Keeping Safe Rust Safe with Galeed

Elijah Rivera[2], Samuel Mergendahl[1], Howard Shrobe[2],
Hamed Okhravi[1], and Nathan Burow[1]

1: MIT Lincoln Laboratory 2: MIT CSAIL

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

1951–2021
LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

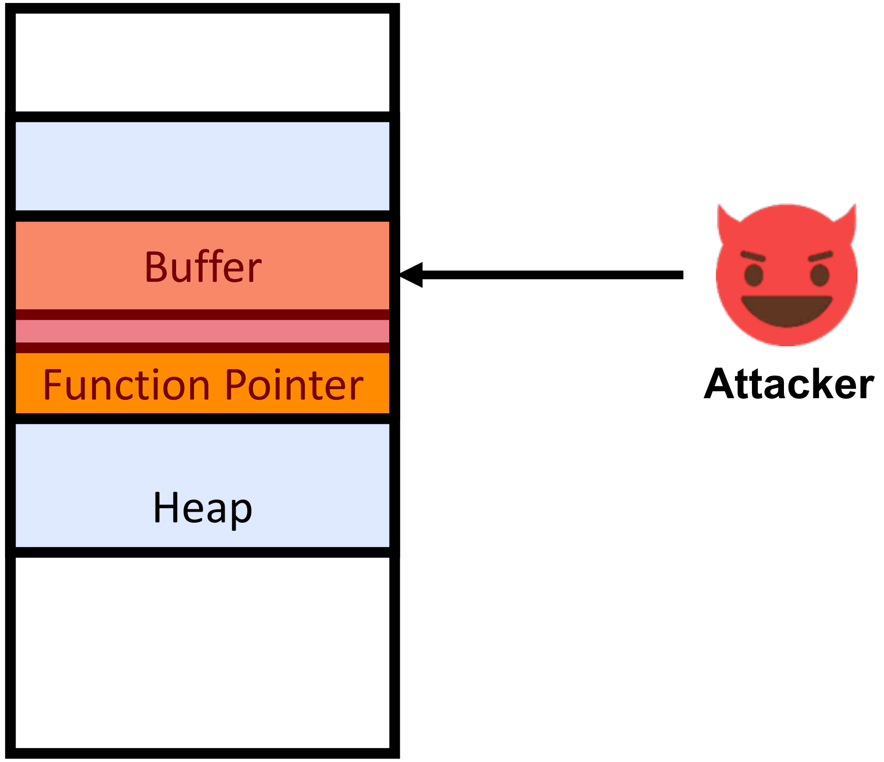
Bottom-Line Upfront

- Unsafe languages like C/C++ contributed to a large fraction of vulnerabilities
- Codebases have started ***incrementally*** porting to safe languages like Rust
- We show that:

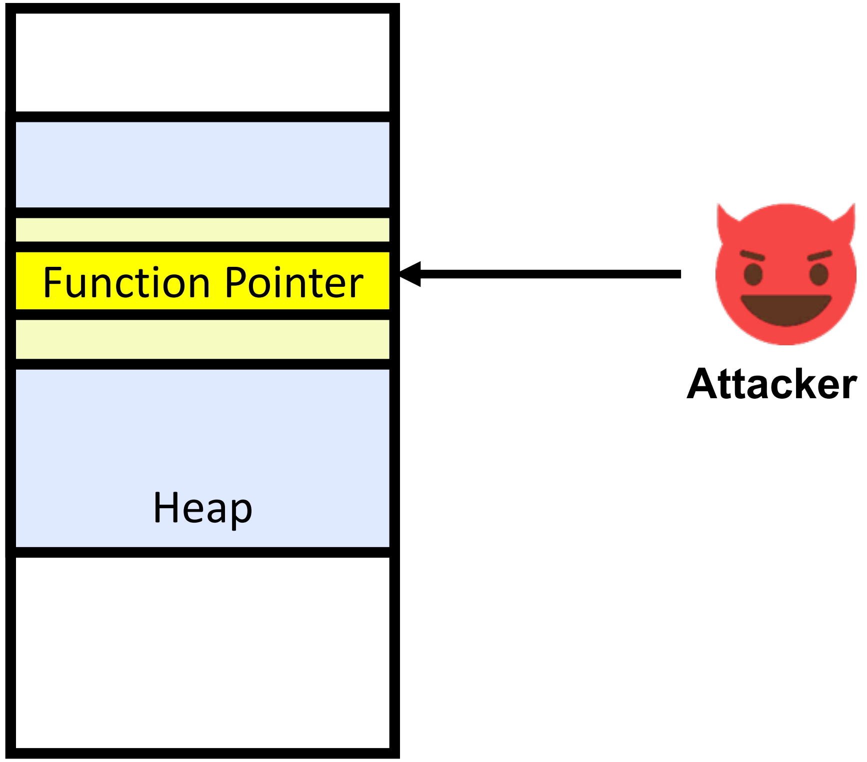
Incremental deployment of safe languages ≠ Incremental security

- Code in an unsafe language can break the safety of code in a safe language
- We design, implement, and evaluate Galeed to:
  - ***Prevent unintended*** interactions between languages
  - ***Secure intentional*** interactions between languages
- Galeed **keeps Safe Rust Safe**

Memory Corruption Attacks



Spatial Memory Violation



Temporal Memory Violation

Memory Corruption is a Solved Problem; Right?

Geek, News

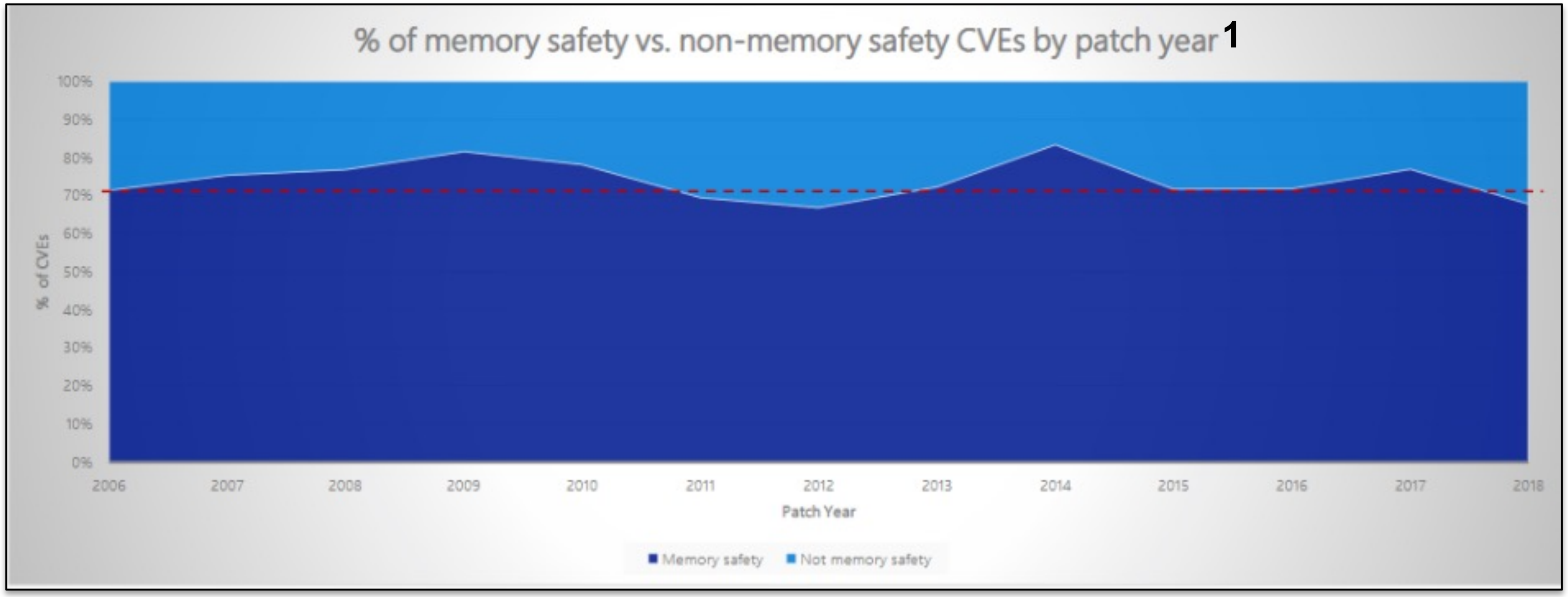
Google Says 70% Security Bugs In Chrome Are 'Memory Safety Problems'

TWEET WHATSAPP SHARE

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

By Catalin Cimpanu for Zero Day | February 11, 2019 -- 15:48 GMT (07:48 PST) | Topic: Security

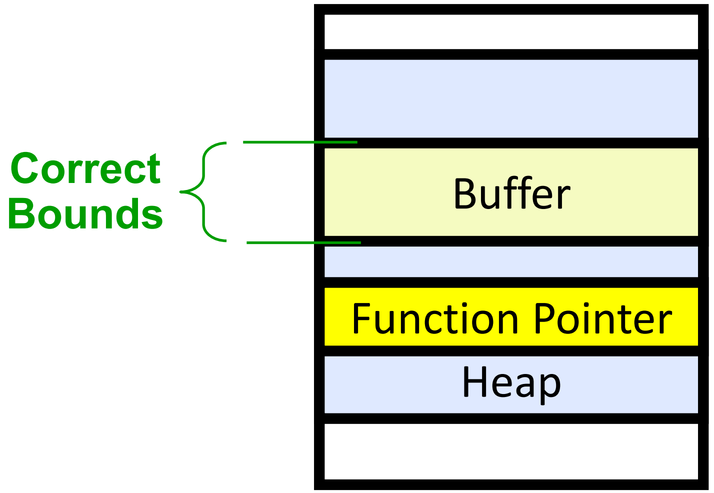


1: Source: Matt Miller (Microsoft Security Response Center), "Trends, challenge, and shifts in software vulnerability mitigation landscape", BlueHat IL, 2019

Rust: Memory-Safe Programming Language

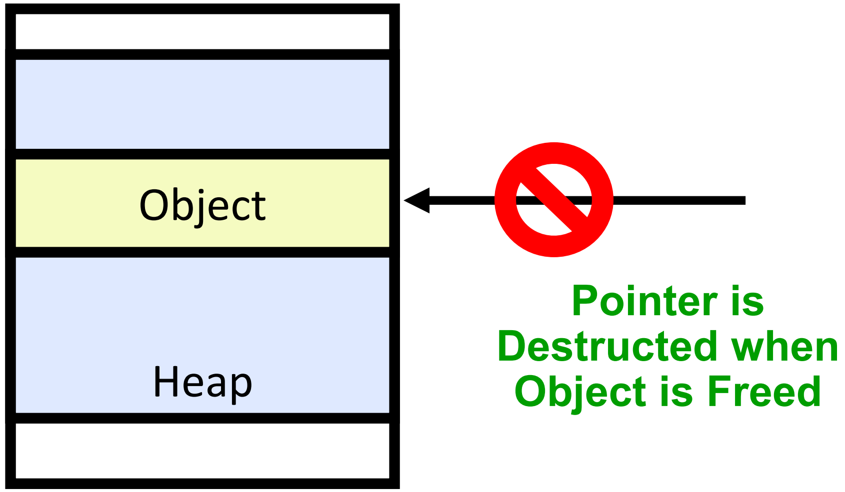
- A systems programming language that is memory-safe
- Small language runtime: is translated to instructions directly; no need for language VMs

- Spatial safety (no buffer overflows):
  - Statically-sized objects: compile-time checks
  - Dynamically-sized objects: runtime bounds checks



Spatial Memory Safety

- Temporal safety (no use-after-frees):
  - Ownership: only one owner of object at a time
  - Burrowing: ownership can be temporarily transferred



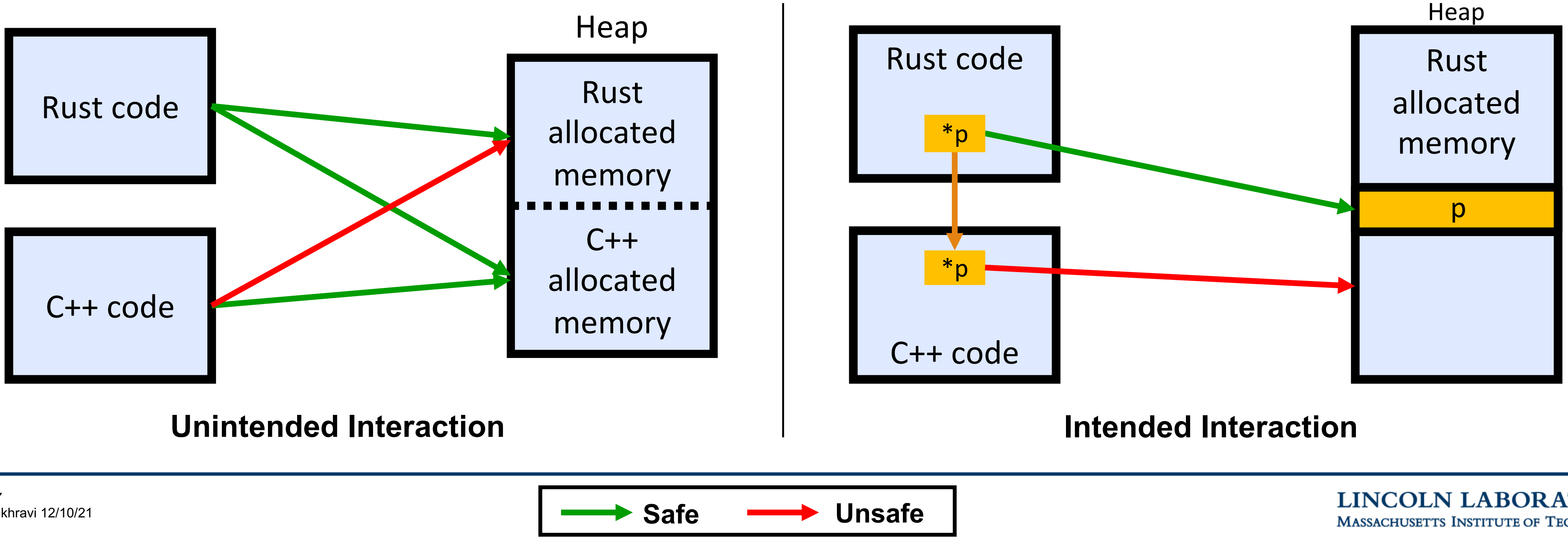
Temporal Memory Safety

Focus on Safe Rust

- **Rust's checks can be disabled by using the `unsafe{}` keyword**
- **Done when Rust's checks are too restrictive**
- **Example: manipulating raw bits for interfacing with hardware devices in device drivers**
- **Unsafe Rust is trivially vulnerable to memory corruption like C/C++**
- **We focus on Safe Rust**

Problem Statement

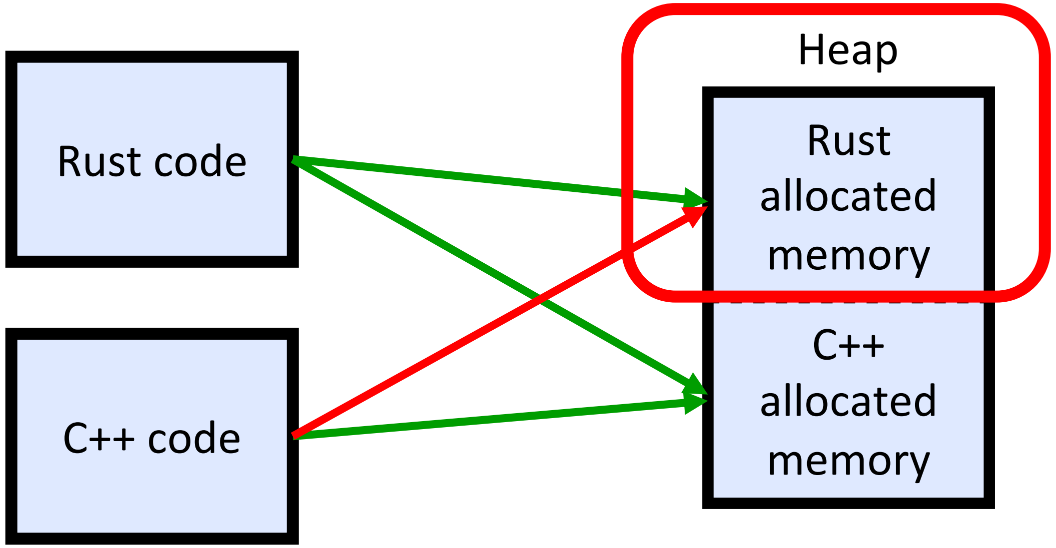
- **All C/C++ code cannot be immediately ported to Rust**
- **Real codebases *<u>incrementally</u>* port to Rust**
- **Rust code often exists alongside other languages, primarily C/C++**
- **Examples: Mozilla (Firefox), DropBox, Microsoft, Amazon, Discord, Facebook, etc.**



Sketch of Our Solution: Galeed

Component 1: Heap Isolation

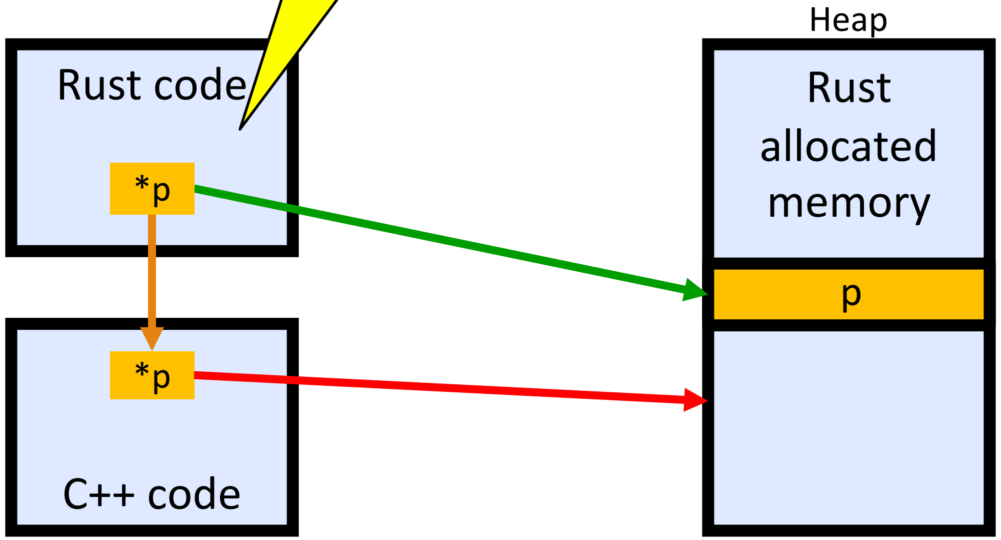
Need to isolate Rust heap when running C++ code → Heap Isolation



Unintended Interaction

Component 2: Pseudo-Pointers

Need to avoid passing actual pointers to C++ → Pseudo-Pointers

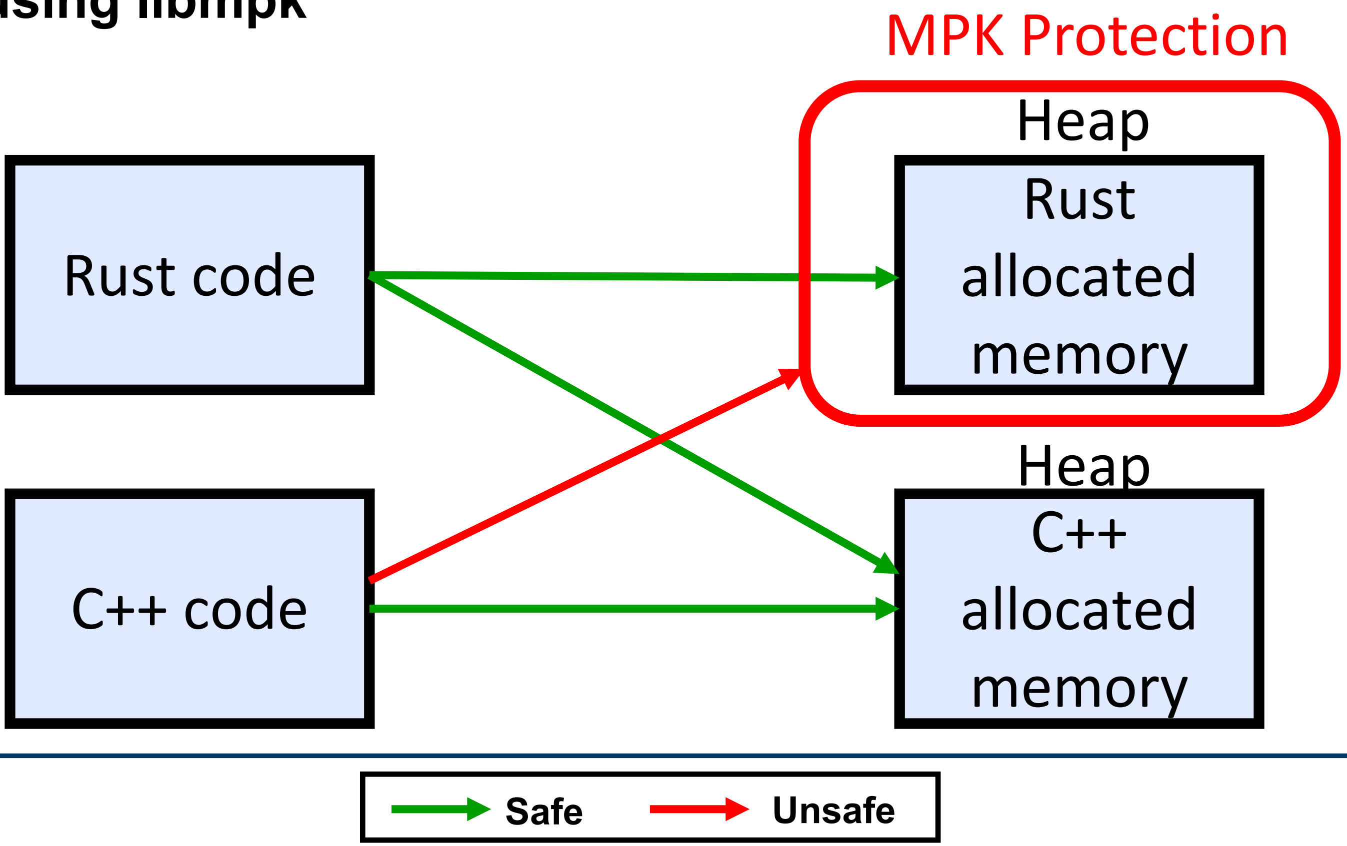


Intended Interaction

Safe **Unsafe**

Galeed Heap Isolation: Preventing Unintended Interactions

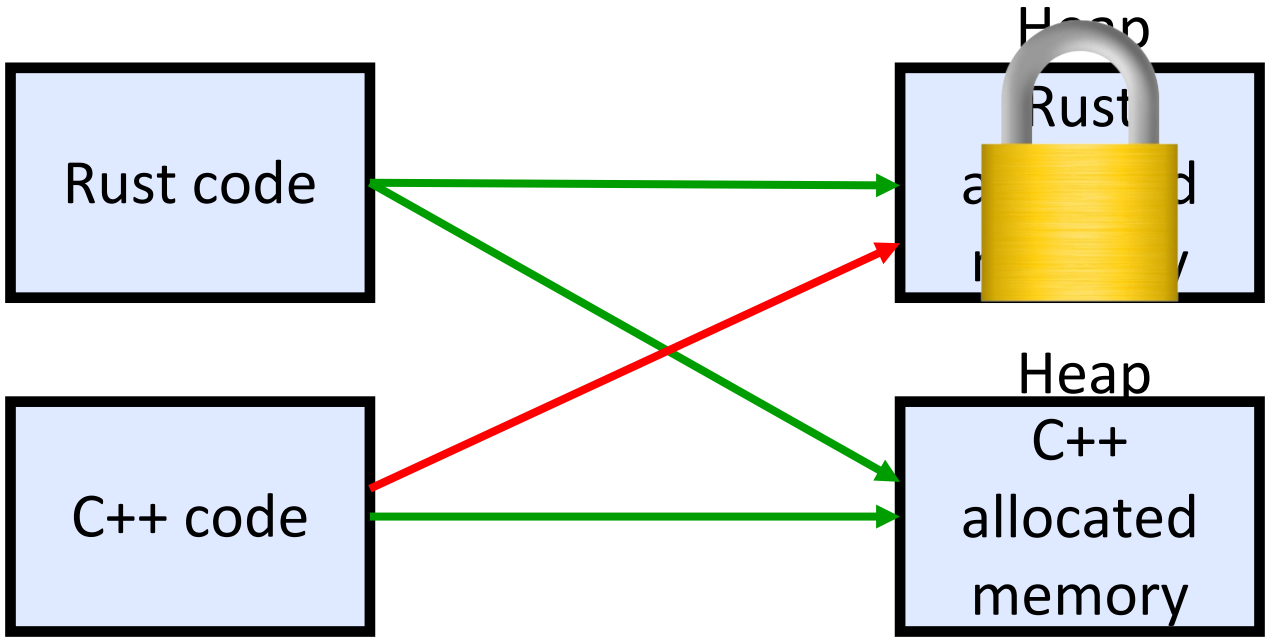
- Uses Intel Memory Protection Keys (MPK) to isolate Rust heap from C++ heap
- Modified Rust standard allocator
- Code to switch permission included around all external call sites
- Implemented using libmpk



Heap Isolation Implementation

Permission Switching Code

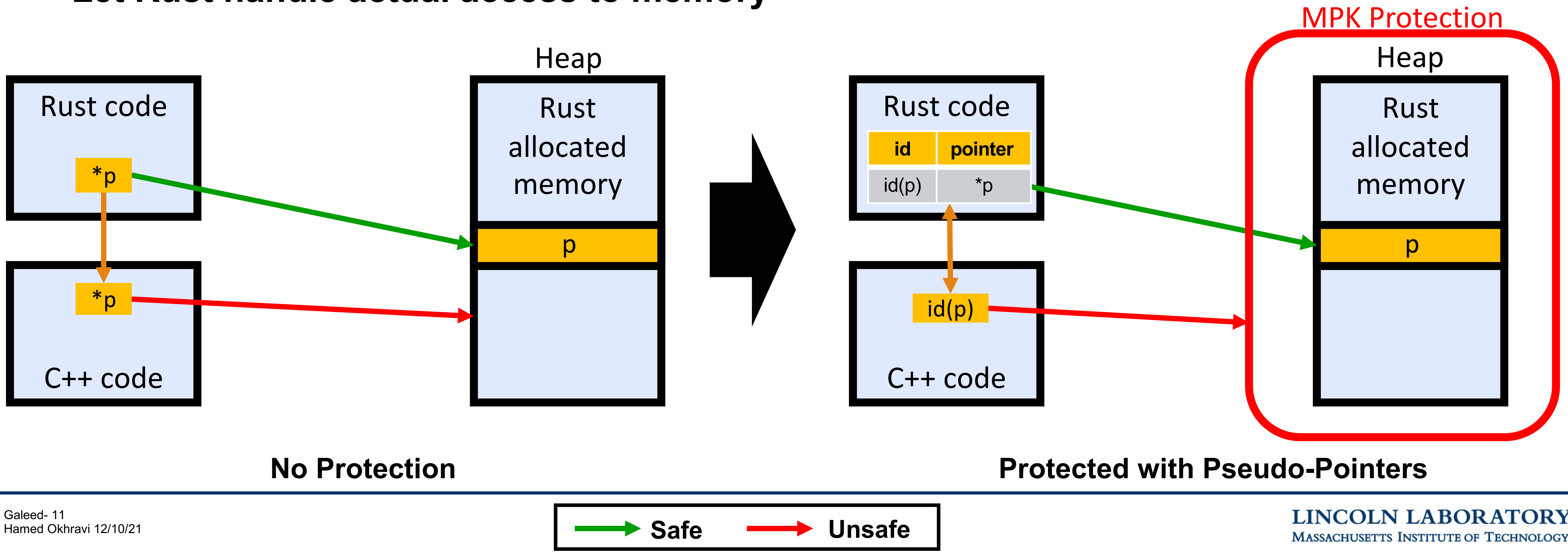
```
asm! (" rdpkru ", in(" ecx") ecx , lateout (" eax") eax , lateout (" edx") _);
eax = ( eax & !PKRU_DISABLE_ALL ) | PKRU_ALLOW_READ ;
asm! (" wrpkru ", in(" eax") eax , in(" ecx") ecx , in(" edx ") edx );
```



Safe **Unsafe**

Galeed Pseudo-Pointers: Securing Intended Interactions

- Replace real pointers with pseudo-pointers (identifiers)
- Pass pseudo-pointers to C++
- Replace C++ pointer operations with calls to getter/setter methods (an LLVM pass)
- Let Rust handle actual access to memory



Pseudo-Pointer Implementation

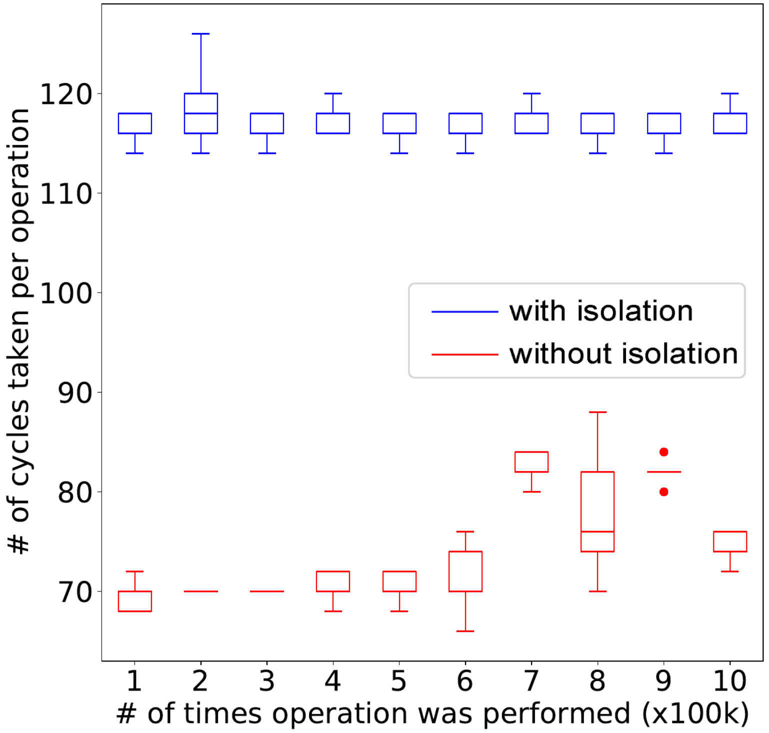
```
int add5 ( MyStruct * const p) {
        p->x += 5;
}
```

No Protection

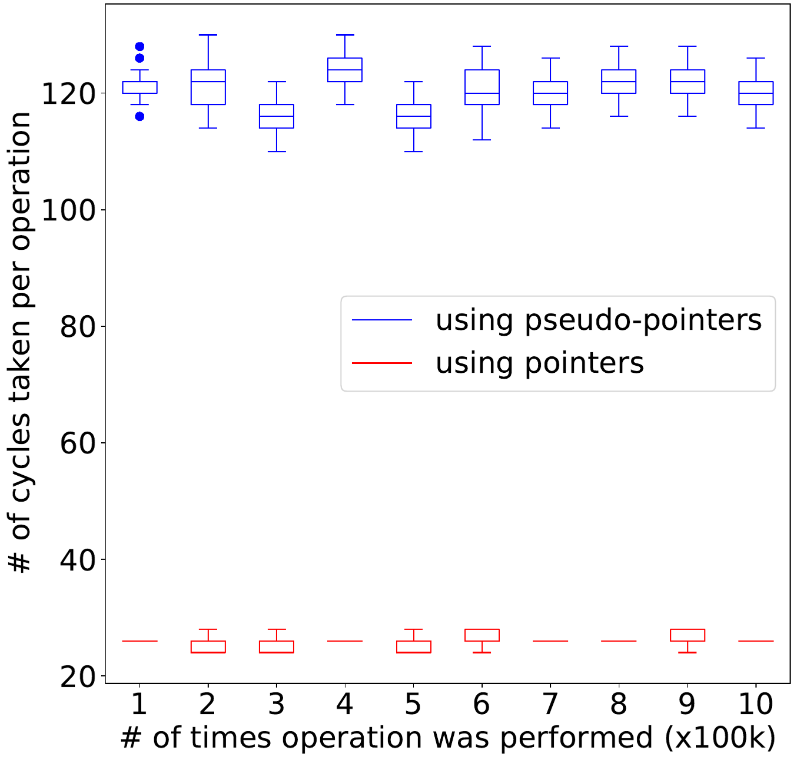
```
int add5 (ID < MyStruct > const p) {
        x = get_x_in_MyStruct (p);
        set_x_in_MyStruct (p, x +5);
}
```

Protected with Pseudo-Pointers

Evaluation: Micro-Benchmarking

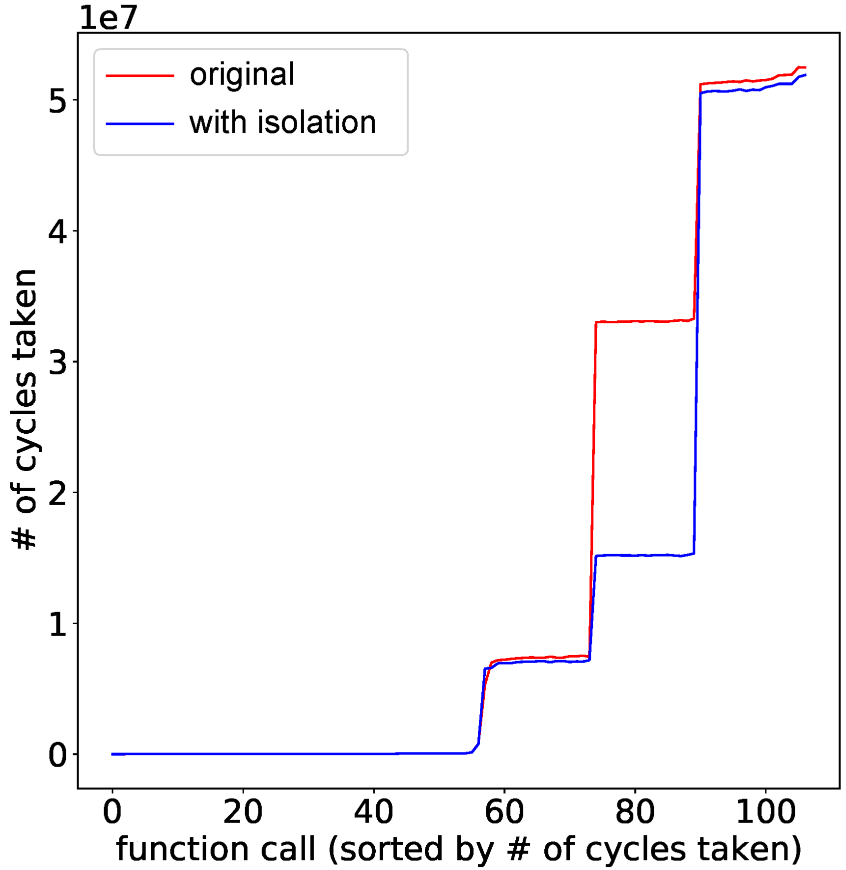


Heap Isolation
Average ~50 cycles

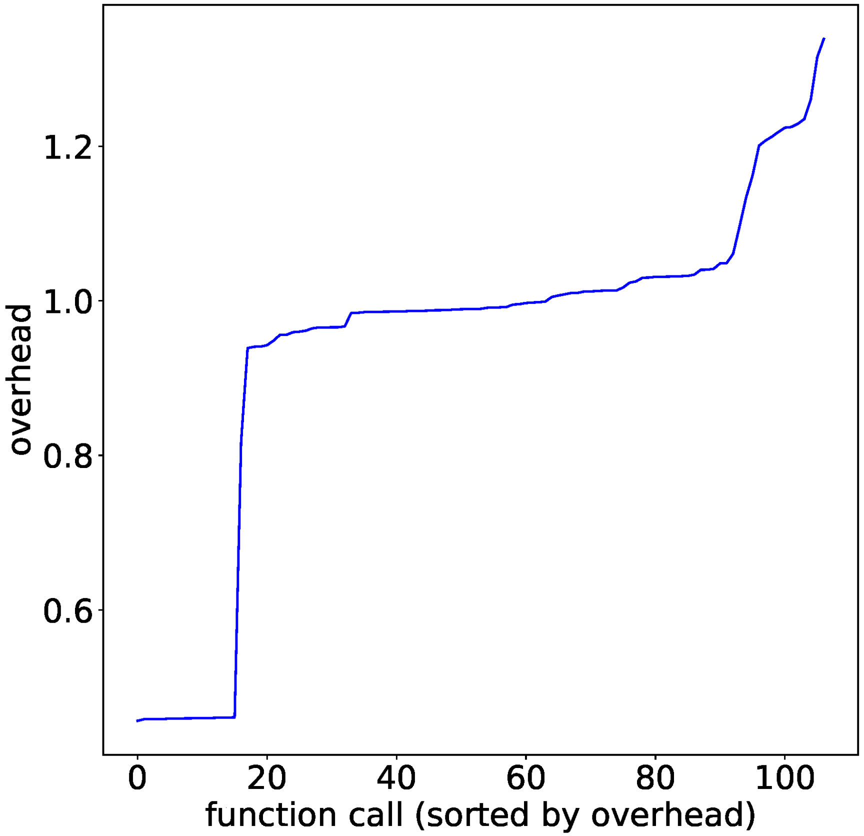


Pseudo-Pointers
Average ~100 cycles

Evaluation: Macro-Benchmarking on Firefox libperf



Cycle Overhead (count)



Runtime Overhead (%)

Lessons Learned

- Rust is being actively developed; releases matter
- Inline assembly still only available in “nightly” builds
- Current MPK interfaces are in C and un-optimized; there is a need for implementing them safely and optimally
- Mixed-language application security is a growing problem and an open area of research

Conclusion

- **Incrementally deploying Rust does not necessarily mean incremental security**
- **Unsafe components of an application can endanger safe components**
- **Galeed prevents unintended interactions**
- **Galeed also secures intended interactions**
- **There is significant space for new research in this area**