Metrics for runtime detection of allocators in binaries

Franck de Goër
Univ. Grenoble Alpes

Laurent Mounier
Univ. Grenoble Alpes

Roland Groz
Univ. Grenoble Alpes

Abstract

Memory management in a binary can be handled by a standard allocator (*e.g.* the `libc` allocator) or by a custom one. For many security and safety analysis focused on memory, the knowledge of the allocator is a requirement. In this paper, we propose an approach to retrieve allocators in binaries, based on heuristics and one single execution, with a scalable instrumentation. In addition, we propose a metric to evaluate the consistency of the detected allocator, in order to confirm or invalidate the result. Finally, we propose an open-source implementation and repeatable experiments. Preliminary results show that our approach allows to successfully retrieve the standard `libc` allocator in `coreutils` programs plus in `mupdf`, `pdflatex` and `readelf` ; and the custom embedded allocator on `jasper`. They also confirm the relevance of our metric for consistency on these examples.

1 Introduction

Many harmful program defects and security vulnerabilities are due to memory errors occurring in the heap. Classical examples are *heap overflow* and *use-after-free* vulnerabilities. Both static and dynamic existing techniques allowing to detect such heap related issues heavily rely on some *a priori* knowledge of memory allocation and liberation functions. However, numerous real-life applications now embed their own allocators, and retrieving the corresponding function pair could be quite challenging, in particular when the source code is not (fully) available.

We propose in this paper a dynamic analysis technique to identify such a function pair, in a single execution, performed on an instrumented version of a target program. This technique can operate on a stripped binary, and is scalable to large programs such as pdf readers (`mupdf`). We also propose a metric to evaluate the consistency of the supposed allocator, in the form of an error rate that should not be over a given threshold. The efficiency of our approach has been validated on a set of programs using the standard `libc` library (`coreutils` programs and `mupdf`, `pdflatex` and `readelf`), plus on a program embedding a custom allocator (`jasper`).

Although retrieving allocators is a challenging problem, and although solving it would have a positive impact on the security of computer systems (as it would allow to apply vulnerability detection techniques to programs embedding their custom allocators), it has not been subject to many researches in the recent years. The most advanced work on allocator detection is [3]. We differentiate from this work on at least three points. First, they need to track the first allocation an allocator performs (either through a system call - `mmap` or `brk`, or through `malloc`) ; second they perform *active* instrumentation, *i.e.* they modify the program behavior to perform analysis (for instance, they replay calls to the supposed allocating function to ensure it does not output the same value twice), whereas we only perform *inspection*, *i.e.* we only observe the program execution with no wanted interference; and third we propose a metric to evaluate the consistency of the inferred allocator to validate (or not) the result. In addition, we provide an open-source implementation and open-source benchmarks.

This paper is organized as follows: Section 2 states the problem we address. Section 3 gives the heuristic criteria we propose, and Section 4 presents the metric we use to validate the results. Details on the implementation are given in Section 5 and experimental results are shown and discussed in Section 6. Section 7 proposes a discussion on our approach and its current limits. Section 8 concludes.

2 Problem Statement

This section describes in details the problem we are addressing in this paper, and in particular its frame, possible applications, and the general idea of our approach.

2.1 Frame

We aim to target **binary** programs, *i.e.* the product of a compilation from source code or hand-written instructions. In the context of our analysis, the **source code is not available**. In the security field, it is a common choice for an approach to be as general as possible. Indeed, this corresponds to real-life scenarios where the program under analysis is a commercial closed-source product or a malware, for examples.

Another important point is that **we do not rely on the symbol table nor the string table** to perform our analysis. In other words, we propose in this paper an approach that can be applied to **stripped binaries**. This is an important point, as a lot of commonly used programs are stripped in order to minimize their static size. In addition, commercial or malicious programs usually do not embed a symbol table.

2.2 Main issues we address

The general problem we address in this paper is to analyze, from one single execution, the memory allocation of a given binary. More precisely, we aim to **detect the top-level allocator** main functions, that we generically denote by `ALLOC` and `FREE`. We present a method in this paper which shows that such an analysis can be performed with a lightweight and scalable instrumentation. We also aim to **propose a consistency criterion to validate the detection**.

2.3 Applications

The applications of this can be manifold. For instance, a lot of approaches to detect memory bugs or vulnerabilities rely on the knowledge of the allocator ([8], [2]). In the case of a custom allocator, it needs to be retrieved before being able to perform such analysis. A typical example is `Valgrind` [13]: to run and perform verification on memory use, it needs to know `ALLOC` and `FREE`.

Another interesting application would be to analyze a large set of binaries, identify the ones that are not using a well-known allocator, in order to automatically generate a dataset of programs using their own embedded one. Such a dataset does not exist yet to our knowledge.

2.4 Proposed approach

Our approach is divided in two parts: the detection of allocators, and the validation of this detection.

2.4.1 Detection

We propose in this paper an heuristic-based approach that we present in Section 3. It consists in two phases: a first **online step**, during which we instrument an execution of the binary under analysis and collect data (see section 5.2); and a second **offline step** which, using the data collected, performs analysis to retrieve the allocating and liberating functions of the given program.

2.4.2 Validation

To validate our results, we propose a metric - see Section 4. The general idea is to run other instrumented executions of the binary, and to perform consistency checks based on some classic properties of allocators. The number of errors (we define an error as a violation of one of these properties - see Section 4.1) compared to the number of calls to `ALLOC` and `FREE` gives a metric to evaluate the likeliness of the supposed allocator: if the error rate is too high, we conclude on the inconsistency of the result of detection; otherwise we validate it. Typically, an unallocated block being freed is not consistent with what is expected from an allocator: this is one kind of errors we consider.

2.4.3 Keystone: functions

The approach we propose to implement the detection of `ALLOC` and `FREE` is based on functions. We track parameters of functions, as well as return values, but we do not consider variables and instructions that happen inside a function, except for sub-calls to other functions. This allows to have a scalable instrumentation which can be applied to large programs, or to a large dataset of programs, in a reasonable time - see Section 6.2.

3 Criteria

In this section, we present the main criteria we use to differentiate the allocator functions (`ALLOC` and `FREE`) from the collection of functions a binary embeds.

3.1 Retrieve `ALLOC`

3.1.1 Production of addresses

Observation – The main functionality of `ALLOC` is to *allocate memory*, *i.e.* to output new blocks of memory that were unallocated before. One way or another, `ALLOC` must provide information about the block that has been allocated.
Assumption – We assume that this functionality is implemented using addresses, and that it is provided through a return value: `ALLOC` outputs an address as a return value, pointing to the new allocated block.
Criterion – From this, we consider that `ALLOC` is a function that produces a high number of new addresses, without taking addresses as parameter. We call this criterion

the *production criterion*.
Discussion – What we call a new address, here, is an address that was never seen before as a parameter of function or as a return value. The intuition behind this criterion is that it is specific to ALLOC to forge new addresses *from scratch*; other functions might return new addresses, but obtained by deriving from existing ones (given as parameters). For example, a function to search an element in an array and that outputs a pointer to the cell that contains it would produce *new addresses* (in the way we define it), but this address would be computed from the base address of the array that is likely to be passed as a parameter. Two other points should be addressed. First, this criterion is consistent with the aim to retrieve *the allocator* and not wrappers (wrappers will not produce new addresses, but more likely output addresses that were returned by the allocator after some sanitizing and/or safety/security checks). However, the second criterion we present in the next subsection can lead to detect a wrapper instead of the allocator (this will be discussed). Second, this criterion will not be relevant to retrieve allocators that do not output addresses as return values *e.g.*, if the address is output through an out parameter or a global variable. The problem of detecting allocators using output parameters is addressed as a future work.

3.1.2 Diversity of callers

Observation – An allocator proposes a high-level functionality useful in many contexts in programming. Its usage is not restricted to particular cases.
Assumption – The top-level allocator is either called directly by functions which need to allocate memory, or by sub-allocators. In the later case, there are multiple sub-allocators calling the top-level allocator (the purpose of a single sub-allocator is doubtful).
Criterion – ALLOC is a function that is called by multiple functions. We call this criterion the *diversity criterion*.
Discussion – This criterion aims to distinguish the top-level allocator interface (*i.e.*, the function being called by other functions in the program) from internal functions of the allocator which are not accessible outside ALLOC. For instance, ALLOC could use an internal function to choose the block to allocate among possible blocks, and then perform some management of the non-allocated memory (*e.g.*, merging free blocks), possibly initialize the block to allocate, and finally return it. In this scenario, we do not want our approach to output the function that chooses the block to allocate (which is likely to output the address of the block to be allocated). As this function is internal, it would not be called by many other functions, so the *diversity criterion* will prune it.

Nevertheless, and as mentioned earlier, this criterion can lead to detect a wrapper instead of the ALLOC function. Indeed, if ALLOC is always called through a wrapper, this criteroin will not be satisfied and the output would be the wrapper itself. In this case, a manual analysis could be required to retrieve ALLOC from thi wrapper.

3.2 Retrieve FREE

3.2.1 Treatment of addresses produced by ALLOC

Observation – The main functionality of FREE is to *free memory* allocated by ALLOC, *i.e.*, to mark blocks as unallocated anymore. These blocks can be then re-allocated. FREE needs to know what block should be freed.
Assumption – We assume that this data is given through a parameter to the FREE function, as the address of the memory block to be freed. This address is assumed to be the one output by ALLOC.
Criterion – FREE is a function that takes, as a parameter, a high number of values that were output by a previous call to ALLOC.
Discussion – This assumption supposes that FREE takes an address as a parameter ; but, as mentioned in Section 3.1.2 relatively to the *production criterion* for ALLOC, FREE could use a global variable, for instance, to get the address of the block to be freed. We could also imagine a stack allocator, where the block to be freed is the last one that was allocated: in this scenario, there is no need for FREE to get any parameter at all. However, in many cases, it seems reasonable to assume that the freeing function takes an address as a parameter corresponding to the block to be freed.

3.2.2 Last accessor of memory blocks

Observation – After a memory block has been freed, and before it is re-allocated, it should not be accessed anymore, except in case of implementation bugs (*use-after-free*).
Assumption – We assume that bugs are rare, and that in the general case freed blocks are not accessed before re-allocation.
Criterion – Considering an allocated block, FREE is the last function to access it before the end of the execution or before a prefix of the same block is re-allocated by a further call to ALLOC. We call this criterion the *last accessor criterion*.
Discussion – Note that, by *last accessor*, we mean that it is the last function to take as a parameter or to output this address. This means, in particular, that some *use-after-free* bugs will not be a problem for our detection: if the use is performed in the core of the function calling FREE, or more generally if the dangling pointer to the freed block is never given as a parameter to a function after the call to FREE, then we will not consider it, and

FREE will still be the last accessor of the block. Furthermore, this heuristic implies to evict internal functions of the allocator that might access the freed block to manage unallocated memory. This is why we need to introduce the *diversity criterion* once again, this time for FREE.

Finally, note that this asumption does not keep us from detecting *use-after-free* bugs once FREE is retrieved.

3.2.3 Diversity of callers

As presented in section 3.1.2, we use the same criterion of diversity of callers to retrieve FREE. This criterion is even more important here than it was in section 3.1.2, because FREE often calls an internal function to manage free blocks (*e.g.*, to merge consecutive blocks in the free memory area, given the address of the block that was just freed). This internal function should not be considered as FREE ; however with the first two criteria we presented, it would be the last accessor of addresses produced by ALLOC and so give improper results.

4 Metric for Consistency

In addition to the heuristic-based approach to detect ALLOC and FREE from an execution that we presented in the last section, we aim to provide a metric to evaluate the consistency of the results. By consistency, we mean the satisfaction of some basic properties derived from the definition of an allocator during instrumented executions. By tracking violations of these properties, we can output a number of errors relatively to the number of invocations of the supposed allocator's functions, to conclude on the validity of the couple (ALLOC, FREE). The number of additional executions can be from zero (we can perform the validation on the trace we used for detection) to many: the more additional executions there is, the more precise the validation is.

4.1 Properties to check

We propose to check two main properties that an allocator should satisfy. For each property, we count an error each time it is violated. **First**, a FREE should only occur on addresses allocated by ALLOC. This means that every time FREE is called with a parameter that was not output by ALLOC before, we detect an error. **Second**, an ALLOC should not return the same address twice if it was not released by FREE in between. Here again, every time we see an *allocated* address being re-allocated, we count an error.

4.2 Error rate

We compute the error rate as the total number of errors (generated by either ALLOC or FREE) divided by the number of calls to ALLOC plus the number of calls to FREE. If this error rate is below a threshold, we validate the detected allocator. Otherwise, we conclude that the couple (ALLOC, FREE) is inconsistent. Our experiments show that a good value for threshold is 0.05 - see Section 6.2.4.

Note that, among the two properties we presented, one is relative to ALLOC and the other is relative to FREE. However, if one or the other of these functions was inferred incorrectly, it would most certainly lead to errors relative to both. For instance, a misdetection of FREE would lead ALLOC to generate errors, because blocks would not be freed properly (considering the FREE function that was inferred). That is the reason why we only consider the global error rate rather than two error rates (one for ALLOC and one for FREE), and we use it to validate or invalidate the allocator **as a couple**, but one of the two functions can still be correct, and we could use other indicators to invalidate each one individually.

5 Realisation

5.1 Get undertyped prototypes

To perform accurate analysis of functions, we rely on information about their prototypes. We must know, for instance, how many parameters a function takes in order to get their concrete values. In addition, we must know which parameter is an address and which one is not. this is what we call *undertyping*: we do not need to know precisely the type of each parameter, but we need to know if it is an ADDR or a NUM. These data (arity and types) are not explicitly embedded in binary programs (they are lost during the compilation process), and thus they are to be retrieved. To do so, we rely on `scat`, a tool we developed and which implements an approach we presented in [7] to recover undertyped prototypes from a binary with a high accuracy (higher than 95% for arity as well as for undertyped parameters). This recovery is compatible with the same constraints we use in this work (stripped binaries, one execution, scalable), so we can perform the whole analysis on a given binary by combining the allocator detection with `scat`. In fact, for every result we present in Section 6, we start from the binary only and use `scat` to recover prototype parameters before retrieving ALLOC and FREE.

5.2 Online: Instrumentation Using `Pin`

The first step of our approach is to run the binary once, under analysis, using `Pin`. `Pin` is a framework presented in [11] to perform dynamic instrumentation of programs using *just-in-time* compilation. The key idea is to collect concrete values of addresses taken as parameter and/or returned by functions at runtime. To be lightweight, we only instrument relevant functions (*i.e.*, functions that deal with addresses either in input or in output), according to the undertyped prototypes (see Section 5.1). Each call to (and each return from) one of these functions is instrumented in order to log the concrete values of its address parameters (resp. return value). With those concrete values, we store several meta-data used during the offline step:

- `val` concrete value of the parameter
- `fid` identifier of the function being called or returning
- `caller` identifier of the function that called `fid`
- `counter` event counter (strictly increasing)
- `pos` position of the parameter in the function prototype, -1 if it is a return value

At the end of the execution, we store the meta-data for each parameter collected in a binary file. As an example, an execution of `mupdf` leads to a log file of 238MB for eight million events, We consider this is acceptable, however it could be improved using compression techniques and better encoding of the data. We address this as a future work.

5.3 Offline: Retrieve `ALLOC` and `FREE`

The goal of the offline step is, given the logs obtained from the online step (see Section 5.2), to determine which function better fits with the criteria we presented in Section 3.

5.3.1 Exclude (or not) libraries

Memory allocation, during an execution, is usually required from both the main core of the program and from library functions. In the latter case, it is likely that the allocator used is the standard `libc` allocator, and each call to it disturbs the detection of the possibly embedded allocator used by the main core of the program. If there is too much noise, the `libc` allocator would become predominant over the embedded one. For this reason, it might be a good idea to exclude external calls to libraries that occur during the execution. In the case the program is using its own allocator, then it is embedded in the binary. If the program uses an external allocator, calls to external libraries dynamically loaded are handled by the Global Offset Table (GOT) mechanism. This mechanism is triggered by a first (internal) call to the corresponding GOT entry that we catch. In both cases, we can retrieve the function allocating memory in the main program without considering external calls, except if there are not enough allocations in the core of the program to detected it properly. Our implementation allows to choose whether calls to libraries should be considered or not.

5.3.2 Retrieve `ALLOC`

To retrieve `ALLOC`, we compute a `score` for each function, relatively to the criteria we presented in Section 3.1, with two rules.

Diversity – *A function called by only three or less different callers has a score of 0.* The diversity criterion aims to eliminate internal functions of an allocator. We assume that a classical allocator interface has three functions: `ALLOC`, `FREE` and `REALLOC`. The threshold of three should filter internal functions only called by one of these three.

Production – *Otherwise, the score of a function is the number of addresses it produces.* We consider that a function ”produces” an address if it outputs an address that we never saw before, neither as a parameter nor as a function output. This is a heuristic: a function could ”forge” a new address by adding an offset to an existing one (and this will not be the result of an allocation).

Finally, `ALLOC` is the function with the highest score, according to the log of the execution. This approach, highly based on heuristics and approximation, gives encouraging results in our early experiments, as presented in Section 6.

5.3.3 Retrieve `FREE`

To retrieve `FREE`, we rely on `ALLOC` we obtained from Section 5.3.2. Here again we use two rules to compute the score of each candidate.

Diversity – This is the exact same rule as for `ALLOC`: *a function called by only three or less different callers has a score of 0.*

Last accessor – *Otherwise, a function score is the number of times it is the last accessor of an address produced by* `ALLOC`, before the end of the execution or before a new `ALLOC` producing the same address. By last accessor, we mean the last function to take the given address as a parameter.

`FREE` is the function with the highest score according to those rules and the log. At this point, we end up with a couple (`ALLOC`, `FREE`).

6 Experiments

We present in this section results obtained during our experiments. As mentioned before, our approach is based on heuristics and should be, in a future work, subject to stronger tests to be validated. However, there is a lack of available benchmark in this domain, and although one can find some examples of programs that do not use the standard `libc` allocator in [3], [1] and [14], it is a heavy task to find open-source programs using custom allocators that are explicitly documented. Our approach can be used to automatically construct such benchmarks, by detecting among a large set of binaries, the ones that does not use the standard `libc` allocator. This will be addressed in a future work. Note that our implementation is open-source and available at [6].

6.1 Benchmark

Note that all programs of our benchmarks have been compiled using `gcc 5.4` using the provided `Makefile`.

6.1.1 `libc` allocator

Is our approach able to retrieve a standard allocator in various situations?

We propose to use, as a benchmark for our approach, programs from the `coreutils` library [9]. These programs are open-source, lightweight and widely used. In addition, they have different purposes, and therefore provide multiple cases of memory allocation. They all use the `libc` standard allocator (`malloc`, `free`). We also test our detection on three larger programs that use the `libc` allocator: `mupdf`, `pdflatex` and `readelf`.

6.1.2 Custom allocator using `LD_PRELOAD`

Is our approach able to detect allocators that behave differently from the standard `libc` allocator?

As mentioned earlier, custom allocator detection suffers from a lack of benchmark. To get round of this, and as preliminary experiments, we work on programs using the `libc` allocator, except that we hook calls to `malloc` and `free` using `LD_PRELOAD` and replace the standard `libc` allocator by a custom one we wrote. The consequence is that when the program under analysis calls `malloc`, it actually calls *our* custom allocation instead. This approach is tested on the same benchmark as the one presented in the previous section, however it faces, in some cases the problem of static linking. `LD_PRELOAD` is used to hook functions dynamically loaded, but it cannot interfere with static libraries ; so if the program under analysis or its libraries use static linking to `malloc` and `free` massively, our custom allocator is only called a few times (the calls corresponding to dynamic linking).

6.1.3 Custom embedded allocator

Is our approach able to detect embedded custom allocators?

We propose one experiment to test our criteria on a custom allocator embedded in a program: `jasper`. `Jasper` is a `C` program to convert images from one format to another. This experiment will be used as an example to present outputs of the program and to emphasis the end-to-end approach - see Section 6.2.3. This program is particularly interesting in the frame of this work, as *use-after-free* vulnerabilities have been reported (see CVE-2016-9591, CVE-2016-9262 and CVE-2015-5221 for instance), and these vulnerabilities rely on the knowledge of the custom allocator. In these cases, it has been retrieved manually, but this emphasis the fact that an automated approach would be helpful to detect memory bugs and vulnerabilities.

6.1.4 Evaluation of our metric

Is our consistency criterion relevant to distinguish allocators from other couples of functions?

To evaluate the metric for consistency we proposed in Section 3, we compute the error rate for several couples of functions in addition to the couple (`ALLOC`, `FREE`) our approach outputs. For possible `ALLOC`, we keep each function that produces at least one new address during the execution ; and for each of these potential `ALLOC`, we consider for possible `FREE` the three functions that are most often the last accessors of addresses produced by `ALLOC`. We perform this experiment on the same benchmark as the one presented in Section 6.1.1, *i.e.* programs using the standard `libc` allocator. Among these couples, we thus have the real allocator (`malloc`, `free`) but also other couples that are half allocators (only one function is correct) or not allocators at all. For each of these couples, we compute the error rate.

6.2 Results

6.2.1 General results on allocator detection

Table 1 presents the results of our detection on `mupdf`, `pdflatex`, `readelf` and some programs of the `coreutils`. For each of these programs, we present the time of execution of the program under analysis to give an idea of the scalability of our implementation. The second column tells if we manage to retrieve successfully (✓) or not (×) the standard `libc` allocator. We also provide the error rate relative to the detection (between 0 and 1). These experiments show two points: first, our approach is able to retrieve the standard `libc` allocator in every case except `uname` where it fails to retrieve `free` ; and second, the error rate is significantly under the threshold of 0.05 each time the correct allocator is detected, and significantly above it in the case of `uname` (0.389).

program	time (in s)	ALLOC/FREE	error rate
mupdf	12.91	✓/✓	8.22×10^{-2}
pdflatex	56.26	✓/✓	5.98×10^{-4}
readelf	3.21	✓/✓	3.60×10^{-4}
base64	0.882	✓/✓	0.00
cat	0.372	✓/✓	0.00
cp	2.852	✓/✓	3.22×10^{-4}
head	0.477	✓/✓	0.00
id	0.611	✓/✓	0.00
paste	0.930	✓/✓	0.00
pinky	0.631	✓/✓	0.00
tail	0.478	✓/✓	0.00
uname	0.388	✓/×	3.89×10^{-1}

Table 1: Detection of ALLOC and FREE on mupdf, pdflatex, readelf and several coreutils programs which use libc allocator

program	#calls	#errors	rate
mupdf	23273/26650	191/3913	8.22×10^{-2}
pdflatex	573492/278652	137/373	5.98×10^{-4}
readelf	4182/4131	0/3	3.60×10^{-4}
base64	168/86	0/0	0.00
cat	156/112	0/0	0.00
cp	10877/10832	3/4	3.22×10^{-4}
head	166/84	0/0	0.00
id	255/100	0/0	0.00
paste	270/226	0/0	0.00
pinky	247/97	0/0	0.00
tail	169/87	0/0	0.00
uname	47/48	2/35	3.89×10^{-1}

Table 2: Error rate on mupdf, pdflatex, readelf and several coreutils programs which use libc allocator

6.2.2 Using LD_PRELOAD

We successfully retrieve our custom allocator, using the LD_PRELOAD hook, for the three main programs we tested: mupdf, readelf and pdflatex ; plus on cp and cat. However, on the majority of coreutils programs we tested, the number of dynamic calls to malloc and free is too low to be able to detect an allocator. In these case, no couple is output (as no good candidate is found). In a few words, when there are enough calls to a custom allocator, our approach is able to retrieve it, otherwise it does not output a wrong couple.

6.2.3 Jasper

For this particular experiment, we provide the trace of each step on GitHub - see [5]: we are able to retrieve the custom allocator embedded in jasper, from the binary only, and with an end-to-end approach. For the purpose of the demonstration, the binary is not stripped, although it would work the same on a stripped version. First, we infer undertype prototypes of functions used by jasper. Note that during this step, the program is actually executed twice (one for arity and once for types). Then, from the undertyped prototypes, we retrieve the custom allocator embedded in jasper: (jas_malloc, jas_free). Finally, the consistency of the result is checked:, we end up with 6 errors in 204 calls, that is an error rate of 2.94×10^{-2}.

6.2.4 Consistency criterion

Table 2 gives more details about the experiments on the detection of the standard libc allocator of Table 1. In particular, it presents, for each program we tested, the number of calls to the supposed ALLOC and FREE functions, the number of detected errors of each kind and the error rate. As mentioned in Section 6.2.1, the error rate on a high number of calls is lower than our threshold (0.05) for each valid inference. In the case of uname, note that the number of calls to the allocator is very small (less than 50), but our consistency criterion still emphasis errors that lead to the conclusion the result is incorrect (and it is indeed - see Table 1).

In addition, Figure 1 presents the results relatively to the relevancy of our consistency criterion. We recall that for each tested program, we compute the error rate on several couples that could be the allocator (including the good one). Note that we plot consistency rate (*i.e.*, 1 - error rate) for more visibility. This experiment shows that, for each program except cat, only two couples have a consistency rate above 0.95, and very few above 0.8. After a manual evaluation, it appears that, in each of these cases, the second couple with a low error rate is the *Global Offset Table* entry corresponding to malloc coupled with free. In the case of cat, the third couple above the threshold is the couple (xmallox, free) which is also a valid allocator ; however, since it is called less times that the couple (malloc, free), it is considered lower in the hierarchy of interesting allocators. We also note that for uname, no allocator is found, and the best candidate has a consistency rate of about 0.6.

7 Discussion

7.1 Validity

Our experiments show that, in various scenarios, the criteria we propose to detect allocators are relevant. We detect the correct allocator in multiple cases: when the standard libc allocator is used, when we replace it with a custom allocator of our own, and on custom allocators

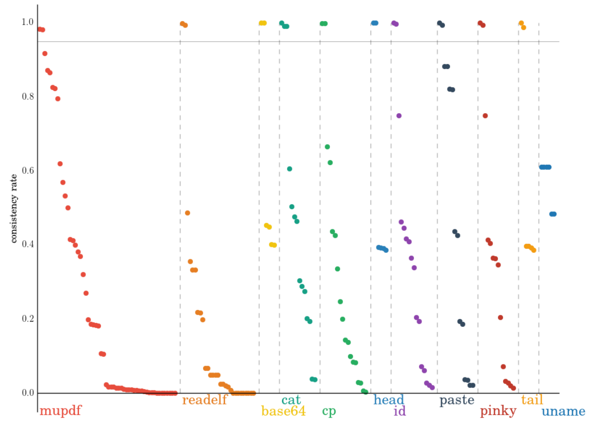

Figure 1: Evaluation of the consistency rate on multiple candidates (ALLOC, FREE)

as well. They also show that our consistency criterion is able to emphasis errors in the allocator detection: for every correct detection, the error rate is significantly below 0.05, and for incorrect couples it is significantly above.

Furthermore, it shows that the instrumentation is scalable, and can be applied to common programs or to a large set of (small) programs.

7.2 Limitations

As illustrated by `uname`, our approach fails to detect the allocator correctly for some programs of `coreutils`. Actually, `malloc` is retrieved successfully in any case, but the detection of `free` is sometimes inaccurate. This is because we miss some internal calls in libraries with our instrumentation. Indeed, in Table 2, we observe that in many cases, the number of calls to `ALLOC` is significantly higher than the number of calls to `FREE`. An execution instrumented by `Valgrind` confirms this: for both `malloc` and `free`, we end up with less calls than `Valgrind` on several examples. However, this is a limitation of the implementation using `Pin` we propose, not a limitation of our approach.

8 Conclusion

In this paper, we proposed criteria to detect allocators, and a metric to evaluate, for a given couple of functions, if it is an allocator. Our experiments show that these criteria, as well as our metric, are relevant: we were able to retrieve allocators in multiple cases, and our metric successfully differentiated allocators from other couples of functions. Furthermore, our approach for both detection and validation can be applied to a binary program from end to end, and requires no more information. We also showed that it is scalable. Finally, we publish an open-source implementation [6] as well as our test-cases. Memory allocators have been studied ([14], [1], [12], [10]), but few works have been done to retrieve them, apart from [3] as discussed in Section 1. The retrieved allocator can then be used by other analyzers to detect memory errors or security vulnerabilities in the heap. Future work will essentially extend and consolidate our heuristics and our metric by experimenting on a larger set of benchmarks.

References

[1] Berger, E. D., Zorn, B. G., and McKinley, K. S. Oopsla 2002: Reconsidering custom memory allocation. *ACM SIGPLAN Notices 48*, 4S (2013), 46–57.

[2] Caballero, J., Grieco, G., Marron, M., and Nappa, A. 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* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 133–143.

[3] Chen, X., Slowinska, A., and Bos, H. Who allocated my memory? detecting custom memory allocators in c binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)* (2013), IEEE, pp. 22–31.

[4] de Goër, F. Custom memory allocator written in c. `https://github.com/Frky/memalloc`.

[5] de Goër, F. Output of the detection of cutsom allocator from end to end in `jasper`. `https://gist.github.com/Frky/297f1d3cb11f555006581359cbd83b84`.

[6] de Goër, F. `scat`: Dynamic analysis of binary programs. `https://github.com/Frky/scat`.

[7] de Goër, F., Groz, R., and Mounier, L. Lightweight heuristics to retrieve parameter associations from binaries. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (2015), ACM, p. 5.

[8] Feist, J., Mounier, L., and Potet, M.-L. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques* (2014).

[9] GNU. Gnu core utilities. `https://www.gnu.org/software/coreutils/coreutils.html`.

[10] Grunwald, D., and Zorn, B. Customalloc: Efficient synthesized memory allocators. *Software: Practice and Experience 23*, 8 (1993), 851–869.

[11] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40*, 6 (June 2005), 190–200.

[12] Masmano, M., Ripoll, I., and Crespo, A. A comparison of memory allocators for real-time applications. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems* (2006), ACM, pp. 68–76.

[13] Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

[14] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management*. Springer, 1995, pp. 1–116.