Ph.D. DISSERTATION

A Study of Memory Safety in Unsafe and Safe Languages

안전한 언어와 비안전 언어에서의
메모리 안전성에 대한 연구

BY

BANG INYOUNG

FEBRUARY 2024

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

A Study of Memory Safety in Unsafe and Safe Languages

안전한 언어와 비안전 언어에서의
메모리 안전성에 대한 연구

BY

BANG INYOUNG

FEBRUARY 2024

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

A Study of Memory Safety in Unsafe and Safe Languages

안전한 언어와 비안전 언어에서의
메모리 안전성에 대한 연구

지도교수 백 윤 흥

이 논문을 공학박사 학위논문으로 제출함

2024년 1월

서울대학교 대학원

전기·정보 공학부

방 인 영

방인영의 공학박사 학위 논문을 인준함

2024년 1월

Abstract

The demand for software security is greater than ever. Nevertheless, effectively mitigating attacks on memory corruption vulnerabilities in intricate software systems remains an unresolved challenge. It is because a large number of legacy codebases are written in memory unsafe languages such as C/C++. Researchers have strived to mitigate memory safety issues for decades, but memory corruption is still the major vector of security attack. This thesis proposes a method to enforce temporal safety in unsafe language. Including the proposed method, multiple exploit mitigation techniques have been suggested, but software-based mitigation schemes pose a significant overhead and hardware-based solutions have limited applicability. In this context, Rust, a memory safe language has been introduced with a focus on memory safety and efficiency. Thanks to memory policy such as strict type system and ownership model, memory safety is enforced in language level, while maintaining efficiency which is better than security optimized C/C++. However, it has been proved that there's still a loophole in the seemingly safe Rust. To resolve this breach, the thesis presents a novel framework against unsafe compartment present in the program, providing trustworthy protection via in-process isolation. Still, attack surface of the program is left as it is, the thesis concludes by exploring ways to solve the problem.

keywords: programming language, memory safety, security, compiler, isolation
student number: 2017-20125

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The demand for software security is greater than ever as digital technologies increase reliance on electronic health records and financial information. Also, the widespread adoption of self-driving cars also escalates the importance of security issues. Nevertheless, effectively mitigating attacks on memory corruption vulnerabilities in intricate software systems remains an unresolved challenge. It is because a large number of legacy codebases are written in memory unsafe languages such as C/C++. Researchers have struggled to mitigate memory safety issues for decades, but memory corruption is still the major vector of security attack. According to Mitre [4], memory safety issues still rank in the TOP 25, as out-of-bounds write and read rank first and seventh respectively, and use after free ranks fourth.

Memory safety consists of spatial safety and temporal safety. First, spatial safety is violated by out-of-bounds read or write, as pointers are dereferenced without any check in C/C++. Including the author, a number of researchers have strived to resolve this issue [5–10]. Secondly, the representative example of the violation of temporal safety is use-after-free. In C/C++, allocation and revocation of memory are left entirely to a programmer. As there's no binding between pointer and pointee, de-allocated region can still be dereferenced. Numerous mitigation techniques have been suggested to tackle the issue [6, 11–18].

As one of the methods to ensure temporal safety in C/C++, `Vatalloc` is presented in §2. Using disposable locks and keys, which are assigned respectively to objects and pointers, every memory accesses are checked to prevent use-after-free attacks. With the support of the *Memory Tagging Architecture* (MTA) introduced in several commodity processors, VA consumption rate is drastically lowered by one order of magnitude, while showing minimal memory overhead and reasonable performance degradation. Including `Vatalloc`, multiple exploit mitigation techniques have been suggested so far, but software-based mitigation schemes pose a significant overhead and hardware based solutions have limited applicability. In this context, Rust, a memory safe language has been introduced by Mozilla with a focus on memory safety and efficiency. Thanks to memory policies such as strict type system and ownership model, memory safety is guaranteed with compile time and runtime checks in language level, while maintaining efficiency which is better than security optimized C/C++. Despite this advantage in security, the restrictions imposed by Rust's strict policies make it difficult or inefficient to express certain designs or computations. To ease or simplify their programming, developers thus often include *untrusted code* from unsafe Rust or external libraries written in other languages. Sadly, the programming practices embracing such untrusted code for flexibility or efficiency subvert the strong safety guarantees by Rust.

In §3, the thesis shows how untrusted code compromises safety in Rust. The thesis presents TRUST, a compilation framework which provides trustworthy protection via in-process isolation against untrusted code present in the program. Its main strategy is allocating objects in an isolated memory region that is accessible to safe Rust but restricted from being written by the untrusted. To enforce this, TRUST employs software fault isolation and x86 protection keys. It can be applied directly to any Rust code without requiring manual changes. Our experiments reveal that TRUST is effective and efficient, incurring runtime overhead of only 7.55% and memory overhead of 13.30% on average when running 11 widely used crates in Rust.

Chapter 2

Enhancing a Lock-and-Key Scheme With MTE to Mitigate Use-After-Frees

2.1 Introduction

Dangling pointers refer to the pointers whose referent objects have been freed. These pointers are common in C/C++, and they themselves do not pose a threat to a program. However, dereferencing dangling pointers introduces susceptibilities to *Use-After-Free* (UAF) bugs, which may lead to arbitrary memory access and control flow hijacking. UAFs are prevalent across applications, as demonstrated in the statistical report of the MITRE where it ranks among the top 25 most dangerous software errors [19]. To date, a lot of techniques [11, 13–18, 20–28] have been invented to stymie UAF attacks in question.

Conventional techniques for preventing UAF bugs can be broadly classified into two approaches: *lock-and-key* and *pointer nullification*. The former has long been proven effective by many studies [11, 15, 26]. In this approach, UAF bugs are prevented by validating every memory access. To be specific, *locks* and *keys* are first assigned to objects and pointers respectively. Every pointer dereference is validated by checking that the key held by a pointer matches the lock held by its referent object. To prevent UAF attacks that exploit dangling pointers, locks are invalidated as soon as the corresponding objects are freed, rendering the keys of associated dangling pointers outdated. In contrast, the latter relies on compiler instrumentation [13, 14, 18, 20] or garbage collection like solutions [16, 17, 24, 25]. Compiler based solutions require source code, which hampers its general adoption in practice. Some of GC-based techniques necessitate hardware modifications [17, 24] or entail runtime overhead, despite being disguised by concurrency and generous provisioning of compute and memory resources [29].

Earlier studies [11] assigned unique integers (typically 64-bit) to represent locks and keys. Unfortunately, the integer-based representation has a critical downside that it requires heavy data structures to manage the locks and keys associated with objects and pointers. Techniques that use this representation tend to suffer from high runtime overhead, as they have to perform frequent load/store and matching operations for each lock and key assigned during program execution. As explained in §2.8, due in part to the high computational cost associated with locks and keys, the research community has largely shifted towards the pointer nullification approach stated above [14].

Most recently, however, Oscar [15], Dangzero [28] and FFmalloc [27] proposed an innovative idea that can reduce the computing cost of the lock-and-key approach, rendering it more attractive to use. Whereas traditional lock-and-key-based techniques have been plagued by excessive runtime overhead resulting from managing and computing integer locks/keys, the more recent approaches have moved towards using *virtual addresses* (VAs) of objects/pointers to represent locks/keys. In this scheme, unique, disposable locks are implicitly generated throughout program execution by assigning different VAs to all objects created in the program in their VA-based representation. When an object is freed, the VAs assigned to it are invalidated. In so doing, dereferencing the dangling pointers that hold the invalidated VAs of the freed object is trapped by a memory fault, leading to the prevention of UAFs. However, the VA-based scheme comes with an inherent drawback. The scheme requires that each VA is allocated to objects only once to ensure the uniqueness of locks, which negatively impacts the system and VAs will be run out rapidly. As VAs should be discarded after a single use, caching mechanisms of memory allocators to efficiently reuse freed memory for near future allocation requests becomes inapplicable. Also, it intensifies memory fragmentation over time, ultimately causing performance and memory overheads.

In this paper, we propose a lock-and-key scheme, called the *VA tagging*, which effectively resolves scalability issues associated with the current VA-based prevention scheme. Our scheme significantly reduces VA exhaustion while maintaining efficient performance and minimal memory overhead. The scheme leverages the *Memory Tagging Extension* (MTE), a hardware feature that has been announced in the latest line of ARM processors, ARMv8.5-A [30]. VA tagging scheme augments the original VA-based locks with *tags*, and memory accesses are allowed only when their tags match with the aid of MTE. Specifically, during the allocation of a new object, the object is assigned a VA along with a tag number, and the associated pointer is also assigned the same tag. When an object is freed, we do not simply invalidate its VAs as in the original VA-based schemes; rather, we modify its tag numbers. This alteration prevents UAF attacks in subsequent dereferences of associated dangling pointers, due to the disparity between the tag numbers assigned to the freed objects and the dangling pointers. Unlike the conventional VA based techniques, each VA can be assigned to objects as many times as the number of available MTE tags, without losing the prevention capability against UAF attacks. By doing so, our scheme effectively alleviates the aforementioned issues caused by single-use only VAs.

To evaluate the feasibility and effectiveness of our VA tagging scheme, we have implemented a technique, called *Vatalloc*, that provides light-weight prevention of UAF attacks. It is noteworthy that our VA tagging scheme can be seamlessly incorportated into any allocators. We have demonstrated this by implementing Vatalloc in two popular allocators, namely *dlmalloc* and *jemalloc*, which have distinct design philosophies. With only 0.9K and 0.2K additional LoC respectively, the VA tagging scheme is successfully applied to both allocators.

The empirical results with SPEC2006 [31] and PARSEC [32] benchmark suites clearly demonstrate the Vatalloc can mitigate UAF attacks with high efficiency in terms of time and space. In short, Vatalloc with allocator modifications only incurs 1.70 % (on dlmalloc) and 3.05 % (on jemalloc) of runtime overhead, and 19.0 % (on dlmalloc) and 3.0 % (on jemalloc) of memory overhead. To measure an accurate estimation of worst case performance of MTE, we simulated both tag update and tag matching. Vatalloc on dlmalloc and jemalloc resulted in 16.9 % and 12.0 % slowdown, with 19.0 % and 3.0 % memory overhead respectively.

In summary, our contributions are:

- We propose a VA tagging scheme with MTE enables light-weight and scalable prevention of UAF attacks without source code of a target program.
- We have experimentally showed that Vatalloc can be implemented on existing memory allocators with minimal modification.
- We performed reasonable proxy measurements to evaluate worst-case performance of Vatalloc despite the absence of real hardware or cycle-accurate simulator. We found that most of the overhead incurred is due only to tag matching on memory accesses, and Vatalloc induces insignificant degradation on performance and memory.

2.2 Background

In this section, we provide background information on MTE, the core hardware feature leveraged by Vatalloc, and two major dynamic memory allocators, dlmalloc and jemalloc, to which we applied Vatalloc for prototype implementation.

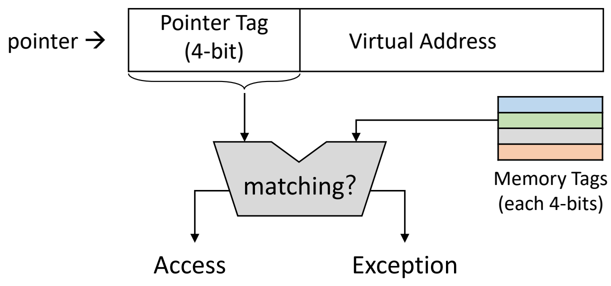

Figure 2.1: Operation of MTE.

2.2.1 Memory Tagging Extension (MTE)

MTE follows the design of tagged memory architecture [33], which aims to efficiently improve the security of the system. This hardware feature has been introduced on the recent ARMv8.5-A [34] architecture. As described in Figure 2.1, MTE deploys two types of tags: *pointer tags* and *memory tags* that are assigned to each pointer and each 16 B (or 32 B) memory block. Pointer tags are attached to the top bits of pointers and *memory tags* that are stored in a separate memory storage. The pointer tag is implicitly propagated through pointer arithmetic to other pointers that equally refer to the object, thus all the associated pointers can be used to access the object. Each tag is 4 bits large, therefore, its value ranges from 0 to 15. When enabled, MTE checks every memory access by comparing the tag numbers of the pointer and target memory. If a mismatch occurs, MTE raises an exception synchronously or asynchronously by configuration. Pointer tags can be easily extracted from the value of pointers, and similarly memory tags can be loaded with minimal latency by preparing a dedicated cache memory for them or placing them inside ECC bits beside the associated memory blocks.

MTE provides two operation modes: precise and imprecise. The former introduces performance overhead, but supports synchronous exception providing the faulting address, which means that a tag mismatch is identified instantly upon an illegal memory access. The latter operates fast, but exception is raised asynchronously, meaning a tag mismatch is reported later after an illegal memory access. In regards to this imprecise mode, ARM features an option to release the delayed notification upon entering the kernel so that the kernel can recognize the occurrence of any tag mismatch. To enable programs to be informed of the tag mismatch exception from MTE, a recent Linux kernel [?] with a MTE support generates a `SIGSEGV` signal when a tag mismatch occurs. The signal is passed to programs along with a code, `SEGV_MTESERR` (i.e., synchronous error) at the precise mode or `SEGV_MTEAERR` (i.e., asynchronous error) at the imprecise mode. Therefore, it is possible that a program deal with tag mismatch of MTE by writing its own signal handler.

2.2.2 dlmalloc

dlmalloc [36] is a former primary dynamic memory allocator of Linux and Android, and the base of ptmalloc [37] currently used in Linux by default. Upon receipt of a memory allocation request, dlmalloc internally captures a memory block, called a *chunk*, and returns its address. Chunks are allocated from a pool, called a *segment*, allocated by the kernel through `srbk` and `mmap` system calls. Chunks are classified by size into *small* and *large* groups, based on a defined threshold (by default, 256 KB). Such a classification reflects the common usage pattern for objects that varies by size: generally, small objects are frequently allocated and have a short life-time, but large objects are rarely allocated and have a long life-time. Therefore, for a proper control of allocation overheads, dlmalloc manages these two types of chunks in different methods. Allocation/deallocation requests for large chunks are page-aligned and directly handled by *mmap* and *munmap*, which implies that dlmalloc does not reuse large chunks as its memory pool. On the other hand, the requests for small chunks are managed for effective reuse. On allocation, a request size is aligned to a multiple of 16 (or 32) bytes on 64-bit architecture. Dlmalloc then searches with a best-fit strategy a *free list (or tree* in which freed small chunks have been collected. Absence of a fitting free chunk results in dlmalloc splitting the top chunk that is the topmost free chunk in each segment as the final resort. On deallocation, dlmalloc consolidates (or coalesces) the just freed chunk with adjacent freed chunks to suppress external fragmentation. To facilitate such management, dlmalloc augments each chunk with a 16-byte sized *header* that consists of several fields as described in Figure 2.3. In addition, dlmalloc inserts node metadata into freed chunks to maintain the free list without consuming additional memory.

2.2.3 jemalloc

Jemalloc [38] is being used in Mozilla Firefox for the Windows, Mac OS X and Linux platforms, and was adopted as the default system allocator on the FreeBSD and NetBSD operating systems. For better multithreading support, Jemalloc relies on the concept of *arenas*, where a specific arena is associated with particular execution threads to overcome lock contention issues between threads. The arena serves the required concurrency of the program and its number is proportional to the number of available CPU cores. Jemalloc categorizes the size of malloc requests into small, large and huge. Requests less than 4KB are classified as *small*, those less than 2MB as *large*, and the rest as *huge*. For small and large allocations, bins are used to organize size classes of chunks, and, in our case, for example, 35 size classes are initialized a priori for each small and large allocations. Each bin determines the size and the capacity of chunks in the associated *runs* consisting of one or multiple pages. The fundamental purpose of a run is to keep track of the allocation state of chunks. It holds an indexed bitmap as a part of its metadata, specifying whether the chunk is freed or in-use. When the malloc request for small or large is placed, a chunk in a run corresponding to the request size is returned to user. In contrast to dlmalloc, jemalloc does not release memory even for huge chunks after deallocation. Instead, it reserves it for later use. For this purpose, jemalloc organizes a separate tree data structure to manage huge chunks.

2.3 Threat Model and Prerequisite

Threat Model: We assume the threat model that is consistent with that of the previous work discussed in §2.8. A target program is not malicious per se, but has dangling pointers. In this work, we assume an adversary can only launch UAF attacks by exploiting the dangling pointers. Defence techniques [5, 39, 40] that are orthogonal and compatible to Vatalloc are assumed to prevent different types of memory attacks such as buffer overflow and type confusion so that the adversary cannot exploit them to subvert or bypass Vatalloc by compromising the integrity of its metadata and MTE's memory/pointer tags.

Prerequisite: Vatalloc is designed to operate at memory allocator-level. To apply Vatalloc, therefore, a target program is required to use explicit interfaces for memory management, such as allocation, deallocation, and reallocation.

2.4 VA Tagging Scheme

We propose drop-in-use VA tagging scheme for efficient prevention of UAF attacks. As described in §2.1, the lock-and-key approach prevents UAF attacks from a mismatch between the lock of an object and the key of a pointer. The performance overhead in this approach is attributed to the expenses incurred in generating unique, disposable locks upon object allocation, executing lock/key matching during memory access, and invalidating these locks upon object deallocation. In this context, our VA tagging scheme significantly reduces the costs by cleverly leveraging both VAs and MTE tags as locks, which leads to a reduction in the assignment and management costs. Our proposed VA tagging scheme leverages MTE tags to allow the reuse of the same VA multiple times. It effectively alleviates heavy kernel intervention for lock management, as opposed to the original VA-based scheme [15].

Figure 2.2 illustrates how the VA tagging scheme effectively prevents UAF attacks. Initially, when an object is allocated by `malloc`, the random tag number is assigned

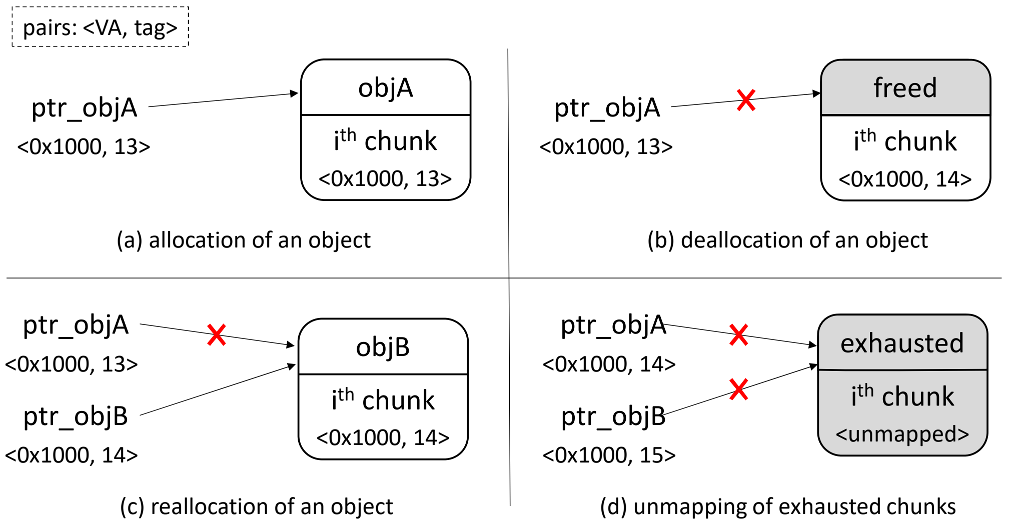

Figure 2.2: UAF prevention based on the VA Tagging Scheme. For the sake of simplicity, we assume that the tag numbers are initially assigned 0 and subsequently increase until reaching 15.

to both the memory chunk of the object and the returned pointer (Figure 2.2.(a)). As explained in §2.2.1, the pointer tag attached to the returned pointer is implicitly propagated through pointer arithmetic to other pointers that equally refer to the object, resulting in no extra overhead required to manage it. The key point of our scheme is the incrementation of the tag number assigned to an object upon deallocation. This tag modification renders subsequent attempts to dereference the dangling pointers for UAF attacks be caught due to the tag mismatch (Figure 2.2.(b-c)).

In our VA tagging scheme, each tag number must be used only once for the same chunk for safety. Therefore, we mark the chunks with no assignable tag numbers as "*exhausted*" to prevent them from being reused later. Unfortunately, these exhausted chunks will lead to memory leak by taking up physical page frames. To address this problem, we need to keep track of the chunks in order to retrieve their frames by unmapping them. However, we should note that most exhausted chunks will be mixed with non-exhausted chunks within a page. Therefore, to avoid a faulty unmapping, we have to keep track of all exhausted chunks by page, and only unmap the page which is completely filled with exhausted chunks. Once a page is unmapped, dangling pointers referring to somewhere in it become invalidated so that UAF attacks by exploiting them will be prevented (Figure 2.2.(d)).

2.5 Vatalloc

For convenience, we name two versions of Vatalloc implemented based on dlmalloc and jemalloc as *Vatalloc-d* and *Vatalloc-j*, respectively. To demonstrate the effectiveness and feasibility of the VA tagging scheme, we implement Vatalloc that realizes the scheme on two real memory allocators, dlmalloc and jemalloc. The two versions of Vatalloc are designed to follow the original design philosophy found in the base memory allocators. For example, dlmalloc, a free-list allocator, is designed to minimize memory overhead, such as by placing its metadata within free chunks. Therefore, Vatalloc-d is designed to reduce memory consumption by devising a relatively complicated but space efficient data structure for its metadata. On the other hand, jemalloc, a bucket allocator, was designed to improve performance in exchange for consuming more memory. Accordingly, Vatalloc-j is designed to reduce performance overhead by using fast but not space-optimized data structure for its metadata.

Vatalloc basically fits into C and C++, but is applicable to other languages that provide dynamic memory allocation interfaces as a form of operators or functions, such as `new` and `delete`, or `malloc` and `free`. Vatalloc is employable in applications either by overriding these functions at run time, or linking against a library containing Vatalloc's substituted functions at loading time.

2.5.1 Tag management

Small chunks (chunks hereafter for short) are prepared for small sized objects. To take advantage of the VA tagging scheme, 4-bit MTE tag numbers must be assigned to the

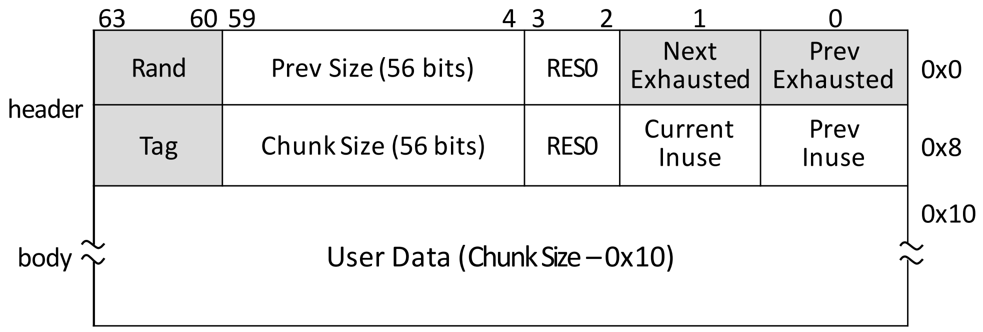

Figure 2.3: Modified Chunk Metadata on Vatalloc-d. The gray field refers to a free bits used as metadata.

chunks and updated whenever there are state transitions of chunks: *creation*, *free*, and *consolidation*.

Upon spawning new chunks from the heap, Vatalloc assigns a tag number to each. Subsequently, the tag numbers are updated as follows: upon receiving free requests, the corresponding chunk remains to service later allocation requests efficiently and its tag number is incremented. Concurrently, Vatalloc increases the tag number of the chunk by 1. This prevents dereferencing of dangling pointers that still hold an unchanged pointer tag that no longer matches, thus preventing UAF attacks.

On dlmalloc: To implement Vatalloc-d, tag count for each chunk, which refers to how many times the chunk is reused, is required to be stored and tracked as Vatalloc assigns random value to each chunk. It is stored in the chunk header, specifically in the topmost 4-bit of the `chunk_size` field, which is the second byte of the header. Figure 2.3 shows the modified chunk's header. By doing so, Vatalloc cleverly avoids performance overhead by tag loading and memory consumption for tag management. Since the `chunk_size` field must be accessed upon the state transitions of chunks, Vatalloc minimizes memory operations for accessing tag count in this way. This placement reduces the `chunk_size` bit from 60 to 56, which is acceptable since no objects are bigger than 2^{48} bytes. To evenly distribute tag numbers, initial value is randomized and stored in the topmost 4-bit of the `prev_size` field, which is the first byte of the header.

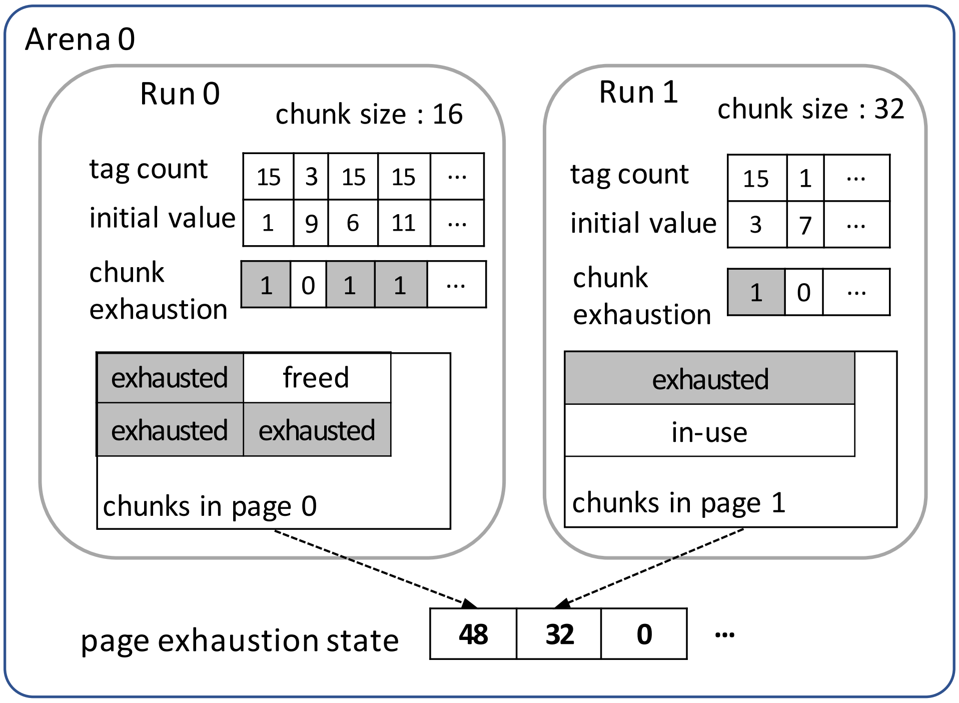

Figure 2.4: Metadata of Vatalloc-j.

The actual value of the tag number is computed by adding the tag count and the initial tag together.

When consolidating adjacent free chunks to minimize fragmentation, tag management becomes crucial. In this process, Dlmalloc checks if there are other free chunks adjacent to the current one, and if so, it consolidate them into one. However, if these chunks have different states, which are initial values and tag counts, Vatalloc recomputes the tag counts and initial value for consolidation. First, the minimal initial value of the two chunks is assigned as the new initial value. Second, the maximum tag count of the two chunks is calculated and delta of the two initial values is added to it and assigned as the new tag count. Although this method preserves tag number consistency, it can lead to faster growth of tag counts, reducing the likelihood of reusing the same virtual addresses. To address this concern, Vatalloc can enforce a consolidation policy based on the recomputed tag count if two adjacent chunks are consolidated. This policy allows consolidation only if the degree of the tag count is equal to or less than a defined threshold, and otherwise, Vatalloc leaves the chunks separate until their tag numbers become close. By default, Vatalloc sets the threshold to 15, which permits any consolidation. We evaluate the performance of Vatalloc for different thresholds in Section 2.6.

On jemalloc: For proper tag management on jemalloc, we changed the configurable tiniest chunk size of jemalloc from 8 to 16 bytes, as memory can only be tagged with 16-bytes granularity in MTE. In jemalloc, chunks are preallocated in a run, an allocation pool, with segregated by their size, and are not coalesced unlike dlmalloc. Therefore, as illustrated in Figure 2.4, Vatalloc manages tag counts and initial vales by using a per-run tag array whose elements correspond to the tag counts and randomized initial value of each chunk.

2.5.2 Exhausted Chunks

As chunks are reused repeatedly, their tag counts gradually increase as in §2.5.1. When their tag counts reach the maximum value (i.e., 15) and cannot be incremented on deallocation anymore, Vatalloc label them as exhausted and excludes from the allocation pools not to be considered in a future reallocation. This exclusion is essential for Vatalloc to maintain its UAF attack prevention capability by preventing multiple objects from having the same VA and tag number. However, it may increase memory overhead because the exhausted chunks will keep occupying physical page frames. Vatalloc relieves this issue by unmapping the pages where exhausted chunks are residing, which will be explained in §2.5.3.

On dlmalloc: To mark the chunk's exhaustion state, Vatalloc-d modifies chunk's header again similar to the case of the tag number. However, note that, as described in Figure 2.3, Vatalloc-d does not mark the exhaustion state of each chunk in its own header. Instead, a chunk's exhaustion state is indicated in the header of the previous and next chunks. This is to avoid a segmentation fault that may occur during a consolidation process. For example, when `chunkA`'s object is freed, dlmalloc needs to visit `chunkA`'s adjacent chunks and identify their states to make a consolidation. However, if any adjacent chunk was exhausted and the page to which this chunk belongs has been unmapped, a segmentation fault will be caused when visiting it. Therefore, Vatalloc preemptively fends off this problem by checking whether or not adjacent chunks have been exhausted before visiting them. To do this, Vatalloc creates and refers to a pair of state bits in the header: `prev_exhausted` and `next_exhausted`. When a chunk is exhausted, Vatalloc sets these state bits from the previous/next chunks.

On jemalloc: As in the tag numbers, Vatalloc-j manages chunk's exhaustion state using an array within a run, which is illustrated in Figure 2.4. Since the per-run array is completely separated from chunks, each element of the array directly presents the exhaustion state of the associated chunk without worrying about a segmentation fault from accessing unmapped chunks unlike the case of dlmalloc. In jemalloc, freed chunks are cached for faster reallocation in the future. Therefore, Vatalloc modifies the jemalloc not to cache exhausted chunks.

2.5.3 Page-level Exhaustion Tracking

Vatalloc needs to release page frames consisting of exhausted chunks for efficient memory use. To do this, Vatalloc unmaps the pages where exhausted chunks are residing. However, we should note that those exhausted chunks are likely to be mixed with non-exhausted chunks within the same pages. Therefore, careless unmapping of the pages may result in a crash. To prevent this problem, Vatalloc defines a new type of page, called an *exhausted page*, that is full of only exhausted chunks. Vatalloc can unmaps only this type of pages safely, as they do not contain non-exhausted chunks.

The challenging issue here is that it is not feasible to artificially group scattered exhausted chunks into exhausted pages since it will break the consistency of VA between chunks and pointer. When a chunk becomes an exhausted one, Vatalloc checks

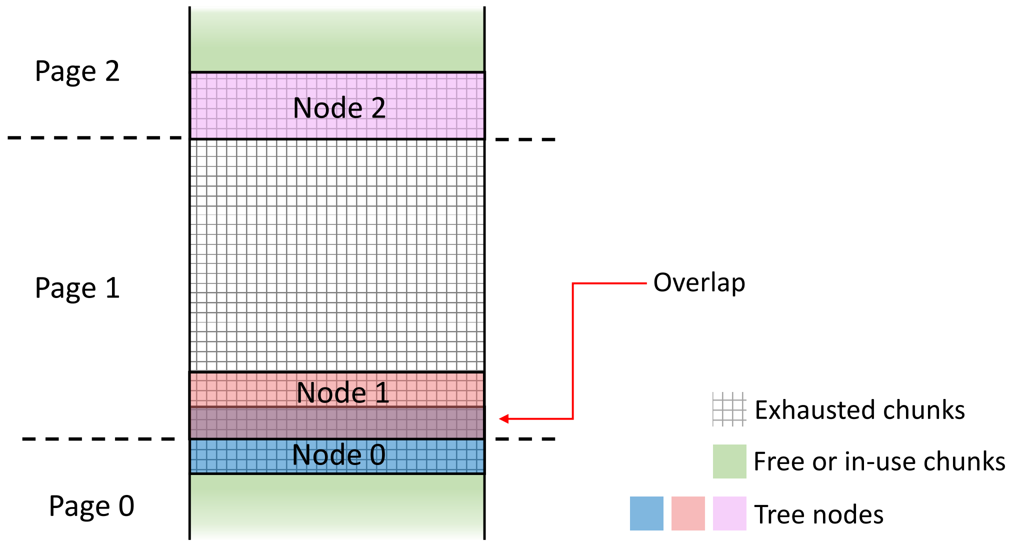

Figure 2.5: A Problematic Placement of Page Exhaustion State Tree Nodes on Vatalloc-d.

whether the page containing the chunk turns to be exhausted ones, and if so, instantly unmaps it to release the occupied physical page frame. To track down the generation of exhausted pages, Vatalloc maintains metadata, called *page exhaustion state*. Vatalloc keeps recording the accumulated size of exhausted chunks by page in this metadata. Using this metadata, Vatalloc can identify exhausted pages by finding the ones whose accumulated size of exhausted chunks are equal to the page size (i.e., 4096 with a 4 KB-paging scheme).

In the description so far, we assumed that Vatalloc performs page exhaustion tracking by each page. This default granularity allows Vatalloc to decrease memory overheads the most by instantly unmapping exhausted pages as soon as generated. However, a little performance degradation will be caused due to frequent page management (i.e., calling the `munmap` system call). Alternatively, we can increase the granularity by performing page exhaustion tracking by a group of pages. In this modified granularity, we may expect Vatalloc to consume more memory on exhausted pages due to a delayed unmapping. Instead, the performance will be improved by decreasing the frequency of

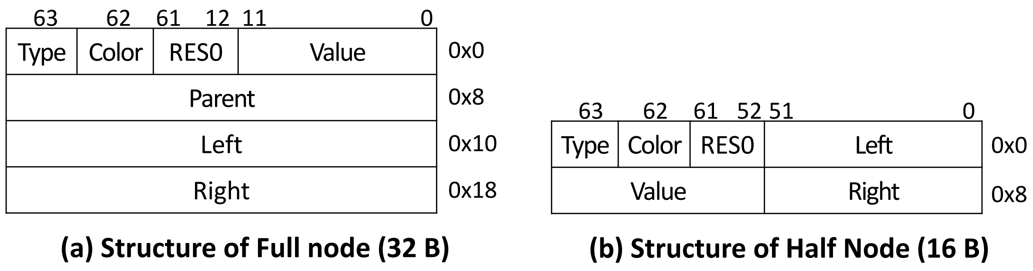

Figure 2.6: Two Types of Nodes of Page Exhaustion State Tree on Vatalloc-d.

page management. We will discuss this trade-off of the tacking granularity in §2.6.

On dlmalloc: As stated earlier in this section, Vatalloc based on dlmalloc seeks to minimize the memory overhead for metadata. For this purpose, we maintain page exhaustion states as a binary search tree because each tree node can be stored elaborately in exhausted chunks without a separate allocation, as will be explained below. In the tree, each node corresponds to a page: `key` is a virtual page number (i.e., VA[48:12]) and *value* is the accumulated size of exhausted chunks. We should note that since the heap grows in one direction, the binary search tree will be right skewed with new nodes possibly having incremental keys, which will be slowed down to linear complexity search time. Vatalloc avoids this problem by using a red-black tree. As described in Figure 2.6.(a) the node of the tree consists of `color`, `parent/left/right` pointers, and `value`. `key` needs not be stored in the node because it's value (i.e., page number) can be easily computed with a shift operation from the address of the node.

As stated ealier, Vatalloc creates tree nodes in exhausted chunks without a separate memory allocation. However, exhausted chunks for tree nodes are selected carefully, given they are eventually unmapped along with the exhausted pages to which they are belong. Therefore, Vatalloc basically creates a node of each page at the beginning bytes of the first exhausted chunk within the page. If the exhausted chunk is larger than a page, Vatalloc simply creates multiple nodes by page boundary. For example, in Figure 2.5, Vatalloc creates three nodes inside exhausted chunks for page 0, 1, and 2. In most cases, a node whose encoding is depicted in Figure 2.6.(a) can be placed in each chunk whose minimum size is 32-byte[1] in dlmalloc. However, we should consider a corner case described in Figure 2.5 as well. Since in dlmalloc chunks are 16-byte aligned, chunks (even the smallest) can span the page boundaries. The problem is that a node of Figure 2.6.(a) is larger than the alignment unit of a chunk. Therefore, in Figure 2.5 the node of the page 0 is expected to overflow to the page 1. This causes two problems. First, the node of the page 1 will be corrupted by the node of the page 0. Second, if the page 1 is unmapped, both nodes of page 0 will be damaged.

To avoid these problems, Vatalloc devises an auxiliary encoding described in Figure 2.6.(b). With this encoding, the size of nodes decreases to 16-byte that is identical to the alignment unit of a chunk. We call these size-reduced nodes *half nodes*, and original nodes *full nodes*. These two types of nodes are distinguished by the `type` bit. In a half node, we set the length of the `right` and `left` to 52 bits, which is enough considering that (1) the size of the entire VA space is usually 48 bits with 4-level paging, and (2) chunks are well aligned at 16-byte boundaries (i.e., we can cut off the bottom 4-bit in `right` and `left`). Furthermore, the size of the `value` is 12 bits, enough to cover larger granularities of the page-level exhaustion tracking considering that chunks are allocated in units of 16-byte (i.e., we can cut off the bottom 4-bit in `value`). Lastly, we omit the `parent` in a half node due to the limited size. It makes sense because unlike the `right` and `left`, the `parent` can be re obtained anytime by traversing from the root node of the tree. However, this would slow down the tree management. Vatalloc alleviates this problem by gradually converting half nodes to full nodes. It is obvious that, in the page where a half node already exists, another exhausted chunk that is large enough to store a full node would be generated soon. In this case, Vatalloc migrates the half node to the new exhausted chunk, while converting it to a full node.

On jemalloc: Jemalloc organizes allocation pools for chunks with memory blocks of 2 MB. To keep track of the page exhaustion state, Vatalloc-j creates an array indicating page exhaustion state by each memory block as illustrated in Figure 2.4. For example,

[1]header (16-byte) + body (16-byte)

if the default granularity (i.e., a page size) is used in tracking, each element of the array corresponds to a page and stores the accumulated size of exhausted chunks on the page. Once the value of the array element reaches the granularity so that the associated page is exhausted, Vatalloc unmaps the corresponding memory.

2.5.4 Large Chunks

As stated in§2.2, dlmalloc and jemalloc handle large-sized chunks in their own way. For example, dlmalloc does not reuse large chunks. On the other hand, jemalloc aggressively reuses even large chunks. Accordingly, Vatalloc applies the VA tagging scheme differently to each allocator.

On dlmalloc: dlmalloc is designed to acquire memory for large chunks from the kernel on the fly, instantly unmapping deallocated large chunks. To abide by it, Vatalloc does not reuse VAs of large chunks, but simply places large chunks at different VAs. Since dlmalloc rely on `mmap` for large chunk allocations, Vatalloc can implement it by making `mmap` allocate large chunks in a way of monotonously increasing VA. More specifically, Vatalloc (1) defines a `break_large`, which corresponds to the `break` of the heap that is managed by `sbrk`, (2) gives it to `mmap` along with the `MAP_FIXED_NOREPLACE` flag to allocate a new large chunk in the VA of the `break_large` value, and (3) adds the just allocated size to the `break_large` value for moved allocation of next large chunks. However, in this implementation, a large chunk allocation can sometimes fail because the `break_large` value may collide with the existing memory allocations, such as for libraries, on different parts of the program other than Vatalloc. In this case, dlmalloc resorts to a trial-and-error method that repeatedly adjusts the `break_large` value until `mmap` succeeds to allocate a large chunk. Admittedly, this method may cause a poor worst-case overhead. To alleviate this problem, Vatalloc introduces a memory layout that is inspired from the conventional disposal of the heap and stack to minimize the likelihood of a collision. As illustrated in Figure 2.7, the address spaces of the small chunks and the large chunks grow in the opposite direction. To sum up, the address

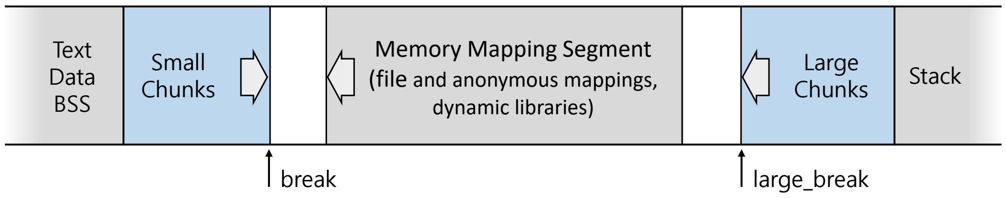

Figure 2.7: The memory layout of a Vatalloc-enabled program.

space of the large chunks is managed by `mmap` with the ever-decreasing `break_large` and that of the small chunks by `sbrk` with the ever-increasing `break`.

On jemalloc: Jemalloc organizes separate data structures and algorithms for reusing large chunks. Therefore, Vatalloc-j applies the VA tagging scheme so that the large chunks are reused unlike the case in dlmalloc. Exactly speaking, as stated in §2.2.3, jemalloc classifies non-small chunks again into large and huge chunks. Firstly, for large chunks, as jemalloc manages them no differently from the small ones, so that Vatalloc applies the VA tagging scheme in the way described in the previous subsections. On the other hand, jemalloc organizes single tree data structure for each huge chunks, since huge allocations rarely happen. Vatalloc carries out tag operation directly accessing internal tree structure, and unmaps huge chunk on deallocation if it is exhausted.

2.5.5 Multithreading Support

As Vatalloc complies with the original design direction of the base memory allocators, it can provide multithreading support equivalent to them.

On dlmalloc: dlmalloc is originally designed to suit single thread programs, and its metadata are shared among threads. Therefore, dlmalloc features simplistic multithreading support by maintaining a global mutex lock for protecting its metadata. Similarly, Vatalloc-d can support multithreading by protecting its metadata using a global mutex lock. Since metadata such as tag numbers and exhaustion states of chunks stored in chunk headers are naturally protected by the existing lock of dlmalloc, only one lock is additionally needed to protect the page exhaustion state.

On jemalloc: jemalloc has a specialized design for efficient multithreading support that minimizes the need of a global mutex lock by maintaining per-thread metadata. Likewise, Vatalloc places all the metadata alongside the existing data structure of jemalloc, and thus providing efficient multithreading support without any addition of a global mutex lock.

2.5.6 Operation Modes and Detection

MTE provides two operation modes: precise and imprecise, as described in §2.2.1. Depending on the currently activated MTE mode, Vatalloc can prevent Use-After-Free (UAF) attacks either immediately (in the precise mode) or lazily (in the imprecise mode). As explained in §2.2.1, in precise mode, the tag mismatch is synchronously notified with the faulting address, which allows for the exact determination of the load or store instruction that caused the tag mismatch. When a tag mismatch occurs, the kernel support MTE to generate a `SIGSEGV` signal. To be specific, the signal is raised instantly if the precise mode is enabled. The signal is also accompanied by a code, `SEGV_-MTESERR` or `SEGV_MTEAERR`, to specify the current MTE mode. Therefore, Vatalloc can write a custom `SIGSEGV` handler to detect UAF attacks by catching the signal and code. This feature allows Vatalloc to instantly detect UAF violations before any dangling pointer is exploited by attackers. Thanks to this MTE tag matching, Vatalloc has a relative advantage over some other Virtual Address (VA)-based techniques, such as FFmalloc, in that it has the capability for instant detection, which they lack. On the other hand, in the situation where high performance is required, imprecise mode can be chosen. In this case, signal of violation is delayed and raised at the upcoming entry to the kernel. In the event of a violation, the signal of the violation is postponed and raised upon the next entry into the kernel. However, it is unlikely that a violation would lead to a serious compromise of the system, as it is detected before it enters the kernel.

2.5.7 Against VA exhaustion

Despite Vatalloc's efficient use of VAs and tags, an application that never finishes may end up exhausting all the VA space. For that case, like conventional GC, procedure of `marking` the pair of VA and tag which is still referenced, and `sweeping` the pair which is unmarked for safe reuse can be considered. DangZero has shown this reclaiming routine effective with the modified kernel. Like Oscar, DangZero requires at least one page frame per object. Since Vatalloc does not have this constraint and the same VA can be reused multiple times, the frequency of reclaiming and the resulting performance degradation is expected to be significantly lower.

2.6 Evaluation

In this section, we demonstrate the efficiency and effectiveness of Vatalloc. We first evaluate two versions of Vatalloc implemented on dlmalloc and jemalloc in terms of performance and memory through a comparison with previous techniques. We also show how the design configurations declared in §2.5 affect Vatalloc. Lastly, we examine the effectiveness of Vatalloc in UAF attack detection. In the following evaluation, we evaluate performance and memory overhead of Vatalloc operating in precise and imprecise mode using emulation.

2.6.1 Experimental Setup

To confirm the functional correctness, the implementation has been carried out on ARM Fixed Virtual Platform v11.11.34. However, as it is not cycle accurate, the implementation on it cannot be used for performance measurement. To our knowledge at the time of this research, there were no publicly available processor supporting MTE. For example, according to our investigation, even the Apple M2, which is designed with up-to-date ARM architecture did not implement MTE. Therefore, we instead conducted proxy measurements on development board, ODROID-HC4 [41] with 1.8Ghz Cortex-

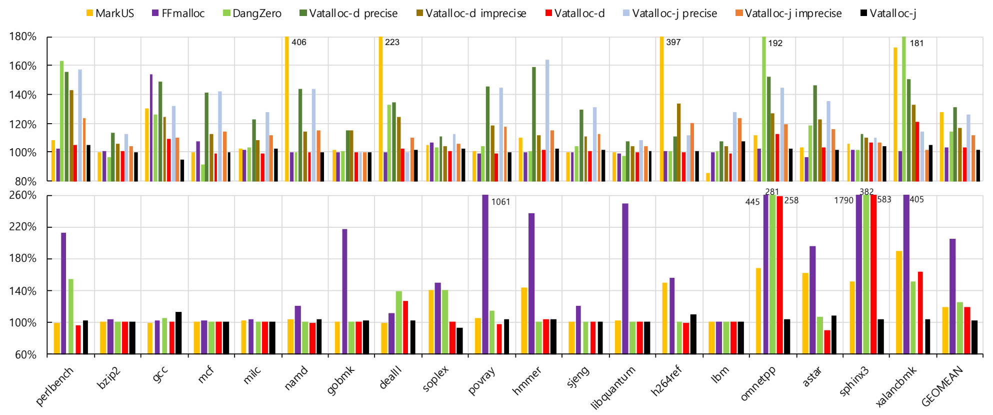

Figure 2.8: Performance and Memory Overhead on SPEC CPU2006.

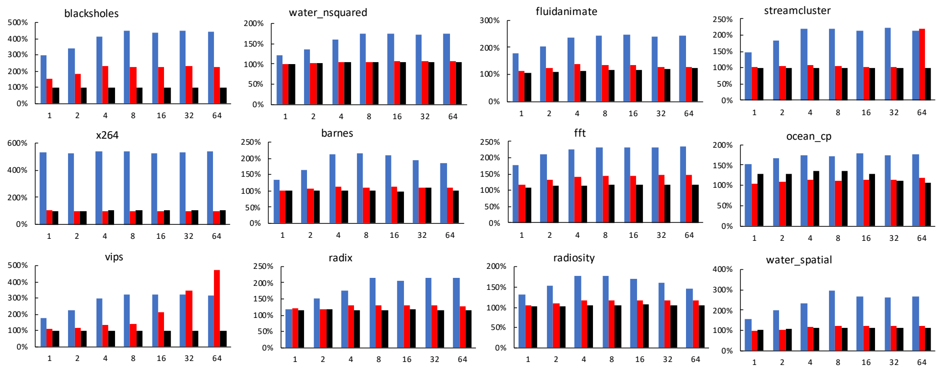

Figure 2.9: Performance Overhead on PARSEC. The blue bar refers to FFMalloc, the red bar refers to Vatalloc-d, and the black bar refers to Vatalloc-j.

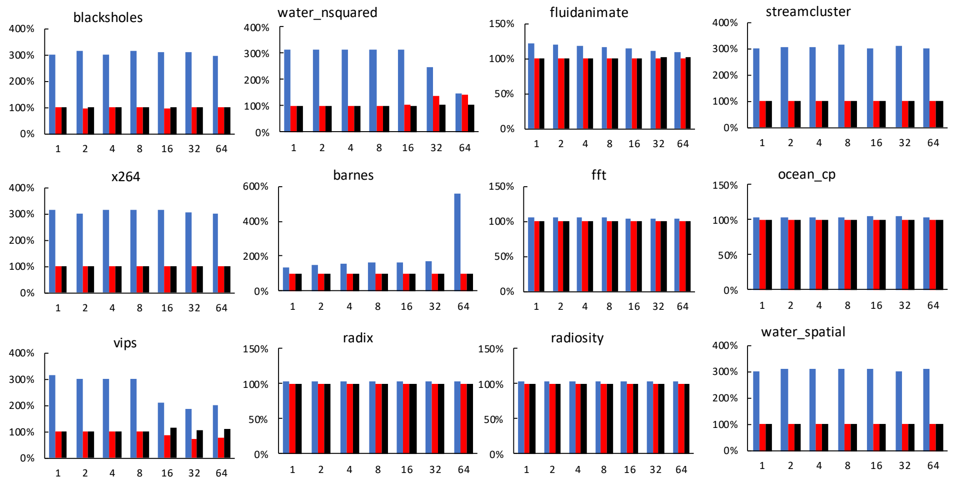

Figure 2.10: Memory Overhead on PARSEC. The blue bar refers to FFMalloc, the red bar refers to Vatalloc-d, and the black bar refers to Vatalloc-j.

A55 quad core processor and 4GB RAM, by shadow-mapping the tag memory. For DangZero, we referred to the numbers that have been reported in their original papers as their prototype only supports Intel architecture.

Modified Code Size Vatalloc-d was built on dlmalloc-2.7 by adding 0.9K lines of source code, and Vatalloc-j was implemented by adding 0.2K lines to the source code upon jemalloc-4.5. Vatalloc can be even implemented in FFmalloc for extension, to further reduce the exhaustion of va space.

2.6.2 Proxy Measurement

To measure an accurate estimation of worst case performance of MTE, both tag update, which is composed of loading and storing tags, and tag matching have to be considered. While recent researches using MTE [42, 43] overlooked the second cost, but from what we have observed, it adds significant overhead to the system. First, we simulated the update of memory tags by reserving a large memory region to store tags and executing memory instructions, which write a single byte to the reserved memory on every load or store operation. The tag comparison overhead is assumed to be mostly

```
ADD     X1 ,  X1 , X2 , LSR #5
        // X0: TAG
        // X1: Tag memory base
        // X2: Target address
STR     X0 , [X1 ]
```

(a) Tag Store

```
MOV       X15 , #TAG_OFFSET
ADD       X15 , X15 , X1 , LSR #5
LDARB     XZR, [X15]  // barrier
LDR       X0 , [X1]
```

(b) Tag Load (instrumentation1)

```
MOV     X15 , #TAG_OFFSET
ADD     X15 , X15 , 0
LDR     XZR, [X15]
LDR     X0 , [X1]
```

(c) Tag Load (instrumentation2)

```
MOV       X15 , #TAG_OFFSET
ADD       X15 , X15 , X1 , LSR #5
LDR       XZR, [X15]
LDR       X0 , [X1]
```

(d) Tag Load (instrumentation3)

Figure 2.11: Instrumentation for Proxy Measurement

hidden since it is performed by a separate MTE hardware logic which runs concurrently with the CPU cores [30]. Meanwhile, tag loading has a potential performance impact on systems in the precise trapping mode. By loading additional tag bits, more memory pressure and cache overhead are induced. Additionally, ARM's weak memory model will incur higher costs to ensure that all the memory operations are observed after the tag is loaded. We pessimistically approximate this effect of memory dependency by instrumenting target programs in two-fold. Firstly, in the *instrumentation1*, before every memory access, loading a corresponding tag is performed using implicit load-acquire barrier as seen in Figure 2.11.(b). However, loading a tag from shadow memory to the cache and then to the CPU requires computations of the tag address, which will be transparently performed by the MTE unit. Also, extra burden to the memory unit is incurred because MTE loads tags from memory to caches but not from caches to CPU. To compensate for this, in the *instrumentation2*, dummy instructions that calculate the tag address and load the tag are inserted before every memory access (Figure 2.11.(c)). These instructions use constant addresses to ensure that cache miss never occurs. By subtracting the runtime results of instrumentation1 from those of instrumentation2, we obtain the tag loading overhead induced by fetching the memory tag from memory into the cache and the runtime overhead caused by the stricter memory ordering restraint. Likewise, to evaluate performance of the imprecise mode, we instrumented target programs (*instrumentation3*) and subtract the performance results of *instrumentation2* from those of *instrumentation3*. Note here that in *instrumentation2*, a load instruction without any forward dependency is inserted as in Figure 2.11.(d). As tag checking operation is performed in MTE asynchronously,

2.6.3 Performance Overhead

Figure 2.8 reports the performance numbers of Vatalloc and other techniques that are measured on SPEC2006 single-threaded benchmarks. We used *-O2* as a default optimization flag, but we used *-O1* for *perlbench* and *dealII*, and *-O0* for *gcc*, *omnetpp* and *namd* as these benchmarks crashed with instrumentation. For convenience, hereafter we refer to measurement of Vatalloc postulating precise mode as *Vatalloc precise*, and imprecise mode as *Vatalloc imprecise* respectively. In some cases, such as `gobmk`, `sjeng`, and `lbm`, uninstrumented Vatalloc is faster than the native execution; we deem that this is due to the unintended effect from the changes in the chunk layout. For the instrumented versions to measure MTE's negative potential impact on the system, Vatalloc imprecise adds up 16.9 % runtime overhead for Vatalloc-d and 12.0 % for Vatalloc-j, which provides nearly similar performance to 18.9 % slowdown of MarkUs and 14 % slowdown of DangZero without page reclaimer. Due to more stringent constraints to measure the effect of precise MTE mode, Vatalloc precise incurs larger runtime overhead (30.9 % for Vatalloc-d and 25.5 % for Vatalloc-j). *hmmer* resulted in the worst number for Vatalloc precise, but the number is significantly dropped in Vatalloc imprecise. The worst slowdown in Vatalloc imprecise is observed in *perlbench*.

Vatalloc-d (w/o MTE) and Vatalloc-j (w/o MTE) refer to uninstrumented versions of Vatalloc-d and Vatalloc-j, where performance degradation of MTE is not considered. Vatalloc-d and Vatalloc-j shows 1.70 % and 3.05 % geomean performance overhead respectively, which means without adverse effect of MTE, our mechanism induces merely negiligible overhead. The measurement result indicates that in systems with MTE for enhanced spatial safety, Vatalloc offers UAF prevention with a slight overhead.

We also tested 12 multi-threaded PARSEC-3.0 workloads to investigate multi-threading performance, and compared the results with those presented in DangSan. We measured uninstrumented Vatalloc, Vatalloc precise and FFMalloc on our system. We can observe that Vatalloc-d precise incurs high overhead on average (geometric mean, 65.8 % at 64 threads). Uninstrumented Vatalloc-d and Vatalloc-j precise introduces 36.8% performance overhead. On the other hand, Vatalloc-j adds negligible overhead on average (8.5 % at 64 threads), which shows that the multithreaded performance of jemalloc is not degraded by Vatalloc. For Vatalloc-j precise, 36.7% slowdown occurs.

2.6.4 Memory Overhead

The memory overheads of Vatalloc mostly arises from metadata and exhausted but not unmapped chunks. Firstly, Vatalloc consumes different amount of memory for metadata. Vatalloc-d does not require additional memory because all its metadata are embedded in the existing metadata of live or exhausted chunks. On the other hand, Vatalloc-j spends some memory on storing metadata in the data structures of jemalloc. In the worst case, Vatalloc-j uses less than 130 bytes for every page, which roughly increases memory consumption by 3.0 %.

Next, exhausted chunks are retained until unmapped at page level when their total size accumulates to the page size. To measure the relevant overhead, we used the maximum resident set size reported in the processor status (i.e., `/proc/pid/status`). We first observed that Vatalloc adds up to 19.0 % of maximum resident memory on dlmalloc, and 3.0 % on jemalloc with respect to SPEC. As reported in Figure 2.8, we compared the peak memory usage of Vatalloc with other techniques. Meanwhile, MarkUs and DangZero add up 18.9 % and 25% respectively, and FFmalloc shows worst memory overhead of 104.1 %. In most cases, Vatalloc-d does not increase memory consumption, but there are exceptional benchmarks in omnetpp and sphinx3, which tend to allocate much larger memory than they actually use. Usually, such excessive allocations are not critical, thanks to demand paging technique, but they are problematic in Vatalloc because all the associated memory tags should be initialized at each allocation. Fortunately, this problem will be addressed because Linux recently added to mmap a PROT_MTE flag that initializes memory tags with a designated tag number when pages are actually mapped to physical memory frames through demand paging. In the case of Vatalloc-j, we can see near-zero memory overheads in most benchmarks. This is because due to size-segregated chunks in jemalloc, exhausted chunks do not increase memory fragmentation. Also, jemalloc has an allocation strategy that allocates more chunks than necessary in advance, which offsets the memory occupation of exhausted chunks.

In PARSEC, Vatalloc also shows a high memory efficiency on the basis of the maximum resident set size. At 64 threads, both Vatalloc-d and Vatalloc-j incur 1.8 % memory overheads on average, which are better than 104.2% of FFmalloc. In vips, Vatalloc-d has rather less memory overhead than the native execution, which is because Vatalloc-d unmaps unused (i.e., exhausted) pages more aggressively than dlmalloc used in the native execution.

2.6.5 Nginx

In order to evaluate Vatalloc on more realistic applications, we measured the performance of Vatalloc-j using Nginx web server as in Figure 2.12. We used wrk HTTP benchmarking tool of Nginx version 1.23.3. ODROID-HC4 functions as a server, while a machine with i9-10900K CPU and 128GB RAM runs as a client. We configured the wrk benchmark to execute 30 seconds per run, sending a 64-byte file. The average degradation factor over the baseline is 3.9% and 4.3% for imprecise mode and precise mode of Vatalloc-j respectively, and is 11.4% for FFmalloc.

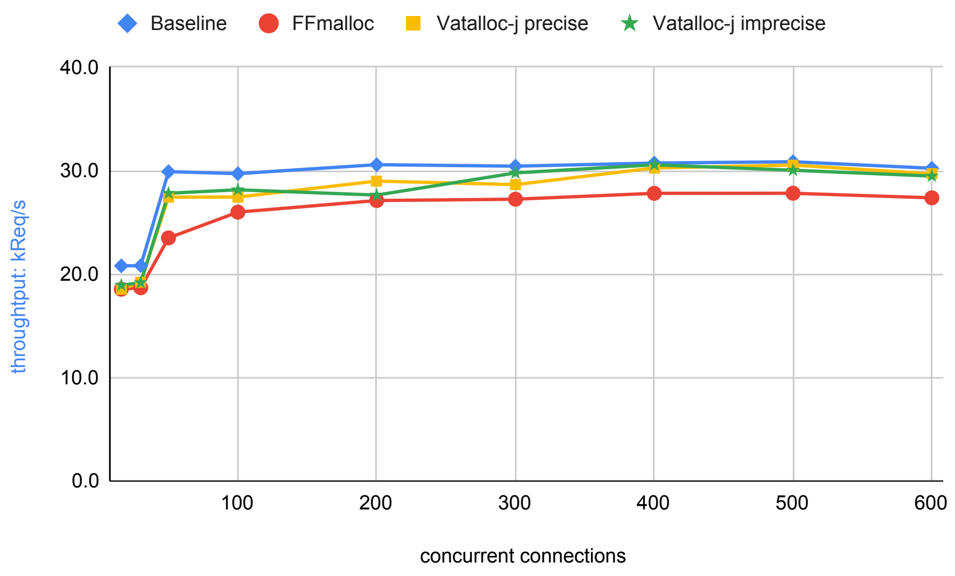

Figure 2.12: Nginx throughput.

2.6.6 Virtual Address Consumption

The detection capability of the VA-based scheme in the lock-and-key approach is retained by default until VAs are exhausted. The fact that each address can be reused multiple times thanks to our VA tagging scheme is a strength in this respect. To prove this we experimented how much VA is consumed after a single program run in Vatalloc, Oscar/DangZero, and FFmalloc, and the results are shown in Table 2.1. Consequently, we were able to observe that the VA tagging scheme of Vatalloc that facilitates VA reuses is extraordinarily helpful in reducing the amount of consumption. Address consumption of Vatalloc-j is about 1200 times lower than Oscar/DangZero for `perlbench`, which implies it would be lower than DangZero without GC-like page reclaimer by the same magnitude. As anticipated, Vatalloc consumes about 16 times lower va space than FFmalloc for `omnetpp` and `xalancbmk`. It shows that Vatalloc can support even long-running programs without failure in UAF attack detection. When combined with the page reclaimer shown by DangZero, performance improvement is expected as the

benchmarks	Oscar/DangZero	FFmalloc	Vatalloc-d	Vatalloc-j
gcc	0.4 TB	82.7 GB	1.1 GB	1.5 GB
perlbench	1.4 TB	6.1 GB	3.9 GB	1.2 GB
dealII	0.9 TB	12.0 GB	8.0 GB	3.8 GB
omnetpp	0.9 TB	47.5 GB	5.4 GB	4.1 GB
xalancbmk	0.9 TB	66.8 GB	7.4 GB	3.8 GB

Table 2.1: Virtual Address Consumption over Single Benchmark Run

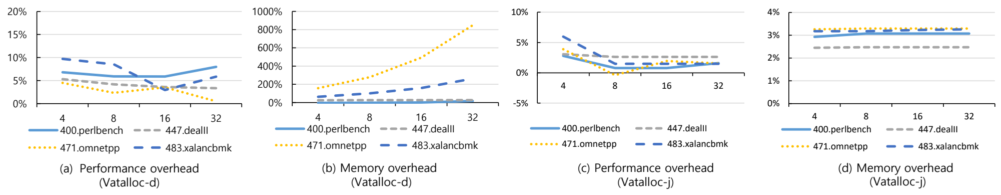

Figure 2.13: Variation in Performance and Memory Overheads by the Granularity of the Page-level Exhaustion Tracking.

number of reclaimation would be drastically reduced.

2.6.7 Granularity of Page-level Exhaustion Tracking

As described in §2.5.3, Vatalloc allows to change the granularity of page-level exhaustion tracking, which is 4 KB by default. To observe the effect of granularity change (from 4 KB to 32 KB), we conducted experiments on Vatalloc-d and Vatalloc-j with allocation-intensive benchmarks, `perlbench`, `dealII`, `omnetpp`, and `xalancbmk`, whose performance is worse relatively. As shown in Figure 2.13, results revealed a rough trend, where memory consumption grows proportionally with granularity, but execution time decreased to a certain point. For memory usage, the trend is clearly found in `omnetpp`, `perlbench`, and `xalancbmk` that perform complex memory allocation/deallocation patterns, because exhausted pages can be unmapped only when they are concatenated as much as the granularity. On the other hand, for execution time, the

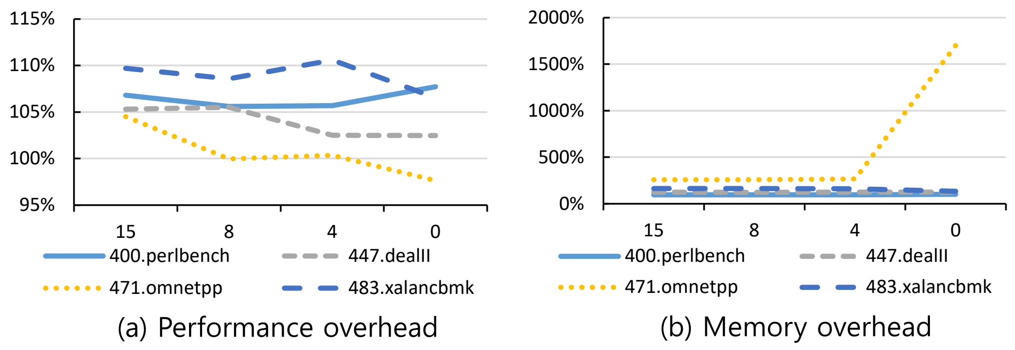

Figure 2.14: Variation in Performance and Memory Overheads of Vatalloc-d according to the Consolidation Threshold.

trend is not steady. With a bit larger granularity, the frequency of `unmap` invocations and page exhaustion state management operations (e.g., search/insertion/deletion) is decreased, which leads to performance improvement. However, extremely-large granularity makes unmapping of pages hard, prolonging the life time of the exhausted chunks. As a result, sparsely located exhausted chunks increases access overhead, reversing the performance trend in `perlbench` and `xalancbmk`.

2.6.8 Consolidation Threshold

As stated in §2.5.1, we can control the threshold of tag number difference in consolidation on Vatalloc-d. The default threshold is 15 (i.e., the maximum tag number difference), which means that Vatalloc allows any consolidation. Similar to §2.6.7, we observed how the change in the threshold affects both performance and memory in allocation-intensive benchmarks. Figure 2.14 reports the results. Lowering the threshold has two opposite effects; (negative) it may cause external fragmentation by restraining consolidations between small chunks, but (positive) it gives more chances to reuse the same VA by suppressing consolidation (i.e., tag number of the chunk with the smaller tag number is wasted via consolidation). Overall, we get better performance numbers with a lower threshold in most benchmarks. It indicates that as these benchmarks tend to make aligned and size-uniform memory allocations, they are insensitive to the external fragmentation issue, and thus the aforementioned positive effect has greater influence on performance than the negative one. When it comes to memory overhead, most benchmarks just show steady numbers. However, `omnetpp` shows exceptionally increasing overhead, which is due to the negative effect with a too-low threshold.

2.6.9 Effectiveness and Compatibility

We assume that metadata of Vatalloc and the control flow of a program are unharmed, and tag memory of MTE is not manipulated by an attacker. In case of Vatalloc-d, an attacker may maliciously point to its metadata to manipulate it. However due to the disparity of pointer tag and the tag number of the chunk, the attempt is neutralized. To evaluate effectiveness in preventing UAF errors, we manually tested Vatalloc with latest UAF vulnerabilities reported, which are CVE-2022-34568, CVE-2022-40674, CVE-2022-3352, CVE-2022-30065 and CVE-2022-36149 [44–48]. Initially, each exploit successfully executes a malicious action, such as taking over arbitrary code execution, the instruction pointer, or writing arbitrary data to memory. We observed that Vatalloc successfully prevents compromises by changing the tag numbers of the target objects upon deallocation, inducing a tag mismatch, thereby nullifying dereferences to them. One of MTE's original purposes is to enforce spatial memory safety probabilistically. In a typical method for this purpose, a pointer and its referent object are given the same but randomly selected tag number. By doing so, malicious attempts of dereferencing a pointer to access outside its referent object are detected probably due to a tag mismatch. As Vatalloc assigns random pointer and referent pairs the same tag numbers, tag numbers are distributed uniformly. In this regard, the methodology remains consistent with Vatalloc and spatial safety is probabilistically enforced alongside Vatalloc.

2.7 Discussion

2.7.1 MTE for spatial memory safety

Another major use of MTE is to enforce spatial memory safety. In a typical method for this purpose, a pointer and its referent object are given the same but randomly selected tag number. By doing so, malicious attempts of dereferencing a pointer to access outside its referent object are detected probably due to a tag mismatch. This typical method is totally compatible with Vatalloc. Since Vatalloc already assigns pointer and referent pairs the same tag numbers, the method can be achieved by Vatalloc randomizing chunks' tag numbers.

2.7.2 Porting to other memory allocators

The VA tagging scheme is generally applicable to many allocators. To demonstrate this, we have implemented Vatalloc based on two major memory allocators, jemalloc and dlmalloc, but it can be ported to other memory allocators as well. For example, we can consider porting Vatalloc to ptmalloc2, the default allocator in latest Linux versions. Unlike dlmalloc, ptmalloc2 does not share chunks among threads, but instead manages them independently in each thread by maintaining per-thread metadata. Despite this difference, as ptmalloc2 basically stems from dlmalloc, the tag management is applicable to ptmalloc2. Also, the page-level exhaustion tracking can be conducted at each thread by maintaining the page exhaustion state thread-by-thread. tcmalloc, introduced by Google is another recent memory allocators developed for more cache-conscious memory allocation and strengthened multithreading support. Implementation of the Vatalloc's tag management mechanism is expected to be well suited with tcmalloc, since tcmalloc makes use of thread-specific and size-segregated chunks like jemalloc. Additionally, the page exhaustion tracking mechanism of Vatalloc is applicable because tcmalloc manages internal allocation pools and caches in a similar manner to jemalloc.

2.8 Related Work

Two major approaches that have been studied for UAF attack detection are *pointer nullification* and *lock-and-key*. DangNull [20], FreeSentry [13] and Dangsan [**?**] follow the former approach. They nullify dangling pointers to detect UAF attacks when the pointers dereferenced. Similar to Vatalloc, the techniques provide deterministic detection. However, they incur excessive performance (e.g., 55 % in DangNull, 25 % in FreeSentry, and 41 % in DangSan) to the system in constantly maintaining their dedicated data structures that keep track of the referring relationships between objects and pointers. To reduce performance overhead, a deferred free scheme [22, 23] has been devised that intentionally delays the reuse of freed objects' memory, inspired by the fact that UAF attacks will be launched shortly after objects are freed. This scheme can be implemented easily by placing the freed objects in quarantine memory for a while. In fact, in spite of this scheme, AddressSanitizer [21] still incurs high performance degradation, because this technique monitors every memory access, aiming to detect not only UAF attacks but also out-of-bounds accesses with a debugging purpose. More importantly, due to the limited size of the quarantine memory where the freed objects are residing, this technique can detect UAF attacks only probabilistically. Therefore, to guarantee deterministic detection of UAF attacks with a low overhead, pSweeper [16], CRCount [18], MarkUs [25], CHERIvoke [17], and Cornucopia [24] have added optimization mechanisms to this deferred free scheme. For example, pSweeper runs pointer nullification in a separate thread. CRCount waits for dangling pointers to be nullified implicitly until the freed object's reference count is zero. MarkUs is similar to CRCount, but instead of reference counting, it runs pointer marking in a separate thread that scans memory to find remaining dangling pointers. Thanks to such optimization mechanisms, these techniques can achieve detection more efficiently, but it comes at a cost. Compared to Vatalloc, they can detect only one type of UAF attacks, and leave undetected the other type of UAF attacks against the freed objects whose memory region has not been reused yet. Moreover, their overheads are still somewhat large, and they necessitate source code to apply their optimization. The only techniques that provide comparable detection with Vatalloc are CHERIvoke and Cornucopia. However, as they ultimately take advantage of the pointer marking mechanism of MarkUs, they also provide only partial detection of UAF attacks. In addition, implementing them requires an architectural capability [49]. There is an effort to implement it in the real evaluation board [50], but it has not yet been integrated into commodity processors, unlike MTE.

Unlike the above-mentioned techniques, CETS [51] and Oscar [15] are based on the lock-and-key approach. They distribute locks and keys to objects and pointers, respectively, and check whether locks and keys match on each memory access. In the case of CETS, locks and keys are defined as 64-bit integers, which requires expensive data structures that manage them by object and by pointer. Even worse, the data structures should be referred to on every memory access, which explains the large performance overhead of this technique. To lessen the overhead and activate detection even without source code, Oscar apply the VA-based scheme that uses VAs as locks and keys, which is realized by ensuring each object to be allocated in unique VAs using virtual memory mapping. However the scheme spends many CPU cycles which are entailed by extremely frequent kernel intervention for memory management (i.e., invoking system calls such as `mmap`, `sbrk` and `munmap`) in order to assign unique VAs to every newly created object as well as to invalidate the VAs of freed objects. Recent work in this category, a VA-based technique, called *FFmalloc* [27], has attained a reduction of runtime overhead by resolving the challenge through batched invalidation of keys, but it achieves this result only at a cost; that is, loss of its detection capability. With FFmalloc, freed objects are still accessible through dangling pointers without being detected until they are finally relinquished to the kernel when freed memory reaches a certain number of consecutive pages. As the latest work in this category, DangZero [28] solves this issue by granting the program's allocator an authority to access page table directly for management and invalidation of keys. Direct page table access is normally performed in *ring 0*, which is the highest privilege running kernels. To do so, kernel modification is required to run user-space applications in *ring 0*, which harms the applicability of the scheme. Vatalloc show quite improvement in performance and memory overheads over Oscar, thanks to the VA tagging scheme of Vatalloc which enables it to reuse VAs several times by capitalizing on MTE. FFmalloc has improved performance compared to oscar, but at the cost of not being capable of deterministic detection. In case of Vattaloc, depending on the desired mode, checks for tag mismatches generate a segmentation fault on safety violation. Vatalloc also shows better security and is more applicable to long-running programs than FFmalloc, because it can significantly delay VAs from being exhausted by supressing consumption of it.

2.9 Conclusion

We proposed a technique, Vatalloc, that provides efficient, drop-in-use, and instant detection against dangling pointers. Such an effective and practical detection capability comes from our novel VA tagging scheme that advances the VA-based lock-and-key scheme, by capitalizing on MTE. The VA tagging scheme dramatically reduced the frequency of TLB misses and kernel-involved page management that mainly impairs the performance of the original VA-based scheme. We realized our scheme by implementing Vatalloc on dlmalloc and jemalloc, and demonstrated its effectiveness.

Chapter 3

TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code

3.1 Introduction

C/C++ have been dominant languages for several decades, but they are unsafe due to the permissive semantics that allows many undefined behaviors often manifested as a myriad of security-critical pervasive bugs, such as buffer overflows, use-after-free, and other memory exploitable errors [11,23,52–54]. The language *Rust* [55,56] was invented to tackle this security problem inherent in C/C++ by introducing new syntax and semantics such as pointer ownership, lifetime, and borrowing. The language features and constructs of Rust are elaborated to facilitate static analysis for safety guarantee, thereby requiring little or no runtime sanity checking. However, despite Rust's strong static security guarantees, its strict semantics and rules for the guarantees potentially limit expressiveness [57] as well as performance [58]. Thus for practicality, Rust relaxes its strict safety rules by allowing programmers to include *untrusted code*, which can be exempt from strict sanity checks at compile time. One source of the untrusted code is code sections in Rust, named *unsafe blocks*, which typically contain a handful of the operations crucial for low-level coding or critical for performance, such as raw pointer manipulation and unprotected type casting. Those operations wrapped within a Rust unsafe block are written in a second language beside Rust, called *unsafe Rust*, which does not necessarily obey all safety rules of the true Rust (preferably, *safe Rust*) language. As the other source of untrusted code, external libraries written in unknown languages are used by programmers to save their development costs. The programming practices encompassing such untrusted code with safe Rust indeed bring diverse benefits in terms of expressive power and efficiency, but will undermine the strong safety guarantees by safe Rust evidently because running untrusted code in the same address space as safe Rust code would put the entire program into danger of being exposed to exploits from unsafe blocks or external libraries.

In this paper, we aim to shed light on this potential security risk of Rust programming with untrusted code sources, and to propose our compilation framework, called TRUST, which is designed to mitigate the risk via *in-process isolation* where the Rust code is transformed by TRUST to quarantine unsafe blocks and external library functions from safe Rust code sections which we will say hereafter as *safe blocks*. We opted for an in-process scheme because it is generally deemed more efficient than the inter-process counterpart for isolation [59] that requires costly OS intervention for virtualization support. In fact, there are several in-process schemes [60, 61] previously developed to address this risk with such efficiency, but they have several limitations, such as lacking Rust-aware program analysis, requiring expensive context switches, or relying on developer annotations.

By quarantining unsafe blocks and external libraries from safe blocks, TRUST prevents exploits in such untrusted code from corrupting critical data of safe Rust without proper permissions. To implement our in-process isolation for Rust applications, TRUST first divides the memory into two regions, *safe* and *unsafe*, which contain safe objects and unsafe ones, respectively. Then, TRUST differentiates access permissions of Rust code blocks to these regions. It permits safe blocks to access both the regions. In contrast, it denies by default any access of unsafe blocks and external libraries to the safe region, although some untrusted code blocks may be given limited permission to read the safe region depending on its security policy. It is noteworthy here that TRUST can perform access control on every individual object in a Rust program by determining which objects are allocated to either of the regions. That is to say, as for safe stack/heap objects whose confidentiality and integrity are of top priority, TRUST allocates them into the safe region so that any access of untrusted code to them can be restricted, with the exception of some untrusted blocks receiving read permission to safe objects for special cases. All other objects will be classified as unsafe ones and placed into the unsafe region.

For automatic instrumentation of the Rust code for in-process isolation, we have made modifications to the Rust framework, including the frontend for IR generation and the backend for binary generation. When the Rust code is instrumented for in-process isolation, TRUST applies different techniques to cope with an unsafe block and an external library because the former can be presented to TRUST as source code, while the latter is assumed available only in binary form. Given the source code for an unsafe block, TRUST applies *software fault isolation*. Firstly, to handle unsafe heap objects in the code, TRUST identifies their allocation sites and replaces the sites with invocations to our customized allocator, which allocates memory objects in the unsafe region with a predefined address range. Next, as for unsafe stack objects, TRUST transforms the code to allocate them separately in a special stack which is positioned in the unsafe region. Finally, all memory operations in an unsafe block are instrumented with masking to deny out-of-region access. In contrast, to deal with external libraries in binary form, TRUST applies an isolation technique based on Intel *Memory Protection Key* (MPK) since we have to conservatively presume that all those library functions are unsafe and use unsafe objects only. As a result, TRUST assigns separate keys to memory pages in the safe and unsafe regions, and switches access permissions every time execution flows to and from external libraries.

We have implemented TRUST[1] by extending the Rust compiler and its runtime libraries. To evaluate the efficiency, we used fifteen widely-used crates, including core components of Rust's standard libraries. Experimental results show that TRUST slows down the test libraries by 12.65% on average. We further evaluate TRUST by comparing it with two existing techniques, namely, XRust and Sandcrust. The experimental results show that TRUST is about three times faster than XRust and more than twice faster than Sandcrust.

3.2 Backgrounds

Ownership in Rust. Rust's ownership policy enables the compiler to obviate memory-safety bugs statically. In a Rust program, a memory object must be *owned* exclusively by one variable at a time during execution. When the program needs to copy or move a pointer to a memory object, the ownership must either be transferred to the new variable permanently or be *borrowed* temporarily. The ownership policy is stringent with zero tolerance, being imposed on data representation, abstraction, and algorithm design. For example, Rust has tree-shaped linked data structures and completely disallows the implementation of mutable data structures like doubly-linked lists. As a workaround to this strict policy, Rust incorporates unsafe Rust, a dialect that allows some relaxed performance and expressiveness rules. Unlike the ones written in safe Rust, programs written in unsafe Rust may manipulate raw pointers, invoke external library functions, or use mutable global variables. In fact, a Rust program is typically composed partly of unsafe Rust code in many forms. A code block can be wrapped with the keyword `unsafe`, or an entire function can be written in unsafe Rust when the function is annotated with the same keyword. Although an unsafe block conventionally refers to the former in Rust, we abuse the term to refer to any code written in unsafe Rust for brevity.

[1]Once published, TRUST will be open-sourced to benefit future studies.

Smart Pointers. Smart pointers are a widely used concept in which a pointer is represented as a composite data type containing the memory address and metadata. Most commonly, the metadata is either the range of addresses that the pointer is expected to target or the pointer's capability. Many standard libraries in Rust use these smart pointers to ensure memory safety that cannot be checked statically at compile time [62]. For this reason, operations on smart pointer metadata are considered unsafe and thus can only be done in unsafe blocks.

Memory Protection Keys. Intel MPK [63], also referred to as Protection Keys for Userspace (PKU), is a per-thread hardware mechanism provided by Intel to help maintain memory page permissions in groups [64]. With MPK, each page is assigned a 4-bit value, called *pkey*, indicating the group to which the page belongs. A processor with MPK has a special register called `pkru` that determines the permission that the current process has for each page group. The permission can be investigated and updated by using special instructions, `rdpkru` and `wrpkru` [63]. The quick switch in permission using an in-process register motivated many earlier studies to use it for in-process isolation [65–68] as TRUST does to quarantine external libraries. In particular, TRUST creates separate memory regions for each component, uses MPK keys and PKRU to grant and revoke access rights, and adopt the existing mechanisms needed to protect this MPK-based protection against the targeted attacks. For example, TRUST uses static analysis and carefully designed entry/exit gates as presented in ERIM [66] and Hodor [67]. Additionally, TRUST monitors and hooks system calls to further prevent the external libraries from escaping the quarantine as suggested in another work, PKU Pitfalls [69].

3.3 Motivation

This section gives examples of the vulnerabilities in untrusted code undermining the memory safety of safe Rust, introduces how in-process isolation mechanisms help, and

```rust
static mut offset_in: i64 = 10;
fn main(){
  ...
  let array = [1,2,3,4,5];
  let secret_code = 12345;
  unsafe {
      let ptr = array.as_ptr().offset(offset_in);
      std::ptr::write(ptr, 10) }
  let vector = vec![1,2,3,4,5];
  unsafe {
      let ptr = vector.as_ptr().offset(offset_in);
      std::ptr::write(ptr, 10) }
  ... }
```

Figure 3.1: Potential memory vulnerabilities caused by unsafety

```rust
fn next(&mut self) -> Option<(A::item, B::item)>{
    if self.index < self.len {
        ...
    } else if A::may_have_side_effect() &&
              self.index < self.a.size() {
        let i = self.index;
        self.index += 1;
        unsafe { self.a.__iterator_get_unchecked(i); }
        None
    } else { ... }}
fn size_hint(&self) -> (usize, Option<usize>) {
    let len = self.len - self.index;
    (len, Some(len)) }
```

Figure 3.2: A buffer overflow vulnerability from unsafe Rust [1]

discusses the limitations of existing mechanisms.

3.3.1 Vulnerabilities in Untrusted Code

As long as the safe Rust code and untrusted code run in the same process, Rust programs are always vulnerable to memory safety bugs [1, 2, 70] no matter how careful

```
fn overflowed_zip(arr: &[i32]) ->
  impl Iterator<Item=(i32,&())>{
    static UNIT_EMPTY_ARR: [(): 0] = [];
    let mapped = arr.into_iter().map(|i| * i);
    let mut zipped = mapped.zip(UNIT_EMPTY_ARR.iter());
    zipped.next();
    zipped }
fn main(){
    let arr = [1,2,3];
let zip = overflowed_zip(&arr).zip(overflowed_zip(&arr));
    dbg!(zip.size_hint());
    for(index, num) in //results in stack/heap overflow
      zip.map(|((num, _), _) | num).enumerate() {
        println!("{}: {}", index, num);
    } }
```

Figure 3.3: The vulnerability in Figure 3.2 being exploited [1]

the compile-time analysis is. As one class of the untrusted code, unsafe blocks written in unsafe Rust are out of the scope of the analysis that is designed solely for safe Rust. Even more useless is this analysis when dealing with the other class of untrusted code, external libraries, which can be written in any language, including unsafe ones like C/C++. Such incompetence of compile-time analysis in guaranteeing memory safety of untrusted code may open doors for bugs of untrusted code to corrupt critical memory objects in safe blocks. Moreover,

Figure 3.1 shows a Rust program where a pointer is defined in a safe block and used in an unsafe block. The example allocates two stack objects at lines 4 and 9, and takes raw pointers targeting the objects at lines 7 and 11, respectively. The raw pointers are incremented at the same lines and then used for modifying the stack objects at lines 8 and 12. The problem in this example is that the program uses a global variable, `offset_in`, which could be modified externally anywhere else. This use of a global variable in offset computation renders the program vulnerable. At lines 8 and 14, an attacker manipulating `offset_in` can control which address is written

```rust
pub fn printw(s: &str) -> i32{
  unsafe {ll::printw(s.to_c_str().as_ptr()) }
}
```

Figure 3.4: Format String Vulnerability CVE-2019-15546 [2]

to, even reaching the safe objects. Figure 3.2 shows another example extracted from the `zip` crate referenced in CVE-2021-28879 [1], where the `self.index` may be set to a value greater than `self.len` (line 7), resulting in an integer overflow in the `size_hint` function (line 13). Corrupting the return value of `size_hint` misleads `__iterator_get_unchecked(i)` called at line 8 to erroneously return an iterator to a memory location outside the object `a`. An attacker could exploit this to create a buffer overflow when a consumed Zip iterator is used again as in Figure 3.3.

Figure 3.4 is an example showing an external library call that may undermine the memory safety of safe objects on the execution stack. The resulting vulnerability is similar to a format string bug in C/C++. It reads as much information from the stack as there are undefined string conversion parameters (e.g., `printw("%s%s%s")`). Such a bug could pave a away for attackers' exploiting external library code to read as much information as they want from the stack and print it to the standard output.

3.3.2 Mitigation by In-Process Isolation

A Rust program is composed of two distinct pieces of code: untrusted code with potential exploits just discussed in the examples above and safe blocks to be protected from such exploits. In-process isolation that isolates safe blocks and their associated data from the rest parts of the code, *i.e.*, quarantining the untrusted code, is a natural fit to solve this protection problem within a program. TRUST automatically identifies the objects that need to be protected from the untrusted code and applies in-process isolation mechanisms [71] to make sure that the untrusted code has no access to them.

XRust [60] and Fidelius Charm [61] take a similar approach as TRUST at run time. XRust enables developers to quarantine unsafe blocks using SFI. All memory access instructions marked unsafe by the programmer are instrumented with bounds

Table 3.1: Comparison of In-Process Isolation Policies

	Full	Protection from			
		Unsafe Rust		External Libs	
	Automation	Stack	Heap	Stack	Heap
XRust [60]	✗	✗	✓	✗	✗
Sandcrust [59]	✗	✗	✗	✓	✓
Fidelius Charm [61]	✗	✗	✗	✓	✓
TRUST	✓	✓	✓	✓	✓

checking to enforce the designed policy. Unsafe blocks can access only the unsafe objects allocated by XRust's additional memory management interfaces. They later show that additional memory management interfaces can be inserted automatically by interprocedural data-flow analysis. They also introduce an alternative to SFI, guard page-based protection, which trades security for performance. Fidelius Charm specializes in in-process isolation for quarantining untrusted external libraries from a Rust program. The kernel is extended to provide a system call interface for switching privilege levels. The protected Rust program uses this interface to switch the privilege level when entering or leaving an external code block. Upon each request, the kernel extension changes the access permission to certain memory pages containing safe objects by updating the page table attributes. Thus, developers are responsible for using the interface to augment their programs to protect sensitive data objects.

3.3.3 Limitations of Existing Mechanisms

Even in a combined form, these tools are inadequate to confine the attacks on untrusted code strictly, not to mention that none of the existing mechanisms quarantine both external libraries and unsafe blocks as shown in Table 3.1. First, most mechanisms require manual program changes. Fidelius Charm and Sandcrust do not demonstrate any automated transformation and leave it as a task for developers. XRust used existing data-flow analysis to transform the programs automatically, but the automatic transformation does not consider wrappers for heap allocators and smart pointers that Rust uses. Second, existing mechanisms require expensive context switches when entering and leaving the quarantined untrusted code context. XRust does not explicitly address the unsafety introduced by external libraries, and Fidelius Charm changes the attributes of page table entries for every context switch. This context switch is expected to exhibit long latency because the attributes of many page table entries must be updated, and the corresponding entries must be flushed. Third, Fidelius Charm, the existing in-process isolation mechanism for external libraries, does not ensure the integrity of the stack pointer. There are many existing in-process isolation mechanisms [66, 72, 73] in which the default context has the lower privilege, and a process temporarily enters a context with higher privilege. Unlike these, Fidelius Charm and TRUST create a context with lower privilege and give full control of the register content temporarily. This leaves the register containing the stack pointer unprotected, and an attacker may forge a *counterfeit stack* and make the stack pointer target it to affect the behavior of safe blocks. Finally, the sandboxing must also consider the stack objects because an attacker corrupting stack objects could mislead the safe Rust to violate memory safety.

Manual Analysis and Transformation. Existing mechanisms require manual program changes. Developers are supposed to identify the objects that untrusted code uses and handle them with the newly proposed memory management functions. XRust [60] is an exception in that it demonstrates how existing data-flow analysis can be used to instrument memory operations involving unsafe objects, but the application of Xrust primarily requires manual code changes, as we found in the open implementation of XRust [74]. For example, to harden an earlier example shown in Figure 3.1, the developer is supposed to notice that `vector` is used in an unsafe block and change the code to allocate memory from the unsafe region. Without this, XRust will not recognize the unsafety and place the `vector` in the safe region.

Inefficiency in Sandboxing External Libraries. Existing mechanisms have high context switch overhead when entering or leaving an external library. Sandcrust [59] uses a separate process running with different virtual address spaces to serve external libraries for the main process. The Rust-written code can use inter-process communication (IPC) to invoke an external library function instead of its original form, a simple function invocation. This remote procedure call obliviously increases the overhead because the arguments and return values must be passed over IPC. Moreover, the developer is responsible for translating the program manually, *i.e.*, for delivering whatever data object an external library needs or updates to and from the external library. Fidelius Charm [61] does not require IPC, but the program still needs to ask the OS kernel for a context switch. Unfortunately, this call to the kernel is still expensive because it updates the page tables to change the permission that the process has for its memory pages. The page table updates change the attributes of Rust program's pages, thereby invalidating the corresponding translation lookaside buffer (TLB) entries and increasing the TLB miss rate.

3.4 Threat Model and Assumptions

We make no assumptions about the untrusted code, which includes unsafe blocks and external libraries. The trusted computing base (TCB) of TRUST includes three components at run time, which are the safe blocks we assume to be trusted , the Rust runtime, and TRUST runtime. Untrusted code may have memory corruption vulnerabilities or a random series of vulnerabilities that an attacker can chain to make arbitrary memory access. Such an attacker can also compose a gadget chain to execute an arbitrary code within the context of untrusted code. We consider remote attackers aware of these vulnerabilities in untrusted code with which the victim program runs. The attackers aim to corrupt the safe objects and make safe blocks misbehave by exploiting the vulnerabilities. Attackers aware of vulnerabilities in external libraries used by a Rust program may exploit them to access otherwise privileged memory meant for safe Rust use only. Developers are assumed to know these assumptions and the benefits of using TRust, to ensure that security-critical memory objects should be used only in safe blocks. If the above assumptions hold, TRUST guarantees that external code cannot read from or write to safe objects used only by trusted code.

3.5 Design and Implementation

TRUST considers any memory object unsafe if it is used in untrusted code consisting of unsafe blocks of Rust, and foreign function interface(FFI) through which external library codes written in other languages such as C/C++ are facilitated in conjunction with Rust. The objects that TRUST can prove statically to be untouched by untrusted code are classified as safe. TRUST isolates unsafe objects in a separate region such that exploitation in untrusted code does not affect outside the region. Only safe blocks are allowed to access both memory regions, whereas external code can not read from or write to safe objects, and unsafe blocks can not write to safe objects. To quarantine untrusted code, TRUST analyzes and transforms a Rust program at compile-time and runs the resulting program with its runtime library.

Analysis and Transformation. TRUST first collects the Rust-specific attributes during compilation from the Rust source code to LLVM IR for the later stages. The attributes include the *unsafety* information that indicates if an LLVM IR instruction belongs to an unsafe block or not and function API information indicating whether a given function belongs to a foreign function interface(FFI). The compiled LLVM IR code is then passed to the points-to and value-flow analyses stage. This stage identifies pointers used in unsafe memory operations and performs necessary actions to isolate them. It marks instructions accessing unsafe stack objects for later relocation and reroutes heap pointer allocation calls to an unsafe allocator. The output from this stage is then passed to the LLVM compiler, which runs several passes to finalize TRUST's static operations. The final stage first relocates all unsafe stack pointers to the unsafe object stack and finally inserts *entry and exit gates* around external library calls.

Runtime. To serve the transformed program and properly quarantine the untrusted code, TRUST hooks several system services and maintains *per-thread metadata*. The heap allocator calls from untrusted code are rerouted to the additional, unsafe heap allocator that serves the requests with the chunks in the unsafe region. TRUST further hooks the memory management system calls from the external libraries to prevent them from altering page table attributes. It also augments `pthread` to initialize or destroy stacks for additional threads.

Challenges. While designing TRUST, we encountered three noteworthy challenges. Firstly, the memory safety of safe Rust is not entirely provable at compile-time. For the memory accesses that the compiler cannot reason about, such as those to heap objects, Rust uses *smart pointers* to ensure spatial and temporal safety dynamically at runtime. For this reason, manipulation of either the pointers or smart pointer metadata in unsafe blocks may result in memory-safety bugs in safe blocks [3, 75]. When identifying the objects or pointer uses that are potentially unsafe, TRUST must take metadata of them into consideration. Secondly, unlike the unsafe blocks, external libraries cannot be isolated with SFI. Such external libraries are assumed to be delivered in the form of executable binaries such as shared objects or static libraries. TRUST cannot analyze and transform them at compile-time regardless of the language that an external library is written in. Some studies have shown that SFI can be applied to binary programs, but they incur relatively high overhead [76]. For this reason, TRUST instead relies on Intel MPK mechanism to isolate the external libraries and restrict their memory access. Finally, Rust's allocation of heap memory through the `Alloc` crate and handling of heap pointers through smart pointers and container crates such as `Box, Vec, String` possess a challenge to points-to and value-flow analysis. Instead of direct invocation of the heap allocator for memory, Rust relies on these crates to invoke the allocator and create smart pointers. The `Alloc` crate exposes functions such as `__rust_alloc, __-rust_realloc, __rust_alloc_zeroed`, wrapping the corresponding heap allocator interfaces. With this design, all heap pointers (safe and unsafe) appear to be aliasing only a handful sources and a single sink. TRUST handles this issue by utilizing the value-flow analysis to construct a call stack for the allocation of unsafe pointers. Then it creates a clone of the call stack, rerouted to the unsafe allocator.

3.5.1 Points-to Analysis

TRUST analyzes a Rust program at *Mid-level IR (MIR)* and LLVM IR level to find the allocation sites for the unsafe objects. TRUST considers an allocation site as safe and classifies an allocation site as unsafe if it finds a flow from the site to a memory access instruction in an unsafe block or to an external library. An allocation site remains safe only if TRUST can soundly conclude that the pointer obtained from the allocation site never flows to external libraries and is used for writing in unsafe blocks, as we describe in the rest of this section.

MIR-level Analysis. TRUST associates each LLVM IR instruction with the block that the instruction belongs to for later analysis while the Rust compiler translates the program from MIR to LLVM IR. Rust compiler first translates the source code to MIR, and performs the Rust-specific static analysis at the level. For example, rules related to ownership are mostly checked at that level. For this reason, each MIR instruction is tagged with the block that it belongs to, either safe or unsafe. However, this tag is not retained when the program is translated to LLVM IR because the Rust compiler does not need to determine which block is an LLVM IR instruction generated from. On the contrary, TRUST needs to distinguish the LLVM IR instructions generated from unsafe blocks for its analysis. To this end, TRUST extends the Rust compiler to attach *unsafety metadata* to LLVM-IR instructions generated.

Points-to Analysis. TRUST performs points-to analysis to classify the memory allocation sites (*i.e.*, `alloca`, calls to `malloc` or similar) into safe and unsafe. We consider `alloca` instruction as a source of pointers as well to take the stack objects into consideration. A context-sensitive value-flow analysis, SVF/SUPA [77, 78], backs our points-to analysis for improved precision. TRUST iteratively performs bottom-up value-flow and points-to analysis to obtain precise yet sound points-to relations and classify the allocation sites using the result. This bottom-up analysis is a graph traversal problem,

```
fm read_vec(idx: usize, vec: &mut Vec<i32>){
    unsafe{ vec.set_len(idx+1); }
    vec[idx] = 256;
    println!("The container has been hacked: {}",vec[idx]);
}
```

Figure 3.5: Unsafe Blocks Allow Programmers to Manually Modify Smart Pointer Metadata [3]

where nodes are pointers, and edges are the instructions using them. In this sense, a program under points-to analysis can be viewed as a *Value-Flow-Graph (VFG)*, as defined by SVF/SUP [77,78]. An allocation site is classified as unsafe if it produces a pointer that flows to at least one instruction in an unsafe block. Otherwise, an allocation site is classified safe, which means that it is within a safe block, and the pointer it produces is not manipulated or used in an unsafe block. Even if a memory object is shared between the safe and unsafe blocks, the object will be allocated from the unsafe region because the pointer's allocation site becomes unsafe. This policy makes the decision of TRUST to allocate an object from a safe region to be sound, *i.e.*, memory objects that untrusted code may access are always allocated from the unsafe region. A drawback of this approach is that a memory object that only safe code accesses could be classified as unsafe if TRUST fails to distinguish the object from a different, unsafe one due to the limited preciseness of points-to-analysis.

Handling Smart Pointers. We found that, even if the points-to analysis does not report any modification of the pointer in an unsafe block, the use of the pointer could become unsafe due to the design of Rust runtime. Some pointers in Rust are represented using smart pointer types that include attributes for several purposes, including dynamic bounds checking at run time. Manipulating such attributes could lead to out-of-bound memory access within a safe block because the dynamic bounds checking using the corrupted attributes fails to find the problem, as an example (Figure 3.5) suggests. Our analysis overcomes this by considering an aggregate allocation site as unsafe if any field of the allocated object is potentially manipulated in an unsafe block and marks the

```
;;original store in unsafe block
{
    store i32 10, i32* %ptr, align 4, !UNSAFE_BLOCK
}

;;transformed bitcasted store
{
    %temp = ptrtoint i32* %ptr to i32,  !UNSAFE-BLOCK
    %masked_ptr = and i32 %temp, %UNSAFE_MASK,  !UNSAFE-BLOCK
    %bitcasted = bitcast i32 %masked_ptr to i32*,  !UNSAFE-BLOCK
    store i32 10, i32* %bitcasted, align 4, !UNSAFE_BLOCK
}
```

Figure 3.6: Instrumenting Store Operations in Unsafe Blocks at LLVM IR Level

corresponding allocation site to be unsafe.

3.5.2 Instrumenting Memory Accesses for SFI

TRUST instruments the store instructions in unsafe blocks to prevent them from directly corrupting the safe objects. In particular, TRUST inserts instructions that mask the address before these store instructions to enforce that the instructions are writing only to the unsafe region as shown in Figure 3.6. The static bound that the inserted instructions use is determined when the program starts. In addition, TRUST instruments some store instructions in the safe block as well to address the *confused deputy* problem. As discussed earlier, some smart pointers are allocated on the unsafe region because they are legally modified in unsafe blocks. This exposes their metadata to arbitrary corruption by an exploit, and this corruption may confuse the safe block that uses the metadata for bounds checking. TRUST automatically identifies such vulnerable pointer flow within the safe block and inserts the bounds check to ensure that the pointer targets within the unsafe region. To find the vulnerable pointer flows, TRUST uses the value-flow graphs. TRUST first finds the paths in the value-flow graph from an unsafe pointer to a store instruction in the safe blocks. Along such a path, TRUST inserts a bounds check that ensures that the unsafe pointer targets the unsafe region as shown in

Figure 3.7. Lastly, the commonly used memory modification functions such as `memset`, `memcpy`, and `memmove` are specially handled instead of being instrumented within. At the call sites, TRUST inserts a bounds check to ensure the whole range of memory to be written is contained in the unsafe region if the destination pointer is unsafe.

3.5.3 Instrumenting External Library Calls

TRUST inserts the entry gate and exit gate before and after an external library invocation to quarantine external libraries using Intel MPK. The external function, the callee, is originally responsible for saving the stack pointer on its stack and restoring it, in which the attacker can create a fake stack from scratch and set the stack pointer when returning to the Rust code. These gates update the `pkru` register to temporarily restrict memory access privileges while the external library runs, switch the stack pointer, so the external library uses the unsafe stack, and save the stack pointer in a protected memory for retrieval after returning from the external library. They also write the appropriate data entries of the thread-specific data structure for TRUST runtime (see §3.5.4).

Entry Gate. In the entry gate, the program saves the safe stack pointer (`rsp`) in the thread-specific data structure residing in the special region (always readable). It updates the domain entry in the data structure to 1 to indicate that the program runs in the context of external libraries, and then the `pkru` register to revoke read-write permission to the safe memory region (with pkey 0), and write permission to the always-readable special region (with pkey 2). Finally, `rsp` is set to point to the unsafe execution stack for external libraries.

Exit Gate. On return from the external library, the program makes a simple call that matches the contents of the `r15` with those saved earlier by the entry gate. In case of a mismatch, TRUST considers it a violation and resets the `r15` register to the correct value. This check is important because attackers may misdirect TRUST to execute the safe block on an unsafe stack, as explained above. Next, it updates the access permission of the safe region and special page (on which the `r15` register contents were saved during entry) to read-writable. At this point, write permissions to the safe region have been restored, so TRUST updates the domain entry value of the thread-specific data to 0. Finally, it loads the saved safe stack pointer from an offset to the `r15`, and updates the RSP.

Comparison to SFI. A potential alternative to our design choice, the use of MPK, is SFI. Prior studies [79, 80] have reported that it is possible to apply SFI even to binary programs, and it is theoretically possible to use SFI for our purpose as well. The rationale behind our choice of using MPK is the frequency of domain changes while running external libraries and the number of memory access instructions. The performance overhead of SFI is primarily related to the total number of memory instructions because each memory instruction must be instrumented with bound checks. In contrast, the overhead of MPK is related to the number of transitions. When quarantining the external libraries, we expect to have a relatively small number of transitions, while each library call may have many memory instructions.

3.5.4 TRUST Runtime

TRUST runtime is composed of per-thread metadata store for TRUST in the safe region, an additional heap allocator that handles unsafe object allocations, hooks to memory management system calls to prohibit the external libraries from altering page attributes, and hooks to `pthread` that initializes the TRUST runtime for new threads. After the unsafe region is established, these components work as follows. First, the unsafe heap allocator and system call hooks ensure that any objects or pages that untrusted code allocates lie in the unsafe region. Failing to do this does not undermine security because the untrusted code is prohibited from accessing the safe region separately, resulting in false positives. Second, the `pthread` hooks prepare the additional stacks that TRUST needs for new threads and destroy them when the thread terminates. Third, per-thread metadata is used by the instrumented code and the runtime to store TRUST-specific data that must be protected from the untrusted code. Fourth, the system call hooks prevent untrusted code from altering the page attributes, which the MPK-based isolation of TRUST relies on.

Establishing the Unsafe Region. TRUST establishes the unsafe region when the program starts by obtaining a large enough number of virtual pages from the OS, and the total size of such pages is 4GB in our implementation. The pages are mapped using the flag `MAP_FIXED` and configured to have the pkey for the unsafe region. As we describe later, any demand on memory chunks or pages on the unsafe region will be served using these mappings. The approach that maps the unsafe region at the startup automatically ensures that the remainder, the safe block, will always obtain chunks or pages from the safe region because the OS does not map already mapped pages unless requested explicitly. It is worth noting that the size of the reserved address range can be increased if the program is expected to use more than 4GB for unsafe objects, and doing so will not incur performance or memory overhead. Requesting the OS kernel to reserve more memory will only set more virtual pages aside, and only the virtual pages that are actually allocated will be mapped to the physical pages. It would be even better if the OS kernel supports a different means that TRUST can use to reserve a particular virtual address range without creating a fixed mapping, but the Linux kernel that our prototype runs on does not have such a functionality, to the best of our knowledge.

TRUST Metadata. TRUST maintains one metadata store for each thread to keep TRUST-specific data and represent the privilege level (e.g., safe, unsafe, or external) for the other components of TRUST runtime. The metadata has a one-bit flag representing its privilege level, called `external`. The `external` bit is to be set when the program enters an external library and to be unset on exit. The metadata also includes the field for safe stack pointer, which TRUST uses to verify the stack pointer integrity on its exit gate as described in §3.5.3.

The Unsafe Heap Allocator. TRUST uses an additional, modified heap allocator to manage a heap on the unsafe region, using `mimalloc` 1.7.0 [81]. Heap allocation requests from untrusted code are redirected to here by the function cloning (**??**), and this allocator serves the requests with the chunks from the unsafe region. The modified allocator manages its heap on the unsafe region by obtaining the new pages for the heap from the established unsafe region. The requests for large chunks, which are typically served directly with `mmap`, are also served using the unsafe region pages. The program uses this unsafe heap in two ways. The unsafe blocks that TRUST compiles from the source code are transformed to invoke the unsafe heap allocator explicitly, whenever needed. The external libraries, on the contrary, cannot be transformed because they are delivered as executable binaries. Therefore, TRUST also inserts a hook to the safe heap allocator to selectively route a heap allocator call to the unsafe allocator. TRUST uses the `external` field in its metadata, which we described earlier in this section, to determine which heap allocator to call. The field is maintained by the entry and exit gates as described in §3.5.3, and this hook uses this bit to determine the current context. Note that while this bit is protected from exploits against untrusted code by being placed within the safe region, malicious corruption of the bit does not undermine the security guarantee of TRUST. The expected outcome of such corruption is to allocate a safe object to the untrusted code, resulting in a false positive. For heap memory reallocation and freeing, the runtime library checks whether the pointer in question lies in the safe or unsafe region and reroutes the calls accordingly.

Hooking System Calls. The runtime hooks memory management system calls (e.g., `mmap, mprotect, or mremap`) to prohibit the external library from manipulating page attributes arbitrarily. In particular, TRUST ensures that the start and end addresses of the memory that is mapped by the invocation lie within the aforementioned virtual address range corresponding to the unsafe region. The runtime also redirects calls to `mmap, mremap` from the safe block attempting to map memory from the specified unsafe region address space. Similar requests from external code are redirected to the unsafe allocator as `malloc` or `realloc` for `mmap` and `mremap`, respectively. The `mprotect` hook rejects any attempts from the external library to prohibit it from altering the page attributes, including the pkey. In addition, the external library is statically checked for the presence of instructions updating `pkru` using the algorithm proposed in ERIM [66].

Hooking `pthread`. The runtime library hooks the thread creation libraries to allocate the unsafe stacks and the thread-specific data structure. For new threads created by safe blocks, the hooked `pthread` function allocates the thread-specific data in the appropriate region. It then allocates the unsafe stacks in the unsafe region and writes their pointers in the respective entries of the thread-specific data structure. Finally, it updates the `r15` register with the pointer to the thread-specific data structure and lets the execution proceed. If an external library creates a thread, the runtime library ensures that the default execution stack for that particular thread is allocated in the unsafe region. The lifetime of a thread created by an external library is considered unsafe. Thus TRUST allocates its data structure on a page in the unsafe region and revokes write access permissions on it after writing the domain entry of the data structure. While executing external code, the runtime library reads the pointer to the thread-specific data by calling `pthread_get_specific` instead of reading the `r15` register. TRUST does this because it has no control over the `r15` register in the external untrusted domain.

Hooking More System Calls The TRUST runtime must hook more system calls to defend against attacks that a recent study [69] enumerated. For example, `sigreturn` is a system call that can be used to bypass conventional PKU-based in-process isolation techniques by corrupting the content of `pkru` register on the stack. TRUST can prevent this by storing `pkru` before the system call so that any attempt to tamper with the register is invalidated. Similarly, TRUST can naturally adopt the hooks that the earlier study found essential to further harden safe Rust from more attacks exploiting vulnerable external libraries. Besides the above, the runtime must hook more system calls to thwart the isolation bypasses, as presented in the earlier study [69]. Jenny [82] has already shown how we can comprehensively filter the system calls using `seccomp` and `ptrace`, so we can combine TRUST runtime and Jenny for completeness. The additional overhead coming from the use of Jenny is expected to be less than 5%, according to Jenny's reported overhead.

3.6 Evaluation

This section shows the performance impact of TRUST, presents the result of our security analysis, and discuss the precision and soundness of the static analysis.

3.6.1 Performance

Experimental Setup. We implement the analysis and transformation for TRUST on Rust 1.49 and LLVM 11.0.1. We run the benchmarks on a system with an Intel i9-10900K CPU, 128GB main memory, and Ubuntu 18.04.6 LTS as the operating system. We use eleven Rust crates that XRust used for comparative study with it, in execution time, memory usage, and effectiveness. For a fair comparison, we use the SFI version of XRust instead of the guard-page version because the guard-page version assumes contiguous access and is vulnerable to the *jump* accesses. We also compare the execution time of TRUST against Sandcrust using Snappy [83] as the benchmark because Sandcrust is evaluated with it. Lastly, we use two large Rust crates to demonstrate that TRUST can analyze and protect real-world software written in Rust. Note that we could not apply XRust to these because it is required to identify the unsafe objects manually to use XRust. For a similar reason, we could not compare TRUST with Fidelius Charm [61] that requires developer annotation. For comparison study, we primarily use figures and present the absolute numbers in **??**.

Execution Time. As Figure 3.8 shows, the overhead of TRUST is about 7.55% on average (geometric mean) while XRust induces 26.39%, albeit the fact that TRUST also protects safe objects on the stack and quarantines the external library. TRUST exhibits lower performance overhead despite its stronger protection thanks to its optimized implementation of SFI where it uses the address masking instead of tests and conditional branches used by XRust. Figure 3.9 shows the result of running snappy with TRUST and along with the overhead of Sandcrust. The overhead of TRUST is 8.79% for compress and 35.12% for uncompress on average, and this is much lower than those of Sandcrust, 107% for compress and 2596% for uncompress, even though Sandcrust quarantines the external libraries only.

Memory Overhead. We measure the maximum resident set size during the execution of a benchmark to evaluate the memory overhead. We use the executions without protection using the default allocator (`glibc`) as the baseline, and present the relative memory usage of the unprotected executions with `mimalloc`, TRust with `jemalloc`, TRust with `mimalloc`, and XRust. TRust with `jemalloc` and `mimalloc` uses `glibc` as its safe allocator and the latter as the unsafe allocator. As Figure 3.10 shows, TRUST uses 35% more memory on average when it uses `jemalloc` as its unsafe allocator, and the overhead goes down to 13% with `mimalloc`, while XRust induces 7% memory overhead.

Large Programs. To show that TRUST can harden large programs, we use it to quarantine the untrusted code in Tokio [84] and Hyper [85]. Tokio is an event-driven, non-blocking I/O platform for writing asynchronous Rust network applications, and Hyper is an HTTP library used as a building block for many applications. Figure 3.11 shows TRUST's runtime performance overhead on Tokio and Hyper benchmarks. We notice that TRUST instruments a high number of instructions in these benchmarks due to the large code base, but most notably, as these benchmarks use heavy parallelism (10-1000x threads), TRUST spends a significant time creating and destroying unsafe stacks during each thread spawn. In fact, because of this, benchmarks such as `tokio-mpsc` and `tokio-spawn` required reducing the default unsafe stack size to run to completion. We also find that Tokio and Hyper depend on many external libraries, requiring frequent MPK access and execution stack switching during transition to and from FFI calls.

3.6.2 Effectiveness

We evaluate the effectiveness of TRUST using synthesized exploits that are inspired by real-world vulnerabilities. We also analyze how TRUST will mitigate the other known

Table 3.2: Result of Using TRUST to mitigate the known vulnerabilities.

CVE ID	Origin	Affected Memory	TRUST	XRust
2021-29939	Unsafe Rust	Stack	✓	✗
2019-15546	External Lib.	Stack	✓	✗
2021-28879	Unsafe Rust	Stack, Heap	✓	Not Tested
2021-28028	Unsafe Rust	Heap	✓	Not Tested
2021-45707	Unsafe Rust	Heap	✓	Not Tested
2018-1000657	Unsafe Rust	Heap	✓	✓
2018-1000810	Unsafe Rust	Heap	✓	✓

vulnerabilities. Table 3.2 summarizes the result of our evaluation using real-world vulnerabilities.

Synthesized Vulnerabilities in Unsafe Blocks. We reproduced two real-world vulnerabilities, CVE-2021-29939 and CVE-2021-28879, in a small program to evaluate the effectiveness of TRUST. These two vulnerabilities are found from StackVec and Zip, respectively, and share the same root cause. Both StackVec and Zip use the `Iterator` which in turn requires the implementation of `size_hint` as in Figure 3.2. As already seen from the Zip crate, an integer overflow in `size_hint` returns an unsound value, which is later used to return an iterator without bounds checking, when used by the `get_unchecked` function from the `Iterator` trait. `StackVec::extend` [86] suffers from the same problem. Looping with an iterator obtained this way allows writing beyond the stack capacity as shown in [87]. Similarly, as shown in Figure 3.2 and Figure 3.3, reusing the consumed `zip` iterators introduces the stack or heap buffer overflow vulnerabilities which may affect safe Rust. In both of these vulnerabilities, the `Iterator` is accessed in an unsafe block where `get_unchecked` is executed. We reproduced these bugs in small programs and applied TRUST to them. For Figure 3.3, depending on where `arr` resides, it is possible to attack either the stack or heap. We make slight modifications to the code shown in Figure 3.3 and attempt to overwrite the stack, and then repeat the attack with `arr` of a heap-based container. The attack succeeds when the program runs without TRUST. However, TRUST stops the attack because `arr` is allocated in the unsafe region, but any attempted overflows into the safe region are prohibited by TRUST and cause the program to halt. We perform a similar attack on `StackVec` using older versions with no bug fix, and again TRUST manages to protect the safe stack. Finally, we applied TRUST to more programs with CVEs [3, 75, 88] and those [89, 90] examined by XRust. Our evaluation finds that TRUST manages to detect vulnerabilities introduced and thwart any attempts to leverage them to contaminate safe Rust.

Analysis on Vulnerable External Libraries. TRUST can also mitigate CVE-2019-15546. This vulnerability is found in `Window::printw` and `Window::mvprintw`. These functions pass a raw pointer to external functions without any sanitization, exposing users to format string bugs. TRUST identifies the objects whose pointer could be exposed and allocates them in the unsafe region, and the functions are considered untrusted for being external libraries. As a result, an attacker exploiting this vulnerability cannot alter any safe object because the operating system will trap such an attempt due to the protection key violation.

3.6.3 Discussion

Precision. To evaluate the precision of TRUST's analysis, we count the number of unsafe objects that TRUST finds and compare the results with the number of unsafe objects that XRust finds with the help of the developer. Figure 3.12 and Figure 3.13 shows the ratio of unsafe heap and stack object allocations when we use TRUST and XRust. Note that we do not count the number of unsafe stack object allocations for XRust because it does not protect stack objects. Out of 11 benchmarks, TRUST leaves the non-trivial amount of safe objects for 9 benchmarks despite its conservative policy that classifies any objects that the untrusted code might use as unsafe. Specifically, TRUST finds fewer unsafe objects than XRust on one (Bytes) and more on five (Json, Image, Regex, Vec, and Btree). In the case of Bytes, we found that the developer annotation for XRust on Bytes makes the unsafe allocator as the global allocator

Table 3.3: Number of memory access by safe Rust

Benchmark	Read From Unsafe	Ratio	Write To Unsafe	Ratio
base64	8.9G	22%	2.3G	59%
bytes	0.5G	43%	0.4G	32%
byteorder	0	00%	0	00%
json	3.8G	74%	1.4G	52%
image	5.7G	41%	1.3G	22%
regex	1.3G	4%	29.0M	81%
vec	5.8G	68%	0.7G	37%
string	0	0%	0	0%
linked-list	0	0%	0	0%
vec-deque	0	0%	0	0%
btree	81.0M	8%	0.4M	1%

for this specific benchmark. This also demonstrates the potential imperfectness of human annotation in that any unsafe objects that XRust misses could be accessed by the untrusted code, enabling it to access the safe heap. We manually investigated the objects that TRust classifies as unsafe while XRust classifies as safe. Figure 3.14 shows a snippet of code from the `Json` benchmark. The `write` function, used by all benches in Json, in turn calls `extend_from_slice`, which modifies `self.code`'s pointer metadata on line 7 in an unsafe block, making `self.code` an unsafe object. However, XRust misses this and classifies `self.code` as safe. Moreover, although XRust considers read operations as unsafe, they also fail to classify `slice` arguments to `write` as unsafe as these end up being used in the unsafe block at lines 8-11. This also explains the difference in the number of unsafe objects that TRUST and XRust finds.

TCB Size. We compare the size of TRUST's TCB with those of the others. As mentioned in §3.4, the TCB of TRUST at run time is composed of the core Rust runtime, the safe block, and TRUST's runtime that manages its metadata intervenes in system calls and mediates external library calls. Like TRUST, XRust also trusts the Rust runtime and the safe block, as well as its runtime library. For this reason, we quantitatively compare the TCB size of TRUST and XRust using the source lines of code (SLOC) of

Table 3.4: Number of memory access by untrusted code

Benchmark	Total Reads	From Safe	Total Writes	To Safe
base64	38.2G	37.0G(96%)	38.5G	0(0%)
bytes	0.2G	0(00%)	0.1G	0(0%)
byteorder	0.0G	0(00%)	0.0G	0(0%)
json	1.2G	0.7G(66%)	0.5G	0(0%)
image	0.3G	0(00%)	1.3M	0(0%)
regex	0.0G	0(00%)	12.0G	0(0%)
vec	0.0G	0(00%)	48.0G	0(0%)
string	0.0G	0(00%)	0.0G	0(0%)
linked-list	0.0G	0(00%)	0.0G	0(0%)
vec-deque	0.0G	0(00%)	0.0G	0(0%)
btree	0.7K	0(00%)	1.9K	0(0%)

their runtime libraries. Surprisingly, XRust runtime is composed of 8 lines of additional code that is invoked at runtime only for bounds checking. Unlike this, TRust runtime is composed of 500 lines of code, most of which are for hooking system calls and handling thread setup and destruction to quarantine the external libraries. The allocator that XRust is using for both safe and unsafe allocation consists of 6.9k lines of code. On the other hand, two interchangeable allocators of TRUST, which are `mimalloc` and `jemalloc`, are comprised of 7k and 24.5k lines of code, respectively.

Impact of Data Flow from Unsafe to Safe. TRUST leaves data-flows from unsafe objects to safe objects because safe blocks can read from unsafe objects and write to safe objects. This may leave an attacker exploiting vulnerabilities in unsafe blocks to find and exploit some logic bugs that can be exploited only by corrupting some unsafe objects, potentially enabling to use of safe blocks as a confused deputy. For instance, a function in a safe block may use user input in an unsafe region to update a security-critical configuration that lives in a safe region. The developer may trust the user input that has passed a series of sanity checks, but an attacker exploiting the unsafe block's vulnerability corrupts the input after the sanity check to corrupt the security-critical configuration in the safe region using the function as a confused deputy.

3.7 Related Work

Our work is related to the prior work on mitigating memory safety vulnerabilities, the in-process isolation mechanisms, and the mechanisms that aim to quarantine some untrusted code from safe Rust.

Mitigating Memory Safety Bugs. Researchers have strived to fight against memory safety bugs. Programs written in C/C++ are prone to such bugs due to the design of the languages and the complexity of modern software. A large body of existing mechanisms aims to mitigate the exploits of such bugs [6, 12, 52, 54, 91–93]. Compared with these, TRUST makes a unique contribution by combining and tailoring in-process isolation mechanisms to protect Rust's safe blocks.

In-process Isolation. In-process isolation is a commonly used building block to harden a program against software attacks. Among these, the SFI-based and MPK-based are most closely related to TRUST because it combines those to implement in-process isolation for a Rust program. Wahbe et al. [94] invented the design of logically separated fault domains within a single address space and instrumentation with bounds checks to prevent untrusted components' access to other components' memory. However, bound checks impose overhead on the execution of all untrusted memory accesses, even with more efficient masking-based SFI [95]. Koning et al. [71] compared the characteristics of in-process isolation techniques, including hardware based approaches, and reported that MPK is suitable for isolating or quarantining a large piece of the program due to its low permission switch latency. The advantage of MPK in permission change latency comes from the lack of supervisor calls, with the risk of leaving the permission change instruction unprotected. ERIM [66] addresses this problem by binary scanning and carefully designed call gates. It ensures that the untrusted code does not have the permission changing instruction (`wrpkru`) by binary scanning and that the instructions in trusted code are not misused by the call gates. Hodor [67] is another work on PKU-based sandboxing. While ERIM uses static binary rewriting to negate unsafe wrpkru instructions, Hodor relies on hardware watchpoints that the modified kernel provides. Despite these two studies' advances, a recent study [65] reported that they can still be bypassed by an attacker using the OS kernel as a confused deputy. Cerberus [96] attempts to complement ERIM and Hodor against attack vectors proposed by this study [69] except for the attack abusing signals. Jenny [82], a recent security implication for PKU-based sandbox, supports secure call gates and system call filtering along with secure signal handling to prevent the abuse of signals.

Compared with these, the contribution of TRUST is in demonstrating its efficient implementation of SFI and MPK, using analysis tailored for unsafe Rust and the external libraries, and identification and defense against the counterfeit stack attack.

Memory Safety of Rust Programs. Crust [97] translates Rust programs into C to verify the memory safety of unsafe code using bounded model checking. Sandcrust [59] is similar to TRUST in that both aim to protect Rust code and data from potentially harmful C libraries. With the help of the Rust macro system and existing sandboxing techniques, function calls annotated by the programmer are translated into RPCs. Like TRUST, XRust [60] and Fidelius Charm [61] aim to isolate the safe part of Rust from untrusted code. The latter uses mprotect system calls to isolate programmer-specified data from unsafe foreign libraries. Unlike Fidelius Charm, which requires invasive modification of source code, the former utilizes a separate heap allocator to isolate the safe code of Rust from unsafe heap objects without user annotation. CLA [98] further motivates TRUST to show that interfacing standalone safe Rust and C/C++ hardened against control flow hijacking can reintroduce the vulnerability. PKRU-Safe [65] uses MPK to isolate Rust from untrusted code as TRUST does. PKRU-safe associates each allocation site with a unique ID and performs profiling with the developer-provided inputs to classify the allocation sites into safe and unsafe ones. Unfortunately, this approach suffers from two drawbacks that TRUST overcomes with function cloning (see **??**) and static points-to analysis (see §3.5.1). In Rust, a handful of smart pointer

APIs produces almost all pointers, causing context-insensitive analysis to associate all such pointers with the same allocation site that has the same ID. TRUST overcomes this problem by using sound and context-sensitive static analysis to find safe allocation sites and clone functions to associate each clone with a distinct ID.

3.8 Conclusion

To our knowledge, TRUST is the first attempt to automatically quarantine from safe Rust blocks all major sources of the untrusted code, including unsafe blocks and external libraries, that undermine Rust's strong security guarantees. It is the first compiler framework that addresses the challenges of automatically identifying and protecting safe objects in a given Rust program. Thanks to this automation, unlike existing mechanisms, TRUST protects safe objects both on stack and heap without human intervention from unsafe blocks and external libraries written in unknown languages. In addition, TRUST comes with an elaborated instrumentation strategy and runtime libraries that help us to attain lower performance overhead (7.55%) than state-of-the-art (36.39%). It also defeats the counterfeit stack attack via our carefully designed gates to the external libraries. In short, TRUST is an automatic mechanism that can quarantine inclusively any untrusted code linked and integrated into safe Rust.

```
define void @"write_to_pointer_offset"(
    {i8*, i64} %sptr, i8 %idx, i8 %data)
{
    %temp = getelementptr inbounds { i8*, 
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i
      i8 %index
    store i8 %data, i8* %offset, align 4;;
    ret void
}

;;changing metadata of a smart pointer
{
    call void @change_smart_ptr_metadata(
        %smart_ptr), !UNSAFE_BLOCK
}

;;CASE 1: Use in same function as metadata
{
    %temp = getelementptr inbounds { i8*, 
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i
      i8 %index
    store i8 10, i8* %offset, align 4;;haz
}

;;CASE 2: Use in callee function
{
    call void @write_to_pointer_offset(
        %smart_ptr, i8 %index, i8 10)
}
```

(a) Original Uninstrumented Code Snippet

```
;;changes to handle CASE 2
define void @"write_to_pointer_offset"(
    {i8*, i64} %sptr, i8 %idx, i8 %data) {
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %test = call i1 @is_in_unsafe_region(i8* %ptr)
    %check = icmp eq, %test, true
    br i1 %check, label %unsafe_handle, label %safe_hand
unsafe_handle:
    %offset = getelementptr inbounds i8, i8* %ptr, i8 %i
    %bitcasted1 = bitcast i32* %offset to i32
    %masked_ptr = and i32 %bitcasted1, %UNSAFE_MASK
    %bitcasted2 = bitcast i32 %masked_ptr to i32*
    store i8 %data, i8* %bitcasted2, align 4;;safe
    br label %end
safe_handle:
    %offset = getelementptr inbounds i8, i8* %ptr, i8 %i
    store i8 %data, i8* %offset, align 4
    br label %end
end:
    ret void
}
;;changes to handle CASE 1
;;CASE 1: transformed
{
    %temp = getelementptr inbounds { i8*, i64 },
      { i8*, i64 }* %sptr, i32 0, i32 0
    %ptr = load i8*, i8** %temp, align 8
    %offset = getelementptr inbounds i8, i8* %ptr,
      i8 %index
    %bitcasted1 = bitcast i32* %offset to i32
    %masked_ptr = and i32 %bitcasted1, %UNSAFE_MASK
    %bitcasted2 = bitcast i32 %masked_ptr to i32*
    store i8 10, i8* %bitcasted2, align 4;;safe
}
```

(b) TRUST Instrumented Version

Figure 3.7: Instrumenting Pointer Offsets From Vulnerable Smart Pointer Metadata

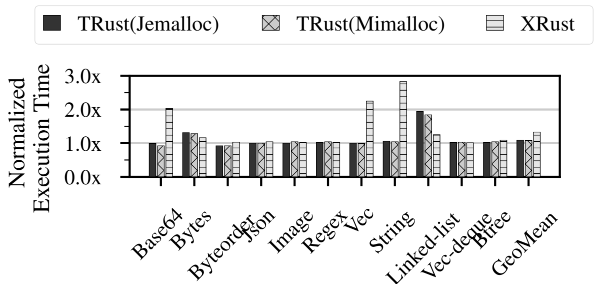

Figure 3.8: Normalized execution time of TRUST and XRust tested with the 11 widely used crates.

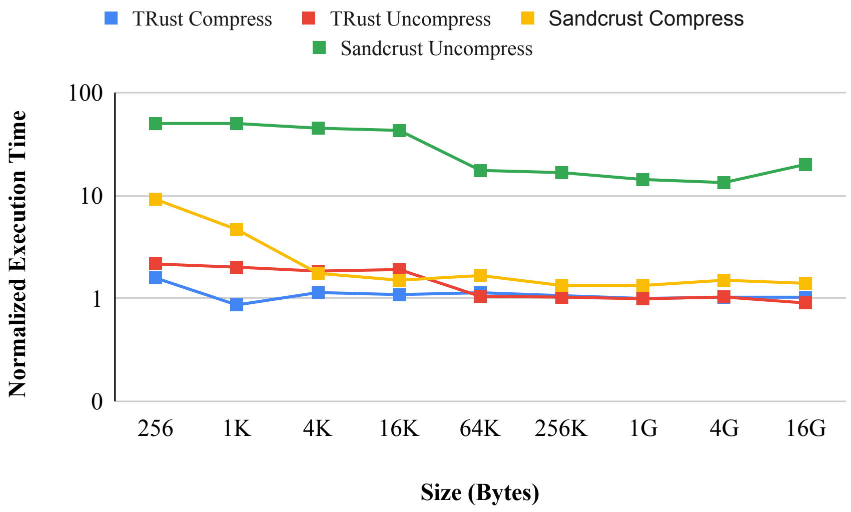

Figure 3.9: Normalized execution time of Snappy with TRUST and Sandcrust. The y-axis is log scale.

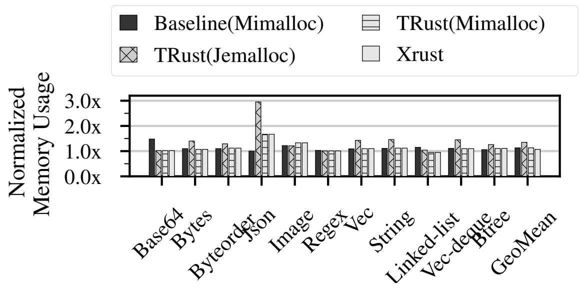

Figure 3.10: Normalized memory usage of TRUST and XRust tested with the 11 widely used crates.

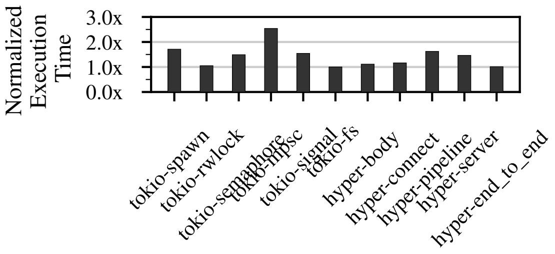

Figure 3.11: Performance Overhead of TRUST on Tokio/Hyper.

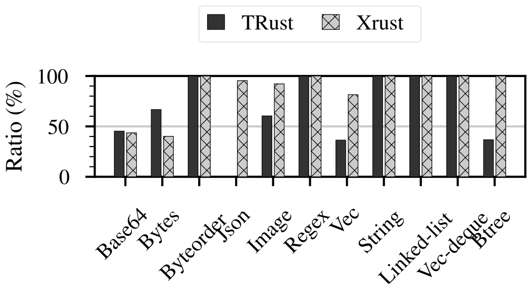

Figure 3.12: Ratio of the Number of Safe Heap Allocations.

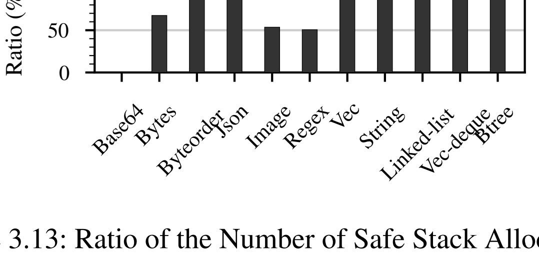

Figure 3.13: Ratio of the Number of Safe Stack Allocations.

```rust
#[inline]
fn extend_from_slice(dst: &mut Vec<u8>, src: &[u8]) {
    let dst_len = dst.len(); let src_len = src.len();
    dst.reserve(src_len);
    unsafe {
        // We would have failed if `reserve` overflowed
        dst.set_len(dst_len + src_len);XX
        ptr::copy_nonoverlapping(XX
            src.as_ptr(),
            dst.as_mut_ptr().offset(dst_len as isize),
            src_len);XX
    } }
fn write(&mut self, slice:&[u8]) -> io::Result<()> {
    extend_from_slice(&mut self.code, slice); Ok(())
}
```

Figure 3.14: Unsafe Code in the Json Library. XRust fails to observe the effects of this Code on the argument pointers.

Chapter 4

Future Work and Conclusion

In this thesis, `Vatalloc`, the method to mitigate memory safety issue in C/C++ have been introduced. Nextly, the thesis introduces Rust, memory safet language which guarnatees memory safety in lanuguage level. The thesis shows that there is a security breach even in memory safe language and proposes a TRUST to remedy it. Although TRUST provides effective isolation against the untrusted compartment in Rust program, memory safety violations can still occur within it. For example, in Figure 3.5, TRUST analyzes the threat and the variable `vec` is isolated in the unsafe memory region. However, the vulnerability still can be reproduced in the unsafe memory region, which means that the attack surface is still open to attackers. It is deemed worthwhile to undertake research on ensuring the integrity of smart pointers. Enhancing Rust memory safety by understanding its memory protection measures and protecting smart pointer metadata from corruption could be the most feasible solution in this case.

Bibliography

[1] "Cve-2021-28879 detail," https://nvd.nist.gov/vuln/detail/CVE-2021-28879, accessed: 2022-01-18.

[2] "Cve-2019-15546 detail," https://nvd.nist.gov/vuln/detail/CVE-2019-15546, accessed: 2022-01-18.

[3] "Cve-2021-28030 detail," https://nvd.nist.gov/vuln/detail/CVE-2021-28030, accessed: 2022-01-18.

[4] "Cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2024.

[5] S. Nagarakatte, J. Zhao, M. M. Martin, and S. 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, pp. 245–258.

[6] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[7] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." in *NDSS*, vol. 17, 2017, pp. 1–15.

[8] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos *et al.*, “Cheri concentrate: Practical compressed capabilities,” *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, 2019.

[9] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, “Delta pointers: Buffer overflow checks without the checks,” in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[10] J. Seo, I. Bang, Y. Cho, J. Shin, D. Hwang, D. Kwon, Y. Cho, and Y. Paek, “Exploring effective uses of the tagged memory for reducing bounds checking overheads,” *The Journal of Supercomputing*, vol. 79, no. 1, pp. 1032–1064, 2023.

[11] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Cets: compiler enforced temporal safety for c,” in *Proceedings of the 2010 international symposium on Memory management*, 2010, pp. 31–40.

[12] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing use-after-free with dangling pointers nullification,” in *NDSS*, 2015.

[13] Y. Younan, “Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers,” in *Network and Distributed System Security Symposium*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:2297018

[14] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, “Dangsan: Scalable use-after-free detection,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[15] T. H. Dang, P. Maniatis, and D. Wagner, “Oscar: A practical page-permissions-based scheme for thwarting dangling pointers,” in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 815–832.

[16] D. Liu, M. Zhang, and H. 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, pp. 1635–1648.

[17] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson *et al.*, "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 545–557.

[18] J. Shin, D. Kwon, J. Seo, Y. Cho, and Y. Paek, "Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++." in *NDSS*, 2019.

[19] "Cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2020.

[20] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[21] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[22] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 573–584.

[23] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2389–2403.

[24] N. W. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin *et al.*, "Cornucopia: Temporal safety for cheri heaps," in *2020 IEEE Symposium on Security and Privacy (SP). Los Alamitos, CA, USA: IEEE Computer Society*, 2020, pp. 1507–1524.

[25] 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)*, 2020, pp. 860–860.

[26] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 269–280.

[27] "Preventing use-after-free attacks with fast forward allocation," Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wickman

[28] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, "Dangzero: Efficient use-after-free detection via direct page table access," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1307??322. [Online]. Available: https://doi.org/10.1145/3548606.3560625

[29] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [Online]. Available: https://par.nsf.gov/biblio/10359164

[30] A. Limited., *Armv8.5-A Memory Tagging Extension White Paper*, 2021.

[31] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[32] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[33] E. A. Feustel, “On the advantages of tagged architecture,” *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 644–656, 1973.

[34] A. Limited., *Arm Architecture Reference Manual for A-profile architecture*, 2023.

[35] *Memory Tagging Extension user-space support*, 2020, https://lore.kernel.org/linux-arm-kernel/20200703153718.16973-1-catalin.marinas@arm.com.

[36] D. Lea, “A memory allocator called doug lea?셖 malloc or dlmalloc for short,” *Available online [March 26, 2010]: http://gee.cs.oswego.edu/dl/html/malloc.html*, 1996.

[37] W. Gloger, “Ptmalloc,” *Consulté sur http://www.malloc.de/en*, 2006.

[38] J. Evans, “Scalable memory allocation using jemalloc,” engineering at Meta.

[39] P. Akritidis, M. Costa, M. Castro, and S. Hand, “Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors.” in *USENIX Security Symposium*, 2009, pp. 51–66.

[40] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. Van Der Kouwe, “Typesan: Practical type confusion detection,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 517–528.

[41] “Odroid-hc4,” https://www.hardkernel.com/ko/shop/odroid-hc4, (accessed July 2021).

[42] C. O. M. C. H. S. M. C. M. P. E. H. O. M. L. L. N. B. M. L. L. Derrick McKee (Purdue University), Yianni Giannaris (MIT CSAIL), in *Preventing Kernel Hacks with HAKCs (NDSS)*.

[43] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "Mtsan: A feasible and practical memory sanitizer for fuzzing cots binaries," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[44] "Cve-2022-34568 detail," https://nvd.nist.gov/vuln/detail/CVE-2022-34568, nVD Published Date: 2022-07-28.

[45] "Cve-2022-40674 detail," https://nvd.nist.gov/vuln/detail/CVE-2022-40674, nVD Published Date: 2022-09-14.

[46] "Cve-2022-3352 detail," https://nvd.nist.gov/vuln/detail/CVE-2022-3352, nVD Published Date: 2022-09-29.

[47] "Cve-2022-30065 detail," https://nvd.nist.gov/vuln/detail/CVE-2022-30065, nVD Published Date: 2022-05-18.

[48] "Cve-2022-36149 detail," https://nvd.nist.gov/vuln/detail/CVE-2022-36149, nVD Published Date: 2022-05-18.

[49] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[50] "Arm morello program," https://developer.arm.com/architectures/cpu-architecture/a-profile/morello, 2020.

[51] "Softboundcets for llvm+clang version 34," https://github.com/santoshn/softboundcets-34, 2014 (accessed April 21, 2020).

[52] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, ser. PLDI ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 158??68. [Online]. Available: https://doi.org/10.1145/1133981.1134000

[53] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The cheri capability model: Revisiting risc in an age of risk,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 457–468.

[54] T. Zhang, D. Lee, and C. Jung, “Bogo: buy spatial memory safety, get temporal memory safety (almost) free,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 631–644.

[55] N. D. Matsakis and F. S. Klock, “The rust language,” in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103??04. [Online]. Available: https://doi.org/10.1145/2663171.2663188

[56] T. R. team, *The Rust programming language*, 2017. [Online]. Available: https://www.rust-lang.org/

[57] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “Rustbelt: Securing the foundations of the rust programming language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3158154

[58] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, “Towards memory safe enclave programming with rust-sgx,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2333??350. [Online]. Available: https://doi.org/10.1145/3319535.3354241

[59] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, “Sandcrust: Automatic sandboxing of unsafe components in rust,” in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, ser. PLOS’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51??7. [Online]. Available: https://doi.org/10.1145/3144555.3144562

[60] P. Liu, G. Zhao, and J. Huang, “Securing unsafe rust programs with xrust,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 234??45. [Online]. Available: https://doi.org/10.1145/3377811.3380325

[61] H. M. J. Almohri and D. Evans, “Fidelius charm: Isolating unsafe rust code,” in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 248??55. [Online]. Available: https://doi.org/10.1145/3176258.3176330

[62] “Smart pointers,” https://doc.rust-lang.org/book/ch15-00-smart-pointers.html, accessed: 2022-01-19.

[63] M. A. Finlayson, *Intel쬐 64 and IA-32 architectures software developer?셲 manual*, 2020.

[64] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, “libmpk: Software abstraction for intel memory protection keys (intel MPK),” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[65] 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*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132??48. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[66] 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)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[67] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-Process isolation for High-Throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[68] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 143??56. [Online]. Available: https://doi.org/10.1145/3381052.3381326

[69] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "Pku pitfalls: Attacks on pku-based memory isolation systems," in *USENIX Security Symposium*, 2020.

[70] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[71] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No Need to Hide: Protecting Safe Regions on Commodity Hardware," in *EuroSys*, Apr. 2017.

[72] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient In-Process isolation for RISC-V and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[73] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 56–71.

[74] "parasol-aser/xrust," https://github.com/parasol-aser/XRust, accessed: 2022-01-18.

[75] "Cve-2021-28028 detail," https://nvd.nist.gov/vuln/detail/CVE-2021-28028, accessed: 2022-01-18.

[76] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 299??08. [Online]. Available: https://doi.org/10.1145/2420950.2420995

[77] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265??66. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[78] ——, "Demand-driven pointer analysis with strong updates via value-flow refinement," *CoRR*, vol. abs/1701.05650, 2017. [Online]. Available: http://arxiv.org/abs/1701.05650

[79] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: Fully verified software fault isolation," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, 2011, pp. 289–298.

[80] L. Deng, Q. Zeng, and Y. Liu, "Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries," in *IFIP International Information Security Conference*, 2015.

[81] D. Leijen, B. G. Zorn, and L. M. de Moura, "Mimalloc: Free list sharding in action," in *APLAS*, 2019.

[82] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 936–952. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[83] "Snappy, a fast compressor/decompressor." =https://github.com/google/snappy.

[84] "Build reliable network applications without compromising speed." https://tokio.rs, accessed: 2022-01-18.

[85] "hyper: Fast and safe http for the rust language." https://hyper.rs, accessed: 2022-01-18.

[86] "Cve-2021-29939 detail," https://nvd.nist.gov/vuln/detail/CVE-2021-29939, accessed: 2022-01-18.

[87] "Memory safety issue in stackvec::extend," https://github.com/Alexhuszagh/rust-stackvector/issues/2, accessed: 2022-01-18.

[88] "Cve-2021-45707 detail," https://nvd.nist.gov/vuln/detail/CVE-2021-45707, accessed: 2022-01-18.

[89] "Cve-2018-1000657 detail," https://nvd.nist.gov/vuln/detail/CVE-2018-1000657, accessed: 2022-01-18.

[90] "Cve-2018-1000810 detail," https://nvd.nist.gov/vuln/detail/CVE-2018-1000810, accessed: 2022-01-18.

[91] G. Novark and E. D. Berger, "Dieharder: Securing the heap," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, ser. WOOT'11. USA: USENIX Association, 2011, p. 12.

[92] R. Mirzazade Farkhani, M. Ahmadi, and L. Lu, "Ptauth: Temporal memory safety via robust points-to authentication," 01 2020.

[93] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 517??28. [Online]. Available: https://doi.org/10.1145/2976749.2978405

[94] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 203??16. [Online]. Available: https://doi.org/10.1145/168619.168635

[95] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93.

[96] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of*

the Seventeenth European Conference on Computer Systems, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266??82. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[97] J. Toman, S. Pernsteiner, and E. Torlak, "Crust: A bounded verifier for rust (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 75–80.

[98] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *NDSS*, 01 2022.

초록

소프트웨어 보안의 중요성은 그 어느 때보다 중요하다. 그러나, 복잡한 소프트웨어 시스템에서 메모리 안전을 효율적으로 도모하는 문제는 여전히 해결되지 않고 있다. 근원적인 이유는 기존의 코드베이스들이 비안전 언어인 C/C++를 기반으로 하기 때문이다. 연구자들이 수 십년간 메모리 안전 문제에 대해 해결책을 제시했으나, 여전히 메모리 오염은 보안 공격의 주요 벡터이다. 본지에서는 메모리안전을 시간적 측면에서 보호하는 방안을 제시한다. 제시한 솔루션을 포함하여, 많은 공격 완화 기술들이 있지만, 소프트웨어 기반 기법은 부하를 수반하며, 하드웨어 기반 기법은 사용성이 제한된다는 단점을 가지고 있다. 이러한 맥락에서 보안성과 효율성 모두를 염두에 둔 Rust 언어가 개발되었다. 엄격한 타입 시스템과 소유권 모델을 비롯한 Rust의 메모리 정책에 힘입어 메모리 안전이 언어 수준에서 보장되며, 성능 또한 보안적으로 최적화된 C/C++보다 빠르다. 그러나 여전히 보안 위협이 있음이 밝혀졌으며, 이를 해결하기 위해 본지는 효율적이고 안전한 프로세스 내 격리를 이용하여 체크되지 않은 코드로 부터 프로그램을 보호하는 컴파일 프레임워크를 제안한다. 그럼에도 공격 공간을 줄이지는 못하므로, 이 문제를 해결하는 방안을 모색하며 본 논문을 마무리한다.

주요어: 프로그래밍 언어, 메모리 안전, 보안, 컴파일러, 격리

학번: 2017-20125