

# **Type-Safe Sandboxing of Memory-Unsafe C/C++ Libraries for Rust Programs**

**Jonathan Keller**

**May 20, 2024**

# Background

## Memory Corruption

- In C/C++, memory management is the programmer's responsibility
- Bugs caused by improper memory mismanagement are common and often lead to exploitable security vulnerabilities (ex. buffer overflow, use-after-free)
- Industry consensus: about 70% of security vulnerabilities in systems software are caused by memory corruption

```
char buf[16];
```

```
strcpy(buf, user_input);
```

buf[0]
buf[1]
buf[2]
...
buf[14]
buf[15]
...
user_is_admin
...
return address

# Background

## Memory Safety

- *Memory-safe* programming languages prevent memory corruption at a language level
- Spatial memory safety: carry bounds information with dynamically-sized objects, check at runtime
  - Performance impact of bounds checks negligible on modern CPUs
- Temporal memory safety: more complicated
  - Most languages use garbage collection -- but this is expensive, requires a complex runtime, limits programmer control over memory management
  - Thus, memory safety historically considered infeasible for systems code

# Background

## Rust

- Modern, memory-safe systems programming language
- No garbage collection -- statically verified manual memory management
- High industry interest for applications like operating systems, web browsers, servers, embedded systems, Linux kernel modules
- U.S. federal government (and others) officially recommend using Rust (and other memory-safe languages) rather than C/C++ for new software

# Background

## Rust

- Rust compiler includes a *borrow checker* that statically verifies temporal memory safety
- Object lifetimes tracked as part of the type system; a reference must not outlive the referenced object

```
fn foo(x: &i32) -> &i32 {  
    let i = 5;  
    return &i;  
}
```

**Error:** returned reference outlives  
the local variable it references

```
fn foo(x: &i32) -> &i32 {  
    let i = 5;  
    return &x;  
}
```

**OK:** reference has same lifetime  
as the function parameter

# Background

## Rust

- Two types of references in Rust:
  - Shared/immutable references `&T`
  - Unique/mutable references `&mut T`
- Strict aliasing:
  - An object cannot be modified at all while a **shared** reference is live
  - While a **mutable** reference is live, the object can only be accessed through that reference
- Unsafe raw pointers also available if needed

# Background

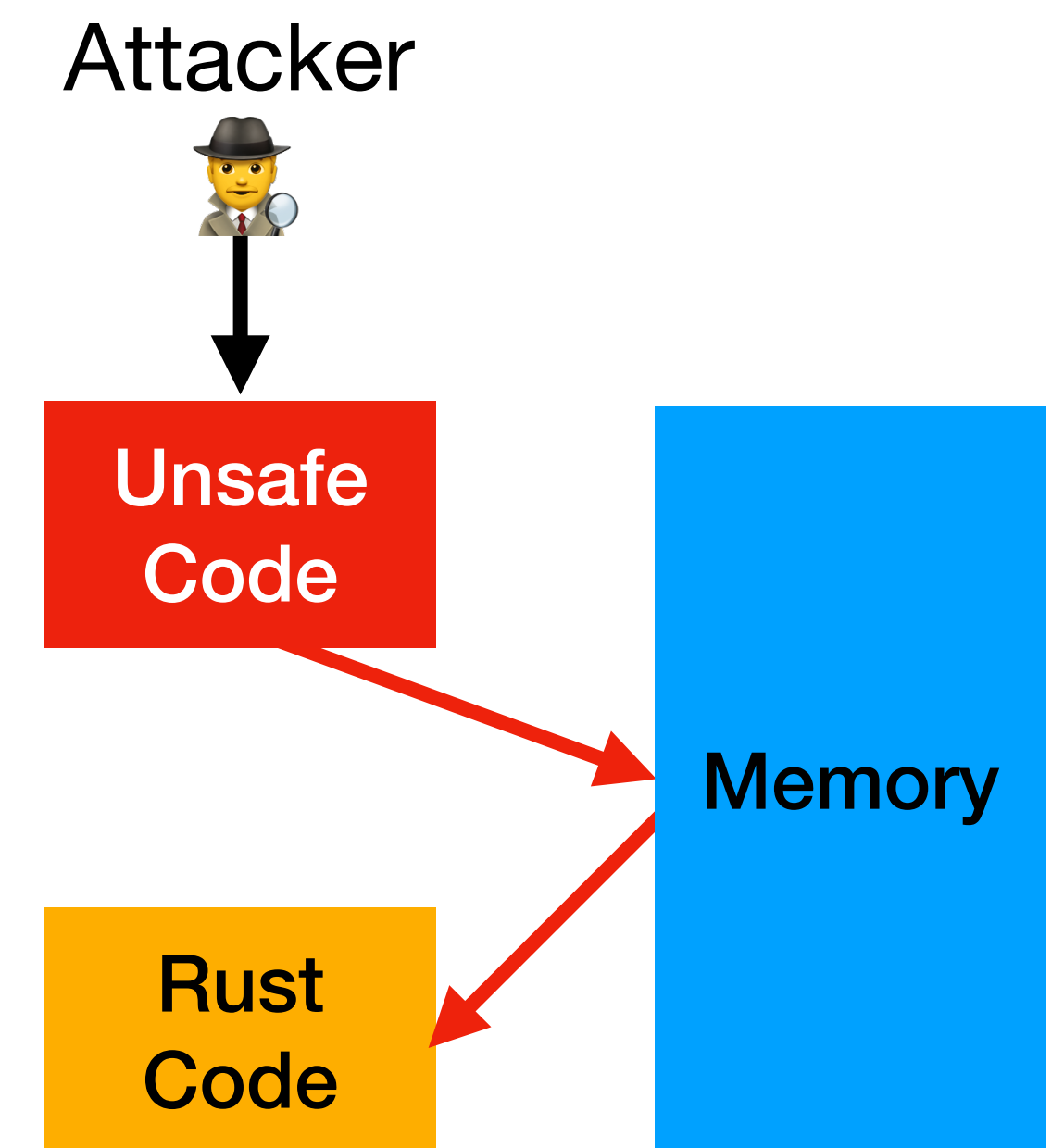
## Unsafe Rust

- Expressing patterns that the borrow checker does not "understand"
- Creating safe interfaces around unsafe data structures
- Interacting with legacy C code
  - Rust has a steep learning curve, rewriting legacy code is expensive
  - Many adopters of Rust are integrating and rewriting gradually, not all at once
  - Thus, it is common for programs to contain a "mixture" of Rust and C code

# Background

## "Mixed" programs

- What happens when a program contains both Rust and C code?
- Rust code is memory-safe, but C code could corrupt memory
- Impact of memory corruption not limited to the C portions -- can corrupt variables, control flow used by Rust as well
- C code can generate invalid values that cause undefined behavior or memory corruption if used by Rust code (e.g. bad pointers)
- For these reasons, calling C functions from Rust code is considered unsafe





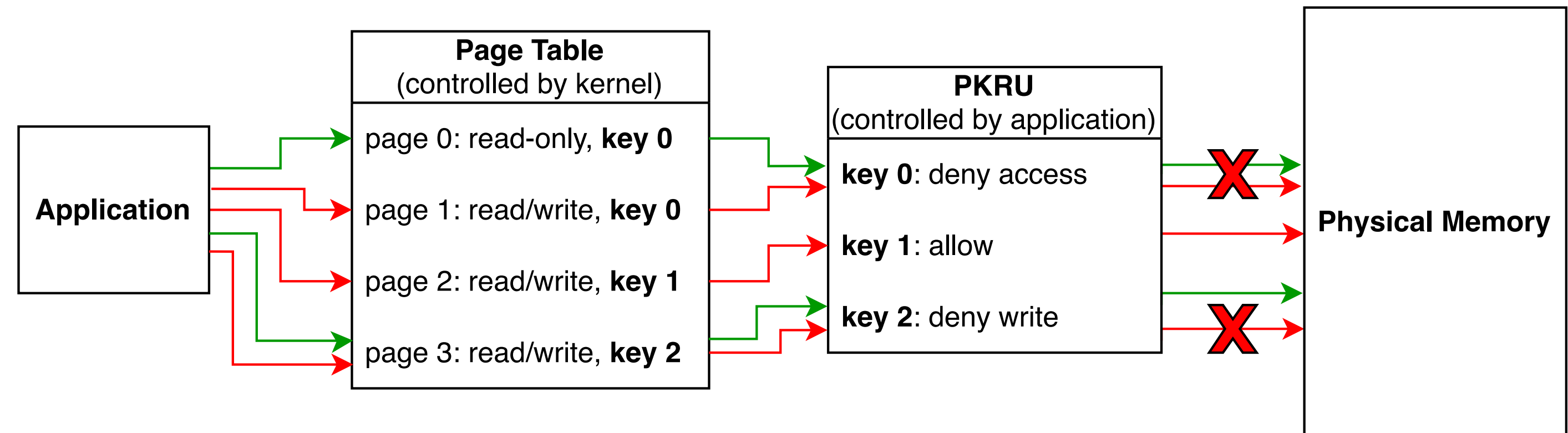
# Existing solution

## Software Fault Isolation

- Separating components of a program such that a fault in one does not corrupt another component's memory
- Various techniques:
  - Process isolation (but context switching and inter-process communication are slow)
  - Binary rewriting (but static analysis is incomplete, dynamic analysis is slow)
  - Memory Protection Keys

# Memory Protection Keys (MPK)

- Introduced by Intel in 2015
- Kernel tags page table entries with 4-bit "keys"



- User programs can disable access to pages protected by each key by writing to protection key register
- No context switch required!

# Fault isolation with MPK

- Much recent research using MPK for fault isolation
  - Park et al., 2019; Vahldiek-Oberwagner et al., 2019; Hedayati et al., 2019; Voulimeneas et al., 2022; Ghoshn et al., 2021; Kirth et al., 2022
- Kirth et al. demonstrate MPK to protect programs containing a mixture of memory-safe and memory-unsafe code
- However, none of this research handles induced memory-unsafety; assumes safe code is correctly written to handle invalid values generated by unsafe code

# Our work

## Goals

- Automatic isolation of a C library in a Rust program
- Secure: no memory-corruption behavior within the C portion can corrupt variables or control flow of the Rust portion
- Enforce validation/safe access when Rust code accesses data generated by C code
- Low overhead
  - No performance impact on Rust code
  - Little to no performance impact on C code
  - Small performance overhead acceptable at boundaries

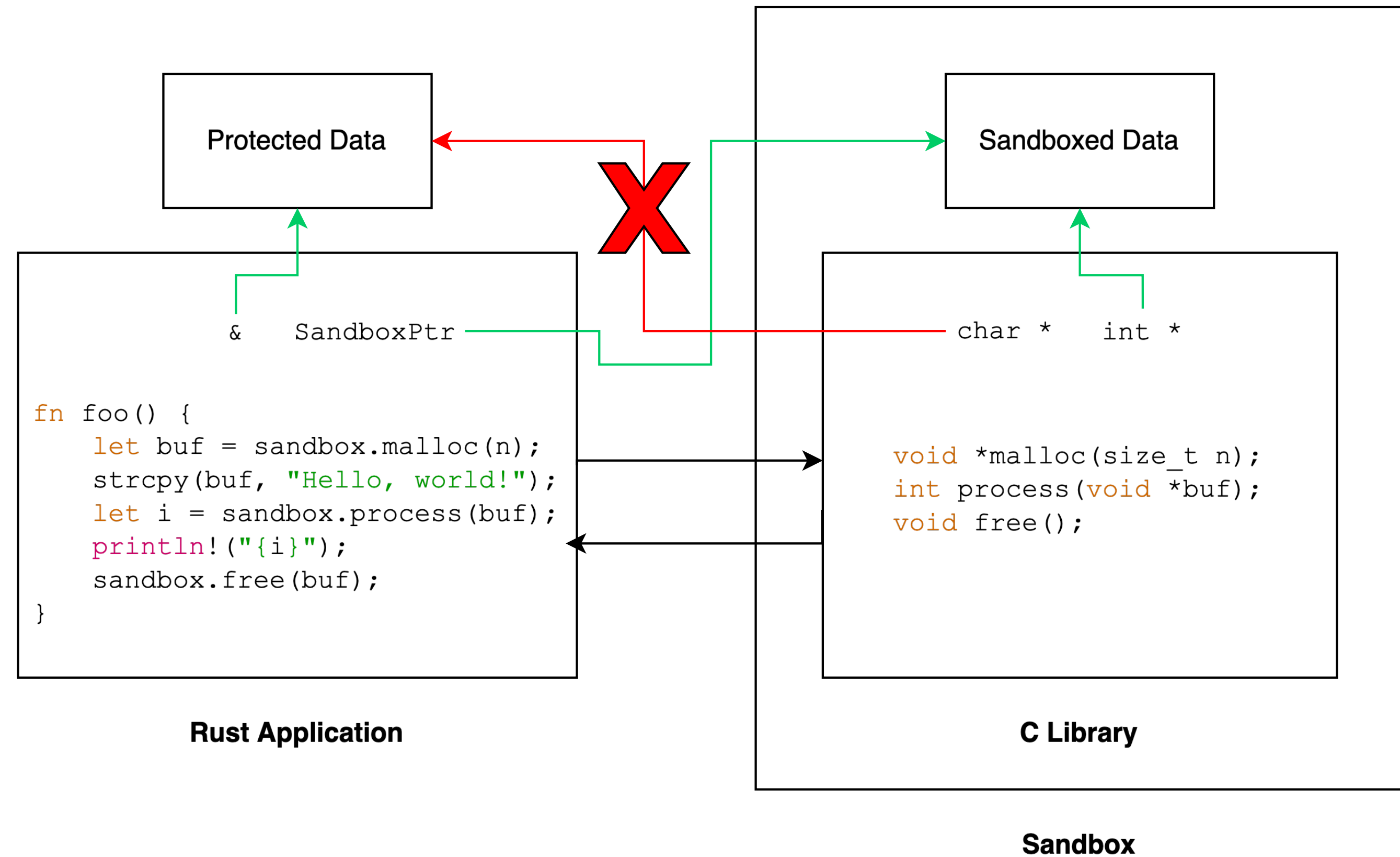
# Our work

## Non-goals

- Minimal code changes OK; we want to replace complex unsafe interactions with a simple, safe API
- We only consider a single-threaded environment
- We only consider Rust code calling C functions, not the other way around
- We don't consider the effects of system calls

# Our work

- MPK "sandboxes" C code and limits its access to memory
- Type-safe, validated references allow Rust code to safely access sandboxed memory
- Modified bindgen generates a **safe** Rust interface to C code



# Sandboxing with MPK

- Memory space divided into two regions using protection keys
- Sandboxed region always accessible
- Protected region writable only when running Rust code
  - C code has read-only access to protected region so it can access e.g. global offset table
  - Not a limitation of our design; could be worked around to improve confidentiality
- Configure PKRU registers on entry/exit to sandboxed code

# Sandboxing with MPK

## Protecting against code-reuse attacks

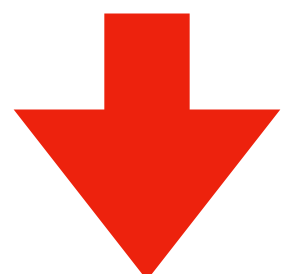
- An attacker who corrupts control flow of sandboxed code may trigger execution of a wrpkru instruction to disable the sandbox
- W<sup>X</sup> protects against injecting wrpkru through shellcode, but an attacker may reuse an existing wrpkru instruction in the binary
- Sandbox entry/exit points must be carefully written to defeat code reuse attacks
- The program should not contain wrpkru instructions anywhere else...or should it?



# Sandboxing with MPK

## Code reuse with misaligned instructions

- x86 instructions are variable length
- An attacker may be able to find a misaligned byte sequence that decodes to wrpkru
- Solutions:
  - Control-flow integrity techniques (e.g. Intel CET)
  - Binary rewriting (Vahldiek-Oberwagner et al., 2019)
  - Hardware watchpoints (Hedayati et al., 2019)



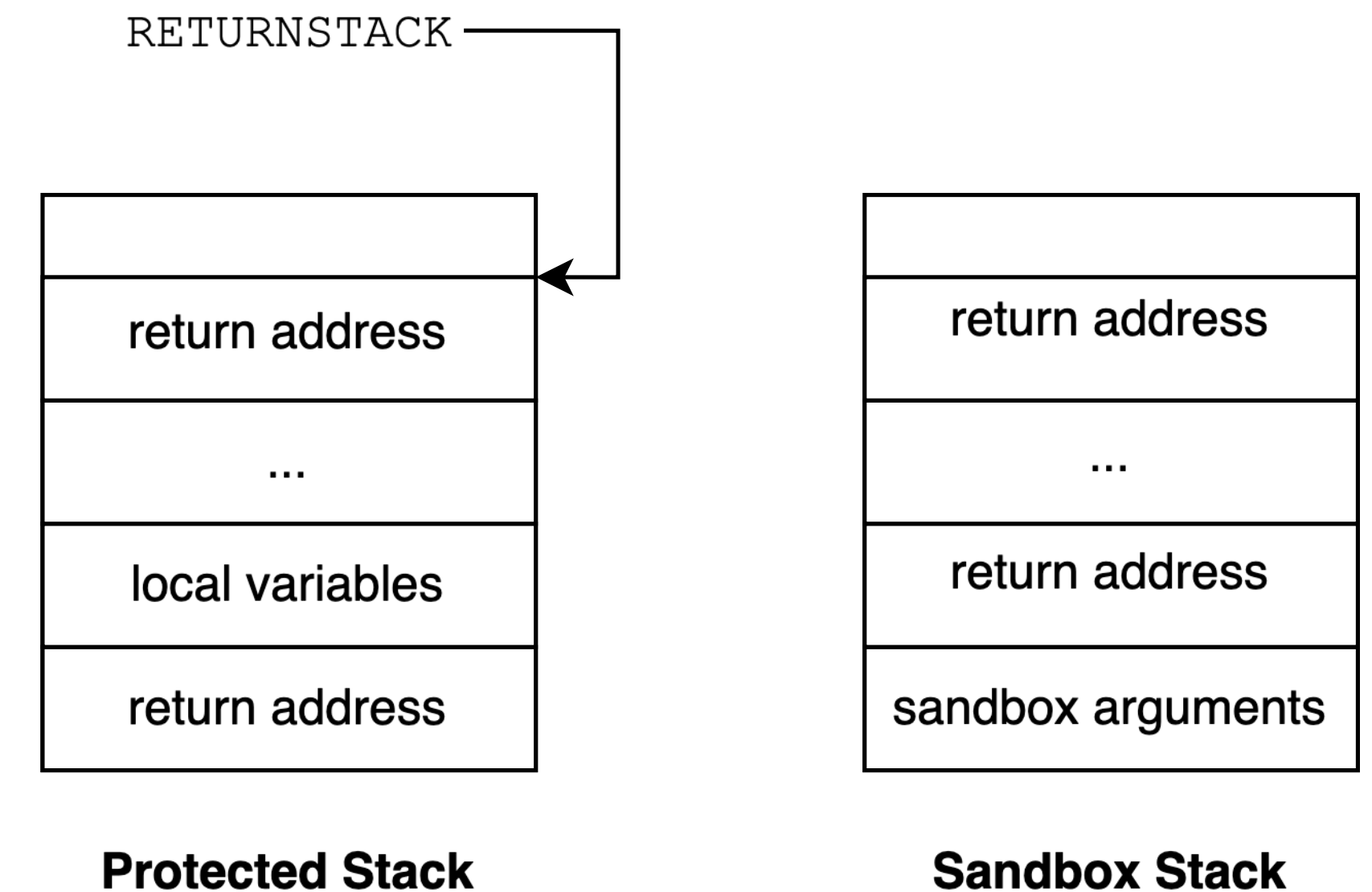
```
49 03 0f: add rcx, [r15]
01 ef:      add edi, ebp

0f 01 ef: wrpkru
```

# Sandboxing with MPK

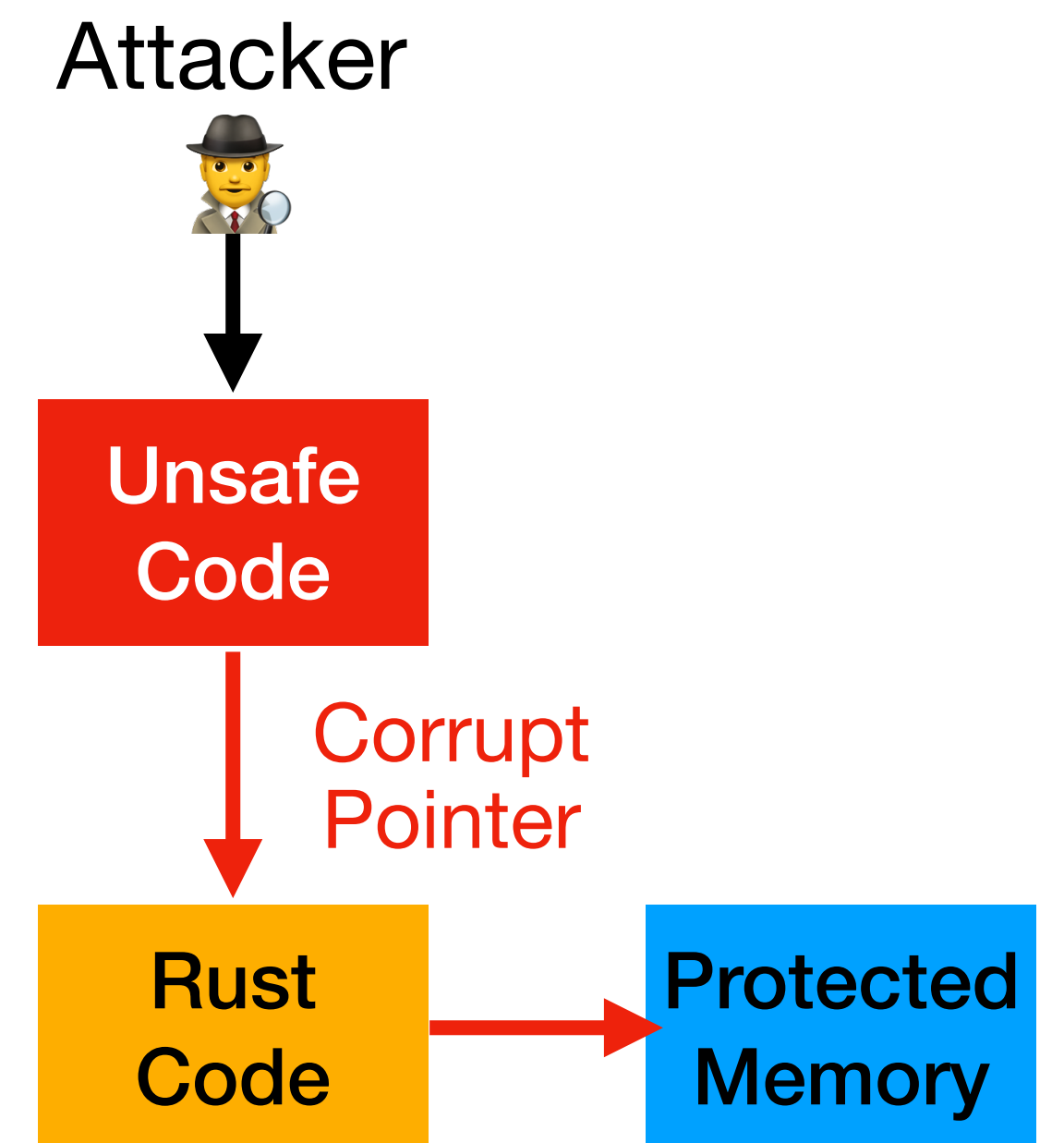
## Stack Switching

- To prevent stack corruption attacks, sandboxed code uses its a separate stack
- When calling a sandboxed function, parameters and return values are stored on the bottom of the sandbox stack
- Rust stack pointer is stored in a global variable that C code cannot write (enforced by MPK)



# Safe Access to Sandboxed Memory

- When Rust code accesses values returned from the sandbox, there is potential for those values to be corrupt/attacker-controlled
- Pointers may be null, misaligned, point to invalid memory, point to protected memory used by Rust code, or violate Rust's aliasing rules
- An attacker might "trick" Rust code into corrupting its own memory
- Thus, we introduce a checked reference type



# Safe Access to Sandboxed Memory

## The `SandboxPtr` type

- Represents a pointer to memory "owned" by the sandbox
- Dynamically checks pointer validity, bounds, alignment
- Statically enforces Rust aliasing rules
  - Can be converted to a safe Rust reference, as long as:
    - Only one mutable reference to sandboxed memory exists at any one time
    - No sandboxed functions are called while a `SandboxPtr` reference is live
- These restrictions could be relaxed with dynamically-checked aliasing

# Generating Safe Bindings

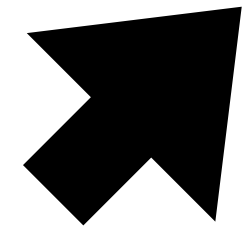
- Rust's `bindgen` procedural macro generates Rust bindings to C code
- We modified `bindgen` to:
  - Configure MPK upon entering/exiting C functions
  - Convert C pointers to/from checked `SandboxPtr` types
- Generated bindings no longer need to be marked `unsafe` (memory corruption within C code is no longer a safety issue on the Rust side)

# Generating Safe Bindings

## Example

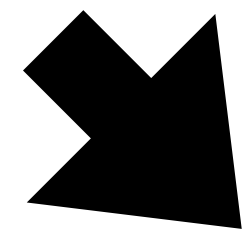
```
const int *foo(int *p);
```

C Function Signature



```
extern "C" {  
    fn foo(p: *mut ::std::os::raw::c_int) -> *const ::std::os::raw::c_int;  
}
```

Generated Rust Binding (standard bindgen)

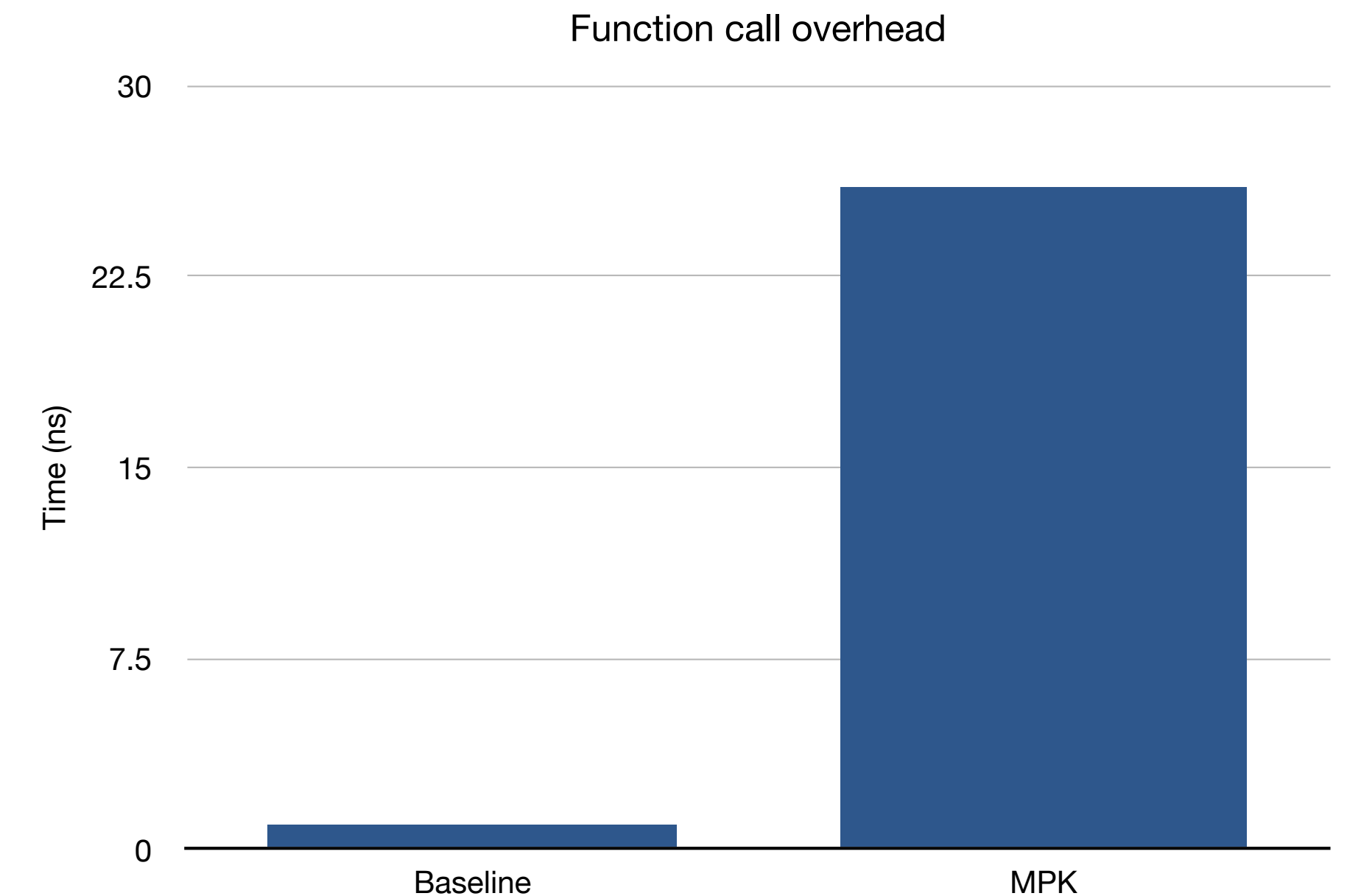


```
impl Sandboxed {  
    pub fn foo(  
        &mut self,  
        p: mpk::SandboxPtrMut<::std::os::raw::c_int>,  
    ) -> mpk::SandboxPtr<::std::os::raw::c_int> {  
        extern "C" {  
            fn foo(p: *mut ::std::os::raw::c_int) -> *const ::std::os::raw::c_int;  
        }  
        unsafe { mpk::SandboxPtr::new(self.0.call(move || foo(p.get()))) }  
    }  
}
```

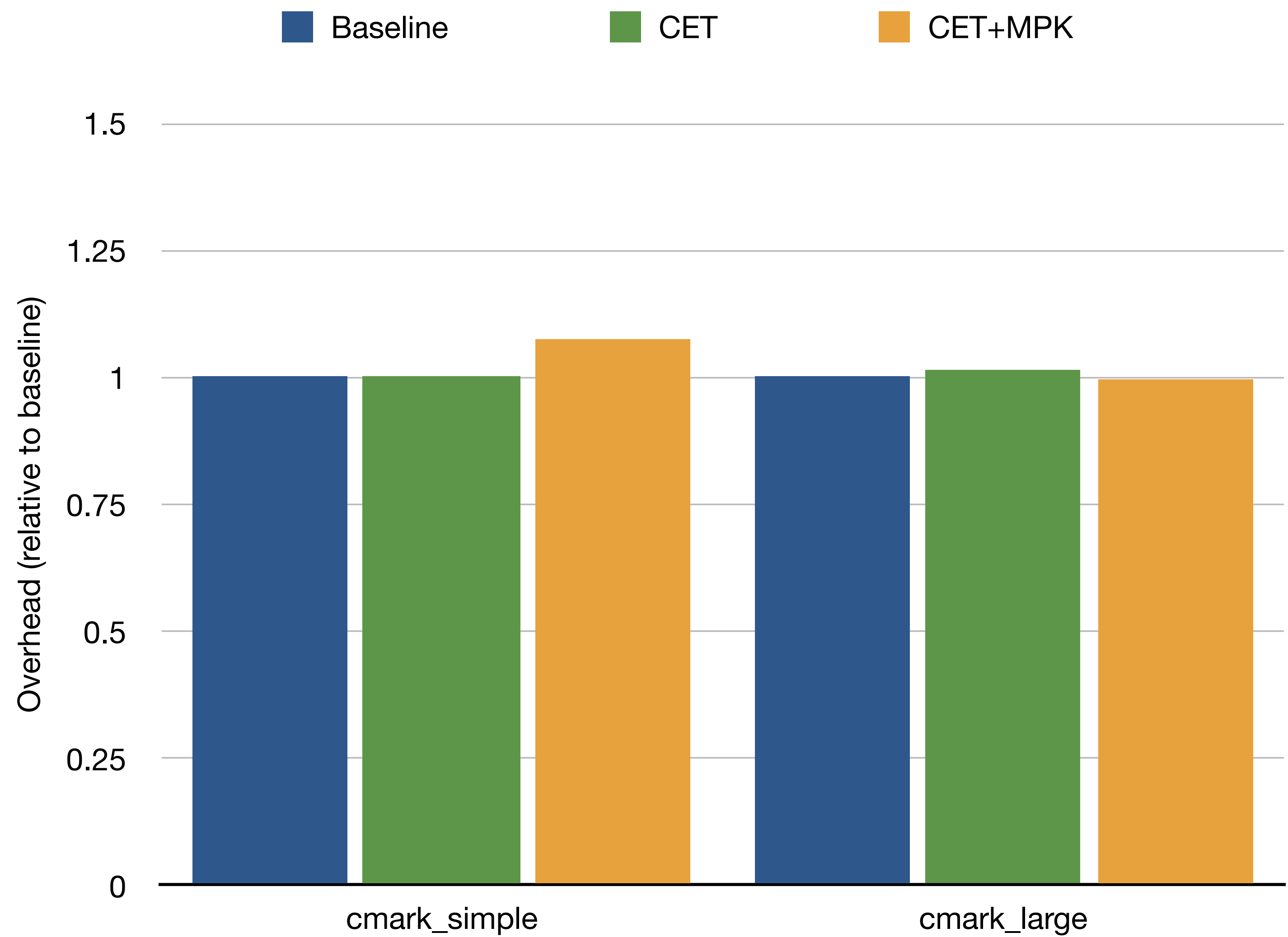
Generated Rust Binding (modified bindgen)

# Performance

- Only source of overhead is configuring MPK on sandbox boundaries
- ~25ns constant overhead per function call (when Rust code calls a C function)
- Benchmarks conducted with Rust's cargo bench tool
- Test hardware: Intel NUC, i5-1135G7 @ 2.4 GHz



# Performance





# Limitations and future directions

- Need to consider system calls for full security
  - Can combine our techniques with existing research in system call sandboxing
- Support for multithreaded environments, dynamic linking, etc.
- libc (and other libraries shared by Rust and C code)
  - How to protect C code from corrupting memory used by libc?
  - Most promising: two implementations of libc

# Questions?