Thesis Dissertation

MEMORY SAFETY: ADDING TEMPORAL SAFETY BLOCKS IN C

Andreas Hadjoullis

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Memory Safety: Adding Temporal Safety Blocks in C

Andreas Hadjoullis

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfillment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2025

Acknowledgments

I would like to express my gratitude to Dr. Elias Athanasopoulos for his guidance and insightful feedback throughout the course of this project. His ideas were foundational to the development and direction of the work and instrumental in shaping the final outcome.

I am also sincerely thankful to my friends for their support and encouragement during my studies.

Summary

Despite the rise of memory-safe languages, C remains indispensable, and with it, persistent heap-based use-after-free (UAF) vulnerabilities. These bugs exploit dangling pointers: when an object is deallocated but still referenced, attackers can hijack freed memory. Process-wide defenses incur prohibitive runtime overhead. Thus, we introduce safe blocks, an abstraction that lets developers annotate security-critical regions for focused temporal-safety enforcement, avoiding global performance penalties.

Inside each safe block, allocations are routed to an isolated safe heap, whose permissions are enforced via Intel's Memory Protection Keys (MPK), preventing any access outside annotated regions. A mark-and-sweep garbage collector (GC) reclaims objects only after they have been explicitly freed and proven unreachable, eliminating dangling pointers within safe blocks. Outside these regions, our system defers to the standard allocator with zero GC overhead.

Our prototype, implemented as an LD_PRELOAD library using mimalloc, requires minimal changes to existing code. Through empirical evaluation on real-world benchmarks, we demonstrate the feasibility and characterize the performance trade-offs of safe blocks, paving the way for targeted optimizations and practical adoption in legacy C systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Memory-unsafe languages such as C and C++ are widely-used [59], forming the backbone of critical software infrastructure, including web browsers, kernels and network applications. There has been a great cause for concern and significant research regarding the security of these applications [58]. More specifically, there has been an emphasis on *use-after-free* (UAF) exploits [49]. UAF is a kind temporal memory corruption bug of very high importance. It has been consistently ranked by the MITRE organization in the last six years as one of the top **ten** most dangerous software errors [43,45–48,50].

When a UAF exploit is executed successfully, it can allow an attacker to alter the control flow of the target process. Several real-world examples highlight the severity of such vulnerabilities, such as the following. One critical bug in Firefox's CSS animation timeline was exploited in the wild to escape the browser sandbox [21]. A race condition in Google Chrome enabled unsandboxed code execution through seemingly legitimate JavaScript API calls [12]. In the Linux kernel, a UAF vulnerability was leveraged to escalate privileges [64]. A high-severity bug was also discovered in the JSON parser of Google's protobuf library [24]. Additionally, Adobe Reader suffered from a flaw in processing format event actions, which could be exploited to execute arbitrary code [44].

The proposed solution to guarantee temporal safety, is the usage of memory safe languages such as Rust. In recent years, Rust has been gaining a lot of traction [52,63] for its safety [5,11], but also for having speed comparable to that of C [15,65]. However, simply rewriting all C code to Rust is unfeasible, which is why translating C to Rust is being explored as an option [19,35]. Unfortunately, this has not yet reached a mature level. We are in need of a more immediate solution.

Dealing with use-after-free bugs is notoriously difficult due to their temporal nature. They certainly will happen, and detecting them is tough. Especially since a significant portion of them is non-deterministic [13]. Static detection methods fail to capture the most complex bugs [13], while exploit defenses, despite being able to achieve temporal safety, always introduce a large overhead [58]. More recent techniques [1, 20] started using ideas derived from older concepts such as *Garbage-Collection*, a form of automatic memory management in which temporal safety is guaranteed.

We argue that applying defenses to the whole binary is excessive, that instead we need to safeguard only sections of it. Specifically, sections that deal with user input, since those are the most critical ones in term of security [14]. We consider a new approach, suggesting the usage of *safe* blocks, where in such blocks we guarantee complete temporal safety.

```
void foo(char *user_input) {
    bar();
    safe {
        char *retval = parse(user_input);
        process(retval);
    }
    baz();
}
```

Listing 1.1: Safe block showcase

Listing 1.1, presents a proof-of-concept illustrating our proposed scheme. In this example, the functions `parse` and `process` are executed within the *safe* block, ensuring that no use-after-free bug can occur during their execution. This safety is achieved by creating a dedicated *safe* heap for safe blocks and prohibiting access to it from outside these blocks by utilizing *Intel's Memory Protection Keys (MPK)* [17]. To then guarantee temporal safety, we apply Garbage-Collection techniques only within the specified *safe* blocks.

Following prior work on use-after-free defenses [1, 13, 20], our focus is limited to heap-based use-after-free vulnerabilities. These are not only the most prevalent in real-world exploits, but also the most challenging to defend against [13].

1.2 Contributions

- Safe-Block Abstraction: We introduce and formalize the concept of "safe blocks", allowing developers to mark critical code regions for targeted temporal safety enforcement.

- Prototype Implementation: We design and implement a runtime system that leverages Intel's Memory Protection Keys and scoped garbage-collection to isolate and protect marked blocks from use-after-free errors.

- Practicality and Usability: We test the integration of our prototype with existing projects to assess how feasible and straightforward it is to adopt our approach.
- Evaluation and Analysis: We empirically evaluate the performance overhead of safe blocks across real-world C applications.

All source code and documentation for this project are publicly available here.

1.3 Thesis structure

In the subsequent chapters of this thesis, we present the necessary background concepts, and the methodology underlying this work. Chapter 2 introduces foundational terminology and examines the nature of use-after-free vulnerabilities within memory-unsafe systems. Additionally, this chapter provides an overview of Intel's Memory Protection Keys mechanism and describes the essential garbage collection techniques employed in our approach. Readers already familiar with these topics may choose to omit this chapter. Chapter 3 introduces the design and theoretical foundations of safe blocks, the central abstraction proposed in this thesis. Chapters 4 and 5 form the technical core of the work, detailing the implementation of safe blocks and how they can be used by a developer. Chapter 5 presents an empirical evaluation of the proposed solution, assessing its security guarantees and performance overhead, as well as our attempts to integrate our system with large and popular C projects. Following this, Chapter 6 discusses the limitations of the current prototype, proposes avenues for improvement, and outlines directions for future research toward broader and more practical enforcement of temporal memory safety. The thesis concludes with Chapter 7, which positions our work within the existing literature, and Chapter 8, which summarizes our findings and contributions.

Chapter 2

Background

2.1 Memory Safety

Memory safety refers to the property of a program that ensures all memory accesses are well-defined, meaning that reads and writes occur only within the boundaries of valid, allocated memory regions. Violations of memory safety are the root cause of a wide range of critical software bugs and security vulnerabilities [58]. These violations can be broadly categorized into two groups: spatial and temporal errors.

- *Spatial safety*: ensures that memory accesses remain within the bounds of the object being accessed. Violations include buffer overflows, out-of-bounds reads, and writes, where a pointer accesses memory beyond its allocated extent.

- *Temporal safety*: guarantees that pointers only access memory that is still *valid*, that is, memory that has neither been deallocated nor reused. Violations include use-after-free, use-after-return, and double-free bugs.

Both C and C++ are memory-unsafe, meaning they do not provide spatial or temporal safety. As explained before, temporal safety is far more difficult to ensure without causing significant overhead [13, 58] and has increasingly become more severe [20, 34].

2.1.1 Use-After-Free

A use-after-free (UAF) occurs when a program continues to access memory through a pointer after that memory has already been deallocated (freed). This typically arises in heap-allocated data structures, although this is also possible with stack-allocated data as well. UAF bugs violate temporal memory safety because the pointer in use no longer refers to a valid object. Even though the memory might still be mapped and accessible, its ownership has been relinquished, and future reuse by the program or attacker-controlled inputs, can lead to corrupted state or control flow hijacking.

A use-after-free exploit occurs when:

1. Memory is allocated.
2. Later that memory is freed while still having a *dangling* pointer to it, meaning a pointer that no longer points to valid memory.
3. An attacker is now able to **legally** allocate data in said free region.
4. The program attempts to incorrectly reuse the initial references, that are now pointing to attacker controlled data.

```c
#include <stdio.h>
#include <stdlib.h>

typedef void (*fp)(); // define function pointer type
void foo(void) { printf("foo\n"); }

int main(void) {
    fp *foo_ptr;

    foo_ptr = malloc(sizeof(fp));
    *foo_ptr = foo;

    free(foo_ptr);     // memory is freed

    (*foo_ptr)();      // dangling pointer reused
    return 0;
}
```

Listing 2.1: Example of use-after-free

Explanation of listing 2.1: After `foo_ptr` is freed, the memory it points to, becomes invalid. Reusing `foo_ptr` to invoke a function, now results either in a call to `foo` or in a segmentation fault.

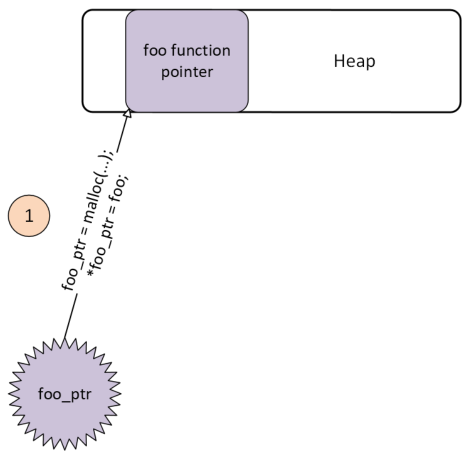
Use-After-Reallocate

While a basic use-after-free may cause a program crash, it becomes significantly more dangerous when the freed memory is reallocated before reuse. This is known as a use-after-reallocate. In such cases, an attacker may intentionally trigger memory allocations (via input for example) that reclaim the previously freed region, populating it with crafted data (such as fake object layouts or function pointers). This transforms the use-after-free into a powerful arbitrary read/write or control-flow hijacking primitive. For instance, if a virtual function pointer is overwritten and then later invoked via a dangling pointer, the attacker can redirect execution to code of their choosing, thus bypassing potential sand-boxing and memory-protection defenses.

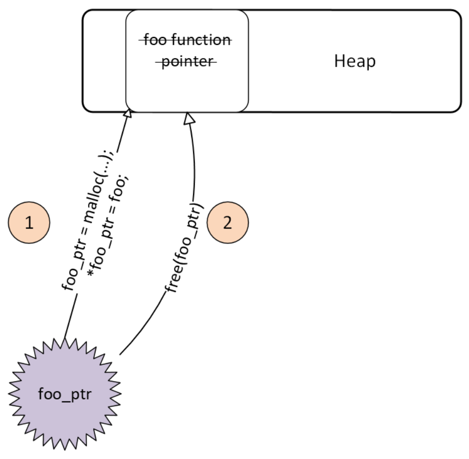
```c
#include <stdio.h>
#include <stdlib.h>

typedef void (*fp)();             // define function pointer type
void foo(void) { printf("foo\n"); }
void bar(void) { printf("bar\n"); }

int main(void) {
    fp *foo_ptr, *bar_ptr;

    foo_ptr = malloc(sizeof(fp));
    *foo_ptr = foo;

    free(foo_ptr);                // memory is freed
    bar_ptr = malloc(sizeof(fp)); // reallocated to a new pointer
    *bar_ptr = bar;               // new content written

    (*foo_ptr)();                 // dangling pointer reused
    return 0;
}
```

Listing 2.2: Example of use-after-reallocate

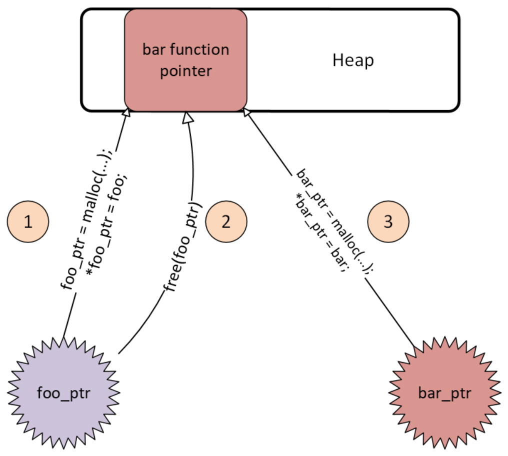
Figure 2.1 is a visual representation of the use-after-free example in Listing 2.2: After foo_ptr is freed, the memory it pointed to becomes invalid. Since we are using the standard glibc allocator (ptmalloc), which uses a first-fit algorithm and both allocations are of the same size, the second malloc will always return the now invalid region that is still pointed to by foo_ptr. When the same memory is reallocated to bar_ptr, the function pointer inside is overwritten with bar. Reusing foo_ptr to invoke a function now results in a call to bar, demonstrating the danger of temporal memory violations when reallocation occurs. In real-world scenarios, such bugs could allow an attacker to overwrite function pointers with malicious payloads.



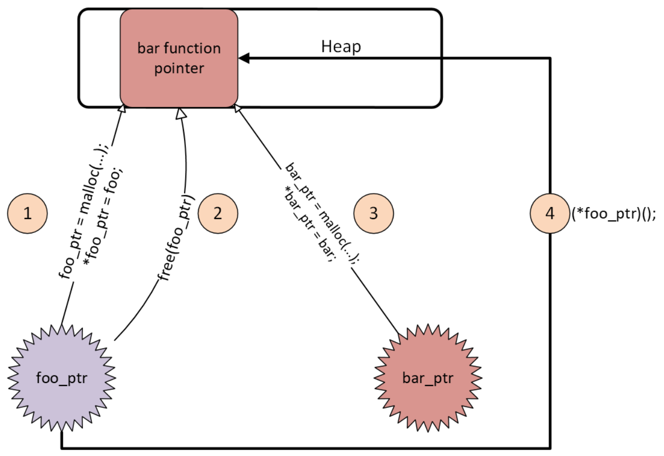
(a) Allocation of `foo_ptr`, assigned to `foo`.



(b) `foo_ptr` becomes a dangling pointer.



(c) Memory is now owned by `bar_ptr`, assigned to `bar`.



(d) Derefencing `foo_ptr` outputs "bar". This is a temporal memory safety violation.

Figure 2.1: Overview of UAF bug across four steps.

2.1.2 Double-Free

A double-free vulnerability occurs when a program calls free() more than once on the same memory address. This results in corruption of the program's internal memory management data structures, which are used by the allocator to track allocated and freed memory regions. Such corruption can cause unpredictable behavior, including program crashes or more severe consequences. In more advanced exploits, a double-free may result in execution of arbitrary code [31, 42].

2.2 Intel's Memory Protection Keys

Intel's Memory Protection Keys, also known as Protection Keys for Userspace (PKU), is a hardware feature available on modern `x86_64` processors that provides a mechanism for enforcing page-based protections, but without requiring modification of the page tables when an application changes protection domains. It works by dedicating 4 previously reserved bits in each page table entry (PTE) to encode a protection key (pkey), allowing memory pages to be grouped into up to 16 protection domains [17].

Access rights for each domain are managed by a special per-core, per-thread register called PKRU (Protection Key Rights Register). This 32-bit register contains two bits- Access Disable (AD) and Write Disable (WD) for each of the 16 pkeys, defining whether read or write access is permitted for a given domain. Since PKRU is a user-accessible CPU register, it can be updated in user space via two dedicated instructions: `RDPKRU` to read and `WRPKRU` to write. Notably, these instructions are low-latency, requiring no system calls, no page table changes, no TLB flushes, and no inter-core synchronization. To initialize and manage protection keys, the Linux kernel (since version 4.6) provides system calls such as `pkey_alloc()`, `pkey_free()`, and `pkey_mprotect()`, along with `glibc` support for runtime access via `pkey_get()` and `pkey_set()` [25].

The thread-level granularity, combined with the efficiency of in-process updates given from MPK, has motivated its adoption in a number of in-process isolation frameworks [6, 33, 54, 60].

```c
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <sys/mman.h>
4  #include <unistd.h>
5
6  int main(void) {
7      int pkey, *buf;
8      buf = mmap(NULL, getpagesize(), PROT_READ | PROT_WRITE,
           MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
9
10     *buf = 0x100;
11     printf("buf at addr: %p contains: %d\n", buf, *buf);
12
13     pkey = pkey_alloc(0, 0);
14     pkey_mprotect(buf, getpagesize(), PROT_READ | PROT_WRITE, pkey);
15
16     pkey_set(pkey, PKEY_DISABLE_ACCESS);
17
18     printf("about to read buf again...\n");
19     /* This will crash, because we have disallowed access. */
20     printf("buf contains: %d\n", *buf);
21
22     pkey_free(pkey);
23 }
```

Listing 2.3: Example of pkeys on Linux

Explanation of Listing 2.3: A memory page is initially allocated using `mmap`, and associated with a protection key via `pkey_mprotect`, and then access is disabled through that key with `pkey_set`. Writing and accessing the page at lines 10 and 11 respectively is allowed, but the subsequent attempt to read from the now protected memory region at line 20 will cause the program to crash. Demonstrating the enforcement of access restrictions imposed by the protection key, independent of the initial `mmap` read/write permissions. It is important to note that `pkey_alloc` and `pkey_mprotect` are system calls, while `pkey_set` (and `pkey_get`) are user-level functions providing fast, per-thread control over memory access rights after the initial key assignment.

2.3 LD_PRELOAD

The *LD_PRELOAD* mechanism in Unix-like operating systems provides a means to selectively load shared libraries before others during the dynamic linking process at runtime [39]. By setting the LD_PRELOAD environment variable, users can specify a list of shared libraries that the dynamic linker (`ld.so`) will load prior to the standard system libraries or application-specific libraries. This preloading behavior allows for the interception and overriding of symbols (functions and variables) provided by other libraries. Consequently, LD_PRELOAD is frequently employed for debugging, profiling, and the implementation of custom library wrappers or extensions without requiring modifications to the target executable [1,4,26].

```c
#define _GNU_SOURCE
#include <dlfcn.h>
#include <stdlib.h>

typedef void *(*_malloc_t)(size_t);
static _malloc_t _malloc = NULL;

void *malloc(size_t size) {
    if (!_malloc) {
        _malloc = (_malloc_t)dlsym(RTLD_NEXT, "malloc");
    }
    foo();

    void *addr = _malloc(size);

    bar(addr);
    return addr;
}
```

Listing 2.4: Example of `LD_PRELOAD`

Explanation of Listing 2.4: This is a `malloc` hook where a custom implementation of the `malloc` function is defined using the LD_PRELOAD mechanism to intercept dynamic memory allocations at runtime. By leveraging the `dlsym` function with the `RTLD_NEXT` flag, it retrieves the original `malloc` implementation from the dynamic linker. Functions `foo` and `bar` could be used for any means desired, such as logging or additional work, before or after actually calling the real `malloc`. Calling the original hooked function is optional. This pattern is commonly used for memory profiling, debugging, or runtime analysis without modifying the original source code. Notably, the hooked function must avoid any operation that could itself invoke `malloc`, directly or indirectly, to prevent infinite recursion.

2.4 Garbage Collection

Garbage collection (GC) is a form of automatic memory management that inherently addresses the problem of temporal memory safety. A garbage collector prevents the creation of dangling pointers, since an object is freed by the collector `iff` the object is provably unreachable from any active part of the program [30,62]. Thus, use-after-free, use-after-reallocate, and double-free errors are not possible. However, these safety guarantees come at the expense of increased memory overhead and runtime performance costs, which vary depending on the specific GC strategy employed [30].

For languages such as C/C++ the usage of *conservative* collection is necessary [9,30]. A collector is conservative, when it receives no assistance from the compiler or runtime system. Consequently, we must treat each contiguous pointer-sized and aligned sequence of bytes as a possible pointer value, called an *ambiguous* pointer. It is up to the collector to distinguish whether an ambiguous pointer is truly a heap pointer. In addition, the safety of conservative collection is vulnerable due to hidden pointers, which can arise from compiler optimizations [8] or from pointer arithmetic such as in XOR lists. To avoid this issue, much like previous work [1,20], in our usage of GC techniques, we will look to deallocate only memory that has been requested to be freed by the programmer. In other words, we will be treating the developers' free calls, as a hint to the memory that should be deallocated.

2.4.1 Mark-Sweep

Mark-sweep collection is one of the four fundamental approaches to GC schemes [30]. As an *indirect collection* algorithm, it first determines which objects are still reachable by the program and then reclaims all others. The algorithm operates in two phases:

1. **Mark**: Shown in Algorithm 1. Traverse the graph of objects, starting from the roots (registers, stack, global variables) which might point to heap objects. `Mark` and recursively follow each heap object found until there is no object left. By the end of this process, we will have marked every reachable heap object.

2. **Sweep**: Shown in Algorithm 2. Parse every object in the heap, any `unmarked` object is considered garbage and will be deallocated.

Algorithm 1: Mark-sweep: marking

```
Function mark_from_roots():
    initialise(worklist);
    foreach $fld \in roots$ do
        $ref \leftarrow *fld$;
        if $ref \neq null \wedge \neg is\_marked(ref)$ then
            set_marked($ref$);
            add(worklist, $ref$);
            mark();
Function initialise($worklist$):
    $worklist \leftarrow \emptyset$;
Function mark():
    while $\neg is\_empty(worklist)$ do
        $ref \leftarrow$ remove(worklist) ;                    // $ref$ is marked
        foreach $fld \in pointers(ref)$ do
            $child \leftarrow *fld$;
            if $child \neq null \wedge \neg is\_marked(child)$ then
                set_marked($child$);
                add(worklist, $child$);
```

Algorithm 2: Mark-sweep: sweeping

```
Function sweep($start, end$):
    $scan \leftarrow start$;
    while $scan < end$ do
        if $is\_marked(scan)$ then
            unset_marked($scan$);
        else
            free($scan$);
        $scan \leftarrow$ next_object($scan$);
```

Figure 2.2 illustrates the state of a mark-sweep garbage collector during the mark phase. The figure presents a simplified object graph alongside a mark stack, which serves as the work list for traversal. Objects currently on the stack (gray objects) are scheduled for further processing, indicating that their references have not yet been fully explored. Objects that have already been marked and removed from the stack are considered fully processed (black object), while any object not yet encountered remains unmarked (white

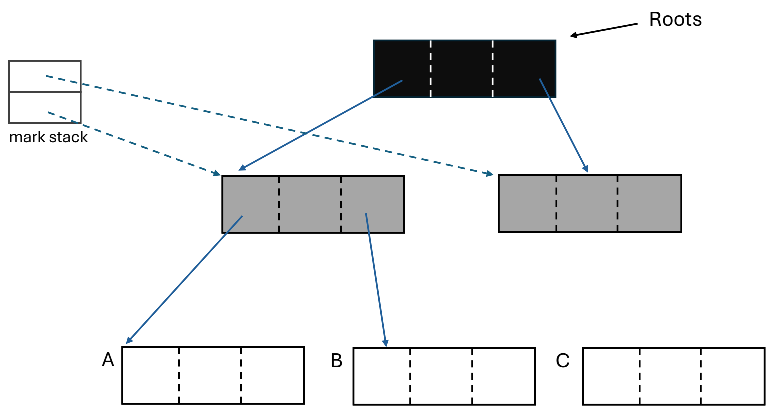


Figure 2.2: A simplified object graph during the mark phase of mark-sweep collection.

objects). In this example, nodes A, B, and C have not been reached and are still unmarked. Once the traversal completes, all reachable objects will have been marked, and any remaining unmarked object, such as C in this case, will be considered unreachable and collected as garbage. The algorithm ensures correctness by maintaining an important invariant: no black (fully processed) object should reference a white (unmarked) one at any point during the traversal. This guarantees that any object still reachable will eventually be discovered and marked, preventing accidental collection of live data.

Chapter 3

Architecture

3.1 Overview

We describe a memory management system composed of a memory allocator and a custom garbage collector, which operates exclusively within safe blocks. A safe block is a code region explicitly marked for temporal safety by the programmer. The goal of this system is not to detect use-after-free bugs, but to prevent them entirely by eliminating the possibility of dangling pointers within these blocks. Manual free calls inside safe blocks are treated as deallocation requests or hints. The system will only fulfill such a request if the targeted object is proven to be unreachable, meaning no remaining references to the object exist.

All allocations outside safe blocks are handled by the standard allocator and are excluded from garbage collection. To guarantee temporal safety within safe blocks without requiring scans of memory used by unsafe code, we must enforce a strict separation between memory allocated inside safe blocks and that used elsewhere. This isolation ensures unsafe code cannot hold references to any safe objects. Meaning, that unsafe code has no read or write permissions to objects allocated in safe blocks.

Our current prototype is limited to uniprocessor (single-threaded) processes. To use the memory management system, the target application only needs to include a simple header file provided by our library, which introduces no external dependencies. Developers specify the desired safe blocks directly in the source code. Importantly, no changes to the application's build process are required. The system is integrated at runtime by loading our library via the `LD_PRELOAD` mechanism.

Our memory management system is built around the following core components:

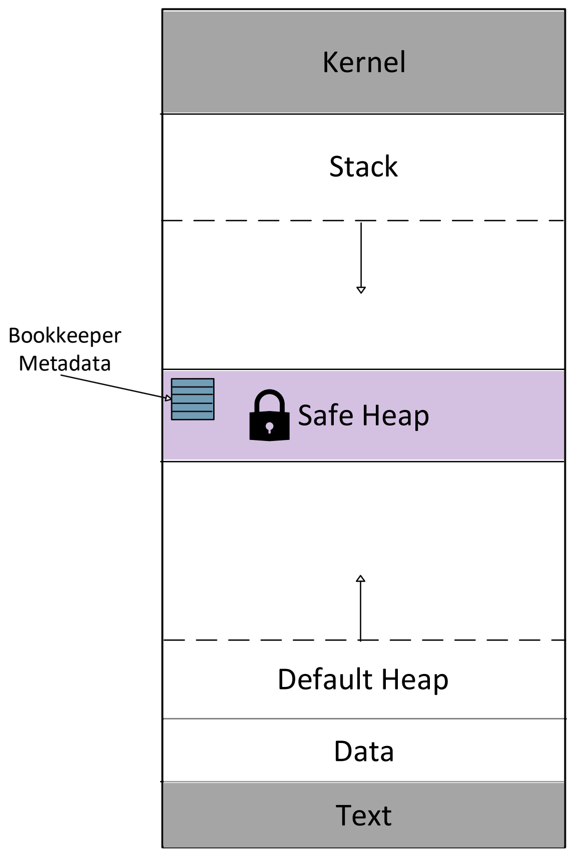
1. **Heap isolation**: A dedicated heap is created for allocations within safe blocks, while also ensuring memory separation from unsafe code.

2. **Metadata bookkeeping**: The system tracks metadata for all allocations in the safe

heap, enabling precise reachability analysis.

3. **Garbage collection**: A mark-sweep collector is used to reclaim unreachable memory that has been requested to be freed within the safe heap.

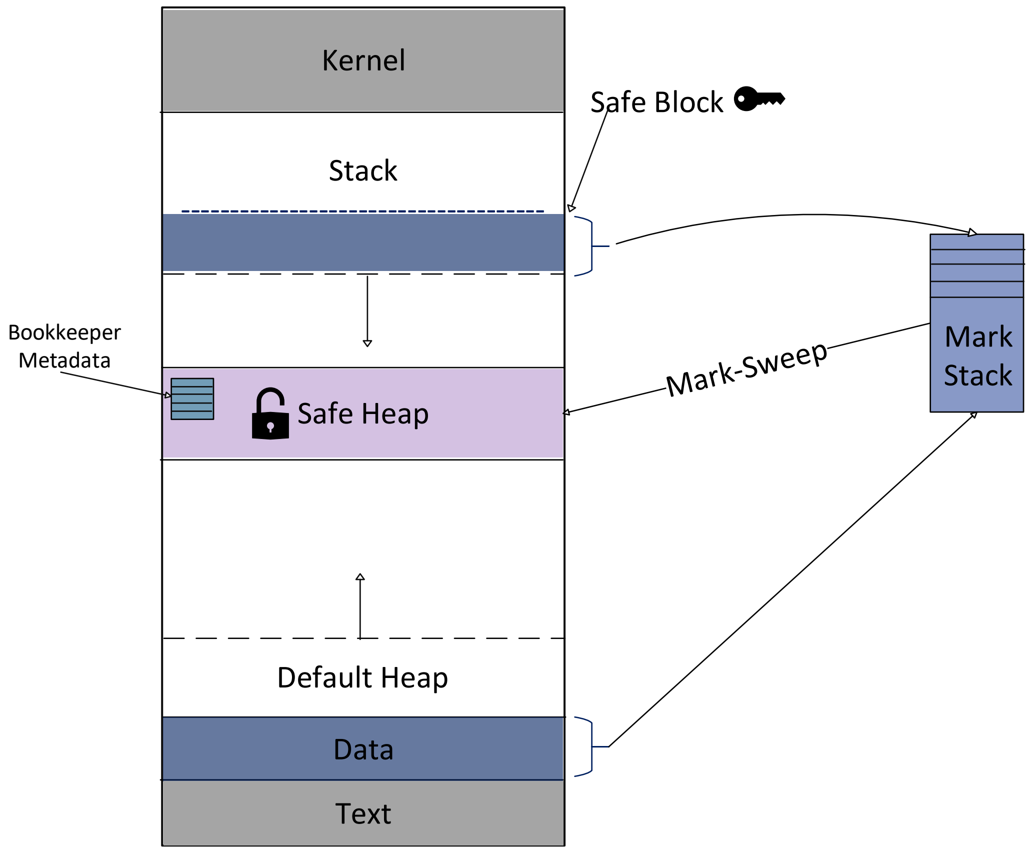
4. **Runtime environment**: Initializes and coordinates the components above, managing their interactions and ensuring correct enforcement of memory safety within safe blocks.



(a) During unsafe mode.

Figure 3.1: Comparison of memory layout outside and within a safe block (Part 1).

Figure 3.1 provides a simplified illustration of a process image. In unsafe mode (shown in subfigure 3.1a), the safe heap is locked and cannot be accessed. In contrast, during safe mode (when executing inside a safe block, as shown in subfigure 3.1b) the safe heap is unlocked. At this point, we use the portion of the stack that corresponds to our safe block along with global variables as roots, in order to initialize the mark stack, and then apply the mark-sweep algorithm to traverse and process the safe heap.



(b) During safe block execution.

Figure 3.1: Comparison of memory layout outside and within a safe block (Part 2).

3.2 Heap Isolation

A straightforward and effective approach to isolating distinct memory regions, used in similar isolation techniques [6, 33], is to create a separate, dedicated heap. This *safe heap* is reserved exclusively for memory allocated within safe blocks. Access to it is enforced using Intel's Memory Protection Keys, ensuring that any code outside of a safe block cannot read from or write to this memory region.

Programs using safe blocks operate in one of two modes: *safe mode*, when executing within a safe block, and *unsafe mode*, when outside. In safe mode, all allocations are handled by the custom memory management system, meaning calls to `malloc` (direct or indirect) return pointers into the safe heap. Code in safe mode retains full access to both the safe heap and the standard (unsafe) heap.

In contrast, when in unsafe mode, allocations are served by the standard allocator, and the program is prevented from accessing the safe heap. Any attempt to read from or write to safe memory in unsafe mode results in a segmentation fault, enforcing strict heap isolation.

3.3 Metadata Bookkeeping

To accurately manage memory within the safe heap, our system maintains metadata for each safe allocated object. This metadata includes the object's address, size, and whether it has been marked for deallocation. The bookkeeping system supports both insertion and removal of entries: newly allocated objects are added to the metadata store, and once an object is confirmed unreachable and reclaimed, its entry is removed.

3.4 Garbage Collection

To ensure temporal safety, our system must eliminate the possibility of dangling pointers. To achieve this, we implement a custom garbage collector based on the mark-sweep algorithm.

Garbage collection begins by identifying the set of root references. In our system, the roots include the portion of the stack used within the current safe block (from the entry point of the safe block up to the current stack pointer), as well as global variables. Importantly, global variables are not limited to those in the target binary, but also include those in all dynamically linked libraries. We do not scan processor registers for root references, for reasons discussed in Chapter 4. We run the collector once the programmer has requested enough times for objects to be deallocated.

Once the roots are identified, the marking phase traverses all reachable objects starting from these roots. After marking, the sweeping phase examines each object in the safe heap. An object is considered eligible for reclamation only if it is unreachable and the program has explicitly requested its deallocation.

Since our collector is conservative, we introduce an additional safeguard: an object must be identified as unreachable across multiple collection cycles before it is actually freed. This reduces the risk of prematurely collecting memory due to false negatives in reachability analysis.

3.5 Runtime environment

The runtime environment is responsible for integrating and orchestrating the various components of our memory management system. It ensures proper initialization, coordination, and enforcement of the system's temporal safety guarantees.

At startup, the runtime redirects all memory allocation functions (such as `malloc` and its variants) to custom handlers. It initializes the isolated safe heap and sets up the metadata bookkeeping subsystem.

During execution, the runtime performs sanity checks to ensure that memory permissions are correctly applied. It transparently routes allocation requests to the appropriate heap, directing them to the safe heap when in a safe block, and to the standard heap otherwise.

The runtime also provides the necessary entry and exit mechanisms for safe blocks, enabling code to transition between safe and unsafe modes. It communicates with the bookkeeping system at runtime to register new safe objects. It also informs the garbage collector about deallocation requests made by the programmer, along with the current location of the safe stack segment in use by the safe block.

3.6 Threat Model

We consider an attacker model in which the adversary can influence the program's memory behavior, specifically by crafting inputs that cause the allocation of memory and accessing memory that has already been freed. The attacker's goal is to escalate control over the program, for instance by hijacking its control flow.

Chapter 4

Implementation

In this chapter, we provide a detailed explanation of the concepts introduced at a high level in Chapter 3. Our system is designed to run using `LD_PRELOAD`, allowing us to intercept standard dynamic memory allocation functions such as `malloc`, `free`, and related calls. As a result, our own memory management components cannot directly or indirectly rely on these functions for dynamic memory allocation.

4.1 Heap Isolation

To maintain two logically and physically separate heaps, we utilize the *mimalloc* allocator [16,41]. With mimalloc, standard allocation functions such as `malloc` and its variants are internally redirected to use mimalloc's backend, rather than the default `glibc` allocator. For example, a call to `malloc(size)` is resolved by mimalloc as a call to `mi_malloc`, which in turn resolves to `mi_heap_malloc(mi_prim_get_default_heap(), size)`.

Mimalloc exposes an API that allows allocations to be made within a user-defined heap using functions such as `mi_heap_malloc` [40]. This is essential for our design: by explicitly specifying the target heap, we avoid the problem of infinite recursion that would occur if our memory management system tried to directly use hooked allocation functions internally. However, indirect calls still remain an issue.

To create a dedicated heap for safe blocks, we first allocate a separate memory region manually using `mmap`. Specifically, we reserve a 4 GiB memory page. In order to later be able to isolate this region, we allocate a protection key using `pkey_alloc`, and apply it to the memory region via `pkey_mprotect`.

Once the memory region is prepared, we create a custom mimalloc heap backed exclusively by this memory. This is done by calling `mi_manage_os_memory_ex` to register the memory region with mimalloc, followed by `mi_heap_new_in_arena` to create a new heap that operates within the specified arena.

It is important to ensure proper alignment of the memory region passed to mimalloc. To meet this requirement, we provide a fixed hint address, `0x300000000000`, during the `mmap` call to increase the likelihood of obtaining an aligned and non-conflicting region.

4.2 Metadata Bookkeeping

The bookkeeper is responsible for tracking all currently allocated safe objects. To do this, it relies on custom data structures, which themselves require dynamic memory. These internal allocations are served using the isolated heap established in the previous step.

Since the primary goal of our current prototype is to validate the core idea rather than optimize performance, we use a simple linear array to store metadata for each safe object. The metadata associated with each object includes: a) Heap address (starting pointer of the object), b) object size, c) flag indicating whether the programmer has requested the object to be freed, d) counter tracking how many times the object was found unreachable.

During initialization, the bookkeeper is provided with the base address and size of the safe heap. This information is used not only to perform internal allocations, but also for bounds checking later during garbage collection.

The bookkeeper exposes a function, `bookkeeper_add`, which is invoked by the runtime environment whenever a new safe object is allocated. But also maintains an internal `bookkeeper_del_entry` function to delete an entry after it's been reclaimed by the garbage collector.

4.3 Garbage Collection

4.3.1 Handling `free` Requests

Rather than immediately returning memory to the allocator when `free(ptr)` is called, our implementation records a "free request" on the corresponding heap object and defers actual deallocation until after a full garbage-collection (GC) cycle. Our function, `bookkeeper_request_free(ptr)` performs the following steps: If the given `ptr` is `NULL`, do nothing and simply return – this is done to match standard `free` semantics. Otherwise, increment a global counter of pending free requests and locate the object's entry in our metadata, and set its `requested_free` flag. Lastly, every `threshold` requests, invoke `trace_roots()` followed by `sweep()` (mark-sweep algorithm).

Running the mark-sweep algorithm is very expensive, ideally we would run it only when necessary, but delaying it for too long might lead to heap fragmentation and thus resulting in worse performance [7].

4.3.2 Root Tracking

Our mark-sweep GC must begin by identifying the set of root pointers-addresses known to hold references into the safe heap. We handle two categories:

Global and Static Data

To discover all writable data regions in the main program and dynamically loaded libraries, we use `dl_iterate_phdr`. This function will walk through the list of our application's shared objects. It takes as an argument, a user callback which will be called once for each object, until all shared objects have been processed. For each shared object's program headers we skip segments that are not both loadable (`PT_LOAD`) and writable (`PF_W`) since data could only be found in a loadable and writable segment [36]. We also compute each segment's start and end addresses by adding the object's base load address to the segment's virtual address and memory size. Then, we need to align both boundaries to machine-word (`uintptr_t`) alignment using clang builtins [38]. In the end, we append the resulting "region" to an array of global data segments.

It is important that we skip segments that belong to our runtime library, since we do not want to pollute the roots with our references.

Once all regions are collected, `trace_roots()` iterates over each region, scanning the words within for candidate pointers. On later collection cycles, we repeat this process since we do not know whether a new shared object has been added, or an old one has been closed.

Stack Data

In addition to global data, we scan a "safe stack" region provided by our runtime environment, ranging from `safe_stack->top` down to `safe_stack->bottom`. This region includes local variables and caller-saved registers that may contain heap references. We avoid manually inspecting each register because, by the time our functions execute, relevant values will have already been pushed onto the stack. Moreover, since we perform multiple passes to detect unreachable objects, it's unnecessary to track registers explicitly. Attempting to do so would also introduce significant architecture-specific complexity.

4.3.3 The Mark Phase

After initializing an empty worklist, `trace_roots()` pushes every heap object referenced by words in the global and stack regions onto this worklist. It then enters the main mark loop:

- Pop an object entry from the worklist.

- Compute its payload region by stripping any tag bits (`UNTAG()`) and aligning its base address up and end address down to `uintptr_t` boundaries.
- Call `mark_from_region(worklist, start, end)`, which scans each word in the given memory region. For each word, if its value (`addr`) falls outside the safe heap's address range, ignore it and continue to the next one. Otherwise, search the metadata allocation table to find which object contains `addr`. If that object is not yet marked, *mark* it and then push it onto the worklist. An object is marked by setting its low-bit (`addr |= 0x1`). Since the object's address is word-aligned, its least significant bit is never set.
- Repeat until the worklist is empty.

This conservative scan treats any word that happens to fall within a live object's address range as a potential pointer.

4.3.4 The Sweep Phase

Once marking completes, the `sweep()` function iterates over every entry in the metadata allocation table:

- If the entry carries the mark bit, clear it (preparing for the next GC cycle).
- Otherwise, if the entry was requested for free and has remained unreachable for at least a small threshold of GC cycles, it is deemed garbage. We then call `mi_free()` on its address and remove the entry from the table.
- Objects not yet requested for free or still within the unreachable threshold have their `unreachable_cnt` incremented and are retained.

In summary, our implementation defers frees, conservatively scans both global and stack roots, uses a simple bit-marked worklist for tracing, and reclaims only objects that are both unreachable and explicitly requested for freeing. This design keeps the collector straightforward and avoids any additional per-object pointer metadata.

4.4 Runtime Environment

4.4.1 Runtime Environment Initialization and Teardown

The runtime environment is responsible for orchestrating three core modules: the heap-isolation, the bookkeeper, and the garbage collector. All of the user-visible allocation functions (`malloc`, `calloc`, `realloc`, and `free`) are hooked via a constructor so that every dynamic allocation passes through our framework.

Constructor (`hook_init`)

We annotate `hook_init` with `__attribute__((constructor))`, ensuring it executes before `main()` and before any user allocations occur. First, to prevent recursion during startup, we set a global `INITIALIZING = true` and stash the default mimalloc heap in `tmp_heap`; any hooked calls during this phase are forwarded to `mi_heap_malloc(tmp_heap, ...)`, a choice motivated by considerations discussed in Section 4.5. Next, we resolve the original allocator symbols using `dlsym(RTLD_NEXT)`, retrieving real versions of `malloc`, `calloc`, `realloc`, and `free`, which are stored in function pointers. We then create the safe heap by calling `create_safe_heap(&safe_heap)` and temporarily grant it read-/write permissions via `pkey_set_perm(..., RDWR)`. This is necessary step, even though we are theoretically still in an unsafe state, due to constraints explained in Section 4.5. Afterward, we initialize the bookkeeper by invoking `bookkeeper_init(safe_heap.heap, ...)`, which registers the heap's base address and size. Finally, we complete initialization by clearing `tmp_heap`, revoking access to the safe heap with `pkey_set_perm(..., NO_ACCESS)`, and setting `INITIALIZING = false`. At this point, all subsequent user allocations are routed through our hooks and enforced by the safe-heap's permission model.

Destructor (`hook_exit`)

We annotate `hook_exit` with `__attribute__((destructor))`, ensuring it executes automatically when the runtime shared object is unloaded. The teardown process mirrors the initialization steps in reverse. First, we re-enable read/write access to the safe heap (`pkey_set_perm(..., RDWR)`) to allow final cleanup without triggering protection faults. If debugging is enabled, we optionally dump internal state via `bookkeeper_dump()`. Next, we call `bookkeeper_exit()` to release or flush any remaining metadata. Finally, we destroy the safe heap using `destroy_safe_heap(&safe_heap)`, which unmaps the memory region and releases its associated pkey.

Sanity Checks and Safe-Block Delimiters

To ensure that the safe heap protections are never accidentally violated, we implement two sanity-check routines: `safe_block_sanity_check()` asserts that, when inside a designated "safe block," the heap's pkey permissions are `RDWR` and that the stack-bottom pointer is nonzero. While `unsafe_block_sanity_check()` ensures that, outside of any safe block, the safe heap pkey is `NO_ACCESS`.

Code regions that legitimately allocate or free memory from the safe heap must bracket these operations with `enter_safe_block(...)` and `exit_safe_block()`. When entering the safe block the current stack's bottom is stored, `in_safe_block = true` is set, and read/write (RDWR) permissions are granted to the safe heap. When the operation is complete, exiting the safe block resets the stack markers, the in_safe_block flag is cleared, and access to the safe heap is revoked using pkey_set_perm(..., NO_ACCESS). Additionally, within a safe block, functions set_exempt() and unset_exempt() can be invoked to allow selective allocations to bypass the safe heap. This could be useful for handling large or special-purpose objects. The important motivation and details for this exemption mechanism are discussed in Section 4.5.

The functions detailed above are not meant to be used directly by the developer but via wrappers given in the header file.

Hook Structure: malloc**,** calloc**,** realloc**,** free

Each allocation hook follows this general pattern:

1. **Initialization bypass:** If INITIALIZING is true, immediately forward to the mimalloc functions on tmp_heap and then return.
2. **Safe block path:** Perform safe_block_sanity_check(). After, if exempt is set, call the original glibc allocation and then return. Otherwise, allocate in the safe heap and record the allocation with bookkeeper_add().
3. **Unsafe block path:** Perform unsafe_block_sanity_check(). Then, temporarily enable RDWR on the safe heap. Call the original glibc allocation. And finally restore NO_ACCESS. We analyze the necessity of briefly disabling and enabling the permissions of our safe heap during unsafe allocation in section 4.5.

Free hook and Stack Capture In the free hook, when in a safe block and exempt is unset, we capture the current stack pointer (via inline assembly on rsp), align it down, and pass it, along with the pointer being freed, into bookkeeper_request_free(...). This allows the garbage collector to include the precise stack-root region during the next GC cycle.

Together, these mechanisms makeup our memory management system.

4.4.2 API: safe_blocks.h

This header exposes the lightweight API that application developers include to bracket sections of code that annotate safe blocks. All symbols are declared weak, so that simply including this header in a project does not force additional link-time dependencies. Calls to any of these functions will only be emitted if the corresponding hook library is actually loaded at runtime.

Weak Hook Declarations

- `enter_safe_block(void *safe_stack_bottom);`
- `exit_safe_block(void);`
- `set_exempt(void);`
- `unset_exempt(void);`

Each of these is marked with `__attribute__((weak))`, meaning that if the hook library is not present, the application will still link cleanly. Calls are guarded by a simple 'if (enter_safe_block) ... ' test so that no undefined-symbol errors occur at runtime.

Safe block Macros The provided header defines three main macros, each serving as a wrapper around the runtime functions just mentioned. Since they are `weak` symbols, each macro checks that its corresponding function/symbol is defined before using it.

`ENTER_SAFE_BLOCK` Ensures the entry of a safe block. It expands to code that first captures the caller's current frame address via `__builtin_frame_address(0)` [23]. And then, calls `enter_safe_block(...)` with that address (marking the bottom of the stack for root scanning).

`EXIT_SAFE_BLOCK` Ensures the exit of a safe block. It expands to code that calls `exit_safe_block` to revoke safe-heap access and clear the stack markers.

`EXEMPT(`*foo*`)` Wraps an arbitrary statement or block *foo* so that, `set_exempt` is called immediately before *foo*, and `unset_exempt` is called immediately after. This allows a single allocation or free to bypass the safe heap logic within a safe block region. We show and discuss the usage of this macro in section 4.5.

Usage Pattern In client code, one simply writes:

```
ENTER_SAFE_BLOCK;
  // any malloc/calloc/realloc/free here will go through
  // the isolated heap, bookkeeper, and GC
EXIT_SAFE_BLOCK;
```

If a particular call must not use the safe heap, the developer can write:

```
ENTER_SAFE_BLOCK;
  EXEMPT(ptr = malloc(huge_size));
  // ptr is allocated via the original glibc malloc, not the safe heap
EXIT_SAFE_BLOCK;
```

Link-Time Simplicity Because all APIs are weak and guarded, integrating our runtime simply requires linking against the shared library with the LD_PRELOAD mechanism. If the library is missing, the macros compile to no-ops (semantically) and all calls fall back to the normal allocation routines without error.

This header thus provides a zero-overhead, opt-in mechanism for developers to isolate and manage memory within critical sections of their code, while remaining entirely transparent if the runtime is absent.

4.5 Issues Encountered

4.5.1 Temporary Heap and Permissions During Initialization

During the execution of the `hook_init` constructor, we perform critical setup for the runtime environment, including installation of hooks, creation of the safe heap, and initialization of the bookkeeper module. At this early stage, however, the system is in a transitional state: our hooks may not be fully installed, internal heap structures may be incomplete, and proper isolation through permission enforcement has not yet been fully established.

Despite our intention to avoid dynamic memory usage during this phase, it is difficult to guarantee that no allocations occur, especially if linked libraries or lower-level runtime components implicitly call `malloc`, `calloc`, or similar routines. To account for this, we route all allocations during initialization to a temporary heap, provided by mimalloc's default heap interface. Since these allocations are internal and not programmer-controlled, there is no need to register or track them with the bookkeeper.

Additionally, we temporarily grant `RDWR` permissions to the safe heap region during initialization. Although execution begins in an unsafe context, mimalloc may need to manipulate internal heap metadata (e.g., during heap creation or resizing). These updates can involve direct access to memory within our isolated heap region. Therefore, enabling read/write access ensures that mimalloc can safely complete its operations during setup without triggering protection faults.

Once all subsystems are correctly initialized, the runtime revokes all access to the safe heap (by setting it to `NO_ACCESS`) and marks the end of the initialization phase by clearing the `INITIALIZING` flag.

4.5.2 Permission Toggling in Unsafe Allocation Paths

Even when operating outside of a safe block (unsafe execution context) we still need to temporarily enable `RDWR` permissions on the safe heap region. At first glance, this might seem unnecessary, as all allocations in this context are routed to the default (unsafe) heap, and the safe heap is not intended to be involved.

However, mimalloc's internal behavior introduces complication. Despite targeting a separate heap, mimalloc may still perform global or shared metadata updates that affect memory mappings residing within the safe heap's address space.

To prevent faults during these operations, we temporarily grant `RDWR` permissions on the safe heap immediately before calling the underlying allocation function in the unsafe path. Once the allocation is complete and control returns, we promptly restore the safe heap's permissions to `NO_ACCESS` to maintain isolation guarantees. This brief window of read/write access is tightly scoped and essential for compatibility with mimalloc's internals.

4.5.3 Handling *Stateful* Library Functions

In our design, any allocation performed inside a `ENTER_SAFE_BLOCK`...`EXIT_SAFE_BLOCK` region is placed on the isolated safe heap, and later reclaimed by the garbage collector only if explicitly requested. However, certain standard library routines like printf, fprintf, puts, etc., maintain their own internal buffers across multiple calls. If such a "stateful function" is first invoked inside a safe block, its buffer lives on the safe heap. A later invocation in an unsafe context will attempt to touch or free that same buffer while the pages are marked `NO_ACCESS`, leading to a segmentation fault.

Root Cause

When a stateful function like `printf` allocates its I/O buffer, it stores the buffer pointer in one or more internal fields (e.g. `stdout->_IO_buf_base`). All subsequent calls reuse that pointer/buffer. Thus, a program such as:

```
ENTER_SAFE_BLOCK;
    printf("first call\n");    // buffer on safe heap
EXIT_SAFE_BLOCK;
printf("second call\n");       // reuses buffer -> segfault
```

will fault because the second call runs with the safe-heap region protected against writes.

Methods Considered

1. PURGE on Exit One idea was to explicitly purge a stateful function's buffers at the end of each safe block. That is, wrap the call as

```
PURGE(printf("..."));
```

so that upon exit we would (a) free the buffer and then (b) locate and nullify every internal pointer to it. In principle the collector can chase all root pointers and zero them out, forcing the library to allocate a fresh buffer on the unsafe heap. However, even for a single `printf` instance we observed seven distinct internal pointers in glibc's data structures. Zeroing each one at every exit is both complex and potentially expensive, especially if many stateful calls occur.

2. EXEMPT An alternative is to mark specific call sites so that all allocations they perform go directly to the unsafe heap. Using a macro such as

```
EXEMPT(printf("..."));
```

the collector simply bypasses the safe heap for that invocation and any sub-allocations it triggers. This avoids pointer chasing and nullifying entirely.

3. Automatic Hooking of Known Stateful Routines A more aggressive approach is to intercept `printf`, `fprintf`, `puts`, and other known stateful functions (much like we hook malloc and such), forcing all their allocations into the unsafe heap unconditionally. While this would relieve developers of per-call annotations, it risks hidden behavior changes if new or obscure stateful routines are missed.

Final Solution

We chose the `EXEMPT` pattern as the cleanest trade-off between performance, correctness, and developer control. It offers simplicity, as the implementation only needs to check a single `exempt` flag during allocation hooks without requiring additional logic at safe block exits. It also delivers strong performance by avoiding costly pointer-chasing or buffer-zeroing at runtime. Lastly, it provides precision, allowing developers to explicitly mark only those calls that must not interact with the safe heap.

In practice, one writes:

```
ENTER_SAFE_BLOCK;
    EXEMPT(printf("stateful call\n"));
EXIT_SAFE_BLOCK;
printf("unsafe next call\n");
```

and the first `printf` and its internal buffer are allocated on the unsafe heap, avoiding any later faults. This pattern has proven sufficient for GNU Coreutils' use of `printf`/`fclose` and similar buffered-I/O routines.

By giving developers explicit but lightweight control over stateful functions, we maintain both the safety guarantees of our isolated heap and the performance characteristics of standard library I/O.

Chapter 5

Evaluation

In this chapter, we evaluate our prototype across three key dimensions:

- Correctness: We verify that safe blocks function as intended, effectively preventing use-after-free (UAF) vulnerabilities within protected regions.
- Practicality and Usability: We assess how easily our prototype can be integrated into existing codebases and whether the adoption process is feasible for real-world projects.
- Performance: We measure the runtime overhead introduced by our system and explore how various tunable parameters impact performance.

5.1 Experimental Setup

All evaluations were conducted on an Intel system with a Tiger Lake 11th Gen Intel® Core™ i7-1185G7 @ 3.00 GHz CPU [29], equipped with 16 GiB of DDR4 RAM. Experiments were run inside an Arch Linux Docker container, hosted on a system running `LMDE 6 "Faye"` with Linux kernel version 6.1. We also used the latest mimalloc version of 2.2.3-1.

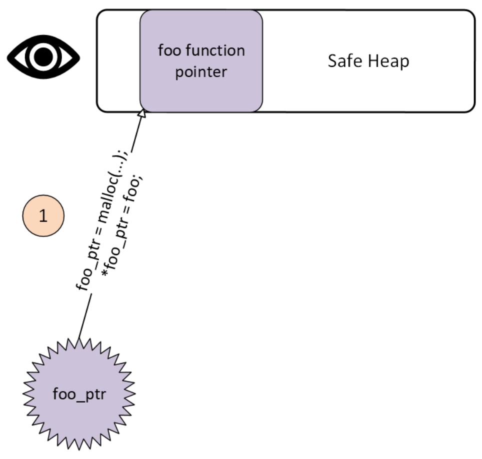
5.2 Correctness

To evaluate correctness, we tested our prototype using several toy examples of use-after-free (UAF) bugs. In each case, we enclosed the vulnerable code within a safe block to determine whether our mechanism could successfully prevent the bug. One such example is shown in Listing 5.1 along with its visual representation in Figure 5.1, which is a rewritten version of the original buggy code presented in Listing 2.2. As expected, no UAF occurred during execution.

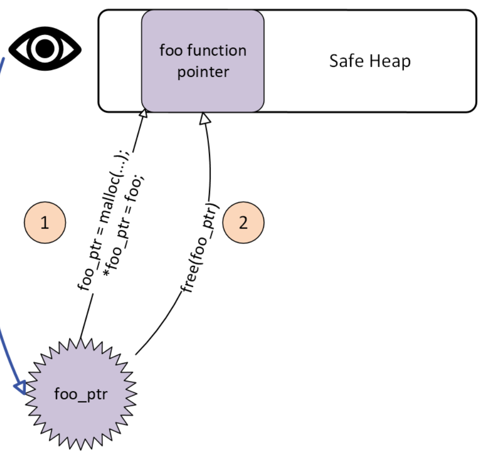
Specifically, the object pointed to by `foo_ptr` was not actually reclaimed by the `free()` call at line 16. We ran the program using a debug build of our prototype, which logs internal information such as the set of references identified during the mark-sweep process. The logs confirmed that the heap object was still reachable through a reference on the safe stack, and thus was not collected, demonstrating that our system correctly prevents UAFs in protected regions.

```
 1 #include "safe_blocks.h"
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 typedef void (*fp)();             // define function pointer type
 6 void foo(void) { printf("foo\n"); }
 7 void bar(void) { printf("bar\n"); }
 8
 9 int main(void) {
10     ENTER_SAFE_BLOCK;
11     fp *foo_ptr, *bar_ptr;
12
13     foo_ptr = malloc(sizeof(fp));
14     *foo_ptr = foo;
15
16     free(foo_ptr);                // memory is requested to be freed
17     bar_ptr = malloc(sizeof(fp));
18     *bar_ptr = bar;
19
20     (*foo_ptr)();
21     EXIT_SAFE_BLOCK;
22     return 0;
23 }
```

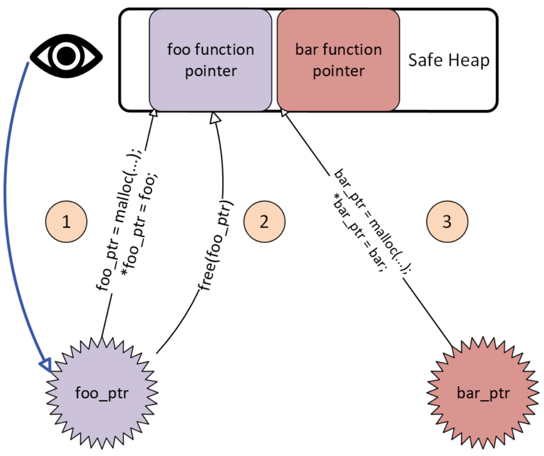
Listing 5.1: Example of a use-after-free bug enclosed in a safe block



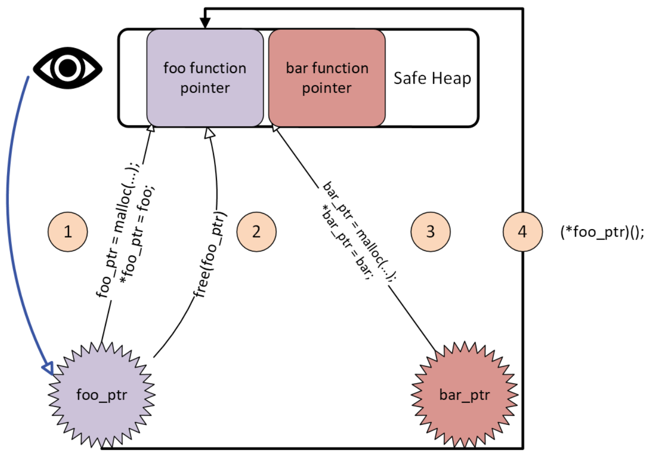
(a) Allocation of `foo_ptr`, assigned to `foo`.



(b) `foo_ptr` is still the owner of that memory.



(c) Different memory is now owned by `bar_ptr`, assigned to `bar`.



(d) Dereferncing `foo_ptr` outputs "foo". Our safe block ensured temporal safety.

Figure 5.1: Overview of how a safe block prevents a UAF bug across four steps.

5.3 Practicality and Usability

To assess the practicality and usability of our prototype in real-world scenarios, we undertook the integration of safe blocks into a selection of widely-used C projects: `nginx` [51], `lexbor` [10], `jq` [18], `libxml2` [61], `llhttp` [28], `gnu-coreutils` [55] and `ncat` [37]. In each case, we manually inserted our safe blocks in various sections of the source code. Specifically, functions responsible for processing user input.

Integration at the build level of each project was fully successful, facilitated by the use of weak symbols and the `LD_PRELOAD` mechanism. However, runtime integration posed substantial challenges. The inherent complexity and maturity of these software systems introduced numerous difficulties, including compatibility issues, subtle runtime behaviors, and performance regressions. These challenges indicate that, while the prototype is promising in terms of minimal build-time disruption, further engineering effort is required to ensure seamless operation in production-grade software. A more detailed discussion of these limitations and potential future improvements is provided in Chapter 6.

Moreover, during our integration efforts, we observed that nearly all of the projects mentioned above employed their own internal mechanisms for managing temporal memory safety. While a detailed analysis of each project's memory management strategy is beyond the scope of this thesis, it is important to note that these mechanisms frequently conflicted with the semantics and expectations of our safe blocks. By "design," we refer to the explicit, project-specific code responsible for managing dynamic memory allocation, deallocation, and reuse. These incompatibilities posed additional challenges during integration and highlight a broader issue regarding the coexistence of custom memory management schemes with external temporal safety frameworks. We elaborate on these findings and their implications in Chapter 6.

5.4 Performance

5.4.1 Experiment

Given the aforementioned challenges in modifying the source code of large, preexisting projects, we opted to evaluate performance using a simpler, self-contained example that exercises our prototype in a meaningful way. This approach enables us to conduct a controlled comparison without the complexities introduced by integrating with full-scale software systems.

It is important to note that, in this evaluation, we wrap the entirety of the library usage within a safe block. In doing so, we effectively apply our temporal safety enforcement across all memory allocations performed by the library, thereby providing an overestimation of the potential performance overhead introduced by safe blocks.

For this experiment, we selected `json-c` [32], a lightweight JSON parsing library written in C that is used in several large projects such as sway [57], thunderbird [3] and libelf [2]. We developed a simple and small benchmark program using `json-c`, shown in Listing 5.2, to measure the performance impact of our system. The program parses the JSON data of a given file, and then simply prints it out, within a safe block.

```c
#include "safe_blocks.h"
#include <json-c/json.h>
#include <stdio.h>
#include <stdlib.h>

char *read_file(char *filename) {...}

int main(int argc, char *argv[])
{
    char *data = read_file(argv[1]);
    ENTER_SAFE_BLOCK;
    struct json_object *parsed_json = json_tokener_parse(data);

    EXEMPT(printf("%s\n", json_object_to_json_string(parsed_json)));

    json_object_put(parsed_json); // parser cleanup
    EXIT_SAFE_BLOCK;
    free(data);
    return 0;
}
```

Listing 5.2: Simple JSON parser

To quantify the overhead of safe blocks, we selected two large JSON datasets from Hugging Face [22, 53]. From each dataset, we extracted the first 1 000, 5 000, and 10 000 lines, yielding six test inputs: `lt-1k`, `lt-5k`, `lt-10k`, `md-1k`, `md-5k`, and `md-10k`.

As described in Chapter 4, our runtime offers a configurable *threshold* parameter: after every `threshold` calls to `free()`, the mark-sweep collector is invoked. To characterize performance, we tested thresholds of 10, 50, 100, 200, 400, 500, and 1 000 on each input, and we also measured a baseline build (the unmodified parser without any safe blocks). Each configuration, including the baseline, was executed ten times to reduce variance, and we report the mean wall-clock time as recorded by GNU `time` [56].

5.4.2 Results

Figure 5.2 and Table 5.1 report the absolute runtimes for each configuration. The unmodified baseline parser completes in negligibly small time, whereas enabling safe blocks incurs substantial overhead. At the most aggressive collection setting (threshold 10), parsing the largest inputs exceeds 1 000s, even at threshold 1 000, runtimes remain but close to 15s.

To better quantify the performance impact of safe blocks, Figure 5.3 and Table 5.2

present the slowdown ratio, defined as

$$\text{Overhead Ratio} = \frac{\text{safeBlocksTime}}{\text{baselineTime}}.$$

For clarity, we grouped results from the two datasets by input size (e.g., combining `lt-1k` and `md-1k` into a single 1k-line input category). The results reveal substantial overhead: at the most aggressive garbage-collection setting (threshold 10), the slowdown reaches approximately 1 014x for 1 000-line inputs and peaks at roughly 57 112x for 10 000-line inputs. Even at the most relaxed threshold (1 000), the overhead remains non-negligible, ranging from 10x to 719x depending on input size.

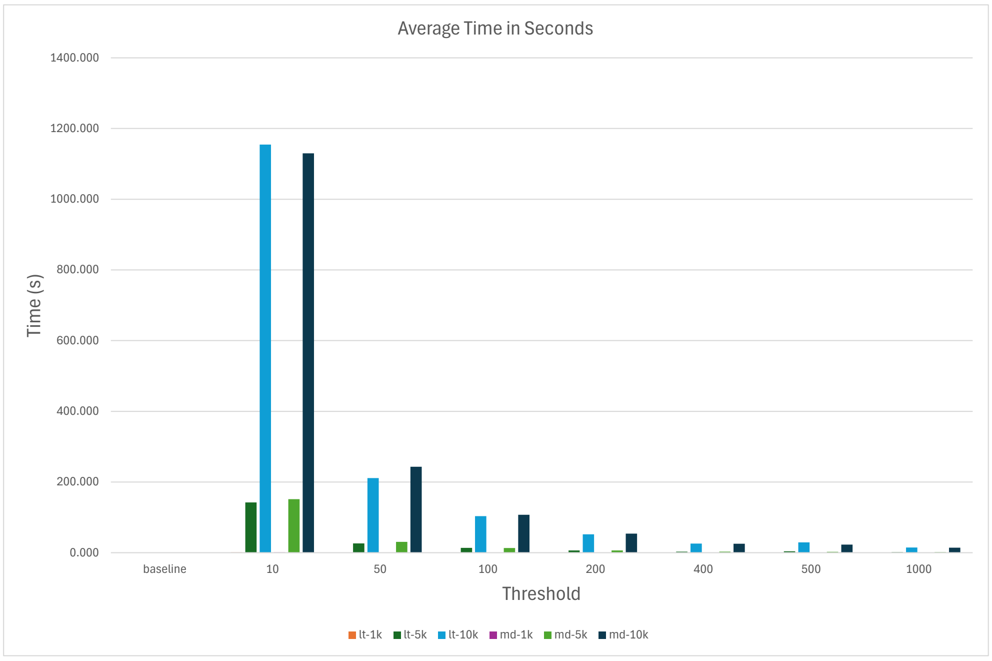


Figure 5.2: Y-axis is the absolute parsing times (mean wall-clock seconds) for each input size and the X-axis is the GC threshold. Lower thresholds trigger more frequent collection, leading to higher runtimes.

Threshold	lt-1k	lt-5k	lt-10k	md-1k	md-5k	md-10k
Baseline	0.001	0.003	0.010	0.001	0.003	0.030
10	1.042	142.106	1154.781	0.986	151.328	1129.684
50	0.190	26.468	211.139	0.199	30.522	243.117
100	0.091	13.483	103.540	0.091	13.131	107.128
200	0.050	6.466	51.882	0.050	6.460	53.975
400	0.030	3.085	25.964	0.030	3.190	25.443
500	0.031	3.973	29.233	0.020	2.760	22.951
1000	0.010	1.641	14.470	0.010	1.628	14.305

Table 5.1: Mean parsing time (seconds) measured with GNU `time`, across six inputs and GC thresholds.

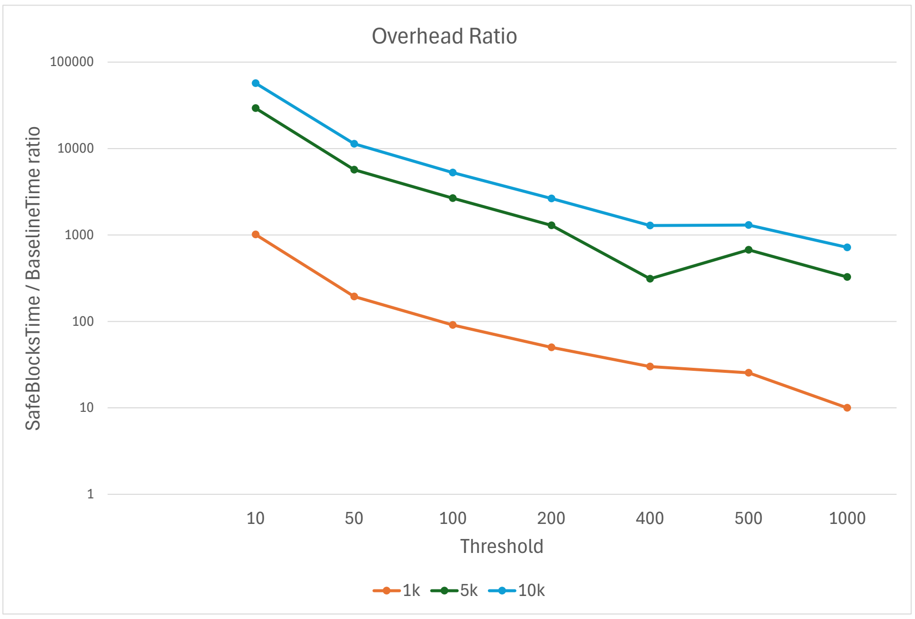


Figure 5.3: Y-axis is the slowdown ratio (safe blocks vs. baseline) on a logarithmic scale. And the X-axis is threshold parameter. Each dotted line corresponds to a different input size (1k, 5k, 10k lines).

Threshold	1k	5k	10k
10	1 014.0	29 343.4	57 111.6
50	194.5	5 699.0	11 356.4
100	91.0	2 661.4	5 266.7
200	50.0	1 292.6	2 646.4
400	30.0	311.5	1 285.2
500	25.5	673.3	1 304.6
1000	10.0	326.9	719.4

Table 5.2: Average slowdown ratio across GC thresholds and input sizes.

As Table 5.3 illustrates, only a fraction of the `free()` requests were actually satisfied. This behavior stems from our mark-sweep policy, which only reclaims objects that are both explicitly freed and provably unreachable. In our small benchmark programs, many pointers remain in scope on the safe stack for the duration of execution, so the collector considers those objects still live and defers their reclamation. In larger or more complex applications, references would naturally go out of scope or be overwritten, allowing the collector to satisfy a greater proportion of free requests.

Size	md		lt	
	Free Requests	Actual Frees	Free Requests	Actual Frees
1k	2 411	603	2 411	403
5k	12 011	3 003	12 011	2 003
10k	24 011	6 003	24 011	4 003

Table 5.3: Comparison of `free()` requests and actual frees by input size and dataset.

5.4.3 Final Observations

Overall, our evaluation demonstrates that the current safe blocks prototype incurs prohibitive performance overhead. This outcome is expected, as our primary objective was to validate the feasibility of the safe blocks abstraction rather than to optimize runtime efficiency. The literature on garbage collection offers numerous techniques for reducing overhead, ranging from generational and incremental collectors to parallel and concurrent algorithms, that could be adapted to improve performance in future iterations of our system [30]. This is discussed in more detail in Chapter 6.

Chapter 6

Discussion

6.1 Limitations

Despite demonstrating the feasibility of safe blocks, our current prototype has several important limitations:

6.1.1 Uniprocessor Design

The prototype assumes a uniprocessor execution model and does not support multithreaded applications. Extending temporal-safety enforcement to concurrent or parallel code would require careful synchronization of the safe heap, the safe stack, and the garbage collector.

6.1.2 High Runtime Overhead

As shown in Chapter 5, invoking the mark-sweep collector even infrequently imposes significant performance penalties. While research on advanced garbage-collection techniques (e.g., generational, incremental, and concurrent collectors) provides many paths to optimization [30], integrating these techniques into our framework remains future work.

6.1.3 Integration Complexity

Each large C codebase typically implements its own memory-management conventions and temporal-safety mechanisms. These project-specific designs often conflict with the assumptions of safe blocks, making seamless integration into existing, mature codebases extremely difficult.

6.1.4 Data Isolation Boundary

Currently, objects allocated within a safe block remain quarantined in the safe heap and cannot be safely returned to "unsafe" code. In practice, many applications must transfer data (e.g., parsed input buffers) from the protected region back into the main program. Supporting controlled sharing of safe-heap objects across the safe-block boundary will require a well-defined transfer protocol or copy mechanism.

6.1.5 Hardware Dependency

Our implementation relies on Intel's Memory Protection Keys (MPK) for enforcing access rights. Porting safe blocks to processors without MPK or to other architectures will necessitate an alternative isolation mechanism.

6.1.6 LD_PRELOAD Approach

While convenient for prototyping, using LD_PRELOAD to interpose on memory allocators can break when applications statically link their allocators or employ custom memory-management libraries. A more robust implementation would involve compiler intervention (such as an `LLVM` pass).

6.2 Future Work

6.2.1 Performance Optimization

Improving the runtime performance of our current prototype is a natural direction for future work. This includes optimizing the internal data structures used for metadata tracking and reachability analysis. For instance, the two-level pointer lookup tree employed by Boehm-GC [27] provides fast, space-efficient metadata access. Additionally, reducing the overhead of frequent mark-sweep invocations, either by batching or by lazily triggering collections, could substantially improve real-world performance.

6.2.2 Concurrent and Parallel Execution

At present, safe blocks are restricted to single-threaded programs. Extending support to multithreaded applications presents both conceptual and engineering challenges. Not only must safe blocks coordinate concurrent access to the safe heap and stack, but the garbage collector itself must become concurrent to avoid long stop-the-world pauses. Because MPK pkeys use a per-thread PKRU register to manage access rights, this design naturally extends to multithreaded contexts, enabling efficient, thread-local enforcement of heap isolation.

6.2.3 Garbage Collector Integration

Rather than building and maintaining a custom garbage collector, a more pragmatic approach may be to integrate an existing mature collector such as Boehm-GC [27] into the safe blocks framework. This would allow us to take advantage of decades of optimization work, including generational, incremental, and parallel collection techniques. However, compatibility layers may be needed to enforce isolation semantics and safe block boundaries within Boehm's architecture.

6.2.4 Cross-Boundary Data Transfer

As discussed in Section 6.1, safe blocks currently do not allow values allocated within the safe heap to be accessed from outside. Many practical applications require returning processed data from the safe context to the host program. A future version of the system could support this through deep copying of the "returned" data into the standard heap.

6.2.5 Compiler-Level Instrumentation

While our use of `LD_PRELOAD` offers a convenient way of using our library, it struggles in the case of static linking, custom allocators, or nonstandard linking behavior. Furthermore, usage of dynamic hooks can introduce infinite-recursion hazards that complicate the implementation. A more robust solution would involve building an `LLVM` compiler pass that automatically instruments memory allocations and function boundaries to insert and manage safe blocks.

6.2.6 Evaluation on Real-World Applications

Finally, applying safe blocks to larger, real-world C programs would be an essential step to evaluate practical utility. This would test not only performance and compatibility, but also the ability of the model to integrate with various coding styles, memory idioms, and modular designs. Gathering empirical feedback from such case studies would likely uncover both previously unseen limitations and new optimization opportunities.

Chapter 7

Related Work

7.1 Temporal-Safety

MarkUs and Minesweeper [1, 20] are both very close to our work. Both rely on tracking dangling pointers. MarkUs using LD_PRELOAD runs programs with the Boehm-GC collector [27] to eliminate dangling pointers, treating `free()` calls as hints/requests. By delaying reclamation until the total size of "to-be-freed" objects exceeds a configurable fraction of the heap, MarkUs achieves an average overhead of only 1.1x (max 2x) on SPEC CPU2006. It also unmaps pages for very large objects to avoid large delays. Adapting Boehm-GC to operate strictly within safe blocks, rather than across the entire process, could help us achieve even lower overhead. MineSweeper retains freed allocations in a quarantine. Using linear sweeps of memory, it makes sure that no item is released from the quarantine unless it has no dangling pointers to it.

7.2 In-Process Isolation

Hardware-assisted in-process isolation techniques leverage Intel's Memory Protection Keys to segregate "safe" and "unsafe" heaps. PKRU-Safe [33] and TRust [6] both partition the address space and use PKRU permissions to enforce temporal-safety policies at page granularity. Unlike our approach, these systems do not integrate garbage collection, instead relying on explicit deallocation and fault isolation for safety. Yet they are a good example of handling two different program contexts.

Chapter 8

Conclusion

In this work, we introduced `safe blocks`, a novel mechanism for enforcing temporal memory safety in C applications by confining security-critical regions to an isolated "safe heap." Within this heap, temporal safety is provided by a mark-and-sweep garbage collector. Our prototype, implemented as an `LD_PRELOAD` library using mimalloc and enforced at runtime via MPK, demonstrates that the safe blocks abstraction is both feasible and worthy of further exploration.

Using a simple benchmark based on the `json-c` parser and two large JSON datasets, we measured the performance overhead of safe blocks across various garbage collection thresholds. Although our current prototype incurs substantial slowdowns, prior work has shown that garbage collection in C can be highly performant [1, 27]; the challenge here is primarily one of optimization.

However, our integration experiments with large real-world C projects (e.g., `nginx`, `libxml2`, `jq`) exposed significant obstacles arising from project-specific memory management conventions. These findings raise important questions about the practicality of deploying temporal safety defenses in large, mature codebases.

We have identified several key directions for future work, including concurrent and parallel execution, robust compiler passes for automatic instrumentation, and mechanisms for controlled data transfer across safe block boundaries. Addressing these areas will be essential to evolve safe blocks from a proof-of-concept into a production-ready framework capable of protecting large, multithreaded C applications with acceptable performance.

In summary, safe blocks offer a promising, targeted approach to temporal memory safety in C. By enabling developers to isolate and protect only the most critical code regions, our abstraction lays the groundwork for practical, incremental adoption of memory safe practices in memory unsafe systems, striking a balance between security and performance.

Bibliography

[1] S. Ainsworth and T. M. Jones. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 578–591, 2020.

[2] Arch-Linux. libelf package information. https://archlinux.org/packages/core/x86_64/libelf/.

[3] Arch-Linux. thunderbird package information. https://archlinux.org/packages/extra/x86_64/thunderbird.

[4] Arch Linux. fakechroot(1) manual page. https://man.archlinux.org/man/extra/fakechroot/fakechroot.1.en, 2025. Accessed: 2025-05-04.

[5] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 156–161, New York, NY, USA, 2017. Association for Computing Machinery.

[6] I. Bang, M. Kayondo, H. Moon, and Y. Paek. TRust: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, Anaheim, CA, Aug. 2023. USENIX Association.

[7] S. Blackburn, P. Cheng, and K. Mckinley. Myths and realities: The performance impact of garbage collection. *Performance Evaluation Review*, 32, 05 2004.

[8] H.-J. Boehm and D. R. Chase. 3. a proposal for garbage-collector-safe c compilation garbage collection and c. In *The Journal of C Language Translation*, 1992.

[9] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[10] A. Borisov. Lexbor: development of web browser engine. https://lexbor.com/.

[11] W. Bugden and A. Alahmar. Rust: The programming language for safety and performance, 2022.

[12] O. Chang. Racing midi messages in chrome. `https://googleprojectzero.blogspot.com/2016/02/racing-midi-messages-in-chrome.html`, 2016.

[13] Z. Chen, D. Liu, J. Xiao, and H. Wang. All use-after-free vulnerabilities are not created equal: An empirical study on their characteristics and detectability. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 623–638, New York, NY, USA, 2023. Association for Computing Machinery.

[14] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti. Pie: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC '15, page 251–260, New York, NY, USA, 2015. Association for Computing Machinery.

[15] M. Costanzo, E. Rucci, M. Naiouf, and A. D. Giusti. Performance vs programming effort between rust and c on multicore architectures: Case study in n-body. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.

[16] L. d. M. Daan Leijen, Benjamin Zorn. mimalloc: A compact general purpose allocator. Technical Report MSR-TR-2019-03, Microsoft Research, 2019. Accessed: May 3, 2025.

[17] L. K. Developers. Protection keys api. `https://docs.kernel.org/core-api/protection-keys.html`, 2025. Accessed: 2025-04-27.

[18] S. Dolan. jq. `https://jqlang.org/`.

[19] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf. Translating c to safer rust. *Proc. ACM Program. Lang.*, 5(OOPSLA), Oct. 2021.

[20] M. Erdős, S. Ainsworth, and T. M. Jones. Minesweeper: a "clean sweep" for drop-in use-after-free prevention. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 212–225, New York, NY, USA, 2022. Association for Computing Machinery.

[21] M. Foundation. Mozilla foundation security advisory 2024-51. `https://www.mozilla.org/en-US/security/advisories/mfsa2024-51/`, Oct. 2024. Accessed: 2025-04-09.

[22] FreedomIntelligence. medical-o1-reasoning-sft. https://huggingface.co/datasets/FreedomIntelligence/medical-o1-reasoning-SFT/tree/main.

[23] GCC Development Community. __builtin_frame_address. https://gcc.gnu.org/onlinedocs/gcc/Return-Address.html#index-_005f_005fbuiltin_005fframe_005faddress. [Accessed May 5, 2025].

[24] GitHub Advisory Database. Ghsa-h5g9-ghrj-76p5: Improper validation of message size in protobuf. https://github.com/protocolbuffers/protobuf/security/advisories/GHSA-h5g9-ghrj-76p5, 2024. Accessed: 2025-04-13.

[25] GNU Project. Memory protection - the gnu c library. https://www.gnu.org/software/libc/manual/html_node/Memory-Protection.html, n.d. Accessed: 2025-04-30.

[26] Google. Addresssanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed: 2025-05-04.

[27] A. J. D. Hans-J. Boehm. The boehm-demers-weiser conservative c/c++ garbage collector. https://github.com/ivmai/bdwgc.

[28] F. Indutny. llhttp - a light-weight http parser for node.js. https://llhttp.org/.

[29] Intel. Intel Core i7-1185G7 Processor (12M Cache, up to 4.80 GHz with IPU) Specifications. https://www.intel.com/content/www/us/en/products/sku/208664/intel-core-i71185g7-processor-12m-cache-up-to-4-80-ghz-with-ipu/specifications.html.

[30] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2012.

[31] jp. Advanced doug lea's malloc exploits. https://phrack.org/issues/61/6, 2003. Phrack Magazine, Issue 61, Article 6. Accessed: 2025-05-01.

[32] json c. json-c: A json library for c. https://github.com/json-c/json-c.

[33] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 132–148, New York, NY, USA, 2022. Association for Computing Machinery.

[34] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proc. of NDSS*, 2015.

[35] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan. In rust we trust: a transpiler from unsafe c to safer rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ICSE '22, page 354–355, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Linux Foundation. Elf program header — system v abi. `https://refspecs.linuxbase.org/elf/gabi4+/ch5.pheader.html`. Accessed: 2025-05-04.

[37] N. S. LLC. Ncat - netcat for the 21st century. `https://nmap.org/ncat/`.

[38] LLVM Project. Clang language extensions. `https://releases.llvm.org/11.0.0/tools/clang/docs/LanguageExtensions.html`, 2020. [Accessed May 4, 2025].

[39] Michael Kerrisk. ld.so — dynamic linker/loader. Accessed: May 4, 2025.

[40] Microsoft. mimalloc heap api reference. `https://microsoft.github.io/mimalloc/group__heap.html#gab374e206c7034e0d899fb934e4f4a863`. Accessed: 2025-05-04.

[41] Microsoft. mimalloc: A compact general purpose allocator with excellent performance, 2019. Accessed: May 3, 2025.

[42] MITRE. CWE-415: Double Free. `https://cwe.mitre.org/data/definitions/415.html`, 2024. Accessed: 2025-05-01.

[43] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2019 archive. `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`, 2019. Accessed: 2025-04-13.

[44] MITRE Corporation. Cve-2021-44710: Adobe reader vulnerability in the processing of format event. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44710`, 2021. Accessed: 2025-04-13.

[45] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2021 archive. `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`, 2021. Accessed: 2025-04-13.

[46] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2020 archive. `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`, 2022. Accessed: 2025-04-13.

[47] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2022 archive. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022. Accessed: 2025-04-13.

[48] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2023 archive. https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html, 2023. Accessed: 2025-04-13.

[49] MITRE Corporation. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html, 2024. Accessed: 2025-04-13.

[50] MITRE Corporation. Cwe top 25 known exploited vulnerabilities (kev) - 2024 archive. https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html, 2024. Accessed: 2025-04-13.

[51] Nginx. Nginx. https://nginx.org/.

[52] M. Ojeda. Rfc: Rust for linux. https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/, Apr. 2021. Accessed: 2025-04-14.

[53] openSUSE. cavil-legal-text. https://huggingface.co/datasets/openSUSE/cavil-legal-text/tree/main.

[54] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software abstraction for intel memory protection keys, 2018.

[55] G. Project. Gnu core utilities. https://www.gnu.org/software/coreutils/.

[56] G. Project. Gnu time. https://www.gnu.org/software/time/.

[57] swaywm. sway: i3-compatible wayland compositor. https://github.com/swaywm/sway?tab=readme-ov-file#compiling-from-source.

[58] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[59] TIOBE Software. Tiobe index for april 2025. https://www.tiobe.com/tiobe-index/, 2025. Accessed: 2025-04-09.

[60] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, Aug. 2019. USENIX Association.

[61] D. Veillard. libxml2. https://gitlab.gnome.org/GNOME/libxml2.

[62] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management*, pages 1–42, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[63] Windows Insider Blog Team. Announcing windows 11 insider preview build 25905. https://blogs.windows.com/windows-insider/2023/07/12/announcing-windows-11-insider-preview-build-25905/, July 2023. Accessed: 2025-04-14.

[64] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 414–425, New York, NY, USA, 2015. Association for Computing Machinery.

[65] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.