CAMP: Compiler and Allocator-based Heap Memory Protection

Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing

{zplin, zhengyu2027, n7l8m4}@u.northwestern.edu
{simone.campanoni, pdinda, xinyu.xing}@northwestern.edu
Northwestern University

Abstract

The heap is a critical and widely used component of many applications. Due to its dynamic nature, combined with the complexity of heap management algorithms, it is also a frequent target for security exploits. To enhance the heap's security, various heap protection techniques have been introduced, but they either introduce significant runtime overhead or have limited protection. We present CAMP, a new sanitizer for detecting and capturing heap memory corruption. CAMP leverages a compiler and a customized memory allocator. The compiler adds boundary-checking and escape-tracking instructions to the target program, while the memory allocator tracks memory ranges, coordinates with the instrumentation, and neutralizes dangling pointers. With the novel error detection scheme, CAMP enables various compiler optimization strategies and thus eliminates redundant and unnecessary check instrumentation. This design minimizes runtime overhead without sacrificing security guarantees. Our evaluation and comparison of CAMP with existing tools, using both real-world applications and SPEC CPU benchmarks, show that it provides even better heap corruption detection capability with lower runtime overhead.

1 Introduction

The heap is a region of memory that is dynamically allocated during runtime. It is widely used for dynamic memory allocation and for storing data structures with variable sizes. Due to its frequent use and complex nature, the heap is vulnerable to spatial and temporal memory errors. The prominence of heap errors as a source of vulnerabilities has been consistently high over the years. According to Microsoft, heap errors were responsible for 53% of Remote Code Execution (RCE) CVEs in their products [40]. Google Project Zero discovered that heap errors accounted for 69% of zero-day vulnerabilities observed in the wild [59] in 2022. Until now, 65% vulnerabilities are confirmed as heap-based zero-day in Linux, 2023 [7].

Over the years, various heap protection techniques have been introduced to enhance the security of the heap, including improved heap management algorithms [49] and the implementation of layout randomization techniques [34, 44]. These advancements have made the heap more secure. However, the continuous discovery and development of new vulnerabilities and exploitation techniques [16, 24, 33, 54, 58] suggest that heap exploitation remains an ongoing challenge.

To address heap exploitation effectively, we believe that detecting and capturing heap memory corruption is a key solution. Previously, various research efforts have been made towards this goal, such as Memcheck [27] and Address Sanitizer (ASAN) [47] for comprehensive protection or works that mitigate use-after-free errors [11,12,23,31,42,49,53] or detect out-of-bound access [19,41,56] for partial protection. However, these techniques either introduce significant runtime overhead or offer weak security guarantees. To be specific, MemCheck [27], though capable of delivering full heap detection, suffers from a substantial 26x overhead. ASAN and its variant ASAN–, as we will discuss in Section 6, face challenges with false negatives in both temporal and spatial heap error detection [32]. Other tools like FFmalloc [11], Delta Pointer [30], and Oscar [18] only offer partial heap protection, limiting their effectiveness.

In this work, we introduce CAMP (**C**ompiler and **A**llocator-based Heap **M**emory **P**rotection), a novel heap sanitizer, for detecting and capturing spatial and temporal heap errors. Unlike previous works [19,22,28,32,46,55,56,61] that require hardware support, CAMP is a software-only tool, consisting of a compiler and a seglist allocator. The compiler instruments the target program to validate the pointer boundary and build point-to relation at runtime. The customized memory allocator tracks memory ranges for each allocation, supports the instrumented instructions, and neutralizes dangling pointers when a memory object is freed.

In comparison with previous works [60,62], the key novelty of CAMP is mainly manifested in its design that leverages the run-time guarantee to optimize static instrumentation and thus reduces overhead. For example, to capture dangling pointer dereference, previous methods need to check each pointer dereference. In our design, CAMP introduces a run-time scheme that could guarantee that no dangling pointer exists and thus allow CAMP's compiler to eliminate corresponding pointer dereference checks. Furthermore, as we will detail in Section 4, CAMP implements its run-time scheme based on the extension of an existing allocator. This allocator has an $O(1)$ computation complexity in pointer validation. As a result, CAMP can perform pointer validation more efficiently.

In summary, this paper makes the following contributions.

- We present a novel approach CAMP that employs a customized allocator and a compiler to safeguard against heap

memory corruption. Additionally, we propose optimization strategies to reduce the performance overhead introduced by the instrumentation.

- We implement `CAMP` by customizing a segregated list allocator – tcmalloc and building our instrumentation optimization mechanism on top of LLVM 12.0 compiler framework. We open-sourced our prototype of `CAMP` at [8].
- We conduct a thorough evaluation of `CAMP` using the real-world application Nginx, as well as the SPEC CPU 2006 and 2017 benchmarks, from both security and runtime overhead perspectives. The evaluation compares `CAMP`' performance with other defense solutions offering similar heap protection levels.

The rest of the paper is organized as follows. Section 2 introduces the background of memory corruption on the heap as well as heap allocators. Section 3 discusses the assumptions of our research and the threat model. Section 4 describes the details of the proposed techniques. Section 5 presents our implementation details. Section 6 evaluates the security and runtime overhead of our proposed techniques. Section 7 provides the discussion of some related issues, followed by the related work in Section 8. Finally, we conclude the work in Section 9.

2 Background

2.1 Heap Memory Corruption & Protection

In general, there are two main types of heap memory corruption, overflow and use-after-free. In the following, we describe those two types of memory corruption in detail and discuss their protection.

Heap Overflow. Generally, each heap object has its own memory space. When the access to a heap object exceeds its memory space, a heap overflow happens. One common way to detect heap overflow is to reserve some memory as heap cookies or red zones. Once the magic value in the reserved area is tempered, the heap overflow could be detected [37]. However, this design naturally has the drawback of being bypassed. For example, the attacker could fail the detection by leaking the heap cookie [5] or overflowing the memory with the red zone intact [45]. An alternative approach is to validate pointers making sure no out-of-bound access happens [19, 38, 56]. This approach is effective in general but often introduces non-negligible overhead [41].

Heap Use-After-Free. When the memory space of a heap object is freed, the references to the object left become dangling pointers. The program should never deference the dangling pointer. Otherwise, a use-after-free would occur. Researchers have proposed many techniques to detect use-after-free. ASAN [47] uses shadow memory to record the memory status and instruments every memory access. Accessing a freed object could be detected immediately by checking the shadow memory. Although ASAN's approach only introduces reasonable overhead, its security guarantee is weak where the attacker could overwrite the shadow status by reallocating the freed object. A more effective approach proposed is to never reuse freed memory [11] or delay the free of memory [57], so that the attacker would not be able to corrupt freed objects. Other than that, once an object is freed, one could nullify all its existing references [17,31,53], which also prevents use-after-free fundamentally.

2.2 Heap Memory Allocator

Heap memory allocators manage dynamic "global" memory and aim for quick allocation/deallocation with minimal memory waste. This summary presents three prevalent types.

Firstly, sequential-fit allocators use a freelist connecting all freed memory objects. When an allocation request is made, the allocator searches the freelist until an adequately sized object is found. It splits larger memory objects, reallocating the excess back into the freelist, and merges neighboring freed objects to minimize fragmentation. Secondly, Segregated List allocators (seglist) manage an array of freelists, each holding freed objects of identical size. Allocation and deallocation require identifying the corresponding freelist for the requested size, avoiding the need for splitting and merging but introducing additional steps. Lastly, the buddy system allocator, similar to seglist, also maintains freelists for varying sizes. When the freelist for a requested size is empty, the allocator splits a larger object to fulfill the request. On deallocation, it reconsolidates the remaining portion back into a larger object.

3 Assumptions & Threat Model

`CAMP` focuses on detecting heap errors, including spatial and temporal heap errors. We assume the target program (written in low-level language) is compiled by `CAMP` and contains at least one heap-based memory vulnerability. As our work focuses on protecting userspace applications, the security of lower-level kernel is out of the scope. In our threat model, the attacker, who is aware of the deployment of `CAMP`, has access to the heap vulnerability and is seeking to exploit it for privilege escalation.

4 CAMP

4.1 A vulnerable toy program

List 1 illustrates a toy program that contains two heap memory corruption vulnerabilities. Specifically, the program allocates 16 bytes of memory (line 2), then accesses the memory object at index 32 which exceeds the boundary of the

```
1  void main() {
2      char *buf = malloc(16);
3      buf[32] = 'x';
4      free(buf);
5      buf[1] = 'y';
6  }
```

Listing 1: A toy vulnerable example.

```
1   void main() {
2       char *buf = malloc(16);
3       __escape(&buf, buf);
4       __check_range(buf, &buf[32], sizeof(char));
5       buf[32] = 'x';
6       // free buf, which neutralizes the dangling
    ↪  pointer stored in &buf
7       free(buf);
8       __check_range(buf, &buf[1], sizeof(char));
9       buf[1] = 'y';
10  }
```

Listing 2: The toy program with CAMP's protection.

memory range (line 3), causing a heap memory overflow. It is noted that this heap overflow could not be detected by ASAN because the overflow skips the red zone. In line 4, the memory object is freed, which makes the pointer buf a dangling pointer. After this, the dangling pointer is dereferenced (line 5), resulting in a use-after-free memory corruption.

4.2 CAMP's Protection

Briefly speaking, CAMP instruments the program to detect memory corruption and prevent exploitation. List 2 illustrates the toy program protected by CAMP. In the following, we describe how CAMP protects the vulnerable toy program.

Pointer Validation. CAMP detects heap overflow by validating result pointers from pointer arithmetic, making sure no out-of-bound pointer is generated. This is achieved by adding a check instruction at the site of the pointer arithmetic to query the runtime and verify that the result pointer is within the range of the base pointer. As demonstrated in List 2, pointers &buf[32] and &buf[1] are generated based on the buffer buf in lines 5 and 9, respectively. CAMP automatically instruments check instructions in lines 4 and 8 to validate these pointers. If the runtime detection determines that &buf[32] is an out-of-bound pointer, CAMP will abort the execution to prevent exploitation. Note that the query requires CAMP to maintain awareness of each memory allocation and its associated memory range, which is recorded during runtime for each heap allocation (line 2).

Neutralizing Dangling Pointers. CAMP prevents use-after-free by neutralizing dangling pointers. At runtime, CAMP constructs the point-to relation by instrumenting the program. When a memory object is freed, CAMP could look up the built point-to relation and identify the dangling pointers to the freed memory. By neutralizing those dangling pointers, use-after-free access is no longer possible. Copying pointers (i.e., *pointer escapes* [51]) is tracked to build the point-to relation.
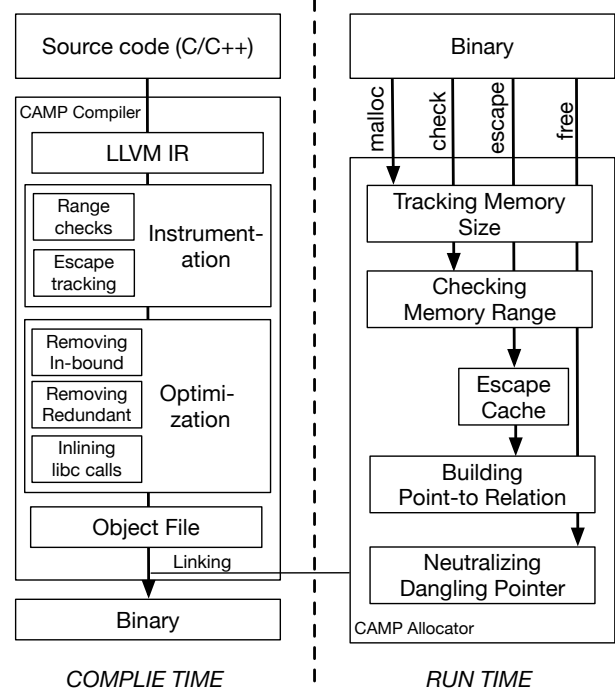

Figure 1: The design overview of CAMP.

For example, the program copies the heap pointer to the variable buf (line 2 of List 2), which is a pointer escape and CAMP instruments an escape tracking instruction after that (line 3). The escape tracking instruction takes as input the address of the variable and the pointer, annotating which address contains a reference to the memory allocation. After this, the program frees the memory (line 7). Inside the free, CAMP will identify the existing dangling pointers to the freed memory and neutralize them as non-congenial. As a result, the variable buf no longer references the freed memory and the program crashes when it is dereferenced (line 9).

4.3 Design Overview

Figure 1 shows the core components of CAMP, which consists of a compiler and a memory allocator. The CAMP compiler, which is built on top of LLVM, takes the source code as inputs and outputs binaries linked with CAMP allocator. During the compile time, the compiler first translates the source code into LLVM IR and then instruments the range checking and escape tracking instruction to defend against heap memory corruption. After this, the compiler applies several novel compiler optimizations to reduce the overhead of protection, without sacrificing security guarantees. At run-time, CAMP's allocator handles the heap memory allocation/deallocation request. In addition, it provides support for instrumented instructions. Specifically, the allocator tracks the memory range for each allocation, so that the allocator could validate the bound information of pointers for each range checking

query. The allocator also handles the escape tracking instruction to build the point-to relation. With this, whenever a memory object is freed, it is capable of neutralizing the corresponding dangling pointers, preventing the UAF access from dangling pointers.

4.4 Compiler Instrumentation

Instrumenting Range Checking. As our program model does not allow casting integers to pointers, initial pointers are all from either explicit memory allocation (i.e., malloc), or taking the address from global or stack variables. [41] Those pointers then are used through *pointer arithmetic* to create new pointers, then to access memory. The range checking in `CAMP` is to ensure that all pointers after arithmetic are still in-bound, therefore, preventing heap overflow. As is shown in List 1, the range checking takes three arguments, which are the base pointer *src*, the result pointer *dst* of the arithmetic, and *dst*'s type size *size*, respectively. The run-time will assert that the memory ranging from *dst* to *dst* + *size* is within the memory range of *src*.

It is noted `CAMP` only protects heap memory, so validating non-heap memory pointer is redundant. To save unnecessary validation, `CAMP` performs dataflow and alias analysis on the LLVM IR to determine the point-to relation of pointers. If a pointer could be determined at compile time that it does not point to the heap, `CAMP` will not instrument range checking for it.

Instrumenting Escape Tracking. The inserted escape tracking allows `CAMP` to build the point-to relation of memory objects at run-time. We follow a similar approach introduced in CARAT [51, 52] and DangNull [31] to insert the tracking after *pointer escapes* (i.e., copying pointers). As is described in Section 4.2, the tracking takes the copied pointer and its stored address as arguments. Different from CARAT and DangNull, which instrument all the potential pointer escapes, `CAMP` skips pointer escapes if the pointer of which is determined not to reference the heap at compile time. As the goal of `CAMP` is protecting heap memory error, skipping those non-heap point-to relations does not jeopardize the security but helps `CAMP` obtain better performance.

4.5 Runtime Support

The runtime of `CAMP` provides underlying support of `CAMP`'s instrumentation. The performance of executing inserted instruction is critical to the overall performance of `CAMP`. Assuming that a program has *m* pieces of memory allocated, and `CAMP` maintain the record of the allocated memory in a linked list. In the worst case, performing a single range checking will cause $O(m)$ time complexity, which apparently would introduce unattainable runtime overhead. Likewise, a dummy design that records *n* pointer escapes in a linked list

would introduce $O(n)$ time complexity when freeing an object. In the following, we describe how our design cooperates with the allocator to provide fast runtime support.

Seglist Allocator. As we mentioned in Section 2, a Seglist Allocator uses different free lists for different sizes of memory objects. For example, tcmalloc [25], a Segregated List Allocator developed by Google, uses *span* as the basic memory management unit. Each span manages a size class of memory objects on several continuous memory pages. The spans are stored in a page table where the page is used as the key. Whenever allocating/deallocating a memory object, tcmalloc could find its span, and then retrieves the freelist with constant time complexity.

`CAMP`'s allocator leverages the design of the Segregated List Allocator to provide fast runtime support. For each span, `CAMP` records the size of memory objects as one of the metadata. Since the seglist allocator splits the memory page equally into objects, given a heap pointer denoted as *ptr*, we could first calculate the index of the object with:

$$idx = (ptr - page\_base)/size$$

where *idx* is the object index to the span, *page_base* represents the start address of the page in the span, and *size* refers to the size of the object. With this information, the lower bound of the memory range can be identified as *page_base* + *idx* * *size* and the upper bound as *page_base* + (*idx* + 1) * *size*. This straightforward approach results in constant time complexity for pointer validation.

Maintaining Point-to Relation. `CAMP` differs from DangNull [31] and CARAT [51, 52] in its approach to encoding the point-to relation. While DangNull and CARAT use a red-black tree structure, which optimizes the time complexity of finding a relation to $O(logN)$, `CAMP` integrates the point-to information into the seglist allocator, optimizing cost to constant time complexity. Specifically, `CAMP`'s seglist allocator maintains an escape table for each span, which is a map of object indices to their escape lists. The escape lists are linked-list structures that chain corresponding escapes to their dedicated objects. When an escape tracking call is made, `CAMP` calculates the memory object's index in the span, retrieves its escape list from the table, and inserts a record into the list. Upon a memory object is freed, `CAMP`'s allocator checks its escape list and neutralizes all the existing dangling pointers to the memory object.

To further boost the overall performance of `CAMP`, we design a cache mechanism for maintaining the point-to relation. New point-to relations are temporarily stored in the cache until it becomes full, at which point the records are transferred to the allocator in a batch, while skipping any duplicates. This cache design boosts runtime speed and reduces memory overhead, particularly in scenarios where the program operates repeatedly in the same block and creates similar point-to relations.

```c
1  struct obj {
2      int a;
3      int b;
4  };
5  struct obj* bar() {
6      // type-casting from void* to obj*
7      struct obj *o = malloc(sizeof(struct obj));
8      __check_range(o, o, sizeof(struct obj));
9      ...
10 }
11 int foo(struct obj *ptr) {
12     __check_range(ptr, &ptr->a, sizeof(ptr->a));
13     ptr->a = 1;
14     __check_range(ptr, &ptr->b, sizeof(ptr->b));
15     ptr->b = 2;
16 }
```

Listing 3: An example of optimizing structure field access checks.

4.6 Compilation Optimization

One of the CAMP's characteristics is that it leaves a large room for potential optimization during the compilation time. This characteristic could boost performance significantly and, at the same time, preserve security. In the following, we describe three optimization algorithms we design for CAMP.

Optimizing range checks with type information. A naive design against out-of-bound access is to have range checks on every pointer arithmetic to ensure that pointers do not exceed memory bounds. As an example, consider the function foo in List 3. The variable ptr in lines 13 and 15 involves two pointer arithmetics, and CAMP must insert range checks for the result pointers (lines 12 and 14) for security purposes. An optimization strategy is that if the compiler knows the pointer is in-bound, the validation of which could be removed to improve the performance. However, the compiler typically lacks information about the memory range of a pointer, making it challenging to determine which pointers are within bounds.

To apply the optimization, we use type information to determine the memory range of a pointer during the compilation process. This involves not only validating pointer arithmetic, but also validating type-casting operations to ensure the memory space referenced by a typed pointer is adequate for its corresponding type. For example, in the code shown in List 3, the function bar allocates a new object obj (line 7). The return type of malloc is void*, not struct obj*, so the compiler inserts a type-casting instruction, after which CAMP inserts a range check to ensure the memory space is sufficient to hold the structure obj (line 8). With this type-casting validation, the compiler can safely infer that the memory space of a typed pointer is at least its type size. As a result, the compiler can conclude that pointers referring to the structure field are in-bound. Therefore, CAMP can remove the range checks in lines 12 and 14. Further, The compiler can also guarantee that the memory of pointer o (line 6) is at least

---

Algorithm 1: Removing Redundant Validation

1 **Input:** A function F ;
2 **Output:** A set of pointer to be validated S ;
3 **Initialize:** *NewPointerSet* = **getNewPointerSet**{F} ;
4 $In = Out = changeSet = dict()$;
5 **foreach** $ptr \in NewPointerSet$ **do**
6   **if** *pointerMapbase[**base**(ptr)] == NULL* **then**
7         $pointerMapbase[\textbf{base}(ptr)] = set()$;
8     $pointerMap[\textbf{base}(ptr)].add(ptr)$;

9 $In = pointerMap$;
10 $changeSet = In - Out$;
11 **while** $changeSet \neq \emptyset$ **do**
12   **foreach** $key, val \in In$ **do**
13   **if** $\exists p, p' \in val$ **and *RedundantPair(p, p')*** **then**
14          $val.remove(p')$;
15          $val.p.offset = \textbf{MAX}(p.offset, p'.offset)$;
16          $Out[key] = val$;
17   **break**;
18   **else**
19          $Out[key] = val$;

20    $changeSet = In - Out$;
21    $In = Out$;
22 **foreach** $key, val \in Out$ **do**
23    $S.add(val)$;

---

the size of its type, so the range check in line 7 can also be optimized and eliminated.

The optimization is made possible by the unique design of security checks in CAMP, which eliminates the risk of accessing to a dangling pointer, such as ptr in function foo. This allows for aggressive optimization of range checks for field pointers without the concern of casing a false-negative for use-after-free vulnerability. As we will show in Section 6, the proposed optimization approach above dramatically improves CAMP's runtime performance, especially for programs that contain pointers associated with types. It should be noted that for pointers that have no type information (e.g., void*, char *) or their type sizes could not be determined at compilation time (e.g., elastic objects [16]), we do not apply the optimization above. As ASAN's security design cannot guarantee those conditions, applying this optimization to it may result in use-after-free and out-of-bound access.

Removing Redundant Instructions. A redundant instruction in this work refers to a range checking that validates pointers that have been validated or an escape tracking that builds point-to relation that has been recorded. The optimization opportunity for this is to remove the redundant ones to obtain better performance.

Intuition suggests that if two range checks validate the same result pointers from pointer arithmetic, one of them could be removed. For example, in List 4, the pointer &↪ ptr->mem[1] in line 9 and 11 aliases. We could simply

```
1  struct obj {
2      char *mem;
3  };
4  void foo(struct obj *ptr, bool flag) {
5      __check_range(ptr->mem, &ptr->mem[0x100], 1);
6      ptr->mem[0x100] = 'x';
7      if (flag) {
8          ptr->mem[0x30] = 'y';
9          ptr->mem[0x1] = 'y';
10     }
11     ptr->mem[0x1] = 'z';
12     ptr->mem++;
13 }
```

Listing 4: Example codes of applying eliminating redundant optimization.

remove the validation for &ptr->mem[1] in line 9. Further, CAMP's security design allows merging several pointer validations into one validation. Initially, CAMP needs to validate the result pointers &ptr->mem[0x100] (line 6), &ptr->mem[0x30] (line 8), and &ptr->mem[0x1] (line 11), respectively. However, if pointer &ptr->mem[0x100] is in-bound, the other two pointers must be in-bound as well. Therefore, we could remove their validations and move the validation of &ptr->mem[0x100] to line 5, as is illustrated in List 4. Formally, given two result pointers $ptr1$ and $ptr2$ from the same base pointer ptr, their validation can be merged if the following function returns True.

> **function** REDUNDANTPAIR($ptr1, ptr2$)
>   **if** $ptr1.offset >= ptr2.offset$ **then**
>   **if** $dominate(ptr1, ptr2)$ **or**
> $post\text{-}dominate(ptr1, ptr2)$ **then**
>   **return True**
>   **return False**
> **end function**

The $offset$ in the first condition represents the maximum access offset from the base pointer. As such, the condition requires $ptr2$ to be in the range of $[ptr, ptr1]$. The second condition ensures the redundancy of validation, where the two validations will be executed together. We follow Algorithm 1 to remove redundant pointer validation. The algorithm takes as input a function F, and outputs a set of pointers to be validated S. We first collect all the result pointers (line 3) and categorize them into a map according to their base pointers (line 5 to 8), where the key is the base pointer, value is the set of result pointers. Then we follow the fix-point algorithm [13] to apply the optimization. In each iteration, we go through the element in the map (line 12). if we find two pointers that satisfy the condition of the redundancy (line 13), we remove the latter one (line 14) and adjust the remaining one's offset with their maximum value (line 15). After this, we update the output of iteration (line 16) and exit the loop to start the next iteration. Noted that if no redundancy pair is found, the output will be just the input of the iteration (line 19). When there is no redundancy pair

left, meaning that the fixed point is reached. Then we collect the remaining pointers from the *out* to S (line 22 to 23). List 4 showed CAMP's instrumentation after applying this optimization, where only the validation in line 5 is preserved, but the security guarantee remains. Note that applying this optimization to redzone-based protection (e.g., ASAN) may remove out-of-bound access checks. Assuming that ptr->mem ↪ is a 0x20 bytes memory chunk with 0x10 bytes red zone, normally the overflow will be detected through the check for ptr->mem[0x30] in line 8. However, applying the proposed optimization will remove this valid check but keep the one for ptr->mem[0x100], which skips the red zone and miss the overflow detection, thereby leading to false negatives.

In line 12 of List 4, the pointer ptr->mem is self-updated. If we break this statement down, there are three operations involved. First, the program retrieves the pointer stored in address &ptr->mem. Then, it creates a new pointer based on the retrieved pointer, where a range checking will be inserted. This check guarantees the newly created pointer references to the same memory object as the old one does. Then, the new pointer is copied into address &ptr->mem, after which CAMP will insert an escape tracking to the pointer copying. One key observation is that the address &ptr->mem should have been initialized somewhere before, which means such a point-to relation must have been recorded. Because the memory object of the new pointer is not changed, recording the same point-to relation is redundant. Therefore, we could optimize this escape tracking for performance without sacrificing precision. Our optimization strategy is to identify those escape pairs that are doing self-updating.

Merging Runtime Calls. Ideally, for the same memory pointer (including alias), the validation can be optimized with the aforementioned approach into a single range check. However, if the pointer arithmetic is dynamic, where the result of pointer arithmetic cannot be determined at the compilation time, CAMP has to instrument different range checks for them. List 5 illustrates such example codes. Line 5 and 7 access the same array ptr that does not have type information. Besides, the access is dynamically based on the runtime value i and j, making the optimization of removing redundancy not applicable. To this end, CAMP needs to instrument both Line 5 and 7 respectively. Each time executing a range check, CAMP has to switch its context into the library, query the memory range, and then validate the pointer. This process is time-consuming. One strategy to optimize this is that we could merge the range checks since they share the same base pointer. This can reduce the cost of constantly switching contexts and save the time of querying the memory range.

To do this, we first follow the same approach in Algorithm 1 of constructing a *pointer map* where the pointer arithmetic with the same base pointer is categorized into the same group. Then, for each group, we go through the CFG of the function and find their nearest **dominator** instruction, where a range query is inserted to initialize the memory chunk's

```
1  void foo(char *ptr, int i, int j) {
2      unsigned int start, end;
3      __get_range(ptr, &start, &end);
4      assert(&ptr[i]>=start && &ptr[i]+1<end);
5      ptr[i] = 'x';
6      assert(&ptr[j]>=start && &ptr[j]+1<end);
7      ptr[j] = 'y';
8  }
```

Listing 5: Example codes of applying merging runtime calls optimization.

CWE (number)	Good Test (Selected/Total/Passed)	Bad Test (Selected/Total/Passed)
Buffer Overflow(122)	3870/3870/3870	2308/3870/2308
Double Free(415)	820/820/820	820/820/820
Use After Free(416)	394/394/394	288/288/288
Invalid Free(761)	288/288/288	288/288/288

Table 1: Security evaluation of CAMP on Juliet Test Suite.

range variables. After this, the original range check then is replaced with assertion to ensure the boundary. List 5 also shows the result after applying this strategy. Line 3 queries the memory range of `ptr` and initializes the memory ranges into variables `start` and `end`. After this, validation for `&ptr[i]` and `&ptr[j]` is done through two assertions in line 4 and line 6.

5 Implementation

In this section, we describe our implementation of CAMP's compiler and the allocator.

CAMP Compiler. CAMP's compiler is built on top of LLVM 12 compiler framework. We implement the instrumentation and the optimization in an LLVM pass, loadable by clang.

To defend against heap overflow, it instruments all pointer arithmetic and type-casting instructions. In the context of pointer arithmetic, the compiler collects the `getelementptr` instruction from the LLVM IR, which represents the sole pointer arithmetic instruction as CAMP prohibits casting integers to pointers. CAMP adds a range checking instruction after every `getelementptr` instruction, with three inputs: the base pointer, the result pointer, and the type size of the result pointer. The `getelementptr` instruction's pointer operand, result value, and type size serve as these inputs respectively. To determine if the `source` operand refers to the heap, CAMP backtracks it following LLVM's SSA to identify its origin. If the source is a stack or global variable, the system presumes it doesn't refer to the heap, skipping the `getelementptr` instruction's instrumentation. The compiler also instruments the `bitcast` instruction, which symbolizes type-casting in LLVM. For each type-casting instruction, CAMP adds a checking instruction with two inputs - the result pointer and its type size - to ensure adequate memory for the object.

Following CARAT's approach [51,52] to tracking escapes, CAMP instruments store instructions if their value operand type is a pointer. If the escape is on heap memory, the compiler inserts an escape tracking CALL instruction before the store instruction.

Memory Allocator. The allocator is built on tcmalloc [25]. Tcmalloc maintains a page table mapping page addresses to a `span`. Notably, a span can handle multiple continuous mem-

ory pages for larger objects. To ensure any heap pointer can find its `span`, we register every memory page used by tcmalloc in the page map. Each span is supplemented with two metadata to facilitate CAMP's runtime checks: the object size and a reference to the escape pointer array containing linked escapes to the objects it manages. For each page, we compress its span's start address and size class into an 8-byte unit and map it into the *size class map*. With this design, CAMP can retrieve the necessary data and validate pointers in constant time. For unrecorded spans, CAMP resorts to the original routine of retrieving the `span`.

To accommodate coding styles that use out-of-bound pointers as memory boundaries, we reserve one-byte memory at the end of each allocation, ensuring such pointers remain in-bound. Each pointer escape prompts CAMP to create a record containing the pointer's location, stored in the span of the referenced object. Linked escape records for the same object are created, and all related records are freed when a memory object is freed. During this process, CAMP neutralizes any pointers still referencing the object. CAMP features a temporal escape array cache mechanism. Every time a new escape track is invoked, CAMP checks the array for an identical record before appending. Once full, all records are committed to the span and the array cleared, ensuring all dangling pointers are neutralized.

6 Evaluation

In this section, we first evaluate CAMP's effectiveness in detecting heap overflow and use-after-free memory corruption on a standard vulnerability benchmark and a set of real-world vulnerabilities. Then, we show CAMP's protection in detail with two case studies on real-world vulnerabilities. After this, we discuss CAMP's security capability with a comparison to related works. Finally, we show CAMP's performance/memory overhead using SPEC CPU benchmarks and real-world applications, and demonstrate its advantage against tools from the most recent research. All the experiments were conducted on a bare-metal machine configured with Ubuntu 22.04 system, 12th Gen Intel i7-12700 CPU at 4.9 GHz, 32GB RAM, and 1T SSD storage.

CVE/Issue ID	Application	Bug Type	CAMP	ASAN--	Memcheck	DangNull	MarkUs	Delta pointer
CVE-2015-3205	libmimedir	Use-After-Free	✔	✔	✔	✔	✔	/
CVE-2015-2787	PHP 5.6.5	Use-After-Free	✔	✔	✔	✖	✖	/
CVE-2015-6835	PHP 5.4.44	Use-After-Free	✔	✔	✔	✔	✖	/
CVE-2016-5773	PHP 7.0.7	Use-After-Free	✔	✔	✔	✔	✖	/
Issue-3515 [50]	mruby	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2020-6838	mruby	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2021-44964	Lua	Use-After-Free	✔	✔	✔	Build Fail	✔	/
CVE-2020-21688	FFmpeg	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2021-33468	yasm	Use-After-Free	✔	✔	✔	✔	✔	/
CVE-2020-24978	nasm	Use-After-Free	✔	✔	✔	✖	✔	/
Issue-1325664 [6]	Chrome	Use-After-Free	✔	✔	✔	Build Fail	✖	/
CVE-2022-43286	Nginx	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2019-16165	cflow	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2021-4187	vim	Use-After-Free	✔	✔	✔	✖	✔	/
CVE-2022-0891	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2022-0924	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-19131	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-19144	libtiff	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2021-4214	libpng	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-3156	sudo	Heap Overflow	Run Well	✔	✔	/	/	Build Fail
CVE-2018-20330	libjpeg-turbo	Heap Overflow	✔	✔	✔	/	/	✔
CVE-2020-21595	libde265	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2020-21598	libde265	Heap Overflow	✔	✔	✔	/	/	Build Fail
Issue-5551 [4]	mruby	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2022-0080	mruby	Heap Overflow	Run Well	✔	✔	/	/	Build Fail
CVE-2019-9021	PHP	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2022-31627	PHP	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-32281	gravity	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2020-15888	Lua	Heap Overflow	✔	✔	✔	/	/	Build Fail
CVE-2021-26259	htmldoc	Heap Overflow	✔	✖	✔	/	/	Build Fail
CVE-2022-28966	Wasm3	Heap Overflow	✔	✔	✔	/	/	Build Fail

Table 2: The security evaluation results of CAMP and related tools on real-world vulnerabilities. ✔ represents that the corresponding tool successfully detected the memory corruption in the vulnerability. ✖ indicates the tool failed to detect the memory corruption that happened. "/" represents the tool does not support protecting the corresponding type of vulnerability. "Run Well" means the application runs well without causing any memory corruption with the PoC input. "Run Fail" represents that the tool failed to run due to compatibility issues. "Build Fail" means the tool failed to compile the targeted application to enforce protection.

6.1 Security Evaluation

Juliet Test Suite. To evaluate the effectiveness of CAMP's protection, we conducted experiments using the Juliet Test Suite, following recent works [23, 32]. The Juliet Test Suite includes test programs for various vulnerability types, each with both bad and good tests. The proof-of-concept (PoC) in the bad tests triggers the corresponding vulnerability, while the PoC in the good tests does not.

As CAMP focuses on preventing heap memory corruption, we only included heap-related vulnerability types from the Juliet Test Suite. It is important to note that CAMP's overflow prevention mechanism is based on the size of the memory object. Therefore, heap overflows that do not exceed the memory boundary are treated as benign because they do not corrupt other memory objects. Following this logic, we used

a customized ASAN[1] to exclude the bad tests in which the overflow is contained within the memory object.

Regarding CAMP's use-after-free protection, it neutralizes dangling pointers, so instead of reporting an error, dereferencing a dangling pointer will cause the program to abort without a report. To evaluate the test cases in the use-after-free category, we used a gdb script to confirm that the abortion of the program was due to the dereference of neutralized dangling pointers.

The results of the selected tests are presented in Table 1, which shows the selected vulnerability type, the number of

---

[1]A customized ASAN that rounds up each allocation and includes a large red zone to prevent overflows from affecting adjacent objects and escaping detection. When evaluating tests for overflow, if ASAN does not report any issues, it indicates that the overflow occurs within an object. In such cases, the test can be safely removed from the test suite.

selected tests, the total number of tests, and the number of tests passed. Some of the tests originally categorized as Heap-Based Buffer Overflow do not contain heap overflow, such as cases `Heap_Based_Buffer_Overflow__c_src_char_cat_*`. These tests trigger a buffer overflow when copying data from the heap to the stack, causing a stack overflow rather than a heap overflow. These cases were excluded from the selected test cases using the customized ASAN. In all selected tests, CAMP passed without producing any false-positives or false-negatives.

Real-world Applications. In addition to the Juliet Test Suite, we also evaluated the security of CAMP by using a set of real-world vulnerabilities. We included all the real-world vulnerabilities used in [11]. Additionally, we collected other types of vulnerabilities from the CVE database [2, 3, 10]. Table 2 lists the selected vulnerabilities. Our dataset includes 14 use-after-free and 18 heap overflow vulnerabilities across 19 applications, including language interpreters, commonly used libraries, browsers, web servers, and commonly used UNIX tools. Our goal was to evaluate CAMP's effectiveness in preventing different heap memory corruption vulnerabilities and its scalability over various real-world applications. As a comparison, we also evaluate related tools, including ASAN/ASAN-- [62], Memcheck [27], DangNull [31], MarkUs [12], and Delta Pointer [30] for their effectiveness in detecting and preventing heap errors.

Table 2 presents the security evaluation results on real-world applications. CAMP successfully detected and prevented all use-after-free vulnerabilities. In the case of heap overflow vulnerabilities, CAMP was able to detect 16 out of 18 and report them. The other two ran well and did not cause any reports or crashes. But upon manual investigation using a debugger, we found that the overflow had occurred, but the memory bounds were not exceeded due to the rounded-up memory allocation of CAMP's seglist allocator. As a result, the overflow is mitigated and no memory corruption happened. We argue that these two cases do not count as false negatives of CAMP as the exploitation is prevented.

For tools providing a comparable level of heap protection, Memcheck was able to detect all the heap errors in the dataset. ASAN-- reported all the heap errors except CVE-2021-26259 [1]. The reason behind this is that ASAN uses red zone to detect heap overflow vulnerabilities. However, CVE-2021-26259 is a non-linear heap overflow that skips the red zone, thus ASAN--'s detection is defeated. Unlike ASAN--, CAMP was able to detect this case successfully as it detects heap overflow based on the memory boundary, making it impossible for non-linear heap overflows to bypass its protection. We argue that CAMP's protection is stronger and more robust than those two tools. As discussed in prior work [32], ASAN and Memcheck's use-after-free protection could be defeated if the attacker reclaims the freed memory that the dangling pointer refers to, thus enabling a possible exploitation against use-after-free vulnerabilities. As a com-

parison, CAMP mitigates the heap error fundamentally by naturalizing all dangling pointers.

For tools offering partial heap protection, DangNull and Delta Pointer showed limited compatibility support. 4 out of 14 use-after-free cases were not able to build with DangNull. Among 10 use-after-free cases that could be successfully compiled, only 4 of them were detected. Others just crashed with the PoC input as if there is no protection. Note that DangNull has a similar use-after-free protection scheme as CAMP, but it fails to detect 6 cases that CAMP could detect. This is because DangNull only tracks the point-to relation from heap to structured object on the heap, which will miss the detection if the dangling pointer is on stack/global memory, or the use-after-free object has no type information. MarksUs showed better compatibility support but failed to detect 6 out of 14 use-after-free vulnerabilities. Delta Pointer showed even worse compatibility support, which can only compile 5 out of 17 cases, but all the out-of-bound in compatible cases were detected. However, due to its design weakness, it is not capable of detecting buffer underflow.

We argue that CAMP provides a much more comprehensive heap error detection capability when compared to similar tools. Our evaluation demonstrates that CAMP outperforms the combination of partial heap protection tools (such as Delta Pointer + DangNull/MarkUS) as well as ASAN--. It is worth mentioning that CAMP does not miss any bugs that ASAN-- can detect. In addition, CAMP outperforms ASAN-- from the following aspects. First, ASAN--'s detection for use-after-free is fragile, which could be bypassed by reclaiming free memory with new heap allocation. As such, some use-after-free vulnerabilities in allocation-intensive applications could not be detected. We observed such a case when evaluating ASAN-- in 600.perlbench of SPEC CPU2017, where ASAN-- missed the bug but CAMP did not. Besides, some use-after-free POCs that accidentally re-occupy the freed memory will not be reported by ASAN--. We tweaked the original POC for CVE-2015-2787 and CVE-2015-6838 and found that the use-after-free was missed by ASAN-- as the freed memory is reclaimed. Second, ASAN-- utilizes a red zone mechanism to flag out-of-bound access. In the case of non-linear overflow, the overflow may jump over the red zone and thus fail overflow detection. Although it is unknown how frequently non-linear overflow may happen in the real world, we argue that the missed detection of such bugs will cause serious security issues. For example, CVE-2021-26259 is a non-linear overflow that would lead to code execution.

6.2 Performance Evaluation

In the following, we evaluate CAMP's performance on the two SPEC CPU benchmarks with the comparison with related tools. Then, we evaluate the effectiveness of each design component of CAMP, including the compiler optimiza-

Benchmark	Time and Memory Overhead				
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Memcheck**
600.perlbench_s	237.95% / 2241.12%	76.95% / 366.92%	143.59% / 358.20%	644.00% / 4.15%	3496.46% / 138.97%
602.gcc_s	78.56% / 135.52%	83.61% / 63.42%	99.47% / 62.77%	-	2888.13% / 30.42%
605.mcf_s	14.62% / 31.55%	24.45% / 3.61%	27.88% / 3.61%	109.33% / -4.24%	601.05% / 22.68%
623.xalancbmk_s	138.94% / 1220.66%	107.86% / 428.07%	109.41% / 433.51%	81.67% / 8.60%	4962.60% / 98.81%
625.x264_s	75.07% / 12.68%	62.26% / 13.52%	75.92% / 13.26%	90.94% / -3.55%	2070.57% / 56.96%
631.deepsjeng_s	1.58% / 0.00%	44.23% / -0.23%	64.08% / -0.23%	18.85% / -0.25%	3251.34% / 25.34%
641.leela_s	3.02% / 514.19%	13.97% / 2832.83%	17.33% / 2833.72%	6.65% / -17.52%	4163.69% / 262.82%
657.xz_s	7.79% / 0.00%	17.45% / 2.98%	13.40% / 2.98%	14.61% / -0.70%	718.87% / 24.45%
619.lbm_s	1.34% / 0.01%	37.32% / 5.94%	29.38% / 5.94%	34.14% / -0.36%	2907.53% / 25.98%
638.imagick_s	45.47% / 0.07%	17.23% / 4.46%	28.56% / 4.47%	21.70% / -2.00%	4452.66% / 22.93%
644.nab_s	62.55% / 26.13%	35.18% / 67.52%	35.14% / 66.63%	1988.66% / -1.34%	3722.35% / 31.80%
Geomean	21.27% / 127.47%	38.27% / 104.72%	44.78% / 104.35%	65.31% / -1.94%	2546.88% / 56.49%

Table 3: The relative time and memory overhead of CAMP, ASAN --, ASAN, ESAN, and Memcheck on SPEC CPU2017. "-" indicates the tool failed to run the corresponding benchmark.

Benchmark	Metric	CAMP	LowFat	Delta Pointer	DangNull	FreeGuard	MarkUs	FFMalloc
SPEC CPU2006	Time	54.92%	160.62%	37.39%	39.99%	10.40%	15.84%	9.50%
	Mem	237.67%	38.60%	0.01%	158.52%	70.89%	2.96%	27.57%
SPEC CPU2017	Time	21.27%	96.96%	-	28.61%	8.23%	11.90%	10.94%
	Mem	127.47%	54.35%	-	314.13%	29.23%	32.35%	62.00%

Table 4: The relative time and memory overhead of CAMP, LowFat, Delta Pointer, DangNull, FreeGuard, MarkUs, and FFmalloc on SPEC CPU2006 and SPEC CPU2017.

tion, and the customized allocator. Finally, we evaluate CAMP on the real-world applications.

SPEC CPU Benchmark. We compared the performance of CAMP with related tools including ASAN [47], ASAN -- [62], ESAN [20], Softbound+CETS [41], and Memcheck [27]. All programs in the benchmark suite were compiled using default configurations and used reference input. Besides, all tools were configured to ignore detected errors to avoid termination. Note that, to ensure a fair comparison, we only enabled ASAN -- and ASAN's heap error detector. Unfortunately, ASAN and Memcheck were unable to compile and run the omnetpp and dealII programs, and therefore, they were excluded from the evaluation. Softbound+CETS showed limited support, failing to compile all programs in SPEC CPU2017 and only supporting 7 programs in SPEC CPU2006, so we only compare with it in SPEC CPU2006. Our evaluation of the tools did not take PACMem [32] into account, as it requires specialized hardware (ARM PA) for the detection of heap memory errors. For baseline evaluation, we used tcmalloc as the default allocator for better comparison as CAMP's allocator is based on tcmalloc. For each tool, we ran the benchmark 10 times and reported the average result to minimize the randomness.

The evaluation results in terms of time overhead and memory overhead are presented in Table 3 for the SPEC CPU2017 benchmark suite. Each row in the table represents a specific application benchmark, with the benchmark name listed in the first column and the subsequent columns display the relative time and memory measured compared to a baseline. Following the most recent works [23, 32], we utilized geometric mean value to represent the average overhead of each tool. CAMP exhibits the best runtime speed compared to other tools, with an average overhead of 21.27%, while ASAN --, ASAN, and ESAN have overhead rates of 38.27%, 44.78%, and 65.31%, respectively. Memcheck has the worst runtime performance, which introduces 2546.88% overhead compared to the baseline. In terms of memory overhead, CAMP has a higher rate of 127.47% compared to roughly 104% for ASAN and ASAN --. ESAN has the best memory overhead performance, with -1.94% which is mainly cased by the difference of allocators. As we discussed in Section 4, CAMP tracks different forms of object point-to relation, which is more comprehensive and could obtain better security guarantee as we showed in Section 6.1. The cost of this design is that the performance overhead will be higher on allocation-intensive programs, such as 600.perlbench_s and
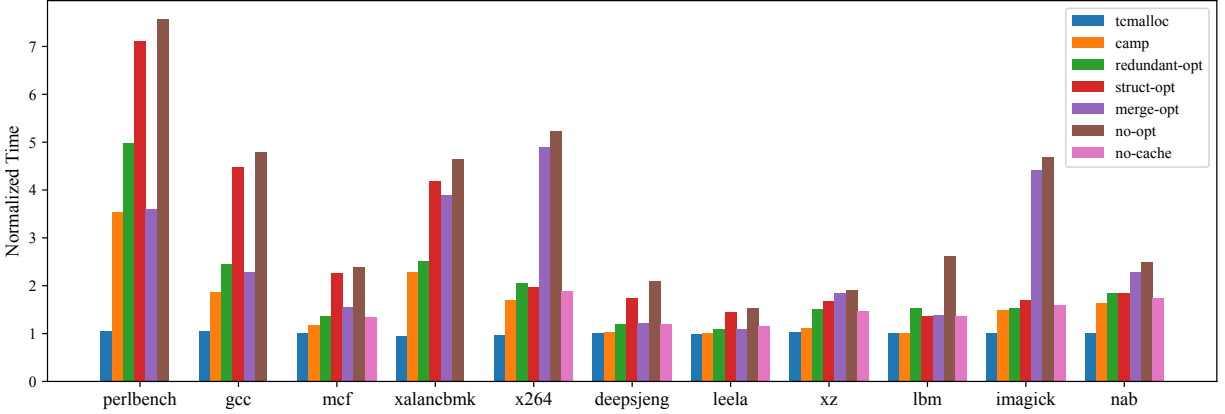
Figure 2: Evaluation result of CAMP breakdown on SPEC CPU2017. From left to right, the bars show the normalized time of tcmalloc replacement, CAMP, CAMP with each optimization disabled, and CAMP without optimization.

623.xalancbmk_s. We found that CAMP recorded a number of escape tracks, resulting in notable time and memory overhead. The average time and memory overhead of CAMP without those two cases will be reduced to 13.21% and 44.24%, respectively. For case 638.imagick_s and 644.nab_s, where CAMP's time overhead is higher than ASAN--, we observed the proposed optimizations are less effective in those two cases. Specifically, the type-based range checks removing and redundant checks removing are less effective on them. For the remaining cases in the benchmark, CAMP's time overhead is minimal with an even lower geomean value of 8.86%.

As for SPEC CPU2006, the evaluation results could be found in Table 8 and Table 9, we put them in Appendix due to the limited space. CAMP still outperforms all other tools in terms of runtime speed. Specifically, it introduces an overhead of 54.92%, while ASAN --, ASAN, ESAN, Soft-Bound+CETS, and Memcheck have the overhead of 56.77%, 67.02%, 123.08%, 319.75%, and 1990.02%, respectively. As we discussed before, CAMP performs worse in allocation-intensive programs. As such, for cases like 400.perlbench, 482.sphinx3, 453.povray, 453.povray, 473.astar, 483.xalancbmk ↪ , CAMP traces a large number of runtime point-to relations and costs more memory on maintaining them, which eventually slows down the overall speed. The time and memory overhead without those cases will be reduced to 34.52%, and 39.3%, respectively. Since the SPEC CPU2006 contains more allocation-intensive cases than SPEC CPU2017, CAMP reports higher overhead on SPEC CPU2006.

We further compare CAMP with other tools that offer partial memory protection, following the same setup of the previous evaluation on the SPEC CPU Benchmarks, we present the results in Table 4. LowFat, which only provides out-of-bounds (OOB) protection, exhibits slower performance than CAMP, with time overheads of 160.62% and 96.96% on the two SPEC benchmarks, respectively. Delta Pointer, which could

only run SPEC CPU2006, incurs 37.39% time overhead. In terms of use-after-free (UAF) protections, FreeGuard, MarkUs, and FFMalloc introduce approximately 10% overhead on the benchmarks. DangNull exhibits 28.61% overhead on SPEC CPU 2017, which is slightly high than CAMP. It is noted that DangNull only includes a similar but weaker use-after-free protection scheme, highlighting the fast runtime speed provided by the CAMP allocator while offering a stronger security guarantee. We argue that CAMP is the optimal solution for achieving comprehensive heap protection in comparison to the other tools currently available. Firstly, as opposed to tools that only offer partial heap protection, CAMP presents a more holistic solution, with the ability to detect both spatial and temporal heap errors. Moreover, even when considering the combination of the faster Out-of-Bounds protection tool (e.g., Delta Pointer) and Use-After-Free protection tool (e.g., FFMalloc), the overhead incurred by CAMP is remarkably similar to their collective overhead (54.92% vs. 46.89%). Despite this, CAMP outperforms this combination in several ways. On the one hand, due to design limitations, Delta Pointer is only capable of detecting overflow errors, leaving underflow errors undetected, a challenge that CAMP effectively addresses. On the other hand, Delta Pointer's support is restricted to a 32-bit address space, allowing for a maximum of only 4GB of memory. In parallel, FFMalloc may demand a considerable amount of physical memory if the program continues to persist [23]. Hence, their combined use could rapidly exhaust the available 4GB of memory, resulting in incompatibility issues.

Component Evaluation. The configuration of CAMP contains various components, including compiler optimization and the integration of a seglist allocator. To determine the impact of each component on performance, we conducted evaluations using different setups of CAMP.

To analyze the impact of CAMP's customized allocator, we

| Benchmark | Time (s) | | Overhead |
	Native	CAMP	
cfrac	2.91	3.82	31.27%
espresso	3.62	3.62	0.00%
barnes	1.35	1.34	-0.74%
redis	2.66	2.68	0.75%
leanN	25.35	26.18	3.27%
alloc-test1	2.99	3.06	2.34%
alloc-testN	2.91	3.42	17.53%
sh6benchN	2.41	2.39	-0.83%
sh8benchN	5.79	8.3	43.35%
xmalloc-testN	2.664	2.306	-13.44%
cache-scratchN	0.43	0.44	2.33%
Geomean	-	-	9.79%

Table 5: Time Overhead on mimalloc-bench. Native represents using the default allocator – ptmalloc, CAMP means using its customized seglist allocator based on tcmalloc.

conducted an evaluation using mimalloc-bench to compare the allocator differences. The evaluation results are presented in Table 5. The "Native" column represents the performance of the system's native allocator (ptmalloc), while the "CAMP " column indicates the results obtained with CAMP's customized allocator. The two allocators exhibited distinct behavior across different test cases. For instance, in the case of cfrac, the Native allocator was 31.27% faster than CAMP, whereas in the xmalloc-testN case, it was 13.44% slower. Overall, CAMP's allocator demonstrated a 9.79% slowdown compared to the native allocator.

To evaluate the effectiveness of compiler optimization, we disabled each optimization one by one in different configurations of CAMP, represented as struct-opt, redundant-opt, merge-↪ opt and no-opt in Figure 2. To gain insight into the role of the allocator cache design, we measured the performance of CAMP allocator cache disabled. Finally, to assess the impact of the seglist allocator, we compared the results to a baseline using tcmalloc. These setups and their results on SPEC CPU2017 are presented in Figure 2.

Our evaluation results confirm that the seglist allocator offers minimal benefit. The tcmalloc baseline, compared to default ptmalloc, only improves average speed by 2.26%. We then took a further look at the contribution of allocator cache design, we found three programs (perlbench, gcc ↪ , xalancbmk) experienced memory exhaustion and were unable to complete the test, so they were excluded from Figure 2. The remaining programs resulted in an average overhead of 40.34%, nearly double the overhead with cache enabled (20.94%). Our investigation revealed that these three programs made many repeated point-to relationships, leading to a high consumption of memory for metadata maintenance. Specifically, all these programs contain a language

| System | Output (req/s) | Latency (µs) | | | | |
		Average	50%	75%	90%	99%
Native	150,368	643.23	625	635	649	910
CAMP	108,322	880	850	870	910	1070
ASAN--	103,688	930	880	900	960	1860
ASAN	97,095	970	900	930	1040	1910

Table 6: CAMP, ASAN, and ASAN--'s output and latency evaluation results on Nginx. In the Latency column, the "Average" represents the average latency of the requested connection, the others show the latency distribution.

interpreter that builds ASTs during input parsing, which creates connections that require CAMP to maintain point-to records. In gcc, as the compiler optimization progressed, more nodes are connected, leading to more point-to records being constructed, and eventually, the program exhausted all the memory available. In perl, after parsing the AST, the execution of the AST utilized the heap as a stack, leading to repeated references that further constructed more pointer escapes. These cases demonstrate the effectiveness of the cache design. With it, repeated point-to relations could be saved, thus reducing both memory and time overhead.

In addition, we found that the proposed compiler optimization significantly reduced the performance overhead. The CAMP without compiler optimization imposed an overhead of 204.09%, while the default CAMP had an overhead of 20.94%. Among the tested programs, imagick saw the best optimization results, with its overhead reduced from 368.18% to 48.47%. Upon analyzing the breakdown of each optimization, we found that the greatest impact came from the structure optimization. If this optimization is disabled, the overhead increases from 20.94% to 113.39%. While disabling redundant optimization, and merging runtime call optimization introduced 64.20%, and 95.73%, respectively.

Nginx. To evaluate the performance of CAMP on a large-scale, real-world application, we conducted experiments on Nginx v1.22.1 using the wrk v4.2.0 benchmarking tool. For these experiments, we configured the tool with 8 threads, 100 connections, and a test duration of 60 seconds. To ensure consistency, we repeated the test 30 times and recorded the average results. The findings are presented in Table 6. On average, CAMP introduces a 27.96% overhead on Nginx's request output, In terms of latency, CAMP adds 36.81% more time. The results reflect CAMP's efficiency on real-world applications with mild overhead. As a comparison, ASAN and ASAN-- incur a 35.43% and 31.04% overhead on request output and have a latency overhead of 50.80% and 44.58%, respectively.

Chromium. In addition to Nginx, we also evaluated the performance of CAMP on Chromium. We used three popular browser benchmarks: Kraken, SunSpider, and Lite Brite.

Benchmark	Time (*ms*)		Overhead
	Native	CAMP	
kraken	1069	1722	61.09%
sunspider	521	813	56.05%
Lite Brite	2930	5520	88.40%
Geomean	-	-	67.14%
google.com	1101	1427	29.61%
facebook.com	831	1199	44.28%
amazon.com	2298	3120	35.77%
openai.com	1444	1791	24.03%
twitter.com	1479	1708	15.48%
gmail.com	1691	2032	20.17%
youtube.com	2143	2628	22.63%
wikipedia.org	984	1535	56.00%
Geomean	-	-	28.59%

Table 7: CAMP's performance evaluation results on the Chromium browser. In the Benchmark column, kraken, sunspider and Lite Brite are three browser benchmarks, whereas the following are websites used to measure the loading time of the browser.

Furthermore, we measured the loading time for websites, as this metric is highly relevant to end users' browsing experience. To measure loading time, we utilized a browser extension and recorded the average loading time for the 8 most popular websites according to the Top Websites Ranking [9]. Each benchmark experiment was repeated 30 times, and the mean value was calculated to mitigate any random variations. The evaluation results are presented in Table 7, with the average overhead of CAMP represented by the geometric mean.

We observed that CAMP introduced a 67.14% overhead on the three browser benchmarks. However, the loading time for web pages increased by only 28.59% on average. It is important to note that these benchmarks focus on specific components of the browser, which may not fully reflect the overall performance. In contrast, loading web pages involves JavaScript engine execution, DOM processing, and other factors, thus it is more representative of real-world browsing. In this regard, we argue that the overhead introduced by CAMP to Chromium is minimal.

7 Discussion

False Positive and False Negative. As a pointer-based protection approach, CAMP shares similar weaknesses with prior works [19, 41, 56]. First, C/C++ allows the use of out-of-bound pointers as the memory boundary, which may cause CAMP to generate false positives. To mitigate this, we reserve additional memory space allocated for each allocation so that those boundary pointers will be still in bound, as is discussed

in Section 5. This approach of changing the memory layout effectively mitigated this issue, thus we did not find any false positives in our evaluation. We consider this to be more of a compatibility issue that should be resolved at the source code level. This not only eliminates the potential for false positives, but also strengthens the security of the code.

Second, in C/C++, integers may be used for pointer arithmetic instead of actual pointers. This poses a challenge for CAMP as it may result in out-of-bounds pointer accesses that go undetected. To address this issue, we enforce a policy within the compiler that prohibits the casting of integers to pointers. Thus, the generation of out-of-bounds pointers from integers can be prevented. In cases where developers have to perform such casts, they could explicitly indicate their intention using compiler attributes to disable the policy locally.

Preventing In-bound Overflow. The design of CAMP's overflow detection is based on memory boundaries, meaning that it only identifies overflows that cross the boundaries as violations. Therefore, CAMP's design shares a similar weakness with prior works [36, 41, 56], as it cannot be used to prevent in-bound heap overflows. However, we found that, by leveraging a proper implementation, CAMP could mitigate some in-bound overflow. Specifically, if pointer arithmetic is performed on an array of a structure, we can use the array size to validate the pointer and ensure that the result pointer stays within the array of the structure, as such it mitigates the in-bound overflow with the type information. However, we do not claim this as CAMP's capability of preventing in-bound overflow. Because the type information is not always available, besides, the array may be dynamic thus CAMP has no clue how to validate it to make sure no overflow is inside the structure. Therefore, we consider the prevention of in-bound overflows to be a future research direction.

Protecting Shared Library. As CAMP uses the compiler for instrumentation, it naturally supports the protection of any programs that the compiler can build, including shared libraries. To achieve this protection, users must re-compile the shared library using the CAMP compiler. By doing so, the library will be instrumented with function calls to the CAMP runtime, ensuring that protection is enforced.

Supporting Multi-threaded Programs. CAMP is compatible with multi-threaded programs. Data race could be avoided by employing locks for the metadata in each span. In addition, we evaluated several multi-threaded programs (e.g., PHP, mruby, Lua, FFmpeg, Chrome, Nginx) in Section 6 and did not observe any incompatibility – CAMP successfully detected the bug in those programs without reporting any false positive.

8 Related Work

Safe allocator. To combating heap memory corruption attacks [16, 24, 33, 54, 58], various safe heap allocators [11, 18, 23, 29, 35, 39] are proposed. Specifically, FFmalloc [11] proposes a one-time memory allocation. DieHarder [44] randomizes heap address space to unstabilize heap exploitation. Oscar [18] prevents temporal memory error with shadow memory allocated for each heap object, thus detecting dangling pointer access. Markus [12] employs a strategy that quarantines freed memory to eliminate any lingering dangling pointers. DangZero [23] interacts with the kernel page table to implement an alias-based UAF detection mechanism, ensuring the virtual memory is never reused. FreeGuard [49] improves performance with freelist and optimizes shadow memory scheme. `CAMP` differs from those works as it does not exclusively rely on allocators to enable protection.

Pointer invalidation. Several works [15, 21, 30, 31, 36, 41, 42] detect memory errors through pointer invalidation. For instance, CETS [42] employs a lock-and-key identifier-based approach to track separate metadata for each pointer to detect dangling pointers. Undangle [15] uses dynamic taint tracking to identify and eliminate unsafe dangling pointers. DangNull [31] nullifies pointers when their associated objects are freed. In regards to spatial safety, Redfat [21] combines redzone and low-fat pointers to detect buffer overflow. Delta Pointer [30] introduces a pointer tag to invalidate overflow pointers, thereby preventing errors. Softbound [41] utilizes shadow memory to track memory bounds and employs runtime checks for efficient overflow detection. `CAMP` achieves full heap protection with pointer validation, however, unlike the aforementioned works, `CAMP` cooperates with the compiler and the allocator to optimize pointer invalidation.

Memory Sanitizer. Memory sanitizers typically offer full heap error detection. ASan [47] employs shadow memory and redzones for detecting temporal and spatial errors. To reduce its overhead, ASan`--` [62] and SANRAZOR [60] propose several compiler optimizations to reduce instrumented checks on memory access, without security compromise. FuZZan [26] presents new metadata structures to decrease memory management overhead. CUP [14] proposes a hybrid metadata scheme that supports all program data including globals, heap, and stack. EffectiveSan [20] presents a dynamic type system to detect memory errors, but it has some limitations on detecting temporal errors. Memcheck [27], part of Valgrind [43], detects full memory errors. `CAMP` differentiates itself from those approaches by its protection scheme, thereby offering superior speed.

Hardware-assisted protection. There are also a bunch of works [19, 22, 28, 32, 46, 55, 56, 61] that leverage hardware to enforce memory safety. Specifically, Low-Fat [19] extends the pointer representation with base and bounds information so that the runtime or hardware can prevent spatial safety violations. In-Fat [56] enhances the hardware-assisted tagged-pointer scheme, employing three complementary object metadata schemes to decrease the number of pointer tag bits required. Several works, including PtAuth [22], AOS [28], and PACMem [32], utilize the Pointer Authentication Code (PAC) feature of ARM to identify memory errors. HeapCheck [46] leverages pointer bits in 64-bit systems for a bounds table, aiding in memory error detection with an 8 KB on-chip SRAM cache. BOGO [61] employs Intel MPX for memory safety, while CHEx86 [48] innovatively targets memory errors via microcode-level code instrumentation. In contrast, `CAMP` doesn't need extra hardware support.

9 Conclusion

Mitigating memory corruption on the heap is a complex task. Existing techniques to address heap memory corruption either provide limited protection or introduce significant runtime overhead, making their adoption in real-world products challenging. By leveraging a carefully designed code instrumentation and a customized allocator, `CAMP` provides comprehensive protection against heap memory corruption. The instrumentation imposes some runtime overhead, but we demonstrate that this overhead can be significantly reduced through a series of optimization strategies that eliminate and consolidate unnecessary instrumentations. Our evaluation of `CAMP` using a large-scale real-world application and SPEC CPU Benchmarks shows that the performance impact is significantly reduced. The low overhead, combined with `CAMP`' ability to effectively detect and prevent heap memory corruption, makes it a promising solution for safeguarding programs against heap memory corruption.

References

[1] CVE-2021-26259: A flaw was found in htmldoc in v1.9.12. Heap buffer overflow. https://nvd.nist.gov/vuln/detail/CVE-2021-26259.

[2] CVE details: The ultimate security vulnerability datasource. https://www.cvedetails.com/.

[3] CVE program. https://cve.mitre.org/.

[4] Heap-based Buffer Overflow in mruby. https://huntr.dev/bounties/4458e0b9-0ad3-4036-a032-1b3c4705b889/.

[5] Heap Cookies for memory protection. https://fuzzysecurity.com/tutorials/mr_me/3.html.

[6] Issue 1325664: Security: pdfium use-after-free in v8. https://bugs.chromium.org/p/chromium/issues/detail?id=1325664.

[7] Linux Kernel CVE Changes. https://www.linuxkernelcves.com/.

[8] The source code of CAMP. https://github.com/cla7aye15I4nd/CAMP.

[9] Top Websites Ranking. https://www.similarweb.com/top-websites/.

[10] VulnDB: The most comprehensive vulnerability database and timely source of intelligence available. https://vuldb.com/.

[11] *Preventing Use-After-Free Attacks with Fast Forward Allocation.*, 2021.

[12] Sam Ainsworth and Timothy M Jones. Markus: Drop-in use-after-free prevention for low-level languages. 2020.

[13] Vasile Berinde and F Takens. *Iterative approximation of fixed points*, volume 1912. Springer, 2007.

[14] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. Cup: Comprehensive user-space protection for c/c++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 381–392, 2018.

[15] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 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*, 2012.

[16] Yueqi Chen, Zhenpeng Lin, and Xinyu Xing. A systematic study of elastic objects in kernel exploitation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[17] Oracle Corporation. Nullify references after reclaiming memory. https://docs.oracle.com/cd/E19159-01/819-3681/abebi/index.html, 2010.

[18] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.

[19] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[20] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[21] Gregory J Duck, Yuntong Zhang, and Roland HC Yap. Hardening binaries against more memory errors. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 117–131, 2022.

[22] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. {PTAuth}: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[23] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[24] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, 2018.

[25] Google Inc. Design of TCMalloc from Google. https://google.github.io/tcmalloc/overview.html.

[26] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. {FuZZan}: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263, 2020.

[27] Nicholas Nethercote Julian Seward. Memcheck: a memory error detector. https://valgrind.org/docs/manual/mc-manual.html.

[28] Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. Hardware-based always-on heap memory safety. In

2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 1153–1166. IEEE, 2020.

[29] Dmitry Vyukov Kostya Serebryany. Scudo Hardened Allocator. https://llvm.org/docs/ScudoHardenedAllocator.html.

[30] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[31] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[32] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[33] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[34] Zhiqiang Lin, Ryan D Riley, and Dongyan Xu. Polymorphing software by randomizing data structure layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 6th International Conference, DIMVA*. Springer, 2009.

[35] Beichen Liu, Pierre Olivier, and Binoy Ravindran. Slimguard: A secure and memory-efficient heap allocator. In *Proceedings of the 20th International Middleware Conference*, 2019.

[36] Hongyu Liu, Ruiqin Tian, Bin Ren, and Tongping Liu. Prober: practically defending overflows with page protection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[37] John McDonald Mark Dowd and Justin Schuh. Magic Value: Potential Mitigations for Heap Overflow. https://cwe.mitre.org/data/definitions/122.html.

[38] John McDonald Mark Dowd and Justin Schuh. Protect Out-Of-Bound by Validating Pointer. https://cwe.mitre.org/data/definitions/823.html.

[39] Microsoft. Daan Leijen. 2020. Mimalloc. https://github.com/microsoft/mimalloc.

[40] M. Miller. A snapshot of vulnerability root cause trends for Micrsoft Remote Code Execution (RCE) CVEs, 2006 through 2017. https://twitter.com/epakskape/status/984481101937651713.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, 2010.

[43] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[44] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[45] Mitch Phillips. Design of Redzone in Address Sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm.

[46] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. Heapcheck: Low-cost hardware support for memory safety. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–24, 2022.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[48] Rasool Sharifi and Ashish Venkat. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 762–775. IEEE, 2020.

[49] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[50] Clayton Smith. Heap use-after-free in mruby. https://github.com/mruby/mruby/issues/3515.

[51] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler-and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[52] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, et al. Carat cake: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[53] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[54] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. {MAZE}: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664, 2021.

[55] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

[56] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[57] Toshihiro Yamauchi and Yuta Ikegami. Heaprevolver: Delaying and randomizing timing of release of freed memory area to prevent use-after-free attacks. In *Network and System Security: 10th International Conference, NSS 2016, Taipei, Taiwan, September 28-30, 2016, Proceedings 10*. Springer, 2016.

[58] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1111–1128, 2020.

[59] Google Protect Zero. 0day "In the Wild". https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY/edit#gid=0.

[60] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. Sanrazor: Reducing redundant sanitizer checks in c/c++ programs. In *OSDI*, 2021.

[61] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2019.

[62] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

Benchmarks	Time					
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Softbound+CETS**	**MEMCHECK**
400.perlbench	300.23%	256.68%	285.49%	629.16%	-	3407.29%
401.bzip2	48.90%	43.98%	48.22%	114.70%	354.91%	912.38%
403.gcc	67.86%	175.04%	173.97%	600.95%	-	1877.21%
429.mcf	27.25%	24.06%	34.85%	158.27%	634.40%	303.66%
433.milc	9.04%	38.82%	48.64%	50.76%	239.02%	1194.67%
445.gobmk	28.14%	36.86%	38.84%	52.46%	356.30%	2418.05%
456.hmmer	119.45%	90.07%	89.83%	270.44%	477.35%	1647.21%
458.sjeng	10.26%	40.23%	48.64%	13.35%	264.45%	2329.69%
462.libquantum	18.74%	14.75%	20.11%	197.61%	-	553.84%
464.h264ref	126.17%	89.88%	125.67%	326.47%	-	2689.47%
470.lbm	7.39%	25.82%	28.41%	51.63%	141.17%	5236.58%
482.sphinx3	114.08%	44.70%	52.97%	199.55%	-	4207.37%
444.namd	90.43%	75.60%	82.01%	57.64%	-	4076.65%
450.soplex	64.42%	42.82%	44.45%	128.66%	-	1518.52%
453.povray	113.95%	105.89%	150.95%	266.29%	-	5385.34%
473.astar	112.80%	30.62%	37.97%	80.75%	-	1085.92%
483.xalancbmk	297.90%	166.85%	203.05%	48.80%	-	5158.20%
Geomean	54.92%	56.77%	67.02%	123.08%	319.75%	1990.02%

Table 8: Time overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. "-" means the case where the tool failed to run the benchmark.

Benchmarks	Memory					
	CAMP	**ASAN−−**	**ASAN**	**ESAN**	**Softbound+CETS**	**MEMCHECK**
400.perlbench	1522.09%	339.64%	285.49%	1.90%	-	163.14%
401.bzip2	0.05%	2.75%	48.22%	-0.98%	127.90%	33.59%
403.gcc	109.95%	187.98%	173.97%	-6.78%	-	44.12%
429.mcf	51.66%	12.02%	34.85%	-0.56%	396.77%	2.29%
433.milc	379.00%	36.77%	48.64%	-1.58%	89.01%	10.50%
445.gobmk	143.05%	612.69%	38.84%	-9.29%	638.64%	230.80%
456.hmmer	4.32%	1061.04%	89.83%	-39.34%	-12.09%	197.22%
458.sjeng	-0.10%	-1.69%	48.64%	-3.26%	1.18%	44.82%
462.libquantum	-0.41%	185.86%	20.11%	-21.56%	-	48.34%
464.h264ref	18.82%	330.51%	125.67%	-19.84%	-	120.71%
470.lbm	-0.01%	11.47%	28.41%	-1.41%	-1.55%	34.39%
482.sphinx3	4093.04%	650.28%	52.97%	-4.03%	-	240.28%
444.namd	3.63%	9.93%	82.01%	-8.45%	-	101.86%
450.soplex	12.65%	44.26%	44.45%	-20.79%	-	21.20%
453.povray	7101.36%	2020.28%	150.95%	-12.61%	-	1188.33%
473.astar	1840.76%	176.09%	37.97%	5.29%	-	50.27%
483.xalancbmk	1800.53%	187.89%	203.05%	14.54%	-	77.51%
Geomean	237.67%	181.81%	180.94%	-8.45%	102.25%	97.14%

Table 9: Memory overhead of CAMP, ASAN--, ASAN, ESAN, Softbound+CETS, MemCheck on the SPEC CPU2006. "-" means the case where the tool failed to run the benchmark.