Heap Layout Optimisation for Exploitation (Technical Report)

Sean Heelan
University of Oxford
sean.heelan@cs.ox.ac.uk

Tom Melham
University of Oxford
tom.melham@cs.ox.ac.uk

Daniel Kroening
University of Oxford
daniel.kroening@cs.ox.ac.uk

1. Introduction

Over the past decade a number of papers [5], [7], [8], [10] have approached the problem of Automated Exploit Generation (AEG) for stack-based buffer overflows. These papers describe algorithms for automatically producing a control-flow hijacking exploit, under the assumption that an input is provided, or discovered, which results in the corruption of an instruction pointer stored on the stack. However, stack-based buffer overflows are just one type of vulnerability found in software written in C and C++. Out-of-bounds (OOB) memory access from heap buffers is a common flaw in such software and, up to now, has received little attention in terms of automation. Heap-based memory corruption differs significantly from stack-based memory corruption in that with the latter the data which the attacker may corrupt is limited to whatever is on the stack, and thus can be varied by changing the execution path used to trigger the vulnerability. In contrast, for heap-based corruption it is the physical layout of dynamically allocated buffers in memory which determines what gets corrupted. Thus, the attacker must reason about heap layout in order to automatically construct an exploit; it is this problem which we address.

To leverage OOB memory access as part of an exploit, an attacker will usually want to position some dynamically allocated buffer D, the OOB access destination, relative to some other dynamically allocated buffer S, the OOB access source.[1] The desired positioning will depend on whether the flaw to be leveraged is an overflow or an underflow, and the control the attacker has over the offset from S that will be accessed. The key point is that the attacker wants to position S and D so that, when the vulnerability is triggered, D is corrupted with the least collateral corruption of other heap-allocated data, and without triggering a page fault. We say a heap layout is optimal if it minimises the distance between the source and destination.

For example, suppose S is a buffer of size x and the flaw is an overflow that writes an attacker-controlled sequence of bytes starting at S and finishing at $S + x + 128$. Furthermore, suppose D is a buffer that contains a function pointer p at offset 0. In order to corrupt p, an attacker must find a way to transition the heap layout into a state such that D is allocated at an address for which $D - S + x \leq 128$.

Allocators do not typically expose an API to allow a user to control relative positioning of allocated memory regions. In fact, the ANSI C specification [2] explicitly states

> *The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified.*

The second major difficulty is that applications that use dynamic memory allocation typically do not expose an API allowing an attacker to directly interact with the allocator in an arbitrary manner. An exploit developer must first discover the allocator interactions that can be indirectly triggered via the application's API, and then leverage these to solve the positioning problem. In practice, both problems are usually solved manually or with minimal partial automation, and this requires expert knowledge of the internals of both the heap allocator and the application's use of it.

In this technical report[2], we first focus on the positioning problem under the assumption that one knows the interactions with the allocator that can be triggered. There is no existing analysis of the complexity of this problem, and no prior empirical work indicating the difficulty of solving real instances automatically. To establish a baseline approach, therefore, we present a pseudo-random black box algorithm that searches for an optimal layout. This has the merit of being practically and theoretically straightforward, and provides a comparison point for future work on more sophisticated solutions. We run this algorithm on a set of carefully-designed synthetic benchmarks, in order to provide an analysis of the factors that are most significant in determining the difficulty of individual instances.

We then present a proof-of-concept system for fully automatic heap layout optimisation (HLO) on the `PHP` language interpreter. This system first maps the application's interface to allocator-interaction sequences. It then uses the above search algorithm using the API exposed by the application. We evaluate the system on three vulnerabilities in `PHP`, showing that—despite its simplicity—the search algorithm can solve real-world instances[3] .

1. Henceforth, when we refer to the 'source' and 'destination' we mean the source or destination buffer of the overflow or underflow.

2. This document is a technical report where we have aimed to present the core techniques and evaluation in a relatively succinct manner. The associated research paper is under review and will be available at https://sean.heelan.io/research/ under 'Heap Layout Optimisation for Exploitation'.

3. For the source code to the evaluation systems see 'Heap Layout Optimisation for Exploitation' at https://sean.heelan.io/research/.

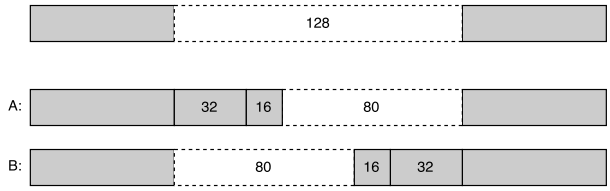

Figure 1. Splitting a block of size 128 on a request for an allocation of size 32 followed by an allocation of size 16. Allocations can be carved from the beginning or the end of free blocks, resulting in two different heap layouts, labelled as 'A' and 'B'. In outcome 'A' the allocation of size 16 ends up after the allocation of size 32 as allocations are carved from the start of the free area, while in the outcome labelled 'B' the allocation of size 16 ends up before the allocation of size 32 as allocations are carved from the end of the free area.

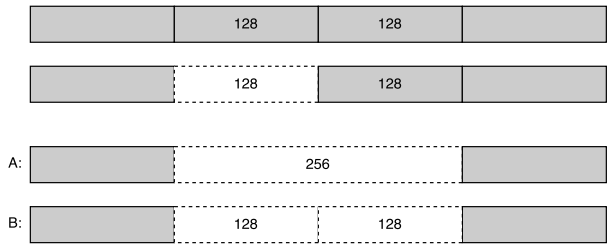

Figure 2. Hole creation via free. The outcome labelled 'A' results from an allocator with immediate coalescing, where a hole of size 256 is produced from two frees of size 128. The outcome labelled 'B' results from an allocator with delayed coalescing, where two holes of size 128 are produced from two frees of size 128.

2. Heap Allocation

Heap allocators are libraries responsible for servicing dynamic requests for memory allocation and typically expose an interface that complies with the ANSI C [2] specification for the functions `malloc`, `free`, `realloc` and `calloc`. As mentioned in section 1, the specification imposes no restrictions on '*the order and contiguity of storage allocated by successive calls*' to these functions. Thus, the developers of an allocator can effectively choose any combination of data structures and algorithms in their implementation to achieve the best results across whatever measure of success they choose, be it minimising fragmentation, maximising speed, increasing security, or some other metric entirely. In the following, we elaborate on those design choices which have the most impact when it comes to manipulating the heap into a desired state, or which are otherwise significant in differentiating allocators.

2.1. Relevant Allocator Policies and Mechanisms

2.1.1. Splitting. Allocators may or may not split blocks of memory to fulfil a request for a smaller block. When blocks are split, the area to use for the allocation may be carved from either the beginning or the end of the existing block.

For example, both `dlmalloc` and `avrlibc` will split blocks, but the former carves from the front, while the latter carves from the end. If carving takes place from the front then there is space to place another block *after* the returned block, while if carving takes place from the end then there is space to place another block *before* the returned block. In contrast with both of these, `tcmalloc` only splits blocks above 32KB in size.

Figure 1 shows two possible state transitions for the heap state, depending on what form of splitting is used. In the first outcome, labelled 'A', the block of size 32 ends up before the block of size 16, while in the second outcome, labelled 'B', outcome the order is reversed. This is significant if one requires a particular ordering for the blocks resulting from two allocations.

2.1.2. Coalescing. When blocks are freed, and have other free blocks which are immediately adjacent, they may be coalesced into a single, larger, free block. Allocators may or may not do this, or they may utilise an intermediate approach where coalescing is delayed, but will eventually happen if some event occurs, such as the number of uncoalesced blocks exceeding a threshold. If immediate coalescing is used, such as in modern `dlmalloc` and `avrlibc`, then it is not possible to have two free blocks next to each other. Should this occur, they will be immediately coalesced into a larger free block. However, if delayed coalescing is used, such as in early versions of `dlmalloc` and some versions of the `Windows` userland [18] and kernel heaps [16], then it is possible to have multiple free chunks adjacent to each other.

Figure 2 shows two possible state transitions for the heap state, depending on what form of coalescing is used. If the aim is to create holes of size 128 then triggering the second free is either useful or counterproductive, depending on whether or not delayed coalescing is in use.

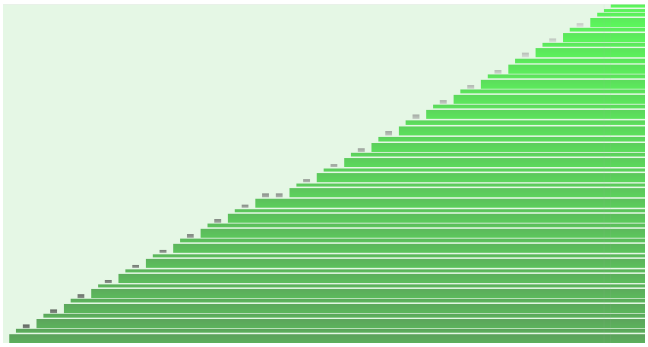

Figure 3. Heap state evolution for `dlmalloc`. The x axis represent time, or more accurately 'ticks' caused by new allocator interactions, while the y axis represents memory addresses from lowest to highest. Green rectangles represent memory regions that are currently allocated. Grey rectangles represent memory regions that were allocated previously but now are free. The height of a rectangle indicates how large the memory region is, while its width represents how long it was allocated for. A light green background represents an area of memory being mapped, while a white background, as can be seen in figure 4, represents an area of unmapped memory.

2.1.3. Fits. When scanning for a suitably sized free block to utilise for an allocation one has several options on where to start the search and on what condition to end it. The most common approach is *best fit* in which the free block with size closest to that of the requested allocation size is guaranteed to be found, split if necessary, and returned. Alternatives include *first fit* and *next fit*. In first fit the first block encountered during the search that is large enough to fulfil the requested allocation is selected, split if necessary, and returned. In next fit the next search for a free block continues in the free list from the last index reached by the previous search. The fit policy in use is relevant to HLO in that it influences which block is selected for a particular size.

2.1.4. Segregated Free Lists. Some allocators, such as `avrlibc`, utilise a single free-list in which all free chunks are stored. While simple to implement, this approach has a significant downside in that if one wishes to utilise a best fit search then the entire list may need to be scanned in order to determine which block to use for an allocation. An alternative approach, as found in modern `dlmalloc`, is *segregated free lists*, in which multiple free lists are used and blocks of the same, or similar, size are kept in the same list. With segregated free lists one can efficiently check to see if a free block of a particular size exists. For the purposes of HLO, a single free list or multiple free lists do not significantly impact the complexity of the problem, or the approach one takes.

2.1.5. Segregated Storage. Segregated storage is a mechanism whereby contiguous areas of memory are set aside for the allocation of blocks of a single size. For example, on the first request for a block of size 32 the allocator might obtain one or more pages from the operating system and subdivide those pages into blocks of size 32. The blocks will then not be further split or coalesced.

Whether segregated storage is in use or not has a significant impact on how one approaches HLO. The impact this single design decision can have on heap layout is highlighted in the contrast between figures 3 and 4. Both figures show the heap state after an identical series of approximately 100 allocator interactions consisting of allocations of three different sizes and a number of frees. Figure 3 shows `dlmalloc`, which does not use segregated storage, and thus we can see that allocations of different sizes end up alternating in memory. Figure 4 shows `tcmalloc`, an allocator which does use segregated storage, resulting in three distinct memory regions, each of which contains allocations of a single size.

2.1.6. Non-Determinism. While most allocators are internally deterministic in their operations, some allocators utilise non-determinism in allocation as part of an effort to make exploitation more difficult. This differs from traditional Address Space Layout Randomisation (ASLR) which randomises the base address of the heap, to prevent an attacker knowing where it is located in memory. Non-determinism in the allocation process works alongside ASLR to add randomisation to the state transitions of the allocator itself. This is problematic for HLO as it means that over multiple runs of a program from the same starting state, the same sequence of interactions may produce a different layout each time. Although not common, this behaviour can be found in at least two mainstream allocators, namely the `Windows 10` the Low Fragmentation Heap (LFH) [22] and `jemalloc`. In this paper we do not address the problem of non-deterministic allocator behaviour.

2.1.7. Treatment of Larger Allocations. Most allocators utilise different algorithms and data structures to handle allocations of sizes that they consider to be small versus those they consider to be large. For example, an allocator might use segregated free lists for allocations up to a certain size and simply use `mmap` and `munmap` to manage allocations above that size. The threshold above which an allocator considers an allocation to be large varies by allocator, and also sometimes by the

Figure 4. Heap state evolution for `tcmalloc`. The axes are as in figure 3. As can be seen, the layout resulting from the same series of allocations is drastically different between the two allocators. In figure 3 the allocations are grouped together with most successive allocations simply being placed at the next highest free address. In contrast, `tcmalloc` results in these allocations being spread out over a much larger area of memory (resulting in the 'zoomed out' viewpoint). 2

.

operating system and architecture on which that allocator is running. Depending on the algorithms used, the desired layout and the starting heap state, this can either make HLO easier or more difficult.

2.2. Allocators

For experimentation and evaluation we selected a number of real world allocators, implementing a variety of different strategies and policies. We will refer to `dlmalloc` and `avrlibc` as *free list based* and `tcmalloc` and `PHP` as *segregated storage based*. An overview of the allocators selected can be found in the appendices, in Table 4, but their relevant nuances are as follows:

avrlibc 2.0. An allocator aimed at embedded systems [6] which utilises best fit search over a single free list. The maximum heap size and its location are fixed at compile time. If an existing free chunk of sufficient size does not exist then a new chunk is carved from the remaining heap space.

dlmalloc 2.8.6. A general purpose allocator [15] utilising best fit search over segregated free lists to store blocks with size less than 256KB. Free blocks less than 256 bytes in size are organised in linked lists, while those above 256 bytes, but less than 256KB, are kept in tries. Allocation requests for sizes greater than 256KB use `mmap`. The `GNU libc` allocator, `ptmalloc`, is based on `dlmalloc`.

tcmalloc 2.5.1. Intended as a more efficient replacement for `dlmalloc` and its derivatives, especially in multithreaded environments [9]. Sizes above 32KB are allocated using best-fit search from a linked-list of free pages, and splitting and coalescing may take place. Segregated storage is used for sizes below 32KB.

PHP 7. The allocator for version 7 of the `PHP` language interpreter. Sizes above 2MB in size are allocated via `mmap`. Sizes below 2MB but above 3/4s of the page size are rounded to the nearest multiple of the page size and are allocated on page boundaries using best-fit over a linked list of free pages. Sizes that are less than 3/4s of a page are rounded up to the next largest predefined small size, of which there are 30 (e.g. 8, 16, 24, 32, ..., 3072), and are allocated from runs using segregated storage.

3. The Heap Layout Optimisation Problem in Deterministic Settings

Allocators offer no guarantees on the positioning of allocations, nor any API to request a specific layout. Thus, given a starting state for the heap, one must utilise calls to `malloc`, `calloc`, `free` and `realloc` to manipulate it into the desired end state. Due to the vast array of data structures and algorithms which could be in use, it is not possible to predict the effect on the heap layout of a sequence of interactions without a model of the allocator implementation. Furthermore, any two allocators may produce drastically different heap states given the same series of allocation and free requests, as demonstrated in the difference between figures 3 and 4.

The most common approach to solving the heap layout optimisation problem is manual. An analyst examines the allocator's implementation in order to gain an understanding of its internals; then, at run-time, they inspect the state of its various data structures in order to determine what interactions are necessary in order to manipulate the heap into the layout that they require. This process is extensively documented in the literature of the hacking and computer security communities, with a variety of papers on the internals of individual allocators [1], [4], [14], [17], as well as the manipulation

and exploitation of those allocators when embedded in applications [3], [13], [19]. In section 4 we present an algorithm aimed at addressing this problem in a black-box fashion.

Further complicating the process is the fact that when constructing an exploit one cannot usually directly interact with the allocator, but instead must utilise the API exposed by the target program to indirectly interact with it. Manipulating the heap state via the program's API is often referred to as *heap feng shui* in the computer security literature [20]. Discovering the relationship between program-level API calls is a prerequisite for real-world HLO but can be addressed separately, as we demonstrate in section 5.2.

Finally, the deterministic setting for heap layout optimisation adds three restrictions to the general problem of heap layout optimisation. They are as follows:

1) The allocator's behaviour must be deterministic. As mentioned in section 2.2, most allocators are deterministic and thus this restriction has limited impact in terms of the relevance of our work to real world implementations. This restriction is necessary as our solution requires that executing a fixed set of allocator interactions from a fixed starting state always results in the same final heap state.
2) The solution generated in order to achieve a particular layout is paramaterised by the starting state of the heap. In other words, if the starting state of the heap changes then the solution generated to achieve a particular layout may no longer be valid. In practice, this means that the attacker must be able to determine the starting state of the heap, or have a means by which to force the heap into a known state.
3) There are no other actors interacting with the heap at the same time as us, or if there are then their interactions are deterministic and known, in terms of what the interaction is and when it takes place.

If the above three properties hold then we consider the setting to be deterministic.

4. Automatic Heap Layout Optimisation

In this section we present a pseudo-random black box search algorithm for heap layout optimisation, and two evaluation frameworks into which we have embedded it to perform heap layout optimisation on both synthetic benchmarks and real vulnerabilities.

While theoretically and practically straightforward, there are two motivations for using this simple search algorithm. First, as mentioned in the introduction, there is no existing proof of the complexity class of the heap layout optimisation problem. Nor does there exist prior work on automated solutions that would provide experimental guidance on the challenges that an automated search may face. Our initial solution provides a baseline that future, more sophisticated, implementations can be compared against.

Second, despite the potential size of the problem if measured by the number of possible combinations of available interactions, there is significant symmetry in the solution space for many problem instances. As our measure of optimality of a layout is based on the relative positioning of two buffers, we do not care about their absolute location nor their relative positioning to any other buffers. Similarly, when creating or filling holes to we also do not care about the absolute locations or adjacency of the holes; we care only that they are created, or filled. In many situations we also do not care about the order in which holes are filled or created, nor does it matter the means by which a hole is created or filled. It is often possible to achieve the same layout using significantly different series of interactions with the allocator. Figure 5 illustrates this idea, showing how two different interaction sequences, resulting in two different layouts, can place the source and destination in an identical layout *relative* to each other. Therefore, due to solution space symmetries, we propose that a pseudo-random black box search could be a feasible solution for large enough number of problem instances as to be worthwhile.

The search algorithm, listed as Algorithm 1, combines interaction sequences[4] to produce candidate solutions. Candidates are continuously created until a solution is discovered which results in an optimal layout, or the search is terminated. Depending on which of our evaluation systems is in use, the representation of a candidate, as well as the $Execute$, $AppendAllocSequence$, $AppendFreeSequence$, $AppendSrcSequence$ and $AppendDstSequence$ functions, differ in their implementation. The differences are explained in sections 4.1 and 4.2.

The algorithm is *pseudo-random* in the sense that there are two parameters which influence the contents of each candidate: the maximum number of interaction sequences in a candidate and the ratio of allocations to frees in each candidate. These are the m and r parameters, respectively, to SEARCH. While one could potentially utilise a completely random search, it makes sense to guide it away from candidates that are highly unlikely to be useful due to extreme values for m and r.

As the goal was to evaluate our search algorithm across multiple allocator designs and arbitrary starting states, we avoided further guiding the search. One could envisage a search which constructs candidates with a form specifically aimed at triggering likely patterns of hole filling and creation. However, as some allocators utilise multiple layout algorithms

4. An *interaction sequence* is an input to the target which, when processed by the target, results in one or more invocations of the `malloc`, `free`, `calloc` or `realloc` functions by the target. For example, an interaction sequence for `PHP` would be `imagecreatetruecolor(1, 1)`. How the set of available interaction sequences is constructed for `PHP` is described in section 4.2.

Algorithm 1 Find Solution Resulting in Optimal Layout

```
1: function SEARCH(g, m, r)
2:     while g ≠ 0 do
3:         tmpInst = ConstructCandidate(m, r)
4:         tmpDist = Execute(tmpInst)
5:         if tmpDist < minDist then
6:             dist ← tmpDist
7:             inst ← tmpInst
8:         end if
9:         g ← g − 1
10:    end while
11:    return inst
12: end function
13: function CONSTRUCTCANDIDATE(m, r)
14:    result = NewCandidate()
15:    i ← random(0, m − 1)
16:    while i ≠ 0 do
17:        if random(1, 100) < r then
18:            AppendAllocSequence(result)
19:        else
20:            AppendFreeSequence(result)
21:        end if
22:        i ← i − 1
23:    end while
24:    if random(0, 1) = 0 then
25:        AppendSrcSequence(result)
26:        AppendDstSequence(result)
27:    else
28:        AppendDstSequence(result)
29:        AppendSrcSequence(result)
30:    end if
31:    return result
32: end function
```

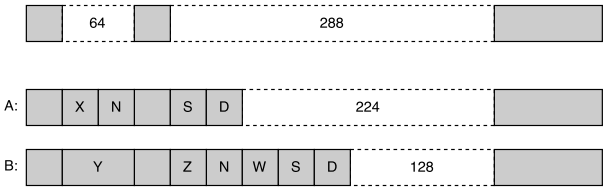

Figure 5. Two different heap states, resulting from two different interaction sequences, which have the same value according to our measure of optimality. Assume that the interaction sequence which allocates the overflow source and destination contains three allocations: the source, labelled 'S', an intervening allocation labelled 'N', which must be captured, and the destination, labelled 'D', all of size 32. In the outcome labelled 'A' a hole of size 32 was was first created to consume N via the allocation of the chunk labelled 'X'. In the outcome labelled 'B', first the allocation labelled 'Y' is made to fill the hole of size 64. After that a hole to capture 'N' is constructed via three allocations and a free, with the remaining two allocations labelled 'Z' and 'W'. Both sequences place 'S' and 'D' adjacent to each other and in the same order.

internally, and as the precise sequences required may vary with starting state, we felt this would likely end up being quite a fragile solution.

We constructed two systems to investigate the problem and to evaluate solutions. The first, described in section 4.1 allows for synthetic experiments to be run against any allocator exposing the standard interface. This allows one to construct a diverse array of problem instances by varying the heap starting state, available allocation and deallocation sizes, and the amount of noise per interaction sequence. We utilised this system to discover the factors which most significantly influence the difficulty of automated heap layout optimisation, and to provide an indication as to how successful a given search algorithm would likely be on real world instances. The second system, described in section 4.2, is a fully automated heap layout optimisation engine designed to work with the `PHP` language interpreter. This system is not only an evaluation framework for search algorithms but also solves the problems of mapping from the application level interface to interaction

sequences, and of automatically discovering potentially useful corruption targets.

4.1. Synthetic Evaluation Framework

To allow for the evaluation of search algorithms we constructed an experimental system which allows one to embed any allocator exposing the ANSI C dynamic memory management interface in a test harness, have it initialised to a given heap state, and then interact with it by sending a series of interaction sequences. The test harness converts the interaction sequences into the corresponding `malloc`, `free`, `calloc` and `realloc` function calls and then invokes them on the allocator. Once the interaction sequences have been relayed to the allocator, the client is sent back the distance from the source allocation to the destination allocation.

In practice, in this use case the SEARCH function produces a list of *commands* with a form similar to `alloc 128 1` to indicate an allocation of size 128 should take place and the resulting chunk has ID 1, and `free 1` indicating that a free should take place of the pointer associated with the chunk with ID 1. The `Execute` function then takes this list of commands and feeds it to a driver which is linked with the desired allocator. The driver converts each command into the corresponding allocator interaction.

Each experiment can be configured via the following parameters:

- **The starting state of the heap**. Usually, an attacker cannot interact with a heap immediately after it has been created. Instead, there will typically be a number of allocations and frees that have taken place before the attacker can begin manipulating the layout.
- **The available interaction sequences**. As previously mentioned, an attacker can rarely interact directly with the heap allocator's API, and instead will have to utilise the API, or input mechanism, of the target program as a proxy. Restrictions may also exist on the order and frequency with which a particular API, and thus the related interaction sequence, may be triggered. In our system, a Python class can be defined which exposes the `GetAllocSequence` and `GetFreeSequence` functions referenced in SEARCH. The class can then programatically decide which sequences are available on each call and select one.
- **The allocator**. As each allocator provides the same API, the driver program can easily be linked with the allocator of choice with no modifications to the rest of the system.

Adding new starting states, interaction sequences or allocators requires only limited local modifications, such as a specification describing a start state, a Python class to generate interactions, or a Makefile addition to produce a new driver linked with a new allocator. This system allows for flexible and scalable evaluation of new search algorithms, or testing existing algorithms on new allocators, interaction sequences or heap start states.

4.2. PHP-Based Evaluation Framework

Using the experimental system outlined in section 4.1 it is not only possible to perform synthetic experiments but also to solve real problem instances. Given a real program, one *could* capture the all the required information to model the problem, including the initial heap state and available interaction sequences. However, for complex targets this is likely to be an error prone process potentially resulting in both false positives and false negatives.

In practice, it is likely to be less problematic to perform the search utilising the exposed API of a target application, rather than explicitly extracting the heap starting state and available interaction sequences to perform the search. To evaluate search algorithms in this situation we chose the `PHP` language interpreter as our target as it is a large, modern application with a non-trivial interface for manipulating its internal state, and the state of its allocator; namely, programs written in the `PHP` language.

In this setting, a candidate is an actual synthesised `PHP` program which is evaluated by passing it to the `PHP` interpreter. The set of available allocator interaction sequences is the set of interactions which may be triggered by some fragment of `PHP` code at a given point in a program. We implemented the system as three distinct phases: firstly, a component that identifies fragments of `PHP` code which provide distinct allocator interactions; secondly, a component which identifies dynamically allocated structures which may be useful to corrupt as part of an exploit, and a means by which to trigger their allocation; thirdly, a search algorithm which pieces together the fragments triggering allocator interactions to produce `PHP` programs as candidates. The third phase is effectively a search over `PHP` programs to find one which optimises the heap layout for a given source and destination allocation. All three phases have nuances which we explain in the remainder of this section.

4.2.1. Identifying Available Interaction Sequences. To etermine the available interaction sequences it is necessary to be able to synthesise self-contained fragments of `PHP` code and determine what allocator interactions that fragment triggers. The latter problem is straightforward: we inject instrumentation into the `PHP` interpreter to record the allocator interactions

that result from executing a given fragment. The former problem is more involved as it requires one to discover and combine valid fragments of PHP, which ideally result in a diverse and expansive set of allocator interactions.

We resolve the synthesis problem by implementing a fuzzer for the PHP interpreter which leverages the regression tests that come with PHP, in the form of PHP programs. This idea is based on previous work which utilised a similar approach for the purposes of vulnerability detection [11], [12]. The tests provide examples of the functions that can be called as well as the number and types of their arguments. The fuzzer then utilises mutation and combination to produce new fragments with new behaviours.

To tune the fuzzer towards the discovery of fragments that are useful for heap layout optimisation, as opposed to vulnerability discovery, we added some further constraints:

- We utilise mutations that are intended to produce an interaction sequence that we have not seen before, rather than a crash. For example, fuzzers will often replace integers with values that may lead to edge cases, such as 0, $2^{32}-1$, $2^{31}-1$ and so on. We are interested in triggering unique allocator interactions however, and so we predominantly mutate tests using integers and string lengths that relate to allocation sizes we have not previously seen.
- Our measure of *fitness* for a generated test is not based on code coverage, as is often the case with vulnerability detection, but is instead based on whether or not a new allocator interaction sequence is produced, and the length of that interaction sequence.
- We discard any fragments that result in the interpreter exiting with an error.
- We favour the shortest, least complex fragments with priority being given to fragments consisting of a single function call.

The output of this stage is a mapping from fragments of PHP code to a summary of the allocator interaction sequences which occur as a result of executing that code. The summary includes the number and size of any allocations, and whether or not the sequence triggers any frees, which can be used by the search algorithm to prioritise sequences, as required.

4.2.2. Specifying Candidate Structure. Different vulnerabilities in PHP require different setup in order to trigger, in terms of function calls, object creation and so on. To avoid hard-coding vulnerability-specific information in the candidate creation process we allow for the creation of candidate templates which define the structure of a candidate. A sample template for CVE-2016-7126, a heap-based buffer overflow in PHP, can be found in Listing 2 in the appendix. The markers in the comments define various components of the candidate, such as the code for allocating the corruption source, the code which triggers the bug etc. In the context of PHP, the $NewCandidate$ function in SEARCH extracts the above structure from a template and the various $Append*$ functions utilise it when considering where in the template to place the various fragments for manipulation, source allocation and destination allocation.

4.2.3. Automatic Identification of Corruption Targets. In most programs there is a diverse set of dynamically allocated structures that one could corrupt to violate some security property of the program. These targets may be program specific, such as values that guard a sensitive path, or they may be somewhat generic, such as a function pointer. Identifying these targets, and how to dynamically allocate them, can be a difficult manual task in itself. To further automate the process we implemented a component which, as with the fuzzer, splits the PHP tests into standalone fragments and then observes the behaviour of these fragments when executed. If the fragment dynamically allocates a data structure and writes what appears to be a pointer to that data structure we consider it to be a potentially interesting corruption target and store the fragment. Before beginning the search the system will randomly select one of the discovered corruption targets and utilise its corresponding code fragment when it decides to allocate the destination.

4.2.4. Search. The search for an optimal solution proceeds as shown in Algorithm 1, with some minor nuances. Candidates are PHP programs constructed using a template and the $Execute$ function provides these candidates to the PHP interpreter to evaluate them. The $AppendAllocSequence$ and $AppendFreeSequence$ functions utilise the database of fragments constructed as described in section 4.2.1. Fragments are pseudo-randomly selected from the database with fragments that trigger allocations or frees of sizes that equal the source or destination receiving a higher probability of being selected. The amount of noise in a sequence is used as a secondary metric in sequence selection, with sequences that have less noise having a higher probability of being selected than more noisy sequences.

Finally, for some vulnerabilities we may not be able to control the order in which the source and destination are allocated, e.g. the `imagetruecolortopalette` function which triggers CVE-2016-7126, as shown in listing 2 in the appendices, both allocates the source and triggers the vulnerability. Therefore, the destination *must* be allocated before this occurs. In such cases, we indicate this in the template provided to the system which then ensures that the destination allocation is triggered before the source allocation and vulnerability are triggered, regardless of the order in which $AppendSrcSequence$ and $AppendDstSequence$ are called.

TABLE 1. Optimisation results after 500,000 instances, averaged across all starting sequences. The full results can be found in tables 5 and 6 in the appendices

Allocator	Noise	% Und. < 4096	% Ovf. < 4096	% Und. Optimal	% Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	0	100	100	100	100	0.00	0.00
dlmalloc-2.8.6	0	100	100	97	100	0.00	0.00
tcmalloc-2.6.1	0	94	94	83	81	1.01	1.24
avrlibc-r2537	1	83	83	44	50	0.43	0.58
dlmalloc-2.8.6	1	81	83	22	58	0.37	0.68
tcmalloc-2.6.1	1	83	89	44	58	1.07	1.52
avrlibc-r2537	4	75	78	33	42	1.55	1.65
dlmalloc-2.8.6	4	75	78	14	44	1.25	1.95
tcmalloc-2.6.1	4	72	83	22	53	1.40	2.47

5. Experiments

To investigate automated heap layout optimisation we chose the allocators mentioned in section 2.2: `tcmalloc`, `dlmalloc`, `avrlibc` and the allocator for `PHP` 7. These allocators offer various combinations of the features discussed in section 2 and are used in a number of real world applications and operating systems.

We conducted two sets of experiments. First, in order to investigate the fundamentals of the problem we utilised the system discussed in section 4.1 to conduct a set of synthetic experiments utilising differing combinations of heap starting states, interaction sequences, source and destination sizes, and allocators. Secondly, in order to evaluate the viability of our search algorithm on real world applications we utilised the system discussed in section 4.2 to automatically construct heap layout manipulating inputs for three security-relevant vulnerabilities in `PHP`. These inputs manipulate the heap into a state such that when the vulnerability is triggered a buffer selected by the user, or automatically by the system, is accessed[5]. All experiments were carried out on a server with 80 Intel Xeon E7-4870 2.40GHz cores and 1TB of RAM, utilising 40 concurrent analysis processes.

5.1. Synthetic Experiments

The goal of the synthetic experiments is to discover the factors which influence the difficulty of problem instances and to experimentally discover the capabilities and limitations of our search algorithm in an environment that we precisely control. It is important to note however that these experiments are not exhaustive, and thus the data does not, and is not intended to, support general claims of applicability. Our experiments were configured as follows:

- As it is not feasible to evaluate layout optimisation for all possible combinations of source and destination sizes we selected 6 sizes which we deemed to be both likely to occur in real world problems and to exercise different allocator behaviour. The sizes we selected are 0[6], 64, 512, 4096, 16384 and 65536. We then utilised the cross product of this set of six to give 36 different source and destination combinations.
- We simulate situations in which the allocation of the source or destination takes place in both noise-free and noisy interaction sequences. In noise-free interaction sequences the overflow source or destination is the only interaction that takes place. In noisy interaction sequences other allocations also take place, and those extra allocations may require more holes to be created in order to capture them, thus increasing the difficulty of the problem. We evaluated the effect of prefixing and suffixing both the source and destination with varying amounts of noise.
- For each source and destination combination, we made available to the analyser an interaction sequence which triggers an allocation of the source size, an interaction sequence which triggers an allocation of the destination size, and two separate interaction sequences for freeing each of the allocations.
- We initialise the heap state prior to executing the interactions from a candidate by prefixing each candidate with a set of interactions. Previous work [21] has outlined the drawbacks that may arise when using randomly generated heap states to evaluate allocator performance. In order to avoid these drawbacks we captured the initialisation sequences of `PHP`, `Python` and `Ruby` and utilised them for our experiments. A summary of the relevant properties of these initialisation sequences can be found in the appendices in table 3.

5. The access may be a read or a write, depending on the type of vulnerability.
6. Allocating a size of 0 will trigger the smallest allocation possible for the allocator in question.

TABLE 2. Results of heap layout optimisation for vulnerabilities in PHP. Experiments were run for a maximum of 12 hours.

CVE ID	Vulnerability Type	Component	Src. Size	Dst. Size	Noise	Initial Dist.	Final Dist.	Time to Best	Candidates to Best
2015-8865	OOB Write	File Parsing	480	264	42	119360	0	3780s	3753540
2016-5093	OOB Read	String Manip.	544	264	1	39680	0	9s	8937
2016-7126	OOB Write	Image Manip.	1	264	48	384088	16	1838s	2157812

The results of the synthetic experiments can be found in table 1. As mentioned, for each combination of allocator, starting state and noisy allocation count, a total of 36 experiments were run. In the column titles, '*Und*' and '*Ovf*' stand for underflow and overflow, breaking down the results into those where the objective was to place the destination before or after the source, respectively. The '*% Und.* $<$ *4096*' and '*% Ovf.* $<$ *4096*' columns indicate the percentage of results where the source and destination were within a page (4096 bytes) of each other. The '*% Und. Optimal*' and '*% Ovf. Optimal*' columns indicate the percentage of results where the source and destination were in an optimal layout. The final two columns, '*Avg. Und. Err.*' and '*Avg. Ovf. Err*' provide the average *normalised* error across all runs for that row, broken down by overflow and underflow. The normalised error as the ratio of the best score achieved for a particular problem to the larger of the source and destination sizes. We use this measure instead of the absolute number of bytes between the source and destination as we discovered empirically that the absolute error is usually a factor of the sizes used in the allocation and deallocation sequences. For example, the error is usually due to Algorithm 1 making bad allocation or deallocation decisions, which creates allocated chunks between the source and destination. If the used sizes are large then the absolute error value will also be large, while if the sizes are small then the absolute error will be small, but in both cases the errors made during the algorithm's decision making process are the same. To avoid this issue we normalise the absolute error value by dividing it by the larger of the two allocation sizes available.

5.2. PHP-Based Experiments

In order to determine if heap layout optimisation is feasible in real world scenarios we selected three vulnerabilities in PHP to evaluate the system described in section 4.2. A successful outcome means the system can, with no human assistance, discover how to interact with the underlying allocator via PHP scripts, identify how to allocate potentially sensitive data structures on the heap, and then construct a PHP script which places such a data structure adjacent to the source of an out-of-bounds memory access. Such an construct, where an input is crafted to allow reading/writing/execution of specific memory within a process, is often called an *exploit primitive*. Primitives form the building blocks of exploits, and thus successfully generating such inputs represents a step towards automation of exploitation of heap based vulnerabilities.

We utilised the following vulnerabilities in our evaluation:

- **CVE-2015-8865**. An out-of-bounds write vulnerability in `libmagic` that is triggerable via PHP up to version 7.0.4.
- **CVE-2016-5094**. An out-of-bounds read vulnerability in PHP up to version 7.0.7.
- **CVE-2016-7126**. An out-of-bounds write vulnerability in PHP up to version 7.0.10.

A summary of the results can be found in table 2. The '*Initial Distance*' is the distance from the source to destination buffers as allocated via the vulnerability trigger without heap layout manipulation. The '*Final Distance*' is the distance from the source to destination buffers utilising the solution discovered by our HLO system. The '*Noise*' is a count of the number of allocator interactions which take place in the sequences related to allocating the source and destination, excluding the interactions to allocate the source and destination.

5.3. Experimental Conclusions

5.3.1. Black-box random search can be an effective mechanism for automatic HLO. Our tests against three real world vulnerabilities in PHP indicate that heap layout optimisation can be performed automatically for real world programs, and that pseudo-random black box search can find optimal, or near-optimal, layouts in the cases that we utilised for evaluation. As shown in the solution for CVE-2016-7126 and CVE-2015-8865, it is possible to achieve near-optimal and optimal results even when there is a large amount of noise.

Our system also demonstrates that it is possible to automate the process in an end-to-end manner, with automatic discovery of a mapping from the target program's API to allocator interaction sequences, discovery of potentially interesting corruption targets, and search for an optimal layout. A sample output can be seen in listing 3 in the appendices. Using a template for

this particular vulnerability, and the learned API for interacting with PHP's allocator via its API, the system has synthesised a new PHP script which places the overflow source within 16 bytes of the overflow destination.

As mentioned previously, our synthetic experiments are not exhaustive and we cannot draw general conclusions regarding effectiveness. However, by evaluating the data we have we can still make inferences about the capabilities and limitations of the approach. We will consider three different measures of success: the number of optimal results achieved, the average error for sub-optimal results, and the number of results achieved with an error of less than a page.

In the case of free-list based allocators, when there is no noise then an optimal layout was achieved in all but one experiment, with most experiments taking 15 seconds or less. Overall, for dlmalloc and avrlibc, 59% of experiments resulted in an optimal layout, the average error is .7 and in 86% of cases the source is placed within a page of the destination. This indicates that even in noisy situations pseudo-random black-box search is effective at placing the source an destination relatively close to each other, if not in an optimal layout.

The extra constraints imposed on layout by segregated storage presents more of a challenge. However, on noise-free runs optimal results are still achieved in 82% of the experiments with an average error of 1.13 and with 95% of experiments ending with the source and destination within a page of each other. Overall, for tcmalloc, 54% of experiments result in an optimal layout, the average error is 1.52 and in 86% of cases the source is placed within a page of the destination. The low average error again indicates that, even in cases where a sub-optimal result is found, the search is still able to find a candidate which places the source and destination relatively close together.

5.3.2. Segregated storage is the allocator policy which most impacted problem difficulty. The most significant policy which impacts difficulty is the physical grouping by size of objects in segregated storage based allocators. As expected, the extra constraints imposed in this situation increase the difficulty of the problem for our search approach. However, as demonstrated in our synthetic experiments and versus PHP, many problem instances are still solvable when segregated storage is in use. There are even cases in which the grouping of objects imposed by segregated storage may actually be helpful. For example, it predominantly isolates the impact of allocations and frees to future allocations and frees of the same size. Thus if a hole of size 256 is created one need not worry that a future allocation of size 128 will utilise that space, as is possible when segregated storage is not in use.

The other differentiating mechanisms and policies between the allocators did not appear to have a significant impact on the difficulty of the problem.

5.3.3. As the noise in interaction sequences increases, so does the problem difficulty. Our experiments clearly show that as noise in the interaction sequences increases so does the the difficulty of the problem instances. This makes intuitive sense as, typically, as more noise is added more holes typically have to be created. In the worst case (dlmalloc) we see a drop off from a full set of optimal results to 29% of the results being optimal when four noisy allocations are included.

It is also worth noting that the increase in error as a result of adding noise does not seem to be as significant for segregated storage based allocators as it is for free list based allocators. For free list based allocators the average error increases by a factor of 3x-4x as the noise increases, while for segregated storage based allocators the increase is predominantly less than 2x. One explanation for this is that, as explained in section 5.3.2 in some situations segregated storage actually makes it easier to create holes of a particular size and to ensure they are not accidentally utilised by a smaller allocation.

In the evaluation on vulnerabilities in PHP, the amount of noise impacts the number of candidates required to find the best result, but the algorithm still finds an optimal result for two of the three experiments, and a near-optimal result for the third. For CVE-2016-5093, which has a single noisy allocation, an optimal result is achieved in 9s while for CVE-2015-8865, which has 42 noisy allocations, it takes 3780s to achieve the same.

6. Conclusion

In this technical report we have outlined a simple, but effective, algorithm for heap layout optimisation and shown that it can produce optimal layouts in a significant number of synthetic experiments, as well as for real world vulnerabilities in the PHP language interpreter. The results are encouraging and there are a number of extensions and directions for future work. The most obvious include improving the search algorithm to handle the unsolved synthetic instances and to improve performance on difficult cases, investigating the extension of the solution to targets other than PHP and researching the possibility for integrating this approach with traditional symbolic execution based automatic exploit generation frameworks.

References

[1] Anonymous, "Once Upon a free()," *Phrack*, Aug. 11 2001, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/57/9.html

[2] ANSI X3.159-1989, *American National Standard Programming Language C*, Dec. 14 1990.

[3] argp, "OR'LYEH? The Shadow over Firefox," *Phrack*, May 6 2016, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/69/14.html

[4] argp and huku, "Pseudomonarchia jemallocum," *Phrack*, Apr. 14 2012, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/68/10.html

[5] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Network and Distributed System Security Symposium*, Feb. 2011.

[6] AVR Libc Developers, "AVR Libc," Accessed: 7th Sep. 2017. [Online]. Available: http://www.nongnu.org/avr-libc/

[7] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–157.

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 380–394. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.31

[9] S. Ghemawat and P. Menage, "TCMalloc: Thread-Caching Malloc," Accessed 8th Sep. 2017. [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[10] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," Master's thesis, University of Oxford, 2009.

[11] S. Heelan, "Ghosts of Christmas Past: Fuzzing Language Interpreters using Regression Tests," in *Infiltrate*, 2014.

[12] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.

[13] huku and argp, "Exploiting VLC - A case study on jemalloc heap overflows," *Phrack*, Apr. 14 2012, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/68/13.html

[14] jp, "Advanced Doug Lea's Malloc Exploits," *Phrack*, Aug. 13 2003, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/61/6.html

[15] D. Lea, "A Memory Allocator," Accessed: 7th Sep. 2017. [Online]. Available: http://g.oswego.edu/dl/html/malloc.html

[16] T. Mandt, "Kernel Pool Exploitation on Windows 7," in *Blackhat DC*, 2011.

[17] MaXX, "Vudo Malloc Tricks," *Phrack*, Aug. 11 2001, Accessed 8th Sep. 2017. [Online]. Available: http://phrack.com/issues/57/8.html

[18] J. McDonald and C. Valasek, "Practical Windows XP/2003 Exploitation," in *Blackhat USA*, 2009.

[19] Solar Designer, "JPEG COM Marker Processing Vulnerability (in Netscape Browsers and Microsoft Products) and a generic heap-based buffer overflow exploitation technique," Jul. 25 2000, Accessed 8th Sep. 2017. [Online]. Available: http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

[20] A. Sotirov, "Heap Feng Shui in Javascript," in *Blackhat USA*, 2007.

[21] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116.

[22] M. V. Yason, "Windows 10 Segment Heap Internals," in *Blackhat USA*, 2016.

Appendix

```php
<?php
$img = imagecreatetruecolor(10, 10);
imagetruecolortopalette($img, false, PHP_INT_MAX / 8);
?>
```

Listing 1. The original bug report for CVE-2016-7126, from which the template in listing 2 was produced.

```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# BEGIN-DST-CREATE
shrike_record_destination(1);
$dst = imagecreatetruecolor(1, 1);
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false, PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

Listing 2. A sample template for a `PHP` vulnerability (CVE-2016-7126). The system is free to insert heap manipulating actions at any location outside the BEGIN-X/END-X regions. The arguments to `shrike_record_source` and `shrike_record_destination` tell the system which allocation made by the following API call is considered the source and destination respectively. Listing 1 shows the original vulnerability report from which this template was produced.

```php
<?php
# BEGIN-PRELUDE
$img = imagecreatetruecolor(10, 10);
# END-PRELUDE

# <...>
$var_vtx_24 = imagecreatetruecolor(1, 1);
# <...>
$var_vtx_24 = 0;
$var_vtx_102 = imagecreatetruecolor(1, 1);
$var_vtx_103 = imagecreatetruecolor(2, 256);
$var_vtx_104 = str_repeat('A', 225);
$var_vtx_105 = str_repeat('A', 225);

# BEGIN-DST-CREATE
shrike_record_destination(1);
$vtx_dst = array_fill(0, 1, '..');
# END-DST-CREATE

# BEGIN-SRC-CREATE-AND-TRIGGER
shrike_record_source(48);
imagetruecolortopalette($img, false, PHP_INT_MAX / 8);
# END-SRC-CREATE-AND-TRIGGER
?>
```

Listing 3. Part of the solution discovered for CVE-2016-7126. Just over 100 API calls are made, using functions which have been discovered to trigger the desired allocator interactions. Frees are triggered by destroying previously created objects, as can be seen with `var_24`. The destination is the second pointer allocated by `array_fill`, and has been automatically discovered as described in section 4.2. The overflow source is the 49^{th} allocation performed by `imagetruecolortopalette`.

TABLE 3. Summary of the heap initialisation sequences used during the synthetic experiments. All sequences were captured by hooking the `malloc`, `free`, `realloc` and `calloc` functions of the system allocator, except for `php-emalloc` which was captured by hooking the allocation functions of the custom allocator that comes with `PHP`.

Title	# Allocator Interactions	# Allocs	# Frees
php-emalloc	571	366	205
php-malloc	15078	12714	2634
python-malloc	6160	3710	2450
ruby-malloc	70895	51827	19068

TABLE 4. Allocator Features. Entries marked with an asterisk indicate that the behaviour may or may not occur, depending on the size of chunk of memory allocated or freed. See section 2.2 for details.

Allocator	Version	Splits Blocks	Carves From	Coalesces Blocks	Segregated Storage	Deterministic
avrlibc	2.0	Yes	Tail	Yes	No	Yes
dlmalloc	2.8.6	Yes	Head	Yes	No	Yes
tcmalloc	2.6.1	*	Head	*	Yes	Yes
PHP	7	*	Head	*	Yes	Yes

TABLE 5. Optimisation results after 500,000 candidates per configuration.

Allocator	Start State	Noise	# Und. < 4096	# Ovf. < 4096	# Und. Optimal	# Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
avrlibc-r2537	php-emalloc	0	36	36	36	36	0.00	0.00
avrlibc-r2537	php-malloc	0	36	36	36	36	0.00	0.00
avrlibc-r2537	python-malloc	0	36	36	36	36	0.00	0.00
avrlibc-r2537	ruby-malloc	0	36	36	36	36	0.00	0.00
	Average		36	36	36	36	0.00	0.00
dlmalloc-2.8.6	php-emalloc	0	36	36	35	36	0.00	0.00
dlmalloc-2.8.6	php-malloc	0	36	36	36	36	0.00	0.00
dlmalloc-2.8.6	python-malloc	0	36	36	35	36	0.02	0.00
dlmalloc-2.8.6	ruby-malloc	0	36	36	36	36	0.00	0.00
	Average		36	36	36	36	0.00	0.00
avrlibc-r2537	php-emalloc	1	31	30	17	19	0.41	0.49
avrlibc-r2537	php-malloc	1	32	30	21	19	0.49	0.67
avrlibc-r2537	python-malloc	1	29	29	16	19	0.48	0.55
avrlibc-r2537	ruby-malloc	1	29	29	11	16	0.36	0.61
	Average		30	30	16	18	0.43	0.58
dlmalloc-2.8.6	php-emalloc	1	27	30	7	24	0.38	0.77
dlmalloc-2.8.6	php-malloc	1	32	30	15	22	0.32	0.78
dlmalloc-2.8.6	python-malloc	1	29	30	7	20	0.37	0.67
dlmalloc-2.8.6	ruby-malloc	1	27	31	4	19	0.39	0.51
	Average		29	30	8	21	0.37	0.68
avrlibc-r2537	php-emalloc	4	28	28	13	15	1.62	1.72
avrlibc-r2537	php-malloc	4	28	28	11	14	1.28	1.60
avrlibc-r2537	python-malloc	4	27	27	12	17	1.58	1.71
avrlibc-r2537	ruby-malloc	4	26	28	10	15	1.71	1.55
	Average		27	28	12	15	1.55	1.65
dlmalloc-2.8.6	php-emalloc	4	28	27	6	13	1.21	1.82
dlmalloc-2.8.6	php-malloc	4	26	27	4	16	1.37	2.08
dlmalloc-2.8.6	python-malloc	4	27	31	4	15	1.00	1.95
dlmalloc-2.8.6	ruby-malloc	4	27	28	4	20	1.41	1.93
	Average		27	28	5	16	1.25	1.95

TABLE 6. Optimisation results after 500,000 candidates per configuration.

Allocator	Start State	Noise	# Und. < 4096	# Ovf. < 4096	# Und. Optimal	# Ovf. Optimal	Avg. Und. Err.	Avg. Ovf. Err
tcmalloc-2.6.1	php-emalloc	0	36	36	29	30	0.32	0.34
tcmalloc-2.6.1	php-malloc	0	36	36	30	32	0.18	0.35
tcmalloc-2.6.1	python-malloc	0	28	28	28	24	3.46	4.07
tcmalloc-2.6.1	ruby-malloc	0	36	36	31	30	0.11	0.20
	Average		34	34	30	29	1.01	1.24
tcmalloc-2.6.1	php-emalloc	1	33	33	16	22	0.66	1.01
tcmalloc-2.6.1	php-malloc	1	31	34	15	22	0.40	0.78
tcmalloc-2.6.1	python-malloc	1	25	25	18	19	2.77	3.58
tcmalloc-2.6.1	ruby-malloc	1	30	34	15	22	0.44	0.71
	Average		30	32	16	21	1.07	1.52
tcmalloc-2.6.1	php-emalloc	4	28	32	8	19	1.18	2.03
tcmalloc-2.6.1	php-malloc	4	28	31	7	19	0.96	1.90
tcmalloc-2.6.1	python-malloc	4	19	24	7	16	2.52	3.78
tcmalloc-2.6.1	ruby-malloc	4	28	31	8	21	0.94	2.16
	Average		26	30	8	19	1.4	2.47