

Does Rust SPARK joy?

Recommendations for safe cross-language bindings between Rust and SPARK

Aïssata Maiga

Supervisors: Cyrille Artho (KTH), Florian Gilcher (Ferrous Systems), Yannick Moy (AdaCore)

Examiner: Elena Troubitsyna

Problem

FFI, or Foreign Function Interfaces = interface two languages

Software is built in several languages in safety critical, mostly C/C++ and Ada/SPARK, Java for high-level ...

Usually done with C/C++ (low level, fast) but too much freedom = too much responsibility!

Research question

Goal: **Recommendations for binding Rust and SPARK by preserving type and memory safety, as well as ownership**

Limitations: Preliminary research from automated tool!

Future work! → 1. performance, 2. sustainability impact, 3. industrial integration and maintenance, 4. generalization

Background 1: Ferrocene



How many planes?

Ferrous Systems/AdaCore project

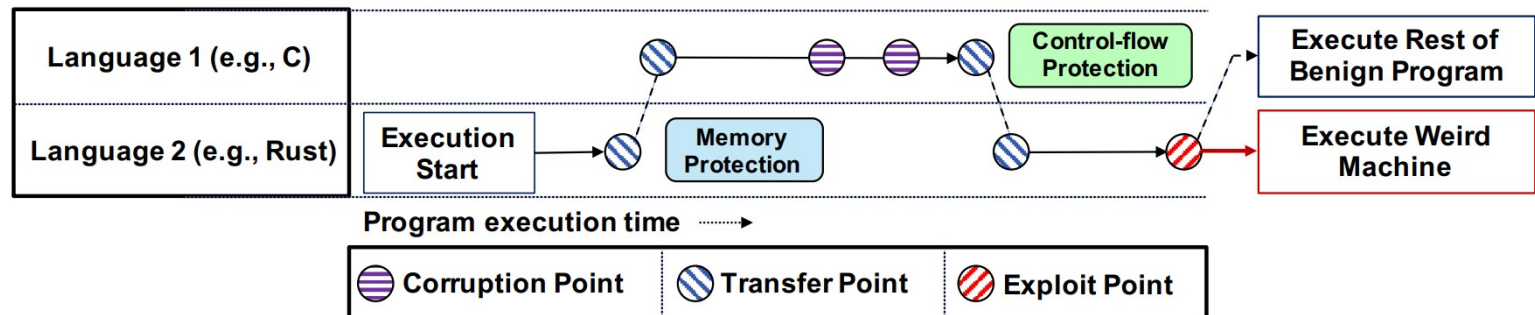
Qualification of one version of the Rust compiler

Produce certified code

Background 2

- Memory errors come from unsafe languages[1] → safety-critical, systems programming, low level
- FFI in software engineering is difficult and error prone, reason in new paradigm [2]
- FFI safe/unsafe languages is potentially more unsafe than the rest! [3]
- FFI resuscitate old avenues of attack → weakest link

Cross-Language Attacks (CLA) transfer back and forth between languages to circumvent deployed defenses. Mergendahl et al., [2]



Background 3: Rust and SPARK

| Paradigm | Languages |
|-----------------|---|
| Functional | Erlang, F#, Haskell, Lisp, Ocaml, Perl, Racket, Ruby, Rust ; |
| Imperative | Ada , C, C++, F#, Fortran, Go, Ocaml, Pascal, Rust ; |
| Object-Oriented | Ada , C++, C#, Chapel, Dart, F#, Java, JavaScript, Ocaml, Perl, PHP, Python, Racket, Rust , Smalltalk, Swift, TypeScript; |
| Scripting | Dart, Hack, JavaScript, JRuby, Lua, Perl, PHP, Python, Ruby, TypeScript; |

Energy Efficiency across Programming Languages
How Do Energy, Time, and Memory Relate?

R. Pereira et al.[3]

| | Energy | | Time | | Mb |
|----------------|--------|----------------|-------|----------------|-------|
| (c) C | 1.00 | (c) C | 1.00 | (c) Pascal | 1.00 |
| → (c) Rust | 1.03 | → (c) Rust | 1.04 | (c) Go | 1.05 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| → (c) Ada | 1.70 | → (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | → (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | → (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

Background 4: Rust safety guarantees

- Rust has type safety, memory safety and ownership
- Unsafe Rust allows to deviate from the borrow checker but invariants must be followed
- Unsafe Rust **limited to 4 operations**:
 - 1) Dereference a raw pointer
 - 2) Call an unsafe function or method
 - 3) Access or modify a mutable static variable
 - 4) Implement an unsafe trait

Invariants example

Safety

Behavior is undefined if any of the following conditions are violated:

1. data must be valid for both reads and writes for `len * mem::size_of::<T>()` many bytes, and it must be properly **aligned**. This means in particular:
 - 1.1 The entire memory range of this slice must be contained within a **single allocated object**! Slices can never span across multiple allocated objects.
 - 1.2 data must be **non-null and aligned** even for zero-length slices.

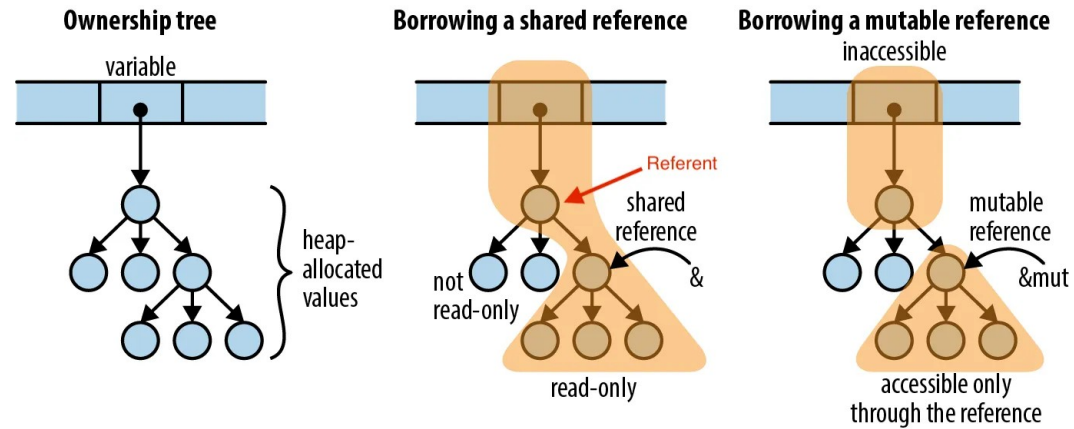
...

https://doc.rust-lang.org/beta/std/slice/fn.from_raw_parts_mut.html

Background 5: SPARK safety guarantees

- SPARK: biggest possible subset of Ada with specification and sound verification
- Pointers ownership rules added in 2019, forbids aliasing, allows beginning aliasing (“observing”)
- Support for mathematical proofs:
 - 1) absence of runtime exceptions
 - 2) verify the fulfillment of security and safety properties
 - 3) or establish that the software follows its specifications/behavior

Background 3: What is ownership?



"Understanding Ownership"
Luis Soares [5]

Rust ownership:

- one place in memory has one owner
- the compiler annotates the types with lifetimes
- the compiler removes the memory when the owner is out of scope.

SPARK ownership:

- inspired by Rust
- the programmer need to implement deallocating functions
- the compiler will not throw an error in case of illegal operations

Method 1: Overview

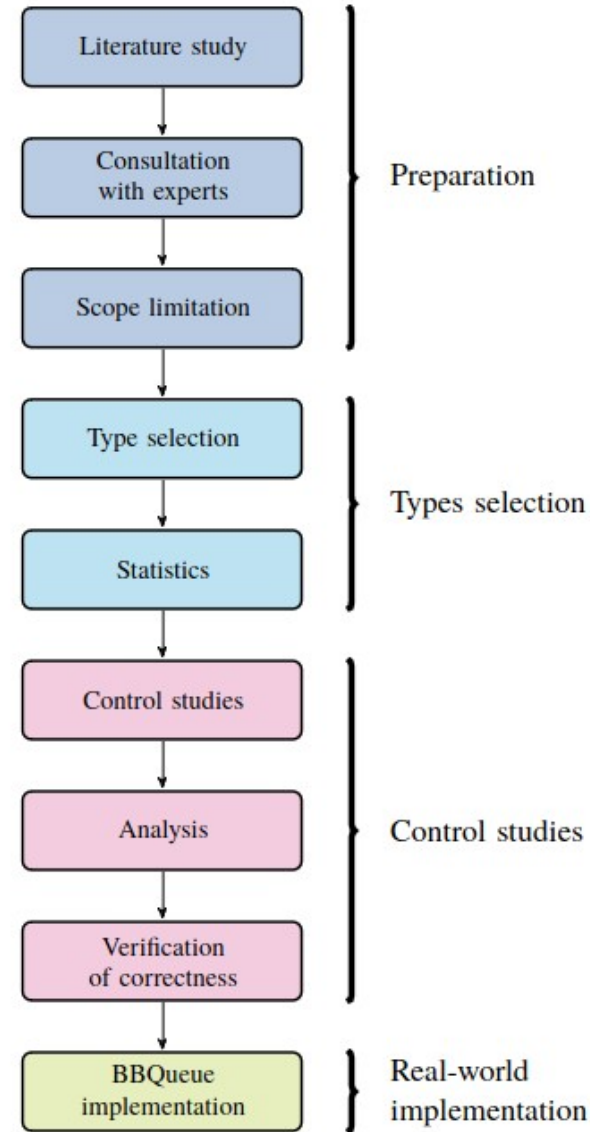


Figure 4.1: Research Process

Method 2: Rust Type selection

- About 2MLoC
- Compiler, crates.io, lib.rs (17 important rust projects)
- A systems' programming language profile.

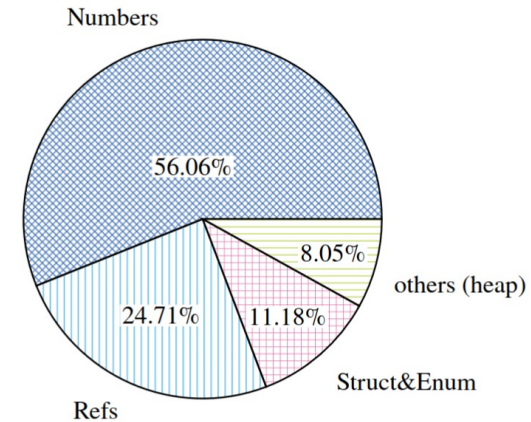
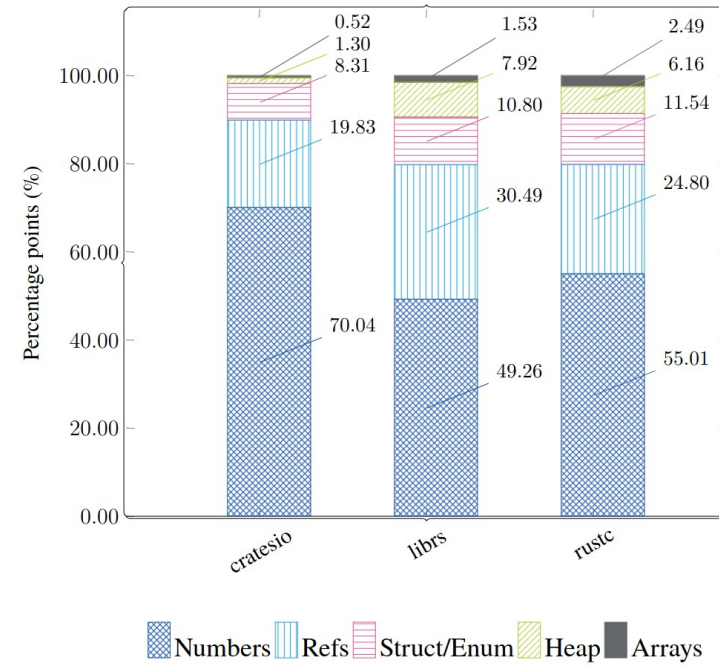


Figure 4.3: Rust types distribution

Method 3: SPARK

Type selection

- 27 kLoC for SPARK
- Code base are proprietary → Three OSS projects + Expert
- Low-level language, very little heap types

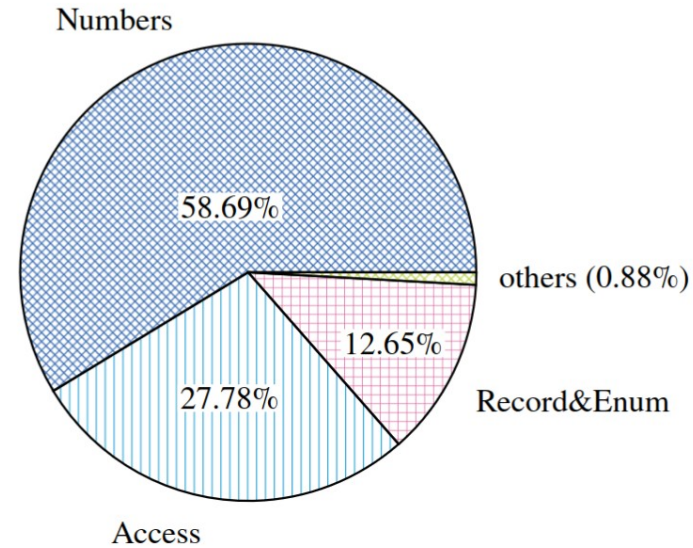
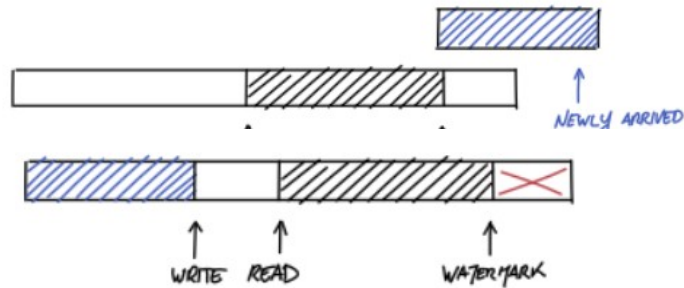


Figure 4.4: SPARK types distribution

Method 4: BBQueue

- Bipartite Circular Buffer, multithread FIFO queue
- Optimized for DMA in embedded systems
- Concurrency ensured by reading atomic variables: lock-free implementation



"The design and implementation of a lock-free ring-buffer with contiguous reservations[6]

```
#[repr(C)]
pub struct BBBuffer {
    // Pointer to:
    // Box<UnsafeCell<u8; 128>>, it is an "Option" for embedded systems
    pub buf: AtomicPtr<u8>,
    pub buf_len: AtomicUsize,

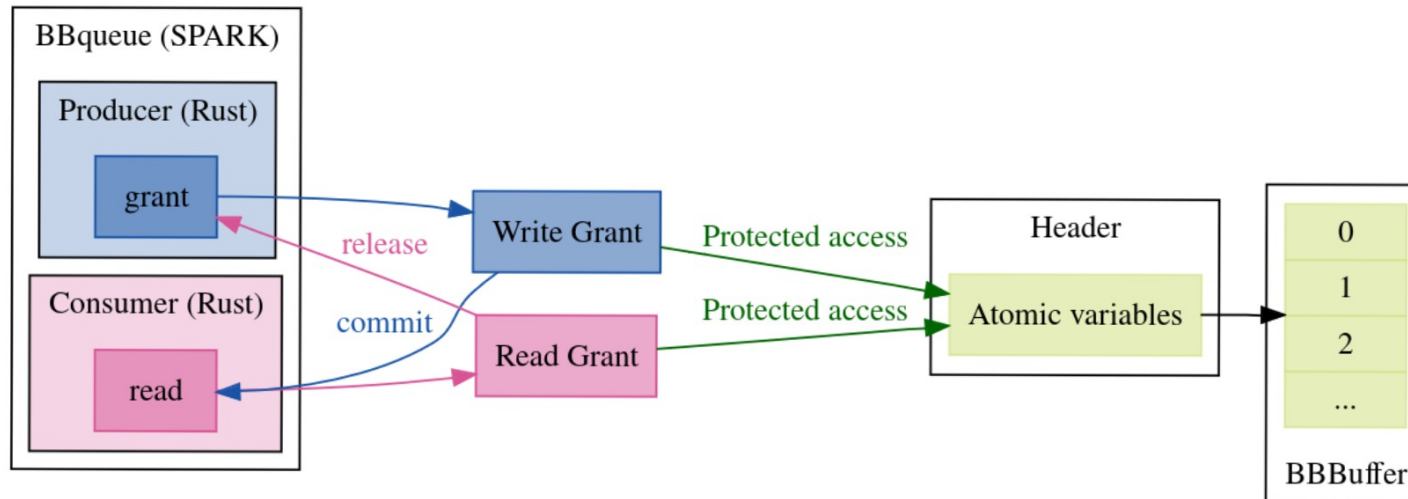
    /// Where the next byte will be written
    pub write: AtomicUsize,

    /// Where the next byte will be read from
    pub read: AtomicUsize,

    ....
}
```

Method 5: BBQueue

- Exists in Rust and SPARK
- Formally proven in SPARK with invariants and the GNATprover
- Data structures are complex enough, continuous memory blocks which makes it ideal for experiment



Experiments overview: stack

- Simpler types, aggregates, composite types
- Focus on type safety (memory safety and ownership taken care by the compiler for Copy)
- Demonstrating memory errors, undefined behavior
- Memory safety verified with valgrind

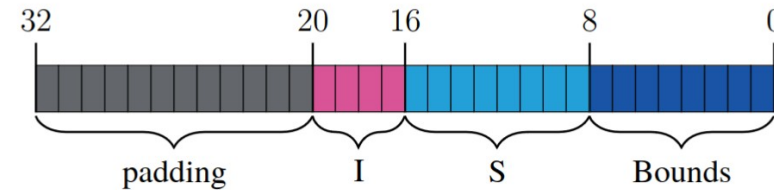



Figure 6.3: Composite type

- Working with compiler and language documentation
- Working with layout

Experiments overview: heap

Same as stack types, but in addition:

- Defining data structure (“extra wrapper”) to hold complex and nested data types
- Implementing native methods/traits for creating, accessing, and modifying complex data types
- “Good FFI citizen”: Implementing method/traits for dereferencing and dropping memory after usage
- Minimizing unsafe code and delegating to the language semantics as much as possible

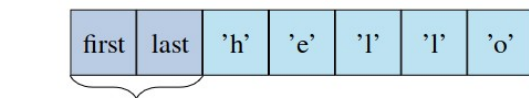


```
// One function allocates memory for a new object.  
extern fn ECDSA_SIG_new() -> *mut ECDSA_SIG;  
// And another accepts a pointer created by new  
// and deallocates it when the caller is done with it.  
extern fn ECDSA_SIG_free(sig: *mut ECDSA_SIG);
```

J. Gjengset, Rust for Rustaceans [7]

Example: SPARK String

- String is a heap type: we need first to find correct sub-types (i32? c_char or u8?)
- Fat pointer (bounds + array)
- Finding instructions in the GNAT compiler book: bounds come last when FFI with C conventions.
- Follow the C ABI

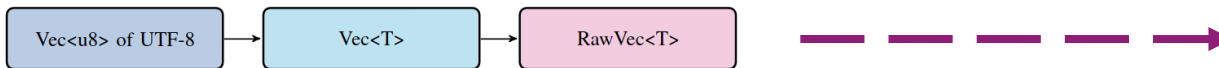


Listing 6.1 Reconstructed Ada String

```
#[repr(C)]
pub struct AdaBounds {
    first: i32,
    last: i32,
}
#[repr(C)]
pub struct AdaString {
    // internal memory allocation that must follow Ada convention
    data: *mut c_char,
    bounds: *const AdaBounds,
}
```

Example: Rust String

- String is still a heap type: Vec of u8, building on inner types
- Still a fat pointer with capacity, len, buffer
- It is not FFI safe and cannot be exported
- Must be encoded as c_char for Ada/SPARK (u8 will work but safety and consistency are key!) len and capacity are not Integers!
- Forgetting must be implemented



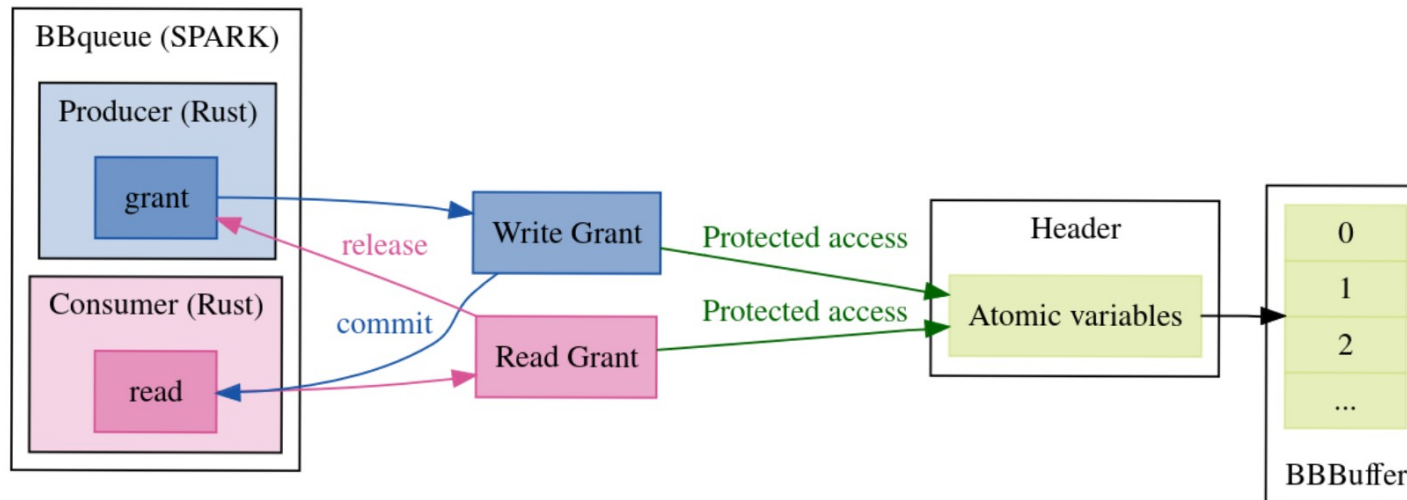
Listing 6.2 Deconstructed Rust String

```
#[repr(C)]
struct RustFFIString {
    ptr: *mut c_char,
    len: usize,
    cap: usize,
}

impl RustFFIString {
    fn from_string(s: String) -> Self {
        let raw_str = RustFFIString {
            ptr: s.as_ptr() as *mut c_char,
            len: s.len(),
            cap: s.capacity(),
        };
        // forgetting allows the language to not drop
        // the object when it goes out of scope
        std::mem::forget(s);
        // returning the object
        raw_str
    }
}
```

BBQueue

- GrantW, GrantR → Header → Buffer
- Grant holds a mutable reference to the header
- The Grant can atomically touch the variables of the header



BBQueue

We must:

1. guarantee consistent layout across the FFI border (guaranteed by the data structure);
2. provide pointers to the grant and the content of the header;
3. ensure that the header fields are of the right size and are atomic;
4. ensure the absence of undefined behavior where it can be introduced;
5. ensure the functions in SPARK interact with the atomic variables in the exact prescribed order — to guarantee thread safety;
6. Respect the good FFI citizen principles.

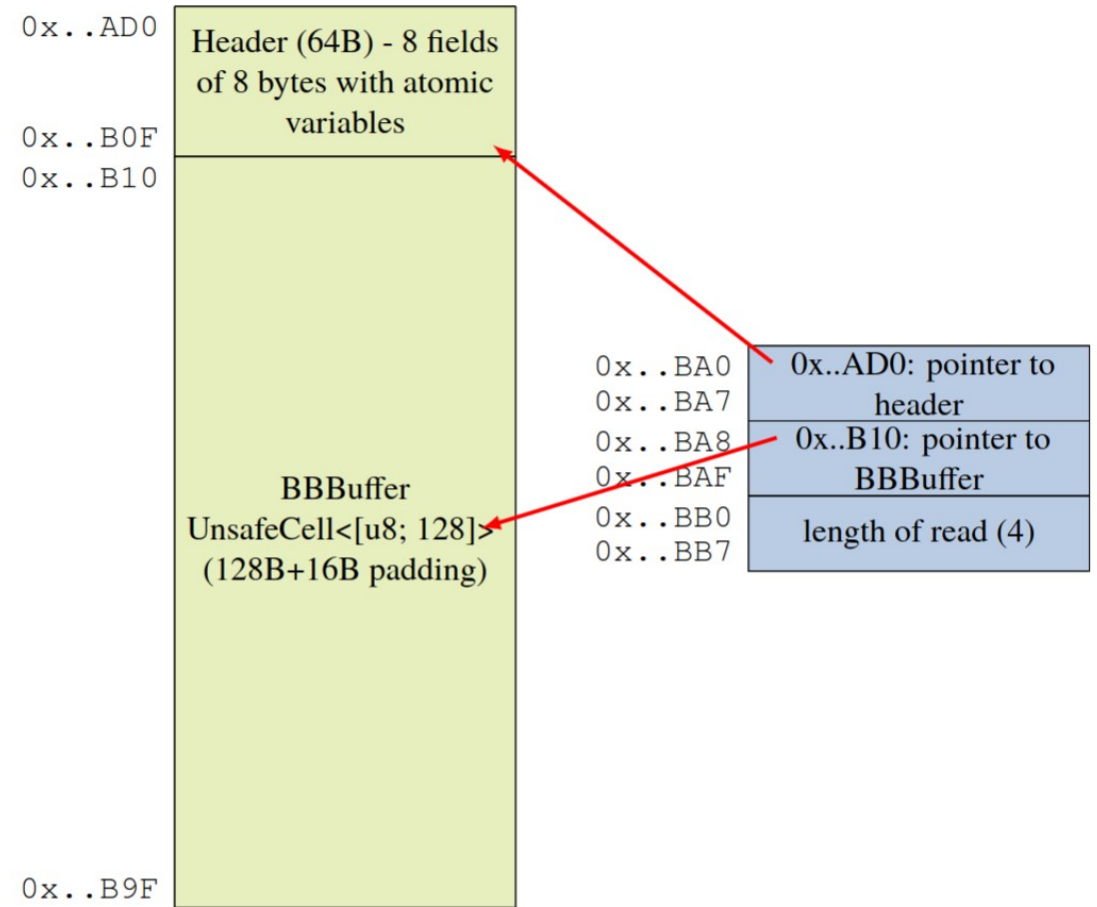


Figure 6.5: BBQueue Read grant in memory (not to scale!)

■ Memory assigned in Rust.
■ Memory passed to SPARK.

Results 1

- Rely on documentation. It is spread between language and compiler docs.
- Good FFI citizen (do not unwind, alloc/dealloc must be local) is a must and empirically no memory errors were found with valgrind
- Great tooling is lost in transaction: the compiler, miri is not mature for FFI, same as GNATprove
- It is extremely error prone as per literature.
- References cannot be used across FFI! Back to pointer logic
- Use separation of concerns



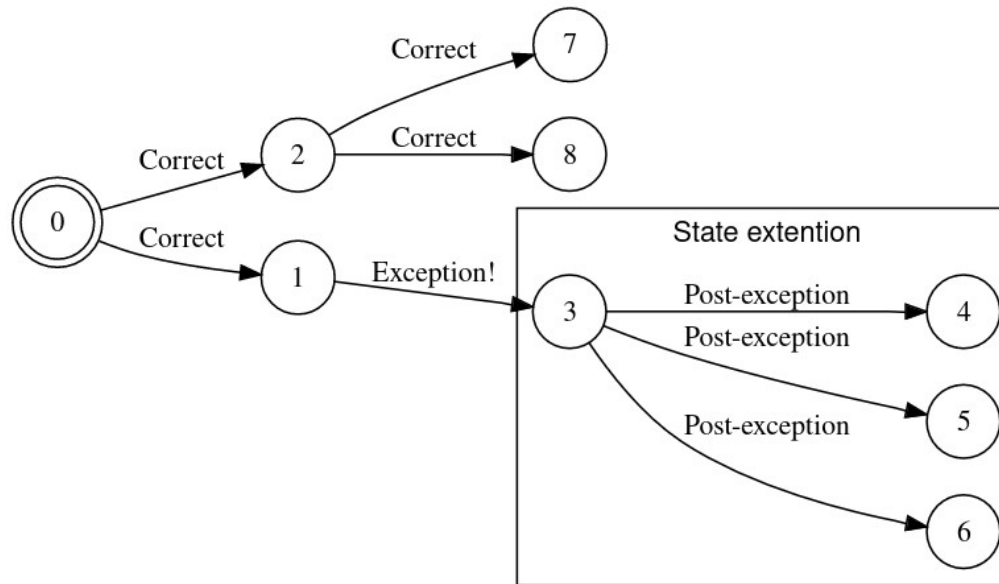
| pub Heap object | priv Heap object |
|---|--|
| <ol style="list-style-type: none">1. Unique2. Can be borrowed3. Must implement language memory guarantees | <ol style="list-style-type: none">1. Fat pointer with data and meta-data (bounds, capacity...)2. Hidden from the user3. Memory management relies on the programmer |

Results 2: memory interaction

- Some memory regions are inaccessible. SPARK must use named pointers.
- SPARK is less flexible in its ownership (no lifetimes) in some respects (assumes ownership over the whole array in one function, no slicing)!
- But SPARK is more flexible in memory allocation: storage pools. Rust has Allocator trait but not stabilized.
- SPARK is more flexible to interact with external code/Rust protects most types

Results 3: Exceptions and panic!()

- Unwinding across FFI is undefined behavior
- In microcontrollers it is impossible to unwind
- It is bad FFI citizen behavior!
- SPARK has no support for exceptions
- We can never go back to a “safe” state



Conclusions: does Rust SPARK joy?

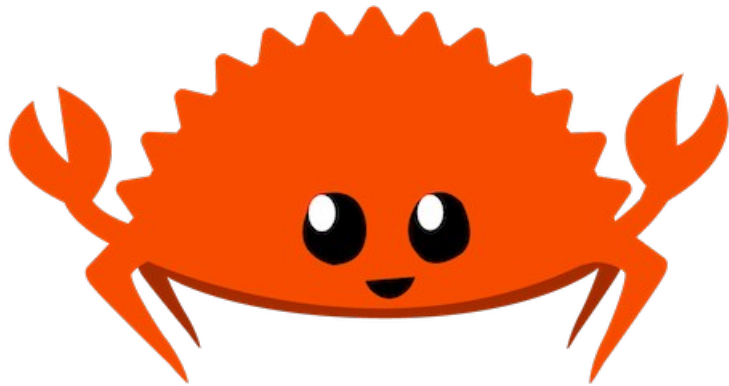
A tool is feasible but!

- Gather information that is compiler and platform specific
- If being careful the **consistency of Rust SPARK** is preserved,
- but in case of error we are back to the weakest link
 - It becomes extremely easy to "lie" to the compiler, human error etc
- Strong reliance on the C ABI
- Absence on tooling and compiler protection

Future work

- Find/automate about types
- Get inspiration from Bindgen/Cbindgen
- Develop tooling for FFI Rust/SPARK
- Assess performance
- Formal verification!
- Generalisations to other safe languages

Thank you for listening!

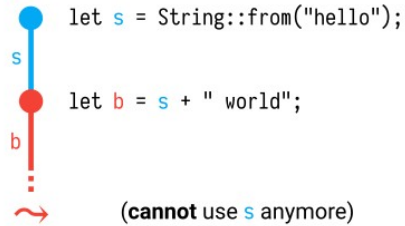


THANK YOU

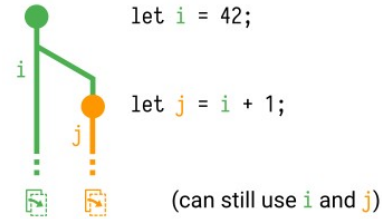
- [1] “NSA Releases Guidance on How to Protect Against Software Memory Safety Issues,” National Security Agency/Central Security Service, 2022. <https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues> (accessed Feb. 13, 2023).
- [2] S. Li, “Improving Quality of Software with Foreign Function Interfaces using Static Analysis,” Doctoral dissertation, Lehigh University, 2014. Accessed: Feb. 13, 2023. [Online]. Available: <https://www.semanticscholar.org/paper/Improving-Quality-of-Software-with-Foreign-using-Preserve-Li/8d0b6db3858946c27657567d42f684a32d34e4f3>
- [3] S. Mergendahl, N. Burow, and H. Okhravi, “Cross-Language Attacks,” in Proceedings 2022 Network and Distributed System Security Symposium, San Diego, CA, USA: Internet Society, 2022. doi: 10.14722/ndss.2022.24078.
- [4] R. Pereira et al., “Energy efficiency across programming languages: how do energy, time, and memory relate?,” in Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, in SLE 2017. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 256–267. doi: 10.1145/3136014.3136031.
- [5] L. Soares, “Understanding Ownership in Rust with Examples,” Coinmonks, May 23, 2023. <https://medium.com/coinmonks/understanding-ownership-in-rust-with-examples-73835ba931b1> (accessed May 24, 2023).
- [6] J. Munns, “The design and implementation of a lock-free ring-buffer with contiguous reservations - Ferrous Systems,” Mar. 06, 2019. <https://ferrous-systems.com/blog/lock-free-ring-buffer/> (accessed Apr. 28, 2023).
- [7] J. Gjengset, Rust for Rustaceans. no starch press, 2021. Accessed: Mar. 06, 2023. [Online]. Available: <https://rust-for-rustaceans.com>
- [8] L. Szekeres, M. Payer, Tao Wei, and D. Song, “SoK: Eternal War in Memory,” in 2013 IEEE Symposium on Security and Privacy, Berkeley, CA: IEEE, May 2013, pp. 48–62. doi: 10.1109/SP.2013.13.
- [9] “Memory Safe Languages in Android 13,” Google Online Security Blog. <https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html> (accessed May 25, 2023).

Copy and move semantics

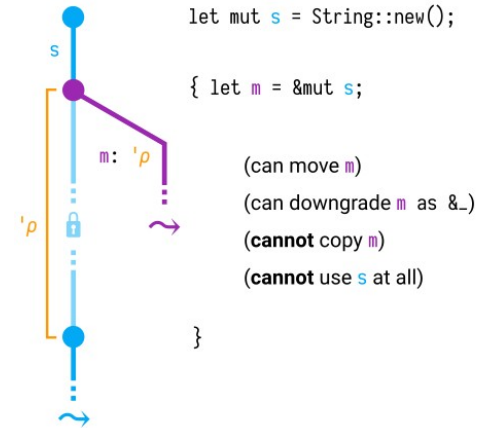
~ move (for types that do not implement Copy)



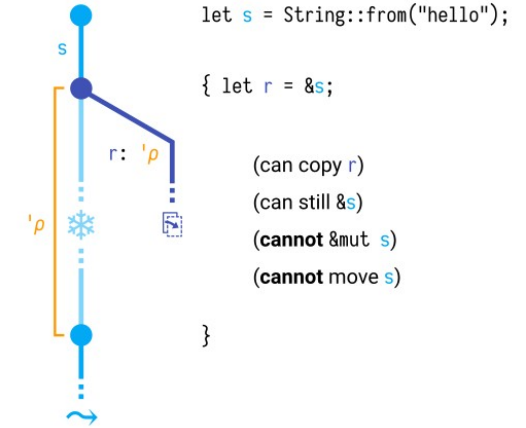
copy (for types that do implement Copy)



🔒 mutable borrow



* borrow



&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

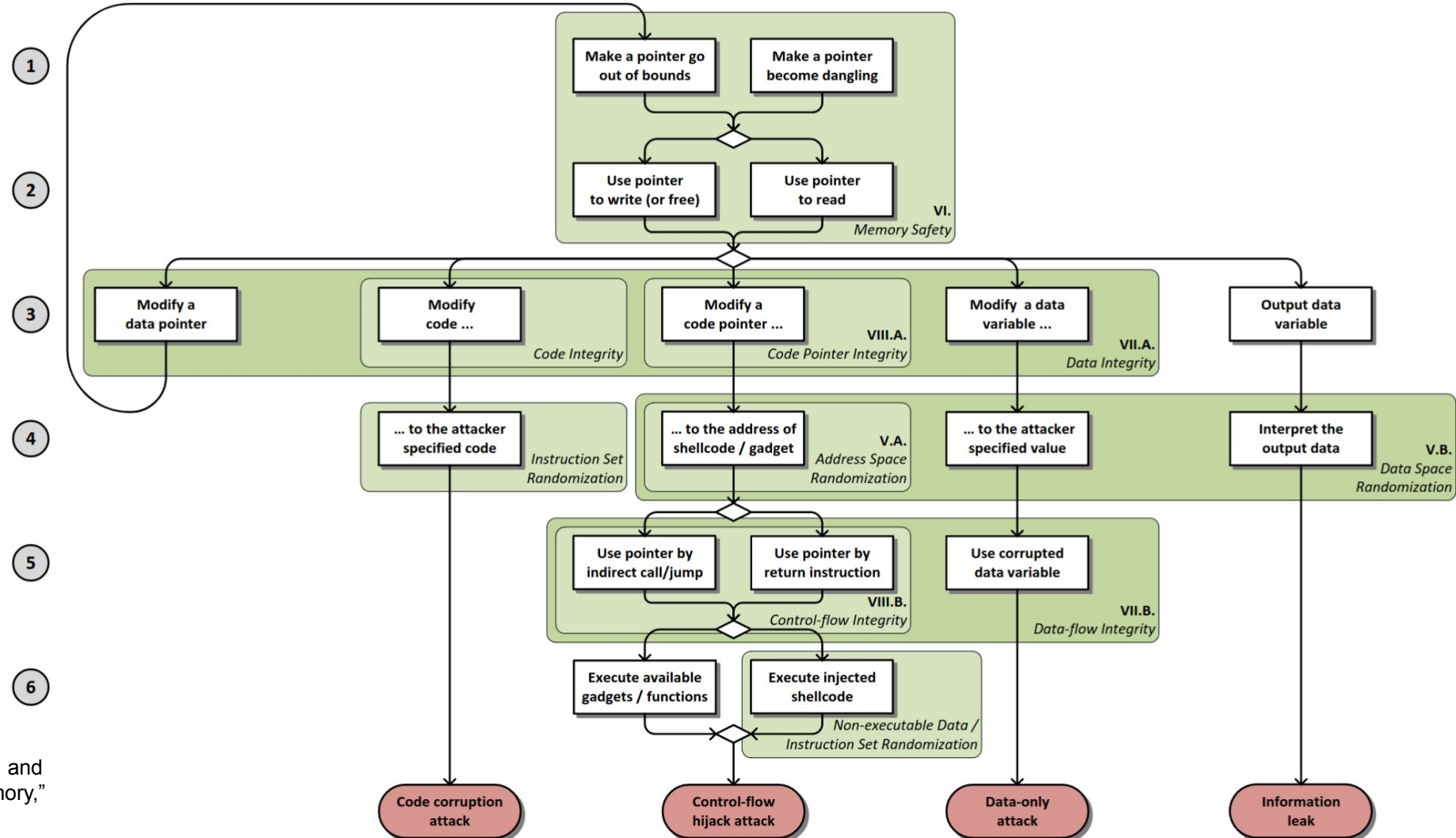


&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

But what is wrong with C?

Memory Management:
no garbage collector, no
memory check at all.
Undefined Behavior: a
lot of undefined
behaviour, MISRA C
cannot cover 27/143
(<https://embeddedcomputing.com/technology/security/misra-c-cert-c-other-standards/the-place-for-misra-c-in-safe-secure-programming-a-comparison-with-spark>)
Type Safety: weak,
casts are authorized
Pointers: structurally
equivalent. everything is
possible
Concurrency



[8] L. Szekeres, M. Payer, Tao Wei, and D. Song, "SoK: Eternal War in Memory,"

Experiments (1) Rust → SPARK

| Name | Description | Test, Issue or Question(s) addressed |
|------------------------|--|--|
| adder_ada | Sending and reading Integers | Sending and accessing scalar types by copy and reference. Verifying if the <code>in out</code> parameter annotation can be used to pass objects by reference in FFI as in a usual SPARK/Ada subprogram |
| swap_ada | Swapping <code>in out</code> Integers by reference | Sending, accessing and manipulating simple data types by reference, using <code>in out</code> (as opposed to the <code>access</code> type) |
| array_sender | Iteration through an Integer array | Iterating through a composite type of known size at compile time. Triggering compiler error by going out of bounds (OOB). Introducing sanitizers <code>-Z sanitizer=address</code> on the Rust and <code>-fsanitize=address</code> on SPARK side to test for undefined behavior. |
| fat_pointer | Sending and accessing composite data type (Struct with String and Integer) | Reconstructing a composite datatype, using layout information provided by the compiler. |
| print_enum | Sending and accessing enums with a given size | Verifying enums behave as expected (as Copy). Comparing Rust's <code>#[repr(u8)]</code> and SPARK <code>Size => 8</code> . |
| fat_pointer_over-write | As above, sending and accessing composite data type and writing over the sent String | Same as above, but with additional implementation of separation of concerns between pointer logic (FFI side hidden from user) and business logic (user side) |
| mem_violation | Passing created String type and reading OOB | Verifying what happens with error unwinding across the FFI border. |
| panics | Calling explicitly a Rust <code>panic!()</code> | Same as above. |
| p pointer | Passing non-anonymous access type to a Struct with two Integer and increment both fields | Working with <code>access</code> types: verify that non-anonymous pointers can be passed and received in Rust as references, and study the behavior of anonymous access types. |

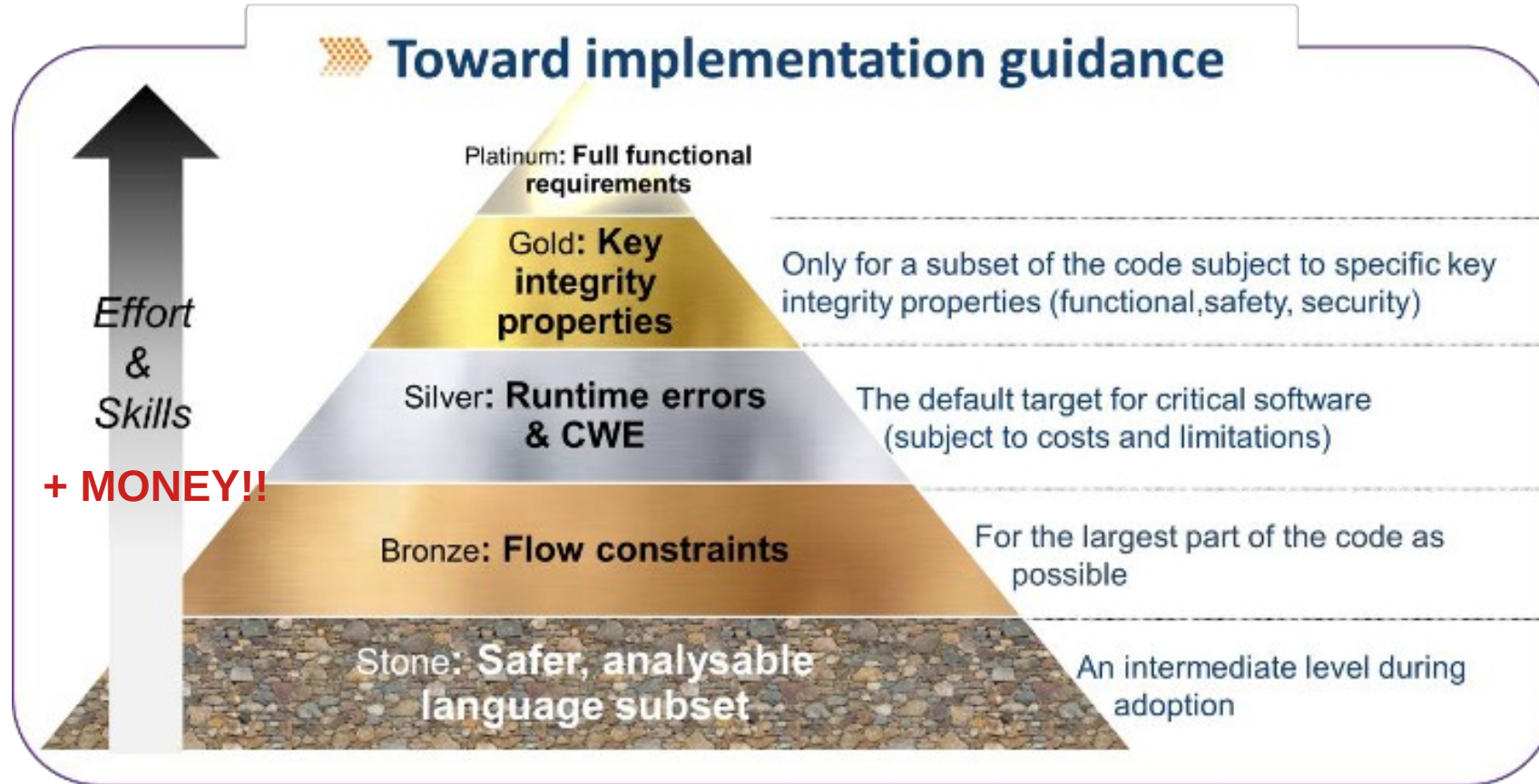
Experiments

(1)

SPARK → Rust

| Name | Description | Test or Issue addressed |
|-----------------------|--|--|
| num_swapper | Swapping Integers sent by copy | Accessing and manipulating simple data types by copy. |
| ada_adder | Sending and reading Integers by copy and reference | Sending and accessing scalar types, both by copy and reference, to be read or incremented. This experiment is also used to analyze the compiler's behavior with miri. ¹ |
| ada_divider | Passing Integers by copy & testing undefined behavior | Manipulating Integers and returning a new copy object. Testing undefined behavior and unwinding a program at the FFI border. |
| enum_sender | Sending and accessing enums with fixed size | Verifying enums behave as expected (as Copy). Verify enums can be reconstructed on the other side of the FFI frontier Comparing <code>#[repr(u8)]</code> and <code>Size => 8</code> . |
| array_sender | Iteration through a char array | Iterating through a type for which size is known at compile time and verify it behaves as expected. The chosen array type is bounded String. |
| ada_forbidden_mem | Sending String dynamically allocated in different regions and trying to recover the object on the Rust side | Understanding where non-anonymous/anonymous access types are allocated in SPARK and what kind of memory pointers are possible to pass. |
| num_allocator | Allocating and deleting Integers on the heap in SPARK | Managing heap allocated memory according to good FFI citizen principles. Writing appropriate allocation and deallocation functions. |
| ada_string_overwriter | Sending and accessing String created in SPARK, with deallocation managed according to the good FFI citizen principle | Same as above, with some additions: separation of concerns between pointer and user data, and implementing traits that are found in the original Rust String |

SPARK



Experiments

(1)

SPARK → Rust

| Name | Description | Test or Issue addressed |
|-----------------------|--|--|
| num_swapper | Swapping Integers sent by copy | Accessing and manipulating simple data types by copy. |
| ada_adder | Sending and reading Integers by copy and reference | Sending and accessing scalar types, both by copy and reference, to be read or incremented. This experiment is also used to analyze the compiler's behavior with miri. ¹ |
| ada_divider | Passing Integers by copy & testing undefined behavior | Manipulating Integers and returning a new copy object. Testing undefined behavior and unwinding a program at the FFI border. |
| enum_sender | Sending and accessing enums with fixed size | Verifying enums behave as expected (as Copy). Verify enums can be reconstructed on the other side of the FFI frontier Comparing <code>#[repr(u8)]</code> and <code>Size => 8</code> . |
| array_sender | Iteration through a char array | Iterating through a type for which size is known at compile time and verify it behaves as expected. The chosen array type is bounded String. |
| ada_forbidden_mem | Sending String dynamically allocated in different regions and trying to recover the object on the Rust side | Understanding where non-anonymous/anonymous access types are allocated in SPARK and what kind of memory pointers are possible to pass. |
| num_allocator | Allocating and deleting Integers on the heap in SPARK | Managing heap allocated memory according to good FFI citizen principles. Writing appropriate allocation and deallocation functions. |
| ada_string_overwriter | Sending and accessing String created in SPARK, with deallocation managed according to the good FFI citizen principle | Same as above, with some additions: separation of concerns between pointer and user data, and implementing traits that are found in the original Rust String |

Experiments

(1)

SPARK → Rust

| Name | Description | Test or Issue addressed |
|-----------------------|--|--|
| num_swapper | Swapping Integers sent by copy | Accessing and manipulating simple data types by copy. |
| ada_adder | Sending and reading Integers by copy and reference | Sending and accessing scalar types, both by copy and reference, to be read or incremented. This experiment is also used to analyze the compiler's behavior with miri. ¹ |
| ada_divider | Passing Integers by copy & testing undefined behavior | Manipulating Integers and returning a new copy object. Testing undefined behavior and unwinding a program at the FFI border. |
| enum_sender | Sending and accessing enums with fixed size | Verifying enums behave as expected (as Copy). Verify enums can be reconstructed on the other side of the FFI frontier Comparing <code>#[repr(u8)]</code> and <code>Size => 8</code> . |
| array_sender | Iteration through a char array | Iterating through a type for which size is known at compile time and verify it behaves as expected. The chosen array type is bounded String. |
| ada_forbidden_mem | Sending String dynamically allocated in different regions and trying to recover the object on the Rust side | Understanding where non-anonymous/anonymous access types are allocated in SPARK and what kind of memory pointers are possible to pass. |
| num_allocator | Allocating and deleting Integers on the heap in SPARK | Managing heap allocated memory according to good FFI citizen principles. Writing appropriate allocation and deallocation functions. |
| ada_string_overwriter | Sending and accessing String created in SPARK, with deallocation managed according to the good FFI citizen principle | Same as above, with some additions: separation of concerns between pointer and user data, and implementing traits that are found in the original Rust String |