MetaSafe: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity

Muhammad Hassnain

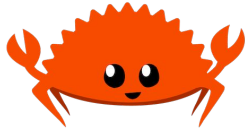
Literature Review

March 06, 2026

Rust 

- Code written in Rust guarantees:
  - Runtime performance of traditional system languages like C/C++
  - Memory Safety
    - No Dangling Pointers
    - No Garbage Collector
    - No Use after free or out of bounds

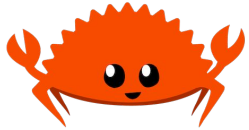
Rust's Magic



```
1
2  fn main() {
3      let my_vec = vec![1,2,3,4];
4      println!("Vec at Index 5 {:?}" , my_vec[5]);
5  }
```

Playground Link

Rust's Magic

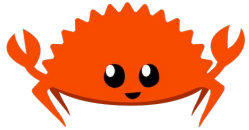


```
1
2  fn main() {
3      let my_vec = vec![1,2,3,4];
4      println!("Vec at Index 5 {:?}" , my_vec[5]);
5  }
```

```
thread 'main' (14) panicked at src/main.rs:4:44:
index out of bounds: the len is 4 but the index is 5
```

Playground Link

Rust's Magic



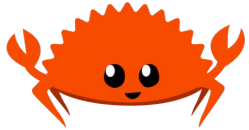
```
1
2  fn main() {
3      let my_vec = vec![1,2,3,4];
4      println!("Vec at Index 5 {:?}" , my_vec[5]);
5  }
```

But how does Rust ensures these things?

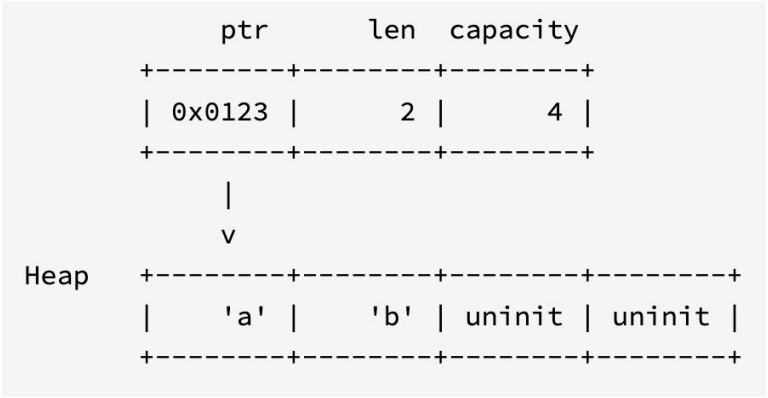
```
thread 'main' (14) panicked at src/main.rs:4:44:
index out of bounds: the len is 4 but the index is 5
```

Playground Link

Rust's Magic

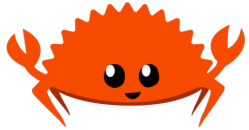


- Turns out it's just MetaData and Runtime Checks

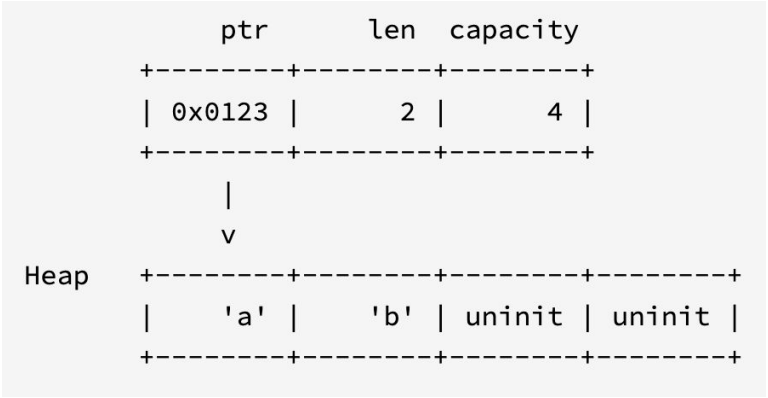


https://doc.rust-lang.org/std/vec/struct.Vec.html

Rust's Magic



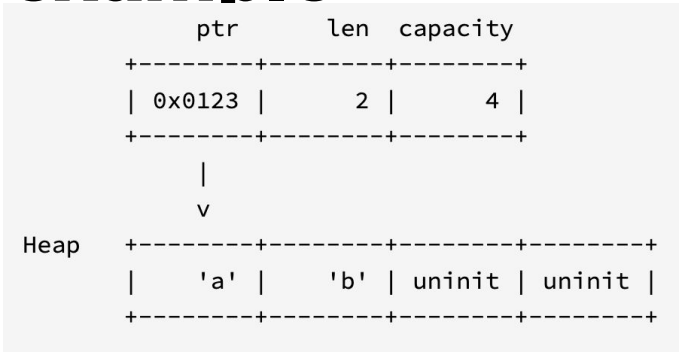
- Turns out it's just MetaData and Runtime Checks



https://doc.rust-lang.org/std/vec/struct.Vec.html

What if someone corrupts this metadata? 😈

An example



https://doc.rust-lang.org/std/vec/struct.Vec.html

Slide Source: Counterexamples in Safe Rust

```rust
fn write_oob(vector: &Vec<i32>, index: usize, element: i32) {

    // Get the pointer to the index location
    let buffer_ptr = vector.as_ptr();
    let ind = buffer_ptr.wrapping_add(index);

    // Open the /proc/self/mem file for writing
    let mut file = OpenOptions::new()
        .write(true)
        .open("/proc/self/mem")
        .unwrap();

    // Seek to the memory address of index
    file.seek(std::io::SeekFrom::Start(ind as u64)).unwrap();

    // Write the provided element into the calculated index position
    file.write_all(&element.to_ne_bytes()).unwrap();

    // Resize the vector to read the updated memory
    let vec_ptr: *const usize = vector as *const Vec<i32> as *const usize;
    let capacity_ptr: *const usize = vec_ptr.wrapping_add(2);
    file.seek(std::io::SeekFrom::Start(capacity_ptr as u64)).unwrap();
    let num = index + 1;
    file.write_all(&num.to_ne_bytes()).unwrap();

    // Print the updated vector element
    println!("I have {:?}", vector[index]);
}
```

Figure 2: Function to read/write out-of-bounds in safe Rust, using the Linux-specific path `/proc/self/mem`.

Smart Pointer	Metadata	Purpose	Safety	Vulnerability
Box	None	Basic Heap Object	Rust Ownership	UAF
Vec	Len, Capacity	Dynamic Buffer	Spatial	Overflow, UAF
Rc	Reference Counters	Shared Reference	Temporal	UAF
Arc	Reference Counters	Thread Safe Shared Reference	Temporal, Thread	UAF, Dataraces
Mutex, Mutex Guard	Locks	Thread Safe Interior Mutability	Thread, Temporal	UAF, Dataraces
RwLock	Reference Counters	Arc + Mutex	Thread, Temporal	UAF, Races

MetaSafe

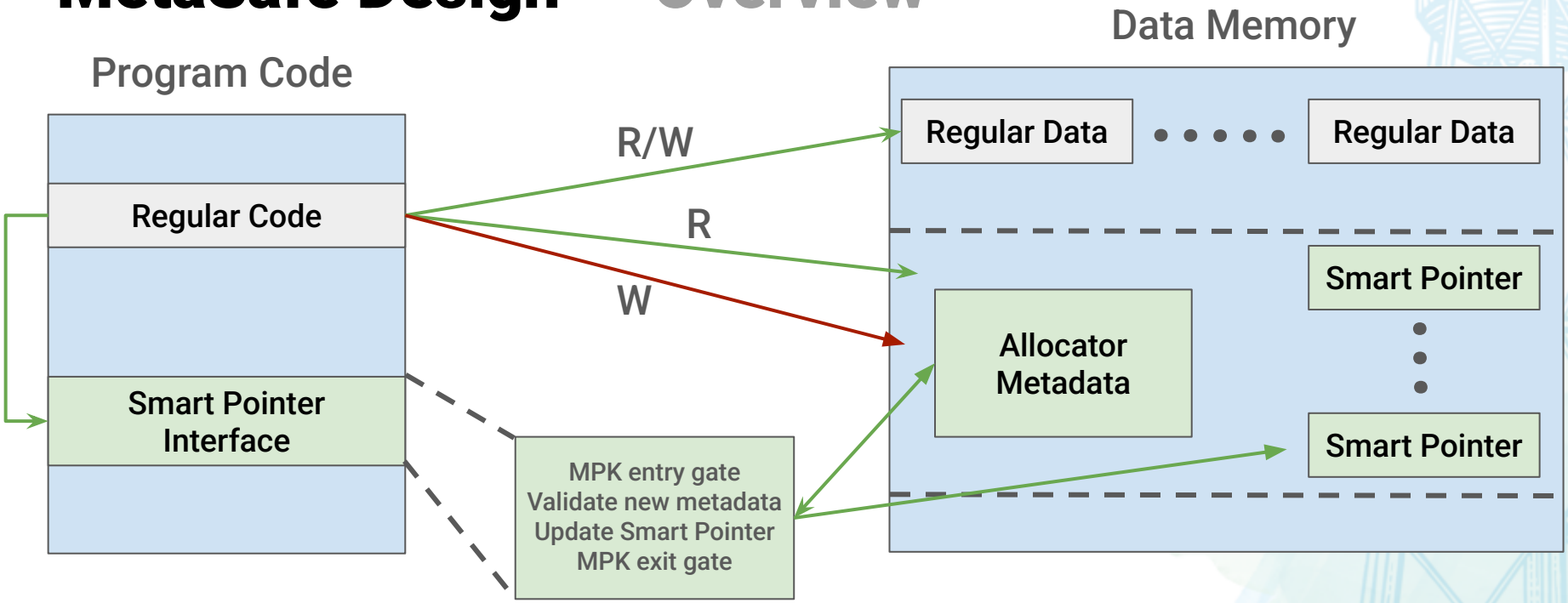
- Isolate metadata and store it in an isolated compartment
- Perform sanitization checks on metadata updates



Assumptions & Threat Model

- There is an isolation b/w Safe and Unsafe Rust (XRust, Galeed, TRust etc)
- Attacker targeting Rust program to corrupt memory safety.
- Attacker also aware of Rust polymorphism and can corrupt function pointer in traits to hijack the program.

MetaSafe Design - Overview



MetaSafe Design - Challenges

1. Ground Truth for Metadata Validation
    - Metadata for smart pointers not static
    - Often updated legitimately at Runtime

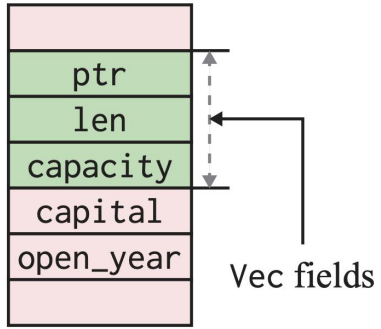
MetaSafe Design - Challenges

1. Ground Truth for Metadata Validation
2. Identifying Smart Pointers
    - No clear criteria
    - Deref Trait?
    - Custom Smart Pointers….

MetaSafe Design - Challenges

1. Ground Truth for Metadata Validation
2. Identifying Smart Pointers
3. Embedded Smart Pointers
   - Composite Types
   - How are they stored?

```
1 struct Firm {
2     employees: Vec<String>,
3     captital: usize,
4     open_year: usize,
5 }
```



(a) An ADT with a smart pointer field.

(b) Layout of the Firm ADT in memory

MetaSafe Design - Challenges

1. Ground Truth for Metadata Validation
2. Identifying Smart Pointers
3. Embedded Smart Pointers
4. Securing References to Smart Pointers
    - References to smart pointers accessible in unsafe/ffi

MetaSafe Design - Implementation

1. MetaUpdate Trait
   - A **validate** function to validate updates to smart pointer metadata.
   - Inserts this at the end of every function in smart pointer implementation that can modify metadata

```
1  impl<T, A> MetaUpdate for Vec<T, A> {
2      fn validate(&self) -> bool {
3          metasafe::isLive(self.ptr) &&
4          metasafe::getSize(self.ptr) >=
5          self.capacity()*sizeof(T) &&
6          self.capacity() >= self.len()
7      }
8  }
```

MetaSafe Design - Implementation

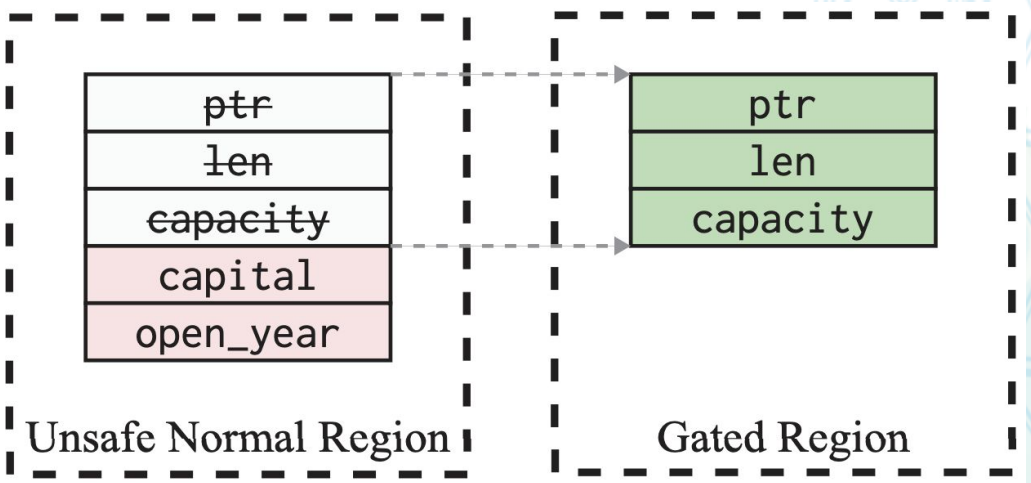
2. Identifying Smart Pointer
    - Identify at HIR level
    - Everything with MetaUpdate trait is a smart pointer
    - Annotate with tags

MetaSafe Design - Implementation

3. Storing Smart Pointers in Gated Region
    - Identify tagged regions and store them in gated region (LLVM IR level).
    - Store Information about this heap chunk (CIndex) for future verification.

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
   - Maintain shadow copies in gated regions
   - Instrument and redirect all memory references



MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    a. Secure and Efficient Redirection for Heap Objects
        i. Type pooled heap

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    a. Secure and Efficient Redirection for Heap Objects
        i. Type pooled heap
        ii. Low ratio of smart pointers in composite types

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    a. Secure and Efficient Redirection for Heap Objects
        i. Type pooled heap
        ii. Low ratio in composite types
        iii. Uses an offset computation relying only on heap metadata from:
            1. Original smart pointer field's address
            2. Composite type object base address

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    a. Secure and Efficient Redirection for Heap Objects
        i. Type pooled heap
        ii. Low ratio in composite types
        iii. Uses an offset computation relying only on heap metadata from:
            1. Original smart pointer field's address
            2. Composite type object base address
        iv. The computation:
            1. Heap allocator determines the pool object belongs to with object's offset.
            2. From base address of poll and object offset computes base address of shadow copy poll and return the offset

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    b. Secure and Efficient Redirection for Stack Objects
        i. Determines the layout of stack frame at compile time
        ii. Similarly maintain and redirect to shadow stack.

MetaSafe Design - Implementation

4. Handling Embedded Smart Pointers
    c. Contiguous Memory Operations
        i. Memory operations are replicated across the shadow memory
        ii. A memcopy on actual buffer mandates memcopy on shadow memory.

These things are identified during compile time.

MetaSafe Design - Implementation

Type-pooled Heap Allocator

- Mimalloc based
- Provides ground truth for chunk bounds and chunk types
- Keeps metadata about allocations
- Provides some functions based on metadata:
  - .is_valid_ptr() → whether pointer is part of a heap or not
  - .getSize() → gets size of the chunk
  - .isLive() → adds a bit to page metadata for this, O(1) operation

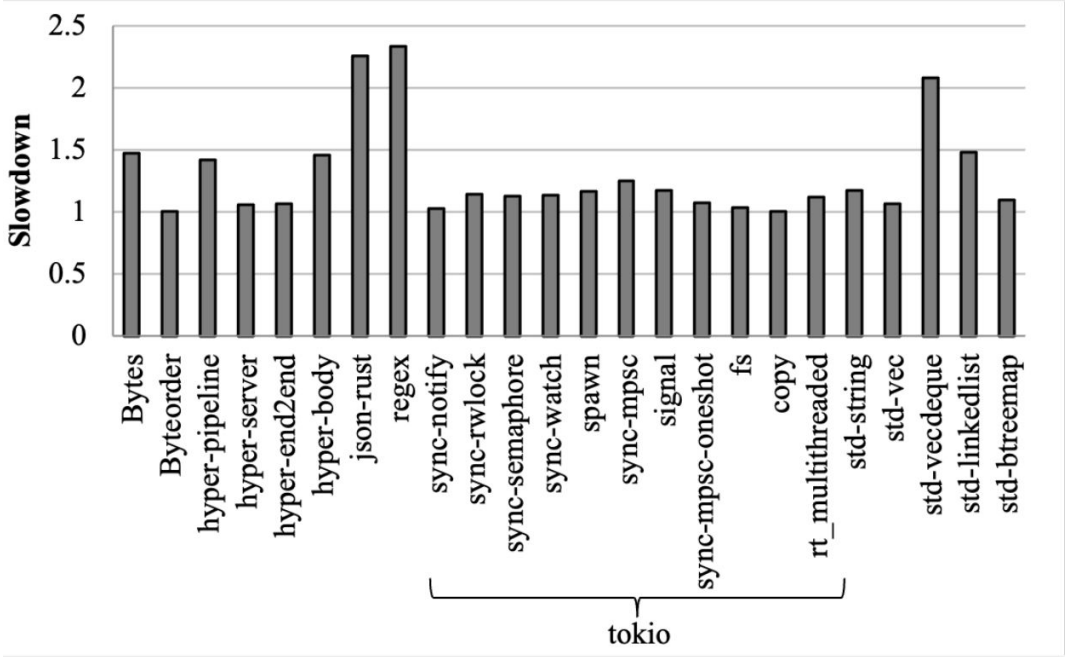
MetaSafe Design - Implementation

Protecting Gated Regions

- Orthogonal to contribution
- Uses MPK keys
- Can use PKRU-Safe or TRust

Evaluation - Micro Benchmark Results - Time

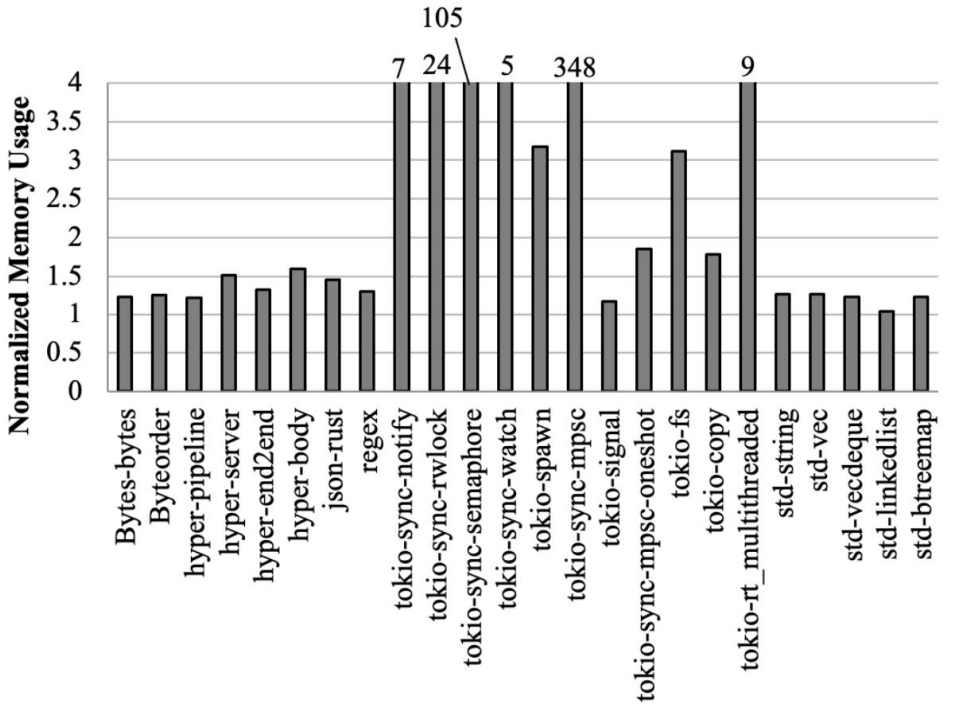
- 25.5% slowdown on average
- Regex and Json uses composite types
- Vecdeque, linkedlist and btreemap is high due to frequent metadata updates.



Evaluation - Micro Benchmark Results - Memory

Measures the maximum allocated physical memory during process's lifetime

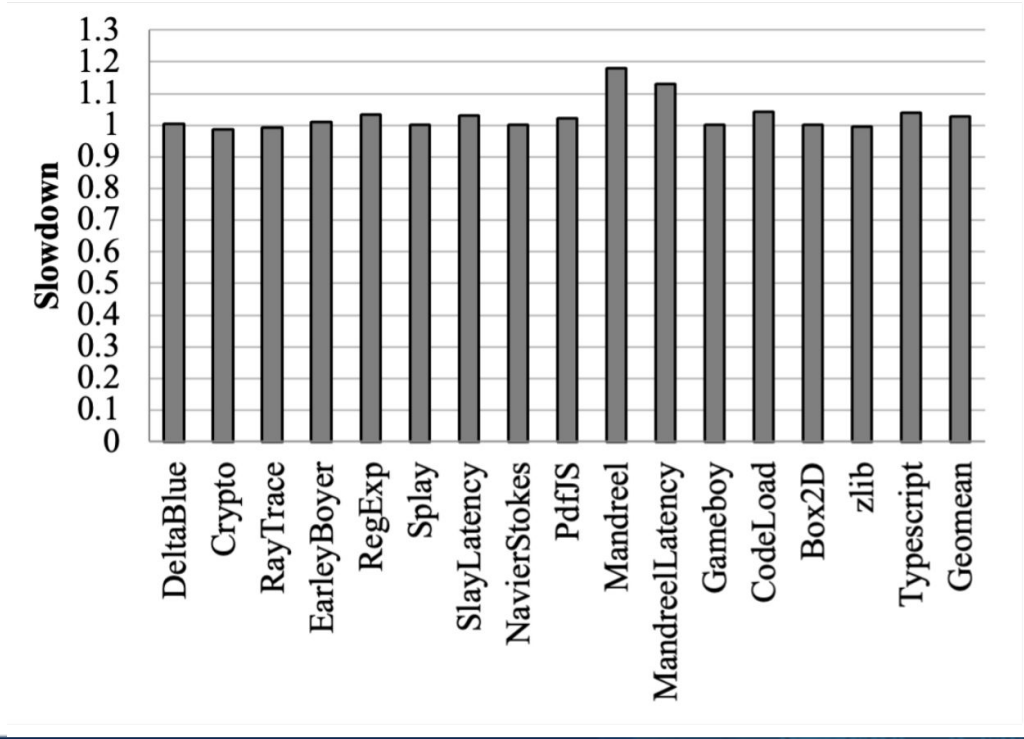
- 27% more memory
- 8.3x on tokio - 31% if all objects on same stack?



Evaluation - Servo Results

Modify servo to use mimalloc.
Two types of benchmarks

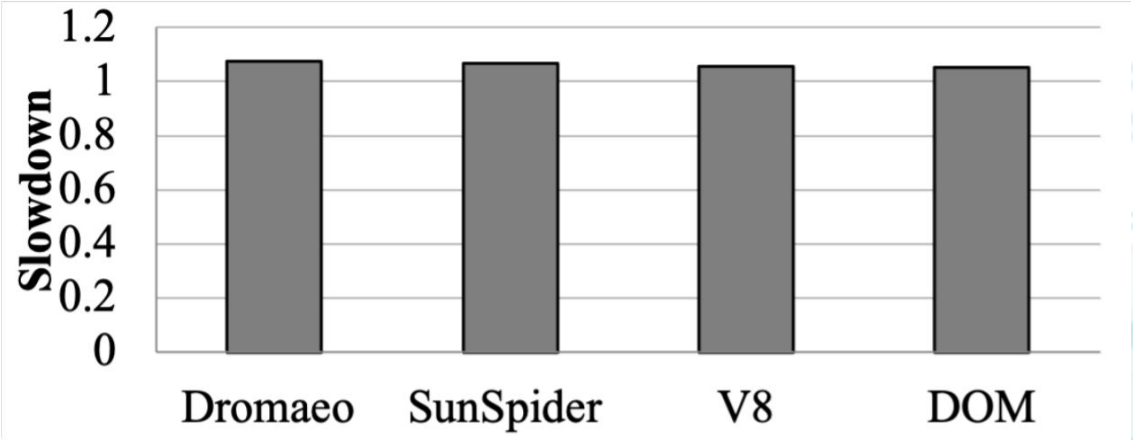
- Octane2 - dimensions of javascript performance



Evaluation - Servo Results

Modify servo to use mimalloc.
Two types of benchmarks

- Octane2 - dimensions of javascript performance
- Dromaeo - Browser related operations



Evaluation - Vulnerabilities

Case Study on two examples

1. https://github.com/servo/rust-smallvec/issues/252

They don't provide any modified code :(

```rust
#![forbid(unsafe_code)]

use smallvec::SmallVec;

fn main() {
    let mut v: SmallVec<[u8; 0]> = SmallVec::new();

    // Spill on heap
    v.push(123);

    // Allocate string on heap
    let s = String::from("Hello!");
    println!("{}", s);

    // Prepare an iterator with small lower bound
    let mut iter = (0u8..=255).filter(|n| n % 2 == 0);
    assert_eq!(iter.size_hint().0, 0);

    // Triggering the bug
    v.insert_many(0, iter);

    // Uh oh, heap overflow made smallvec and string to overlap
    assert!(v.as_ptr_range().contains(&s.as_ptr()));

    // String is corrupted
    println!("{}", s);
}
```

Output:

```
Hello!
@BDFHJ
double free or corruption (out)

Terminated with signal 6 (SIGABRT)
```

Evaluation - Vulnerabilities

Case Study on two examples

2. https://rustsec.org/advisories/RUSTSEC-2019-0006.html

This is ncurses

https://github.com/davsec-lab/page-wars/tree/main/evaluation/ncurses-example

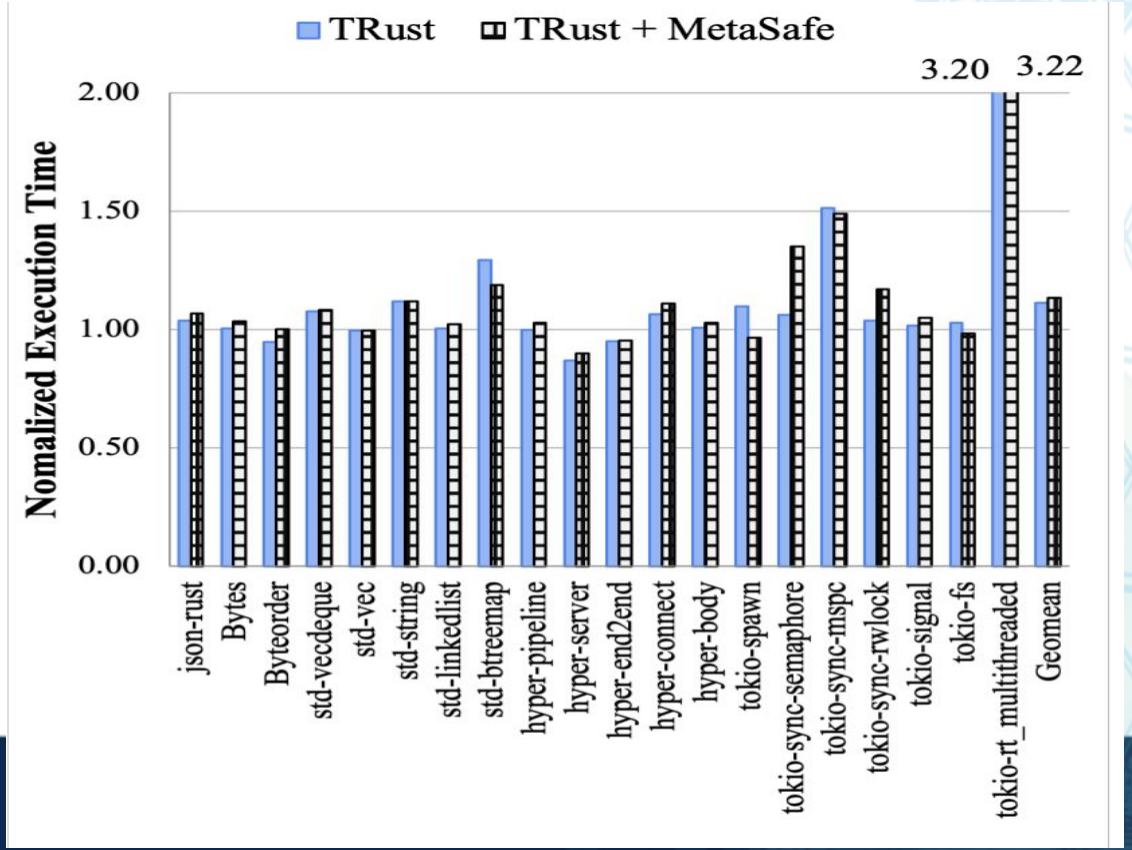
```
pub fn instr(s: &mut String) -> i32
{
  /* XXX: This is probably broken. */
  unsafe
  {
    let buf = s.as_bytes().as_ptr();
    let ret = ll::instr(mem::transmute(buf));

    let capacity = s.capacity();
    match s.find('\0')
    {
      Some(index) => s.as_mut_vec().set_len(index as usize),
      None => s.as_mut_vec().set_len(capacity),
    }

    ret
  }
}
```

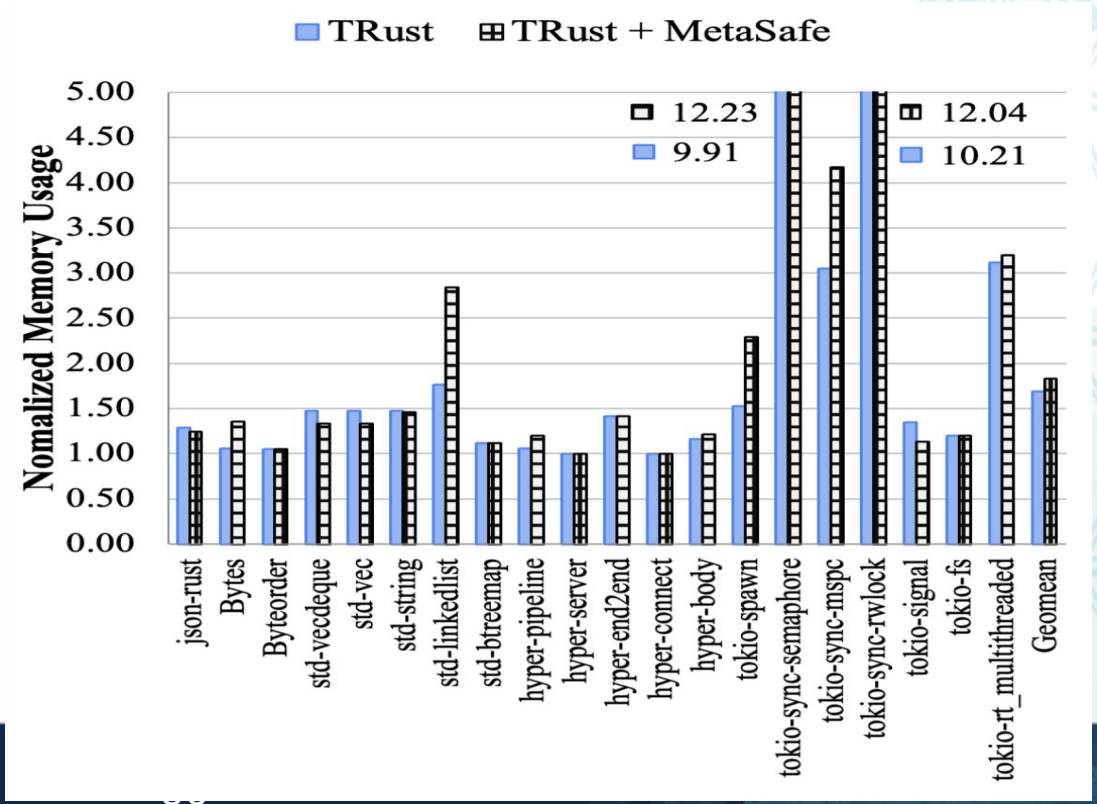
Evaluation - MetaSafe + TRust - Time

- Switch off MPK
- TRust → 11%
- TRust + MetaSafe → 13%



Evaluation - MetaSafe + TRust - Time

- TRust → 69%
- TRust + MetaSafe → 83%



Limitations