Who Allocated My Memory? Detecting Custom Memory Allocators in C Binaries

Xi Chen Asia Slowinska Herbert Bos
Vrije Universiteit Amsterdam, The Netherlands
x.chen@vu.nl, {asia,herbertb}@few.vu.nl

***Abstract*—Many reversing 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. As we show, we can furnish existing reversing 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.

I. Introduction

Many reversing techniques for data structures depend on the analysis of memory allocated on the heap [1]–[5]. 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 compiler, and Dropbox. 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/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.

The main goal of MemBrush is to furnish existing reversing 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 reuse the memory analysis techniques for general-purpose allocators in applications that 'roll their own'. To demonstrate it, we use MemBrush to support an existing reverse tool called Howard [2]. 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 even see.

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 [6]–[11]. 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, and so on. 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 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 [12]. Our current implementation works with x86 C/C++ binaries on Linux generated by the gcc optimising compiler, but the approach is not specific to any particular OS or compiler.

II. 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).

Rather than aiming for this or that custom memory allocator, the objective of MemBrush is to detect *any* CMA. In Section II-A, we therefore introduce popular types of custom memory allocators. Then, in Section II-B, we list the essential characteristics of CMAs that lay the foundation for our detection algorithm described in Sections III-VI.

A. A Taxonomy of CMAs

Since comprehensive overviews of CMAs can be found in surveys by Wilson et al. [13] and Berger et al. [14], we limit ourselves to a summary of the approaches in this section. Like Berger et al. [14], 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 [15] ("zones") and Linux [16].

Regions (also known as *arenas*, *groups*, and *zones* [17], [18]). 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 [19] (which refers to them as "pools"), PostgreSQL [20] (which refers to them as "memory contexts"), and Nginx [21].

Obstacks. An obstack [22] 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.

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* [14] are a combination of regions and general-purpose allocators that extend region semantics with individual object deletion.

B. 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 VIII exhibit these characteristics. As we will see in Sections III-VI, 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:

(A1) Normally, a `c_malloc` function returns a pointer `p` that references a heap memory region. As we discuss in Section III, 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.

(A2) 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.

(A3) Unless the `c_malloc` function initializes memory chunks prior to returning them, the application should write to these chunks before reading them.

(A4) A `c_malloc` should not return the same object twice until that chunk is released first with a call to a `c_free` function.

(A5) 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.

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 V 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.

(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).

(D3) When a c_free releases a memory object, a c_malloc may return it on future application's requests.

(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.

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.

(R2) Like deallocation functions (D1), c_realloc functions also access the metadata used by a c_malloc.

(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.

(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.

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, we permit such exceptions as long as they are *rare*. However, in practice, we did not come across them.

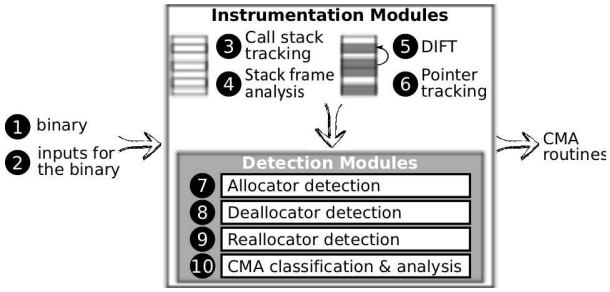

Fig. 1. MemBrush: high-level overview.

III. A BIRD'S EYE VIEW OF MEMBRUSH

We now discuss the CMA detection procedure. MemBrush consists of *instrumentation* modules and *detection* modules (see Figure 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 [23] 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`.

We implemented MemBrush using Intel's Pin dynamic binary instrumentation framework [12]. Pin provides a rich API to monitor context information, e.g., register or memory contents, on program instructions, function- and system calls.

The main components of Figure 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 [24].
- **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. [2].
- **Partial reconstruction of physical stack frame:** ④ To analyze CMA routines, MemBrush needs to identify stack-based procedure arguments. Like [2], our 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. [25].

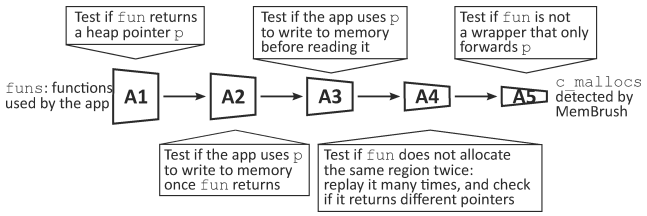

Fig. 2. Detection of c_malloc functions.

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.

- **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 [26]. Like most other DIFT engines [27], we propagate information on direct flows only: we copy tags on data move operations, or them on ALU operations, and so on. 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. [2] 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 II-B, 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.

IV. Custom Allocator Detection

To detect c_malloc routines, MemBrush searches for functions that match A1-A5 from Section II-B. 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.

MemBrush starts by identifying a crude set of c_malloc candidates, i.e., functions that return pointers referencing heap memory regions (A1). While the application executes, MemBrush uses the pointer tracking module ⑥ to track all pointers derived from the addresses returned by the general-purpose memory allocators. 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 tracks all pointers derived from the return value of each c_malloc candidate, and monitors if they are used to write to memory. 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).

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. Since the replay loop might corrupt the state of the application or cause a memory leak, we restart the application after this step. While ensuring to do the replay for every candidate function, MemBrush replays a number of randomly chosen invocations of the candidate.

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 ③ and pointer tracking modules ⑥.

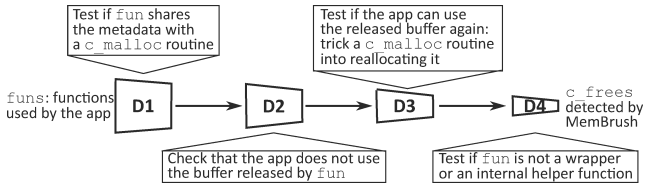

Fig. 3. Detection of c_free functions.

V. 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 II-B. Figure 3 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.

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. 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 IV. 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 IV, 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.

VI. 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 II-B in pipeline-fashion. Figure 4 presents an overview of the 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

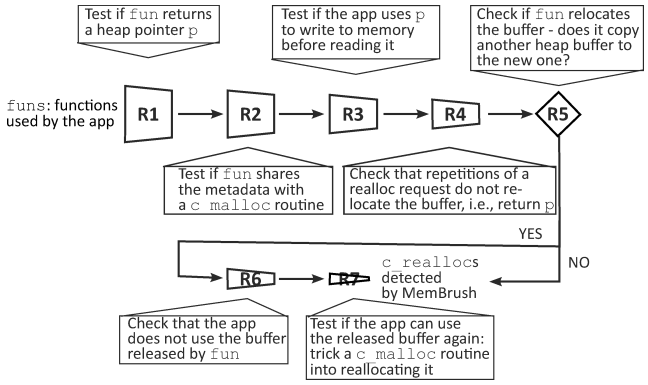

Fig. 4. Detection of c_realloc functions.

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). A simple test could check if 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 IX, 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.

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.

VII. 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 II-A.

A. Buffer Size Estimation

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[1] 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 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 IV). 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.

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 VIII, MemBrush's mechanism yields good results in practice. 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.

[1]We follow exactly the same procedure for c_realloc routines.

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.

TABLE I
MEMBRUSH'S CRITERIA TO CLASSIFY CMAS.

B. 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 V).

Table I 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 VIII-C.

VIII. EVALUATION

In this section, we evaluate MemBrush. We discuss its accuracy (Section VIII-A), present some statistics illustrating the detection procedure (Section VIII-B), and finally we demonstrate the practical benefits of applying MemBrush to an existing binary analysis technique for reversing data structures (Section VIII-C).

A. 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.

Application	Allocators		Deallocators		Reallocators	
	TPs	FPs	TPs	FNs	TPs	FNs
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	-

TABLE II
THE ACCURACY OF MEMBRUSH'S ALGORITHM. THE TOP PART OF THE TABLE REPORTS THE RESULTS FOR 5 REAL-WORLD APPLICATIONS, AND THE BOTTOM ONE — FOR THE SPECINT 2006 BENCHMARKING SUITE.

The accuracy of the CMA routines detection. Table II 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 [28]). Additionally, we applied MemBrush to the SpecINT 2006 benchmarking suite. 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 (TPs), and the number of false positives (FPs).

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 [29], 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

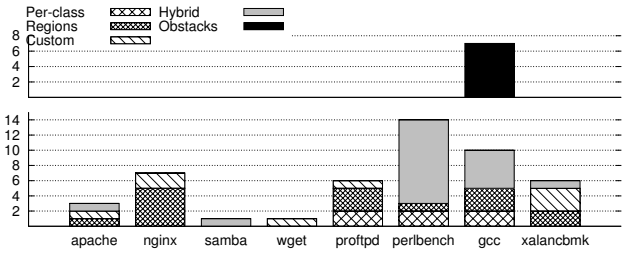

Fig. 5. The accuracy of MemBrush's procedure to classify CMA routines. The bottom part of the graph presents the allocators that were classified correctly, and the top one summarizes misclassifications.

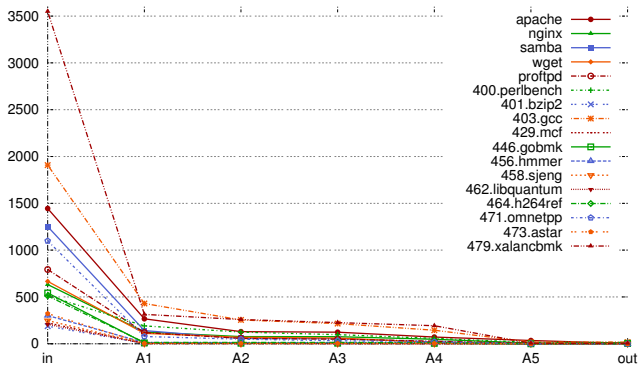

Fig. 6. The number of allocator candidates analyzed by MemBrush when verifying characteristics A1-A5. In Apache, there are 35 functions after the A5 step, and as they belong to different shared libraries, they map to 3 functions in the `libapr/libapr-util` libraries.

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. [25]. 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 5 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 VII-B). 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, 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.

B. 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. Figure 6 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.

C. 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 [2] 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 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` [30], a hierarchical, reference counted memory pool system. MemBrush detects two CMA routines: the `__talloc()` allocator and the `__talloc_free()` deallocator. Table III 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.

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%

TABLE III
THE ACCURACY OF THE DATA STRUCTURE ANALYSIS WITHOUT AND WITH MEMBRUSH'S DETECTION OF CMA FUNCTIONS.

IX. 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) [29], 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 VIII-A, this is not always enough. However, we could extend our technique as proposed by ElWazeer et al. [25].

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.

X. RELATED WORK

Custom memory allocation is a mature field. Many real world applications use CMAs, typically to improve runtime performance. Well-known examples include the Apache and Nginx webservers, the gcc compiler, among many others.

Many research projects, like [31]–[34], propose new memory managers designed for low overhead, and high-performance memory allocation. Other approaches, e.g., DieHard [35], Hound [36] and Cling [37], use custom memory managers tailored to improve the memory safety of applications using them. They help mitigate heap corruptions, dangling pointers or reads of uninitialized data.

Many approaches that detect buffer overflows, use-after-free or double-free attacks [6]–[11] 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.

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

XI. 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

[1] C. Jung and N. Clark, "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-42, 2009.

[2] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *Proc. of the 18th Annual Network & Distributed System Security Symposium*, ser. NDSS'11, 2011.

[3] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proc. of the 17th Annual Network and Distributed System Security Symposium*, ser. NDSS'10, 2010.

[4] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 binary executables," in *Proc. Conf. on Compiler Construction*, ser. CC'04, 2004.

[5] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *CC'08/ETAPS'08: Proc. of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, 2008.

[6] "Valgrind." [Online]. Available: http://valgrind.org

[7] B. Perence, "Electric Fence." [Online]. Available: http://perens.com/FreeSoftware/ElectricFence

[8] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *1992 Winter USENIX Conference*, 1992.

[9] D. Dhurjati and V. Adve, "Efficiently Detecting All Dangling Pointer Uses in Production Servers," in *Proc. of the International Conference on Dependable Systems and Networks*, ser. DSN'06, 2006.

[10] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proc. of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA'12, 2012.

[11] A. Slowinska, T. Stancescu, and H. Bos, "Body Armor for Binaries: preventing buffer overflows without recompilation," in *Proc. of USENIX Annual Technical Conference*, ser. USENIX ATC'12, 2012.

[12] Intel, "Pin - A Dynamic Binary Instrumentation Tool," http://www.pintool.org/, 2011.

[13] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," 1995.

[14] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Reconsidering custom memory allocation," in *Proc. of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA'02, vol. 37, no. 11, 2002.

[15] The FreeBSD Project, "FreeBSD Kernel Developer's Manual. ZONE(9)." [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=uma

[16] M. T. Jones, "Anatomy of the Linux Slab Allocator," 2007. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/

[17] D. T. Ross, "The AED free storage package," *Communications of the ACM*, vol. 10, no. 8, 1967.

[18] D. R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," *Software: Practice and Experience*, vol. 20, no. 1, 1990.

[19] The Apache Software Foundation, "Developing modules for the Apache HTTP Server 2.4." [Online]. Available: http://httpd.apache.org/docs/2.4/developer/modguide.html

[20] The PostgreSQL Global Development Group, "PostgreSQL 9.2.4 Documentation. Section 43.3. Memory Management." [Online]. Available: http://www.postgresql.org/docs/9.2/static/spi-memory.html

[21] nginx, "nginx documentation." [Online]. Available: http://nginx.org/en/docs/

[22] "The GNU C library. Obstacks." [Online]. Available: http://www.gnu.org/software/libc/manual/html_node/Obstacks.html

[23] Doug Lea, "A Memory Allocator." [Online]. Available: http://g.oswego.edu/dl/html/malloc.html

[24] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in vivo multi-path analysis of software systems," in *Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'11, 2011.

[25] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable Variable and Data Type Detection in a Binary Rewriter," in *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'13, 2013.

[26] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems." in *Proc. of the 8th Annual International Conference on Virtual Execution Environments*, ser. VEE'12, 2012.

[27] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an Emulator for Fingerprinting Zero-Day Attacks," in *Proc. of the 1st ACM European Conference on Computer Systems 2006*, ser. EuroSys'06, 2006.

[28] Lockless, "Lockless Performance." [Online]. Available: http://locklessinc.com

[29] G. Balakrishnan and T. Reps, "WYSINWYX: What You See Is Not What you eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 23:1–23:84, 2010.

[30] Samba, "talloc Documentation." [Online]. Available: http://talloc.samba.org/talloc/doc/html/index.html

[31] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *Proc. of the 2006 Intl Symposium on Memory Management*, ser. ISMM'06, 2006.

[32] A. Jula and L. Rauchwerger, "Two memory allocators that use hints to improve locality," in *Proc. of the 2009 International Symposium on Memory Management*, ser. ISMM'09, 2009.

[33] R. Liu and H. Chen, "SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability," in *Proc. of the 3rd ACM Asia-Pacific Workshop on Systems*, ser. ApSys'12, 2012.

[34] S. Lyberis, P. Pratikakis, D. S. Nikolopoulos, M. Schulz, T. Gamblin, and B. R. de Supinski, "The myrmics memory allocator," in *Proc. of the 2012 Intl. Symposium on Memory Management*, ser. ISMM '12, 2012.

[35] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *Proc. of the 2006 ACM Conference on Programming language design and implementation*, ser. PLDI'06, 2006.

[36] G. Novark, E. D. Berger, and B. G. Zorn, "Efficiently and precisely locating memory leaks and bloat," in *Proc. of the 2009 Conference on Programming language design and implementation*, ser. PLDI'09, 2009.

[37] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proc. of the 19th USENIX conference on Security*, ser. SSYM'10, 2010.