Safe Interactions with Foreign Languages through

Encapsulated Functions

Leon Schuermann, Jack Toubes, Tyler Potyondy, Mae Milano, Amit Levy

May 7th, 2024

RustNL 2024

Motivation

Memory and **type safe** languages are *safer*, *more reliable*, and *more secure*

- Increase developer efficiency
- Encourage *fearless* development and optimizations
- Prevent entire classes of bugs
  (use-after-free, type confusion, out-of-bounds accesses, ...)

SIGN IN / UP

The Register®

CSO

115

Microsoft is busy rewriting core Windows code in memory-safe Rust

Now that's a C change we can back

Thomas Claburn

Thu 27 Apr 2023 | 20:45 UTC

Microsoft is rewriting core Windows libraries in the Rust programming language, and the more memory-safe code is already reaching developers.

David "dwizzle" Weston, director of OS security for Windows, announced the arrival of Rust in the operating system's kernel at BlueHat IL 2023 in Tel Aviv, Israel, last month.

"You will actually have Windows booting with Rust in the kernel in probably the next several weeks or months, which is really cool," he said. "The basic goal here was to convert some of these internal C++ data types into their Rust equivalents."

The Linux Kernel

6.9.0-rc5

Contents

Rust

English

Documentation related to Rust within the kernel. To start using Rust in the kernel, please read the Quick Start guide.

The Rust experiment

The Rust support was merged in v6.1 into mainline in order to help in determining whether Rust as a language was suitable for the kernel, i.e. worth the tradeoffs.

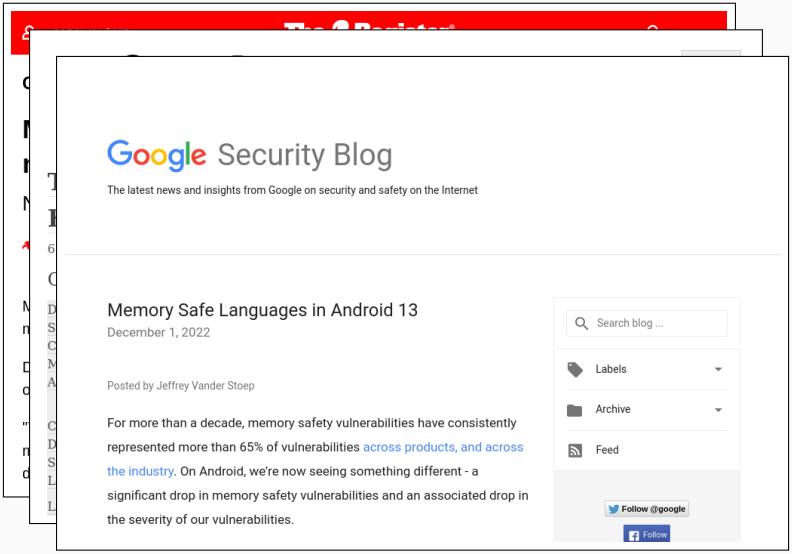
Currently, the Rust support is primarily intended for kernel developers and maintainers interested in the Rust support, so that they can start working on abstractions and drivers, as well as helping the development of infrastructure and tools.

If you are an end user, please note that there are currently no in-tree drivers/modules suitable or intended for production use, and that the Rust support is still in development/experimental, especially for certain kernel configurations.

This documentation does not include rustdoc generated information.

- Quick Start
- General Information
- Coding Guidelines
- Arch Support
- Testing

Safe Languages in Constrained Environments

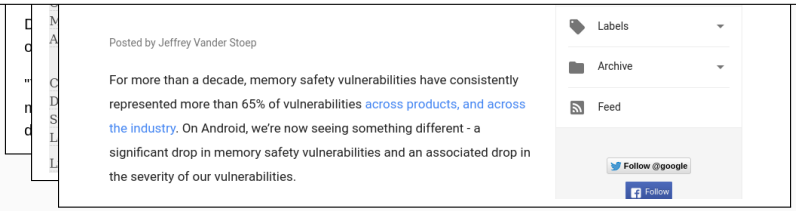


Safe Languages in Constrained Environments

Posted by Jeffrey Vander Stoep

For more than a decade, memory safety vulnerabilities have consistently represented more than 65% of vulnerabilities across products, and across the industry. On Android, we're now seeing something different - a significant drop in memory safety vulnerabilities and an associated drop in the severity of our vulnerabilities.

Looking at vulnerabilities reported in the Android security bulletin, which includes critical/high severity vulnerabilities reported through our vulnerability rewards program



Safe Systems Programming Languages

Conventional wisdoms

- Dynamically enforced type & memory safety
- Garbage collection → unpredictable timing behavior & memory allocations
- High-overhead FFIs

Safe Systems Programming Languages

Conventional wisdoms

- Dynamically enforced type & memory safety
- Garbage collection → unpredictable timing behavior & memory allocations
- High-overhead FFIs

New class of safe systems programming languages

- Compile-time type + memory-safety
- Reference-counting or region-based memory management
- Direct ABI compatibility & low-level control of assembly

The Need for Safe Cross-Language Interactions

So … why does my operating system still have access violations?

This is what we're trying to solve!

The Need for Safe Cross-Language Interactions

Parts of our systems will continue to be written in unsafe languages

The Need for Safe Cross-Language Interactions

Parts of our systems will continue to be written in unsafe languages

- Operating systems are built on **legacy software**

The Need for Safe Cross-Language Interactions

Parts of our systems will continue to be written in unsafe languages

- Operating systems are built on **legacy software**
- **External constraints** prevent use of safe languages

The Need for Safe Cross-Language Interactions

Parts of our systems will continue to be written in unsafe languages

- Operating systems are built on **legacy software**
- **External constraints** prevent use of safe languages
- Software is provided as **reusable libraries**

The Need for Safe Cross-Language Interactions

Parts of our systems will continue to be written in unsafe languages

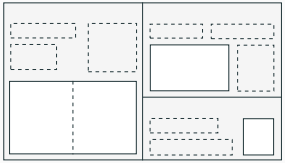
- Operating systems are built on **legacy software**
- **External constraints** prevent use of safe languages
- Software is provided as **reusable libraries**

Takeaways:

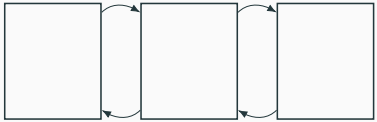
1. There will not be a single systems programming language
2. Software must interact with code written in other (unsafe) languages
3. We need a mechanism to safely interact between languages

Encapsulated Functions

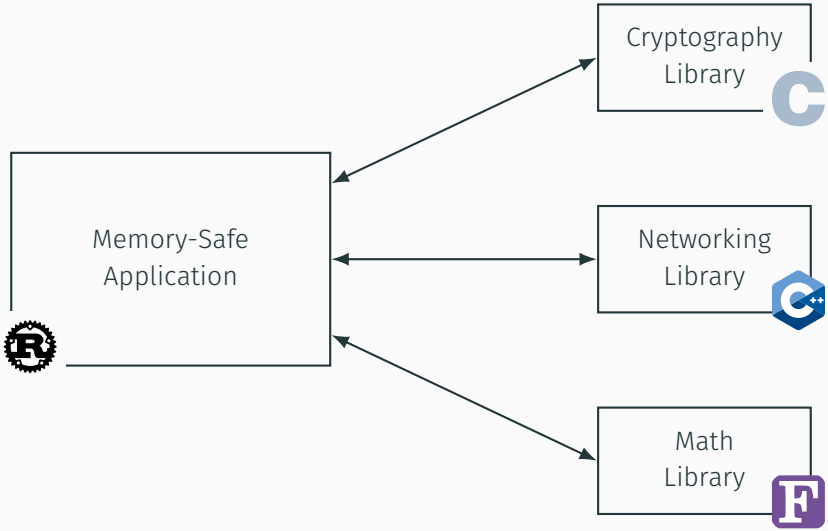
A system to facilitate safe interactions with components written in foreign languages.

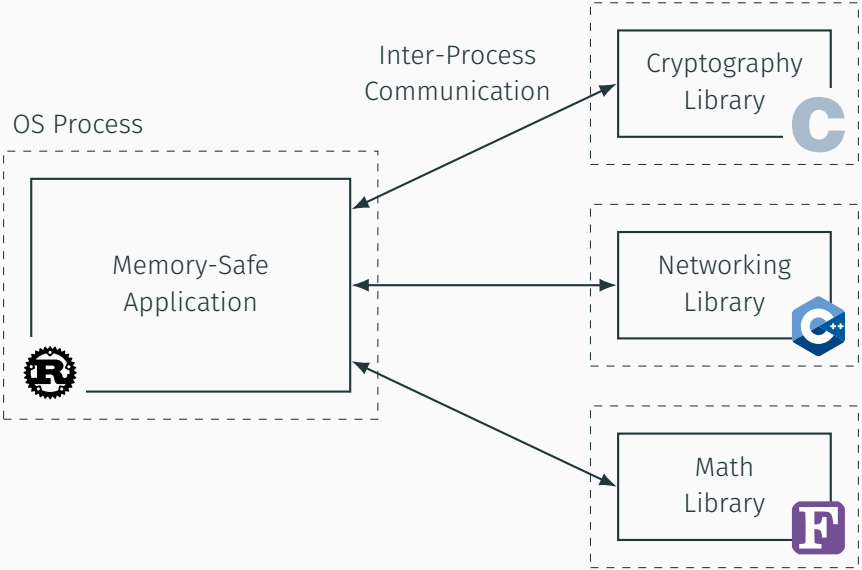


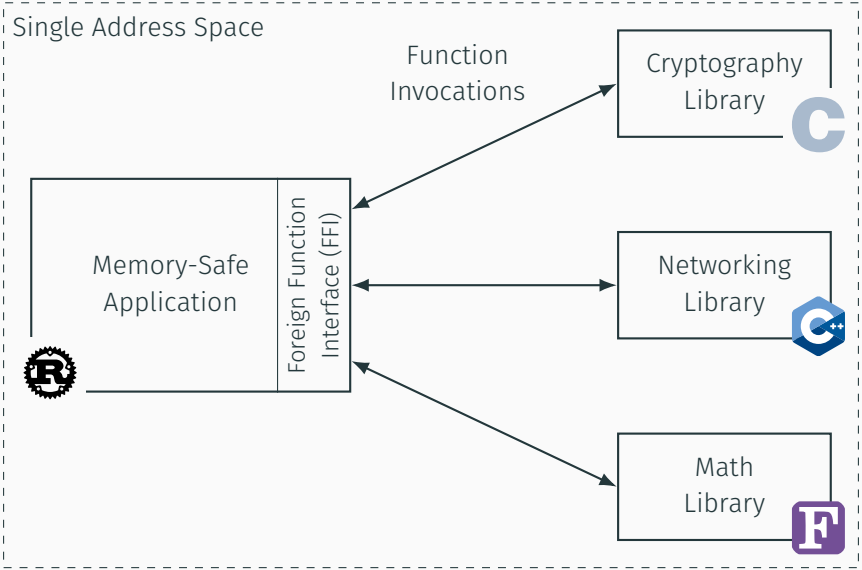
Memory Safety



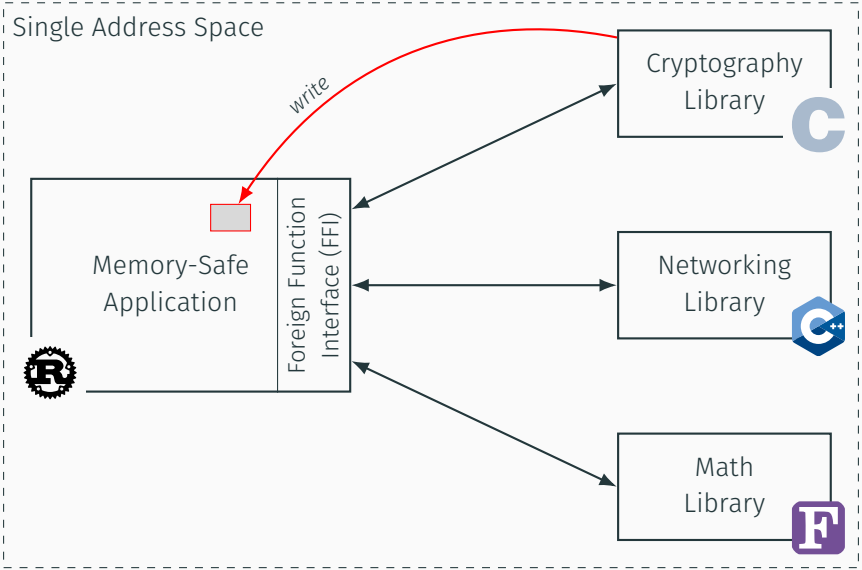
Type Safety

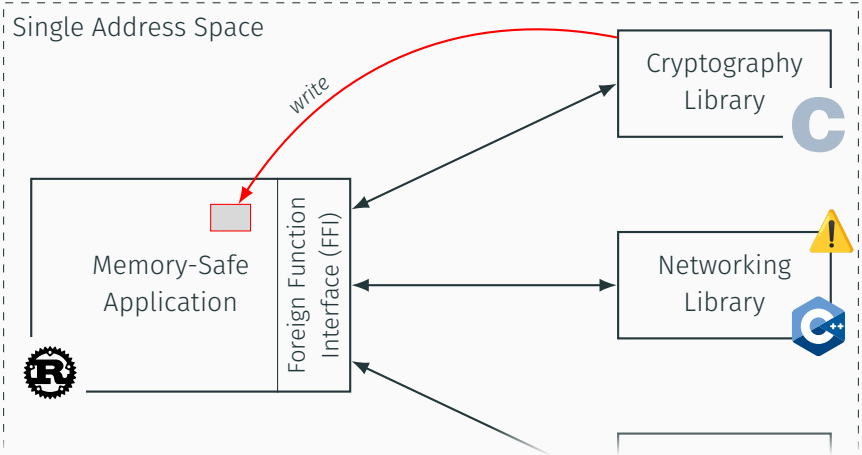

Cryptography
Library
Memory-Safe
Application
Networking
Library
Math
Library


OS Process
Memory-Safe Application
Inter-Process Communication
Cryptography Library
C
Networking Library
C++
Math Library
F

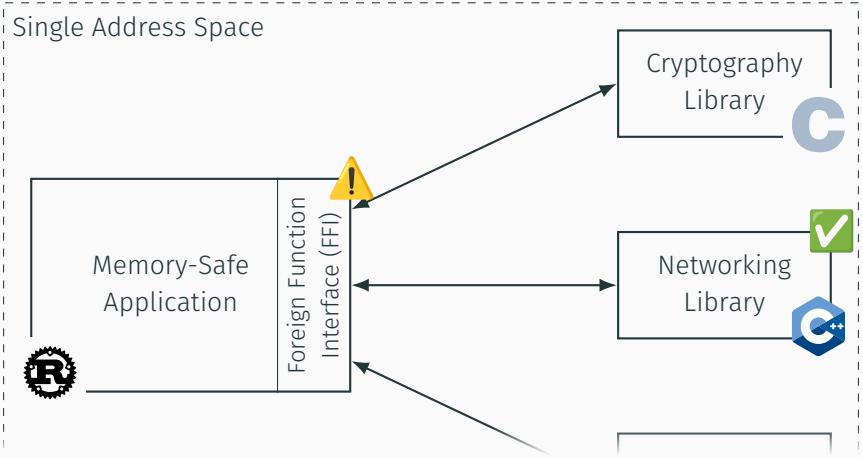

Single Address Space
Function Invocations
Cryptography Library
C
Memory-Safe Application
Foreign Function Interface (FFI)
Networking Library
C++
Math Library
F

```rust
extern "C" {
  fn aes_encrypt(
    key: *const u8, buf: *mut u8, len: usize) -> bool;
}

pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}

pub fn encrypt(mut bytes: Vec<u8>) -> Message {
  let res: bool = unsafe {
    aes_encrypt(KEY, bytes.as_mut_ptr(), bytes.len())
  };

  Message::Encrypted(res, bytes)
}
```


Single Address Space
write
Cryptography Library
Memory-Safe Application
Foreign Function Interface (FFI)
Networking Library
Math Library



→ Memory-safety bugs in libraries compromise system safety



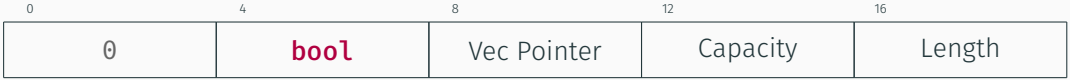
→ Memory-safety bugs in libraries compromise system safety

→ Differing cross-language semantics introduce additional hazards!

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```rust
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:



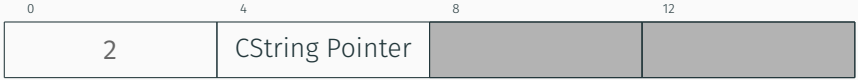
`Message::Unencrypted`:

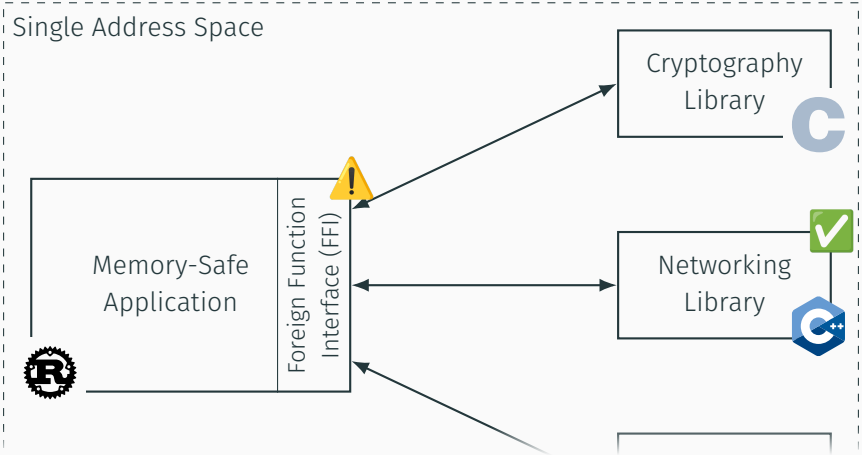
```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

```rust
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:

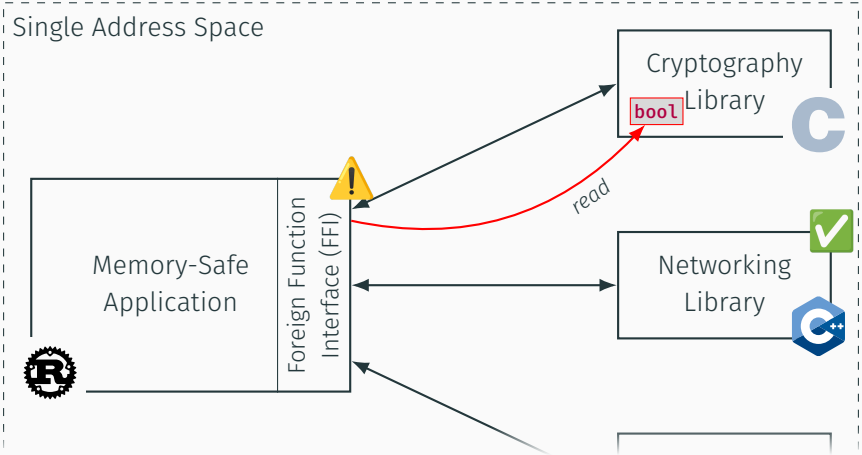
`Message::Unencrypted`:





→ Memory-safety bugs in libraries compromise system safety

→ Differing cross-language semantics introduce additional hazards!



→ Memory-safety bugs in libraries compromise system safety

→ Differing cross-language semantics introduce additional hazards!

```rust
pub enum Message {
  Encrypted(bool, Vec<u8>),
  Unencrypted(CString),
}
```

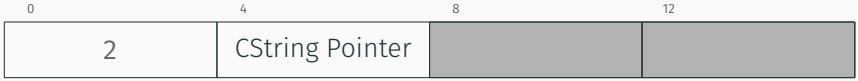
```rust
enum bool {
  false = 0,
  true = 1,
}
```

`Message::Encrypted`:

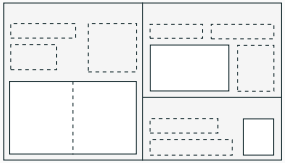
Store:

`Message::Unencrypted`:

Read:

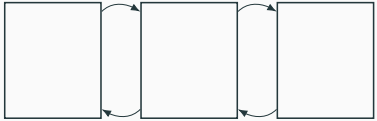


Encapsulated Functions



Safe Trampolining

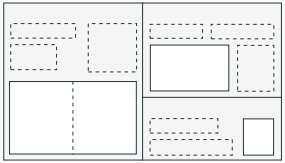
- Isolating memory between *safe* and *untrusted*, *foreign* code
- Platform-agnostic
- Retains function call semantics



Type Infrastructure

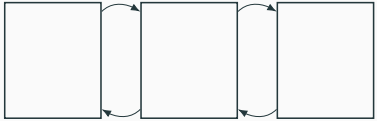
- Securing cross-language interactions
- Enforced at compile time
- Incuring minimal runtime overheads

Encapsulated Functions



Safe Trampolining

- Isolating memory between *safe* and *untrusted*, *foreign* code
- Platform-agnostic
- Retains function call semantics



Type Infrastructure

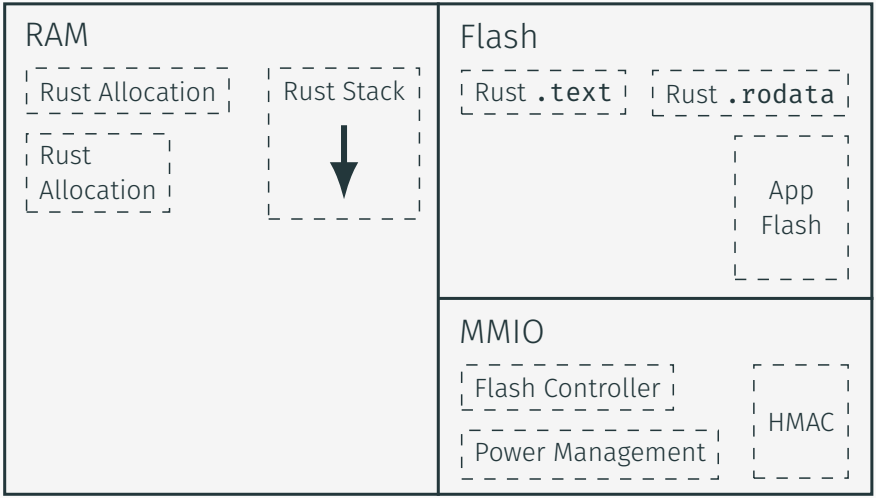
- Securing cross-language interactions
- Enforced at compile time
- Incuring minimal runtime overheads

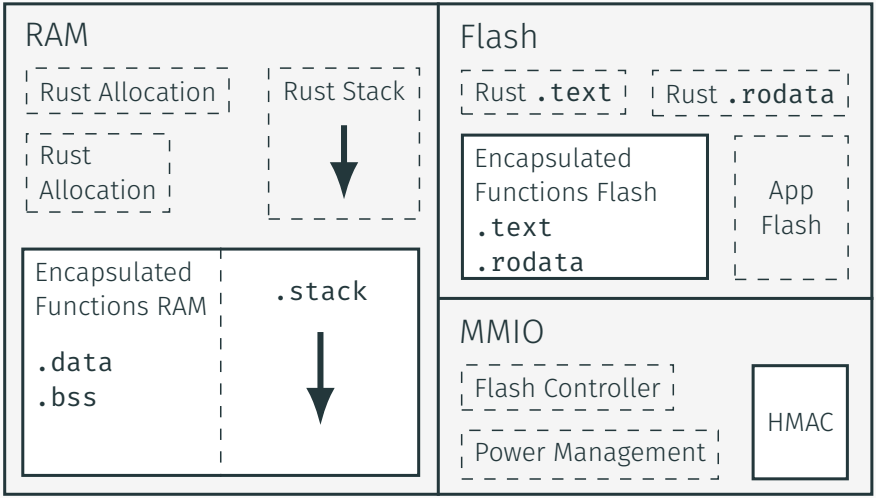
Memory Protection

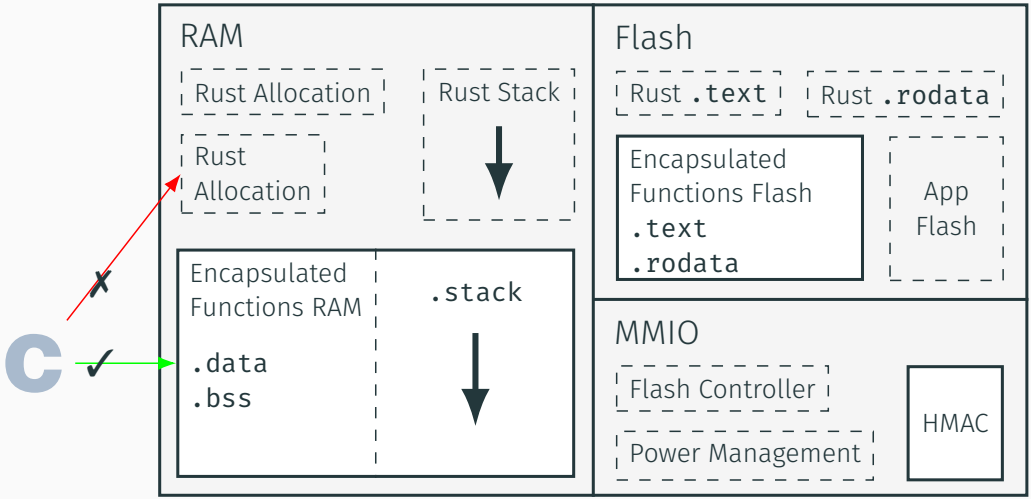
Encapsulated Functions isolates untrusted (foreign) code using a **memory protection mechanism**.

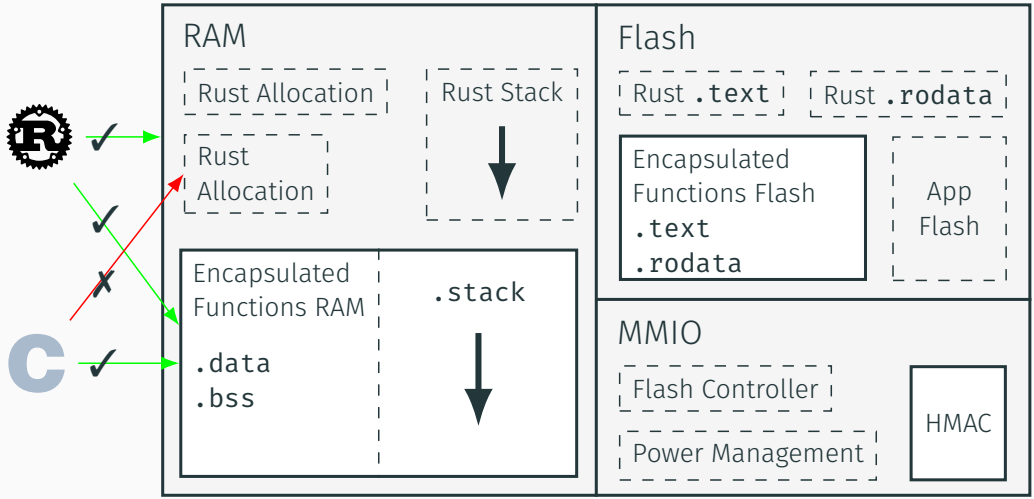
Agnostic over mechanism used:

- Hardware-Mediated Memory Protection
  - Address-Space Isolation
  - Memory Protection Unit
  - x86 Memory Protection Keys
- Software Fault Isolation


RAM
Rust Allocation
Rust Allocation
Rust Stack
Flash
Rust .text
Rust .rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC


RAM
Rust Allocation
Rust Allocation
Rust Stack
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC


RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C


RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C

Memory Safety by Isolating Untrusted Code

Key Challenges:

1. Loading Foreign Libraries into a Disjoint Set of Allocations
2. Setting Up Memory Protection
3. Switching *Protection Domains*

Key Challenges:

1. Loading Foreign Libraries into a Disjoint Set of Allocations
2. Setting Up Memory Protection
3. Switching *Protection Domains*

Switching Protection Domains

Problem: We must restrict *untrusted*, foreign code to its memory allocations!

Switching Protection Domains

Problem: We must restrict *untrusted*, foreign code to its memory allocations!

Intuition:

1. Enable protection.
2. Execute function.
3. Disable protection.

Switching Protection Domains

Problem: We must restrict *untrusted*, foreign code to its memory allocations!

Intuition:

1. Enable protection.
2. Execute function.
3. Disable protection.

Challenges:

- Cannot enable memory protection in the *host* language.
- Instead, need to switch on the function call itself.
- Maintain access to function parameters and return value.

Switching Protection Domains

Problem: We must restrict *untrusted*, foreign code to its memory allocations!

Intuition:

1. Enable protection.
2. Execute function.
3. Disable protection.

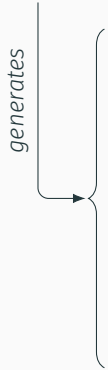
Challenges:

- Cannot enable memory protection in the *host* language.
- Instead, need to switch on the function call itself.
- Maintain access to function parameters and return value.

Our solution: A *callee-specific*, *safe* trampoline.

```
int randombytes_buf(void *, size_t);
```

```
int randombytes_buf(void *, size_t);
```




```
extern "C" fn randombytes_buf(
    *mut c_void, usize) -> c_int;
```

```
int randombytes_buf(void *, size_t);
```


```
extern "C" fn randombytes_buf(
    *mut c_void, usize) -> c_int;


        = invoke
```

```
int randombytes_buf(void *, size_t);
```

generates

```
extern "C" fn randombytes_buf(
    *mut c_void, usize) -> c_int;

        = invoke
```

invoke

1. Copy stacked arguments
2. Enable memory protection
3. Invoke foreign function
4. Disable memory protection
5. Copy return value

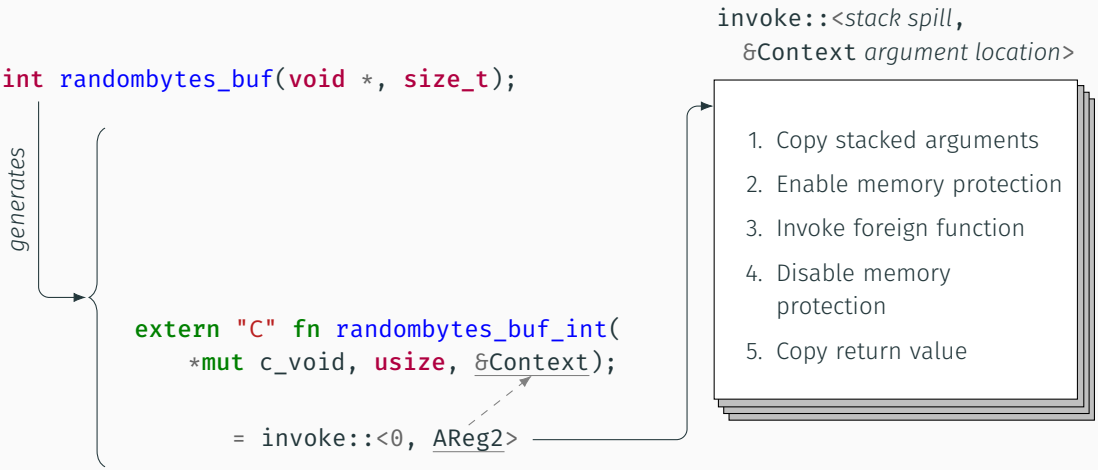
```
int randombytes_buf(void *, size_t);
```

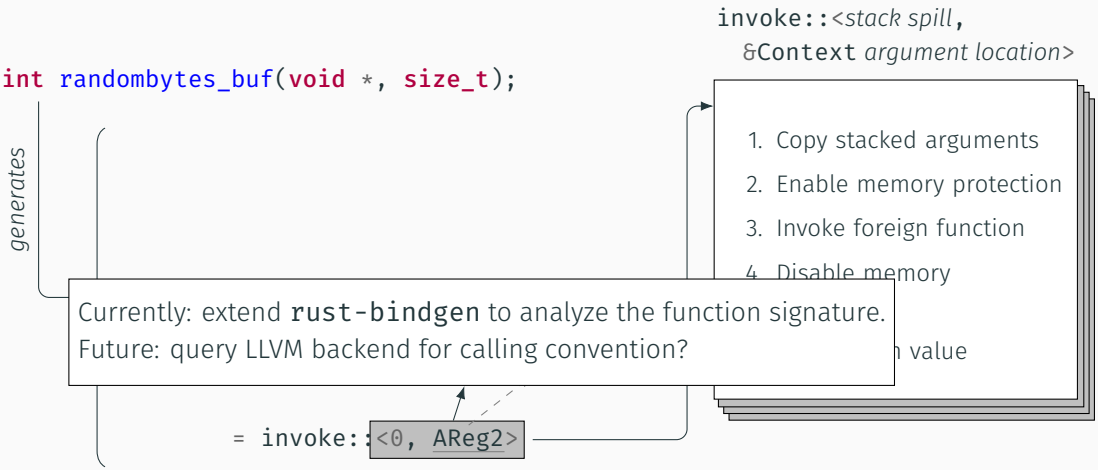
generates

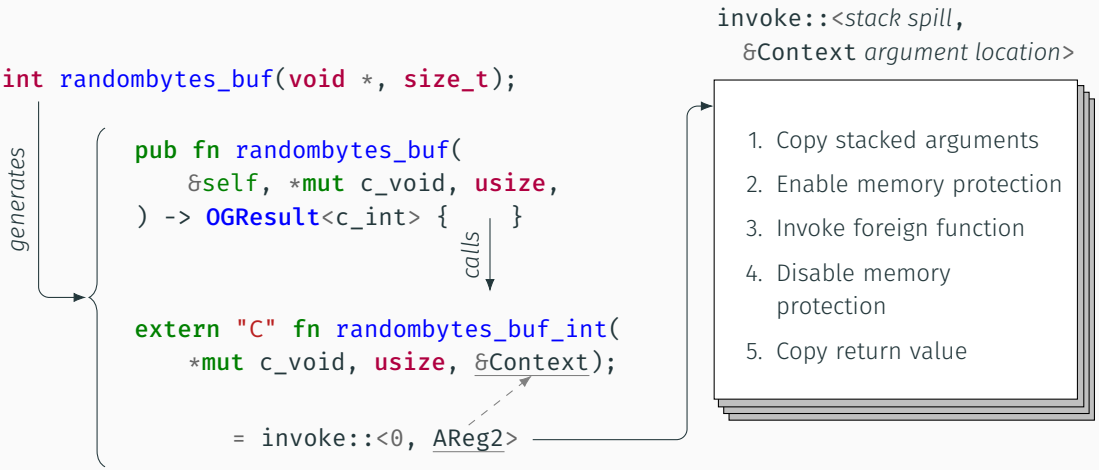
```
extern "C" fn randombytes_buf_int(
    *mut c_void, usize, &Context);

        = invoke
```

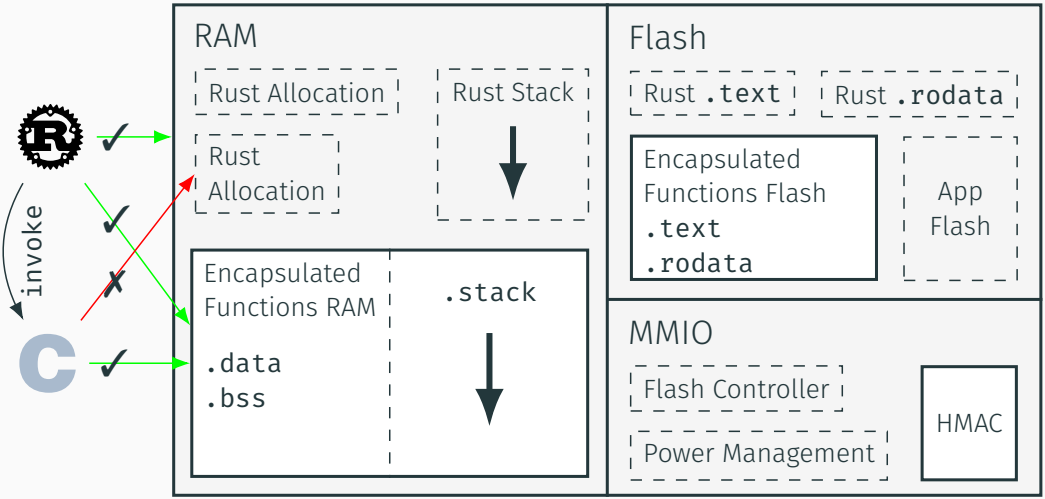
invoke

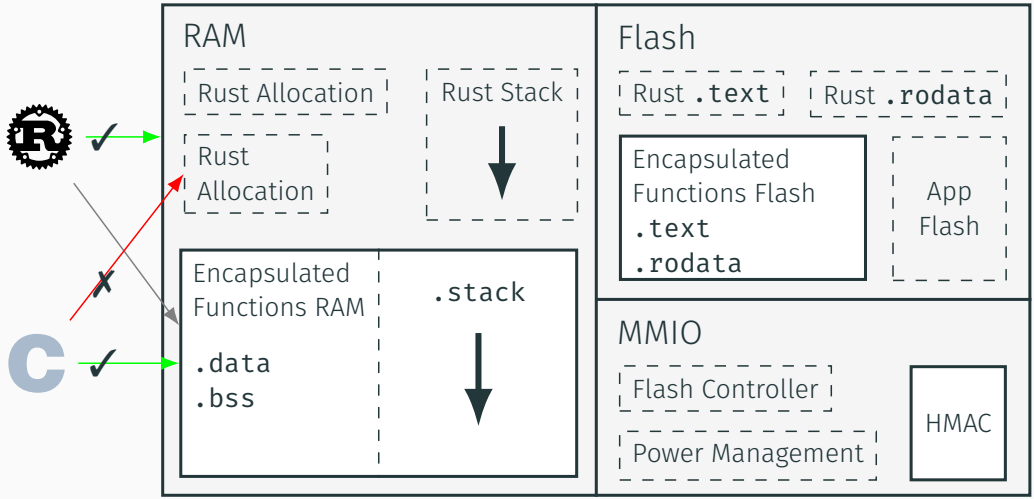
1. Copy stacked arguments
2. Enable memory protection
3. Invoke foreign function
4. Disable memory protection
5. Copy return value

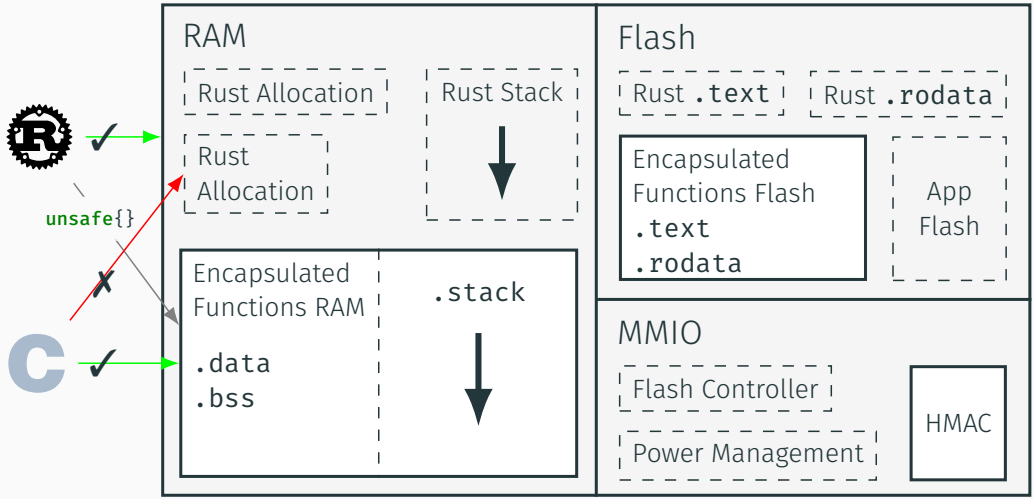

int randombytes_buf(void *, size_t);
generates
extern "C" fn randombytes_buf_int(
*mut c_void, usize, &Context);
= invoke::<0, AReg2>
invoke::<stack spill,
&Context argument location>
1. Copy stacked arguments
2. Enable memory protection
3. Invoke foreign function
4. Disable memory protection
5. Copy return value

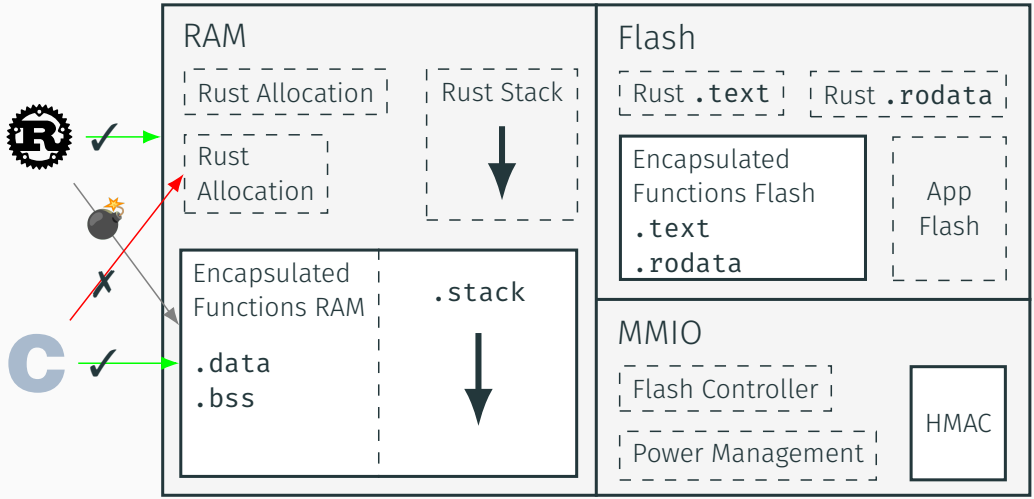

invoke::<stack spill,
&Context argument location>
int randombytes_buf(void *, size_t);
generates
1. Copy stacked arguments
2. Enable memory protection
3. Invoke foreign function
4. Disable memory
Currently: extend rust-bindgen to analyze the function signature.
Future: query LLVM backend for calling convention?
= invoke::<0, AReg2>


int randombytes_buf(void *, size_t);
generates
pub fn randombytes_buf(
&self, *mut c_void, usize,
) -> OGResult<c_int> { }
calls
extern "C" fn randombytes_buf_int(
*mut c_void, usize, &Context);
= invoke::<0, AReg2>
invoke::<stack spill,
&Context argument location>
1. Copy stacked arguments
2. Enable memory protection
3. Invoke foreign function
4. Disable memory protection
5. Copy return value

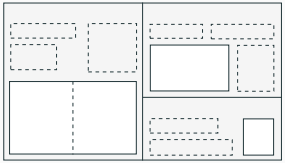

RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
invoke
C


RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C


RAM
Rust Allocation
Rust Stack
Rust Allocation
unsafe{}
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC

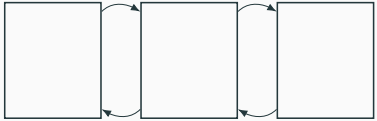

RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C

Encapsulated Functions



✅ Safe Trampolining

- Isolating memory between *safe* and *untrusted*, *foreign* code
- Platform-agnostic
- Retains function call semantics



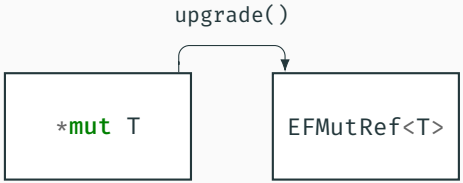
Type Infrastructure

- Securing cross-language interactions
- Enforced at compile time
- Incuring minimal runtime overheads

Gradual Reference Validation

`*mut T`: Arbitrary Pointer into Foreign Memory

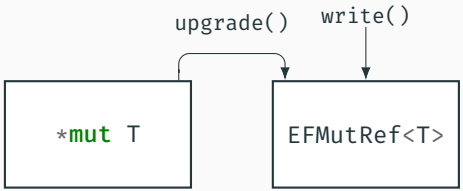
Gradual Reference Validation



`*mut T`: Arbitrary Pointer into Foreign Memory

`EFMutRef<T>`: Well-aligned, Mutably Accessible Object

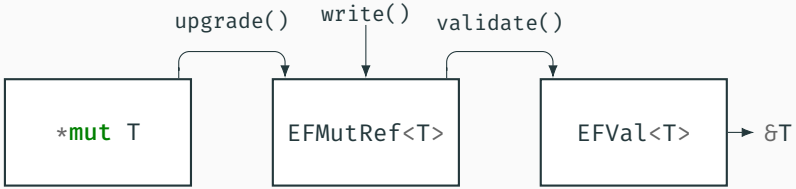
Gradual Reference Validation



`*mut T`: Arbitrary Pointer into Foreign Memory

`EFMutRef<T>`: Well-aligned, Mutably Accessible Object

Gradual Reference Validation

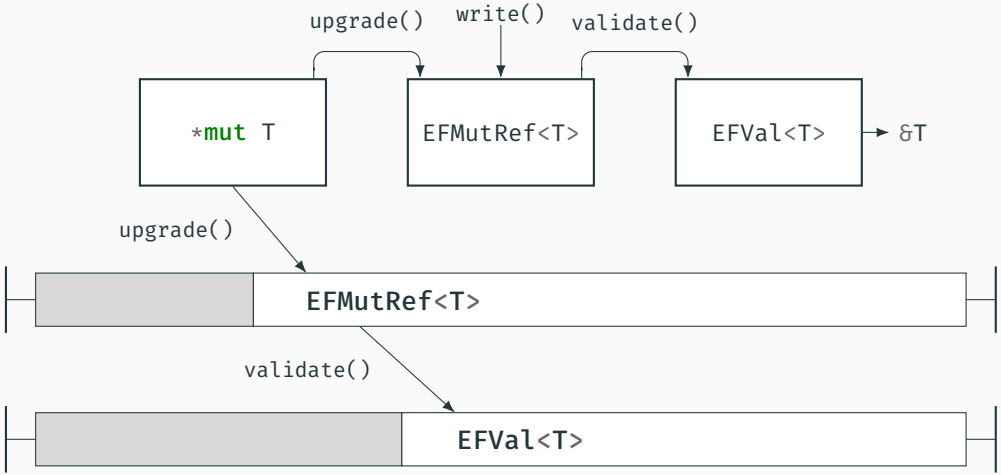


`*mut T`: Arbitrary Pointer into Foreign Memory

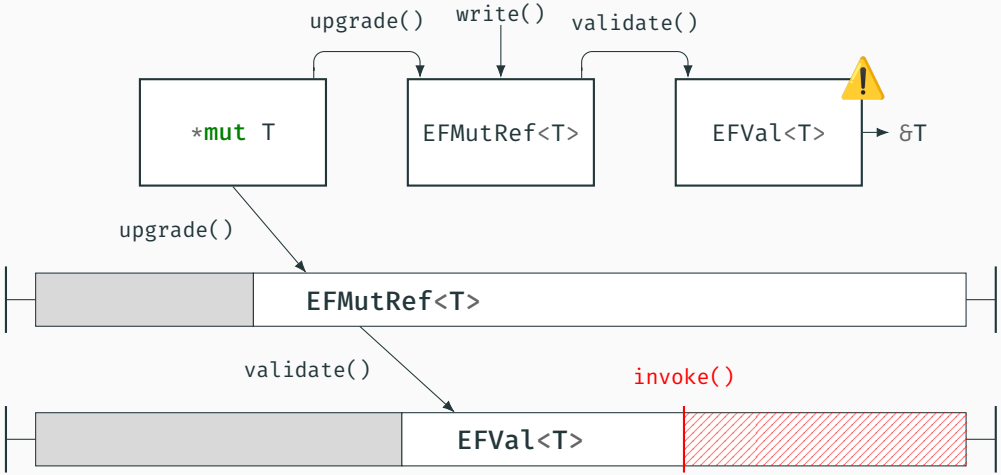
`EFMutRef<T>`: Well-aligned, Mutably Accessible Object

`EFVal<T>`: Object Conforming to Rust's Requirements on *Valid Values*

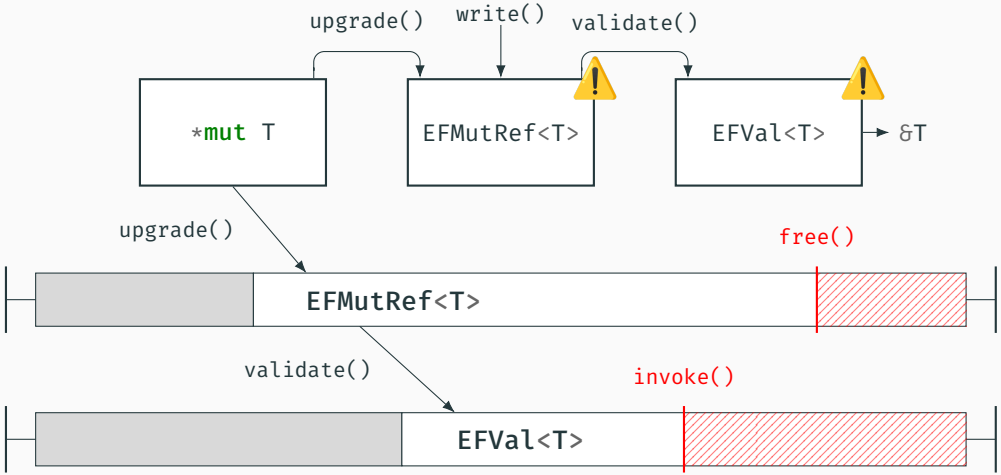
Gradual Reference Validation



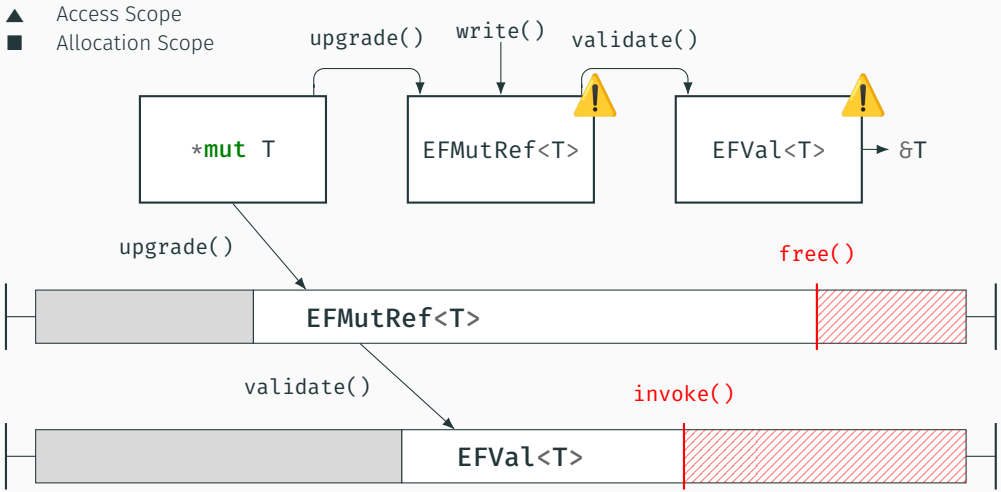
Stale Reference Types Can Violate Safety



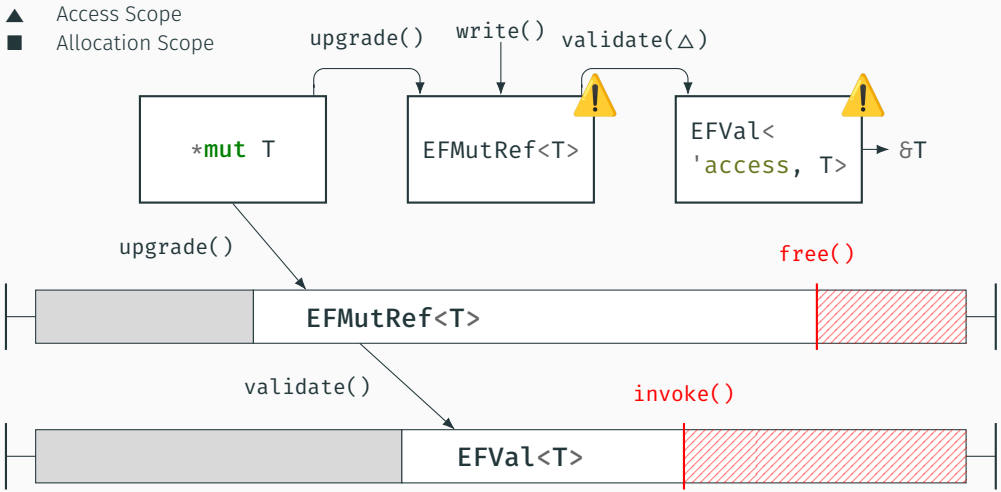
Stale Reference Types Can Violate Safety



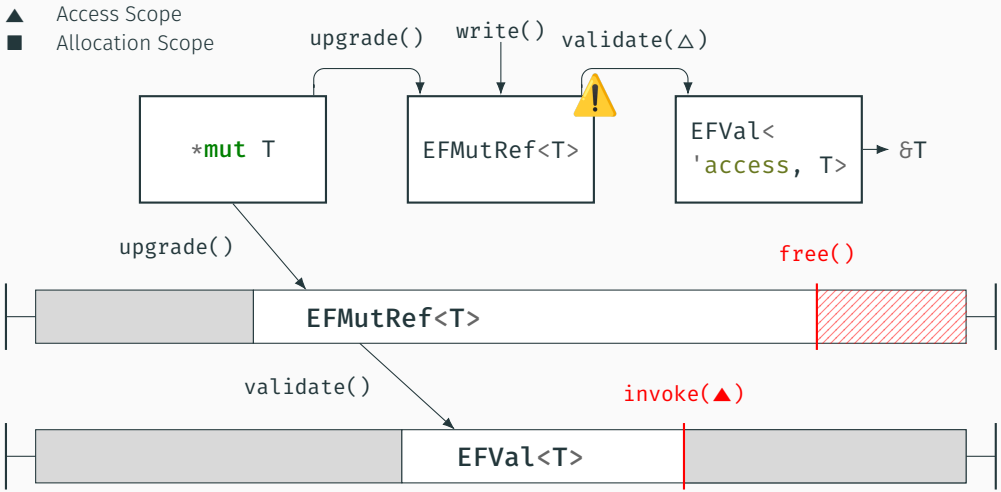
Invalidating References With Scopes



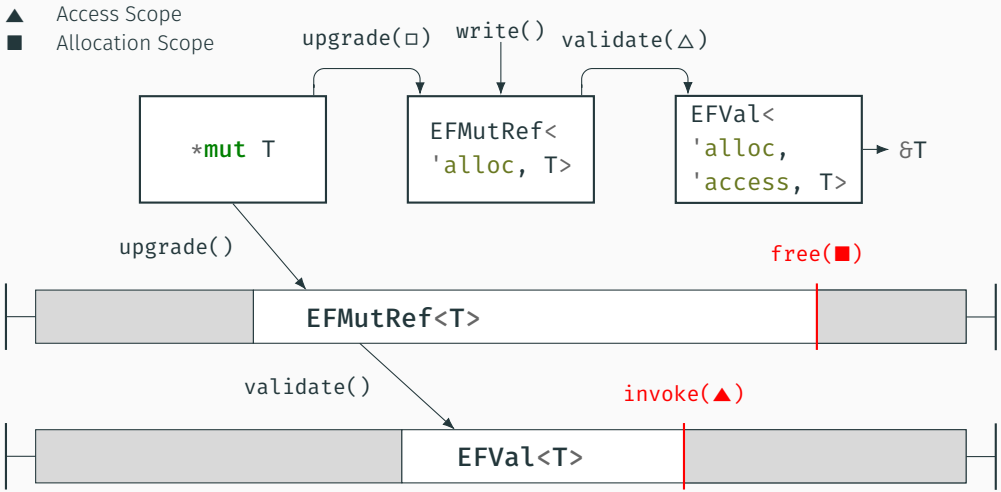
Invalidating References With Scopes

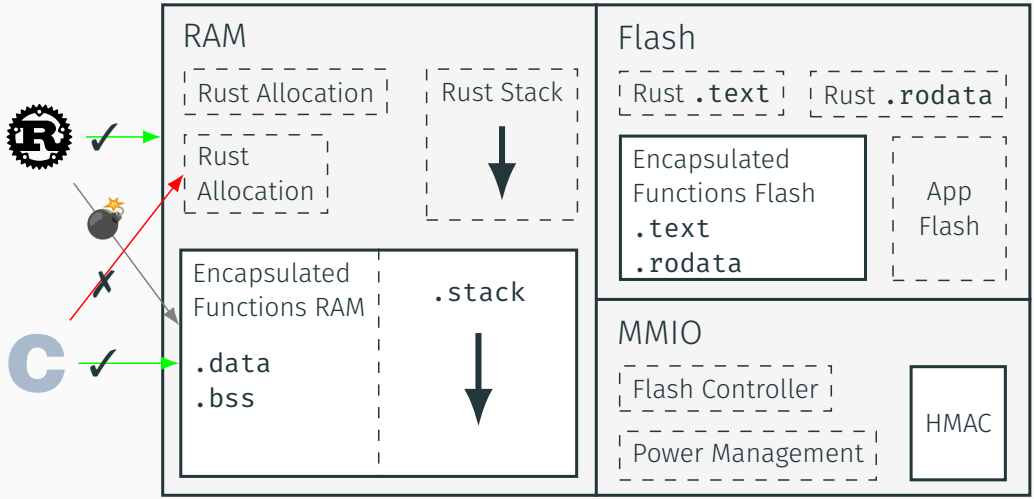


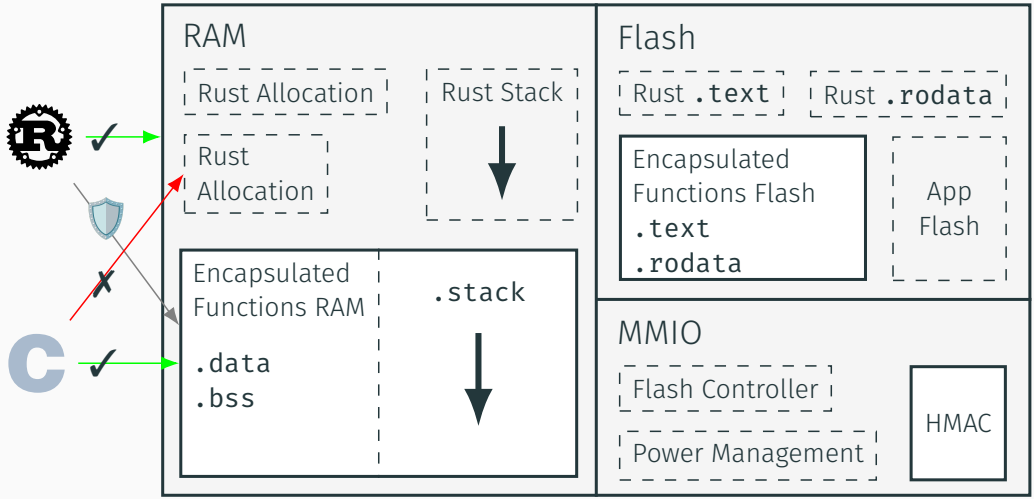
Invalidating References With Scopes



Invalidating References With Scopes




RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C


RAM
Rust Allocation
Rust Stack
Rust Allocation
Encapsulated Functions RAM
.data
.bss
.stack
Flash
Rust .text
Rust .rodata
Encapsulated Functions Flash
.text
.rodata
App Flash
MMIO
Flash Controller
Power Management
HMAC
C

Implementation

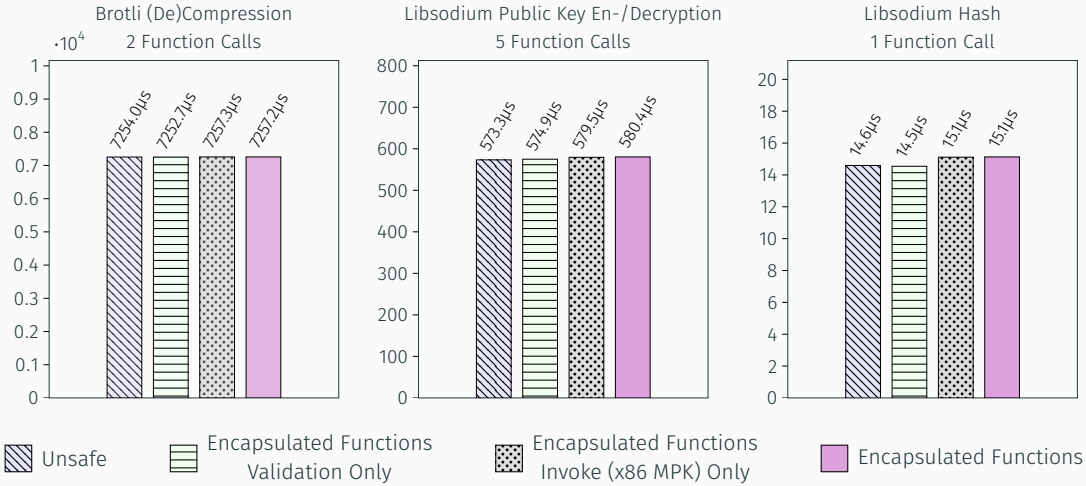
Research Prototype

- Linux Userland with Intel MPK
- Tock OS Kernel on OpenTitan (RISC-V PMP)
- Extends `rust-bindgen` to generate EF bindings from C headers

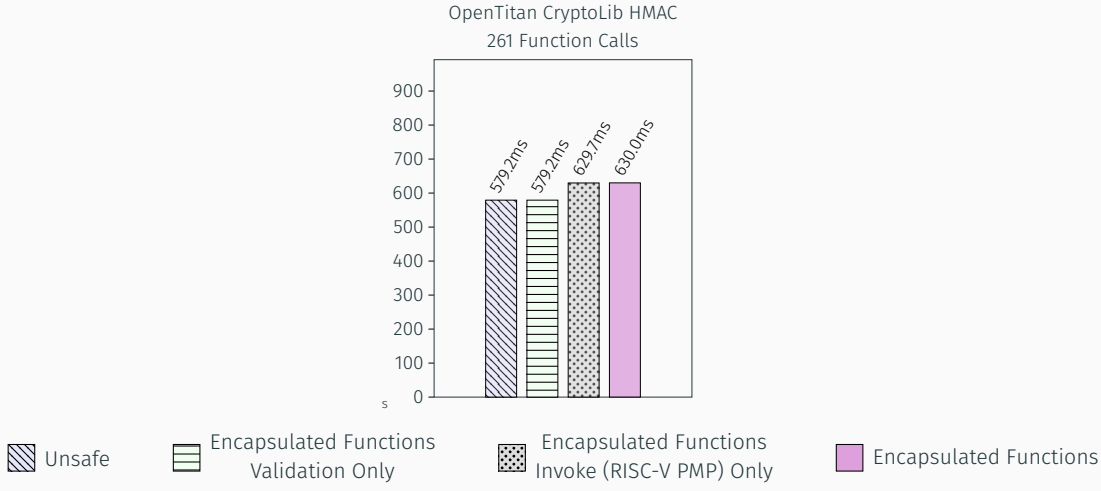
Example Libraries

- `libsodium` – a popular high-level encryption library
- `Brotli` – fast compression library used in Google/Dropbox storage infrastructure
- `CryptoLib` – hardware-accelerated and hardened cryptography library for OpenTitan

Evaluation — Intel MPK — CloudLab cl6420



Evaluation — RISC-V PMP — OpenTitan FPGA



Conclusion

Encapsulated Functions Secures Interactions with Foreign Languages

→ Protects against bugs in foreign code through *safe trampolining*

→ Secures cross-language interactions with a set of *type abstractions*

Future Work

- Calling convention analysis is brittle – use LLVM backend?
- Safety and soundness analysis
- Libraries may rely on global symbols and shared state – rework dynamic loading