METASAFE: Compiling for Protecting Smart Pointer Metadata To Ensure Safe Rust Integrity

Martin Kayondo, Inyoung Bang, Yeongjun Kwak, Hyungon Moon, and Yunheung Paek

USENIX Security 2024

SEOUL NATIONAL UNIVERSITY

UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Rust: A Memory Safe System Programming Language

- Rust is gaining popularity as a memory safe programming language
  - Replacing C/C++ in some production software (Linux, Microsoft, Android)
  - Reportedly resulted memory bug reduction (Android: 76% ➔ 25%)

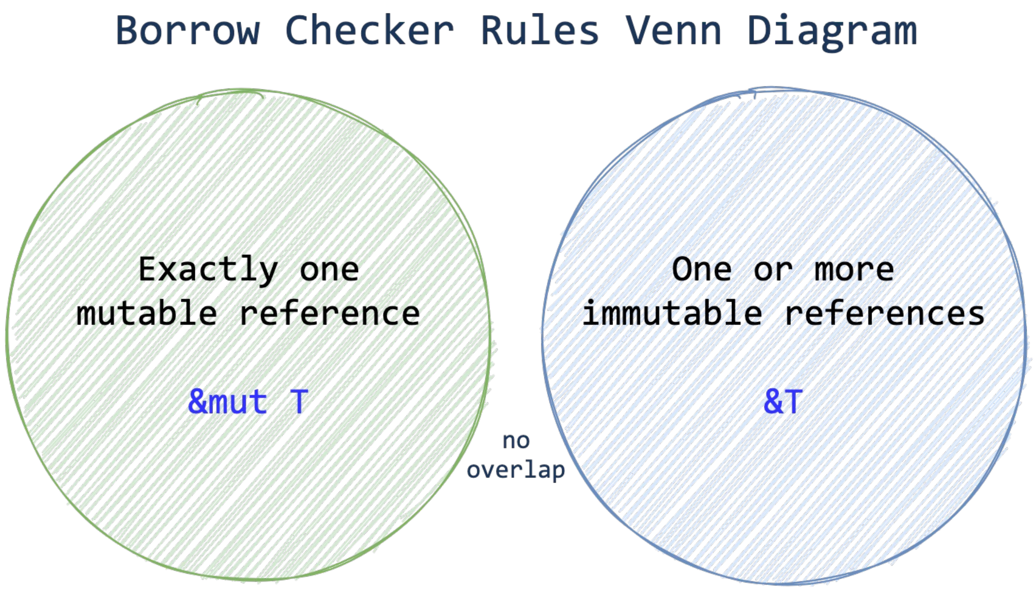
- Aspires to maintain runtime performance
  - In some cases faster than C/C++

- Currently gradually replacing C/C++, Python, Java

CLOUDFLARE

Memory Safety in Rust: Policy

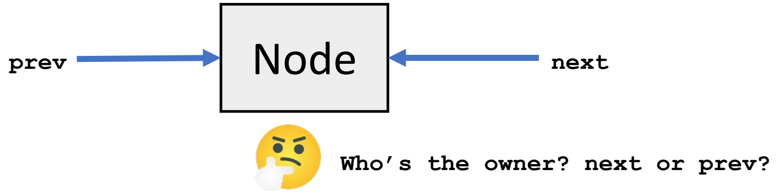
- Memory Access Policy:
  - Ownership: A memory object shall have one owner at any point in time.
  - Borrowing: A memory object may be borrowed:
    - Immutably by one or more entities
    - Mutably by a single entity
  - Lifetimes: A memory object can only be accessed when it is live
- Policy Enforcement:
  - Compiler-based (borrow checker, lifetime analyzer)



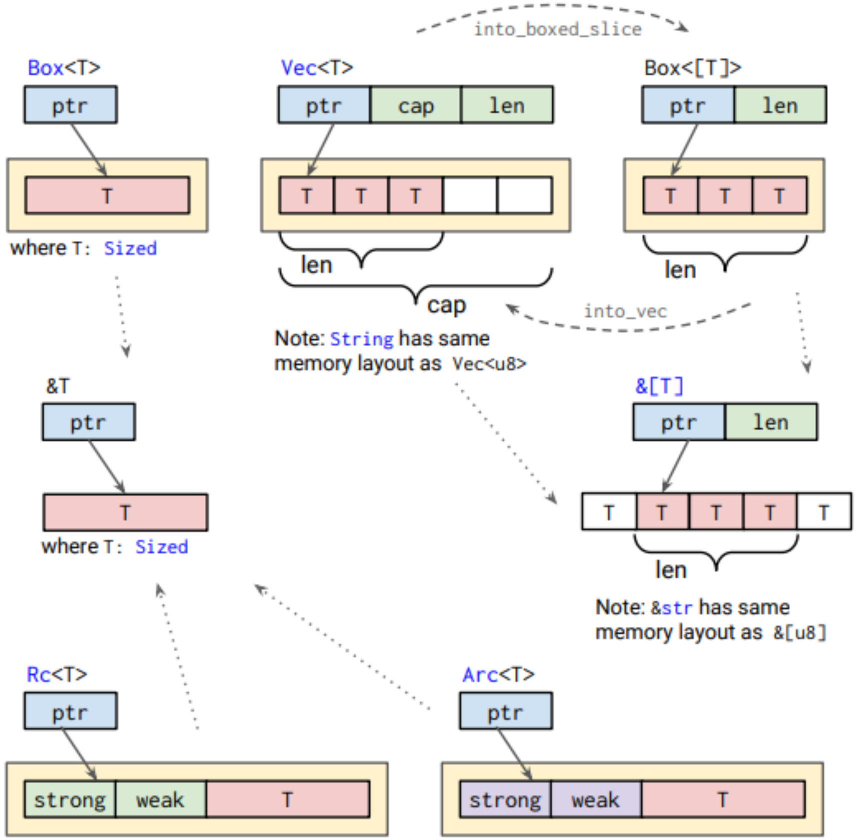
https://livebook.manning.com/book/code-like-a-pro-in-rust/chapter-1/v-2/

Smart Pointers and Their Metadata in Rust

- Rust Memory Rules are too strict
  - Limit expressive power
  - Impossible to implement some widely-used DS.
    - How to design a Doubly Linked List?



- Smart Pointers to the **SAFE** rescue
- A way to enforce memory safety rules at **runtime**
  - Buffer Pointers with buffer length metadata
    - Vec<T>, Slice<T>
  - Shared Pointers with reference counters
    - Rc<T>, Arc<T>
  - Interior Mutability with special metadata
    - RefCell<T> with mutable borrower counters
    - Mutex<T>, RwLock with lock metadata

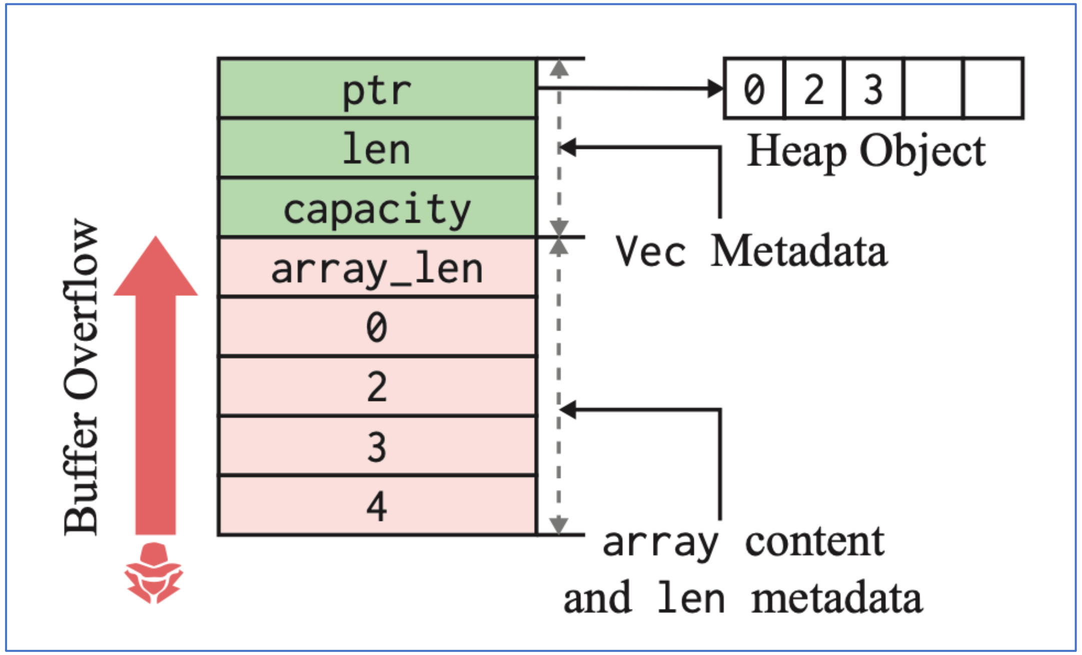


https://tc.gts3.org/cs3210/2020/spring/l/lec09/lec09.html

Smart Pointer Metadata Storage

```
use std::rc::Rc;
fn main {
    let mut xvec = vec![0,2,3];
    let mut array = [0,2,3,4];

    unsafe {
        let mut ptr = array.as_mut_ptr()
                      .offset(10);
        *ptr = 10;
    }
    //...
    //...
```



Several existing CVEs on unsafe Rust and unchecked length-related buffer overflows.

Smart Pointer APIs and Metadata Access

```rust
use std::ptr;
use std::alloc::{dealloc, Layout};
fn test(input: *mut String){
    unsafe{
        //destructor of the pointed-to value
        ptr::drop_in_place(input);
        dealloc(input);
    }
}
fn main(){
    let x = Box::new(String::from("hello"));
    let p = Box::into_raw(x);
    test(p);
    println!("{:?}",p);
    let tmp1 = unsafe{ Box::from_raw(p)}
    println!("{:?}",tmp1)
}
```

User drops the string pointer

User creates Box pointer using *from_raw*
Box doesn't check received pointer
println! reads from new pointer ➔ UAF

CVE-2021-25900

```rust
fn main() {
    let mut buffer = vec![0,2,3];
    unsafe {
        buffer.set_len(10);
    }
    println!("Element at 9: {}", buffer[9]);
}
```

User overwrites Vec metadata using *set_len*
Vec doesn't validate new length

buffer[9] makes OOB read due to new invalid length metadata

Related Works: Enhancing Rust Safety

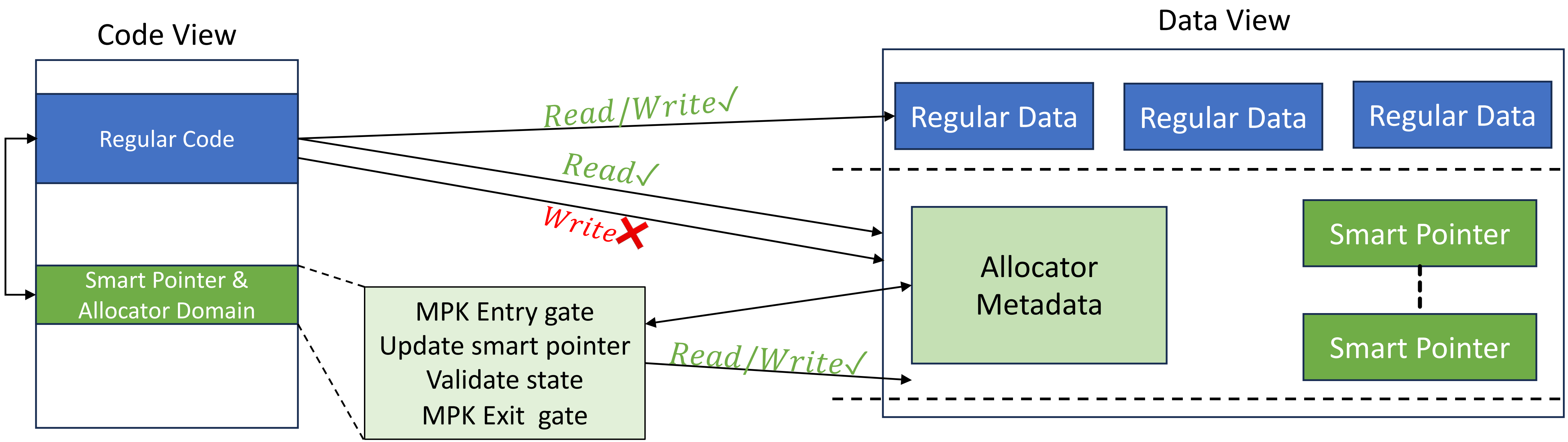
- Most works focus on Unsafe Vs Safe Rust memory Isolation.
- TRust: USENIX Security 2023
  - Protects Safe Rust by isolating memory used by Safe Rust and Unsafe/FFI
  - Uses static analysis, Intel MPK + SFI to achieve runtime performance
- PKRU-Safe: CCS 2022
  - Similar to TRust, isolates safe Rust and FFI with Intel MPK
  - Relies on dynamic profiling instead of static analysis
- Galeed: ACSAC 2021
  - Similar to TRust, isolates safe Rust and FFI with Intel MPK
  - Uses pseudo-pointers to provide strict Temporal access to shared Rust objects by FFI
- XRust: ICSE 2020
  - Provides isolation between safe and unsafe Rust memory
  - Does not specifically consider FFI

- None of Existing works give special care to smart pointers, eg. Validating metadata updates
- METASAFE aims NOT to REPLACE unsafe Rust Isolation works, rather to COMPLEMENT them

Protecting Smart Pointer Metadata

- METASAFE:
  - Protects smart pointer metadata & validates updates
- Requirements:
  - Identification of Smart Pointers
    - The Rust Compiler treats smart pointers similarly to other data structures (except Box)
  - Separate isolated storage
    - Need for storing smart pointers in a gated region.
  - Controlled write access to metadata
    - Preventing illegal access to smart pointer metadata
  - Authentication of metadata updates through unsafe APIs
    - Ensure unsafe APIs write valid smart pointer metadata

METASAFE Overview



- Categorize code between regular and smart pointer domain
- Categorize data into regular, allocator and smart pointer metadata
- Enforce access control on allocator and smart pointer metadata
- Metadata updates validated by comparison with ground truth

Identifying Smart Pointers

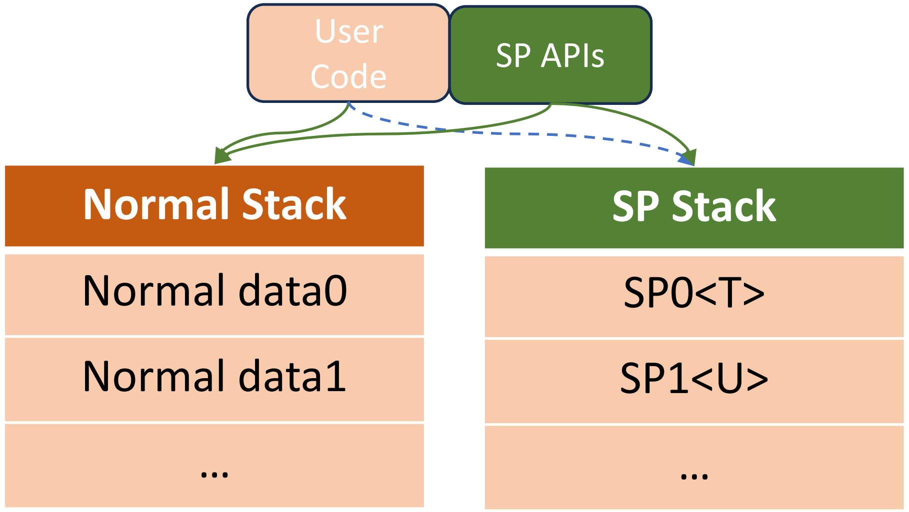
- Identifying Smart Pointers at Compile time
  - Require Smart Pointer Developers to implement a special trait (MetaUpdate)
- Diversity of Smart Pointer types & uses
  - Challenge for authenticating metadata updates
  - The MetaUpdate trait requires implementation of a *validate* function
- Insert calls to validate function after API call that takes mutable sp.

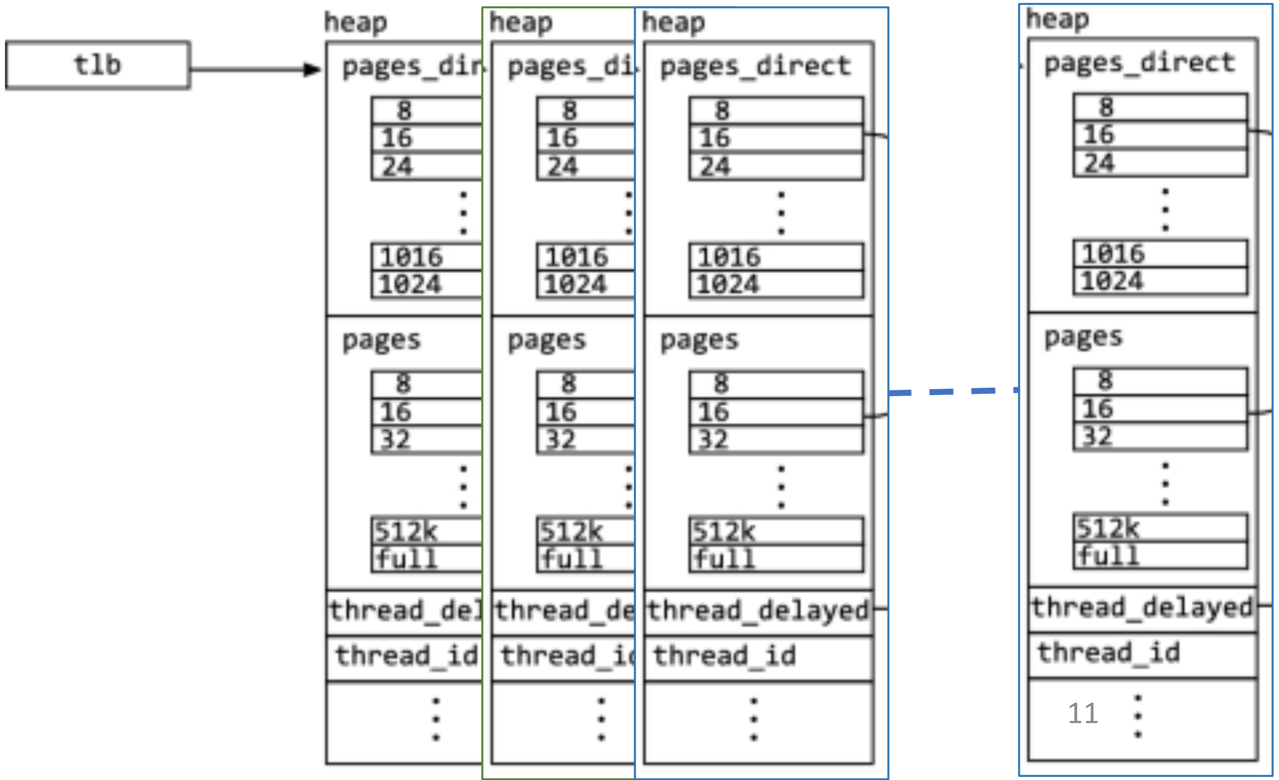
```
impl<T, A> MetaUpdate for Vec<T, A> {
    fn validate(&self) -> bool {
        metasafe::isLive(self.ptr) &&
        metasafe::getSize(self.ptr) >=
        self.capacity()*sizeof(T) &&
        self.capacity() >= self.len()
    }
}
```

```
fn main() {
    let mut buffer = vec![0,2,3]
    unsafe {
        mpk_enable_sp_write();
        buffer.special_set_len(10);
        if !(<Vec<i32> as MetaUpdate>::validate(&buffer)) {
            panic!("METASAFE: Failed to validate Vec<i32>");
        }
    }
}
```

Isolated Storage for Smart Pointers & Metadata

- Separate Compartmentalized Storage for Metadata
  - Stack ➔ Allocate a separate stack for smart pointers (similar to safestack)
  - Heap ➔ Use Allocator with grouping property (Arenas) (tcmalloc, **mimalloc**)
    - Heap[0] for FFI
    - Heap[1] for Smart Pointers
    - Heap[2…TypeN] for user data

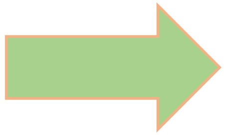




Protecting Smart Pointer Metadata

- Enforcing In-process Isolation.
  - Use Intel-MPK to enforce different access permissions on gated region
- Deciding the boundary of gated region access.
  - Find call sites to smart pointer APIs in application context.
  - Insert WPKRU instructions to enable write before API call.
  - Insert WPKRU instructions to disable write before return inside API function

```
fn main() {
    let mut buffer = vec![0,2,3];
    unsafe {
        buffer.set_len(10);
    }
    println!("Element at 9: {}", buffer[9]);
}
```

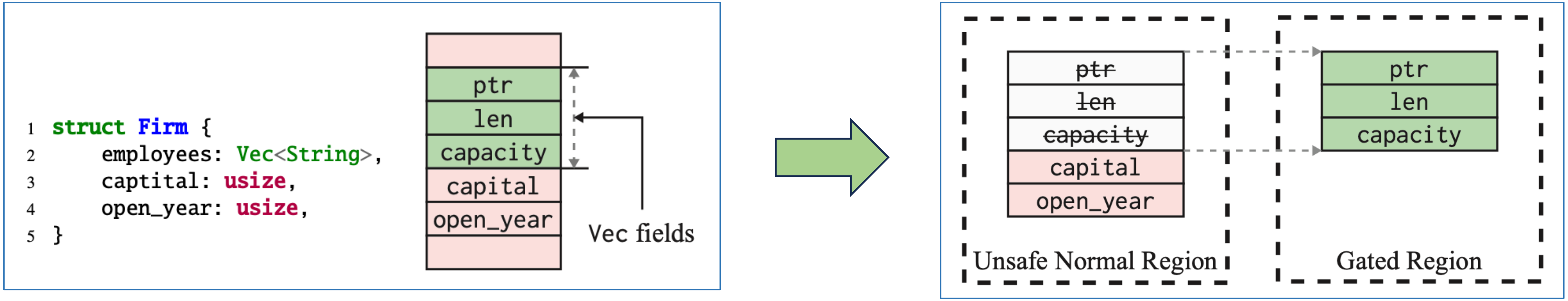


```
fn main() {
    let mut buffer = vec![0,2,3]
    unsafe {
        mpk_enable_sp_write();
        buffer.special_set_len(10);
    }
}

impl<A,T> Vec<A,T> {
    fn special_set_len(&mut self, len: usize) {
        ...
        mpk_disable_sp_write();
    }
}
```

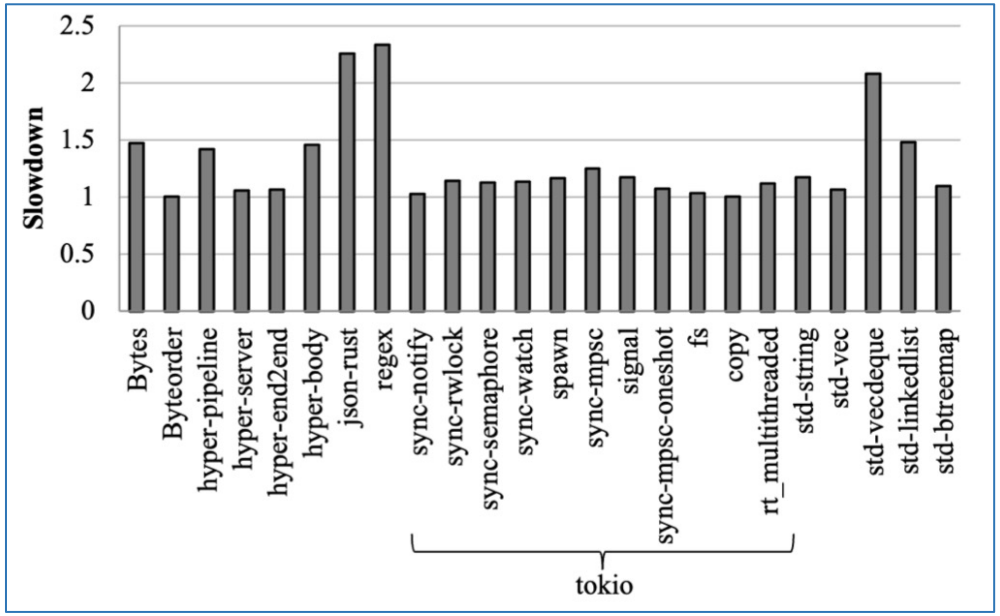
Struct-Inlined Smart Pointers

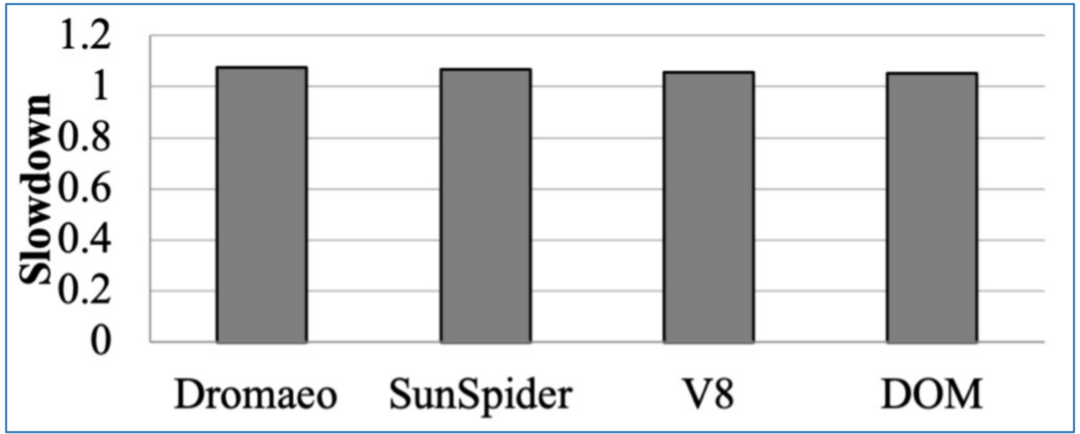
- Protecting In-struct Embedded Smart Pointers
  - How to control access to in-struct embedded smart pointer
    - Treat whole struct as smart pointer ➔ Not safe
    - Treat smart pointer as user-data ➔ Defeats METASAFE
    - Use shadow memory for inlined smart pointers.



Evaluation: Performance

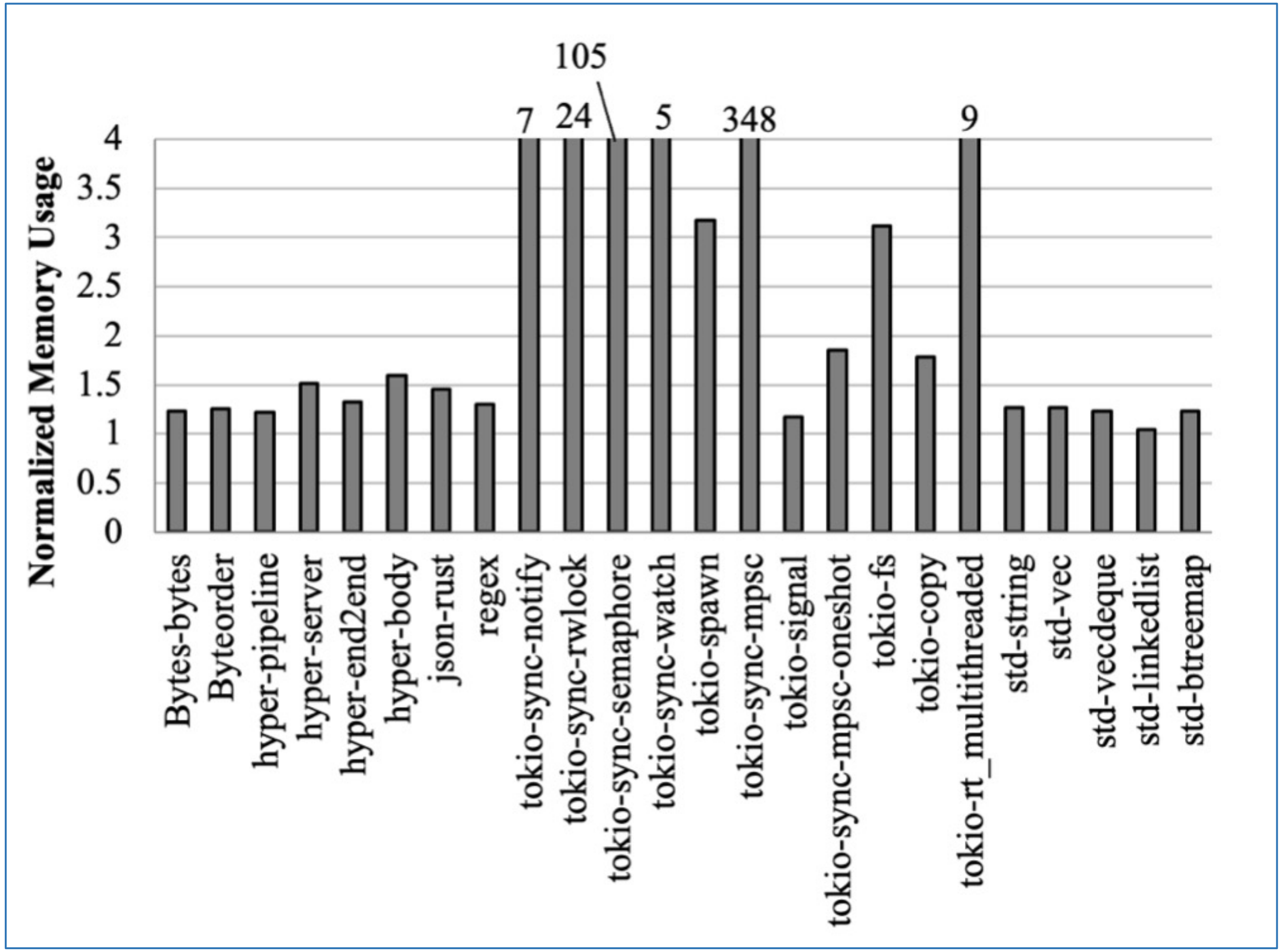
- METASAFE Alone:
  - 25.5% performance overhead on micro benchmarks
  - 3.5% performance overhead on servo browser





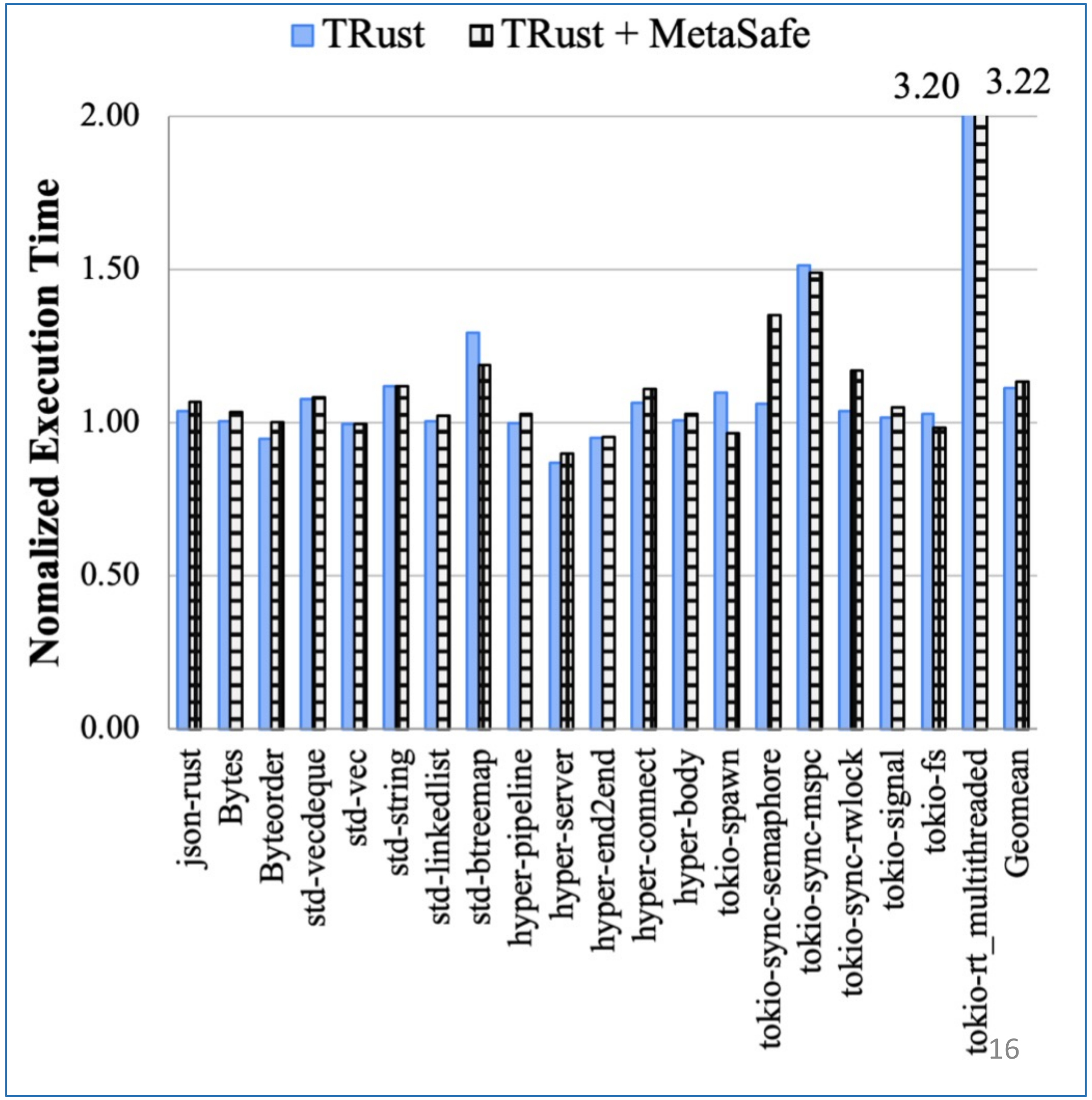
Evaluation: Memory Usage

- METASAFE Alone:
  - 27% Memory usage overhead on single-threaded micro-benchmarks
  - Upto 8x more memory usage for heavily multithreaded micro-benchmarks
    - 31% memory overhead if separate stacks are disabled



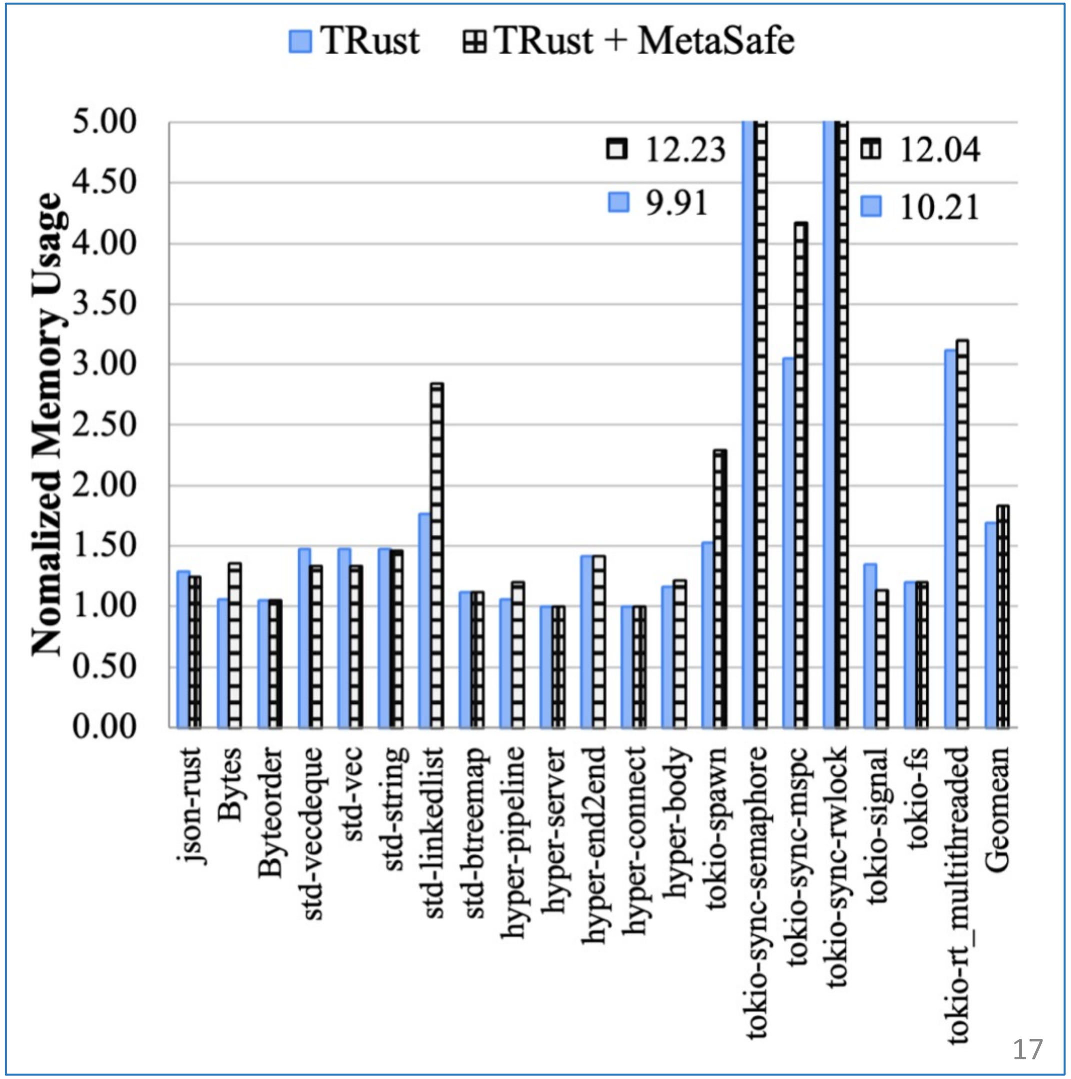
Evaluation: Performance

- METASAFE + TRust
  - TRust: A mechanism for isolating unsafe Rust to protect Safe Rust
  - METASAFE + TRust incurs 13% performance overhead on micro-benchmarks.
  - TRust Alone incurs 11% performance overhead



Evaluation: Memory Usage

- METASAFE + TRust
  - METASAFE + TRust incurs 89% memory overhead
  - TRust alone incurs 69% memory overhead
  - More memory for separating smart pointers



Conclusion

- METASAFE presents a mechanism to recognize and protect smart pointer metadata, thus enhancing Rust memory safety.
- It allows developers to mark smart pointers and provide means of validating metadata updates.
- Relying on Intel-MPK and Compiler instrumentation, METASAFE incurs acceptable runtime overhead for realworld programs such as servo.
- Artifact Open Sourced at:
  https://github.com/seccompgeek/trust23-metsafe24.git
  kayondo/metasafe for built image

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED