Ensuring Memory Safety for the Transition from C/C++ to Rust

Oliver Braunsdorf, Maurice Hieronymus, Konrad Hohentanner, Benjamin Orthen

November 2, 2023

Advantages of Rust

- Fun programming
  - Explicit error handling
  - Helpful compiler errors enable "Compiler-Driven Development"
  - Modern syntax but low-level control over hardware
  - cargo as one tool for building, dependency management, testing
  - ...
- Security by Design ⇒ reduced cost for bug fixing after deployment
  - Traits & Zero-Sized Types enable secure API design
  - Type system enables memory-safe and data-race-free programming

Focus: Memory Safety

- **Spatial Memory Safety** "is a property that ensures that all memory dereferences are within bounds of their pointer's valid objects. An object's bounds are defined when the object is allocated."

  In Rust: strong type system enables...
  - ...compiler to check if all accesses to statically-sized objects are in-bounds
  - ...runtime to check if all accesses to dynamically-sized objects are in-bounds

- **Temporal Memory Safety** "is a property that ensures that all memory dereferences are valid at the time of the dereference, i.e., the pointed-to object is the same as when the pointer was created. When an object is freed, the underlying memory is no longer associated to the object and the pointer is no longer valid."

  In Rust: Ownership + Borrowing ensures that only 1 Owner exists, and no access is possible after automatic Drop

[0] `https://nebelwelt.net/teaching/17-527-SoftSec/slides/02-memory_safety.pdf`

The Issue with Adopting Rust

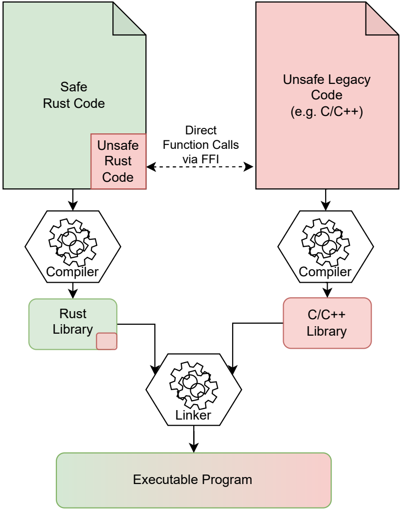
- Organizational problems when adopting a new programming language: existing (complex) software cannot be easily re-written in new language
  - Complex & maybe tedious endeavor
  - Employees have to be trained
  - New developers have to be hired
  - Development and Test-processes have to adapted

Gradually Transitioning to Rust

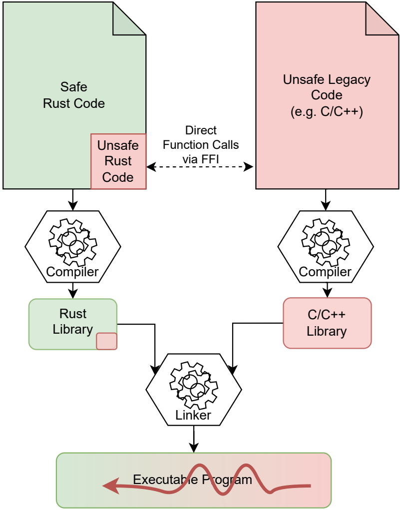
- Step-by-step migration of existing code
  - Assess code base: identify small self-contained modules for Rust replacement
  - Re-write
  - Test & Validate
  - Repeat

- Leads to mixed-language binaries: integrate Rust and `C/C++`Code within same address space
- Powered by Rust's Foreign Function Interface (FFI)

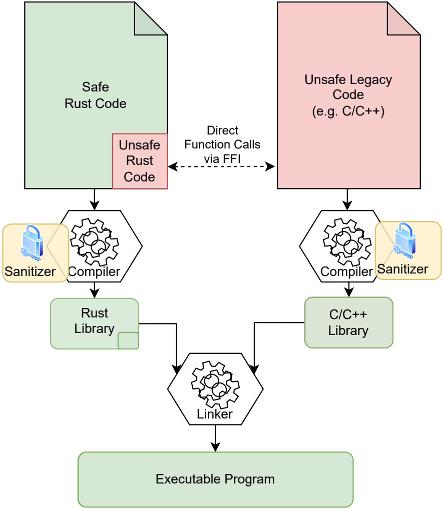
Rust and C/C++ Interoperability through FFI



The Problem with FFI

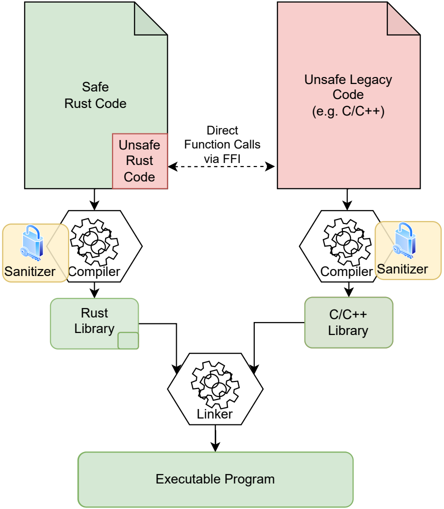


Solution: Memory-Safety Sanitizer



Solution: Memory-Safety Sanitizer

- Goal: complete memory safety for whole mixed-language binary
- Compiler-based Memory Safety
  - + Easily activated by compiler flag
  - + No source code changes
  - + Detects out-of-bounds access
  - + Detects out-of-lifetime access
  - - Runtime and memory overhead
- LLVM maintained implementations:
  - Address Sanitizer(ASAN)
  - Hardware-assissted Address Sanitizer (HWASAN)



Solution: Memory-Safety Sanitizer

How Sanitizers Work

In General

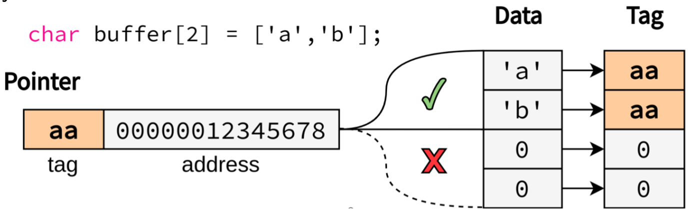
- On pointer allocation: store metadata about memory object
- On pointer propagation (assignments, pointer arithmetic): propagate metadata accordingly
- On pointer dereference: check metadata to validate that...
  - pointer accesses memory-object in bounds of allocation, and
  - accessed memory object is not yet deallocated
- On pointer deallocate: invalidate metadata

Solution: Memory-Safety Sanitizer

How Sanitizers Work

HWASAN

- Tagging-based sanitizer using ARM Top-Byte-Ignore feature
- On pointer allocation: store 8-bit tag in upper pointer bits and shadow memory
- On pointer propagation: nothing, implicit
- On pointer dereference:
  - Load tag from shadow memory
  - Compare with tag in pointer
  - Allow access if tags match



Hohentanner et al. - HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading

Solution: Memory-Safety Sanitizer

Sanitizer in Rust

```
RUSTFLAGS="-Zsanitizer=hwaddress" cargo +nightly run -Zbuild-std --target
aarch64-unknown-linux-gnu
```

- `-Zsanitizer=[address|hwaddress|...]`: select sanitizer
- `-Zbuild-std`: re-build the std lib with sanitizer
- Only available on nightly toolchain at the moment

More info:
`https://doc.rust-lang.org/beta/unstable-book/compiler-flags/sanitizer.html`

DEMO

Current Research: Sanitizer Optimizations for Safe Rust

Idea

- Additional memory-safety checks for safe Rust code unnecessary
- Core Goal:
  - Only instrument C/C++ and `unsafe` Rust Code
  - Omit instrumentation for all Rust objects that can be proven to be safe by the Rust compiler & runtime

Example: Only object `b` is affected by unsafe code. Object `a` does not need instrumentation.

```
extern fn foreign_function;
fn main() {
  let mut a = [0,1,2];
  let mut b = [3,4,5];
  unsafe {
    let x = b.as_mut_ptr();
    foreign_function(x);
  }
}
```

Current Research: Sanitizer Optimizations for Safe Rust

Approach

- Primary way of exchanging non-primitive data structures $\Rightarrow$ Raw Pointers: `*mut T` / `*const T`
- Rust stops guaranteeing memory-safety when raw pointer is accessed
- $\Rightarrow$ Use sanitizer to jump-in for raw pointers

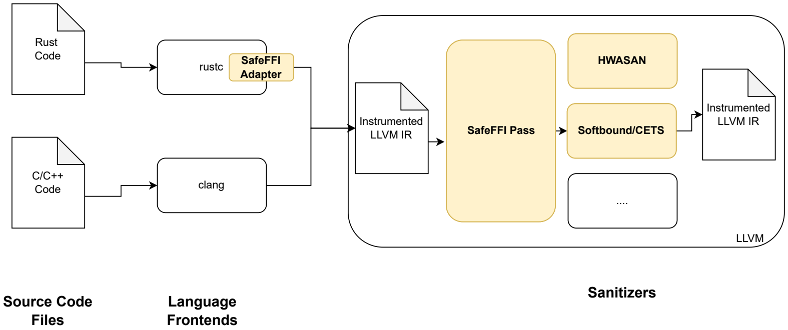
- During compilation: at conversion between Safe Rust Pointers and Raw Pointers, keep track of spatial and temporal memory-safety properties
  - Safe $\rightarrow$ Raw: emit know size, liveliness guaranteed
  - Raw $\rightarrow$ Safe: emit expected size & liveliness asserted

Description	Symbol
Rust references (mutable or immutable)	&T
Arrays	[T;n]
Slices	&[T]
String references	&str
Boxes	Box<T>
Trait Objects	&dyn T
Function Pointers	

Safe Rust Pointers (statically or dynamically sized)

Current Research: Sanitizer Optimizations for Safe Rust

Architecture



Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Security

Crate	CVE	HWASAN	HWASAN+SafeFFI
slice-deque	CVE-2018-20995	✓	✓
compact_arena	CVE-2019-16139	✓	✓
arr	CVE-2020-35887	✓	✓
simple-slab	CVE-2020-35892	✓	✓ (ensure ✓)
simple-slab	CVE-2020-35893	✓	✓

Figure: Spatial Vulnerabilities

Crate	CVE	HWASAN	HWASAN+SafeFFI
smallvec	CVE-2019-15551	✓ (Trap)	✓ (Trap)
bitvec	CVE-2020-35862	✓	✓
nano_arena (implements allocator?)	CVE-2021-28032	x	x
heapless	CVE-2020-36464	✓	✓

Figure: Temporal Vulnerabilities

Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Performance

- Measured with curl-rust (with statically linked openssl, libcurl, libz)
- **56.20% saving** of HWASAN's check instructions
  - $324,865$ total memory accesses
  - $182,585$ elided checks
  - 352 additional check functions added

[1] `https://github.com/rustls/rustls`

Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Performance

- Measured with curl-rust (with statically linked openssl, libcurl, libz)
- **56.20% saving** of HWASAN's check instructions
  - $324,865$ total memory accesses
  - $182,585$ elided checks
  - 352 additional check functions added

- **70.59% saving** of HWASAN's check instructions when replacing OpenSSL C code by Rust TLS implementation[1]
  - $274,299$ total memory accesses
  - $193,637$ elided checks
  - 219 additional check functions added

[1] `https://github.com/rustls/rustls`

Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Incentive for Transition

The more legacy code you replace by Safe Rust code,
the more performance you regain,
while always maintaining memory safety.

Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Performance

- Direct performance gain dependent on application profile
- Example: Rust Implementation of Leighton-Micali Signatures[2]
  - No I/O, high CPU utilization
  - many memory object manipulation operations
  - ⇒ worst case for memory-safety sanitizers
- SafeFFI is **1.78x faster** than unoptimized HWASAN

Vanilla	HWASAN	HWASAN+SafeFFI
2.858s	8.260s	4.648s

Table: Run time of LMS example

[2] `https://github.com/Fraunhofer-AISEC/hbs-lms-rust`

Current Research: Sanitizer Optimizations for Safe Rust

Preliminary Results

Limitations

- `std::mem::transmute()`
- Cast from raw pointer to safe pointer for arbitrary dynamic datastructures
  - Possible heuristic for `&[T]`, `&str`, `Vec<T>`: `from_raw_parts(buf,len)`

Current Research: Sanitizer Optimizations for Safe Rust

Outlook

Short-Term

- Optimize SafeFFI LLVM Pass to recognize more safe pointers
- More tests with real-world mixed-language programs e.g., Chromium, Firefox browsers
  - Security tests with known vulnerabilities of real-world programs
  - Performance tests (Rust benchmarks?)

Long-Term

- Implement optimizations for more sanitizers or memory-safety hardware, e.g.,
  - CHERI
  - ARM MTE (Memory Tagging Extensions)
- Extend concept for other LLVM-based programming languages: Swift, GO-LLVM

THANK YOU!

- Contact: Oliver Braundsorf
  - `https://github.com/obraunsdorf`
  - `https://obraunsdorf.dev/`
  - `https://www.linkedin.com/in/obraunsdorf/`

Backup

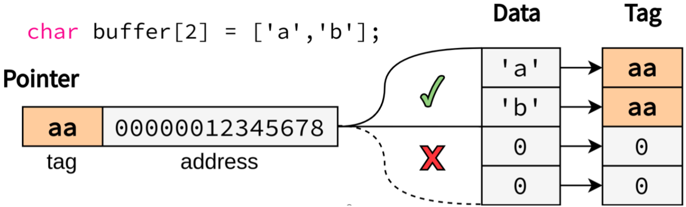
Backup

Example: Calling OpenSSL C functions from Rust

```
extern fn RSA_public_encrypt;
pub fn public_encrypt(
  key: &RsaKey,
  from: &[u8],
  to: &mut [u8]
) -> usize {
  unsafe {
    RSA_public_encrypt(
      from.len() as c_int,
      from.as_ptr(),
      to.as_mut_ptr(),
      key.as_ptr());
  }
}
```

Backup
HWASAN

- Tagging based memory sanitizer using ARM Top-Byte-Ignore Feature
- Per-object 8-bit tag stored in upper pointer bits and shadow memory
- On memory access:
  - Load tag from shadow memory
  - Compare with tag in pointer
  - Allow access if tags match

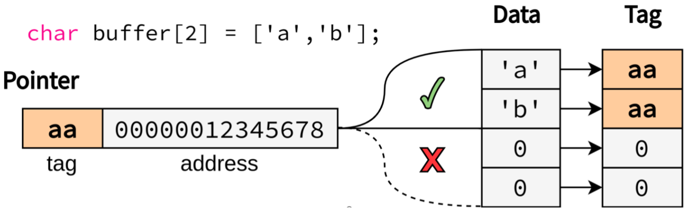


[2] HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading

Backup

HWASAN

- Pros:
  - Low memory overhead (ca. 35%) with the default 16-to-1 granularity
  - Run time overhead of ca. 2x
- Cons:
  - Only 8 bits of entropy → Tag reuse between objects
  - Only 1 tag per object → no intra-object detection for structs/classes



[2] HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading

Backup

Utilize Existing Approach: Softbound & CETS

- Compiler-based memory safety for C, published 2009
- Softbound: spatial memory safety

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

```
if ((ptr < base) || (ptr+size > bound)) {
 abort();
} else {
value = *ptr;
}
```

Backup

Utilize Existing Approach: Softbound & CETS

- Compiler-based memory safety for C, published 2009
- CETS: temporal memory safety

```
ptr = malloc(size);
ptr_key = next_key++;
ptr_lock_addr = allocate_lock();
*(ptr_lock_addr) = ptr_key;
```

```
if (ptr_key != *ptr_lock_addr) { abort(); }
value = *ptr;
```

Backup

Utilize Existing Approach: Softbound & CETS

- Performance with Softbound/CETS port to LLVM9
- Benchmark program: sha256sum (C coreutils)
- Tested different sizes of input data to be hashed
- Results:
  - Runtime without inlining of Softbound/CETS functions: 5x - 30x; geo. mean: 14.58x
  - Runtime with inlining using LTO: 5x - 25x; geo. mean: 12.62x
  - Memory without inlining of Softbound/CETS functions: geo. mean: 09.53x
  - Runtime with inlining using LTO: geo mean: 10.92x

‡ real results might be worse, because unsafe pointers are underapproximated at the moment

Backup

Utilize Existing Approach: Softbound & CETS

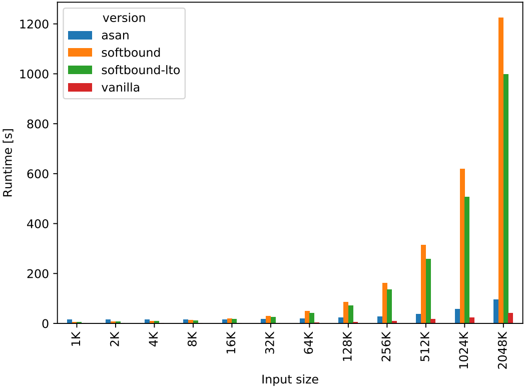


Figure: Runtime Overhead

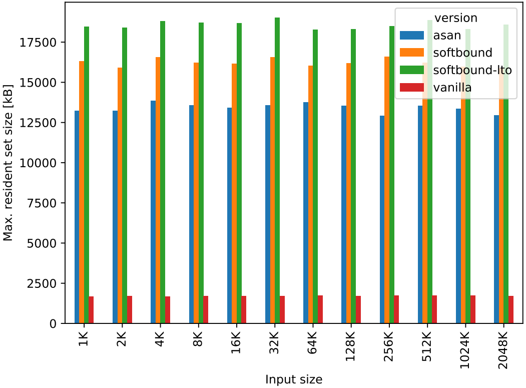


Figure: Memory Overhead

Backup

Utilize Existing Approach: Softbound & CETS

Example: Calling OpenSSL C functions from Rust

```
extern fn RSA_public_encrypt;
pub fn public_encrypt(
  key: &RsaKey,
  from: &[u8],
  to: &mut [u8]
) -> usize {
  unsafe {
    RSA_public_encrypt(
      from.len() as c_int,
      from.as_ptr(),
      to.as_mut_ptr(),
      key.as_ptr());
  }
}
```

Backup

Status 29.07.2021

- Softbound/CETS ported from LLVM 3.4 to LLVM 9 (incl. Compiler-RT)
- Softbound/CETS as Sanitizer in Clang
  - ✓ Compiling and running simple tests, detecting memory safety violations
  - ✓ Compiling Nginx, git, tmux, ...
  - ✗ Running these applications aborts with false positive due to missing support for variadic arguments in Softbound/CETS
- Softbound/CETS as Sanitizer in Rust compiler
  - ✓ Compiling and running simple tests, detecting memory safety violations
  - ✓ Compiling libraries `aho-corasick`, `rand`
  - ✗ Running library tests aborts with false positive due to ?
  - Instrumentation of all Rust code, no optimizations implemented

Backup

Status 27.01.2022

- Softbound/CETS ported from LLVM 3.4 to LLVM 9 (incl. Compiler-RT)
- Softbound/CETS as Sanitizer in Clang
  - ✓ Compiling and running simple tests, detecting memory safety violations
  - ✓ Compiling Nginx, git, tmux, ...
  - ✗ Running these applications aborts with false positive due to missing support for variadic arguments in Softbound/CETS
- Softbound/CETS as Sanitizer in Rust compiler
  - ✓ Compiling and running simple tests, detecting memory safety violations
  - ✓ Compiling libraries `aho-corasick`, `rand`
  - ✓ Compiling and running real world no-std crates (LMS)
  - ✓ Compiling the Rust standard library
  - ✓ Compiling and running real world pure rust crates with std-lib (ripgrep, coreutils)
  - ✓ Optimizations (discussed in the following)
    - elide dereference checks for safe pointers
    - elide metadata propagation for safe pointers

Backup

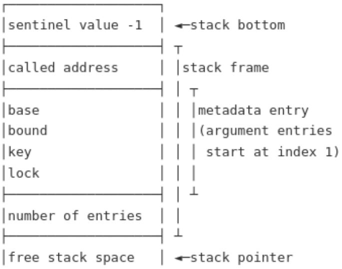
Problem: Nested Pointers in Aggregate Types

- Aggregate Types ≈ Arrays, Structs
- Used heavily in Rust for two-valued types, e.g.
  - `Option<T>` / `Result<T>` ≈ (discriminant $\in \{0, 1\}$, value $\in T$)
  - Fat pointers aka. *unsized* or *dynamically sized* types
- **Solution:** Recursively traverse the struct
  - for stack objects: push metadata for each contained pointer type onto shadow stack.
    Status: ✓
  - for heap objects: store metadata for each contained pointer type based in runtime metadata Trie?
    Status: ✗

Backup

Problem: Missing Metadata for Function Parameters

- Happens if: instrumented Rust functions are called from uninstrumented C functions:
  `C --arg--> Rust`
- Special case: instrumented rust functions are passed as function pointers and later called as callback from uninstrumented C functions
- Consequential problem: instrumented Rust functions assume that shadow stack is setup correctly, but no metadata has been pushed. Popping from the shadow ⇒ smashing the shadow stack
- **Solution:** ✓
  - Extend shadow stack ABI: additionally push address of called function onto shadow stack
  - Check if correct address is present before popping from the stack



Backup

Problem: Missing Metadata for Function Return Values

- Happens if: instrumented Rust functions read return value from called uninstrumented C functions:
  `Rust <--ret-- C`
- Can be avoided if *all* uninstrumented C functions are wrapped in Softbound/CETS-aware wrapper functions
- However, if not all uninstrumented C are wrapped: Popping metadata from shadow stack when no metadata is there ⇒ smashing the shadow stack

```
┌───────────────────┐
|sentinel value -1  |  <-stack bottom
├───────────────────┤  ┬
|called address     |  |stack frame
├───────────────────┤  | ┬
|base               |  | |metadata entry
|bound              |  | |(argument entries
|key                |  | | start at index 1)
|lock               |  | |
├───────────────────┤  | ┴
|number of entries  |  |
├───────────────────┤  ┴
|free stack space   |  <-stack pointer
```

- **Solution:** ✓
  - Encode if callee function pushed metadata for return value by setting LSB of `called address`
  - Can be done because otherwise LSB is always 0 (8-byte alignment of function addresses on x86)