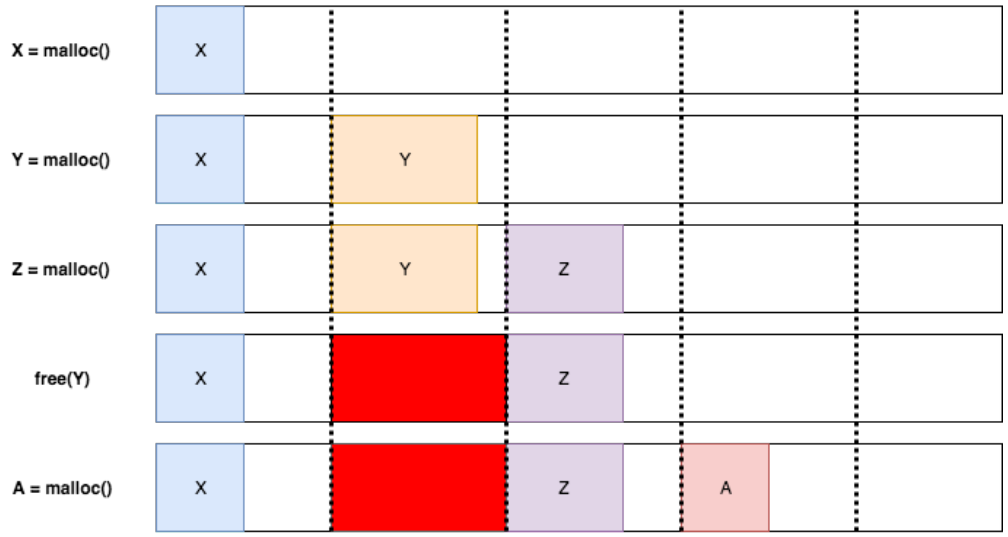

Figure 3.1: A simplified view of the virtual memory address space when using Dangless: memory pages are never re-used.

allocation, we rather *re-map* the physical memory region into a new virtual memory region that's entirely controlled by us. This means that the same allocation will be visible at two virtual memory addresses: the canonical address, managed by the system allocator but not returned to the user code; and the remapped address, managed by Dangless. Note that this re-mapping occurs on the page level (as all virtual memory management has to be), leading to each allocation using up at least one virtual memory page, even if it's smaller than that. See Figure 3.2 and Figure 3.3. A detailed explanation of these diagrams will follow in Section 3.3.3.

Virtual memory is plentiful: on the x86-64 architecture, pointers are 64-bit long, which in theory means 2^{64} bytes of addressable memory. In practice however, on all current processors that use this architecture, only 48 bits are used, which limits the size of the address space we can work with to 2^{48} bytes, or 256 terrabytes. That's also not unlimited, but in practice it's close enough to be sufficient. **TODO: Need to elaborate on this somewhere, or cite**

Normally, the difference between physical and virtual memory is entirely hidden from the user code, and is dealt with only by the operating system kernel. This allows the overwhelming majority of users and developers - even programmers working with lower-level languages such as C++ - to work and

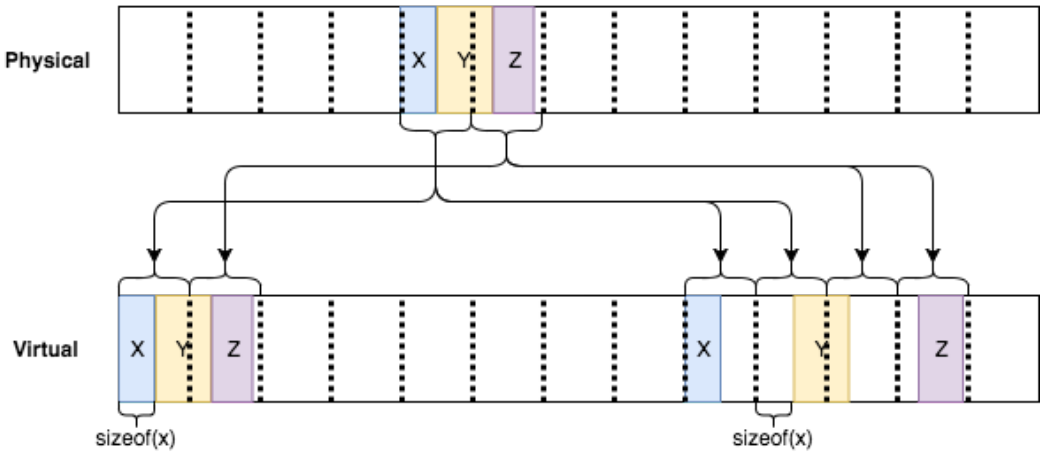

Figure 3.2: Remapping the same physical memory region into a new virtual memory region: the same allocation will be visible in virtual memory at two addresses.

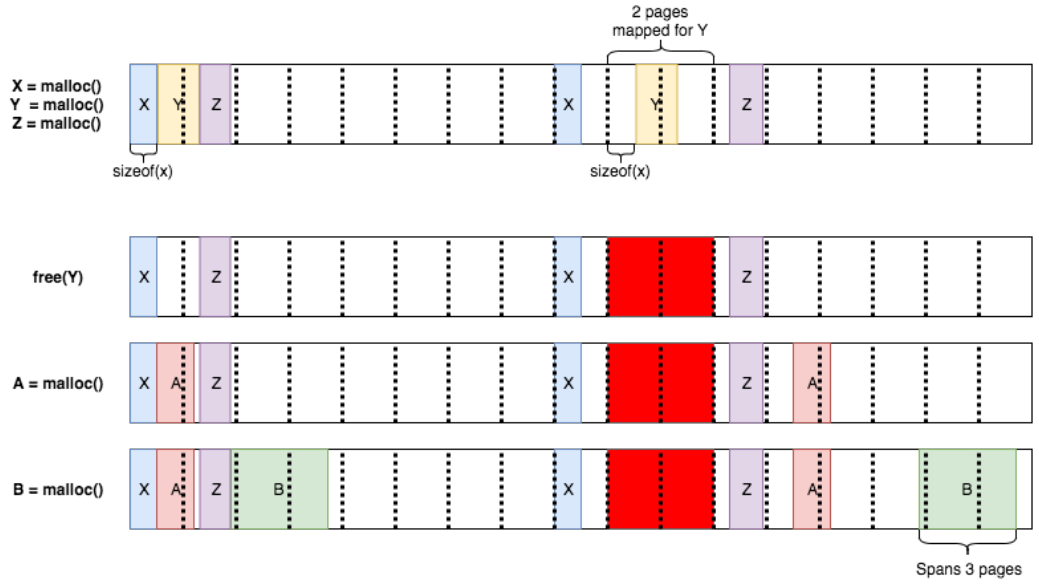

Figure 3.3: View of the virtual memory with Dangless remapping allocations

develop software without ever being aware of the difference, while enjoying the benefits of it. The Linux kernel does provide some system calls that allow the virtual memory to be manipulated, notably `mprotect()` which is used to manage access permissions (readable, writeable, executable) of memory regions. This is useful for example when developing just-in-time (JIT) compilers such as the ones employed by browsers to run JavaScript code. Another example is `mremap()`, which allows a memory region to be moved almost for free, an ability that makes it useful for garbage collectors for instance. Of course, `mmap()` and `munmap()` also primarily work by manipulating virtual memory mappings.

These system calls are sufficient to implement the functionality of Dangless, with some caveats – in fact, this is exactly how Oscar operates **TODO: reference**. The biggest issue is that of performance: system calls are expensive compared to normal memory allocations, and in this scheme, for every single memory allocation, at least one extra system call would be required. The costs and possibilities for optimizations are explored in depth by the Oscar paper.

3.2 Initialization

Dangless has to be initialized before any memory allocation is performed, otherwise those allocations will not be protected. The `REGISTER_PREINIT` option, enabled by default, controls whether Dangless should automatically register its initialization function (`dangless_init()`) to the `.preinit_array` section of the executable, to be called automatically during start-up.

During initialization, Dangless initializes and enters Dune by calling `dune_init()` and `dune_enter()`. Dangless relies on Dune to enter into a virtualized environment where it can have direct access to the page tables. Dangless also uses Dune to register its own pagefault handler function, which enables us to detect when a memory access has failed due to the protection that Dangless offers.

It's important to note that heap memory allocation can and does happen *before* Dangless is initialized – whether manually or automatically – for example as part of the glibc runtime initialization. This case needs to be handled, so all of the `dangless_` functions simply pass the call through to the underlying (system) allocator without doing anything else if they are called before initialization. A noteworthy edge-case that Dangless has to be able

to handle is when an allocation happens before Dangless initialization, so is not protected, but then it's used and finally deallocated after initialization.

3.3 Performing an allocation

Whenever Dangless is asked to allocate some memory via a call to `dangless_malloc()`, `dangless_calloc()`, or `dangless_realloc()`, a number of steps have to happen: physical memory has to be allocated, virtual memory has to be allocated, and the mapping between the two created. Most of the process is the same regardless of the exact function called. The only exception is `dangless_realloc()`, which I will cover later.

3.3.1 Allocating physical memory

The first step Dangelss has to perform is to acquire the physical memory it can use to satisfy the allocation. It does not currently defer allocating physical memory like kernels typically do, although in principle it could. Since the goal of Dangless is only to provide security benefits, Dangless has no strategy of physical memory management that we would find in normal implementations. In fact, the way this is done ultimately does not matter for Dangless' purposes. Due to these reasons, Dangless delegates the responsibility of actually performing (physical) memory allocation to the memory allocator that was in place before Dangless "hijacked" the memory management function symbols.

Specifically, it uses `dlsym(RTLD_NEXT, "malloc")` to determine the address of the original `malloc()`, etc. functions. Then it simply calls these functions whenever it needs physical memory allocation done: primarily when the user code requests an allocation, but sometimes also for internal purposes, such as for keeping track of available virtual memory regions.

There is a caveat to using `dlsym()`: when `dlsym()` is first called on a thread, it allocates a thread-specific buffer for holding a `struct dl_action_result` object using `calloc()` [9]. This means that without special handling for this case, execution can easily get into an infinite loop:

1. User calls `malloc()`, which is a strong alias of `dangless_malloc()`
2. `dangless_malloc()` defers the physical memory allocation to the underlying allocator by calling `sysmalloc()`

3. `sysmalloc()` does not yet have the address of the original `malloc()` function, so it calls `dlsym()` to get it
4. `dlsym()` notices it's running on this thread for the first time, so it calls `calloc()` to allocate a buffer
5. `calloc()` is a strong alias of `dangless_calloc()`, which calls `syscalloc()` to allocate physical memory
6. `syscalloc()` does not yet have the address of the original `calloc()` function, so it calls `dlsym()`
7. Repeat steps 4-6 forever...

To get around this, `syscalloc()` uses a static buffer of `CONFIG_CALLOC_SPECIAL_BUFSIZE` size for the very first allocation. This allows `dlsym()` to complete and populate the addresses of the original allocation functions, which are used normally for all subsequent calls. The same approach was used by other projects that implement their own memory allocator replacements [17].

Finally, when `sysmalloc()`, etc. returns, we have a completed physical memory allocation. However, what is returned to us is a virtual memory address, while we need a physical one in order to create a second virtual memory mapping. We could perform a pagetable walk to find the corresponding physical memory address, but this is unnecessary, as the mapping provided by Dune is very simple, so it's sufficient to use Dune's `dune_va_to_pa()` function from `libdune/dune.h` that is far cheaper computationally than a page-table walk.

The current implementation of Dangless cannot handle the system allocator returning a (guest) virtual memory region that is backed by non-contiguous (guest) physical memory. This should not normally be a problem, unless Dangless is used together with code that implements the system calls used by memory allocators, such as `brk()` and `mmap()`. This is not a limitation of the design, and can be addressed easily if necessary.

Note that it does not matter whether the host physical memory is contiguous or not: any `mmap()` (or `brk()` for that matter) allocation is mapped into the guest memory contiguously by Dune.

3.3.2 Allocating virtual memory

Given a physical memory address of the user allocation, Dangless needs to allocate the same amount of virtual memory pages that user code will interact with. In the memory layout created by Dune, there's plenty of virtual memory that is not used, nor will ever be used normally due to the size limitations of the various memory regions that Dune enforces.

For simplicity, and in order to minimize the chance of conflicts, by default Dangless upon the first memory allocation request that it can't satisfy due to not having sufficient virtual memory, will take any unused entries from the top-level page table (PML4) and initialize its virtual memory allocator with them marked as available. This behaviour can be disabled using the `AUTO_DEDICATE_PML4ES` option. Users can also dedicate virtual memory to Dangless using the `dangless_dedicate_vmem(void *start, void *end)` function (declared in `dangless_malloc.h`).

The amount of virtual memory available to Dangless has to be very large, as each allocation will use up at least one whole 4 KB page from it. This is done so that during deallocation the page can be marked as unmapped in order to cause any further accesses to it fail. (The most precise level of granularity it is possible to do this on x86-64 based systems today is the 4 KB page.) This means that 1 GB of virtual memory can be used to satisfy $1GB/4KB = 256 * 1024 = 262144$ allocations, assuming each of them is less than 4 KB in size. This is because currently Dangless lacks any mechanisms for detecting that a virtual memory region is no longer referenced, meaning that it will never mark a virtual memory page as available for reuse.

To keep track of the virtual memory available to it, Dangless employs a simple freelist-based span allocator. A freelist is simply a singly linked-list of `vp_span` objects each representing a free span of virtual memory, ordered by their end address:

```
1 struct vp_span {
2     vaddr_t start;
3     vaddr_t end;
4
5     LIST_ENTRY(vp_span) freelist;
6 };
7
8 struct vp_freelist {
9     LIST_HEAD(, vp_span) items;
```

```
10 };
```

(The NetBSD `queue.h` [13] v1.68 is used for the linked list handling macros.)

When virtual memory is needed, the freelist is walked until a `vp_span` object representing a region of sufficient size is found. When one is found, the allocated space is removed from the beginning of the span (by adjusting `start`), and the span is deleted if it is now empty. If no such span is found, the allocation fails.

When an allocation fails, Dangless checks if it's allowed to auto-dedicate virtual memory by consulting the `AUTO_DEDICATE_PML4ES` option. If this option is enabled, and Dangless hasn't yet done so, it will proceed to do this before re-trying the allocation.

Otherwise, Dangless concludes that it's unable to satisfy the user's memory allocation. If `ALLOW_SYSMALLOC_FALLBACK` is enabled (defaults to off), then Dangless proceeds by simply acting as a proxy to the system allocator, and gives up on attempting to protect the allocation. Otherwise, Dangless prints an error message and terminates the application.

Note that in the current, simple implementation of the virtual memory allocator there is only a single freelist, which is sufficient because we do not ever re-use any virtual memory. If we were to add a garbage collector-like solution, then this approach would likely lead to significant fragmentation with a negative performance impact on each allocation. In this situation, a possible enhancement would be to have several independent freelists of different page sizes, similar to common memory allocator designs. Other improvements are also possible: memory allocation is a well-understood problem.

3.3.3 Remapping

Now that Dangless has the physical memory address and a brand new virtual memory address, all that is left to do is mapping the virtual memory to the physical memory by modify the corresponding page table.

In a normal Linux userland application, in order to do this we would have to perform a system call, given that page table manipulation requires ring 0 privileges meaning that it's only available to the kernel. However, thanks to Dune, the process is running inside its own virtualized environment, in which we can act as the kernel, and for instance read control register 3 (`cr3`) containing the (guest) physical address of the page table root.

Having applied the `dune-ix-guestppages.patch` patch to Dune, the host memory pages used to hold the guest's page tables are mapped into the guest virtual memory, allowing us to manipulate them during runtime from inside the guest environment.

It is important to understand that the system allocator will generally place smaller allocations adjacent to each other, and often inside the same page. Consider the earlier example shown on Figure 3.2: X and Y share the same page, with Y overflowing a bit onto the next page that it's sharing with Z and some unallocated memory at the end.

Now, when Dangless is re-mapping X, it has to do so with the entire page that holds X, given that that's the granularity of virtual memory management on x86 (as well as most modern architectures). The remapped page will unavoidably also contain a part of Y, although that address for Y is never going to be presented to the user code. However, the user code could still, by error, access Y through the page dedicated to X, for instance via an out-of-bounds access into X. This will result in the same (undefined) behaviour that the program would display when compiled and ran without Dangless.

In turn, consider the allocation Z on the same diagram: it does not start at the beginning of page, since that is where the second part of Y lives. So when remapping Z (more precisely: the page holding Z), in order to get a pointer to Z itself inside the page, we have to take into account its in-page offset, as shown on the diagram. You can observe the same behaviour for Y.

The allocation Y holds an additional caveat: it *spans* two pages instead of one due to its in-page offset, even though its size is less than the size of a page. As an effect, when remapping Y, we have to allocate two virtual pages.

In the same fashion, it's true in general for all allocations that we re-map, that due to the remapping they will each use at least one whole extra virtual memory page in addition to the one(s) managed by the system allocator. This is the main cause of the virtual memory overhead of using Dangless, as well as its physical memory overhead, although small: the page tables that have to be allocated in order to contain the mappings for the remapped regions.

3.3.4 Deallocations

Whenever Dangless is given a pointer in `realloc()` or `free()`, the first thing it needs to figure out is whether the pointer is a canonical address, i.e. references a virtual memory region managed directly by the system allocator,

or does it point to a remapped virtual memory region managed by Dangless. The former is possible in two circumstances:

1. The allocation happened before Dangless was initialized. This means that the process wasn't yet executing in Dune mode, and therefore Dangless could not have accessed the page tables directly to perform the remapping.

2. The allocation happened when Dangless did not have sufficient virtual memory dedicated to its allocator to be able to perform the remapping. Unless the user manages the virtual memory that Dangless can use for such purpose (e.g. by calling `dangless_dedicate_vmem()`) this means that Dangless has ran out of virtual memory.

If we can determine that the pointer we received was canonical, then Dangless had nothing to do with the original allocation, and therefore now also has nothing to do besides forwarding the call to the system allocator's `free()` function – `sysfree()`.

The challenge then is detecting whether the given pointer was successfully remapped previously, and if so, obtaining the original virtual address that can be passed to `sysfree()`. This is necessary if we don't want to make assumptions about the underlying allocator's implementation details. Typically, memory allocators will not behave correctly if a different virtual memory address is used for deallocation than the one returned during allocation, even if both map to the same physical memory address.

To obtain the original virtual memory address, we make use of Dune's simple memory layout. First, we perform a page walk on the virtual memory address to obtain the corresponding physical address. Then we compare it to physical address that would result in that virtual address being assigned by Dune, by calling `dune_va_to_pa(ptr)`, where `ptr` is the potentially-remapped memory address we're trying to free. If they match, then `ptr` was mapped into virtual memory by Dune itself, meaning that it's not a memory address assigned by Dangless: as discussed earlier, we can just forward the call to `sysfree()` and we're done.

Otherwise we have to determine the canonical pointer belonging to this allocation. In other words, given the physical memory address `PA`, we have to determine the canonical virtual address `VA` such that `dune_va_to_pa(VA)= ↩ PA`. Once again, this is something that Dune's memory layout makes easy

to do, as these conversions are performed using simple arithmetic: we simply invert the logic of `dune_va_to_pa()`. Finally, we can proceed to call `sysfree()` to perform the physical memory deallocation.

What is left to do is invalidating the page table entries for the remapped virtual memory address, to ensure that any later dangling pointer access will fail. Locating the relevant page table entries is done by performing a page-table walk down to the 4K pages, which are then overwritten by an entry that does not have the *present* bit set. Dangless uses the 64-bit value `0xDEAD00` for this purpose, in order to make the invalidated entries easily identifiable. We then flush the Translation Lookaside Buffer (TLB) which is used by the CPU's Memory Management Unit (MMU) to cache the results of page walks to improve performance, to make sure that the old, now invalidated entry values will not accidentally be used again (which would defeat the entire purpose of Dangless).

In order to determine how many of the page table entries we need to invalidate, we have to know how many 4K pages did the allocation span. Recall that Dangless places each allocation on one or more dedicated virtual memory pages that will never be used for anything else. `malloc_usable_size()` is used to get the size of the allocation in bytes. (Of course, this has to be done before calling `sysfree()`.) Note though that determining the number of spanned pages is not as easy as it might seem at the first glance, because the allocation can start anywhere within a memory page. This means that, for instance a 512-byte region can span 1 or 2 pages depending on where it begins within the first page (Figure 3.3.4).

3.3.5 Handling re-allocations

Handling `realloc()` is a combination of a deallocation and an allocation, with a couple of tweaks. I will present the logic here in the conceptual level, as the Dangless-specific technical details of it are the same as with allocations and deallocations already covered previously.

First, the call `realloc(NULL, some_size)` is valid according to the standard and is equivalent to `malloc(some_size)`. Second, we have to be able to handle the unusual case when the original pointer did not originate from Dangless, but directly from the system allocator: this can happen for instance if the original allocation occurred before Dangless was initialized. We deal with both of these cases by treating them the same way as a `malloc()` call: simply allocate some new virtual memory to remap the

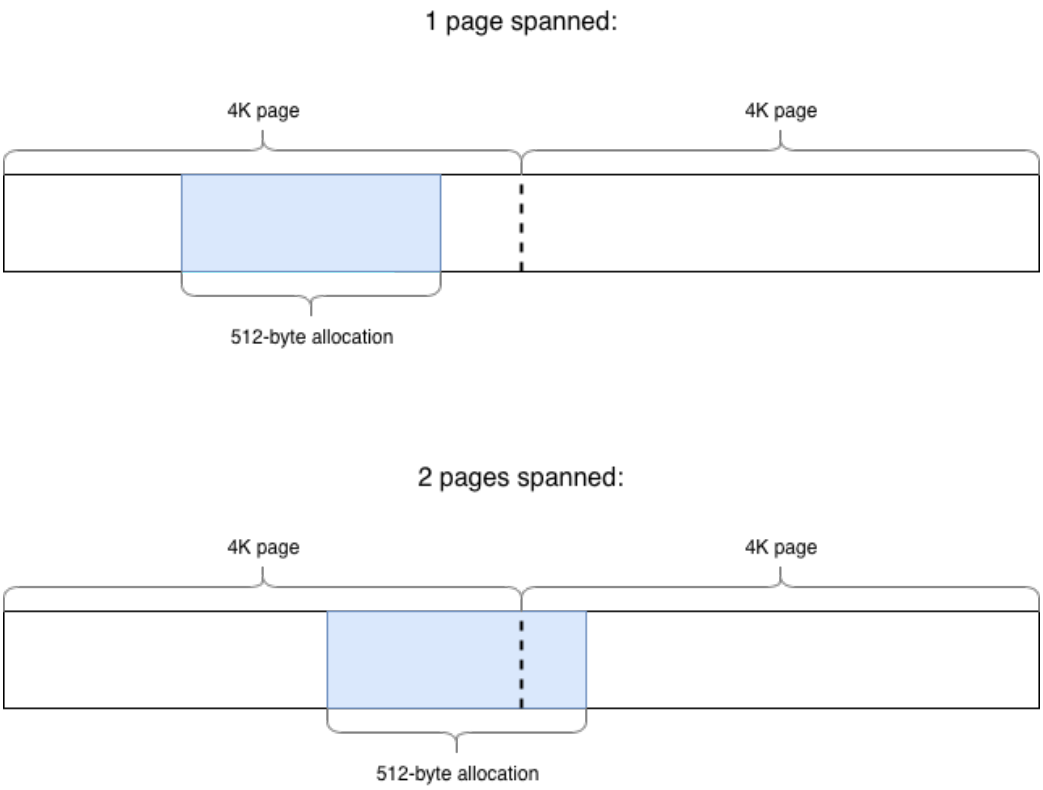

Figure 3.4: A memory allocation can span different number of pages depending on where it begins

system allocator's result to.

Finally, we want to perform the reallocation in-place when possible: that is, re-use the same remapped virtual memory region. There are 3 cases to consider with regards to how the number of pages spanned by the allocation changes:

1. Stays the same: nothing to do.

2. Decreases, i.e. the new size is less than the old size such that the allocation is now held by fewer pages than before: invalidate the pages that were cut off, in the same way as we would handle a deallocation.

3. Increases, i.e. the new size is greater than the old size such that the allocation is now held by more pages than before: allocate a new virtual memory region large enough to fit the new allocation size, and invalidate the entire old region. This could potentially be optimized: if the virtual memory allocator owns the pages that are directly after the old virtual memory region (ensuring that these pages are not in use nor have they been invalidated before), then we could simply grow our virtual memory region in-place.

It's worth noting that if the system allocator was unable to perform the reallocation in-place, that does not mean that we can't perform our work in-place: in this case it's sufficient if we just update the physical memory addresses of the original virtual memory region and flush the corresponding TLB entries. However, this optimization is not currently implemented in Dangless.

3.4 Fixing up vmcalls

3.4.1 The problem

One of the goals of Dune is to remain as simple as possible, and not re-implement most functionality of operating system kernels when not absolutely necessary. This means that most system calls performed by the application running inside Dune are not actually handled by Dune itself, but rather are passed on to the host kernel via the `vmcall` instruction. (`vmcall` is identical to `syscall`, except it exits the virtual environment first.) This

includes tasks as common as I/O operations (such as `printf()` or `fopen()`), or even memory management (such as `mmap()`).

This presents a challenge for Dangless, which can be efficient because it can directly manipulate the page tables inside the virtual environment (guest machine) itself, without having to manipulate the host's page tables (which would only be possible via a system call such as `mprotect()`). However, any virtual memory addresses returned from Dangless will only be valid inside the guest environment, meaning that attempting to pass such a memory address to the host kernel when performing a system call (vmcall) will fail with `EINVAL`.

To demonstrate the problem, first consider the following:

```
1 puts("Hello world!\n");
```

The `puts()` standard library function used here is a comparatively simple wrapper around the `write()` system call, and unlike `printf()`, does not perform any formatting or other manipulation of the string [12]. Recall that in C, strings are represented by null-terminated character arrays, and are typically passed to functions as `const char*`: a (virtual) memory address pointing to the first character of the string. Since in this example, the argument to `puts()` is a string literal, the string data is stored in the executable's data section (such as `.rodata`) without any dynamic memory allocation that would go through Dangless. The result is that the pointer passed to the `write()` vmcall references virtual memory that is mapped by the host machine, so the call will succeed.

```
1  void determineAnswer(char* buffer) {
2      strcpy(buffer, "Fourty-two");
3  }
4
5  char* answerBuffer = malloc(64 * sizeof(char));
6  determineAnswer(answerBuffer);
7
8  puts("The answer is: ");
9  puts(answerBuffer);
10 puts("\n");
11
12 free(answerBuffer);
```

The problem here is that we are passing a `malloc()`-d buffer to `puts()`. Since `malloc()` refers to `dangless_malloc()`, it will perform virtual mem-

ory remapping inside the guest system, yielding a pointer value in `answer` that is valid inside the guest machine, but not in the host machine. (Of course, the data is present in the host physical memory, but is not mapped to host virtual memory.) The `write()` system call when executed by the host, is not going to be amused by this fact, and is going to return the error code `EINVAL`, indicating an invalid argument.

In this case by knowing how Dangless works it is easy to spot the problem. However, often dynamic memory allocation is performed and the resulting pointer is passed to a system call in a way that's not immediately obvious from the user code, such as when done internally by the standard library implementation. The simplest example is probably `printf()`, which can call `malloc()` in some circumstances [8] [1]. I've also already mentioned how `dlsym()` when running for the first time will call `calloc()`.

3.4.2 Intercepting vmcalls

In order to fix this problem, Dangless needs to intercept any system calls that are about to be forwarded to the host kernel. This capability is not provided by default in Dune, so I've implemented it (`dune-ix-vmcallhooks.patch`), allowing Dangless to register a pre- and post-hook function that will be called before and after a `vmcall` instruction is performed by Dune, respectively. In the pre-hook, Dangless can access and modify the system call number, any of the arguments, and even the return address. Similarly, the post-hook exposes the syscall return code.

3.4.3 Determining which arguments to rewrite

The next problem is how to determine what the system call arguments are, and which of them can possibly reference a Dangless-remapped pointer. Note that it is not sufficient to find the pointers among the arguments themselves, as pointers can be nested: the arguments can be pointers to arrays or structures which in turn contain pointers – sometimes, after a few layers of indirection. Examples include the `readv()` and `writev()` system calls, which are used by GNU implementation of the `<iostream>` standard C++ header such as when writing to `std::cout`, i.e. `stdout`:

```
1 ssize_t readv (int fd, const struct iovec *v, int n);
2 ssize_t writev(int fd, const struct iovec *v, int n);
3
```

```
4  struct iovec {
5      void  *iov_base; /* Starting address */
6      size_t iov_len;  /* Number of bytes */
7  };
```

When handling either of these functions, not just the `const struct iovec *v` pointer has to be fixed, but also the `void *iov_base` pointer inside the pointer `struct iovec`.

Another example is the functions `execve()` and `execveat()`:

```
1  int execve(const char *filename, char *const argv[],
       char *const envp[]);
2  int execveat(int dirfd, const char *pathname, char
       *const argv[], char *const envp[], int flags);
```

Besides the `const char *filename` simple pointer, the parameters `char *const argv[]` and `char *const envp[]` are both a null-terminated array of pointers, in which every entry has to be fixed.

Finally, pointers to more complicated structures are also sometimes passed as system call arguments:

```
1  ssize_t recvmsg(int sockfd, struct msghdr *msg, int
       flags);
2
3  struct msghdr {
4      void         *msg_name;       /* optional address */
5      socklen_t     msg_namelen;    /* size of address */
6      struct iovec *msg_iov;        /* scatter/gather
           array */
7      size_t        msg_iovlen;     /* # elements in
           msg_iov */
8      void         *msg_control;    /* ancillary data,
           see below */
9      size_t        msg_controllen; /* ancillary data
           buffer len */
10     int           msg_flags;      /* flags on
           received message */
11 };
```

So, we need some way to determine which arguments of the system call can be a pointer, and identify what data structure it points to in order to find any nested pointers. Essentially, what we need is to be able to tell for a

system call number what arguments it takes and what type they are. For this purpose, I have created a Python module `linux-syscallmd` (source on GitHub [7]) that parses the Linux kernel header file `include/linux/syscalls.h` and exposes system call metadata to user code. The Dangless script at `make/gen_vmcall_fixup_info.py` then uses this information to generate a file containing C code that can be `#include`-d to utilize this data inside Dangless:

```
1  // sources/src/platform/dune/vmcall_fixup_info.h
2
3  enum vmcall_param_fixup_type {
4      VMCALL_PARAM_NONE,
5      VMCALL_PARAM_FLAT_PTR,
6      VMCALL_PARAM_IOVEC,
7      VMCALL_PARAM_PTR_PTR,
8      VMCALL_PARAM_MSGHDR
9  }
10
11 struct vmcall_param_fixup_info {
12     enum vmcall_param_fixup_type fixup_type;
13 };
14
15 struct vmcall_fixup_info {
16     i8 num_params;
17     struct vmcall_param_fixup_info ↩
           params[SYSCALL_MAX_ARGS];
18 };
19
20 // sources/src/platform/dune/vmcall_fixup_info.c
21
22 static const struct vmcall_fixup_info ↩
       g_vmcall_fixup_info_table[] = {
23     // generated by make/gen_vmcall_fixup_info.py
24     #include ↩
           "dangless/build/common/vmcall_fixup_info.inc"
25 };
```

The generated array is indexed by the number of a system call to find its corresponding entry. Then we can iterate through the arguments and act on them based on the `enum vmcall_param_fixup_type` value.

As an example, the entry for the `clone()` system call looks like this:

```
static const struct vmcall_fixup_info s_clone_info = {
    .num_params = 5,
    .params = {
        // unsigned long flags
        [0] = {
            .fixup_type = VMCALL_PARAM_NONE
        },

        // void *child_stack
        [1] = {
            .fixup_type = VMCALL_PARAM_FLAT_PTR
        },

        // int *ptid
        [2] = {
            .fixup_type = VMCALL_PARAM_FLAT_PTR
        },

        // int *ctid
        [3] = {
            .fixup_type = VMCALL_PARAM_FLAT_PTR
        },

        // unsigned long newtls
        [4] = {
            .fixup_type = VMCALL_PARAM_NONE
        }
    }
};
```

3.4.4 Rewriting the pointers

Now that we know which arguments to rewrite or "fix-up", we can use the same logic as `dangless_free()` to get the canonical pointer from a potentially-remapped one (see Section 3.3.4). We then replace the remapped pointer value with the canonical one in the system call arguments.

In case of nested pointers this involves modifying the referenced in-memory data, meaning we cannot simply replace the pointer. This is because the original pointer was a remapped pointer, allocated via Dangless, and will

be deallocated via Dangless. However, due to the nested pointer fix-up, the user code can now potentially access the canonical (non-remapped) pointer, opening the door to dangling pointer errors. Furthermore, should a canonical pointer be passed to `dangless_free()`, it cannot invalidate the remapped memory region as it doesn't know where it might be.

To demonstrate, consider the following code:

```
1  char *first = malloc(32);
2  strcpy(first, "Hello ");
3
4  char *second = malloc(32);
5  strcpy(second, "world!");
6
7  struct iovec iov[2];
8  iov[0].iov_base = first;
9  iov[0].iov_len = strlen(first);
10 iov[1].iov_base = second;
11 iov[1].iov_len = strlen(second);
12
13 writev(STDOUT_FILENO, iov, 2);
```

Due to pointer rewriting the system call succeeds, but afterwards we end up with `iov[0].iov_base != first` and `iov[1].iov_base != second`, as they have been replaced by their canonical counterparts to make the system call succeed on the host kernel.

Later, we deallocate the buffers:

```
1  free(second);
2  free(first);
```

This is fine, since the `first` and `second` variables were not affected by the pointer rewriting, so Dangless correctly invalidates the remapped regions in `dangless_free()`. But then, later:

```
1  fprintf(stderr, "Attempted writev() with '%s' and
       '%s'!\n", iov[0].iov_base, iov[1].iov_base);
```

Here we have an attempted memory access through two dangling pointers. Recall that, due to pointer rewriting, `iov[0].iov_base` and `iov[1].iov_base` are the canonical pointers (as returned by `sysmalloc()`) and so do not point into the remapped region that was invalidated by the earlier `dangless_free()` calls. Therefore, this error will not be caught by Dangless!

To resolve this situation, for every nested pointer fix-up, Dangless records the pointer location (a pointer to the user pointer) as well as the original value stored there (the Dangless-remapped pointer value) in a buffer. After the vmcall returns but before it jumps back into user code, we then go through the records and restore any such rewritten pointer values to their original ones, preventing the user from being exposed to non-remapped pointer values.

3.4.5 Limitations

This approach is limited in that Dangless can only handle system calls, arguments, and argument types that `linux-syscallmd` recognizes when building Dangless.

For instance, `linux-syscallmd` currently does not understand or process preprocessor macros, such as `#if` and `#ifdef` sections. This means that system call signatures not relevant for the current system will also be parsed, leading to conflicting signatures for some system calls, such as `clone()`. `linux-syscallmd` currently does not handle this situation, and will just pick the last occurrence of the same system call in the source file, which may be different than the signature actually used by the kernel. Because of this, Dangless has special handling of the `clone()` system call, but naturally that can't extend to e.g. system calls introduced in the future.

As of writing, I do not know of a better way to approach this. Fixing the present limitation would involve knowing what values were used for each preprocessor macro while building the kernel, and I'm not aware of any way in which the kernel exposes this information.

Another issue is understanding which arguments can hold pointer or nested pointer values. For the vast majority of system calls this is straightforward, as `linux/syscall.h` consistently marks the pointer arguments as `__user *`, and the pointed type is obvious, whether it's `char` or `struct iovec`. But some system calls will interpret the same argument differently depending on the context of the call, such as `ptrace()`. The signature of `ptrade()` is as follows:

```
1 long ptrace(enum __ptrace_request request, pid_t pid,
    void *addr, void *data);
```

Notice that `addr` and `data` are both untyped (`void`) pointers. How they are interpreted differs depending on the value of the `request` argument. Some examples:

- `PTRACE_TRACEME`: both `addr` and `data` are ignored.
- `PTRACE_PEEKTEXT`: `addr` does *not* correspond to pointer in the address space of the caller, but rather, refers to a location in the address space of the target application. The same pointer value may reference a memory region that's unmapped, or worse, mapped for something completely different in the address space of the calling process. As such, Dangless should not touch it. `data` is ignored.
- `PTRACE_POKEDATA`: `addr` refers to a memory address in the target process. `data` might not be a pointer value at all, but rather the word to be copied into the target process' memory.
- `PTRACE_GETREGS`: `data` is an actual pointer in the calling process, while `addr` is ignored.
- `PTRACE_GETREGSET`: `addr` is not a pointer value at all, but rather an enumeration. `data` is an actual pointer to the calling process' memory, but it references a `struct iovec` value, meaning that it will contain nested pointers that also have to be fixed up by Dangless.

`ptrace()` is a special case due to its very specialized nature, and it's unlikely to be used at all in the vast majority of applications that Dangless would be relevant for. Because of this Dangless ignores `ptrace()`, even though it would be possible to cover all of these scenarios.

There may be other system calls that have similar behaviour, although likely not as pathological as `ptrace()`. These are currently not handled in any way by textttlinux-syscallmd nor Dangless. Supporting all of these scenarios would inevitably involve extending Dangless to handle each on a case-by-case basis.

TODO: Probably have a chapter about using Dangless, such as how to build Dangless, what system requirements does it have, and how to make it work on an existing application.

Chapter 4

User guide

Most of this information in less detail is also described in the repository README file.

4.1 System requirements

Most requirements are posed by Dune itself:

- A 64-bit x86 Linux environment
- A relatively recent Intel CPU with VT-x support
- Kernel version of 4.4.0 or older
- Installed kernel headers for the running kernel
- Root (sudo) privileges
- Enabled and sufficient number of hugepages (see below)

The remaining requirements posed by Dangless itself are fairly usual:

- A recent C compiler that supports C11 and the GNU extensions (either GCC or Clang will work)
- Python 3.6.1 or newer
- CMake 3.5.2 or newer

4.1.1 Hugepages

Besides the above, Dune requires some 2 MB hugepages to be available during initialization for setting up its safe stacks. It will also try to use huge pages to acquire memory for the guest's page allocator, although it will gracefully fall back if there are not enough huge pages available.

To make sure that some huge pages remain available, it's recommended to limit or disable transparent hugepages by setting `/sys/kernel/mm/transparent_hugepage/` to `madvise` or `never` (you will need to use `su` if you want to change it).

Then, you can check the number of huge pages available:

```
$ cat /proc/meminfo | grep Huge
AnonHugePages:     49152 kB
HugePages_Total:     512
HugePages_Free:      512
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

In my tests, it appears that at minimum **71** free huge pages are required to satisfy Dune, although it's not quite clear to me as to why: by default for 2 safe stacks of size 2 MB each, we should only need 2 hugepages.

You can dedicate more huge pages by modifying `/proc/sys/vm/nr_hugepages` (again, you'll need to use `su` to do so), or by executing:

```
sudo sysctl -w vm.nr_hugepages=<NUM>
```

... where `<NUM>` should be replaced by the desired number, of course.

When there isn't sufficient number of huge pages available, Dangless will fail while trying to enter Dune mode, and you will see output much like this:

```
dune: failed to mmap() hugepage of size 2097152 for safe stack 0
dune: setup_safe_stack() failed
dune: create_percpu() failed
Dangless: failed to enter Dune mode: Cannot allocate memory
```

4.2 Building and configuring Dangless

The full Dangless source code is available on GitHub at https://github.com/shdnx/dangless-malloc. After cloning, you will have to start by setting up its dependencies (such as Dune) which are registered as git submodules in the `vendor` folder:

```
git submodule init
git submodule update
```

Then we have to apply the Dune patches as described in Section 2.4 and build it:

```
cd vendor/dune-ix

# patch dune, so that the physical page metadata is accessible inside
git apply ../dune-ix-guestppages.patch

# patch dune, so that we can register a prehook function to run before
git apply ../dune-ix-vmcallprehook.patch

# patch dune, so that it doesn't kill the process with SIGTERM when ha
git apply ../dune-ix-nosigterm.patch

# need sudo, because it's building a kernel module
sudo make
```

Now configure and build Dangless using CMake:

```
cd ../../sources

# you can also choose to build to a different directory
mkdir build
cd build

# you can specify your configuration options here, or e.g. use ninja (
cmake -D CMAKE_BUILD_TYPE=Debug -D OVERRIDE_SYMBOLS=ON -D REGISTER_PRE
make
```

You should be able to see `libdangless_malloc.a` and `dangless_user.make` afterwards in the build directory.

You can see what configuration options were used to build Dangless by listing the CMake cache:

```
$ cmake -LH
-- Cache values
// Whether to allow dangless to gracefully handle running out of virtu
ALLOW_SYSMALLOC_FALLBACK:BOOL=ON

// Whether Dangless should automatically dedicate any unused PML4 page
AUTODEDICATE_PML4ES:BOOL=ON

// Choose the type of build, options are: None(CMAKE_CXX_FLAGS or CMAK
CMAKE_BUILD_TYPE:STRING=Debug

// Install path prefix, prepended onto install directories.
CMAKE_INSTALL_PREFIX:PATH=/usr/local

// Whether to collect statistics during runtime about Dangless usage.
COLLECT_STATISTICS:BOOL=OFF

// Debug mode for dangless_malloc.c
DEBUG_DGLMALLOC:BOOL=OFF

// Debug mode for vmcall_fixup.c
DEBUG_DUNE_VMCALL_FIXUP:BOOL=OFF
...
```

You can also use a CMake GUI such CCMake [2], or check the main CMake file (`sources/CMakeLists.txt`) for the list of available configuration options, their description and default values.

4.3 API overview

TODO: A lot of this should be moved to ch3 section 1: initialization

Dangless is a Linux static library `libdangless.a` that can be linked to any application during build time. It defines a set of functions for allocating and deallocating memory:

```
1 // sources/include/dangless/dangless_malloc.h
2
3 void *dangless_malloc(size_t sz) ↩
    __attribute__((malloc));
4 void *dangless_calloc(size_t num, size_t size) ↩
    __attribute__((malloc));
5 void *dangless_realloc(void *p, size_t new_size);
6 int dangless_posix_memalign(void **pp, size_t align, ↩
    size_t size);
7 void dangless_free(void *p);
```

These functions have the exact same signature and behaviour as their standard counterparts `malloc()`, `calloc()`, and `free()`. In fact, because the GNU C Library defines these standard functions as weak symbols [10], Dangless provides an option (`OVERRIDE_SYMBOLS`) to override the symbols with its own implementation, enabling the user code to perform memory management without even being aware that it's using Dangless in the background.

Besides the above functions, Dangless defines a few more functions, out of which the following two are important.

```
1 void dangless_init(void);
```

Initializes Dangless as described in Section 3.2. Whether or not this function is called automatically during application start-up is controlled by the `REGISTER_PREINIT` option, defaulting to On.

```
1 int dangless_dedicate_vmem(void *start, void *end);
```

Dedicates a memory region to Dangless' virtual memory allocator, as described in Section refsec:dangless-alloc-virtmem. Whether or not any dedication happens automatically is controlled by the `AUTO_DEDICATE_PML4ES` option.

TODO: Section on adding Dangless to an existing application
TODO: Maybe a reference guide to all the configuration options available?

Chapter 5

Evaluation

TODO: This needs a lot more work.

5.1 Performance on SPEC 2006

TODO: ...

5.2 Limitations and improvement opportunities

- virtual remapping should be able to handle getting a virtual memory region from the system allocator that's not backed by a contiguous physical memory region - realloc() should try to do an in-place reallocation with the virtual remapped region; if the system allocator couldn't perform it in-place, then we simply have to adjust our existing region by updating each PTE's PA to the new physical memory region. - realloc() should try to grow the remapped virtual memory region in-place if the following pages are owned by the virtual memory allocator (which ensures that these are not in use by anything else, nor have they been invalidated before) - multi-threading support - clone() support - smarter virtual memory auto-dedication logic - on >2 MB, >1 GB allocations, hugepages could be used - garbage collector

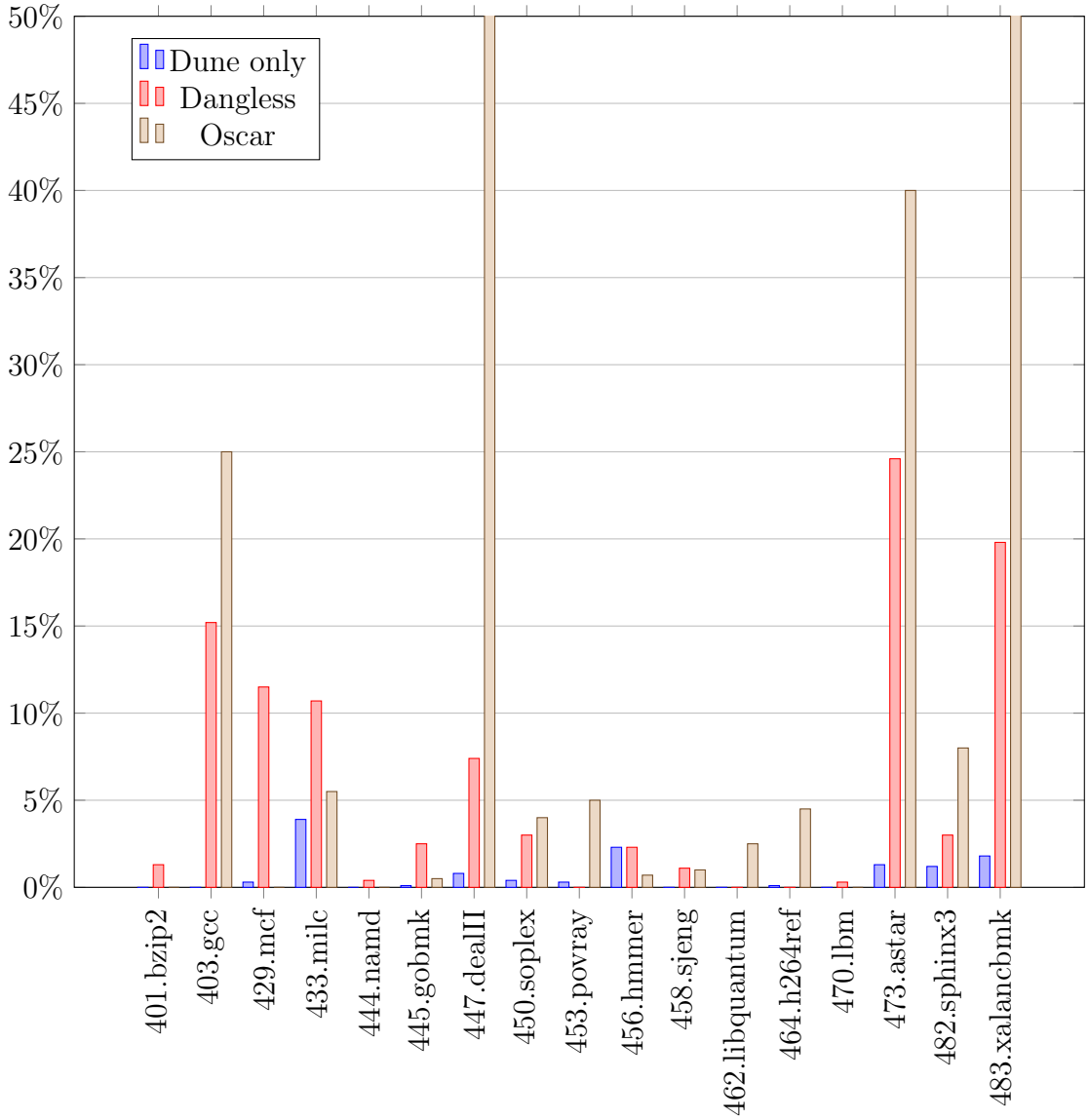

Figure 5.1: Dune vs Dangless vs Oscar: performance overhead

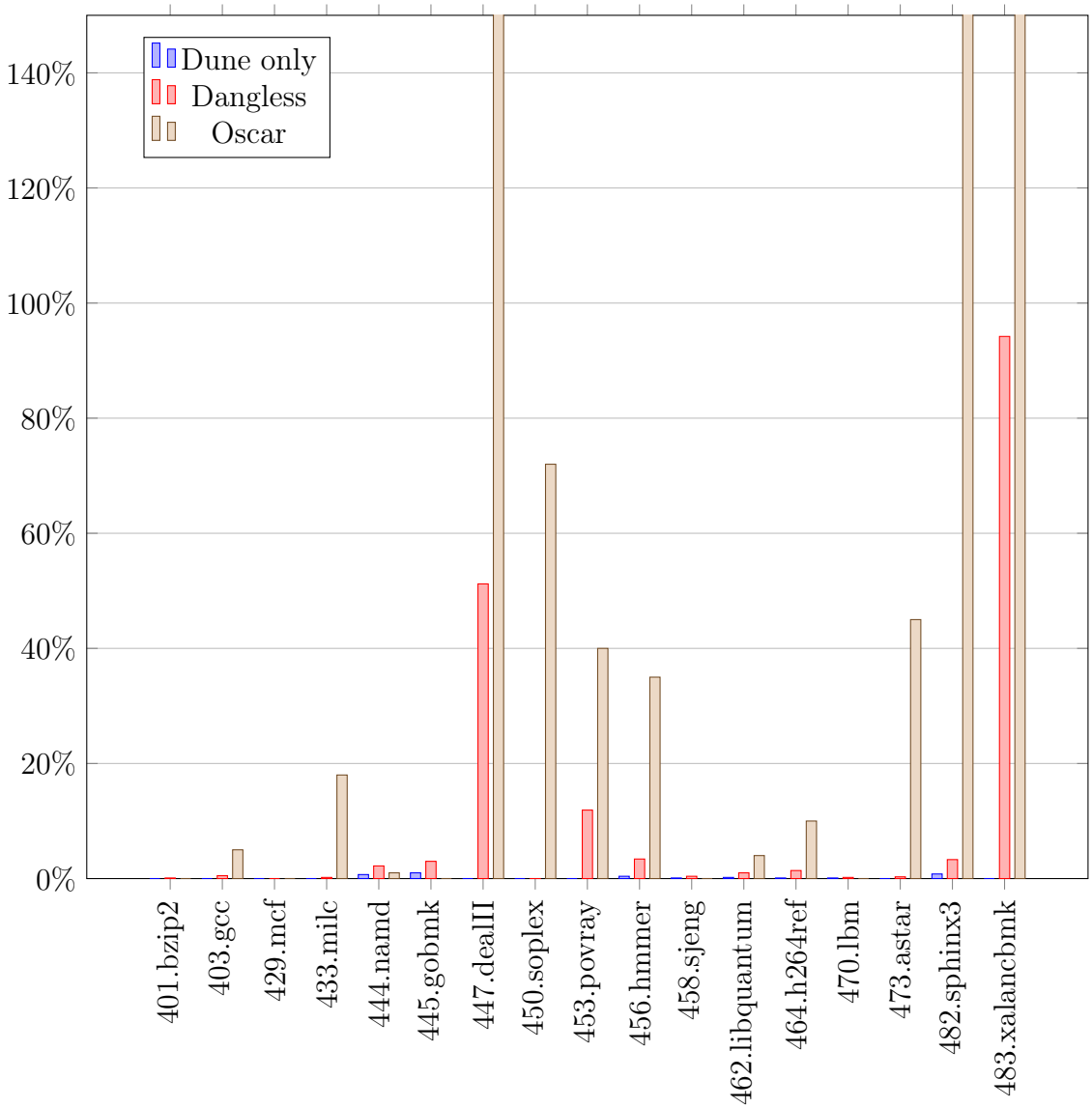

Figure 5.2: Dune vs Dangless vs Oscar: memory overhead

Bibliography

[1] 0ctf 2017 - easiestprintf (pwn 150). https://blog.dragonsector.pl/2017/03/0ctf-2017-easiestprintf-pwn-150.html. Accessed 2019-07-09.

[2] Ccmake - a graphical user interface for cmake. https://cmake.org/cmake/help/latest/manual/ccmake.1.html. Accessed 2019-09-10.

[3] Docker.com website. https://www.docker.com. Accessed 2019-09-26.

[4] Dune project official website. http://dune.scs.stanford.edu. Accessed 2019-07-31.

[5] Dune source code, forked by the ix-project. https://github.com/ix-project/dune. Accessed 2019-07-31.

[6] Dune source code, original repository. https://github.com/project-dune/dune. Accessed 2019-07-31.

[7] Github source for linux-syscallmd. https://github.com/shdnx/linux-syscallmd. Accessed 2019-07-09.

[8] GNU C library - printf() can use malloc(). https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html#Hooks-for-Malloc. Accessed 2019-07-08.

[9] GNU C library source - dlsym() calls calloc(). https://github.com/lattera/glibc/blob/59ba27a63ada3f46b71ec99a314dfac5a38ad6d2/dlfcn/dlerror.c#L141. Accessed 2019-06-29.

[10] GNU C library source - weak definitions of memory management functions. https://code.woboq.org/userspace/glibc/malloc/malloc.c.html#__calloc. Accessed 2019-06-29.

[11] Google thread-caching memory allocator (tcmalloc). https://github.com/gperftools/gperftools. Accessed 2019-10-22.

[12] Hello world analysis. http://osteras.info/personal/2013/10/11/hello-world-analysis.html. Accessed 2019-07-08.

[13] NetBSD queue.h header file - reference. https://netbsd.gw.com/cgi-bin/man-cgi?queue. Accessed 2019-06-30.

[14] Rumpkernels documentation. https://github.com/rumpkernel/wiki/wiki/Repo. Accessed 2019-09-26.

[15] Rumpkernels.com website. http://rumpkernel.org. Accessed 2019-09-26.

[16] Unikernels.org, list of projects. http://unikernel.org/projects/. Accessed 2019-09-26.

[17] The universal elite game trainer for cli (linux game trainer research project) - hooking memory allocation functions. https://github.com/ugtrain/ugtrain/blob/d9519a15f4363cee48bb3c270f6050181c5aa305/src/linuxhooking/memhooks.c#L153. Accessed 2019-06-29.

[18] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[19] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, 2012. USENIX.

[20] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

[21] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 269–280. IEEE, 2006.

[22] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. *ACM SIGPLAN Notices*, 38(7):69–80, 2003.

[23] Robert Gawlik and Thorsten Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 396–405. ACM, 2014.

[24] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *NDSS*, 2014.

[25] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4):461–472, 2013.