ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization

Zheng Yu
Northwestern University

Ganxiang Yang
Northwestern University

Xinyu Xing
Northwestern University

Abstract

In software development, the prevalence of unsafe languages such as C and C++ introduces potential vulnerabilities, especially within the heap, a pivotal component for dynamic memory allocation. Despite its significance, heap management complexities have made heap corruption pervasive, posing severe threats to system security. While prior solutions aiming for temporal and spatial memory safety exhibit overheads deemed impractical, we present ShadowBound, a unique heap memory protection design. At its core, ShadowBound is an efficient out-of-bounds defense that can work with various use-after-free defenses (e.g. MarkUs, FFMalloc, PUMM) without compatibility constraints. We harness a shadow memory-based metadata management mechanism to store heap chunk boundaries and apply customized compiler optimizations tailored for boundary checking. We implemented ShadowBound atop the LLVM framework and integrated three state-of-the-art use-after-free defenses. Our evaluations show that ShadowBound provides robust heap protection with minimal time and memory overhead, suggesting its effectiveness and efficiency in safeguarding real-world programs against prevalent heap vulnerabilities.

1 Introduction

In the domain of software development, a notable majority of programs are developed using unsafe languages, such as C and C++, which while granting detailed control over low-level operations and memory management, also invariably expose them to a range of vulnerabilities. The heap, acting as a critical component for dynamic memory allocation, assumes an essential role in these programs by managing data objects of sizes that are not fixed or predetermined. However, the complexity of heap management has made it a hotspot for vulnerabilities, with heap corruption emerging as a widespread issue [56, 60]. Not only are heap vulnerabilities prevalent, but they also pose severe threats to system security, as they can be exploited to manipulate data, bypass security defenses, and execute arbitrary code, often having serious consequences for affected systems [30, 68, 74]. Consequently, comprehensive heap memory protection is imperative.

A comprehensive heap memory protection system should encompass both the temporal and spatial memory safety, ensuring that use-after-free and out-of-bound vulnerabilities are rendered completely unexploitable through its protection mechanisms. Numerous prior works [24, 39, 45, 46, 51, 75, 76] target this objective, all of which successfully provide both temporal and spatial memory safety. However, all of them suffer from an overhead exceeding 1.5x, rendering them impractical defenses in real-world programs. Additionally, some of them mainly used for bug detection or debugging, are not robust in terms of security [56]. Although these tools are not efficient, some prior works [17, 70, 71] can provide robust temporal memory safety with a low time overhead. Consequently, we believe it is possible to build a comprehensive memory protection system with them.

Inspired by these observations and thoughts, we propose ShadowBound to provide a comprehensive heap memory protection mechanism. The core of ShadowBound is an efficient out-of-bounds defense, designed to prevent the exploitation of out-of-bounds bugs by instrumenting boundary checks into programs. Moreover, it is capable of seamlessly integrating with various Use-After-Free (UAF) defenses without encountering compatibility issues. While the concept behind ShadowBound is straightforward, developing such an out-of-bounds defense is not straightforward. We pinpoint two challenges that necessitate addressing. Firstly, the challenge lies in minimizing the time overhead of the out-of-bounds defense. Secondly, compatibility issues with UAF defenses present a hurdle. Most state-of-the-art UAF defenses necessitate the introduction of a new allocator. Thus, it is beneficial for the out-of-bounds defense to avoid any requirements for the allocation algorithm, consequently contracting the design space. Almost all previous works [27, 33, 34, 36], which focused on spatial memory safety as discussed in Section 2, encountered similar challenges.

We address these challenges through two perspectives. Initially, we designed a shadow memory-based metadata management mechanism to store the boundary of each pointer's corresponding heap chunk. This strategy ensures that ShadowBound necessitates only a single load instruction and a handful of arithmetic instructions to extract boundaries during checks. Moreover, the shadow memory is orthogonal to the allocation algorithm, thereby enabling its integration with various allocators, specifically those utilized in UAF defense. Such a design lays the foundational framework for constructing an efficient and comprehensive heap memory protection mechanism.

Building upon the established foundation, we deployed a series of customized compiler optimizations, specifically tailored for boundary checking. Unlike traditional methods, which instrument boundary checking at the point of pointer dereference, ShadowBound instruments these checks at the site of pointer arithmetic, thereby facilitating easier optimization. Generally, a compiler can retrieve more information at the pointer arithmetic site. At the pointer dereference site, the pointer might be passed externally, which hinders the compiler from fetching any of the pointer's properties. Conversely, the compiler can consistently utilize the computing process information at the pointer arithmetic site. We designed five optimizations based on this information, which significantly reduce the time overhead of ShadowBound.

We implemented ShadowBound atop LLVM 15 and integrated it with three state-of-the-art UAF defense mechanisms: PUMM [71], FFMalloc [70], and MarkUs [17]. To understand its security and practicality implications, ShadowBound was applied to common benchmarks and real-world programs. Specifically, we utilized ShadowBound to safeguard 19 programs against 34 exploitable out-of-bound bugs. With the protection provided by ShadowBound, all exploits were successfully mitigated. Additionally, we undertook synthesis vulnerability testing, generating 244 inputs to trigger out-of-bounds bugs in various ways. Furthermore, ShadowBound successfully prevented the triggering of these bugs. For performance evaluation, we assessed ShadowBound using the SPEC CPU2017 and SPEC CPU2006 benchmark suites, as well as three real-world applications: Nginx, Chakra, and Chromium. ShadowBound exhibits a 5.72% and 10.58% time overhead on SPEC CPU2017 and SPEC CPU2006, respectively, and introduces negligible overhead to the tested real-world programs.

In summary, this paper provides the following contributions:

- We introduce ShadowBound, which employs a novel metadata design utilizing a compact method to encode boundary information in shadow memory. This approach enables ShadowBound to quickly fetch the boundaries of a pointer, thereby making it compatible with various Use-After-Free (UAF) defenses and providing both spatial and temporal safety with minimal overhead.
- We propose a series of novel optimization techniques customized for boundary checking at pointer arithmetic sites, significantly reducing time overhead. These optimizations have been implemented in ShadowBound, based on LLVM 15. An ablation study demonstrates their effectiveness in reducing time overhead.
- Through a thorough evaluation, ShadowBound has been proven to offer robust spatial memory protection in various environments. It consistently maintains minimal time and memory overhead across a spectrum of benchmarks and real-world applications, affirming its full compatibility with three state-of-the-art UAF defenses.

	Associated Address	Access	Exp?
S1	Inaccessible	Crash	✖
S2	Accessible & No Overlap	Benign	✖
S3	Accessible & Overlap	Data Leakage or Corruption	✔

Table 1: Consequences of heap out-of-bounds bugs.

2 Background & Design Overview

2.1 Heap Memory Protection

A comprehensive heap memory protection mechanism should ensure both temporal and spatial memory safety for a program. In the realm of temporal memory safety, several Use-After-Free (UAF) defenses stand out for their remarkable performance, notably state-of-the-art solutions such as MarkUs [17], FFMalloc [70], and PUMM [71]. These tools provide full temporal memory safety, setting them apart from partial or probabilistic memory protection methods [43,47,54,55]. Moreover, they outperform many of their predecessors. For instance, MarkUs leverages a garbage collection algorithm to identify live pointers, thereby preventing the creation of dangling pointers and offering improved performance compared to earlier solutions [40,53]. FFMalloc introduces an efficient one-time-allocation (OTA) allocator strategy, surpassing older solutions like Oscar [22] and other OTA allocators [23]. PUMM, in contrast, utilizes static analysis to pinpoint code units in charge of specific tasks, deferring the reallocation of memory freed by the active unit until it finishes its operations.

However, even with these advanced temporal memory protection mechanisms in place, the challenge of developing a comprehensive heap memory protection mechanism persists due to the absence of an adequate spatial memory protection mechanism. For instance, tools like MEMCHECK [46], TAILCHECK [27], ASAN [51] and its variants [75,76] are redzone-based spatial memory error detection systems, which are vulnerable to bypasses [56]. While ESAN [24,36], SoftBound [45], and PACMEM [39] offer robust spatial memory protection, they still suffer from significant time overheads. Moreover, the designs of TAILCHECK, ASAN, and ESAN require a custom allocation algorithm to adjust the heap layout for out-of-bound checking. This can lead to conflicts when UAF defense introduces new allocators, such as MarkUs and FFMalloc. Although tools like DeltaPointer [33] and SGXBound [34] are relatively more efficient in terms of time overhead, they limit memory allocation to 4GB. Considering that UAF defense also amplifies memory overheads, this hinders their utility in large-scale applications.

2.2 Out-Of-Bounds Exploitation

Out-of-bounds vulnerability is a specific manifestation of spatial errors. Depending on the behavior of a program and the logic of the memory allocator, the consequences of a heap out-of-bounds bug can vary significantly. We present a comprehensive summary of these potential consequences in Table

1, where the associated address refers to the memory address involved in an out-of-bounds read or write instruction. If the system has removed permission to access the associated address (S1), the out-of-bounds access will trigger a page fault and result in a crash. Another consequence is when the address remains accessible but still falls within the original heap chunk, without overlapping with other heap chunks or the freed regions (S2). This situation can occur because most allocators align the size of each heap chunk to 8 or 16 bytes, so they allocate more memory space than the program initially requested. Consequently, out-of-bounds access within such a region may happen. In the final scenario, the memory remains accessible, and the associated address may overlap with another region (S3), including other heap chunks, freed regions, or even extending beyond the heap boundary.

The bug in S1 is inherently non-exploitable, as it consistently results in a crash. In S2, despite the presence of an out-of-bounds bug within the logic of the program, such out-of-bounds access will never occur due to the extra space allocated by the allocator. In these instances, we can assume that the allocator "fixes" the bug. Conversely, an out-of-bounds error in S3 is notably susceptible to exploitation. An attacker can strategically manipulate the layout of the heap, causing the associated address to overlap with critical data. This could lead to data leakage, corruption, or even enable privilege escalation, compromising the entire system. To exploit an out-of-bounds bug in S3, the attacker must first create an out-of-bounds pointer and subsequently attempt to trigger a dereference with this pointer. If we can detect or eliminate the creation of out-of-bounds pointers that may be dereferenced later, any exploitation attempt will be rendered ineffective. ShadowBound follows such a line of thought, which can defend against out-of-bounds exploitation by either transferring the S3 to S1/S2 or detecting all S3.

2.3 Design Overview

ShadowBound diligently works to prevent the exploitation of out-of-bounds bugs by instrumenting boundary checks into programs. It encompasses two principal modules: the metadata management module and the compiler module. The metadata management module maintains shadow memory, recording the boundaries of each pointer's corresponding heap chunk within the shadow memory at allocation sites. Concurrently, the compiler module instruments boundary-checking instructions at pointer arithmetic sites, precluding the generation of out-of-bounds pointers. Additionally, optimization is an essential role of the compiler module; it employs a suite of customized optimization techniques, specifically tailored for boundary checking, which ensures ShadowBound incurs very low time overhead. Notably, both modules are meticulously designed to guarantee seamless integration with various UAF defenses, enabling ShadowBound to collaborate with other UAF defenses to provide exhaustive heap protection.

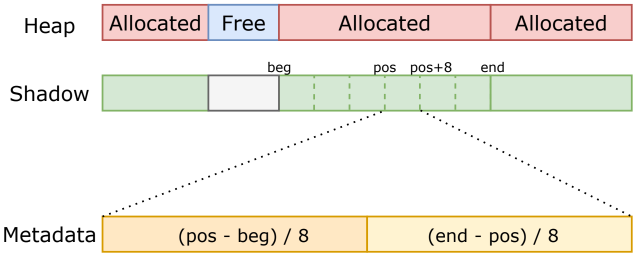

Figure 1: Shadow Memory Mapping and Metadata Layout

3 Threat Model

ShadowBound can be deployed either in conjunction with other UAF defense mechanisms or independently. When deployed alongside other UAF defenses, it is presumed that the target program contains one or more heap out-of-bounds and use-after-free vulnerabilities. If ShadowBound is used independently, the assumption is limited to the presence of heap out-of-bounds vulnerabilities. In this threat model, an attacker can only attempt to exploit these vulnerabilities to potentially escalate privileges. Our goal is to prevent these vulnerabilities to being exploitable.

We assume the shadow memory will not compromise the security for three reasons. First, doing so would require exploiting vulnerabilities in our concise and robust shadow memory management code, which comprises less than 100 lines, giving us confidence in its security. Second, compromising the system in another way would necessitate exploiting a program vulnerability that allows an underflow write, overcoming a substantial 4TB space between the heap and shadow memory. This level of intrusion would require sophisticated exploitation primitives, which ShadowBound is designed to detect and thwart preemptively. Finally, the attacker has no chance to infer the shadow memory addresses of objects. Since the attacker can only exploit vulnerabilities to leak some information, similar to the second reason then exploitation will be detected by ShadowBound.

4 Metadata Management

In this section, we introduce our metadata management design, covering its layout, creation process, and how ShadowBound leverages this metadata to enhance program security. We will also discuss the compatibility of metadata management with various UAF defense mechanisms.

4.1 Shadow Memory Layout

ShadowBound utilizes shadow memory to track the boundaries of each pointer's corresponding heap chunk in the program. As shown in Figure 1, ShadowBound maps each aligned 8 bytes of the program's heap memory into 8 bytes of shadow memory. Within this shadow memory representation, the front 4 bytes are dedicated to storing one-eighth of the length from the aligned 8 bytes to the beginning of the respective heap chunk. Conversely, the back 4 bytes are used

```
1 void foo(void *ptr, int n) {
2     bound_check(ptr, ptr + sizeof(int));
3     int *arr = (int *) ptr;
4
5     for (int i = 0; i < n; ++i) {
6         bound_check(arr, arr + i + 1);
7         other_function(&arr[i]);
8     }
9 }
```

Listing 1: Example: Instrumentation of ShadowBound

to record one-eighth of the length from the aligned 8 bytes to the end of the corresponding heap chunk. The creation and initialization of shadow memory occur immediately following the allocation operation. The feasibility of this design stems from two fundamental observations.

Firstly, nearly all mainstream allocators, such as ptmalloc [62], tcmalloc [31], and jemalloc [18], default to 8-byte or 16-byte aligned allocations. This intrinsic behavior ensures that any valid 8-byte aligned memory address always belongs to the same memory chunk. Consequently, there is no need to maintain boundary information for every byte; instead, we can efficiently store the boundary information for every 8-byte aligned block. This approach significantly optimizes memory usage. The second observation is that the maximum single-time allocation size for these allocators is limited to 8 GB (2^{33} bits). AddressSanitizer [51]'s default allocator also has such a feature. Building upon the previous observation, we can deduce that the length between any 8-byte aligned block and the beginning or end of the corresponding chunk is always a multiple of 8. Hence, 30 bits are sufficient to store this length information, and we can use 8 bytes or 64 bits to store both length values.

4.2 Boundary Checking

As mentioned in subsection 2.3, unlike traditional checking methods that instrument checking instructions at the pointer dereference site, ShadowBound instruments these checks at the pointer arithmetic site. LLVM provides two types of pointer arithmetic instructions. The `getelementptr` (GEP) instruction takes a base pointer and a set of indices to calculate the result pointer. If these indices are not properly checked, an attacker can exploit a GEP to perform an out-of-bounds access. The `bitcast` instruction (BC) allows the conversion of a base pointer from one type to a result pointer with another type. However, if the pointer cannot sufficiently accommodate the target type, it may potentially result in an overflow issue. Listing 1 provides an example of how ShadowBound checks these two types of instructions. In line 3 of the code, an attempt is made to convert a `void*` pointer to an `int*` pointer. To ensure that the resulting pointer can hold at least one integer, ShadowBound inserts a check at line 2. In line 6, a new pointer is generated, and ShadowBound inserts a check to ensure that the new pointer and the old pointer are located within the same memory chunk.

```
1 void bound_check(uint64_t old, uint64_t res) {
2     if (!IsHeapAddress(old)) return;
3     uint64_t align = old & ~7;
4     uint64_t shadow = align + OFFSET;
5     uint64_t pack = *(uint64_t*) shadow;
6     uint64_t beg = align - ((pack & 0xffffffff) << 3);
7     uint64_t end = align + ((pack >> 32) << 3);
8     if (res < beg || res >= end)
9         error("Heap out-of-bounds Detected");
10 }
```

Listing 2: Pseudocode for boundary checking

The pseudocode for the boundary check, as shown in Listing 2, describes a checking function that takes two parameters. The function first checks if the pointer is located in the heap region (Line 2). If so, it aligns the base pointer to an 8-byte boundary and calculates the corresponding shadow memory address (Lines 3-4). It then loads the metadata from the shadow memory (Line 5). The next two lines calculate the boundary. Specifically, for the beginning address, the lower 32 bits of the metadata are taken and shifted left by 3 (effectively multiplying by 8) to get the actual distance from the aligned address to the beginning of the corresponding heap, and then this distance is subtracted from the aligned address to determine the beginning address (Line 6). Conversely, for the end address, the upper 32 bits are taken and shifted left by 3 to get the distance from the aligned address to the end of the corresponding heap, then this distance is added to the aligned address to obtain the end address (Line 7). Finally, the function verifies whether the result pointer falls outside these computed boundaries (Line 8). If the result pointer is less than the starting address or greater than or equal to the ending address, it triggers an error, leading to program termination. Due to the boundary-checking code's reliance on just a single load instruction and a handful of arithmetic operations, the verification process is exceptionally quick, resulting in minimal execution time.

4.3 Compatibility with UAF Defense

ShadowBound introduces continuous memory as shadow memory to track the boundaries of each heap chunk. This technique empowers ShadowBound to seamlessly integrate with virtually all UAF defense. Within the scope of this research paper, we have successfully integrated three state-of-the-art UAF defenses into ShadowBound, MarkUs [17], FFMalloc [70], and PUMM [71]. Each of these defenses implements a secure allocator, which means they need to modify the allocation algorithms, potentially resulting in changes to the heap layout. It's noteworthy that the shadow memory operates independently of the allocation algorithm and heap layout. This ensures that ShadowBound does not conflict with the design of secure allocators.

In addition to secure allocators, some research employs pointer invalidation algorithms for UAF defense [38, 65, 72]. While this approach may not currently provide optimal performance, there is potential for improvement in its effectiveness in the future. Consequently, we also demonstrate Shadow-

```
1  struct obj {
2      int x, y, z;
3  };
4  void foo() {
5      char *c = malloc(3 * sizeof(int));
6      struct obj* o = malloc(sizeof(struct obj));
7      ...
8  }
9  void bar(char *c) {
10     c[0] = 'x';
11     c[1] = 'y';
12     c[2] = 'z';
13     escape(c + 1);
14 }
15 void zoo(struct obj *o) {
16     o->x = 1;
17     o->y = 2;
18     o->z = 3;
19 }
```

Listing 3: Example of Runtime-Driven Checking Elimination

BOUND's compatibility with these tools. Typically, these approaches involve tracking all pointers to allocated objects and explicitly invalidating them once the referenced objects are freed, requiring program instrumentation. It's worth noting that SHADOWBOUND also necessitates program instrumentation, but there is no conflict because SHADOWBOUND instruments pointer arithmetic instructions, while they only need to instrument dereference instructions.

5 Compiler Optimization

In this section, we introduce the compiler optimization employed by SHADOWBOUND, which play a important role in reducing time overhead while maintaining robust security.

5.1 Runtime-Driven Checking Elimination

This optimization is based on a simple idea: if each heap chunk has infinite space, out-of-bounds access becomes impossible, rendering all boundary checks redundant and eliminable. The core of this concept relies on the runtime environment's ability to provide the compiler with extra information, enabling it to eliminate specific boundary checks that static analysis techniques alone cannot resolve. By leveraging runtime information, certain optimizations become feasible, empowering the compiler to remove unnecessary boundary checks, and significantly enhancing system efficiency. However, it's impractical to allocate infinite or even very large spaces for every chunk due to the potential for high memory overhead. Therefore, SHADOWBOUND chooses an improved approach to balance time overhead and memory overhead. Specifically, SHADOWBOUND reserves a fixed *n* bytes for every heap chunk, denoted as reserved space. Then, SHADOWBOUND will try to find all eliminable boundary checks using the reserved space provided by the runtime.

As mentioned in subsection 4.2, SHADOWBOUND inserts boundary checks for every pointer arithmetic instruction. Each instruction contains a base pointer argument and generates a result pointer. These two pointers are used as the arguments passed to the boundary checking. Notably, if the offset between the result pointer and base pointer can be confirmed to be less than *n* bytes at compile time, and the result pointer will never be used as a base pointer in another boundary checking, SHADOWBOUND can safely remove the boundary checking. This is because SHADOWBOUND already reserves *n* bytes for every heap chunk, ensuring that every live pointer, which may be used as a base pointer in boundary checking, has at least *n* bytes of space available.

```
struct obj {
    int *a;
    int len;
};
void foo(struct obj *o, int len) {
    // initialize the object
    o->len = len;
    o->a = malloc(len * sizeof(int));
    ...
void bar(struct obj *o) {
    for (int i = 0; i < o->len; ++i)
        o->a[i] = i;
}
```

Listing 4: Example of Security Pattern Identification and Directional Boundary Checking Optimization

We can use Listing 3 as an example to illustrate how the optimization works. In the function `bar`, the argument `c` generates three pointers in lines 10 to 12, all with offsets less than 8. If we set *n* to 8, SHADOWBOUND can safely remove these checks. However, we cannot eliminate the check for line 13 because the pointer `c + 1` is passed to another function, indicating that it may potentially be used as a base pointer for boundary checking in that function. If SHADOWBOUND were to eliminate the boundary check for a pointer that has already pointed to the reserved space, it could lead to false negatives due to the elimination in the `escape` function.

Furthermore, the optimization concept can also be applied to eliminate structure member accesses, as demonstrated in the function `zoo` of Listing 3. The object is allocated at line 6, and the return type of the allocation function is `void*`, not `struct obj*`. Consequently, the compiler inserts a type-casting instruction, which prompts SHADOWBOUND to insert a boundary check to ensure that the allocated memory space can accommodate the structure `obj`. This type-casting validation happens in runtime and allows the compiler to confidently determine that the memory space associated with a typed pointer is at least the size of its type. Consequently, the compiler can infer that pointers referring to structure fields are in-bounds. As a result, SHADOWBOUND can safely eliminate the boundary checks for lines 16 to 18.

5.2 Directional Boundary Checking

The boundary checking of SHADOWBOUND consists of two parts: the underflow check ensures that the memory access address is greater than or equal to the lower boundary of the allocated memory chunk, while the overflow check verifies that the address falls within the upper boundary. We can observe that during boundary checking, the base pointer is always valid. If not, ShadowBound will throw an error when creating the base pointer and the program terminated. So that if we can determine the sign (positive or negative) of the offset between the base pointer and the result pointer, we can optimize by inserting only one side of the checking code. For instance, in line 12 of Listing 4, the offset variable `i` starts from zero, ensuring that it is always positive, eliminating the need to check for potential underflow pointers.

To implement this optimization, ShadowBound leverages LLVM's range analyzer [42], a powerful tool that provides insights into the possible value ranges of variables during program execution. By analyzing the sign of the offset, we can determine which boundary checks cannot be violated. For instance, if range analysis indicates a consistently positive offset, we optimize by focusing solely on the overflow check, thus eliminating the need for underflow verification. It is worth emphasizing that the use of range analysis also ensures that checked arithmetic instructions will not result in integer overflow (or wraparound), thereby maintaining the security assumption of ShadowBound. This approach is particularly beneficial in sections of code that are loop-intensive, as loops tend to undergo multiple iterations with consistent direction.

5.3 Security Pattern Identification

Many programs frequently employ simple code patterns to prevent out-of-bounds, making it unnecessary to validate code that has already been safeguarded by the original program. In this context, ShadowBound focuses on the identification of two primary patterns: Constant Array Argument and Length-Pointer Association.

Constant Array Argument The pattern is used to describe function arguments, where specific function arguments consistently accept constant-length arrays and the access offset for these arguments is always bound by the same constant. These arguments can be either stack arrays, global arrays, heap arrays, or constant-length arrays within a structure. If the arguments always belong to the first two types, we can directly remove the associated checks, as ShadowBound primarily focuses on heap security. To accomplish this, ShadowBound employs whole-program analysis to identify these function arguments and subsequently eliminates the redundant boundary checks associated with them.

The identification algorithm focuses on functions that are not the target of indirect call instructions. This is because if a function can be the target of an indirect call, it becomes challenging to analyze which properties the function's arguments consistently hold, as we may not be able to identify all call sites for this function. To determine if a function is the target of indirect call instructions, ShadowBound examines every instruction in the program and verifies that the function only appears as the called-function argument in call-related instructions. If the function is stored in memory or passed as a function argument, ShadowBound assumes it might become the target of an indirect call.

After identifying eligible functions, ShadowBound inspects the pointer-type arguments of each function. It traverses the entire program to identify all call sites associated with each function. ShadowBound evaluates each call site to determine whether the function's arguments consistently exhibit the characteristics of a constant length. To simplify the analysis, this assessment revolves around two key criteria: firstly, whether the pointer is instantiated within the same function where the call site is located, and secondly, whether the length of the pointer remains constant. If an argument is discovered to be loaded from memory or supplied as a function argument, ShadowBound refrains from engaging in further recursive analysis and instead assumes that it fails to meet the requisite criteria.

If an argument consistently meets the criteria of being a constant-length pointer after assessing all call sites, ShadowBound proceeds to the final step. In this concluding phase, ShadowBound meticulously examines the pointer arithmetic instructions related to the argument within the corresponding function. It leverages the LLVM range analyzer to confirm that the offsets generated by these instructions are strictly bound by the corresponding constant length. For those arguments that successfully pass all verifications, ShadowBound can confidently eliminate the associated checks.

Length-Pointer Association The Length-Pointer Association pattern is a common structural configuration in programs, characterized by two essential members: one representing a pointer and the other indicating the length of the data pointed to. These two members are referred to as the pointer member and the length member in the pattern. This pattern establishes a direct relationship between the pointer member and the corresponding length member. As demonstrated in Listing 4, through the initialization code at lines 7 to 8 for the structure `obj`, we can discern that the structure `obj` exemplifies this pattern, with `a` serving as a pointer to an integer array and `len` denoting the length of that array.

Identifying the Length-Pointer Association pattern is not always straightforward. For example, the modification for the pointer member and length member may not happen in the same scope, which may require heavy dataflow analysis techniques to address it. To simplify the identification process, we have introduced specific constraints that facilitate recognition. Firstly, both members must consistently undergo modifications within the same scope. Secondly, modifications to these two members should exclusively occur at the allocation or deallocation site. This typically means that the pointer is assigned as the return value of an allocation function or as a null pointer. Finally, the length field must precisely match the allocated memory's size at the allocation site. These constraints streamline the pattern identification process.

ShadowBound follows a systematic process to identify the Length-Pointer Association pattern. Initially, it identifies

```c
void foo(char *p) {
    char *a = p + 1;
    char *b = a + 2;
    char *c, *d;
    if (random() > 0.5)
        c = a + 3;
    else
        c = b + 4;

    for (d = c; d < p + 100; d++)
        *d = 'x';
}
```

Listing 5: Example of Merge Metadata Extraction

all allocation function call sites and checks if their return values are assigned to structure members. If this condition is met, ShadowBound further verifies if the length argument of the allocation function call is stored in another structure member. If both conditions are satisfied, ShadowBound assumes the potential presence of the pattern. Subsequently, ShadowBound scans the entire program to determine whether the two members of the structure fulfill all the pattern requirements that we mentioned before at all modification sites.

Upon confirming that these two members match the pattern, it means that the length member consistently indicates the length of the pointer member, ShadowBound proceeds to check all pointer arithmetic instructions that utilize the pointer member as the base pointer. If the offset between the base pointer and the result pointer is safeguarded by the length member, ShadowBound confidently removes the corresponding boundary checks. This meticulous process ensures precise optimization based on the Length-Pointer Association pattern.

5.4 Merge Metadata Extraction

Within its boundary-checking process, ShadowBound employs a two-stage approach, which involves an initial *extraction* stage to obtain the base pointer's boundary, followed by a subsequent *checking* stage to validate the result pointer. It's essential to highlight that the extraction stage is notably more time-consuming than the second stage due to the requirement of a load instruction for extracting metadata from the shadow memory. Importantly, the load address is exclusively determined by the base pointer. Consequently, in cases where multiple result pointers are computed from the same base pointer, it becomes possible to optimize by merging the extraction stage of these result pointers' boundary-checking processes, thereby avoiding redundant address loading.

To maximize the merging of extraction stages, ShadowBound employs a backtrace algorithm. Specifically, ShadowBound enumerates all pointer arithmetic instructions and attempts to backtrack their sources. We illustrate how the algorithm works using Listing 5. We start with the pointer `d`, which is initialized with `c` and then self-incremented, making `d` dependent on `c`. While `c` could be computed from either `a` or `b`, it is evident that both `a` and `b` are ultimately derived from `p`. Consequently, all pointers in the function `foo` are generated from `p`, establishing `p` as the source of pointers `a, b, c`, and `d`. Once all pointer sources are collected, for those pointers sharing the same source, we insert the boundary extraction code after the source. This eliminates the need to extract the boundary again in the subsequent checking.

5.5 Redundant Checking Elimination

In this subsection, we introduce two common techniques employed by ShadowBound to eliminate redundant checks. The first optimization typically occurs when both the allocation and the pointer arithmetic take place within the same scope, or when the base pointer is an array member of a structure. In such cases, the size of the base pointer is inherently determined by its allocation size or type. Consequently, the compiler can optimize this scenario by comparing the allocation size with the offset between the base pointer and the result pointer. If the offset is consistently smaller than the allocation size, ShadowBound can confidently assert that a pointer arithmetic will never result in an out-of-bounds condition. This empowers ShadowBound to eliminate the corresponding checks effectively.

The second technique entails conducting a thorough analysis of the relationships between pointer arithmetic instructions within a program. In particular, it commences by identifying cases where one pointer arithmetic instruction either dominates or post-dominates another instruction, provided that they share the same base pointer. In such instances, ShadowBound will proceed to compare the offsets of these two pointer arithmetic instructions. If it is consistently observed that the dominated computing instruction's offset is greater than that of the other instruction, then the underflow checking for this instruction can be safely eliminated. Conversely, if it is consistently found that the dominated computing instruction's offset is smaller than that of the other instruction, then the overflow checking for this instruction can be safely eliminated.

6 Implementation

In this section, we describe our implementation of the compiler and runtime support for ShadowBound, as well as our integration of state-of-the-art UAF defense mechanisms into the ShadowBound framework.

6.1 Compiler & Runtime Support

ShadowBound is build upon LLVM 15.0.6 framework [37] and consists of two components. The first, known as the function pass, is responsible for the insertion and optimization of checking instructions at the LLVM Intermediate Representation (IR) level. The second component is the runtime module, designed for shadow memory allocation and the management of associated metadata. These two components work seamlessly together, forming the foundational architecture of ShadowBound.

Function Pass. The function pass is implemented as an internal sanitizer within LLVM. Unlike other sanitizers, such as AddressSanitizer [51] or Memory Sanitizer [59], this tool is not designed to detect a specific class of bugs. Instead, it serves as a defense mechanism. Users can enable ShadowBound using the `-fsanitize=overflow-defense` flag. The function pass collects all pointer arithmetic instructions, including `getelementptr` and `bitcast`, and employs optimization algorithms that are mentioned in section 5 to determine which instructions require checking and how they should be checked.

Given that various optimization methods may interact with each other, an improper optimization sequence could potentially hinder both optimization performance and compilation speed. Therefore, we meticulously select the order in which each optimization method is applied, ensuring that our choices result in the best possible outcome. The function pass begins with Runtime-Driven Checking Elimination, as it effectively eliminates a significant number of checking instructions, reducing the burden for subsequent optimizations. Following that, the function pass executes Redundant Checking Elimination, Security Pattern Identification, and Directional Boundary Checking Optimization in sequence. Finally, the function pass concludes with the Merge Metadata Operation.

It is noteworthy that Security Pattern Identification necessitates a whole-program analysis. However, performing a whole program analysis requires access to information from all functions within the program, whereas the LLVM function pass is limited in its scope, and capable of retrieving information only from the currently processed function. To address this challenge, we have implemented an external analyzer similar to KINT [67]. This analyzer is capable of both dumping and analyzing the IR code of each function, subsequently saving the analysis results to a configuration file before compiling the program. The function pass then accesses and utilizes the information stored in this configuration file for its optimization, ensuring a more holistic approach to security pattern identification and mitigation.

Runtime Module. The runtime module of ShadowBound has two primary functions. First, it manages metadata, which includes the allocation of shadow memory. The size of the shadow memory in ShadowBound is equal to that of the original heap, mirroring the configuration of MemorySanitizer. Thus, our implementation closely resembles MemorySanitizer's shadow allocation method. Moreover, we introduce reserve operations at allocation sites, as detailed in Section 5. Second, the runtime module checks the arguments of frequently used libc functions, such as `memset` and `strcpy`, to guard against heap out-of-bounds incidents within these functions. To instrument the necessary checking code for these libc functions, we use a technique similar to AddressSanitizer. This method effectively identifies the functions needing validation and inspects their arguments for potential out-of-bounds access.

CVE/Issue ID	Link	Program	Prevention Type
CVE-2021-32281	[10]	gravity	✔ OOB Detected
CVE-2021-26259	[8]	htmldoc	✔ OOB Detected
CVE-2020-21595	[6]	libde265	✔ OOB Detected
CVE-2020-21598	[7]	libde265	✔ OOB Detected
CVE-2018-20330	[1]	libjpeg-turbo	✔ OOB Detected
CVE-2021-4214	[11]	libpng	✔ OOB Detected
CVE-2020-19131	[4]	libtiff	✔ OOB Detected
CVE-2020-19144	[5]	libtiff	✔ OOB Detected
CVE-2022-0891	[13]	libtiff	✔ OOB Detected
CVE-2022-0924	[14]	libtiff	✔ OOB Detected
CVE-2020-15888	[3]	Lua	✔ OOB Detected
CVE-2022-0080	[12]	mruby	✔ Benign Running
Issue-5551	[29]	mruby	✔ Transformation
CVE-2019-9021	[2]	php	✔ OOB Detected
CVE-2022-31627	[16]	php	✔ OOB Detected
CVE-2021-3156	[9]	sudo	✔ Benign Running
CVE-2022-28966	[15]	wasm3	✔ OOB Detected

Table 2: Heap out-of-bounds Prevention Results for ShadowBound on Real-World Vulnerabilities.

6.2 UAF Defense Integration

As discussed in subsection 4.3, ShadowBound theoretically can collaborate with three state-of-the-art UAF defense tools: MarkUs, FFMalloc, and PUMM. In this subsection, we delve into the implementation of each tool and present how we seamlessly integrate them within the ShadowBound framework, showcasing the integration achieved with minimal effort.

Both MarkUs and FFMalloc introduce new allocators. However, we encountered a challenge in which the heap region utilized by these allocators conflicted with our shadow memory region. Specifically, we designate two separate regions for the heap and shadow memory. The original heap memory of FFMalloc and Markus is not allocated within these regions. Therefore, we modify the code to adjust the heap region. It is important to note that we do not alter their allocation algorithm; we merely "shift" the allocated objects. Furthermore, we implemented code to facilitate the allocation and initialization of shadow memory after each allocation function. This adjustment ensured that these allocators gained the ability to allocate shadow memory for each heap chunk and initialize it.

The implementation of PUMM differs from that of MarkUs and FFMalloc. Instead of creating a new allocator, PUMM wraps the original allocator. This wrapping mechanism involves deferring specific deallocation operations based on a policy. PUMM generates this policy through binary-based analysis, which obviates the necessity for many modifications to enable integration. To integrate PUMM, our approach entails compiling the program with ShadowBound and then utilizing PUMM to analyze the binary and generate the policy. Subsequently, PUMM can wrap the program's allocator by the generated policy.

7 Evaluation

In this section, we address three key questions to assess the effectiveness and efficiency of ShadowBound:

1. Does ShadowBound effectively prevent heap out-of-bounds-based exploits?
2. What is the quantitative impact of ShadowBound on both runtime performance and memory utilization?
3. How do individual customized compiler optimization algorithms influence time overhead?

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.

7.1 Security Evaluation

Real-World OOB Prevention To validate ShadowBound's ability to prevent real-world heap out-of-bounds-based exploits, we conducted a comprehensive evaluation. This assessment included 34 real-world vulnerabilities; half of them were collected from previous works [75, 76] and Magma [28] dataset, while the others were selected from the CVE database [20, 21, 66] and program-specific issue reports. These vulnerabilities spanned a diverse set of 19 applications, encompassing servers, video encoders, language interpreters, widely adopted libraries, and UNIX utilities. For each case, we collected the corresponding Proof of Concept (PoC) that triggers program crashes. We observed how these programs, when compiled with ShadowBound, prevented the associated exploitation attempts.

In Table 2, we present the evaluation results on vulnerabilities selected by us. The results based on vulnerabilities collected from prior works are shown in Table 7 in the Appendix. ShadowBound successfully prevents all cases, demonstrating its capability to prevent real-world heap out-of-bounds-based exploitation. In each case, we observed the three distinct forms through which ShadowBound effectively prevented vulnerabilities: *OOB Detected*, *Benign Running*, and *Transformation*:

- **OOB Detected (OD)**: ShadowBound successfully detects heap out-of-bounds read or write instructions when an attempt is made to compromise a program used as a proof-of-concept (PoC). In simpler terms, ShadowBound acts as a vigilant guard, spotting and halting any unauthorized attempts to access memory beyond its boundaries.
- **Benign Running (BR)**: In certain scenarios, ShadowBound "fixes" the vulnerability by allocating additional memory space for each memory chunk. This extra space transforms what would otherwise be an unauthorized memory access into a legitimate operation, similar to creating an additional safety buffer. We confirmed this using a semi-automatic gdb script. The script fetched both the original requested size and the actual allocated size to determine the address range of the reserved space, and then checked if the out-of-bounds access fell within this range.
- **Transformation (TF)**: ShadowBound transforms heap out-of-bounds vulnerabilities into alternative, non-exploitable forms. During our manual verification process, we observed instances of null-pointer dereferences in these cases. These dereferences occurred because PoCs attempted to dereference pointers loaded from the reserved memory space created by ShadowBound for each memory chunk, and the reserved space is cleared to zero during allocation. Thus, the null-pointer dereference happened.

Program	#Input	#OD	#BR	#TF
cxxflit	1	1	0	0
libpcap	4	2	2	0
libxml2_reader	127	127	0	0
libxml2_xml	61	46	15	0
proj4	3	0	3	0
zstd	48	45	3	0
Total	244	221	23	0

Table 3: Heap out-of-bounds Prevention Results for ShadowBound on Synthesis Vulnerabilities.

Synthesis Vulnerability Prevention To demonstrate ShadowBound's robustness in security, we conducted an extensive evaluation using the RevBugBench dataset [77]. The dataset was originally designed for fuzzing testing and comprises a series of programs, each of which has been injected with over 1,000 distinct vulnerabilities collected from the real world. We employed AFL++ [26] to fuzz each program for 24 hours, generating numerous inputs capable of triggering program crashes. Subsequently, we assessed the prevention ability by using these inputs. It's essential to clarify that although the vulnerabilities were sourced from real-world cases, they might not appear at the same time in the real world. Therefore, these vulnerabilities triggered by inputs are synthetic but they are very close to real-world vulnerabilities.

Furthermore, recognizing that not all inputs result in heap out-of-bounds errors and certain inputs might execute the same program path, we implemented input refinement using the following methodology. Initially, we employed ASAN to discern inputs that cannot trigger heap out-of-bounds errors. For instance, inputs that solely lead to stack out-of-bounds or null-pointer dereference issues were automatically filtered out. Notably, every input will be considered unique if it has a distinct execution path leading to the final crash site. This is because even if another input results in the same crash site, their execution paths may diverge. Consequently, one of the redundant inputs was removed. The identification of triggered bug sets was facilitated by RevBugBench when provided with the program and inputs.

Following the meticulous cleanup process, we confirmed

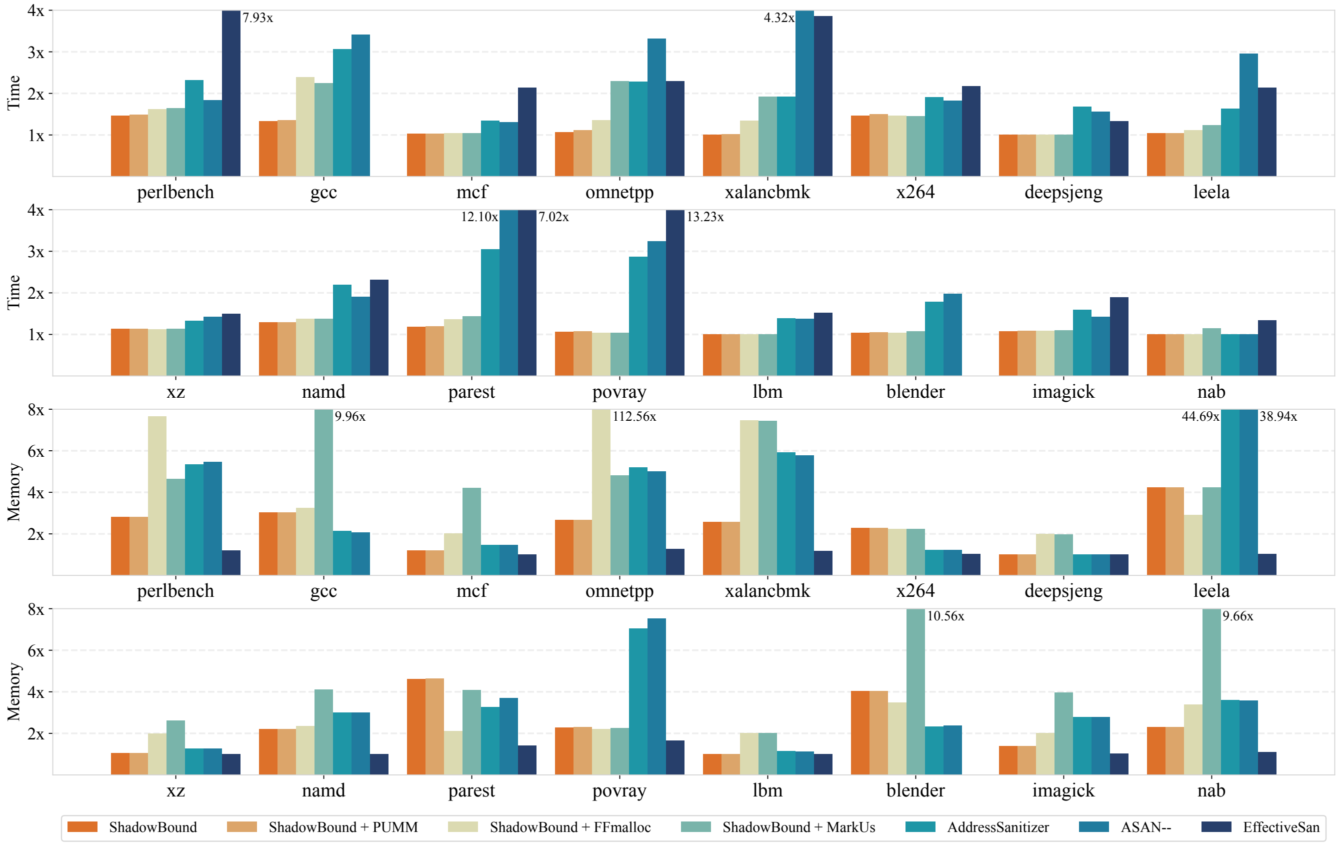

Figure 2: The time and memory overhead of SHADOWBOUND, SHADOWBOUND + PUMM, SHADOWBOUND + FFMalloc, SHADOWBOUND + MarkUs, AddressSanitizer, ASAN−−, ESAN on SPEC CPU2017. The upper two lines represent the results for time overhead, while the lower two outline the results for memory overhead. A value of 1x signifies no overhead. The geomean time overhead of each system is 5.72%, 6.60%, 9.95%, 16.20%, 62.03%, 79.85% and 138.76%. The geomean memory overhead of each system is 54.59%, 55.29%, 218.20%, 302.51%, 116.55%, 112.33%, 2.70%.

that all remaining inputs are indeed able to trigger heap out-of-bounds errors, with each input following a distinct trigger path. Subsequently, we compiled the benchmark programs with SHADOWBOUND and conducted testing using these inputs. The results, as detailed in Table 3, showcased SHADOWBOUND's effective prevention of these errors, with the majority being detected (#OD), while the remaining inputs allowed the program to execute without issues (#BR). All the benign running cases were also confirmed by the gdb script mentioned previously. Unlike our observations in real-world bug evaluations, we did not encounter any instances of *Transformation* (#TF), which we believe is normal given its dependency on the underlying program's logic.

7.2 Performance Comparison

SPEC CPU Benchmark To perform a comparative analysis between SHADOWBOUND and related tools that offer comprehensive heap memory protection, including both out-of-bounds (OOB) and Use-After-Free (UAF) defense, we integrated three state-of-the-art UAF defense tools: PUMM, FFMalloc, and MarkUs with SHADOWBOUND. We then conducted an extensive evaluation, including benchmarking against established tools such as ESAN, AddressSanitizer, MEMCHECK, and ASAN−−, using the SPEC CPU2017 suite [58]. To provide clarity regarding the overhead introduced by SHADOWBOUND itself, we also present results for SHADOWBOUND performance on SPEC CPU2017. Furthermore, we compared SHADOWBOUND with DeltaPointer, the state-of-the-art tool primarily focused on out-of-bounds detection. However, it's worth noting that DeltaPointer exclusively supports SPEC CPU2006 [57], and our attempts to migrate it to SPEC CPU2017 were unsuccessful. Consequently, our comparison between SHADOWBOUND and DeltaPointer is based on SPEC CPU2006. Regrettably, due to code compatibility issues, we had to exclude `471.omnetpp` and `401.bzip2` from the SPEC CPU2006 cases, as LLVM 15 failed to compile these benchmarks due to these codes are too old. To ensure fairness in our comparisons, we configured all tools to disregard detected errors, preventing premature termination. Additionally, we only enabled ASAN−−, ESAN, and AddressSanitizer's heap error detector. In our evaluation, we excluded PACMem and SGXBound due to their reliance on

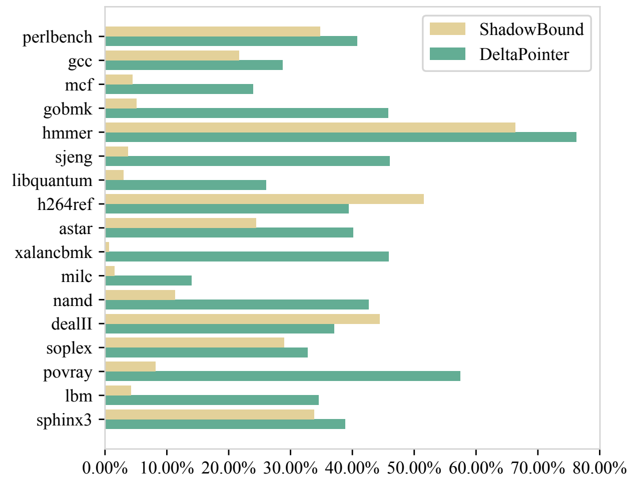

Figure 3: The runtime overhead of SHADOWBOUND and DeltaPointer on SPEC CPU2006, the geometric mean overhead of each system is 10.58% and 37.08%.

specialized hardware for heap memory error detection, as well as the unavailability of their source code. For each tool, we executed the benchmarks ten times and reported the average results to mitigate the impact of randomness.

Figure 2 presents the results obtained from evaluating SPEC CPU2017 benchmarks. Notably, ESAN encountered issues and failed to execute the `gcc` and `blender` benchmarks. As for MEMCHECK, it exhibited a significant performance slowdown of over 4x in every test case, its geomean time and memory overhead is 2426.66% and 92.95%, rendering its results unfit for inclusion in the figure. Furthermore, it is worth highlighting that both ASAN−− and ESAN's results were inferior to AddressSanitizer. This discrepancy is attributed to the fact that AddressSanitizer in LLVM 15 deployed more advanced optimizations, whereas ASAN−− and ESAN are based on older LLVM versions. The time overhead of SHADOWBOUND itself is 5.72%, which demonstrates the efficiency of SHADOWBOUND. All three variants of SHADOWBOUND also performed admirably. In particular, SHADOWBOUND + PUMM demonstrated the lowest average overhead (6.60%), surpassing all tools that provide comprehensive heap memory protection. The memory overhead of SHADOWBOUND itself averaged at 54.59%, and SHADOWBOUND + PUMM also exhibited the best memory overhead among all variants, better than ASAN and ASAN−−. While it is slightly higher than that of ESAN, we believe it is acceptable given the notable performance improvements it offers.

To enhance our understanding of the scalability and performance implications of SHADOWBOUND, we measured several parameters related to heap usage. These include the number of instrumentation points, the number of allocation and deallocation operations per second, the amount of shadow memory allocated for each program, and both heap allocation and access frequencies (see Figure 6 in the Appendix). These measurements were obtained by inserting counting instructions following each load and store operation, as well as allocation and deallocation functions. We conducted linear regression to determine how significantly these parameters affect the performance overhead. We found that performance is most closely related to the frequency of heap access. The regression coefficients for other parameters are close to zero, indicating that they have little or no real relationship with time overhead. Based on the frequency of heap accesses, we used 1GHz, 2GHz, and 3GHz as thresholds to categorize the programs into four groups: *lightweight*, *normal*, *heavy*, and *intensive* heap access. We found the geometric mean time overhead of each group is 2.35%, 3.58%, 6.52%, and 29.66%, respectively. This tiered classification also demonstrates a direct correlation between the intensity of heap access and the performance overhead. Based on the evaluation result, it becomes evident that its adoption is particularly advantageous for lightweight, normal and heavy heap access scenarios, where the method's low to moderate performance overhead can enhance security and monitoring without significantly impacting functionality. On the other hand, for data-intensive and high-performance computing applications, which fall into the intensive heap access categories, the method's higher performance overhead necessitates a more cautious approach.

Figure 3 presents the time overhead of SHADOWBOUND and DeltaPointer on SPEC2006. SHADOWBOUND exhibits lower overhead in nearly all test cases, with the exceptions being `h264ref` and `dealII`. The geometric mean of the time overhead for SHADOWBOUND is also significantly better than that of DeltaPointer. In terms of memory overhead, DeltaPointer achieves an average memory overhead of zero, while SHADOWBOUND incurs an average memory overhead of 68.21%, which is higher than DeltaPointer. It's worth mentioning that while DeltaPointer may have lower memory overhead, it restricts available address space to 4GB, limiting its scalability for large-scale programs. Additionally, DeltaPointer does not provide defense against underflow errors, whereas SHADOWBOUND offers improved security in this regard.

Regarding the variation in time overhead observed in SPEC CPU2006 and SPEC CPU2017, which stand at 10.58% and 5.72% respectively, we have identified the cause. This discrepancy arises from certain test cases within SPEC2006, such as `hmmer`, `dealII`, and `h264ref`, which contain functions exhibiting distinctive patterns that are challenging for SHADOWBOUND to optimize effectively. To exacerbate matters, the test input often leads to these functions consuming over 50% of the program's execution time, amplifying the impact of the instrumentation. However, we believe that these issues are more prevalent in smaller-scale programs. Our subsequent experiments demonstrate that larger-scale programs are less likely to encounter such problems.

NGINX To assess SHADOWBOUND's performance in a real-world, large-scale application, we conducted experiments using Nginx v1.22.1 and the wrk v4.2.0 benchmarking tool. These experiments utilized a configuration of 8 threads, 100

System	Output	Latency (μs)				
	(req/s)	Average	50%	75%	90%	99%
NATIVE	158,847	611	592	604	623	748
ShadowBound	147,550	650	640	649	668	767
SB + MarkUs	124,361	777	759	770	803	890
SB + FFMalloc	110,406	870	860	880	900	1000
SB + PUMM	79,229	1220	1200	1220	1270	1460

Table 4: Evaluation Results of Native, ShadowBound and its variants: Output and Latency Analysis on Nginx. In the Latency column, Average denotes the average latency of the requested connections, while the remaining values depict latency distribution.

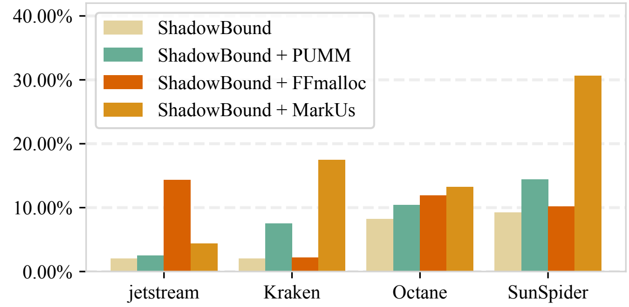

Figure 4: Runtime overhead comparison of ShadowBound and its variants on the Chakra engine: The geometric mean overhead for each system is 4.17%, 7.28%, 7.86%, 13.28%.

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 4. ShadowBound experiences 6.47% and 2.54% time overhead for the average and tail (99%) latencies, respectively; these results illustrate ShadowBound's efficiency and practicality in real-world applications. Among variants of ShadowBound, ShadowBound + MarkUs delivers the best performance, experiencing 27.23% and 18.98% time overhead for the average and tail latencies, respectively.

Chakra In addition to Nginx, we also evaluated the performance of ShadowBound, as well as ShadowBound + PUMM, ShadowBound + FFMalloc and ShadowBound + MarkUs on the Chakra. Chakra is a high-performance JavaScript engine developed by Microsoft. Our assessment encompassed four key benchmarks integrated into Chakra: Octane, Kraken, SunSpider, and JetStream. As depicted in Figure 4, our evaluation demonstrates that ShadowBound consistently delivers excellent performance on the Chakra platform. The results indicate that ShadowBound introduces minimal overhead, showcasing its robust compatibility with real-world applications. Particularly noteworthy is the performance of ShadowBound + PUMM and ShadowBound + FFMalloc, which demonstrates exceptional efficiency. These findings underscore ShadowBound's capability to deliver acceptable overheads on large-scale applications.

Website	Native	ShadowBound	Overhead
www.google.com	1202	1237	2.93%
www.facebook.com	932	950	2.01%
www.amazon.com	2399	2444	1.87%
www.openai.com	1544	1577	2.16%
www.twitter.com	1580	1634	3.45%
www.gmail.com	1791	1822	1.75%
www.youtube.com	2244	2374	5.79%
www.wikipedia.org	1085	1133	4.42%
www.netflix.com	1415	1448	2.36%
Geomean	-	-	2.74%

Benchmark	Octane	Kraken	SunSpider	Geomean
ShadowBound	3.60%	3.30%	5.50%	4.03%

Table 5: Runtime overhead on Chromium: website loading times and JavaScript benchmarks.

Chromium To assess the performance and compatibility of ShadowBound, we conducted an evaluation using the Chromium browser. Chromium is one of the largest and most widely used software, making it an ideal candidate for showcasing ShadowBound's capabilities. We employed three JavaScript benchmarks, Octane, Kraken, and SunSpider, to test ShadowBound's performance. Additionally, we measured the loading times of the 9 most popular websites according to the Top Websites Ranking [63], a critical metric for enhancing the browsing experience. To ensure accurate loading time measurements, we recorded the average loading time for the nine most popular websites.

It is important to note that during our experiments, we encountered difficulties when attempting to dynamically load the allocators for FFMalloc and MarkUs into Chromium; Chromium immediately raised a segmentation fault upon running the program. According to the stack trace, FFMalloc aborted at the re-allocation function, and MarkUs aborted when creating the new thread. Additionally, PUMM requires fuzzing the program and analyzing the trace files. However, PUMM's analysis algorithm proved to be very slow and memory-consuming in our experiments. We attempted analysis for over 24 hours, yet PUMM still failed to produce results and ran out of memory. Consequently, we are presenting results exclusively for the ShadowBound. Our evaluation includes three benchmarks integrated into Chromium and assesses the access speed of nine popular websites. For each benchmark experiment, we conducted 30 repetitions and calculated the geometric mean values to mitigate the impact of random variations. The results are presented in Table 5.Nearly all website loading time overheads remained below 5%, demonstrating that ShadowBound has minimal influence on the browsing experience for users.

7.3 Ablation Study

In our evaluation of ShadowBound's performance on the SPEC CPU2017 suite, we demonstrate the system's perfor-

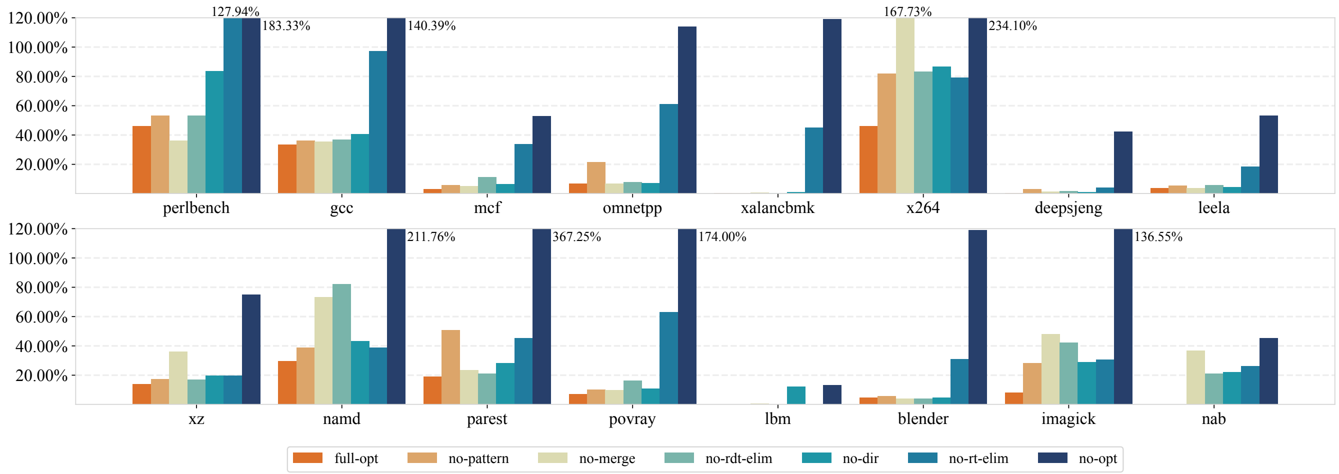

Figure 5: Ablation study result of ShadowBound on SPEC CPU2017. From left to right, the bars show the time overhead of ShadowBound with full optimization, ShadowBound with each optimization disabled, and ShadowBound without optimization. The geomean value of each setting is 5.72%, 9.51%, 11.56%, 11.76% 12.86%, 29.28% and 99.69%.

mance with full optimization enabled. These optimizations include Runtime-Driven Checking Elimination (rt-elim), Directional Boundary Checking (dir), Security Pattern Identification (pattern), Merge Metadata Operation (merge), and Redundant Checking Elimination (rdt-elim). To isolate the effects of each optimization, we conducted an ablation study consisting of seven experiments. Five of these experiments involved disabling one of these optimizations, while two additional experiments enabled or disabled all optimizations together. These experiments provided valuable insights into the system's complete capabilities and its original performance, allowing us to precisely measure the contributions of each optimization to system performance.

The results presented in Figure 5 highlight the significant performance improvements achieved with ShadowBound's optimizations, compared to scenarios without them. The optimization reduces overhead from 99.69% to 5.72%. Among these optimizations, Runtime-Driven Checking Elimination stands out; without it, the overhead increases to 29.28%. Disabling other optimizations, specifically Directional Boundary Checking Optimization, Merge Metadata Extraction, Redundant Checking Elimination, and Security Pattern Identification, results in overheads of 12.86%, 11.76%, 11.56%, and 9.51%, respectively. While the effectiveness of these optimizations may vary across different test cases, each one is essential in the majority of them. These results emphasize the importance of these optimizations in enhancing ShadowBound's performance. Furthermore, the results indicate that moving the checking position from pointer dereference to pointer arithmetic itself does not reduce overhead. As shown in subsection 7.2, the overhead for ASAN is 62.03%, and the overhead for unoptimized ShadowBound is 99.69%. The primary difference between them lies in the checking position. Therefore, performing checks at the point of pointer arithmetic does not lead to a reduction in overhead.

8 Discussion

Intra-Object Out-Of-Bounds As ShadowBound checks at the granularity of heap chunks or objects, which is determined at the time of allocation, it does not provide support for preventing out-of-bounds errors that occur within the same object. For instance, it cannot prevent an out-of-bounds access from one array field to another field within the same structure. Consequently, ShadowBound shares the same weakness as prior works [24, 32, 33, 36, 39, 51]. The boundary within an object's inner fields is not determined by the allocated size but rather by the object's type. If desired, ShadowBound has the capability to relocate shadow memory initialization from the allocation site to the type casting site. We will consider this as a potential area for future work.

Pointer Casting Given that ShadowBound relies on pointers for boundary checking, it's important to note that developers can write code containing implicit type casts and pointer-integer conversions in low-level languages like C/C++. For implicit type casts, although C/C++ allows this practice, a properly configured LLVM always generates an explicit type casting instruction [49]. Therefore, ShadowBound can effectively handle implicit type casts. With regard to the conversion between pointers and integers, there is potential for out-of-bounds issues during integer computation instructions. This vulnerability is also noted in several prior works [24, 36, 45, 73]. However, it's crucial to emphasize that developers typically have benign intentions, and some established C/C++ standards [44, 64] discourage conversions between integers and pointers unless the developers have a comprehensive understanding of the potential consequences. Consequently, occurrences of such errors are exceptionally rare. In fact, during our extensive evaluation, we did not encounter any such cases. Nevertheless, we acknowledge this as a potential source of false negatives.

Stack Protection While the design of SHADOWBOUND does not provide protection against stack out-of-bounds vulnerabilities, it is indeed feasible to enhance SHADOWBOUND's capabilities in this regard. LLVM IR employs explicit instructions for allocating objects on the stack. Consequently, SHADOWBOUND can establish the bounds of stack objects by storing this information in the shadow memory at the time of stack allocation. Alternatively, SHADOWBOUND can be integrated with mature solutions such as shadow memory-based solutions [19, 35], hardware-based solutions (e.g., ARM PA [61]), or hybrid solutions (e.g., Intel CET [52]). Even though these solutions also use shadow memory, SHADOWBOUND was designed with the possibility of integration in mind, so the shadow memory region of the stack is reserved. We believe that these methods are sufficient to protect the stack from attacks such as Return-Oriented Programming (ROP) while providing minimal overhead.

Dynamic Library SHADOWBOUND is capable of safeguarding a dynamic library, provided it has been instrumented by our tool. Should a dynamic library not undergo instrumentation, SHADOWBOUND retains the ability to detect out-of-bounds errors occurring externally. For instance, if a pointer originating within the library triggers an overflow outside of it, SHADOWBOUND can identify such errors. This is due to SHADOWBOUND's comprehensive monitoring of all memory management APIs, enabling precise tracking of every object's boundaries.

Future Optimization SHADOWBOUND still possesses optimization potential in both compiler optimization and metadata management aspects. Regarding compiler optimization, SHADOWBOUND does not fully harness the capabilities of Link-Time Optimization (LTO) [41] and Profile Guided Optimization (PGO) [50]. We believe that boundary checking can be further optimized by leveraging additional interprocedural analysis techniques and profiling feedback data. In terms of metadata management, given that SHADOWBOUND is a pure software solution, metadata extraction can be further enhanced by incorporating certain hardware features to maintain the metadata, such as Intel MPX [48] and CHERI [69].

9 Related Work

Checking Optimization Compiler optimization techniques are commonly employed to optimize out-of-bounds checking. Many previous works [24, 34, 36, 39] have leveraged general optimization techniques to enhance checking performance or eliminate redundant checks. Additionally, there are some works [75, 76] that have designed custom optimization techniques to optimize Address Sanitizer's [51] checking. These optimization techniques can reduce a certain amount of overhead but cannot have an order of magnitude impact. The optimization technology of SHADOWBOUND has significantly reduced overhead and has made SHADOWBOUND practical.

Boundary Annotation LLVM introduced a new feature called `-fbounds-safety` [25]. This feature allows programmers to manually designate length variables for pointers, thereby enabling the compiler to insert boundary checking instructions for these pointers. While a direct performance comparison between SHADOWBOUND and `-fbounds-safety` is not feasible due to the latter's reliance on human annotations, combining `-fbounds-safety` with SHADOWBOUND could be beneficial. This is because neither is consistently superior to the other in all scenarios. For pointers with lengths that are already annotated, the `-fbounds-safety` flag may not require extra instructions to fetch the array length, potentially resulting in better performance. However, `-fbounds-safety` cannot address out-of-bounds errors caused by type downcasting, while SHADOWBOUND can, as it instruments pointer casting instructions. On the other hand, annotating all pointer length variables requires significant human effort and may not always be feasible. In contrast, SHADOWBOUND can manage these cases automatically without the need for human annotation.

Pointer Tagging Pointer Tagging is a common technique for storing the boundary in out-of-bounds detection or defense tools. Some studies [33, 34] store complete boundary information in the high-order bits of the pointer, eliminating the need to retrieve boundary information from memory. However, the available high-order bits are not sufficient to contain all the necessary boundary information. These approaches reduce available memory space, which may lead to potential compatibility issues with large-scale programs. To encode more information in the pointers without shrinking the memory space, other studies [24, 27, 36] have implemented custom allocators that allow the original pointer to convey additional boundary information. PACMem [39] uses ARM PA to generate a key stored in the high-order bits. This key is then used as an index to store the actual boundary information in shadow memory. The complexity of these encoding algorithm usually introduce additional arithmetic and memory operations that could potentially impact performance.

10 Conclusion

This work introduces SHADOWBOUND, an innovative heap memory protection design. SHADOWBOUND utilizes advanced metadata management and customized compiler optimization to provide a robust heap out-of-bound defense. We integrated it with three state-of-the-art use-after-free defenses without compatibility issues. Ensuring both temporal and spatial memory protection, SHADOWBOUND maintains a minimal performance and memory impact. Experimental results demonstrate its effectiveness and efficiency in defending against heap memory vulnerabilities, especially in programs written in unsafe languages.

Acknowledgement

We thank the anonymous reviewers for their valuable suggestions and feedback. This work was partially supported by Northwestern University fellowship, NSF grant CNS-2221122. This paper gives the views of the author, and not necessarily the position of the funding agency.

References

[1] CVE-2018-20330. Available from NVD, CVE-ID CVE-2018-20330 https://nvd.nist.gov/vuln/detail/CVE-2018-20330, 2018.

[2] CVE-2019-9021. Available from MITRE, CVE-ID CVE-2019-9021 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9021, 2019.

[3] CVE-2020-15888. Available from MITRE, CVE-ID CVE-2020-15888 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15888, 2020.

[4] CVE-2020-19131. Available from NVD, CVE-ID CVE-2020-19131 https://nvd.nist.gov/vuln/detail/CVE-2020-19131, 2020.

[5] CVE-2020-19144. Available from MITRE, CVE-ID CVE-2020-19144 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-19144, 2020.

[6] CVE-2020-21595. Available from MITRE, CVE-ID CVE-2020-21595 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-21595, 2020.

[7] CVE-2020-21598. Available from NVD, CVE-ID CVE-2020-21598 https://nvd.nist.gov/vuln/detail/CVE-2020-21598, 2020.

[8] CVE-2021-26259. Available from MITRE, CVE-ID CVE-2021-26259 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26259, 2021.

[9] CVE-2021-3156. Available from NVD, CVE-ID CVE-2021-3156 https://nvd.nist.gov/vuln/detail/CVE-2021-3156, 2021.

[10] CVE-2021-32281. Available from MITRE, CVE-ID CVE-2021-32281 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32281, 2021.

[11] CVE-2021-4214. Available from NVD, CVE-ID CVE-2021-4214 https://nvd.nist.gov/vuln/detail/CVE-2021-4214, 2021.

[12] CVE-2022-0080. Available from MITRE, CVE-ID CVE-2022-0080 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0080, 2022.

[13] CVE-2022-0891. Available from NVD, CVE-ID CVE-2022-0891 https://nvd.nist.gov/vuln/detail/CVE-2022-0891, 2022.

[14] CVE-2022-0924. Available from NVD, CVE-ID CVE-2022-0924 https://nvd.nist.gov/vuln/detail/CVE-2022-0924, 2022.

[15] CVE-2022-28966. Available from MITRE, CVE-ID CVE-2022-28966 https://nvd.nist.gov/vuln/detail/CVE-2022-28966, 2022.

[16] CVE-2022-31627. Available from NVD, CVE-ID CVE-2022-31627 https://nvd.nist.gov/vuln/detail/CVE-2022-31627, 2022.

[17] Sam Ainsworth and Timothy M. Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[18] Jason Evans April. A scalable concurrent malloc(3) implementation for freebsd. In *Proceedings of the BSDcan Conference*, 2006.

[19] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

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

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

[22] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical Page-Permissions-Based scheme for thwarting dangling pointers. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[23] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[24] 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*, 2018.

[25] Enforcing bounds safety for production C code. https://llvm.org/devmtg/2023-05/slides/TechnicalTalks-May11/01-Na-fbounds-safety.pdf.

[26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[27] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. 2020.

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

[30] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[31] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, 2021.

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

[33] 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*, 2018.

[34] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[35] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[36] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[37] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *Proceedings of International Symposium on Code Generation and Optimization*, 2004.

[38] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of The Network and Distributed System Security Symposium*, 2015.

[39] 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.

[40] Daiping Liu, Mingwei Zhang, and Haining Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[41] LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html.

[42] LLVM ScalarEvolution Class Reference. https://llvm.org/doxygen/classllvm_1_1ScalarEvolution.html.

[43] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[44] MISRA C. https://misra.org.uk/.

[45] 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.

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

[47] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[48] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: An empirical study of intel mpx and software-based bounds checking approaches. 2017.

[49] Opaque Pointers. https://llvm.org/docs/OpaquePointers.html.

[50] Sergey Yakushkin Pavel Kosov. LLVM PGO Instrumentation: Example of CallSite-Aware Profiling. https://llvm.org/devmtg/2020-09/slides/PGO_Instrumentation.pdf.

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

[52] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[53] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[54] 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.

[55] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. Guarder: A tunable secure allocator. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[56] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, 2019.

[57] SPEC2006. https://www.spec.org/cpu2006/.

[58] SPEC2017. https://www.spec.org/cpu2017/.

[59] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *Proceedings of IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[60] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[61] Inc. Qualcomm Technologies. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[62] The GNU Allocator. https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html.

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

[64] Software Engineering Institute Carnegie Mellon University. Sei cert c coding standard rules for developing safe, reliable, and secure systems, 2016.

[65] 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.

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

[67] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[68] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. *maze*: Towards automated heap feng shui. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[69] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[70] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free attacks with fast forward allocation. In *Proceedings of 30th USENIX Security Symposium*, 2021.

[71] Carter Yagemann, Simon Pak Ho Chung, Brendan Saltaformaggio, and Wenke Lee. Pumm: Preventing use-after-free using execution unit partitioning. In *Proceedings of 32nd USENIX Security Symposium*, 2023.

[72] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of The Network and Distributed System Security Symposium*, 2015.

[73] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the 2010 ACM SIGSAC Conference on Computer and Communications Security*, 2010.

[74] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[75] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. Sanrazor: Reducing redundant sanitizer checks in c/c++ programs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[76] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[77] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. FIXREVERTER: A realistic bug injection methodology for benchmarking fuzz testing. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

A Statistic of SPEC CPU2017 Benchmark

Program	Instrument Count		Operation Per Second		Shadow Memory	Alloc Freq	Access Freq
	#extraction	**#checking**	**alloc**	**dealloc**	(KB)	(MB/s)	(GHz)
deepsjeng	9	17	0.02	0.01	121160	3.59	0.06
leela	113	132	176,400.76	176,400.73	58484	223.81	0.20
omnetpp	2590	3015	1,698,303.59	1,698,266.30	326756	214.96	0.96
nab	780	1214	1,414.20	1,077.32	176892	7.03	1.12
mcf	49	99	2,224.47	2,224.47	80364	8.58	1.23
blender	15479	22445	70,776.19	70,776.14	1297236	55.24	1.34
gcc	13571	17516	816,831.53	794,791.98	10995204	2,935.12	1.39
xz	279	631	0.45	0.37	78908	13.41	1.50
lbm	10	244	0.05	0.05	110196	3.17	1.85
perlbench	6377	7909	597,526.88	592,088.88	657444	53.42	2.11
povray	2111	2672	547.42	547.32	51372	0.09	2.45
imagick	4237	5374	123,346.30	123,346.10	61668	16.00	2.51
xalancbmk	10449	12589	968,286.71	968,286.70	647992	459.19	2.96
namd	1274	4088	127.59	126.66	199664	2.69	3.26
x264	4759	6677	20.68	20.67	456384	1.07	3.26
parest	33368	39771	631,792.47	631,792.30	1091476	697.83	5.34
Regr. Coef.	3.88×10^{-4}	3.43×10^{-4}	2.68×10^{-6}	2.53×10^{-6}	2.10×10^{-6}	6.42×10^{-3}	4.99

Table 6: **Statistic of SPEC CPU2017 Benchmark**. The table is sorted by **Access Freq**, heap access frequency, and it is divided into four groups, categorizing programs as lightweight, normal, heavy, and intensive heap access from top to bottom, respectively. The **Instrument** part indicates the number of instrumentation points inserted by the system. **Operation Per Second** section indicates the number of allocation and deallocation operations per second. **Shadow Memory** displays the amount of shadow memory allocated for each program. **Alloc Freq** represents the rate of memory allocation per second. **Regr. Coef.**, the linear regression coefficient, indicates the correlation between these indices and performance overhead. The coefficient is close to zero suggesting there is minimal or no relationship between them.

B Realworld Security Evaluation

Source	CVE/Issue ID	Program	Result
SanRazor	CVE-2015-9101	lame	✔OD
	CVE-2016-10270	libtiff	✔BR
	CVE-2016-10271	libtiff	✔OD
	CVE-2017-7263	potrace	✔OD
	2017-9167-9173	autotrace	✔OD
	2017-9164-9166	autotrace	✔OD
Asan−−	CVE-2006-6563	proftpd	✔OD
	CVE-2009-2285	libtiff	✔OD
	CVE-2013-4243	libtiff	✔OD
	CVE-2014-1912	python	✔OD
	CVE-2015-8668	libtiff	✔OD
Magma	CVE-2016-1762	libxml	✔BR
	CVE-2016-1838	libxml	✔BR
	CVE-2019-10872	poppler	✔OD
	CVE-2019-9200	poppler	✔OD
	CVE-2019-7310	poppler	✔OD
	CVE-2013-7443	sqlite	✔OD

Table 7: Security evaluation for ShadowBound on vulnabilities from prior works.