TRust: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code

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

USENIX Security 2023

SEOUL NATIONAL UNIVERSITY

UNIST

Rust, memory safe language replacing C/C++

- Rust was invented to help developers build highly safe systems
  - Increasingly popular in industry
  - Rust has been merged into the mainline linux (Oct, 2022)
- Focused on Memory Safety
  - Mostly compile time checks
- Blazingly Fast
  - Better than C/C++ optimized for safety
- Targeted at replacing C/C++, and possibly Python, Java in some aspects

Memory safety in Rust

- Memory Policies
  - Rust compiler won't allow variable temp to dereference the heap object buffer is pointing to
  - Helps Rust compiler detect memory safety violations

```
void* buffer = malloc(SIZE);
void* temp = buffer; X
```

- Guarantee memory safety with compile time and runtime checks
  - Type Checks
  - Bounds Checks

```
char* string = "string";
*string = 'S'; X
```

Unchecked codes threatening memory safety

- Unsafe block
  - Breakage of some Rust memory policies for expressiveness and performance issue
    - Pointer arithmetic on raw pointer address
    - Direct manipulation of metadata of Rust data structure
- External libraries written in foreign language
  - Rust can be mixed with libraries written in other languages
    - Calling libc function in Rust
    - Assembly codes for low level programming
    - Source code is not always available and can be served as binary executables

Example: vulnerabilities in unchecked codes

```
fn main(){
        let array = [1,2,3,4,5];
        let secret_code = 12345;
        unsafe{
                let ptr = array.as_ptr().offset(10);
                std::ptr::write(ptr, 10); }
}
```

Overflow associated with unsafe Rust

```
fn rust_fn(cb_fptr: fn(&mut i64)){
    let fptr: /*Function pointer*/
    unsafe { vuln_fn() }

    fptr();
}
```

Left: Rust code calling C written function[1)]

```
void vuln_fn() {
    int64_t a[1] = {0};
    int64_t array_index = 47; // value
    set by corruptible source
    int64_t array_value = get_attack();

    // Arbitrary Write to Rust fptr
    a[array_index] = array_value;
}
```

Right: C code overwriting Rust function pointer

Mergendahl, S., Burow, N., & Okhravi, H. Cross-language attacks. *NDSS* 2022

Mitigation by In-Process Isolation

- Rust program is composed of two distinct pieces of code:
  - Safe blocks and unchecked code with potential exploits
- Safe Rust is protected by Rust's memory policies
- Vulnerabilities in unchecked code can undermine protection in Safe Rust
- It is natural to solve this issue by in-process isolation

Existing In-process Isolation Mechanisms

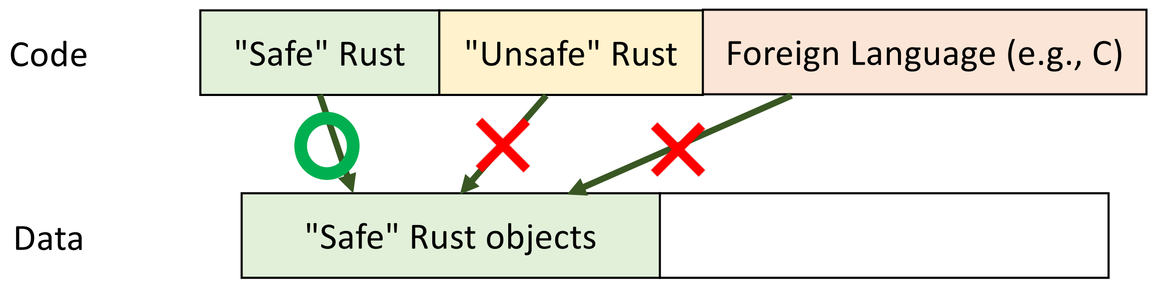
Table 1: Comparison of In-Process Isolation Policies

	Full	Protection from			
		Unsafe Rust		External Libs	
	Automation	Stack	Heap	Stack	Heap
XRust [31]	✗	✗	✓	✗	✗
Sandcrust [28]	✗	✗	✗	✓	✓
Fidelius Charm [15]	✗	✗	✗	✓	✓
TRUST	✓	✓	✓	✓	✓

- Concurrent work: PKRU-safe
  - Uses dynamic profiling to isolate heap objects from user annotated external libraries
- So far, no fully automated technique exists
- So far, no technique protects safe Rust from both unsafe Rust and external libs

Goal of TRust

- Goal of TRust: A mechanism that protects safe Rust objects from both unsafe code and external libraries in a fully automated way
  - ➢ Safe Rust objects: memory objects that are not “touched” by untrusted codes

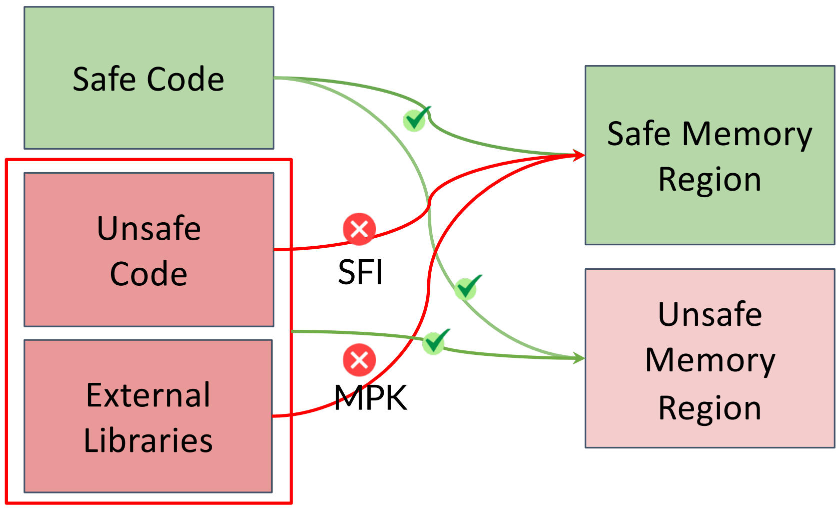


Challenges to achieve the goal

- Unlike the unsafe blocks, external libraries cannot be isolated with SFI
  - external libraries can be delivered in the form of executable binaries
  - IPC or kernel intervention are expensive to use
  - Intel MPK is perfect fit for this purpose
  - Automatically instrument entry and exit gate before and after calling external functions
- Automatic identification of unsafe objects and their allocation sites is difficult
  - Rust’s encapsulation on heap allocation hinders identification of allocation sites
  - Causes context-insensitive analysis to conclude all pointers share a few allocation sites

Overview of TRust

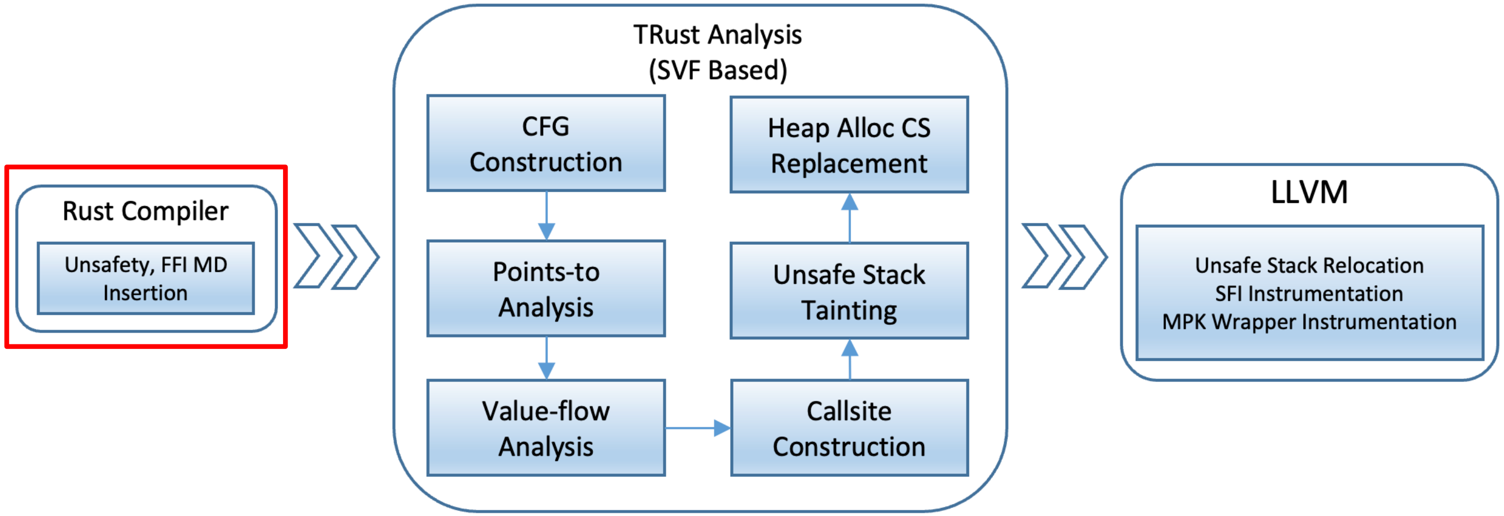
- Uses context-sensitive static analysis[1)] to identify unsafe objects "touched" by untrusted code in fully automated way
- Classifies an allocation site unsafe if it finds a flow from the site to a memory access instruction in untrusted code
- Applies SFI to isolate from unsafe code, and MPK to isolate from external libraries



1) https://github.com/SVF-tools/SVF

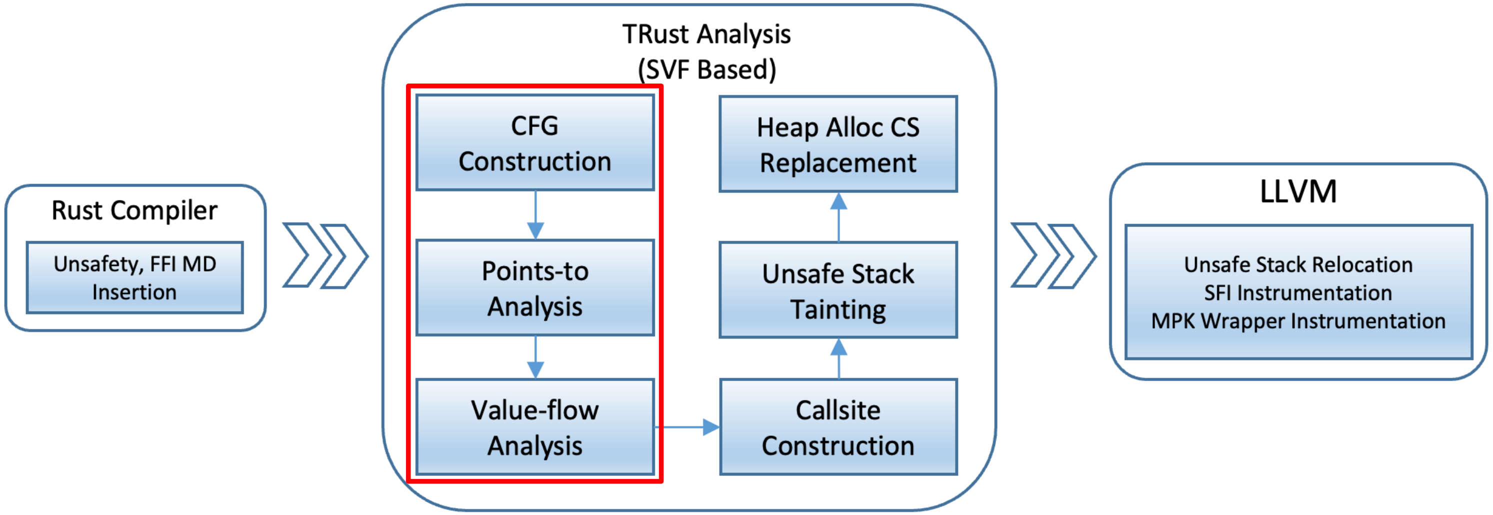
Distinguish unsafe instructions and external calls

- TRust modifies Rust compiler
- Collect unsafety information
  - Mark all instructions used in unsafe blocks as unsafe code
- Collect external library invocations
  - Mark all calls to foreign functions as external library call



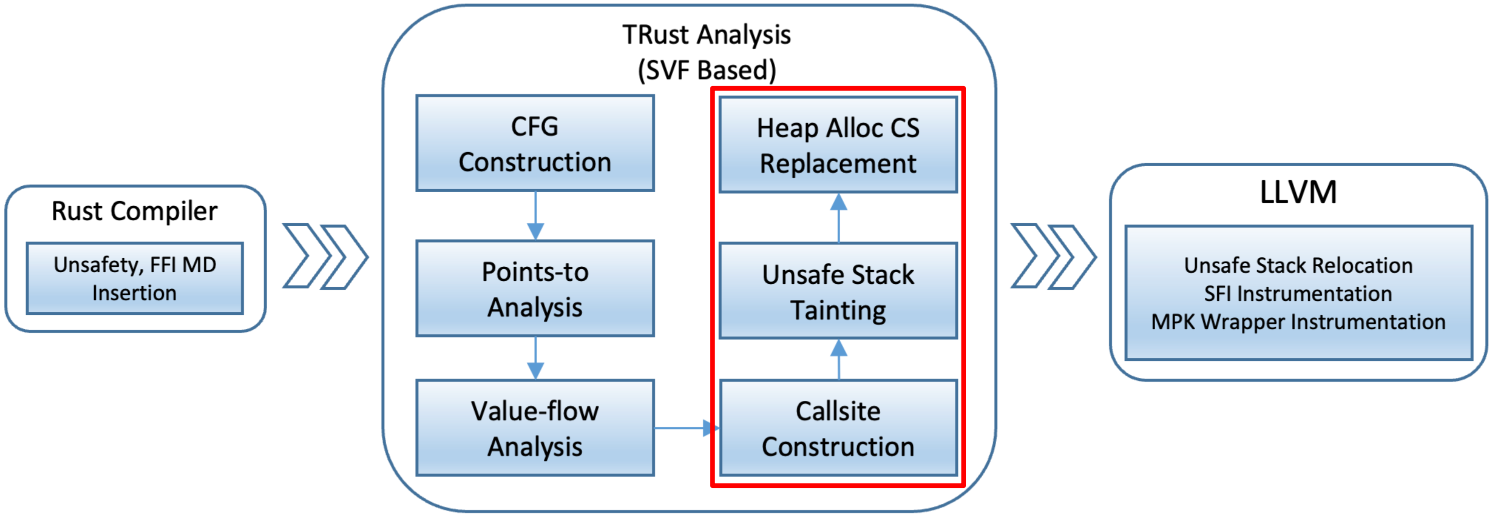
Identify unsafe pointers and allocation sites

- Collects pointer information
  - Find all pointers used in unsafe instructions
  - Such pointers become unsafe pointers
- Performs points-to analysis and value-flow analysis
  - Identify the allocation sites of unsafe pointers and track its uses.



Function cloning to improve precision

- Based on Value-flow Graph, finds all meaningful callsites contribute to allocation
  - ➢ Clone the callee functions
- call sites are replaced with call to cloned functions
- Function cloning contributes to improved analysis precision
  - ➢ distinguishing allocation sites and enabling context-sensitive identification of unsafe flows



How programs are transformed?

- Automatically identify unsafe objects that are touched by unsafe Rust
- Automatically identify allocation sites of unsafe objects
- Quarantine untrusted code using MPK and SFI

```
pub fn main(){
    let buf = Vec::new();
    let password = String::new();

    unsafe{
    //external library call
    C_written_func();
    //offset is out of bound
    let ptr = buf.as_ptr().offset(NUM);
    //out-of-bound read
    }
}
```

Original Rust Program

```
pub fn main(){
    let buf = Vec::new_unsafe();
    let password = String::new();

    unsafe{
    entry_gate();
    C_written_func();
    exit_gate();
    let ptr = buf.as_ptr().offset(NUM);
    if(!in_unsafe_region(ptr))
        raise_error;
    }
}
```

TRUST Protected Program

Evaluation: performance

- TRust shows 9.6% overhead with jemalloc, 7.6% with mimalloc, while XRust induces 26.4%
  - ➢ Although only TRust protects safe objects on the stack and quarantines external libraries
  - ➢ Thanks to address masking and selective SFI using unsafety and foreign function call metadata

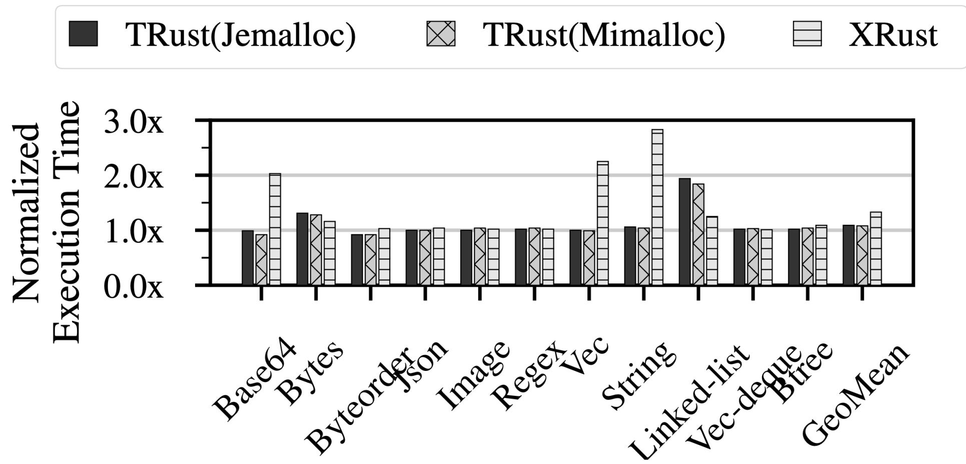


Figure 9: Normalized execution time of TRUST and XRust tested with the 11 widely used crates.

Evaluation: memory

- TRust with Jemalloc shows 35% overhead, 13% with Mimalloc as unsafe allocator
  - ➢ Due to initialization of size-segregated bins
- Xrust induces 7% overhead
  - ➢ PTmalloc based

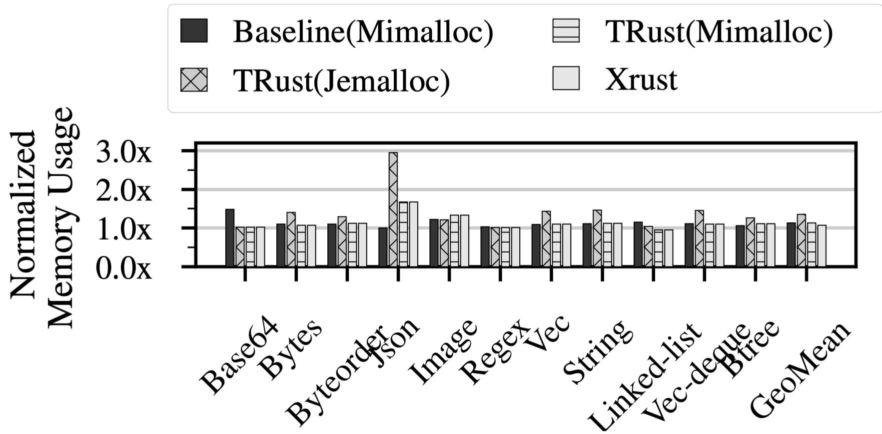


Figure 11: Normalized memory usage of TRUST and XRust tested with the 11 widely used crates.

Conclusion

- Rust is gaining reputation for its memory safety while maintaining efficiency
- Unsafe code and external libraries in Rust may undermine whole program's security
- TRust is the first attempt to automatically protect safe Rust from unsafe blocks and external libraries
- With an elaborated instrumentation using both SFI and MPK, induces lower performance overhead than previous techniques

Thank you!