Does Rust SPARK joy? Safe bindings between safe languages, applied to the BBQueue library.

AÏSSATA MAIGA, Royal Institute of Technology, Sweden

TODO:

- Rust and SPARK safe languages that implement "some" features from formal systems. Type check can be one and SPARK has contracts...
- Could we benefit from those language guarantees when interfacing it.
- Tested in a master thesis with control studies and finalized with a library that exist in both languages
- the result is that most of the consistency can be maintained but one must be especially careful at the FFI frontier

CCS Concepts: • **Formal verification** → **SPARK, Rust**; • **Safety** → *Memory and Type safety, ownership*; • **Memory** → Foreign Function Interfaces.

Additional Key Words and Phrases: safety-critical code, embedded systems

ACM Reference Format:
Aïssata Maiga. 2018. Does Rust SPARK joy? Safe bindings between safe languages, applied to the BBQueue library.. In . ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Safety-critical domains such as avionics, automotive, and medical must achieve high-grade safety and reliability in software systems. One of the ways to achieve this goal is to rely on formal methods, as emphasized by standards such as DO-178C and ISO 26262.

This paper adapts the findings of a Master's thesis on interfacing Rust and SPARK. It involves bridging through Foreign Function Interfaces (FFI) without resorting to the C programming language, which is typically the **lingua franca** for these types of operations. It summarizes best practices for creating safe bindings between Rust and SPARK while maintaining memory safety, type safety, and ownership and presents insights from porting the library BBQueue - a circular bipartite buffer with an implementation in both languages.

Rust is a systems programming language that gained considerable traction during last decade. It implements memory, type safety and an ownership system that eliminates many memory vulnerabilities low level programmers were used to with C and C++. SPARK is the most complete subset of Ada, which supports formal verification. Similarly, it is strongly typed and implements memory safety with its own flavor of ownership. While the Rust compiler is undergoing a qualification process[3], SPARK has been used for decades in the safety-critical industry and offers full support for formal methods. In avionics, SPARK is used in Level A software, for which DO-178C recommends formal verification methods. Similarly, ISO 26262, encourages the use of formal methods for the highest Automotive Safety Integrity Levels (ASIL D). Similar arguments can be made regarding IEC 62304 (medical devices) and CENELEC EN 50128 (railways).

Manuscript submitted to ACM

2 BACKGROUND

TODO: explain problem: memory safety in programming.The potential of Rust and SPARK: a brief on their features, relevance in safety-critical applications, and combining.

The significance of safety-critical applications calls for enhanced software security. A primary concern is memory safety[1][12]. Most low-level and safety-critical code is written in C/C++. Despite the implementation of programming guidelines like MISRA C to promote safety and reliability, these guidelines are not a universal solution. The C programming language lacks the internal mechanisms to ensure memory safety - mechanisms such as bounds checking or strong types [16][15]. Furthermore, these guidelines do not preclude the existence of or dependencies on legacy code in both C and C++, which cannot be made safe retroactively[15][8].

The vulnerabilities linked to unsafe language are estimated to be 65-70% of security vulnerabilities[7][2]. Those vulnerabilities' impact is unpredictable, and they are regularly causing significant damage to systems and companies worldwide[11]. Memory safety is of course only a first step, as the ideal goal is the implementation of formal methods [12].

This is where Rust and SPARK come into the picture, as both are languages with robust built-in safety features. Rust, although relatively new, has gained considerable attention and a growing community due to its inherent memory safety features. Most temporal and spatial guarantees are enforced at compile time, with the compiler ensuring that any heap object always has a valid owner. SPARK, created in the early 1980's [4], has an established reputation in the industry with even higher safety guarantees, including support for formal verification. SPARK is based on a mathematical framework and it natively supports formal methods. SPARK utilizes and further reduces Ada's strong typing by suppressing elements difficult to reason about (famously, aliasing). SPARK also implements contract-based programming (a list of conditions listed by the programmer). While contract-based programming is not a silver bullet and does not protect from logical errors, those contracts provide boundaries to check the programs respect their specifications.

However, despite the discussion around memory-safe languages, they are mostly interfaced with C and C++ (because of their long history and the resources already available there) and there has been limited research on effectively combining safe languages together. Such a combination would open the possibility of leveraging their strengths. Indeed, most research focuses on combining safe/unsafe language and limit damages[10][9]. Mergendahl et al. found that the combination of safe and unsafe languages can reopen old avenues of vulnerabilities, and encourages engineers to proceed with the utmost care, and this is a legitimate question when interfacing safe languages.

Both Rust and SPARK are low-level languages with extensive experience, established frameworks, and guidelines for interfacing with C/C++ via Foreign Function Interfaces (FFI). The Rust compiler includes safety mechanisms akin to formal methods in the sense that the compiler does some of the reasoning on memory safety, while SPARK offers a systematic approach to verifying correctness. This allows a level of confidence in correctness that is unachievable by unsafe languages.

However, can both languages be combined in a way to leverage their respective safety guarantees? Such a combination would be an interesting path for safety-critical, systems programming and low-level code in general.

3 INTERFACING RUST AND SPARK

describe of the approach for interfacing Rust and SPARK. summary memory safety, type safety, and ownership in the context of interfacing the two languages.

Both Rust and SPARK are safe languages.

They implement type safety and memory safety, and both have a concept of ownership.

Ownership is particularly relevant for heap-allocated types, as the compiler can determine the size of any `Copy` type at compile time and handle it accordingly.

Those types are copied and put on the stack, and there is no memory allocation to track. Rust ownership guarantees a unique owner for each heap value, and the ability to temporarily transfer ownership, known as borrowing. For definitive transfer, Rust implements the move semantics. If a variable `foo` is moved inside a variable `bar`, `foo` is considered as "moved" and cannot be accessed anymore.

Rust can relax those conditions with interior mutability and unsafe code, but the rules of unsafe code are strict and limited to a specific set of operations (such as dereferencing a raw pointer. Unsafe operations must be handled with great care as they can introduce vulnerabilities.

Rust allows only one mutable reference (`&mut T`) or multiple immutable references (`&T`), and both sets are mutually exclusive.

In SPARK, the concept of ownership is similar: a pointer assignment to another pointer operates as an ownership transfer. If pointer `a` is transferred to pointer `b`, `a` loses permission to the underlying object, except to read - called "observability" in SPARK lingo, following the Concurrent-Read-Exclusive-Write (CREW) model. The move semantics, by locking read permissions, effectively prevents data races.

Despite their differences, Rust and SPARK repose on the same ownership paradigm, implement formal verification (to a certain extent for Rust), and are relevant for the safety-critical industry. Those traits make them desirable for interfacing. Interfacing the languages implies adhering to their assumptions and expectations, and respecting the principles of memory safety, type safety, and ownership. Another factor is understanding the transfer of memory from one language to the other. Also, understanding how both Rust and SPARK manage their pointers/references is necessary. Finally, the interfacing must preserve the safety guarantees of the Rust compiler with respect to &T and &mut T references and, on the SPARK side, ensure that contracts and proofs are upheld.

4 WHAT IS BBQUEUE?

BBQueue is a lockless and thread-safe bipartite circular buffer. BBQueue is designed for communication between two concurrent threads, a common technique in embedded systems or device drivers. It particularly shines when it comes to Direct Memory Access (DMA). DMA allows a process to read from or write to memory directly. It helps save energy by reducing CPU usage and can be significantly more efficient, reportedly up to 20 thousand times faster than simpler protocols. It is important to note that DMA requires continuous memory[13, 14].

BBQueue allows partial commits and releases and can take elements of different sizes. The BBQueue algorithm uses atomic pointers to simulate a circular buffer linked end to end. Atomic variables ensure thread safety and support concurrency, as they prevent two threads from simultaneously accessing the same memory chunk.

There are some differences between the Rust and the SPARK implementations:

The Rust implementation adds two wrappers in the form of a single producer and a single consumer and does not rely on the standard library (`no_std`, as embedded systems need to minimize the size of binaries).

The SPARK implementation does not implement the producer/consumer wrappers and focuses on formal verification. It proves properties such as the validity of the slice (lower and upper limits are within bounds of the buffer), writability of the slice (lower and upper limits are within bounds of the writable part), and correct size of the slice (equal to the size the user requested, or zero).

The SPARK version achieves this by implementing invariants that are checked by SPARK's automatic provers[5, 6].

In both Rust and SPARK implementations, BBQueue is designed as a lock-free FIFO queue. This means its algorithm ensures thread safety by accessing atomic variables in a specific, well-defined order.

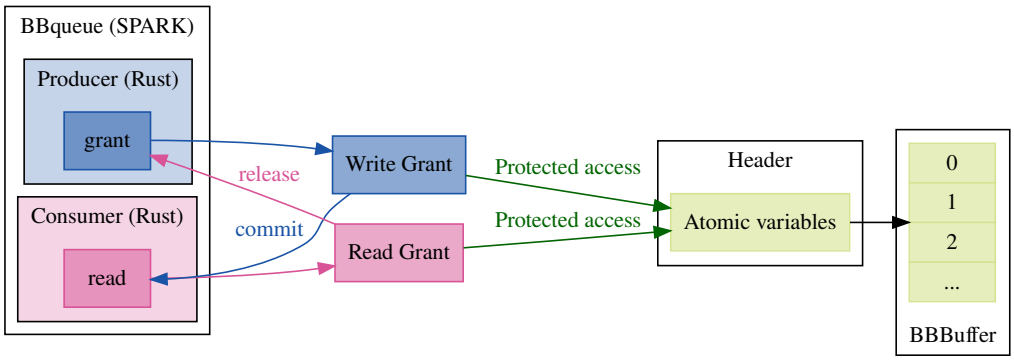

Fig. 1. Communication flow in BBQueue

Table 1. BBQueue structure (Producer-Consumer wrappers only in Rust)

Producer (only in Rust)	Consumer (only in Rust)
1. **Grant** Gives space to write bytes. Space is guaranteed to have single ownership and be continuous in memory.	3. **Read** Gives a mutable slice to read (continuous in memory), but the custom use is an immutable read.
2. **Commit** Makes memory available to read.	4. **Release** Frees the memory and gives the space back.

As per Table 1 and Figure 3, BBQueue's Producer-Consumer model (Rust), the Producer uses a Write Grant (`GrantW`) to provide a space to write bytes and then "commits" to make this memory available to be read. The Consumer issues a Read Grant (`GrantR`) to get a mutable slice to read from and "releases" this slice at the end of usage, to free memory and return the space.

The fact that the BBQueue implementation in both languages follows the same algorithm and guarantees continuous memory makes it an ideal candidate for porting between the two languages.

5 PORTING STACK AND HEAP TYPES BETWEEN RUST AND SPARK

5.1 Stack/Copy types

The BBQueue experiments are based on lessons learned from porting heap and `Copy` types in a series of control studies.

In Rust, a type is marked as `Copy` when it can be duplicated simply by copying its bits. These types have known sizes at compile time, and, when not behind a pointer, are allocated on the stack. The advantages are simpler memory management, as tracking ownership is not an issue anymore. The compiler will simply clean the stack when the stack frame is not necessary anymore.

The BBQueue library does utilize several `Copy` types. An essential aspect of ensuring the correctness of those types is verifying the size and alignment. This necessitated a detailed understanding of how these types (integers, characters, etc.) are represented in each language. The use of correct and corresponding data types is crucial: for example, it is

important to avoid machine-dependent types and numbers of the correct size (in Rust, the name indicated their size, such as u64, and i32). Characters had different encodings depending on the platform. Most stack types are well-behaved, and when finding the appropriate corresponding data structure, passing through FFI usually works. For composite types, it was important that the type was continuous in memory and has a predictable layout, and this is a guarantee offered by BBQueue (continuous buffers). The most reliable approach was to adhere to the C ABI with `#[repr(C)]`, or `Convention => C` on the SPARK side. BBQueue, which passes buffers of continuous data, is important information.

Even if the stack types are relatively straightforward, the programmer must carefully select them and pay close attention to layout and alignment.

5.2 Heap types

The BBQueue experiments are also based on lesson learned from porting heap types, such as `Strings`, from Rust to SPARK, and the effective use of Rust's heap types was integral to the implementation.

While following the guideline to maintain memory safety, type safety, and ownership, managing memory safety was complex. We already covered type safety with `Copy` types, but memory safety and ownership came into the picture for heap types. When interfacing languages through FFI, the compiler guarantees disappear: ensuring memory safety by design becomes impossible. In other words, the memory safety mechanisms of Rust and SPARK, which are built into their respective compilers, become inaccessible. For non-copy types, Rust's borrow checker becomes unusable, which is also the case for SPARK safety mechanisms, including the prover.

For this reason, mutable or immutable references are unsuitable to be passed through the FFI border (Rust cannot reason about references in another language).

As a result, one needs to reason and implement ownership through unique data structures. Ownership needs to be transferred and reimplemented according to the existing paradigms. For example, when passing a read grant to SPARK from Rust, Rust is not the owner of that data structure anymore. It cannot be allowed to go out of scope and no mutable operations on it are permitted on the Rust side. To ensure that the object doesn't go out of scope and get "dropped" by the compiler, Rust needs to be instructed to "forget" that piece of memory. In other words, we need to restrict the guarantees by design to not introduce undefined behavior (such as deleting a memory piece read in another part of the program). From the SPARK side, when receiving a buffer from Rust, SPARK assumes to be the only owner of this buffer, so Rust cannot pass a slice.

Raw pointers to a unique object are the best mechanism to ensure ownership is transferred correctly. This is doubled by the need to follow the "good FFI citizen principles", meaning that the side allocating memory must be the side freeing memory in cross-language data structures.

Many Rust data structures, including the official BBQueue library, are not FFI-safe. This gives the Rust compiler the possibility to rearrange memory for efficiency. Rust heap structures need to be deconstructed to their constituent fields and then repackaged in a new wrapper annotated with `#[repr(C)]`. `#[repr(C)]` makes a data structure FFI-safe as it guarantees a layout similar to the C ABI, in terms of alignment and layout. As a direct consequence, we used a version of the BBQueue library that was guaranteed to be FFI safe, which is a version made for a custom operative system.

To summarize, learning how to pass heap types between SPARK and Rust emphasized several points: the need to select appropriate data types, accurate modeling of the data structures, keeping track of the ownership, assuring similar layout and continuous memory, and reimplement the safety guarantees as the program needs to be made less safe to maintain coherency.

Table 2 summarizes the logic exposed to the user versus the hidden logic of a heap object passed through FFI.

Table 2. FFI Heap object (for example, String)

`public` Heap object	`private` Heap object
1. Unique	1. Fat pointer with data and meta-data (bounds, capacity...)
2. Can be borrowed or moved	2. Hidden from the user
3. Must implement language memory guarantees	3. Memory management relies on the programmer

6 BBQUEUE

The BBQueue experiment was designed to assess how to interface Rust to SPARK, and the other way around. Due to time constraints, interfacing SPARK to Rust is left as future work. BBQueue has relatively complex data structures (headers, grants), and a relatively complex algorithm, which makes it suitable as a research subject. The goal was then to draw conclusions about how to proceed to interface through FFI for those two safe languages.

From Rust to SPARK, the process involved the below learning steps:

(1) The analysis of data structures and algorithm implementations in both Rust and SPARK to ensure they were similar
(2) Testing the official Rust version of BBQueue in both single-threading and multi-threading scenarios to understand its behavior
(3) Study and estimation of the usefulness of the unofficial, FFI-safe, BBQueue implementation
(4) Implementation of FFI from Rust to SPARK

An important note on point 1. The algorithm followed the same logic while there were some differences in the data structures. Those differences do not prevent "observing" the content of the queue, but not modifying it as it would violate the safety guarantees offered by BBQueue. In other words, the differences in implementation prevented us to utilize SPARK API on Rust Grants and headers, but we could still access the data in read-only mode. Throughout these steps, especially during step 4, we focused on maintaining memory safety, type safety, and proper ownership.

In the end, the data structure described in Figure 2 was passed from Rust to SPARK. The left part of the figure shows the header and the buffer, and the right part the additional wrapper send to SPARK. The extra wrapper represents a `Read Grant` with a pointer to the header and the pointer to the buffer, as well as a variable describing the length of read that is not used in SPARK.

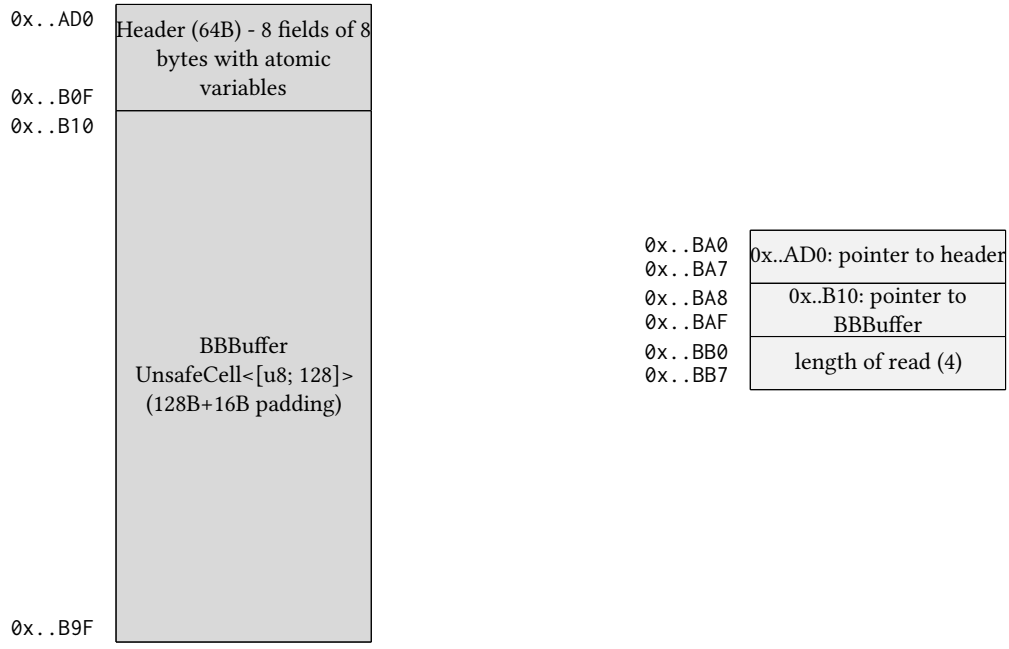

Fig. 2. BBQueue Read grant in memory (not to scale!)
Memory assigned in Rust.
Memory passed to SPARK.

Key observations from the process of porting BBQueue included the following:

(1) Complex structures should be passed by pointer, not by reference. References are specific to the Rust compiler, which allows it to reason about memory safety in a manner akin to formal methods. In this experiment, a part of the program was outside of the Rust compiler's control. Therefore, it was dangerous and unreasonable to rely on references;
(2) Identifying and selecting appropriate types proved to be nontrivial. BBQueue uses a range of types specific to embedded systems.;
(3) SPARK invariants, which were introduced in the SPARK version of this library to formally prove the correctness of the algorithm, were inaccessible when a part of the program was originating from Rust;
(4) Unfortunately the BBQueue's Rust and SPARK implementations differ. Despite careful matching of types and layouts, engineers made different design choices in both implementations;
(5) FFI safety was maintained through the `#[repr(C)]` attribute in Rust, adhering to the C ABI. As expected, this allowed us to correctly receive the type correctly on the SPARK side. Additionally, we tagged the type in SPARK with the convention Ada_pass_by_reference, to ensure the compiler would not try to copy types for which it knew the sizes.

Using the library in a different language necessitated a consistent layout across the FFI border, a pointer to both grant and header content (as a reminder, the grant is the only data structure authorized to touch the header, which in turn is authorized to touch the buffer: not respecting those design rules do not guarantee thread safety), making sure all variables were atomic, removing possible undefined behavior (such as unwinding through the FFI border), and the

exact prescribed order of interaction with atomic variables. We could ensure all the points except the last ones, since headers and methods were different: although data structure reading and passing from the queue was possible, API design differences between Rust and SPARK complicated the direct use of SPARK methods on Rust data structures.

The passing of the BBQueue data structure provides insights into FFI safety and memory management between Rust and SPARK. BBQueue is composed of stack and heap types. Researching stack types required reading up documentation while heap types require more careful memory management, such as following good FFI citizen principles, and properly transferring ownership. The experiment also highlighted the significance of good design and meticulous error-checking. Despite these complexities and differences, the experiment proved to be memory safe when monitored with tools like Valgrind. We believe that it is highly feasible to automate this process.

6.1 Strategy on error

BBQueue target-domain is embedded systems, where software malfunction or exception can have serious real-world implications, such as hardware damage. As such, robust error-handling mechanisms are a necessity. The usual strategy is to abort and reset the system, even if the cost of introducing redundancy is high in terms of resources and delays.

In embedded systems, there is often no possibility to carefully monitor the internal state of the applications and catch errors or exceptions. In our particular case, unwinding after an error through the FFI barrier will lead to undefined behavior. Furthermore, recovering from an error introduces state extension that we must avoid, as there is no guarantee that a post-recovery state did not introduce more subtle errors.

In BBQueue's case, if a panic occurred while writing or reading to the buffer, trying to recover might leave the queue in an inconsistent state - the atomicity of operations might not be respected. Trying to recover and continue can lead to further errors or corrupt data.

By choosing to abort instead of unwinding, we effectively minimize the risk of propagating an inconsistent or erroneous state as described in Figure 3.

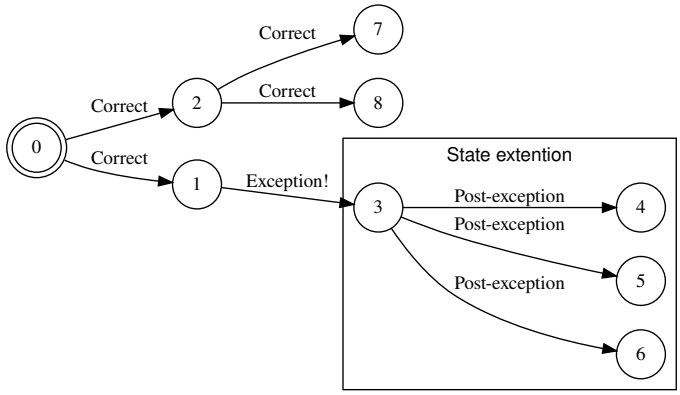

Fig. 3. Faulty state extension

7 SUMMARY

This paper partly summarizes the results of a thesis on porting the BBQueue library from Rust to SPARK, building on a series of control studies for stack and heap types. BBQueue is a bipartite circular buffer that is `no-std`, lock-free and

thread-safe through atomic variables. It is implemented in both Rust and SPARK, optimized for DMA and designed for concurrency. Additionally, the SPARK implementation add formal verification of properties like slice validity.

BBQueue's implements the same algorithm in both Rust and SPARK and roughly the same data structures. This makes this library interesting for cross-language portability testing. Portability was achieved through careful management of stack types and heap types, paying special attention to size and alignment, and ensuring memory safety for the latest. Through the experiment, we also ensure memory continuity, and used pointers to transfer ownership, and respected "Good FFI citizen" principles when it was time to clean the memory.

The experiment provided important insights into FFI safety and memory management between Rust and SPARK. Those insights include respecting the language semantic expectations, monitoring ownership and ensuring continuous memory.

ACKNOWLEDGMENTS

I would like to thank Cyrille Artho, who supervised my thesis, for giving me the chance to present my results in a reputable journal even if I am terrified right now. (yes it will be rewritten) I also want to thank my boss Florian Gilcher and Yannick Moy at AdaCore, as well as James Munns -who has always been an inspiration for me- and Fabien Chouteau.

REFERENCES

[1] [n. d.]. CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[2] 2022. NSA Releases Guidance on How to Protect Against Software Memory Safety Issues. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues

[3] 2023. Ferrocene Language Specification. https://github.com/ferrocene/specification original-date: 2022-05-11T07:23:37Z.

[4] Bernard Carré and Jonathan Garnsworthy. 1990. SPARK an annotated Ada subset for safety-critical programming. In *Proceedings of the conference on TRI-ADA '90 (TRI-Ada '90)*. Association for Computing Machinery, New York, NY, USA, 392–402. https://doi.org/10.1145/255471.255563

[5] Fabien Chouteau. [n. d.]. BBQueue SPARK. https://github.com/Fabien-Chouteau/bbqueue-spark

[6] Fabien Chouteau. 2021. From Rust to SPARK: Formally Proven Bip-Buffers | The AdaCore Blog. https://blog.adacore.com/from-rust-to-spark-formally-proven-bip-buffers

[7] Alex Gaynor. 2020. What science can tell us about C and C++'s security · Alex Gaynor. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/

[8] Siliang Li. 2014. *Improving Quality of Soft ware with Foreign Function Interfaces using Static Analysis.* Doctoral dissertation. Lehigh University. https://www.semanticscholar.org/paper/Improving-Quality-of-Soft-ware-with-Foreign-using-Preserve-Li/8d0b6db3858946c27657567d42f684a32d34e4f3

[9] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C. S. Lui. 2022. Detecting Cross-language Memory Management Issues in Rust. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III*. Springer-Verlag, Berlin, Heidelberg, 680–700. https://doi.org/10.1007/978-3-031-17143-7

[10] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-Language Attacks. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2022.24078

[11] Yannick Moy. 2019. Proof of Pointer Programs with Ownership in SPARK. https://people.cs.kuleuven.be/~dirk.craeynest/ada-belgium/events/19/190202-fosdem/13-ada-pointers.pdf

[12] Yannick Moy and Anthony Aiello. 2020. When testing is not enough. Software complexity drives technology Leaders to Adopt Formal Methods. https://issuu.com/rtcgroup/docs/cots-2007-july-web/s/10872490

[13] James Munns. 2019. The design and implementation of a lock-free ring-buffer with contiguous reservations - Ferrous Systems. https://ferrous-systems.com/blog/lock-free-ring-buffer/

[14] James Munns. 2022. BBQueue. https://github.com/jamesmunns/bbqueue original-date: 2018-12-31T00:26:47Z.

[15] Oliver Scherer. 2021. Engineering of Reliable and Secure Software via Customizable Integrated Compilation Systems. https://doi.org/10.5445/IR/1000134165

[16] L. Szekeres, M. Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, 48–62. https://doi.org/10.1109/SP.2013.13

A BBQUEUE IN SPARK AND RUST

- SPARK BBQueue https://github.com/Fabien-Chouteau/bbqueue-spark.
- Official Rust BBQueue https://github.com/jamesmunns/bbqueue
- FFI safe Rust BBQueue https://github.com/tosc-rs/mnemos/tree/main/source/abi/src/bbqueue_ipc.

B CONTROL EXPERIMENTS

- Control studies and BBqueue experiments: https://github.com/Dajamante/ada_rust_programs.
- Statistic tool : https://github.com/Dajamante/stat_ada_rust_code.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009