Empir Software Eng (2016) 21:753–777
DOI 10.1007/s10664-015-9362-z

On the detection of custom memory allocators in C binaries

Xi Chen · Asia Slowinska · Herbert Bos

Published online: 29 March 2015

Abstract Many reverse engineering techniques for data structures rely on the knowledge of memory allocation routines. Typically, they interpose on the system's `malloc` and `free` functions, and track each chunk of memory thus allocated as a data structure. However, many performance-critical applications implement their own custom memory allocators. Examples include webservers, database management systems, and compilers like gcc and clang. As a result, current binary analysis techniques for tracking data structures fail on such binaries. We present MemBrush, a new tool to detect memory allocation and deallocation functions in stripped binaries with high accuracy. We evaluated the technique on a large number of real world applications that use custom memory allocators. We demonstrate that MemBrush can detect allocators/deallocators with a high accuracy which is 52 out of 59 for allocators, and 29 out of 31 for deallocators in SPECINT 2006. As we show, we can furnish existing reverse engineering tools with detailed information about the memory management API, and as a result perform an analysis of the actual application specific data structures designed by the programmer. Our system uses dynamic analysis and detects memory allocation and deallocation routines by searching for functions that comply with a set of generic characteristics of allocators and deallocators.

Keywords Custom memory allocation · Dynamic binary analysis

Communicated by: Romain Robbes, Massimiliano Di Penta and Rocco Oliveto

X. Chen · A. Slowinska (✉) · H. Bos
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
e-mail: asia@few.vu.nl

X. Chen
e-mail: x.chen@vu.nl

H. Bos
e-mail: herbertb@few.vu.nl

1 Introduction

Many reverse engineering techniques for data structures depend on the analysis of memory allocated on the heap (Jung and Clark 2009; Slowinska et al. 2011; Lin et al. 2010; Balakrishnan and Reps 2004; Reps and Balakrishnan 2008). Typically, they interpose on the system's `malloc` and `free` functions, and track each chunk of memory thus allocated as data structure. Doing so is well and good for applications that use the standard memory allocation and de-allocation functions, but unfortunately many larger and performance-critical programs do not. Instead, they implement their own custom memory managers, typically designed for efficiency. Well-known examples of such applications include the Apache webserver, the PostgreSQL database management system, the gcc optimizing compiler, and Dropbox, among many others. As reverse engineers do not have access to source, the precise memory allocation and deallocation functions are not known. As a result, all techniques that build on the interposition of such functions fail.

The problem is that they only see the allocations by the system's general purpose allocators, but not the subdivision of these allocations into smaller fragments by the application's custom memory allocator (CMA). Unfortunately, the larger chunks that are visible to the reverse engineer serve merely as a pool for the more relevant allocations of the actual data structures. Phrased differently, the large chunks themselves are mostly meaningless, while the smaller fragments are reused by various functions and system calls. Missing them makes it exceedingly difficult to observe any meaningful access patterns and detect the objects designed by the programmer.

In this paper, we describe a set of techniques to detect memory allocation and deallocation functions in stripped C binaries with high accuracy. We implemented the techniques in a tool called MemBrush and evaluated it on a large number of custom memory allocators. We also evaluated our techniques on several C++ binaries, but while the initial results look promising, this was not the focus of our work and needs further evaluation in the future. MemBrush is an off-line, heuristics-based tool which uses dynamic taint analysis. By examining the heuristics during a binary's execution, MemBrush will first extract a set of possible allocator/deallocator candidates. Later these candidates are validated by replaying them. In the evaluation section we show that MemBrush can detect allocator/deallocators with a high accuracy which is 52 out of 59 for allocators, and 29 out of 31 for deallocators in SPECINT 2006. While the memory overhead is considerable, due for instance to the (taint) information we store during runtime, we demonstrate that the method is practical by applying it to real-world programs.

The main goal of MemBrush is to furnish existing reverse engineering tools, disassemblers and debuggers with detailed information about the memory management API implemented by a CMA. Knowing the CMA's allocation, deallocation, and reallocation routines, allows us to interpose on them and take the memory analysis techniques for general-purpose allocators and reuse them in applications that 'roll their own'. To demonstrate its usefulness, we use MemBrush to support an existing reverse engineering tool called Howard (Slowinska et al. 2011). Howard is a tool to extract low-level data structures from a stripped binary. Thanks to MemBrush, Howard was able to extract heap structures that it would otherwise not have been able to detect. As a result, we can perform an analysis of the application specific data structures designed by the programmer.

In addition, researchers have shown that knowledge of memory allocation and deallocation routines is useful for retrofitting security in existing binaries—for instance to protect against memory corruption (valgrind, http://valgrind.org; Hastings and Joyce 1992; Dhurjati and Adve 2006; Caballero et al. 2012; Perence, B: Electric Fence, http://perens.com/FreeSoftware/ElectricFence; Slowinska et al. 2012). Currently, these security measures are powerless if the

application uses CMAs. Again, with MemBrush these existing techniques should simply work, regardless of the memory allocator.

High-level Overview The key observation behind MemBrush is that memory allocation functions have characteristics that set them apart from other routines. For instance, a `malloc`-like routine will return a heap address and `malloc`'s clients will use pointers derived from that address to access memory. MemBrush checks these characteristics at runtime taking care to filter out routines that exhibit similar behavior (like wrappers, iterators, etc.) as much as possible.

Like all dynamic analysis, MemBrush's results depend on the code that is covered at runtime. Specifically, it will not find CMA routines in code that never executes. This paper is not about code coverage techniques. Rather, we use test suites to cover as much of the application as possible. Fortunately, applications that employ CMAs, typically use the allocation routines frequently—after all, that is why they have them in the first place. Thus, finding inputs that exercise the CMA code is not very difficult, and MemBrush correctly identified almost 90% of all the CMA routines in all the applications we tested.

In summary, MemBrush is able to unearth most CMA routines in arbitrary (gcc-generated) binaries with a high degree of precision. While it is too early to claim that the problem of CMA identification is solved, MemBrush advances the state of the art significantly. For instance, we managed to accurately analyze the complex CMA systems used by the Nginx webserver, or the ProFTPd file server.

We implemented all dynamic analysis techniques using Intel's Pin dynamic binary instrumentation framework (Intel 2011). Our current implementation works with x86 C (and some C++) binaries on Linux generated by the gcc optimising compiler, but the approach is not specific to any particular OS or compiler.

Outline The remainder of the paper is organized as follows. In Section 2, we start with a background information and we discuss the essential characteristics of CMAs that lay the foundation of our detection algorithm. Sections 3-7 describe the details of MemBrush's technique to detect the CMA routines. We evaluate MemBrush in Section 8, and discuss its limitations in Section 9. Finally, we talk about the related work in Section 10, and we conclude in Section 11.

This paper is an extended version of our WCRE 2013 publication (Chen et al. 2013).

2 Background and Observations

Programmers incorporate custom memory allocators into their applications to improve performance, and in the case of region-based allocators – to reduce the programming burden and eliminate a source of memory leaks.

Under the hood, CMAs use general-purpose memory allocation routines, such as `malloc` and `mmap`, to allocate large buffers, and then define their own custom functions to allocate these buffers into smaller ones. Applications use the resulting blocks to store structured data items such as arrays, structs, or C++ objects. When an application releases a block, a CMA does not immediately return the memory to the general-purpose allocator. Instead, it may serve it on a future request by the application and defer the real deallocation (for instance, until the time that no more requests are to be expected from the application). As we explain in Section 2.2, when detecting CMAs, it is important to ignore wrappers—functions that only perform certain tests, e.g., for null pointers, before returning an object already obtained from a general-purpose or custom allocator.

Rather than aiming for a particular custom memory allocator, the objective of MemBrush is to detect *any* CMA. In Section 2.1, we therefore introduce popular types of custom memory allocators. Then, in Section 2.2, we list the essential characteristics of CMAs that lay the foundation for our detection algorithm described in Sections 3–6.

2.1 A Taxonomy of CMAs

Since comprehensive overviews of CMAs can be found in surveys by Wilson et al. (1995) and Berger et al. (2002), we limit ourselves to a summary of the approaches in this section. Like Berger et al. (2002), we distinguish the following five categories:

Per-class Allocators (also known as *slab* allocators). A per-class allocator retains memory to contain data objects of the same type (or size). It implements the same API as a general-purpose memory allocator (`malloc`/`free`), i.e., it supports allocation and deletion of individual objects. Slab allocators are widely used by many Unix and Unix-like operating systems including FreeBSD (The FreeBSD Project: FreeBSD Kernel Developers Manual. ZONE(9), http://www.freebsd.org/cgi/man.cgi?query=uma) ("zones") and Linux (Jones 2007).

Regions (also known as *arenas*, *groups*, and *zones* (Ross 1967; Hanson 1990)). Each object allocated by an application is assigned to a region, i.e., a large chunk of memory. Programmers can only deallocate all objects from a region at once – individual deallocations are not possible. This limitation facilitates allocation and deallocation of memory with a low performance overhead, at the cost of an increased memory usage. Example applications using regions include Apache (The Apache Software Foundation: Developing modules for the Apache HTTP Server 2.4, http://httpd.apache.org/docs/2.4/developer/modguide.html) (which refers to them as "pools"), PostgreSQL (The PostgreSQL Global Development Group: PostgreSQL 9.2.4 Documentation. Section 43.3. Memory Management, http://www.postgresql.org/docs/9.2/static/spi-memory.html) (which refers to them as "memory contexts"), and Nginx (nginx: nginx documentation, http://nginx.org/en/docs/).

Obstacks An obstack (The GNU C library. Obstacks, http://www.gnu.org/software/libc/manual/html_node/Obstacks.html) is a more generic version of a region. It contains a stack of objects, within which an individual object is freed along with everything allocated in this obstack since the creation of the object. An example application using obstacks is the gcc compiler (GNU libiberty: Obstacks, http://gcc.gnu.org/onlinedocs/libiberty/Obstacks.html).

Custom Patterns This category includes all allocators that implement the same API as a general-purpose memory allocator (`malloc`/`free`), but are tailored to the needs of a particular application. For example, one of the allocators used by Nginx falls into this category.

Hybrid approaches The research community has proposed various approaches to provide e.g., high-speed allocation and cache-level locality. For instance, *reaps* (Berger et al. 2002) are a combination of regions and general-purpose allocators that extend region semantics with individual object deletion.

2.2 Essential Characteristics of CMAs

Having looked at the different categories of CMA, we now summarize their common features. It is important to emphasize that these features aim to capture the fundamental behavior of CMAs and not some implementation artifact of specific variants. For instance, all of the eight CMA implementations that we analyze in Section 8 exhibit these characteristics. As we will see in

Sections 3–6, these characteristics form the basis for our detection algorithm. We will discuss allocation, deallocation, and reallocation routines in turn. In a generic sense, we will refer to these custom functions as c_malloc, c_free, and c_realloc, respectively.

Allocation Routines c_malloc functions subdivide large memory chunks obtained from a general-purpose allocator into small ones, and serve the small ones upon the application's requests. We make the following basic observations about a custom allocator's behavior:

1. Normally, a c_malloc function returns a pointer `p` that references a heap memory region. As we discuss below, in some cases this rule should be relaxed. E.g., a c_malloc does not need to literally *return* `p`, but it might pass it through an outgoing argument (*return a pointer*).
2. Applications use `p` or a pointer derived from `p`, e.g., `(p+offset)`, to write to memory. Here also, we expect some deviations from such behavior. For instance, it is possible that the occasional application allocates a memory block that it does not use. However, this should be the exception, rather than the rule. If the application (almost) never writes to memory referenced by `p`, then the function that returns it does not serve as an allocator (*write to the allocated memory*).
3. Unless the c_malloc function initializes memory chunks prior to returning them, the application should write to these chunks before reading them (*no read before write*).
4. A c_malloc should not return the same object twice until that chunk is released first with a call to a c_free function (*no aliasing*).
5. Since we aim to exclude wrapper functions, we require that a c_malloc not only checks and passes a pointer obtained from another internal function, but also performs some computations to derive the address of a newly allocated object (*no wrappers*).

As we will see in Sections 4 and 8, these features accurately capture the behaviour of allocation functions. Observe that, A1, A4 and A5 refer to the internal characteristics of an allocator, while A2 and A3 specify how an application interacts with it.

A1 captures the basic characteristic of a memory allocation routine: once the allocator allocates a memory buffer, the application needs to learn about its location. MemBrush assumes that the allocator returns a pointer that references the heap memory region. As we demonstrate with our experiments, this assumption usually holds in practice, and is sufficient to deal with real world applications (refer to Section 8). However, in theory, a c_malloc does not need to literally *return* a pointer, but it might pass it through an outgoing argument. Similarly, it does not need to return a *proper C/C++ pointer*, but it might provide an offset that the application subsequently adds to the address of a buffer to compute the pointer. Since we have never encountered the latter scenario in practice, we consider it to be purely hypothetical, and we mention it for the sake of completeness only. However, as we discuss in Section 9, MemBrush could be extended to also cater to the two aforementioned corner cases.

Unlike, say, sorting or encryption routines, an allocation function is expected to return a different result every time it is called. Indeed, only when a memory chunk is released, a c_malloc can allocate it again. A4 captures this observation.

Features A1, A4 and A5 alone let MemBrush identify c_malloc routines, but they may also lead to an overapproximation. A prominent example of a possible misclassification would be iterators. When traversing a container, an iterator returns pointers to elements of the container, satisfying A1, and just like an allocator it provides a sequence of distinct memory objects, satisfying A4. Only by observing how the application uses the memory buffers – with rules A2 and A3 – can MemBrush distinguish an allocator from an iterator. Indeed, the objects returned by an iterator are usually initialized already, so they do not need to be updated first (A3), and can be only read (A2).

Deallocation Routines When an application frees a chunk of memory obtained from a c_malloc routine, c_free reclaims the chunk, so that it can be served again on future requests. The algorithms in Section 5 are based on the following characteristics of deallocators:

(D1) CMAs keep track of which parts of memory are in use, and which parts are free. They record the locations and sizes of free blocks in some kind of *metadata*, which may be a list, a tree, a bitmap or another data structure. Thus, a c_free function accesses the metadata that is also maintained by a c_malloc function (*metadata sharing*).
(D2) When a c_free releases a memory region, the application should not access it anymore unless there is a bug (and we assume bugs are rare) *(no use-after-free)*.
(D3) When a c_free releases a memory object, a c_malloc may return it on future application's requests. (Observe that this feature is strongly related to A4.)
(D4) Since we aim to exclude wrapper and internal helper functions, we select the outermost function that shares the metadata with a c_malloc. The intuition is that if a function does use the metadata, it should be considered a part of the CMA. If both a caller and its callee share the metadata with a c_malloc, MemBrush selects the outer one. Since each of them does use the metadata, both should be considered a part of the CMA (*understand the interface functions*).

Observe that the above characteristics of deallocation routines reflect the interaction between c_malloc and c_free functions. As we will see in Section 5, to detect c_free routines, MemBrush searches for functions that it can couple with the already identified c_malloc routines.

Reallocation Routines Finally, c_realloc functions allow applications to modify the size of a previously allocated memory block. To guarantee that the new block is contiguous in memory, c_realloc may have to relocate it elsewhere. We consider the following features of c_realloc routines:

(R1) Like c_malloc in A1, c_realloc functions return a pointer p to a heap memory region (*return a pointer*).
(R2) Like property (D1) for deallocation functions (D1), c_realloc functions also access the metadata used by a c_malloc (*metadata sharing*).
(R3) As in (A2) and (A3), applications use p or a pointer derived from p to write to memory, and write to the allocated memory before reading it (*write to the allocated memory*).
(R4) Once a c_realloc modifies the size of a buffer, future repetitions of the same request do not require any action, so also do not relocate it (idempotence).
(R5) A c_realloc preserves the contents of a memory block up to the lesser of the new and old sizes. Thus, if the block is relocated, a c_realloc copies the old contents to the new location (*copy on relocation*).

When R5 finds that a c_realloc function relocates a buffer, we additionally verify R6–R7 below:

(R6) As a c_realloc combines a c_malloc and a c_free, it also releases a memory object, and the application should not access it anymore (as in D2).
(R7) Like c_free in D3, if a c_realloc releases a memory object, a c_malloc might return it on future application's requests.

Even though the above features reflect the expected behavior of CMAs, we emphasize that MemBrush allows for occasional deviations. For example, it is possible that an application has a use-after-free bug, and uses a chunk of memory even though it has been deallocated already, violating D2. Also, even though an application should not read uninitialized memory (a breach of

A3), we might occasionally observe such behavior. As we will see later, in Section 8, we permit such exceptions as long as they are *rare*. However, in practice, we did not come across them.

3 A Bird's Eye View of MemBrush

We now discuss the CMA detection procedure. MemBrush consists of *instrumentation* modules and *detection* modules (see Fig. 1). The instrumentation modules, ③-⑥, provide support (such as dynamic information flow tracking) for the detection modules, while the detection modules,⑦-⑩, search for the CMA routines. In this section, we briefly introduce the various components, and in the next four sections, we explain the detection modules in detail.

In this paper, we search for CMA routines that operate on top of the `mmap`/`brk` system calls or the `libc`library (i.e., that internally call `malloc`/`free`) to allocate large chunks of memory. However, we can configure MemBrush to detect the Doug Lea allocator (Doug Lea: A Memory Allocator, http://g.oswego.edu/dl/html/malloc.html) used by the GNU C library as well. To do so, we would simply choose not to search for allocators based on `malloc`, but solely on `mmap`/`brk`, and we do not assume any knowledge about the `libc` semantics.

We implemented MemBrush using Intel's Pin dynamic binary instrumentation framework (Intel 2011). Pin provides a rich API to monitor context information, e.g., register or memory contents, on program instructions, functions and system calls.

The main components of Fig. 1 are the following:

Inputs: ①② The main input to MemBrush is a (possibly) stripped `x86` binary ① and its inputs ②. For this paper, we used existing test suites to cover as much of the application as possible. If needed, we can also employ a code coverage tool for binaries like S2E (Chipounov et al. 2011).

Call stack tracking: ③ To analyze if a function's behavior is characteristic for a CMA routine, membrush monitors the function and its callees. For that, it keeps track of the context in the function call stack.

Our implementation follows Slowinska et al. (2011). In a nutshell, as membrush runs the application in a binary instrumentation framework, it can dynamically observe `call` and `ret` instructions, and the current position of the runtime stack. A complicating factor is that sometimes `call` is used not to invoke a real function, but only as part of a `call`/`pop` sequence to read the value of the instruction pointer. Similarly, not every `ret` has a corresponding `call` instruction.

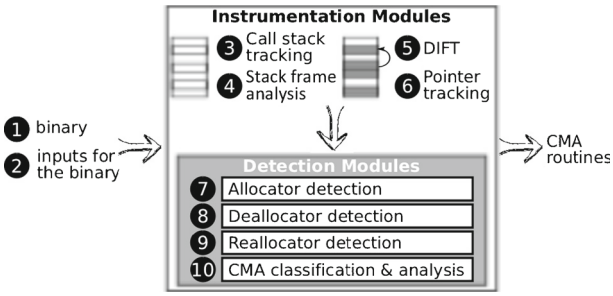


Fig. 1 MemBrush: high-level overview

For the purpose of our analysis, we define a *function* as the target of a `call` instruction which returns with a `ret` instruction. Values of the stack pointer at the time of the call and at the time of the return match, giving a simple criterion for detecting uncoupled `call` and `ret` instructions.

Partial reconstruction of physical stack frame: ④ To detect CMA routines, membrush needs to partially reconstruct the physical stack frame of a function. Specifically, it identifies all the stack-based procedure arguments. Like Slowinska et al. (2011), our current implementation is based on dynamic analysis. In a nutshell, we monitor how a function calculates pointers to access stack variables pushed by its caller. If necessary, we can extend it with a static analysis presented by ElWazeer et al. (2013).

Additionally, to determine a first set of candidates for c_malloc and c_realloc routines, membrush monitors the return value of each executed function, and checks if it is a pointer dereferencing a heap memory region. Since in `gcc` generated binaries, 32-bit return values are normally passed using the `EAX` register, membrush implements this policy as well[1].

Dynamic information flow tracking (DIFT): ⑤ As we shall explain later, the detection modules rely on dynamic information flow tracking (for data flow analysis). Our tracker is an extended version of `libdft` (Kemerlis et al. 2012). Like most other DIFT engines (Portokalidis et al. 2006), we propagate information on direct flows only: we copy tags on data move operations, `or` them on ALU operations, and clean tags on common `ia32` idioms to zero memory, such as `xor $eax, $eax`. We do not propagate any information on indirect data flows, such as conditional statements.

Pointer tracking: ⑥ membrush monitors how the application uses pointers returned by the c_malloc and c_realloc candidates. To this end, the pointer tracking module tracks how pointers to heap memory derive from other pointers, and where they are stored. Our implementation is based on (Slowinska et al. 2011) which extends the generic DIFT module ⑤ with pointer propagation rules.

Detection modules: ⑦⑧⑨⑩ The detection modules identify the actual CMA API: c_malloc, c_free, and c_realloc. membrush's algorithms check for the characteristic features discussed in Section 2.2, and search for the routines in turn. In the first step ⑦, membrush determines c_malloc routines. Then ⑧, it tries to find c_free functions that can be coupled with the already detected allocation functions. In the last step ⑨, it identifies c_realloc routines. Finally ⑩, we perform an additional analysis of the detected CMA routines to classify the CMA according to the taxonomy from Section 2.1.

The current implementation of membrush handles `gcc`-generated binaries. However, the characteristics of the CMA routines that membrush relies on are generic and compiler independent. In principle, we could port membrush to other platforms or use it to analyze binaries compiled with different compilers. The implementation would differ though, e.g., depending on the calling convention, we would need to adjust our analysis of the arguments passed to functions or the calling context. We leave that as a future work.

4 Custom Allocator Detection

To detect c_malloc routines, membrush searches for functions that match the requirements A1-A5 from Section 2.2. Figure 2 represents the procedure as a linear pipeline, in which each stage progressively filters out functions that do not comply with the corresponding features.

[1]While return values proved sufficient in practice, we could extend the technique to handle results returned in parameters also.

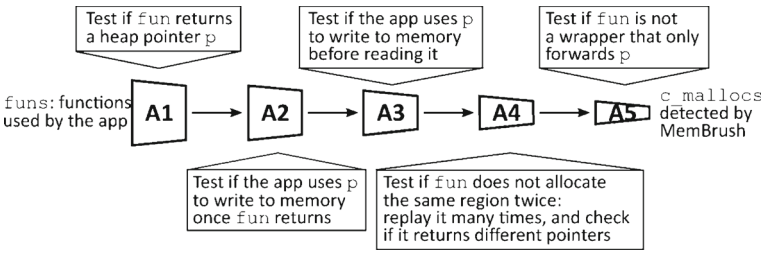


Fig. 2 Detection of c_malloc functions

membrush starts by identifying a crude set of c_malloc candidates, i.e., functions that return pointers referencing heap memory regions (A1). As we said before, we currently focus on functions that pass the return value using the EAX register. While the application executes, membrush uses the pointer tracking module ⑥to track all pointers derived from the addresses returned by the general-purpose memory allocators, such as `mmap` or `malloc`. This way, it also follows a custom allocator calculating the locations of allocated objects. membrush monitors the return values of all functions invoked at runtime, and selects the ones that return either a tracked pointer or a single constant that might indicate an error, e.g., NULL.

To verify A2, membrush uses dynamic taint analysis to track all pointers derived from the return value of each c_malloc candidate, and monitors if they are used to write to memory. More specically, when a c_malloc candidate returns, MemBrush assigns a unique color to the returned value. The taint analysis engine keeps propagating the color until a memory access occurs. At that point, membrush checks the color of the dereferenced pointer, to identify the source c_malloc candidate.

To assess A3, membrush additionally examines if the application uses these pointers to write to a memory location before reading it. Unless the allocator initializes the memory itself, the presence of such read-before-writes suggests either that the candidate is no c_malloc function, or (if the occurrence is rare) that the application is buggy. To deal with allocators that initialize their own memory, membrush tags all memory locations written by the candidate function (or its callees) with a unique identifier, so that is able to spot the uninitialized reads later.

Next, we retain from the remaining c_malloc candidates only those functions that never return the same memory region again until it is deallocated by a c_free (A4). We start with a big picture, and continue with the implementation details.

Our approach draws on load testing. The basic idea is that we insert a "call loop" that repeats specific invocations of the candidate functions many times. As long as we ensure that the application does not release the allocated region with a call to a c_free routine, we would expect a proper c_malloc to return a stream of distinct addresses in accordance with (A1). The candidate progresses to the next stage if either (1) it (or one of its callees) invokes the general-purpose allocator to allocate a new memory region and returns a pointer referencing it, or (2) it begins to return a non-pointer value consistently, possibly indicating that the application has run out of memory and cannot allocate any extra. In contrast, we drop the c_malloc candidate if (1) the application crashes, (2) the return value is a pointer already seen during the load test, or (3) the return value is neither a pointer nor an invariable error message.

The implementation relies on a partial reconstruction of the physical stack frame of the c_malloc candidate ④. First, we pause the execution at a `call` instruction that transfers the control flow to the candidate function, and we store the CPU context of the call site. Specifically, we record the values of the registers and the stack-based arguments. In order to replay the invocation, membrush repeatedly resets the CPU context to the recorded one, restarts the execution at

the `call` instruction, pauses it again when the function returns, and examines the return value. Figure 3 illustrates the procedure. Observe that the replay loop might corrupt the state of the application or cause a memory leak. Indeed, it allocates a number of buffers, which are never released. To err on the safe side, we restart the application after this step. membrush ensures to do the replay for every candidate function. However, it does not replay all its invocations, but a number of randomly chosen ones.

Finally, we filter out allocator wrappers (A5). membrush classifies a c_malloc candidate as a wrapper if (1) it (or one of its callees) invokes a function actually categorized as an allocator, and (2) whenever it returns a pointer, it passes a value received from a callee without modifying it. The implementation builds on the call stack ③and pointer tracking modules ⑥.

5 Custom Deallocator Detection

To detect c_free routines, membrush searches for functions that it can couple with the already identified c_malloc routines. A c_free function matches a c_malloc routine if they share their metadata, and allocate/release the same memory regions. The procedure is similar to that for c_malloc functions in that membrush filters candidate functions in a linear pipeline of stages where each stage verifies one of the conditions D1-D4 of Section 2.2. Figure 4 illustrates a high-level picture.

The first stage is based on the observation that CMA routines share some kind of metadata that records the positions of free blocks. Hence, a c_free routine accesses data in memory which c_malloc also uses to derive the return values (D1). membrush first pinpoints the metadata, and then monitors the application to identify the functions that read or modify it, which become c_free candidates.

membrush determines the metadata while c_malloc functions execute. First, when a c_malloc accesses a heap or static memory location for the first time, membrush tags it with a unique identifier. Then, it employs the DIFT module ⑤ to maintain a data flow graph which records how these values propagate and how they are combined. When the c_malloc routine returns, membrush pinpoints the metadata: it consults the graph, and lists all memory locations that contributed to the calculation of the return value. Observe that the metadata might represent either pointers or indices/offsets which a CMA uses to compute the addresses of allocated regions. As membrush employs a generic DIFT approach, it is impervious to such implementation details.

Finally, membrush monitors the application to identify the functions that access the metadata used by c_malloc routines. Each of them becomes a c_free candidate. As in Section 4, membrush analyzes the candidates in turn.

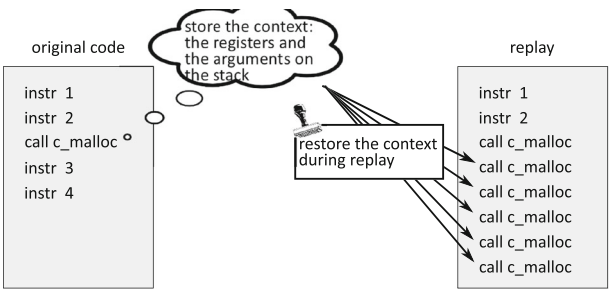


Fig. 3 The replay procedure

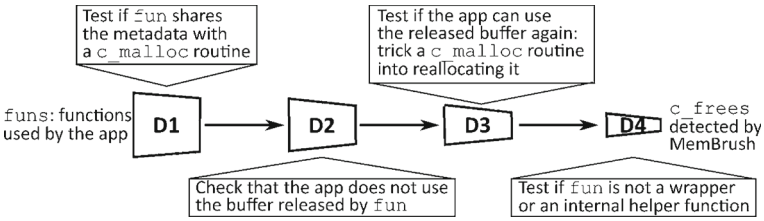


Fig. 4 Detection of c_free functions

The next two stages build on the observation that c_malloc and c_free routines handle the same memory regions. First, membrush verifies that once a c_free candidate releases a buffer, the application does not access it any more (D2). Then, it tries to make the CMA serve again a memory chunk that has just been reclaimed by a c_free candidate (D3). Both steps require that, for each c_free invocation, membrush pinpoints at least one matching c_malloc invocation, i.e., a c_malloc which allocated a buffer reclaimed by a call to the c_free candidate.

In a nutshell, membrush has two ways to couple c_malloc and c_free invocations. The first one relies on an accurate parameter match between the two functions. membrush requires that all the arguments of the c_free candidate are either the arguments or the return value of a past c_malloc invocation. In the second (more generic) method, a c_malloc and a c_free invocation match if they use the same metadata. Observe that the mapping need not be one-to-one. For instance, for region based allocators, we expect multiple c_malloc invocations to match a single c_free candidate.

Following D2, membrush requires that once a c_free candidate releases a buffer, the application does not access it any more (D2). Unless there is a use-after-free bug in the application, the presence of such accesses suggests that the candidate is not a c_free function. In practice, we tolerate some use-after-free accesses to allow for bugs in the code, but the number of such accesses should be less than ϵ. In our experiments, we used $\epsilon = 1\%$.

To analyze an invocation of a c_free candidate, membrush identifies a matching c_malloc invocation, and monitors all accesses to the associated heap buffer. If the application still uses this buffer after the c_free candidate returns, it means that the candidate function did not actually release the memory, so it does not progress to the next step.

D3 states that when c_free reclaims a chunk of memory, the CMA may serve it again on future requests. To verify a c_free candidate, we trick c_malloc into reallocating the reclaimed memory. When the candidate deallocator returns, we search the current execution trace for a c_malloc invocation that allocated a buffer in the memory that was apparently just freed, and we replay it many times in a call loop, as explained in Section 4. We retain the c_free candidate if the allocator returns the same pointer as the invocation being replayed. In contrast, we drop the candidate if the c_malloc function fails to reallocate that memory region—because it crashes, returns an error message, or requests more memory from the general-purpose allocator. As in Section 4, we restart the application after this step.

Finally, we decide which functions form the CMA interface (D4). If multiple functions in the same call stack reached this step, we pick the outermost one. The intuition is that functions above the CMA interface never directly access the metadata. Thus, if a function uses it, it must be CMA-related.

6 Custom Reallocator Detection

To detect c_realloc routines, we again generate a set of candidates candidates, and then verify them against R1-R7 of Section 2.2 in pipeline-fashion. Figure 5 presents an overview of the

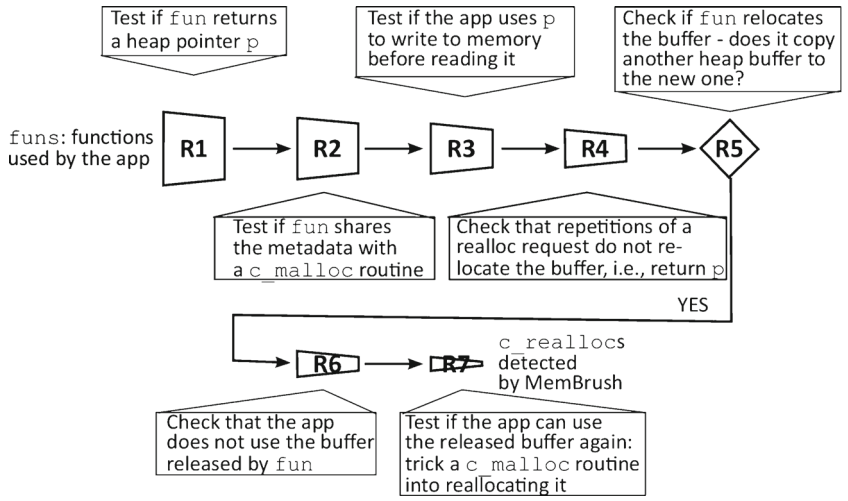


Fig. 5 Detection of c_realloc functions

algorithm. We will see that detection of reallocation routines reuses many steps of the previous sections. This makes sense, because a reallocation combines properties of deallocation and allocation.

First, we identify c_realloc candidates as those functions that return pointers to heap objects, and that share the metadata with c_malloc routines (R1 and R2). The implementation of this stages draws heavily on the checks for A1 and D1. Next, to verify if the application uses a pointer returned by a c_realloc candidate to write to the reallocated heap buffer in a write-before-read fashion (R3), we reuse the verification of A2 and A3.

R4 requires that if a c_realloc candidate repeatedly serves a specific request, only the first invocation should trigger an action and may relocate the buffer. Again, we confirm this behavior by replaying the invocations. Specifically, when the candidate returns, membrush replays this invocation many times in a call loop, and retains the candidate only if the returned value remains constant.

Next, we analyze if an invocation of a c_realloc candidate relocates a memory block to modify its size (R5). To confirm that the memory block is not relocated (but only resized), a simple test could check that a pointer returned by the candidate indicates an object allocated by a c_malloc function that is not yet freed. Observe, however, that this requires an ability to accurately pinpoint all objects released by c_free routines. As we explain in Section 9, there exist CMA implementations which make it very challenging.

membrush, on the other hand, leverages the fact that c_realloc preserves the contents of reallocated memory blocks. Thus, when a c_realloc function relocates an object, it also copies the old contents. To detect the copy operation, membrush uses the DIFT module ⑤. It monitors if the c_realloc candidate (or any of its callees) copies data from a buffer already allocated by a c_malloc. In case of a relocation, membrush expects a copy of a contiguous block from an address returned by a c_malloc to the return value of the candidate. The source of this operation is the reallocated buffer.

This mechanism selects only these c_realloc candidates that relocate a memory block at least once during our analysis. Indeed, otherwise the application does not perform the buffer copy operation, and membrush cannot proceed to the sixth step (R6). This requirement potentially limits the completeness of the detection algorithm if a c_realloc function is always called to shrink

a memory region, and never to extend it. Even though we did not find it to be a problem during our experiments, an alternative implementation could also examine if the reallocation was in place, i.e., returned a pointer to an already allocated object.

When the previous stage concludes that an invocation of a c_realloc candidate relocates a buffer, we also confirm that the application does not access the reallocated buffer anymore (R6), and that the memory block is in fact freed (R7). This check is identical to the verification of D2 and D3—again, we monitor the released memory, and we trick c_malloc routines into reallocating it. The reallocated buffer determines the c_malloc invocation we need to replay.

7 Additional Analysis of the CMA Routines

We now unearth additional characteristics of CMAs. First, we describe membrush's heuristic to estimate the size of buffers requested through c_malloc/c_realloc functions, and then we discuss how we distinguish between the different types of allocators from Section 2.1.

7.1 Buffer Size Estimation

Besides finding the CMA, it is often useful also to determine the size of the memory that is allocated. For instance, knowing the size of memory chunks helps in tasks like reverse engineering, and hunting for heap-based bugs. Before we describe membrush's procedure to estimate how much memory the application requests from a custom allocator routine, observe that it is not a trivial task. After all, since the application may well allocate more memory than it will need during our tests, we cannot just monitor how much of the buffer is actually used. membrush, instead, first collects a number of sample c_malloc[2] invocations along with an upper boundary on the size of the allocated buffers. Then, it tries to devise a formula capturing the relation between an argument of the c_malloc function and the associated size.

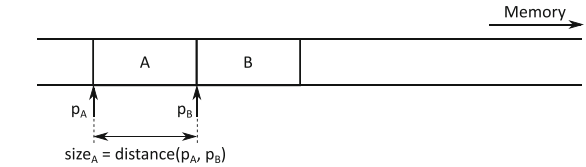
The collection of samples is again based on the replay mechanism. membrush replays a number of a c_malloc function invocations many times, and for each of them, it monitors the stream of returned values. When the allocator serves two consecutive requests from the same region obtained from the general purpose allocator, membrush measures the distances between them. They represent the upper bound on the size of the allocated buffers. Additionally, if membrush finds that the CMA stores the metadata between the chunks returned to the application, it excludes these bytes from the distance measurement.

Observe that, we should only include the distances between memory chunks adjacent to each other, lest we significantly overestimate the upper bound on their size. To this end, membrush waits for the c_malloc function to invoke the general-purpose allocator to allocate a new memory region, and serve the requests from it (refer to the verification of A4 in Section 4). This way, we are certain that we keep track of all the buffers allocated in that region, so our estimation of their size is as accurate as possible.

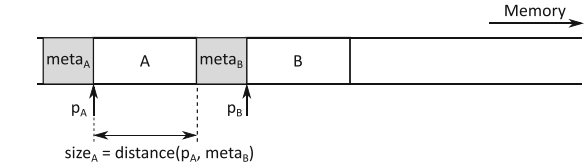
Depending on the location of the metadata, we can distinguish between the following three cases:

1. The CMA stores the metadata and the allocated buffers *apart* (refer to Fig. 6a). In this case, the distances between the newly allocated buffers accurately represent their sizes.
2. The CMA stores the metadata *in front* of the newly allocated buffer. This scenario is illustrated in Fig. 6b, where a c_malloc routine is invoked twice to subsequently allocate buffers A, and B, and return pointers p_A, and p_B, respectively. During the allocation of B, c_malloc

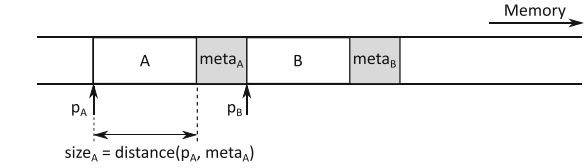
[2]We follow exactly the same procedure for c_reallocsmall routines.



(a) The metadata and the allocated buffer are stored apart.



(b) The metadata is stored in front of the allocated buffer.



(c) The metadata is stored behind the allocated buffer.
- If the allocator does not initialize the newly allocated memory region, MemBrush accurately estimates the size of the buffer.
- If the allocator initializes the newly allocated memory region, though, MemBrush cannot spot the boundary between the buffer and the metadata, and as a result it may overestimate the size of the buffer.

Fig. 6 Buffer size estimation: grey regions represent metadata kept by the allocator, white regions represent the newly allocated buffers

does not access the region occupied by buffer A, yet it updates $\texttt{meta}_\texttt{B}$ – the metadata associated with B. membrush monitors the memory locations modified by c_malloc to distinguish the buffer allocated by the application (A) from the metadata ($\texttt{meta}_\texttt{B}$). The distance between the beginning of $\texttt{meta}_\texttt{B}$ and A accurately determines the size of the buffer.

3. The CMA stores the metadata *behind* the newly allocated buffer. This scenario is illustrated in Fig. 6c, where a c_malloc routine is again invoked twice to subsequently allocate buffers A, and B, and return pointers $\texttt{p}_\texttt{A}$, and $\texttt{p}_\texttt{B}$, respectively. If the allocator *does not initialize* the newly allocated buffer, membrush can again pinpoint the metadata associated with A, and precisely estimate the size of the memory request. Conversely, if the allocator *does initialize* the newly allocated buffer, membrush cannot find the boundary between the buffer (A) and the metadata ($\texttt{meta}_\texttt{A}$). As a consequence, the distance between the beginning of the buffers represents an overapproximated size of the memory request.

In the second step, for each c_malloc routine, membrush tries to derive a formula describing the size of an allocated buffer as a function of an argument of the c_malloc. Specifically, when we denote the size of the allocation request and the value of one of the arguments of the c_malloc function by $size$ and arg, respectively, we assume that the CMA uses one of the following formulas:

$$size = a_1 * arg + b_1 \quad \text{or} \quad size = a_2 * 2^{arg} + b_2.$$

Next, for each argument variable of the allocator, arg_i, we consider all the collected pairs of the maximum estimated size and arg_i, (max_size, arg_i), and we search for values of a_1, b_1, a_2, and b_2 such that

$$max\_size \geq a_1 * arg_i + b_1 \quad \text{and} \quad max\_size \geq a_2 * 2^{arg_i} + b_2.$$

Finally, we select (a_1 and b_1) or (a_2 and b_2) that *fit* the samples best, i.e., minimize the cumulative distance between the values of the formula and the boundary sizes.

As we show in Section 8, membrush's mechanism yields good results in practice with only very few exceptions. It does not work only if the object size is determined when the application initializes an instance of an allocator, and not when it allocates a buffer. Then, different invocations of the allocator function result in different allocation sizes, yet we cannot find a relation between them and the function's arguments. For this reason, we did not manage to fully analyze the `ngx_array_push(a)` function in the `nginx` webserver. It serves requests from the array `a` passed as an argument. The problem is that the application determines the size of the buffers when creating the array.

7.2 Classification of CMAs

To classify CMAs, we examine two characteristics: the sizes of allocated buffers, and the relation between the allocation and deallocation routines. Additionally, we need a means to distinguish generic regions from obstacks.

First, we check if a CMA splits a region obtained from a general-purpose allocator into equal-sized chunks. To this end, we monitor objects whose addresses are derived from the base of a particular `malloc`/`mmap` buffer, and we compare their sizes. Next, we assess if a deallocator releases individual or multiple objects at once. To find it out, we check how many c_malloc invocations match a single invocation of a c_free (refer to Step 1 in Section 5).

Table 1 summarizes the decision procedure. As the basic criteria are stringent enough to distinguish all allocator types except from obstacks, we adopt just one extra one. Observe that, since obstacks allow for the freeing of objects allocated since the creation of any object in the region, allocations following a call to a c_free function do not necessarily start at the bottom of the region, but at any location inside it. Thus, we monitor streams of addresses of objects within individual regions, and we check if their increasing subsequences start at the same location.

Even though it was not necessary in our experiments, we could additionally validate the per-class allocators. Instead of comparing only the sizes of allocated objects, we can also examine their low-level data structures. We demonstrate this procedure in Section 8.3.

8 Evaluation

In this section, we evaluate membrush. We discuss its accuracy (Section 8.1), present some statistics illustrating the detection procedure (Section 8.2), and finally we demonstrate the practical

Table 1 Membrush's criteria to classify CMAs

Allocator	Equal-sized chunks	Individual object deallocation	Multiple
Per-class	✓	✓	×
Regions[a]	×	×	✓
Obstacks[a]	×	×	✓
Custom patterns	×	✓	×
Hybrid approaches	×	✓	✓

[a] We use additional criteria to distinguish regions from obstacks

benefits of applying membrush to an existing binary analysis technique for reverse engineering data structures (Section 8.3).

8.1 Accuracy of membrush's Detection Algorithm

In this section, we evaluate the accuracy of membrush. We start with an overview of the applications we tested, and we report how well membrush managed to pinpoint the CMA routines. Then, we continue with a classification of CMAs. Finally, we discuss the accuracy of membrush's heuristic to estimate the size of buffers requested through c_malloc functions.

The accuracy of the CMA routines detection. Table 2 presents an overview of the applications we analyzed with membrush. The list contains five real-world programs, including the Apache and Nginx webservers, smbget from the Samba networking tool, the ProFTPD file server, and wget (configured to use the lockless allocator (Lockless: Lockless Performance, http://locklessinc.com)). Additionally, we applied membrush to the SpecINT 2006 benchmarking suite. It contains both applications that employ a CMA and applications that do not. To verify membrush's accuracy, we compare the results to the actual CMA routines in the programs. Thus, all the results presented in this section were obtained for binaries for which we could also consult the source code and get the ground truth. For each application, we report the number of detected CMA routines compared to the number of the CMA routines in the application (true positives, TPs), and the number of functions mistakenly classified as CMA routines (false positives, FPs).

Table 2 The accuracy of membrush's algorithm. in detecting c_mallocsmall, c_freesmall, and c_reallocsmall routines. The top part of the table reports the results for 5 real-world applications, and the bottom one — for the SpecInt 2006 benchmarking suite

Application	Allocators		Deallocators		Reallocators	
	TPs	FPs	TPs	FPs	TPs	FPs
apache	3/5	-	4/6	-	0/1	-
nginx	7/7	-	2/2	-	0/0	-
smbget (samba)	1/1	-	1/1	-	1/1	-
wget	1/1	-	1/1	-	1/1	-
proftpd	6/6	-	5/5	-	0/0	-
400.perlbench	14/16	-	5/5	-	0/0	-
401.bzip2	0/0	-	0/0	-	0/0	-
403.gcc	14/17	4	5/5	-	0/0	-
429.mcf	0/0	-	0/0	-	0/0	-
446.gobmk	0/0	-	0/0	-	0/0	-
456.hmmer	0/0	-	0/0	-	0/0	-
458.sjeng	0/0	-	0/0	-	0/0	-
462.libquantum	0/0	-	0/0	-	0/0	-
464.h26ref	0/0	-	0/0	-	0/0	-
471.omnetpp	0/0	-	0/0	-	0/0	-
473.astar	0/0	-	0/0	-	0/0	-
483.xalancbmk	6/6	-	6/6	-	0/0	-
Total:	52/59	4	29/31	-	2/3	-

Overall, membrush detected correctly 52 out of 59 c_malloc functions (88%), 29 out of 31 c_free routines (94%), and 2 out of 3 c_realloc functions (67%). As we discuss below, many false negatives stem from compiler optimizations, and we could prevent lots of them. As far as the false positives are concerned, there were four. Even though strictly speaking, these functions are false positives, in practice they were wrappers of an inlined allocator. Thus, by just looking at the binary, membrush has no means to provide more accurate results (Balakrishnan and Reps 2010), and the identified functions do provide the application with memory chunks acting as proper allocators.

For the false negatives, we often missed a custom allocator because we did not even classify it as a c_malloc candidate in the first step. We identified two reasons for this: (1) the allocator passes a pointer in an outgoing argument, and not in the return value, or (2) instead of a pointer to a heap object, the allocator returns an offset, which the application adds to the base of a buffer (often using a macro) before accessing the memory. E.g., in Apache, the `apr_rmm_malloc`, `apr_pool_create_ex` custom allocators, and also the `apr_rmm_realloc` reallocator, show this behavior. The same holds for the two missing allocators in 400.perlbench, and one of the misses in 403.gcc. In order to reduce the first source of false negatives, we could extend membrush to consider results returned in parameters also, using the techniques described by ElWazeer et al. (2013). To handle the allocators returning an offset instead of a pointer, we could use dynamic information flow tracking to tell if the value returned by a function is later used to derive a pointer dereferencing heap memory. We leave it as a future work.

The remaining two false negatives in 403.gcc stem from compiler optimizations. In the first case, the application always jumps to, and never calls, one of the custom allocators. In the second case, the `alloc_page` routine is inlined. membrush detected four functions, which are, strictly speaking, wrappers of `alloc_page`, but in practice behave as allocators. We formally classified them as false positives, even though they would be useful results in practice.

The two misses in the custom deallocator detection in Apache are caused solely by the false negatives in the allocator detection. `apr_rmm_malloc` and `apr_pool_create_ex` are the only allocators that can reallocate the memory released by `apr_rmm_free` and `apr_pool_destroy`, respectively. Since we did not detect the allocators, we did not manage to trick them into reallocating the just reclaimed memory either. As a result the two deallocator candidates did not pass the D3 filter.

In summary, we see that membrush's algorithm proves effective with very few false positives. The reason for all the important false negatives is that we do not identify the values returned by a function accurately enough. However, we can employ existing techniques to further improve the procedure.

The accuracy of the CMA classification. Figure 7 presents the types of custom memory allocators classified by membrush. The bottom part of the graph contains correctly classified functions, and the top one – misclassifications. In the 403.gcc benchmark, membrush erroneously mistook obstacks for region based allocators. Even though these allocators are conceptually obstack-based, each obstack is implemented as a list of chunks, and not as a region split into individual buffers. The CMA inserts new nodes in the list whenever an allocation occurs, and deletes a number of the most recently added ones upon deallocation. Thus, the addresses of allocated chunks, i.e., list elements, do not form increasing subsequences as we expected (refer to Section 7.2). However, as obstacks are a more generic version of regions, we are not too concerned with this misclassification.

The accuracy of the buffer size estimation. In general, membrush either accurately estimated how much memory the application requests from a custom allocator routine, or did not provide any results. It means, that membrush's analysis is accurate, and the results are not misleading. membrush did not manage to deal with 7 out of 59 allocators. As we mentioned already,

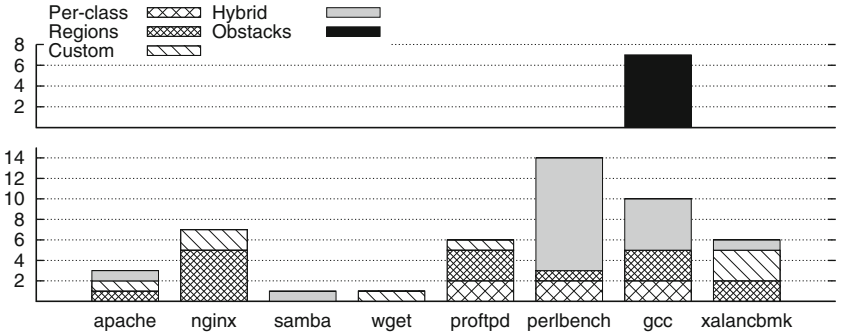


Fig. 7 The accuracy of membrush's procedure to classify CMA routines. The bottom part of the graph presents the allocators that were classified correctly, and the top one summarizes misclassifications

in all these cases, the application determines the size of the buffers when creating an allocator, and not when allocating an object. Examples include the `ngx_array_push` function in nginx, and the `apr_array_push` function in Apache. For all the remaining allocators, we found that the size of the allocation is either of the form $(arg + b)$ or it is a constant. These results were correct.

8.2 Effectiveness and Necessity of Filtering Stages

We now present some statistics illustrating the analysis procedure. Due to space constraints, we limit the discussion to the detection of the allocation routines. Analyzing the power of each heuristic individually is hard, as filtering stages frequently depend on previous ones. For example, to be able to test whether a pointer is used to write to memory (A2) or incurs a write before read (A3), we must rst get this pointer from an allocator candidate (A1). Without A1, we cannot evaluate the power of A2 and A3. Likewise, heuristic A5 tries to detect wrappers, but it cannot exam the relation between them without the candidates. And so on. Moreover, even if we can evaluate some heuristics separately in theory, doing so may not be practical. For instance, our replay mechanism (A4) becomes prohibitively expensive if there is not a set of aggressive pre-filtering stages to precede it.

Figure 8 shows how many allocator candidates membrush analyzed in each step of its detection procedure. For all the applications, the A1 filter identifies up to 430 c_malloc candidates (with a median of 78), and their number gradually drops as membrush proceeds. Each time, it finds at least 1 wrapper function (193 for 483.xalancbmk, with a median of 14), often invoking the general-purpose allocator.

8.3 Practical Benefits - a Show Case

In this section, we demonstrate the benefits of applying membrush to a binary analysis. We show that by furnishing an existing reverse engineering tool with information about the interface implemented by a CMA, we significantly increase the accuracy of the analysis.

Howard (Slowinska et al. 2011) is a tool to reverse data structures in stripped binaries. To analyze the memory allocated on the heap, it interposes on the system's `malloc` and `free` functions, and tracks each chunk of memory thus allocated as a data structure. Thus, when the binary uses a CMA, Howard does not analyze the data structures at the granularity used by

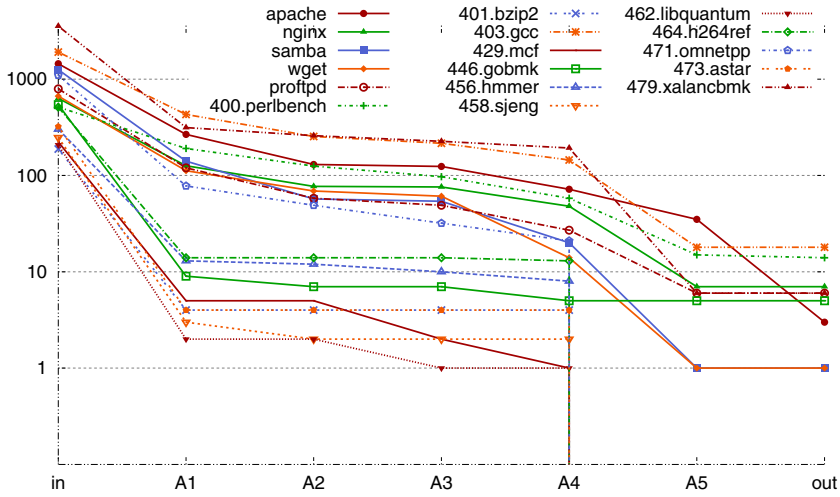


Fig. 8 The number of allocator candidates analyzed by membrush when verifying characteristics A1-A5. In Apache, there are 35 functions after the A5 step, and as they belong to different shared libraries, they map to 3 functions in the `libapr`/`libapr-util` libraries

the application, and its accuracy is low. However, with the knowledge acquired by membrush, Howard can interpose on the routines used by the CMA, and further perform its analysis.

As an example, we analyze heap memory in the smbget utility in Samba. As the core memory allocator, it uses `talloc` (Samba: talloc Documentation, http://talloc.samba.org/talloc/doc/html/index.html), a hierarchical, reference counted memory pool system. membrush detects two CMA routines: the `__talloc()` allocator and the `__talloc_free()` deallocator. Table 3 presents the results obtained by Howard in two cases: (1) when it analyzes buffers allocated by the general purpose allocation routines, and (2) when it also interposes on the `__talloc()` and `__talloc_free()` functions found by membrush. We split the results into four categories:

- **OK**: Howard identified the entire data structure correctly (i.e., a correctly identified structure field is not counted separately).
- **Flattened**: fields of a nested structure are counted as a normal field of the outer structure.
- **Missed**: Howard misclassified the data structure.
- **Unused**: single fields, variables, or entire structures that were never accessed during our tests.

As expected, when we use the vanilla version of Howard, all the memory that belongs to the heap buffers that are later used by the CMA, is erroneously classified as arrays. Thus, we get meaningful results only for the remaining 58.5% of the arrays and 53.2% of the structs allocated on the heap.

In contrast, when we combine Howard with membrush, the accuracy of the analysis increases significantly. Now, 93.2% of the arrays and 91.3% of the struct variables allocated on the heap are classified correctly. We counted 8.7% flattened structures. They are all caused by a large `tevent_req` structure containing two nested substructures. As the addresses of the substructures fields are always calculated relative to the beginning of `tevent_req`, Howard had no means of classifying these regions as individual structures. The results show that by using membrush, Howard is able to analyze the data structures actually used by smbget, instead of the large buffers further split by the CMA routines.

Table 3 The accuracy of the data structure analysis without and with membrush's detection of CMA functions

Category	Without membrush		With membrush	
	Arrays	Structs	Arrays	Structs
The results in the number of variables:				
OK	58.5%	53.2%	93.2%	91.3%
Flattened	0%	0%	0%	8.7%
Missed	41.5%	46.8%	6.8%	0%
Unused	0%	0%	0%	0%
The results in the number of bytes:				
OK	60.4%	51.7%	92.4%	90.2%
Flattened	0%	0%	0%	9.8%
Missed	39.6%	48.3%	7.6%	0%
Unused	0%	0%	0%	0%

9 Limitations

membrush is not flawless. In this section, we discuss some generic limitations we have identified.

Compiler optimizations. In general, membrush detects CMA routines at runtime, so the analysis results correspond to the optimized code, which may be different from what is specified in the source. This is known as WYSINWYX (What You See Is Not What You eXecute) (Balakrishnan and Reps 2010), and it might lead to inaccuracies. For instance, in the 403.gcc benchmark, membrush has no means to identify an inlined allocator, leading to the four functions formally classified as false positives. Observe that analyzing the code that executes is of course the right thing to do. Otherwise, we would not be able to analyze the real behavior of the binary or perform proper forensics.

Function parameter identification. In order to identify the CMA routine candidates, and later accurately match c_free and c_malloc invocations, membrush monitors the return value and the arguments of functions. Our current implementation assumes that functions pass the return value using the `EAX` register, and the parameters using the stack. As we saw in Section 8.1, this is not always enough. However, we could extend our technique as proposed by ElWazeer et al. (2013).

Identification of the value returned by a c_malloc routine. We assume that c_malloc returns a pointer to the newly allocated buffer. In theory, however, it could pass an offset to a large chunk of memory, that the application subsequently adds to the base of the chunk. This issue can be solved with a more careful implementation. (It did not pose a problem in our experiments.)

Identification of the buffers released with a c_free routine. Even though membrush can accurately detect c_free routines, there exist CMA implementations which make it very challenging to pinpoint all the memory that is freed. For instance, when one of the deallocators in the Apache webserver releases a pool, it also reclaims all its subpools, which are separate regions obtained from the general purpose allocator. Finding out in an implementation-agnostic way is difficult.

Memory overhead. Our current implementation is somewhat constrained by memory limitations. Specifically, we use 4 bytes per byte used by the application for the tag, and also need to maintain the heap states, the data flow inside the functions, etc. For large applications, like Firefox, we can not easily do this on our 32 bit systems.

Performance overhead. The current implementation of membrush is slow. We use the "replay" mechanism to validate the candidate for both c_malloc and c_free. For c_malloc, it is acceptable. However, when we have a lot of allocators, then we need to "replay" all possible c_free and pair them. This step costs a lot of time (easily a few hours).

Determinism. Finally, we rely on deterministic execution of the applications. Basically, we need to detect the candidates in the first run, then replay them in the second run to do validation. For c_malloc detection, it is fine, because we do not need to know which instance of c_malloc we need to replay and can just randomly pick any number of the c_malloc instance and validate them. For c_free, it is much harder because we need to pair them. If needed, we can rely on the existing record and replay mechanisms, such as (Guo et al. 2008; Honarmand and Dautenhahn 2013).

10 Related Work

In this section, we discuss some of the recent research related to both custom memory allocation (Section 10.1) and the identification of a specific functionality in a binary (Section 10.2).

10.1 Custom Memory Allocation

Custom memory allocation is a mature field. Surveys by Wilson et al. (1995) and Berger et al. (2002) provide comprehensive overviews of various types of CMAs. Many real world applications use CMAs, typically to improve runtime performance. Well-known examples include the Apache and Nginx webservers, the Firefox web browser and the gcc compiler, among many others.

Many research projects, like (Schneider et al. 2006; Jula and Rauchwerger 2007, 2009; Liu and Chen 2012; Lyberis et al. 2012), propose new memory managers designed for high-performance memory allocation. Other approaches, e.g., (Berger and Zorn 2006; Lvin et al. 2008; Novark and Berger 2010; Novark et al. 2009; Perence, B, Electric Fence, http://perens.com/FreeSoftware/ElectricFence; Akritidis 2010; Serebryany et al. 2012), have used custom memory managers tailored to improve the memory safety of applications using them. They detect or help mitigate heap corruptions, dangling pointers or reads of uninitialized data. For instance, Electric Fence (Perence, B, Electric Fence, http://perens.com/FreeSoftware/ElectricFence) and AddressSanitizer (Serebryany et al. 2012) place an inaccessible memory region after each block allocated by `malloc`. Once a buffer overflow vulnerability is exploited, and an application overflows a heap buffer, the mechanisms spot an illegal memory access and raise an alert. Electric Fence relies on the CPU page protection – for each heap buffer, it allocates an extra page that is marked as inaccessible. DieHarder (Novark and Berger 2010) (a descendant of DieHard (Berger and Zorn 2006)) finds memory bugs probabilistically. Specifically, its modified `malloc` function also adds redzones around memory regions returned to the user, but instead of marking them as inaccessible, it populates the newly allocated memory with special magic values. If a magic value in a redzone is overwritten, this will later be detected when the redzone is examined on `free`.

Many approaches that detect buffer overflows, use-after-free or double-free attacks (Valgrind, http://valgrind.org; Hastings and Joyce 1992; Dhurjati and Adve 2006; Caballero et al. 2012; Slowinska et al. 2012) rely on information about the programs' data structures—specifically, the buffers that they should protect. Thus, in the presence of CMAs, their scope is limited to memory chunks obtained from the general-purpose allocators. They would all directly benefit from membrush—to offer a finer grained protection, and to detect attacks on the actual data structures used by applications. For instance, BinArmor (Slowinska et al. 2012) protects heap memory regions from being overflown by assuring that pointers keep pointing to the same buffer,

i.e., it forbids a pointer assigned to buffer `A` from accessing buffer `B`. Currently, it targets memory regions allocated by `malloc`, which is inaccurate if the application uses a CMA.

10.2 Identification of a functionality in a binary

While membrush aims to automatically identify CMA routines in existing binaries, a few recent approaches, e.g., (Lutz 2008; Gröbert et al. 2011; Calvet et al. 2012), analyze binaries to see if they use cryptographic primitives. These projects are related to membrush in the sense that they also perform a dynamic analysis of a binary to search for predefined characterisitcs.

Noe Lutz (2008) proposes to use three indicators to recognize cryptographic code in execution traces: presence of loops, a high ratio of bitwise arithmetic instructions, and entropy change in the data manipulated by the code. However, these heuristics may lead to inaccuracies. For instance, arithmetic instructions are used commonly, and may prove an unreliable indicator of the presence of cryptographic primitives. Aligot (Calvet et al. 2012), on the other hand, proposes a method that is independent of the actual implementation. It exploits the knowledge of the particular input-output relationships of cryptographic functions. For an executed function, Aligot extracts its input and output parameters, and compares them with those of known cryptographic routines. This way, Aligot detected e.g., RC4, AES or MD5 algorithms used by binaries.

membrush is very different from these approaches. It relies neither on particular implementation features nor on a collection of known routines.

The most important outcome of our literature study, is that there is, to our knowledge, no work on detection of custom memory allocation routines.

11 Conclusion

Custom memory allocators are very common in real-world applications, where they are used instead of the standard allocation functions for performance reasons. Unfortunately, many existing binary analysis techniques depend on the ability to intercept the memory allocation functions. Up to now this was not possible. In this paper, we presented a set of techniques for identifying custom memory allocation, deallocation, and reallocation functions. Each of these three categories is handled by a separate pipeline of filters that aim to test fundamental properties that most hold for almost any implementation. We evaluated our techniques on a diverse set of custom memory allocator implementations and verify their accuracy on both SpecInt and several real-world applications that are known to use custom memory allocators. In practically all cases, we showed that we can find the allocation routines with great accuracy. Finally, we showed that the outcome of our research is immediately useful by using the results in the Howard data structure extraction tool.

Acknowledgments This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, and the EU FP7 SysSec Network of Excellence.

References

Akritidis P (2010) Cling: a memory allocator to mitigate dangling pointers. In: Proceedings of the 19th USENIX conference on security, SSYM'10. USENIX Association. http://dl.acm.org/citation.cfm?id=1929820.1929836

Balakrishnan G, Reps T (2004) Analyzing memory accesses in x86 binary executables. In: Proceedings of the conference on compiler construction, CC'04

Balakrishnan G, Reps T (2010) WYSINWYX: what you see is not what you execute. ACM Trans Program Lang Syst 32:23:1–23:84. doi:10.1145/1749608.1749612

Berger ED, Zorn BG (2006) DieHard: probabilistic memory safety for unsafe languages. In: Proceedings of the 2006 ACM conference on programming language design and implementation, PLDI'06. doi:10.1145/1133981.1134000

Berger ED, Zorn BG, McKinley KS (2002) Reconsidering custom memory allocation. In: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02, vol 37. doi:10.1145/583854.582421

Caballero J, Grieco G, Marron M, Nappa A (2012) Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proceedings of the 2012 international symposium on software testing and analysis, ISSTA'12. doi:10.1145/04000800.2336769

Calvet J, Fernandez JM, Marion JY (2012) Aligot: cryptographic function identification in obfuscated binary programs. In: Proceedings of the 2012 ACM Conference on computer and communications security CCS'12. ACM, New York, pp 169–182. doi:10.1145/2382196.2382217

Chen X, Slowinska A, Bos H (2013) Who allocated my memory? Detecting custom memory allocators in C binaries. In: Proceedings of the 20th working conference on reverse engineering, WCRE'13, pp 22–31. doi:10.1109/WCRE.2013.6671277

Chipounov V, Kuznetsov V, Candea G (2011) S2E: a platform for in vivo multi-path analysis of software systems. In: Proceedings of the 16th international conference on architectural support for programming languages and operating systems, ASPLOS'11

Dhurjati D, Adve V (2006) Efficiently detecting all dangling pointer uses in production servers. In: Proceedings of the international conference on dependable systems and networks, DSN'06. IEEE. doi:10.1109/DSN.2006.31

ElWazeer K, Anand K, Kotha A, Smithson M, Barua R (2013) Scalable variable and data type detection in a binary rewriter. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation, PLDI'13

Gröbert F, Willems C, Holz T (2011) Automated identification of cryptographic primitives in binary programs. In: Proceedings of the 14th international conference on recent advances in intrusion detection, RAID'11. Springer, Berlin, pp 41–60. doi:10.1007/978-3-642-23644-0_3

Guo Z, Wang X, Tang J, Liu X, Xu Z (2008) R2: an application-level kernel for record and replay. In: Proceedings of the 8th USENIX conference on operating systems design and implementation, OSDI'08. http://dl.acm.org/citation.cfm?id=1855755

Hanson DR (1990) Fast allocation and deallocation of memory based on object lifetimes. Softw Pract Experience 20(1). doi:10.1002/spe.4380200104

Hastings R, Joyce B (1992) Purify: fast detection of memory leaks and access errors. Winter USENIX conference

Honarmand N, Dautenhahn N (2013) Cyrus: unintrusive application-level record-replay for replay parallelism. In: Proceedings of the 18th international conference on architectural support for programming languages and operating systems, ASPLOS'13. http://dl.acm.org/citation.cfm?id=2451138

Intel (2011) Pin - A dynamic binary instrumentation tool. http://www.pintool.org/

Jones MT (2007) Anatomy of the linux slab allocator. http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/

Jula A, Rauchwerger L (2007) Custom memory allocation for free. In: Proceedings of the 19th international conference on languages and compilers for parallel computing, LCPC'06. Springer, Heidelberg

Jula A, Rauchwerger L (2009) Two memory allocators that use hints to improve locality. In: Proceedings of the 2009 international symposium on memory management, ISMM'09. ACM Press, New York. doi:10.1145/1542431.1542447

Jung C, Clark N (2009) DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. In: Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture, MICRO-42. doi:10.1145/1669112.1669122

Kemerlis VP, Portokalidis G, Jee K, Keromytis AD (2012) Libdft: practical dynamic data flow tracking for commodity systems. In: Proceedings of the 8th annual international conference on virtual execution environments, VEE'12

Lin Z, Zhang X, Xu D (2010) Automatic reverse engineering of data structures from binary execution. In: Proceedings of the 17th annual network and distributed system security symposium, NDSS'10

Liu R, Chen H (2012) SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In: Proceedings of the third ACM SIGOPS Asia-Pacific workshop on systems, ApSys'12. USENIX association. http://dl.acm.org/citation.cfm?id=2387841.2387856

Lutz N (2008) Towards revealing attackers' intent by automatically decrypting network traffic. Master's thesis. ETH, Zurich

Lvin VB, Novark G, Berger ED, Zorn BG (2008) Archipelago: trading address space for reliability and security. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems, ASPLOS XIII. doi:10.1145/1346281.1346296

Lyberis S, Pratikakis P, Nikolopoulos DS, Schulz M, Gamblin T, de Supinski BR (2012) The myrmics memory allocator. In: Proceedings of the 2012 international symposium on memory management, ISMM'12. ACM Press, New York. doi:10.1145/2258996.2259001

Novark G, Berger ED (2010) DieHarder: securing the heap. In: Proceedings of the 17th ACM conference on computer and communications security, CCS'10. ACM, New York, pp 573–584. doi:10.1145/1866307.1866371

Novark G, Berger ED, Zorn BG (2009) Efficiently and precisely locating memory leaks and bloat. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation, PLDI'09. ACM, New York, pp 397–407. doi:10.1145/1542476.1542521

Portokalidis G, Slowinska A, Bos H (2006) Argos: an emulator for fingerprinting zero-day attacks. In: Proceedings of the 1st ACM european conference on computer systems 2006, EuroSys'06. doi:10.1145/1218063.1217938

Reps T, Balakrishnan G (2008) Improved memory-access analysis for x86 executables. In: Proceedings of the joint european conferences on theory and practice of software 17th international conference on compiler construction, CC'08/ETAPS'08

Ross DT (1967) The AED free storage package. Commun ACM 10(8). doi:10.1145/363534.363546

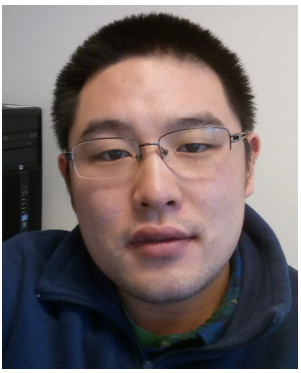
Schneider S, Antonopoulos CD, Nikolopoulos DS (2006) Scalable locality-conscious multithreaded memory allocation. In: Proceedings of the 2006 international symposium on memory management, ISMM'06. ACM Press, New York, p 84. doi:10.1145/1218063.1217938

Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) AddressSanitizer: a fast address sanity checker. In: Proceedings of USENIX annual technical conference

Slowinska A, Stancescu T, Bos H (2011) Howard: a dynamic excavator for reverse engineering data structures. In: Proceedings of the 18th annual network & distributed system security symposium, NDSS'11

Slowinska A, Stancescu T, Bos H (2012) Body armor for binaries: preventing buffer overflows without recompilation. In: Proceedings of USENIX annual technical conference, USENIX ATC'12

Wilson PR, Johnstone MS, Neely M, Boles D (1995) Dynamic storage allocation: A survey and critical review



Xi Chen was born in NanTong, China, in 1985. He received the B.A degree in Computer Science from the Beijing University of Posts and Telecommunications, Beijing, China, in 2008. He got his M.Sc degree in Computer Engineering from the Dresden University of Technology, Dresden, Germany, in 2011. In 2012, he joined the network and system security group in Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, as a PhD candidate. His research focuses on binary reverse engineering, static/dynamic data-flow analysis and zero-day attack prevention.

Asia Slowinska is an assistant professor in the Systems and Network Security group at the VU Amsterdam. Her work focuses on developing techniques to automatically analyze and reverse engineer complex software that is available only in binary form. Further, she looks into mechanisms that proactively protect software from malicious activities.

Herbert Bos is a full professor in Systems and Network Security at Vrije Universiteit Amsterdam. He obtained his Ph.D. from Cambridge University Computer Laboratory (UK). He is very proud of all his (former) students, three of whom have won the Roger Needham Ph.D. Award for best Ph.D. thesis in systems in Europe. In 2010, Herbert was awarded an ERC Starting Grant for a project on reverse engineering that is currently keeping him busy.