Securing UnSafe Rust Programs with XRust

Peiming Liu
peiming@tamu.edu
Texas A&M University
College Station, Texas, U.S.A

Gang Zhao
zhaogang92@tamu.edu
Texas A&M University
College Station, Texas, U.S.A

Jeff Huang
jeff@cse.tamu.edu
Texas A&M University
College Station, Texas, U.S.A

ABSTRACT

Rust is a promising systems programming language that embraces both high-level memory safety and low-level resource manipulation. However, the dark side of Rust, *unsafe Rust*, leaves a large security hole as it bypasses the Rust type system in order to support low-level operations. Recently, several real-world memory corruption vulnerabilities have been discovered in Rust's standard libraries.

We present XRust, a new technique that mitigates the security threat of unsafe Rust by ensuring the integrity of data flow from unsafe Rust code to safe Rust code. The cornerstone of XRust is a novel heap allocator that isolates the memory of unsafe Rust from that accessed only in safe Rust, and prevents any cross-region memory corruption. Our design of XRust supports both single- and multi-threaded Rust programs. Our extensive experiments on real-world Rust applications and standard libraries show that XRust is both highly efficient and effective in practice.

1 INTRODUCTION

Long-existing system programming languages such as C/C++ offer programmers the ability to manipulate low-level resources but in error-prone ways. Countless severe bugs have been found due to the unsafe nature of these languages [8, 17, 32]. Rust [23] is a rising language that tries to bridge the gap between memory safety and low-level system programming. With new language features such as ownership, borrowing, and lifetime, Rust guarantees a program to be memory safe if it could be compiled (at the absence of *unsafe Rust*). The type system of Rust and its encapsulation on low-level operations have been formally proved to ensure memory safety [22, 33].

However, the static restrictions of Rust can be too strict to admit many valid programs due to reasons including (1) by nature, static analysis is conservative and (2) the underlying computer hardware is inherently unsafe and certain operations could not be done with safe Rust [10]. This problem is addressed by *unsafe Rust*, which escapes from the static checks [36]. With unsafe Rust, programmers are able to manipulate raw pointers, perform unprotected type casting and other dangerous operations just like in C/C++. Therefore, a Rust program is free of memory errors only when its unsafe code is correctly implemented and does not violate memory safety properties [22]. However, requiring all the unsafe Rust code to be correctly implemented is difficult. Bugs in unsafe Rust code may result in severe vulnerabilities, as witnessed by several memory errors discovered recently [5, 37, 38]. What is worse is that a memory error in unsafe Rust may corrupt arbitrary data in the whole address space, *i.e.,* bugs in unsafe Rust can be exploited to hijack function pointers or steal sensitive data in safe Rust.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380325

Table 1: Unsafe Rust code in practice (Rust-lang contains the code for Rust compiler and all the Rust standard libraries).

	LoC	LoC (unsafe)	unsafe %
collected crates	2,480,761	18,490	0.75%
Rust-lang	327,792	3,163	0.96%

To understand how the unsafe portion of Rust is used in real-world applications, we randomly selected 500 crates from *crates.io* and counted the number of lines of unsafe code (shown in Table 1). The result indicates that most real-world Rust programs only rely on a very small fraction of unsafe code (< 1%) on average. Although in practice most memory objects in Rust are statically protected by Rust's type system, a bug residing in unsafe Rust code could simply ruin the entire effort and put the whole program at the risk of being attacked!

In this paper, we present XRust, a novel approach to mitigate the security threat brought by unsafe Rust while imposing minimal overhead to Rust programs. While there exist several prior attempts [7, 21, 28–30] on C/C++ to retrofit full memory safety of the language (which is often expensive), our goal is not to bring memory safety to unsafe Rust, but to ensure the integrity of data in safe Rust (at the presence of memory errors in unsafe Rust code). In XRust, the heap is logically divided into two mutually exclusive regions: an unsafe region and a safe region. Memory objects created and/or accessed by unsafe Rust (referred to as *unsafe objects*) are placed in the unsafe region and can be corrupted. All other *safe objects* are stored in the safe region and can never be corrupted. The separation between safe and unsafe objects can be enforced by in-process memory isolation techniques [9, 48]. In this work, we explore two methods using instrumentation and memory guard pages to achieve in-process isolation.

As depicted in Figure 1, XRust works as follows:

(1) In the original code, the two objects `buf` and `password` are treated equally and are placed in the same heap region. A heap-based attack exploiting a memory corruption of `buf` in unsafe Rust code can cause arbitrary write to the whole address space, including corrupting `password`;

(2) In the XRust-protected code, `buf` is placed in the unsafe region separated from `password`, because `buf` is used in unsafe Rust. When using instrumentation, runtime checks are inserted to prevent cross-region data flows from the

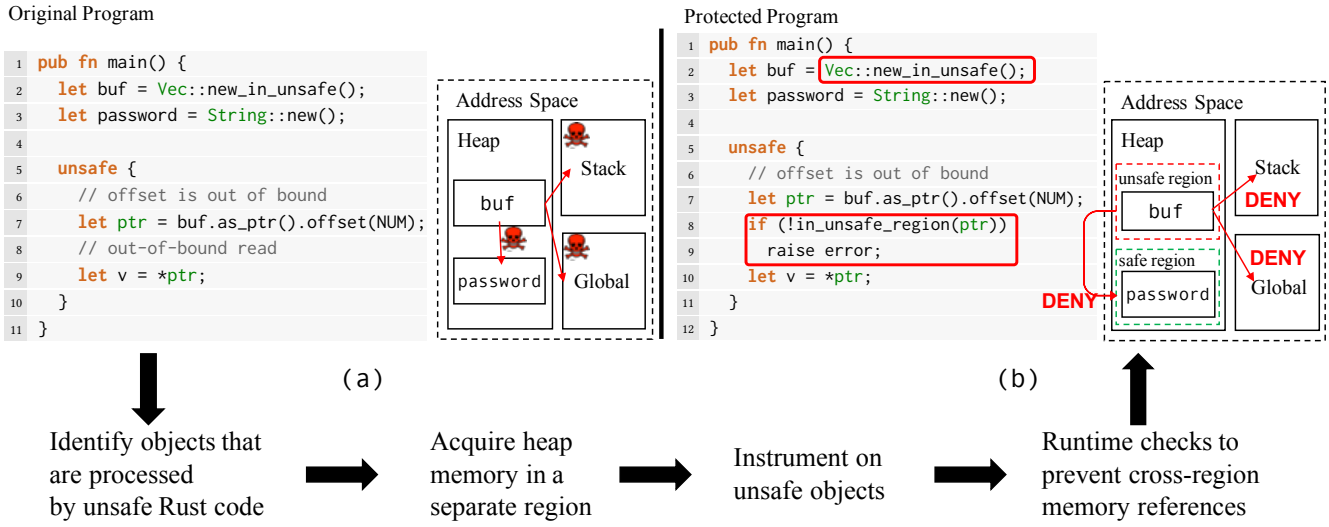

Figure 1: A technical overview of XRust (using instrumentation-based memory isolation).

unsafe region to the safe region. When using guard pages, isolation is enforced by placing inaccessible memory pages between the two regions.

We note that XRust does not attempt to guarantee full memory safety of unsafe Rust, but the safety of memory objects in safe Rust. The main goal of XRust is to provide effective protection while imposing negligible overhead. Also, XRust only targets memory corruption on heap objects. Stack protection techniques such as stack canaries [14] and SafeStack [24] have been deployed widely in real systems. Proposals [20, 35] to support SafeStack in Rust have also been implemented.

To our knowledge, XRust is the first attempt to isolate the side effect of unsafe Rust automatically and it achieves both effectiveness and efficiency by leveraging unique language features of Rust. The design of a new type of multi-region heap allocator is seamlessly incorporated into the Rust framework while achieving backwards compatibility. To support a separate memory region in XRust is also challenging and we made extensive modifications in the Rust compiler ranging from high-level language features to the low-level heap allocator. In summary, we highlight our contributions as follows:

- XRust is the first approach to automatically protect safe Rust from memory corruption errors in unsafe Rust. A recent work, Fidelius Charm [2], shares a similar goal but it requires programmers to mark and restore unsafe data before and after unsafe code blocks at memory page level. More importantly, FC is limited usability when handling *shared* unsafe objects in safe Rust code as discussed in Section 5.1.1.
- We design and implement a novel heap allocator that supports safe and unsafe memory regions, and efficiently checks cross-region references using instrumentation or guard pages.
- We evaluate XRust extensively on real-world Rust applications and memory errors. Our result shows that XRust incurs 0.15% median overhead on tested crates (2.8% on Rust standard libraries) and it effectively defends against attacks that exploit known real-world memory vulnerabilities in Rust.

2 OVERVIEW

In this section, we first discuss the rationale behind the design of XRust. We then illustrate how XRust works on a motivating example based on a real vulnerability in Rust.

2.1 Why XRust?

The clear separation between safe and unsafe Rust naturally divides objects into two *mutually exclusive* sets: The sets of *safe* and *unsafe* objects, based on whether they are used in unsafe Rust. At a high-level view, since only unsafe objects are under the risk of being corrupted in Rust programs, the isolation enabled by XRust between memory regions used by safe and unsafe objects ensures that potential memory corruptions can only impact the unsafe region and can never cross the boundary to corrupt safe objects.

In this subsection, we first discuss how unsafe Rust is used in practice, and then discuss the protection strength of XRust with respect to both *spatial* and *temporal* memory safety.

2.1.1 Unsafe Rust in practice. We studied several popular open-source Rust projects as well as the Rust standard library to understand the usage of unsafe Rust in the real world.

As summarized in Table 1, Rust programs only contain less than 1% unsafe code on average, and unsafe Rust is typically used only for low-level operations and optimizations. The statistics provide strong evidence that most objects are only processed by safe Rust and by isolating the side-effect of unsafe Rust, XRust is able to protect all of them. Apart from this, we also conducted in-depth inspections of the source code on the usage of unsafe Rust (related to memory safety). We summarize our findings into three categories:

Unbounded Memory Accesses. Instead of using object references, programmers sometimes use raw pointers and unchecked pointer arithmetic to access a piece of consecutive memory. *E.g.,* in `base64`, instead of using a `vector`, the developers access the encoding buffer directly through a raw pointer and iterate over the memory by adding offsets to the pointer. This pattern is normally used to access an internal buffer and to skip default bound checkings (in `image`, `base64`, `vec`, etc)

Unchecked Conversions. This includes both *type* conversion as well as *data format* conversion (*e.g., utf-8* to *utf-16*). This is mainly used for developing low-level functionalities such as decoding/encoding binary data and serialization as in `string`, `byteorder`, `bytes`, etc.

Internal States Override. When using well-encapsulated safe APIs from Rust libraries, the internal states of an object is normally maintained internally by Rust (*e.g.,* , pushing an element into a vector increases the size of the vector). However, when developers access an object in unexpected ways, the internal states need to be manually adjusted. *E.g.,* after initializing the buffer of a vector using raw pointers, the size of the vector needs to be overridden accordingly. The operation is unsafe as programmers are responsible to provide the correct value and unmatched internal states may lead to undefined behaviors. This is typically used for the purpose of low-level optimizations as in `vecdeque`, `vec`, etc.

2.1.2 Observations behind XRust. Based on the empirical studies above, we make two observations:

Observation #1: Being aware that the unsafe Rust code is not checked by the compiler, Rust programmers tend to avoid heavy usage of unsafe Rust in practice and only rely on unsafe features to perform necessary low-level operations [41]. This indicates that in reality, it is likely that most objects in a Rust application are safe objects, and critical data such as `password` (with high-level semantics) is unlikely to be processed in unsafe Rust.

Apart from this, we make the second observation based on Rust's object memory model that indicates how indirect calls, which is essential to control flow integrity, is handled by Rust.

Observation #2: Unlike C++ which stores the *virtual function table* (vtable) pointers of an object adjacent to its data members [26], Rust stores them separately. Internally, Rust achieves polymorphism and dynamic dispatching by transforming objects into *trait objects* [34]. As illustrated in Figure 2, the reference to a trait object is a *fat pointer* consisting of two pointers: one points to the data members of the object and the other points to the vtable. This implicitly puts the heap data and vtable pointers into two regions. For unsafe objects, only its data members are allocated in the unsafe heap region. Thus, overflow to corrupt vtable pointers is a cross-region reference and will be prevented by XRust.

2.1.3 Protection Strength of XRust. The two observations above lead to the following properties of XRust:

Spatial Memory Safety The observations imply that by preventing cross-region references, XRust can efficiently defend Rust programs against:

(1) Non-control data attacks in unsafe Rust code that corrupt objects outside the unsafe region;

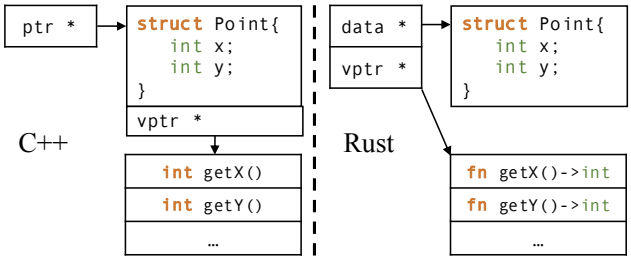

Figure 2: Memory layout of objects in C++ vs Rust.

(2) Control-oriented attacks in unsafe Rust code that corrupt the vtable pointer of a trait object or raw function pointers outside the unsafe region, *e.g.,* to hijack control flow to malicious code.

These protections are valuable in practice because (1) there is a high chance that most sensitive data in Rust user applications are safe objects (observation #1), and (2) vtable pointers of trait objects are the major source of indirect jumps in Rust and they are protected by XRust (observation #2).

Temporal Memory Safety. XRust is able to prevent temporal memory errors from corrupting safe objects as well. In Rust, temporal errors can only happen on unsafe objects because safe Rust code statically eliminates all temporal errors by analyzing the *lifetime* of references and the *ownerships* of objects. So, when a temporal error (*e.g.,* use after free) occurs, the pointer used to access memory must point to an unsafe object. Since our multi-region allocator will not reuse memory previously used for unsafe objects to allocate any safe object (Section 4.2), the freed memory of an unsafe object will only be used to hold another unsafe object. When temporal errors occur, memory accesses on the freed pointer will still be within unsafe heap region so that the temporal errors can not escape the unsafe region to corrupt safe objects.

2.2 A Motivating Example

Listing 1 shows a code fragment simplified from the `rust-base64` library. For versions before 0.5.1, the library contains an integer overflow bug that eventually leads to a heap buffer overflow. On line 4, the vulnerable function first tries to reserve a buffer on the heap and the size of the buffer is calculated by the vulnerable function `encoded_size` that contains a integer overflow error.[1] A heap overflow can happen when the integer overflow leads to a smaller buffer and this vulnerability can be exploited to overwrite data in safe Rust. For Rust applications depending on this library, the unsafe code may only account for a small fraction of the entire code. However, this bug can still result in memory corruptions in the entire address space.

XRust significantly mitigates this vulnerability. It first identifies `buf` as an unsafe object because it is used in unsafe Rust (line 9), by analyzing the data flow from the safe Rust to unsafe Rust. Then instead of reserving heap memory for the objects normally (line 4), it reserves the memory for `buf` in the unsafe region, by rewriting

[1]Note that Rust does check integer overflows for the debugging build by default, but not in the optimized release build.

```
1  pub fn encode_config_buf<T>(buf: &
2    // reserve a large enough buffer
3    // store the encoded string
4    buf.reserve(encoded_size(len));
5    // using unsafe operation to sto
6    // string to buffer
7    unsafe {
8      // buf object is used in unsaf
9      let mut output_ptr = buf.as_mu
10     while condition {
11       // do pointer arithmetic and
12       // memory directly
13       ptr::write(output_ptr.offset
14       ...
15     }
16   }
17 }
```

Listing 1: A real buffer overflow in `rust-base64` due to unsafe Rust code (CVE-2017-1000430).

the function to call an extended API. Finally, accesses to `buf`, which is an unsafe object, are restricted to be within the unsafe memory region. When using instrumentation, the memory reference on line 13 will be instrumented as follows:

```
1 let ptr = output_ptr.offset(..);
2 if (!in_unsafe_region(ptr))
3   raise error;
4
5 write(ptr, ...);
```

At runtime, attempts to access addresses outside the unsafe heap region are detected by XRust, thus the vulnerability cannot be exploited to perform attacks on safe objects.

We observe that, even with instrumentation which often imposes high overhead for other languages such as C/C++ by other techniques, XRust is still fast (3.6% overhead on median). This is because XRust only checks memory references on unsafe objects, which avoids heavy instrumentation to propagate the meta information as required by techniques such as SoftBound [28], and it avoids the expensive whole-program reaching definition analysis as required by DFI [7] to determine valid data flows. Moreover, XRust checks only cross-region data flow (rather than object bounds), which can be achieved in constant time with the help of our allocator (Section 4.2.1). In our design, we also leverage guard pages to protect cross-region references (Section 5.2), which is even more efficient than using instrumentation.

Figure 3 shows the technical design of XRust, which consists of three key components: 1) extensions made to Rust and the Rust compiler to provide high-level APIs for allocating objects in the unsafe regions; 2) a new heap allocator that supports an unsafe heap region; and 3) runtime protections to prevent cross-region memory references. In the next three sections, we present the details of each component.

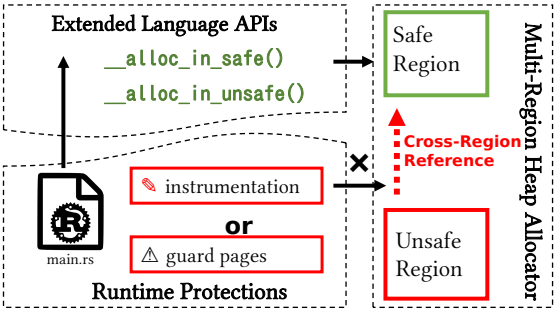

Figure 3: Three key components of XRust.

3 LANGUAGE EXTENSIONS

In this section, we first introduce necessary background on how Rust encapsulates its heap allocation interfaces and then present our extensions.

3.1 Heap Allocation in Rust

Instead of allowing programmers to acquire and release heap memory directly through `malloc` and `free`, Rust provides high-level abstractions on heap memory through encapsulation on heap operations. The release of a heap object is automatically inserted by Rust compiler and programmers are not allowed to free the memory manually to avoid errors like double frees. It also gives Rust the flexibility of changing the allocator globally (even for pre-compiled libraries) without recompiling the code by defining a *global allocator* [2] [15]. These encapsulations and the loose connection between the language and the allocator implementation require extra abstraction layers between these two components.

There are two ways to acquire a piece of heap memory in Rust[3]. In most cases, this can be achieved by creating a `Box<T>` object. For low-level library developers, it could be done by directly interacting with the `Alloc` *trait* (trait is similar to Java's *interface*). The `Box<T>` objects are wrapped pointers that can only point to heap objects and are internally created using `box` expressions[4]. For example, the expression `box 42` allocates four-byte heap memory that stores a 32 bit integer of value 42, and it returns a `Box<i32>` object pointing to the allocated heap object as the result. Those `Box<T>` objects will be dropped later by the compiler-inserted code when their owners go out the scope, *i.e.,* the owner function returns or the owner block terminates. In Rust's standard libraries, neither `box` expressions nor the default implementation of the `Alloc` trait is bounded to a specific allocator. They both rely on the Rust compiler to generate glue code to bind the program to a specific allocator during code generation phase.

For heap allocation through the `Alloc` trait, the default implementation delegates all its tasks to a set of functions with the `__rust` prefix. Specifically, `__rust_alloc()` for heap allocation, `__rust_dealloc()` for heap deallocation, and `__rust_realloc()`

[2]The feature of switching allocators globally is not in a stable state yet. The description in this paper is based on the latest Rust (version 1.32) by the time of writing.
[3]Calling malloc-like function through FFI is out of the scope.
[4]box expression is an unstable feature as well.

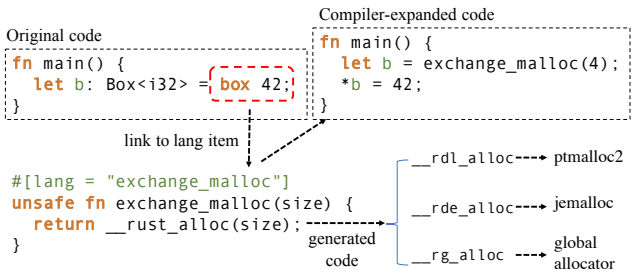

Figure 4: Rust workflow for linking heap allocations.

for heap reallocation, etc. These functions do not have actual implementations, but are treated as special internal symbols by the Rust compiler and implemented by compiler generated code to invoke different allocators, *e.g.,* the allocator for static libraries and for executable binaries.

For heap allocation through `box` expressions, it requires two *lang items*: "`exchange_malloc`" for allocation and "`box_free`" for deallocation. Lang items [45] are pluggable features in Rust whose functionalities are not hard-coded into the language but are implemented in libraries, using a special marker (`#[lang = "..."]`) to indicate their existence. Figure 4 illustrates the workflow. At compile time, for each `box` expression, the Rust compiler searches all the dependent libraries to find functions marked by these two lang items. The compiler then generates code by calling the function marked as `exchange_malloc` to allocate heap memory, and inserts calls to the function marked as `box_free` to drop `Box<T>` objects. In Rust's standard libraries, the default implementation of `exchange_malloc` delegates heap allocation to `__rust_alloc()`.

3.2 Language Support for Unsafe Region

To support a different heap region, we add corresponding "`unsafe`" interfaces for each of the allocation functions. For example, we add `__rust_unsafe_alloc` as the entry point for allocating heap memory in the unsafe region. The compiler is also extended to generate code to invoke these extended functions for handling the unsafe heap region.

We then build high-level APIs for the extended interfaces by extending Rust's standard library. Additional methods are added to the `Alloc` and `GlobalAlloc` traits to deal with the unsafe heap region. For example, the function `unsafe_alloc()` is added to the `Alloc` trait to provide interfaces for allocating memory in the unsafe region. Based on this, high-level classes in the standard libraries can be extended as well. For example, `Vec::unsafe_with_capacity()` is added to the `Vec` structure to create a vector that puts the internal memory buffer in the unsafe heap region, which allows programmer to interact with unsafe allocation interfaces on their own demands.

The newly added interfaces are backward compatible with existing Rust programs. By default, calls to the extended interfaces (*e.g.,* `unsafe_alloc()`) are delegated to the pre-existing functions (*e.g.,* `alloc()`). The compiler-generated code also delegates the requests from `__rust_unsafe_alloc` to the standard API if the underlying allocator does not support a separate unsafe region. In this way, all existing Rust programs can be compiled without modification. When programmers use the extended interfaces but with an allocator that does not support the unsafe region, the allocation can still be completed, but the allocated heap chunks will not be placed in a separate unsafe heap region. The default implementation is then overridden by our extended allocator and linked properly by the compiler. Invocations on them are passed to the proper API to allocate and free heap memory in the unsafe region.

For `box` expressions, we add a new operator `unsafe_box` to create a `Box` object in the unsafe heap region. The grammar of `unsafe_box` expressions is identical to `box` expressions, and the result of `unsafe_box` expressions has the same type (`Box<T>`) as the result of `box` expressions. Generating the same type ensures that the `unsafe_box` operator can fit into the existing Rust type system. The only difference between `Box` objects created by `box` and `unsafe_box` expressions is that internally they are put into different heap regions, but all other operations (dereference, type casting, pattern matching, etc.) are identical. Similarly, `unsafe_box` will be linked to a new lang item `unsafe_exchange_malloc` at compile time, which handles the allocation of unsafe objects.

4 MULTI-REGION HEAP ALLOCATOR

Our allocator implementation is based on `ptmalloc2` [18] and supports multi-threading. We first introduce the architecture of `ptmalloc2` and then present our extensions.

4.1 Architecture of ptmalloc2

`ptmalloc2` was forked from `dlmalloc` [25] and later merged into `glibc` with threading support. `ptmalloc2` maintains separate heap segments and freelist data structures using multiple *per-thread arenas*, such that threads can rely on different arenas to perform heap allocation/deallocation simultaneously without synchronization.

In `ptmalloc2`, each arena can manage a list of heap segments (except for the main arena, which only has one heap segment). A heap segment is a large piece of mmapped memory from where free chunks are retrieved and returned to users. Arenas also keep the freelist data structures of their heap segments (*i.e., bins*) used to hold free chunks. Bins are divided into four different types based on chunk sizes: *fast*, *unsorted*, *small*, and *large*, and each is handled differently. To handle a heap allocation, `ptmalloc2` chooses the appropriate bins based on the requested size. More details can be found in [42].

4.2 XRust Extensions on ptmalloc2

In our design to extend `ptmalloc2` for handling heap (de)allocation in the unsafe region, we followed most of its current design. The interactions with the unsafe heap region are achieved through extended APIs such as `unsafe_malloc()`. These APIs are merged into Rust and linked with extended language APIs. The data structures used by the unsafe region are lazily initialized upon the first request for allocating memory in the unsafe region. For applications that do not use the unsafe region, the extended allocator acts the same as unmodified `ptmalloc2` and no overhead is imposed.

4.2.1 Unsafe Region in Heap. In our heap allocator, the set of *arenas* for handling allocations in unsafe region and those for allocations in safe region are disjoint, *i.e.,* the unsafe arenas will not be reused

for allocating objects in safe region and *vice versa*. This ensures that for every internal heap segment managed by the allocator, it only contains the objects in the same region so that overflow originated from unsafe objects will not corrupt safe objects.

We extend the architecture of `ptmalloc2` to enable fast checks on cross-region errors. Intuitively, cross-region references can be checked by determining whether the pointer is within the range of any unsafe heap segment. This could lead to huge runtime overhead since the number of unsafe heap segments is unbounded (especially for multi-thread programs). To address the issue, we use a pre-allocated bitmap to record the type of heap segments (*safe* or *unsafe*), which can be quickly indexed by the start addresses of heap segments. This introduces negligible memory overhead since the heap segments in `ptmalloc2` is 1 megabytes aligned by default, thus the memory overhead is 1 bit *per* megabyte. The bitmap is protected by `PROT_READ` and can only be accessed inside the allocator upon the creation of an unsafe heap segment. Under this design, checking an memory reference takes only constant time regardless how many unsafe heap segments have been allocated.

4.2.2 Multi-thread Support. To maximize the performance of multi-thread programs, we adopt the *per-thread arena* mechanism to allow accessing the free lists for unsafe heap region concurrently. For multi-thread programs, threads are assigned with different arenas to allocate heap memory. Since every arena manages a disjoint set of heap segments, they can be accessed concurrently without synchronization.

Our design of the multi-region heap allocator also renders time-of-check-to-time-of-use (TOCTTOU) attacks almost impossible. To trigger such an attack, an unsafe pointer needs to be verified to be within unsafe region first (time of check) and later be used to corrupt a safe object (time of use) because the unsafe object is first freed and the same address is reused for a safe object by other threads before the time of use. However, since unsafe heap segments are maintained separately by different arenas in our allocator, a freed unsafe chunk will only be reused to hold another unsafe object, which makes the attack difficult. Besides, Rust prohibits programmer from calling `drop` manually to deallocate objects to avoid errors, which makes it even harder to launch the attack.

4.2.3 Cross-Region References inside Allocator. To fully prevent cross-region memory references, the allocator need to be free of cross-region errors as well. For example, memory errors can be exploited to corrupt the metadata of heap chunks because `ptmalloc2` stores the metadata adjacent to user data [3].

To address this problem, we insert runtime checks to ensure that the unsafe region inside the allocator would never be able to reference data outside the region. For example, the free chunks in the unsafe region would only be linked to other free chunks within the unsafe region. Whenever the allocator attempts to access the metadata of chunks in the unsafe region, checks are added to ensure that the allocator can never perform cross-region references based on corrupted data.

5 CROSS-REGION REFERENCE PREVENTION

Cross-region memory references can be prevented by in-process memory isolation techniques, which have already been widely studied. It has been shown that isolation can be enforced with negligible overhead through hardware-based protection, *e.g.,* by using Intel MPK [48] or ARM memory domain [9, 16]. In our prototype implementation, we explored two schemes to detect cross-region references. The first one is to instrument memory references on unsafe objects; the second is to utilize memory protection pages (*i.e.,* guard pages) to detect overflows

We explain how cross-region references are prevented by instrumentation as well as by memory guard page in Section 5.1 and Section 5.2, respectively.

5.1 Code Instrumentation

We first perform an inter-procedural data flow analysis to identify the allocation sites of unsafe objects in Rust programs, based on a recent data flow framework [31]. Any allocated object that is later accessed by unsafe code is considered as an unsafe object. Every allocation site is a taint source, and every unsafe instruction is a taint sink. We record every object that flows from a source to a sink. Based on the results, we rewrite the program to allocate objects in the unsafe memory region.

5.1.1 Shared Unsafe Objects in Safe Rust. We also revealed a crucial technical caveat during the process of developing XRust: To completely isolate the side effect of unsafe Rust code, the instrumentation should be applied not only on unsafe Rust code but *all unsafe objects*, which means that not only the data directly touched by unsafe code, but also everything transitively reachable from such data need to be instrumented.

Consider the following code that creates a vector of length 3 on line 1 and calls an unsafe function `set_len()` on line 3 to set the length of the vector to 10 manually (without resizing the buffer).

```
1 let v = vec![1,2,3];
2 unsafe {
3   v.set_len(10);
4 }
5 let elem = v[9];
```

The memory reference on line 5 is an out-of-bound read because the vector has only allocated the memory space for storing three integers. The code above passes the Rust compiler because the vector length is changed by unsafe code. Moreover, no exception will be thrown at runtime by the assertion inserted for the memory reference on line 5, which only checks if the vector index is less than the vector length.

This is an example of how unsafe Rust can be used to override the internal states of an object, and it can lead to a memory corruption outside unsafe Rust (but on unsafe objects). Therefore, to provide complete protection, all memory references on unsafe objects (inside or outside unsafe Rust) need to be checked. In fact, one of the real vulnerabilities in Rust (`VecDeque`, see Section 6.4.1) belongs to this category. This also indicates that one of the related works, FC [2], fails to provide a complete isolation from unsafe Rust code as it only protects unsafe Rust code.

For instrumentation, we apply a context-insensitive pointer analysis (using SVF [43]) to identify memory references on unsafe **objects**. Since static pointer analysis is conservative, the points-to set of a pointer can contains both safe and unsafe objects. To address

```
let mut ptr;
if (condition) {
  ptr = __rust_alloc();
  shadow[ptr] = SAFE;
} else {
  ptr = __rust_unsafe_alloc();
  shadow[ptr] = UNSAFE;
}
if (shadow[ptr] == UNSAFE) {
  if(!in_unsafe_region(ptr))
    raise error;
}
let v = *ptr;
```

Listing 2: An example for distinguishing between safe and unsafe objects.

this issue, we use shadow memory to mark the types of pointers and only perform checks on pointers of unsafe objects at runtime.

Take the program in Listing 2 as an example, the points-to set of `ptr` contains both safe and unsafe objects. To check only references on unsafe objects, a shadow memory is allocated and indexed by the pointer's address. This method is inspired by SoftBound [28], but instead of storing bound information of a pointer in shadow memory, we only use 1 bit to store whether a pointer points to an unsafe object at runtime. More detail on how the metadata is propagated and passed into function can found in the SoftBound paper [28]. Compared to SoftBound, this has much lower space overhead: As heap objects managed by `ptmalloc2` are 16 bytes aligned, XRust imposes at most 1 bit overhead for 16 bytes memory, thus the memory overhead is $< 1\%$. The protection on shadow memory can be done by using approaches discussed in CPI [24] with negligible overhead.

5.2 Guard Page

A more efficient approach can be implemented by imposing two guard pages below and above each heap segment. Since the guard pape cannot be accessed, cross-region references can be detected when it touch the guard page. To bypass this protection, cross-region references must stride across an entire guard page to avoid being detected.

This approach is often more efficient than code instrumentation, though in theory guard page is incomplete (e.g., a direct long jump from the unsafe region to the safe region without touching the guard page). There are reports on how the Linux's stack guard page can be bypassed to launch attacks [13], and it could be mitigated by enlarging the size of guard pages [12]. Nevertheless, complete and efficient hardware-based techniques such as Intel MPK and ARM memory domains can also be integrated into XRust, as explored in recent work [9, 48].

Using guard pages also avoids pointer analysis needed by instrumentation. Unlike instrumentation, which requires pointer analysis to locate unsafe objects to insert assertions before memory accesses on them, guard page enforces isolation automatically after the objects are allocated into different regions. If a cross-region data flow occurs on an unsafe object, the guard page will be accessed and a segment fault will be issued automatically by the operating system.

6 EVALUATION

We have conducted extensive experiments to evaluate the effectiveness and efficiency of XRust. To evaluate the efficiency, we deployed XRust on six widely-used real-world applications. We also studied five core components of Rust's standard library where unsafe Rust is used ubiquitously to examine XRust in extreme cases. We measured the overhead of XRust under two protection schemes: guard page and instrumentation. Our experimental results show that XRust incurs 0.15% overhead on median (2% on average) when applying guard pages to detect cross-region references, and 3.6% overhead on median (21% on average) when using instrumentation. We also compared XRust with DFI [7] to understand the overhead that could be introduced by imposing a full protection of data-flow integrity.

To evaluate the effectiveness, we studied all the three publicly reported memory corruption errors that we could find in real Rust programs [5, 37, 38]. We designed attacks to exploit these errors and applied XRust to defend against them.

The experiments ran on an AMD Ryzen 2600X with 6 cores@3.6GHz processor in 64 bit mode with 32GB RAM. All experiments were done on Ubuntu Bionic Beaver (18.04 LTS).

6.1 Efficiency

All the real-world Rust applications are popular projects (with more than one million downloads) collected from crates.io, the official package central repository of Rust, and they all contain unsafe Rust code. We use their built-in benchmarks to measure the performance of XRust for fairness. All the Rust standard libraries are measured using the benchmarks from the Rust compiler. The results are reported in Table 2 (averaged over 50 runs).

When using guard page, the overhead comes from the inserted checks performed in the heap allocator to avoid errors caused by corrupted metadata in the unsafe region (discussed in Section 4.2.3). Most cross-region references outside the heap allocator are automatically detected and reported by the operating system upon illegal accesses on guard pages. This approach also introduces 8 KB (two pages) memory overhead for each unsafe heap segment to place guard pages right below and above every unsafe segment.

As reported in Table 2, the overhead is negligible for most real-world applications (less than 0.5% for `base64`, `byteorder`, `image` and `regex`). One important factor that affects the performance is the frequency of heap allocations performed in the unsafe heap region. The highest overhead (9.6%) is reported on `bytes`, which heavily relies on unsafe heap allocation. We discuss the allocation statistics in Section 6.2.

When using instrumentation, the overhead is higher than using guard pages, because it requires a region check before each memory reference on unsafe objects. Nevertheless, the overhead is still low in most cases (less than 5% for `base64`, `byteorder`, `image` and `regex`, and 16% for `bytes`). The highest overhead (103%) is reported on `base64` (different from that of using guard pages). By analyzing the instrumented program, we found the reason is that in

Table 2: Performance of XRust and DFI on real-world Rust applications and standard Rust libraries (grayed rows).

App	Ver.	LoC	#Down-nload	Native (ms/iter)	XRust				DFI	
					g-page	overhead	inst.	overhead	exec.	overhead
base64	0.5.1	2K	2.32M	3527.72	3529.59	0.06%	7167.63	103.15%	9721.60	175.58%
byteorder	1.2.7	2.3K	4.70M	25.91	26.01	0.03%	26.76	3.28%	64.53	149.05%
json	0.11.13	4.3K	0.39M	2213.17	2260.96	2.16%	2298.91	3.87%	12985.72	486.75%
bytes	0.4.10	7.9K	1.80M	6.24	6.84	9.62%	7.25	16.19%	33.471	436.39%
image	0.20.1	13.3K	0.54M	2151.26	2152.87	0.07%	2189.92	1.77%	13426.83	524.14%
regex	1.0.6	48.1K	6.03M	2157.80	2162.48	0.22%	2187.68	1.17%	15251.78	606.82%
Median	-	-	-	2154.53	2157.68	0.15%	2188.80	3.6%	11353.66	461.57%
Average	-	-	-	1680.35	1689.79	2.03%	2313.03	21.57%	8580.66	396.46%
vec	1.30.0	-	-	0.40	0.42	4.08%	0.90	123.08%	4.32	555.00%
string	1.30.0	-	-	2.00	2.03	1.52%	4.16	108.30%	5.57	178.50%
linked-list	1.30.0	-	-	0.16	0.17	6.76%	0.20	13.70%	0.52	225.00%
vec-deque	1.30.0	-	-	0.71	0.71	1.13%	0.72	2.26%	4.01	464.79%
btree	1.30.0	-	-	21.97	22.58	2.80%	23.88	8.69%	114.81	422.58%
Median	-	-	-	0.71	0.71	2.80%	0.90	13.70%	4.32	422.58%
Avg.	-	-	-	5.05	5.18	3.26%	5.92	51.21%	25.85	369.17%

`base64` the checks are inserted into a performance-critical function, which occupies over 98% execution time of the program.

For the Rust standard libraries, the overhead is slightly higher than the real Rust applications, with approximately 3% for guard pages and 50% for instrumentation. The reason is that to bridge between unsafe low-level operations and high-level Rust language features, Rust's standard library typically uses more unsafe Rust code, which increases the number of inserted runtime checks. Also, the performance overhead is highly-related to the tests performed on the benchmarks. Since we used the original test suites (for evaluation fairness) and the tests examine different aspects of the benchmarks, it could lead to different performance numbers. For example, one of the four test cases for `vec-deque` aims at testing the speed of allocating new objects, which imposes little overhead as no memory references need to be instrumented. It explains the differences between the overhead reported in Table 2.

Comparison to DFI, DFI [7] provides a strong protection against control and data attacks, by ensuring the integrity of data flows at runtime with respect to a statically computed data-flow graph. Unfortunately DFI incurs prohibitive overhead in practice (e.g., around 4X runtime overhead on average in our experiments[5]). In their original work, Castro et al. use a static reaching definition analysis to determine the set of write instructions for each memory read, and maintain a runtime definition table (RDT) to record the last write instruction to each memory location at runtime. This incurs both large runtime overhead (for checking all reads and writes) and space overhead (for storing the RDT) even after several optimizations. Differently, XRust ensures the data-flow integrity from unsafe Rust to safe Rust by isolating the unsafe memory region, thus it is much faster (over an order of magnitude) than the full DFI, as reported in Table 2.

[5]The DFI prototype was implemented by ourselves on top of LLVM following the paper [7], since DFI is not available.

Table 3: Allocation statistics in safe and unsafe heap regions.

App	#allocation (safe)	#allocation (unsafe)	% unsafe allocation
base64	57.50M	25.41M	30.64%
byteorder	13.47K	4	0.02%
json	18.07M	0.39M	2.11%
bytes	34.06M	126.98M	78.85%
image	9.21M	10	0.00%
regex	19.83M	675	0.00%
Avg.	23.11M	25.46M	18.60%
vec	611.89M	37.49M	5.78%
string	516.80M	40.69M	7.21%
linked-list	2.93K	68.70M	99.95%
vec-deque	1.40K	12.54M	99.98%
btree	2.99K	20.51K	87.28%
Avg.	115.21M	42.26M	60.04%

6.2 Allocation Statistics

Table 3 reports the statistics of heap allocations in safe and unsafe regions in our experiments. For most applications, they only use unsafe Rust in limited locations and thus the allocations in the unsafe region only account for a small fraction of the total amount of heap allocations. One exception is `bytes`, which has around 80% allocations in unsafe code. It is because `bytes` is a library that deals with low-level data structure. It relies on unsafe Rust heavily to access the low-level binary data. For other applications such as `regex` and `json`, almost all objects are safe. The data also confirms our observation that in high-level user applications, programmers typically avoid heavy use of unsafe Rust.

For the standard Rust libraries, the statistics are the opposite. For three out of the five libraries, almost all the objects are unsafe. However, this is not surprising since unsafe Rust is widely used in standard libraries to deal with low-level operations.

Table 4: Performance of the heap allocator with different numbers of unsafe heap segments.

Size (byte)	#Thread: 1			#Thread: 2			#Thread: 4			#Thread: 8			Avg.
	ptm-alloc [1]	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*ptm-alloc*	*unsafe ext.*	*over-head*	*over-head*
16~1*k*	25.9*M*	24.5*M*	5.3%	5.2*M*	5.1*M*	1.3%	3.8*M*	3.5*M*	6.6%	2.3*M*	2.2*M*	3.9%	4.3%
32~2*k*	25.6*M*	23.7*M*	7.8%	4.3*M*	4.2*M*	3.7%	2.7*M*	2.7*M*	0.5%	2.2*M*	2.2*M*	0.3%	3.1%
64~4*k*	16.0*M*	15.5*M*	3.1%	1.7*M*	1.6*M*	5.5%	1.3*M*	1.2*M*	8.3%	1.9*M*	1.8*M*	5.3%	5.6%
128~8*k*	14.9*M*	13.8*M*	7.9%	1.4*M*	1.3*M*	11.4%	1.0*M*	1.0*M*	3.8%	1.8*M*	1.7*M*	7.9%	7.7%
256~16*k*	13.7*M*	12.9*M*	6.5%	1.1*M*	1.1*M*	3.3%	1.2*M*	1.1*M*	7.9%	1.7*M*	1.6*M*	6.4%	6.0%
Avg.	19.2*M*	18.1*M*	6.1%	2.8*M*	2.7*M*	5.0%	2.0*M*	1.9*M*	5.4%	2.0*M*	1.9*M*	4.7%	5.3%

```
// using VecDeque as an example
pub fn exploit() {
  //target object to corrupt (on stack)
  let t = target_trait_obj;
  let mut a = VecDeque::with_capacity(num);
  {
    // allocated just below deque
① let mut b = Vec::with_capacity();

② // overflow here (in Section 6.4.1);
    deque.reserve();
    // corrupt metadata
    prepare_to_attack();
③ deque.push_back(&t);
④ }  // free of b
}
```

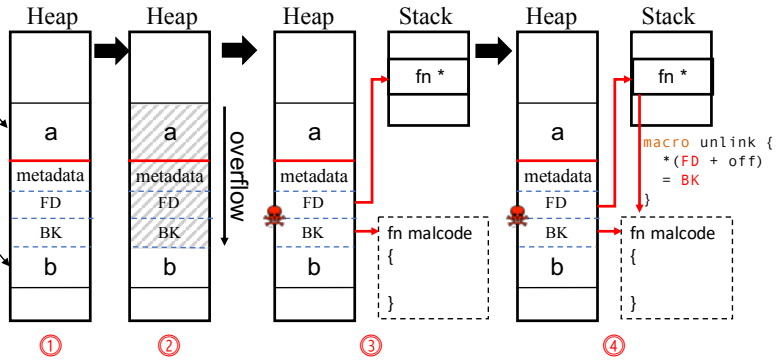

Figure 5: A proof-of-concept attack performed on `VecDeque`.

6.3 Performance of the Allocactor

The customized heap allocator is a core part of XRust and the checks inserted inside the allocator can affect performance. To quantify its performance, we have heavily tested the allocator using a benchmark from `rpmalloc` [19] and compared with unmodified `ptmalloc2`. In our settings, the benchmark iterates 20,000 times in total and in each iteration it allocates and frees 30,000 heap objects of various sizes. In addition, all the objects are allocated via extended interfaces and placed in the unsafe region. This experiment could be viewed as a worst case stress testing since the only functionality of the benchmark is to allocate and deallocate heap memory, and hence provides insights on the worst case performance of the allocator.

The results are reported in Table 4. The overall performance of the extended allocator is about 5% slower than `ptmalloc2` on average when tested with 1, 2, 4 and 8 threads.

6.4 Effectiveness on Real Vulnerabilities

By the time of writing, we found three reported memory corruption errors in real-world Rust programs. We carefully studied each of them and found that XRust is capable of preventing all these errors.

6.4.1 Corruption in `VecDeque`. `VecDeque` is a double-ended queue implemented with a growable ring buffer and it is a part of Rust's standard library. A buffer overflow vulnerability (CVE-2018-1000657) was discovered inside the `VecDeque::reserve` function only recently. The simplified code is listed below:

```
1 pub fn reverse(&mut self, additional: usize) {
2   let new_cap = used_cap + additional;
3   if new_cap > self.capacity() {
4     self.buf.reserve(..);
5     unsafe {
6       self.handle_cap_increase(..);
7 } } }
```

The root cause of the bug is on line 4, where the function mixes up its internal capacity with its user-visible capacity. Because the user-visible capacity is one element smaller than the actual size of the buffer, The unsafe function `handle_cap_increase` can cause the pointer to point to out-of-bound memory address and upon next push, a value can be written outside the buffer.

The vulnerability can be exploited to overflow one element outside the buffer, Because there is no public attack on this vulnerability, we manually built a proof-of-concept case with a vulnerable program using the function, and performed an `unsafe unlink` exploit [39] to make an arbitrary write to vtable pointers as in Figure 5. This attack would fail on recent glibc since extra security checks were added into the library. Workaround to bypass the checks could be found in [39]. The result shows that XRust is able to detect the attack consistently because both the stack and the data segment are outside the unsafe heap region. A cross-region write to corrupt the stack data and vtable pointers is detected by the heap allocator since the metadata is corrupted by the overflow.

6.4.2 Corruption in `str::repeat`. A buffer overflow bug was reported in the function `str::repeat` (CVE-2018-1000810), which is also a part of Rust's standard library. The root cause of the bug is an instance of integer overflow to buffer overflow bugs. The simplified code is listed below:

```
1  pub fn repeat(&self, u: usize) -> Vec<T> {
2    let mut buf = Vec::with_capacity(n * len);
3
4    while condition {
5      unsafe {
6        ptr::copy(buf.as_ptr(),
7                  buf.as_ptr().add(len),
8                  len);
9        ...
10 }  }  }
```

The function is used to create a string that repeats a fixed number of times. On line 2, when calculating the capacity of the `Vec` to hold the string by $n * len$, an integer overflow could happen, which in turn results in a smaller buffer and causes an overflow when using unsafe code to store the value on line 6. We similarly conducted the same proof-of-the-concept attack on it as on `VecdDeque`, and XRust can detect the overflow as well.

6.4.3 Corruption in `Base64`. The details of this error (CVE-2017-1000430) have been presented in Section 2.2. Attacking this vulnerability is more difficult than the previous two cases, because it requires triggering an overflow on a 64 bit integer. To perform a proof-of-concept attack, we changed the `Base64` code to use 16 bit integer. The experiment setting is similar to the other two cases, and XRust is able to defend the attack in this case as well.

7 RELATED WORK

Techniques against memory-based attacks. Securing software against memory-based attacks has been an extremely important yet challenging problem. Many approaches have been proposed for C/C++ programs, such as SoftBound+CETS [28, 29] to provide full memory safety, CPI [24] to secure code pointers, CFI [1] to defend against control-flow attacks, and DFI [7] to defend against data-flow attacks. Our method of using separate memory to store metadata of pointers is inspired by SoftBound, but instead of storing bound metadata in the shadow memory, we only require one bit per pointers to indicate whether it points to an unsafe object or not. Also, instead of instrumenting every memory reference to check the bounds of objects, we only instrument memory references on unsafe objects and only check if the references are within the unsafe heap region.

Techniques target specifically at control-flow attacks such as CFI [1], CPI [24] have seen wide adoption in commodity operating systems and compilers [46], because they are practical to thwart most abnormal control transfers with only a small or negligible overhead. However, they can be bypassed by data-only attacks such as Heartbleed [17]. DFI is a promising technique to deal with memory corruption, however, as shown in our experiments, a full DFI incurs prohibitive runtime overhead.

[1]Measured by the number of memory operations per CPU second.

Techniques specific to Rust. A few research efforts have been invested into Rust to formally prove memory safety properties of the language [22, 33, 49], and to statically verify Rust programs [47]. Among them, FC [2] shares a similar goal as XRust, though with very different techniques. At the technical level, FC uses operating system-level support whereas XRust extends the heap allocator. FC only isolates data from unsafe foreign functions by `mprotect` system calls instead of all sources of unsafe Rust as addressed by XRust. Also, as discussed earlier in Section 5.1, FC does not handle the *shared* unsafe objects in safe code.

The Rustbelt project [22] has formally proved the memory safety of a realistic subset of Rust, including several standard Rust libraries with the existence of unsafe Rust. CRUST [47] uses bounded model checking to verify memory safety in unsafe Rust code by first translating Rust programs into C, and is shown effective in discovering memory errors in the Rust standard libraries.

Isolation of resources. Isolation is a common approach to mitigate the influence of untrusted resources and XRust can be viewed as a new isolation approach within the address space of a single Rust program. Researchers have proposed numerous techniques to isolate memory regions in different domains. Native Client [51] provides memory sandbox for untrusted library code by loading libraries into limited containers. Codejail [50] propose another approach to limit libraries by making the program data read-only. Linux's seccomp [11] can also be utilized to limit system calls from accessing security-crucial data. For web browsers, the isolation of resources (*e.g.,* SOP [4]) is extremely important to defend agaist malicious websites.

There also exist a variety of isolation techniques by partitioning the code path. Wedge [6] provides memory isolation among sthreads, each of which contains a thread of control and security policy defined by the programmer. Shreds [9], on the other hand, splits each thread's execution into multiple segments marked by `shred_enter()` and `shred_exit()`, and provides isolated compartments of code and data between different segments through compiler and architectural support. These two are different from XRust as they both require programmers' effort to modify the program, whereas XRust is fully automated by leveraging unique features of the Rust programming language. PtrSplit [27] partitions the programs by marking `sensitive` pointers and selectively checking the bounds of pointers. The resource isolation schemes are also studied for Java programs. For example, Robusta [40] and Arabica [44] isolate the Java native interface (JNI) from safe Java code.

8 DISCUSSIONS AND LIMITATIONS

We note that XRust targets memory safety issues brought by unsafe Rust only. XRust assumes Rust's memory safety guarantees to be valid, which requires a correct design and implementation of Rust and its framework (including its standard libraries) so that no memory error will occur in the absence of unsafe Rust code. This is essential as safe abstractions provided by programming languages are inherently encapsulations on unsafe operations. Attackers cannot modify the code segments since they are unwritable and they cannot control the program loading process. These requirements ensure that the integrity of the instrumented dynamic checks and

the heap allocator can safely set up the isolation between safe and unsafe memory regions.

XRust does not handle dynamic code generation. This is a difficult problem because the new code cannot be analyzed or instrumented statically by a compiler. This limitation is shared by techniques relying on static analysis, *e.g.*, SoftBound [28], DFI [7], and CPI [24] will all fail to protect against vulnerable code generated dynamically. A potential solution is to track dynamically generated code and continue the analysis at runtime. We leave it as future work.

9 CONCLUSION

We have presented XRust, a novel approach to protect safe memory objects in Rust from being corrupted by unsafe Rust code. The key idea is to separate the address space of a Rust program into two non-overlapping regions with a customized heap allocator and automatically insert runtime checks to efficiently detect cross-region references on unsafe objects. Our extensive evaluation on both popular real-world Rust applications and standard Rust libraries shows that XRust is highly effective and efficient: it prevents attacks on the all the known Rust vulnerabilities while exhibiting small or negligible overhead. We stress that it is promising to apply XRust to secure Rust applications in practice.

The source-code of the XRust compiler can be obtained through https://github.com/parasol-aser/XRust. The configured docker image can be obtained through https://hub.docker.com/repository/docker/geticliu/xrust-icse2020

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.

[2] Hussain MJ Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 248–255.

[3] andigena. 2016. ptmalloc fanzine. https://lwn.net/Articles/725832/.

[4] Adam Barth. 2011. *The web origin concept.* Technical Report.

[5] Rust base64 project. 2017. CVE-2017-1000430. https://www.cvedetails.com/cve/CVE-2017-1000430/.

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association.

[7] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 147–160.

[8] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium*, Vol. 5.

[9] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 56–71.

[10] cmr. 2018. Unsafe Rust. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[11] Jonathan Corbet. 2009. Seccomp and sandboxing. *LWN. net, May* 25 (2009).

[12] Jonathan Corbet. 2017. Preventing stack guard-page hopping. https://lwn.net/Articles/725832/.

[13] Jonathan Corbet. 2017. The Stack Clash. https://blog.qualys.com/securitylabs/2017/06/19/the-stack-clash.

[14] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.

[15] Alex Crichton. 2015. Tracking issue for changing the global, default allocator (RFC 1974). https://github.com/rust-lang/rust/issues/27389.

[16] ARM documentation. [n.d.]. ARM Memory Domain. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html.

[17] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 475–488.

[18] Wolfram Gloger. 2006. Wolfram Gloger's malloc homepage. http://www.malloc.de/en/.

[19] Mattias Jansson. 2017. rpmalloc-benchmark. https://github.com/rampantpixels/rpmalloc-benchmark.

[20] japaric. 2017. issue 39699. https://github.com/rust-lang/rust/issues/39699.

[21] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

[22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 66.

[23] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press.

[24] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2014. Code-Pointer Integrity.. In *OSDI*, Vol. 14. 00000.

[25] Doug Lea. 1996. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

[26] Stanley B Lippman. 1996. *Inside the C++ object model*. Vol. 242. Addison-Wesley Reading.

[27] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2359–2371.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *ACM Sigplan Notices*, Vol. 45. ACM, 31–40.

[30] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, Vol. 37. ACM, 128–139.

[31] Phasar-Team. 2018. Phasar framework. https://github.com/secure-software-engineering/phasar.

[32] Jonathan Pincus and Brandon Baker. 2004. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy* 2, 4 (2004), 20–27.

[33] Eric Reed. 2015. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015).

[34] Rust-team. 2011. Trait Objects. https://doc.rust-lang.org/book/trait-objects.html.

[35] Rust-Team. 2015. issue 26612. https://github.com/rust-lang/rust/issues/26612.

[36] Rust-Team. 2017. Unsafe Rust. https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html.

[37] Pedro Sampaio. 2018. CVE-2018-1000657. https://bugzilla.redhat.com/show_bug.cgi?id=1622249.

[38] Pedro Sampaio. 2018. CVE-2018-1000810. https://bugzilla.redhat.com/show_bug.cgi?id=1632932.

[39] Shellphish. 2011. unsafe unlink. https://github.com/shellphish/how2heap/blob/master/glibc_2.26/unsafe_unlink.c.

[40] Joseph Siefers, Gang Tan, and Greg Morrisett. 2010. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 201–211.

[41] Brain Smith. 2017. Stop using unsafe code. https://github.com/alicemaz/rust-base64/issues/29.

[42] splotifun. 2015. Understanding glibc malloc. bhttps://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

[43] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 265–266.

[44] Mengtao Sun and Gang Tan. 2012. Jvm-portable sandboxing of javaâĂŹs native libraries. In *European Symposium on Research in Computer Security*. Springer, 842–858.

[45] Rust team. 2015. Lang items. https://doc.rust-lang.org/1.5.0/book/lang-items.html.

[46] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.. In *USENIX Security Symposium*. 941–955.

[47] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A Bounded Verifier for Rust (N). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 75–80.

[48] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. 2018. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. *arXiv preprint arXiv:1801.06822* (2018).

[49] Aaron Weiss, Daniel Patterson, and Amal G Ahmed. 2018. Rust Distilled: An Expressive Tower of Languages. *CoRR* abs/1806.02693 (2018).

[50] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. 2012. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security*. Springer,

859–876.

[51] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 79–93.