Encapsulated Functions: Fortifying Rust’s FFI in Embedded Systems

Leon Schuermann
lschuermann@princeton.edu
Princeton University
Princeton, New Jersey, USA

Arun Thomas
arun@zerorisc.com
zeroRISC Inc.
Somerville, Massachusetts, USA

Amit Levy
aalevy@princeton.edu
Princeton University
Princeton, New Jersey, USA

Abstract

Memory-safe languages like Rust are increasingly popular for systems development. Nonetheless, practical systems must interact with code written in memory-unsafe languages. This is especially true in security and safety-critical embedded systems, where subsystems such as cryptographic implementations are subject to industrial and governmental certification requirements. Direct interactions with such libraries, however, expose memory-safe languages to significant risks: Any single bug in either the foreign code or the cross-language interactions may arbitrarily violate the memory safety of the wrapping language.

We present Encapsulated Functions, a framework for safely invoking untrusted code in a memory-safe system with minimal overheads. Encapsulated Functions combines hardware-based memory protection mechanisms with a set of Rust type abstractions to facilitate safe interactions with untrusted and unmodified third-party libraries.

CCS Concepts: • Computer systems organization → Embedded software; • Software and its engineering → Software safety; • Security and privacy → Operating systems security.

Keywords: Rust, memory safety, foreign function interface, memory protection

ACM Reference Format:

1 Introduction

Systems are increasingly built using modern, memory-safe languages such as Rust, Go or Swift. These languages can aid developers in writing correct software and prevent entire classes of bugs due to their design. However, practical systems must often integrate existing libraries, such as cryptographic implementations written in non-memory-safe languages or provided in the form of binary blobs. Unfortunately, bugs in those libraries can arbitrarily violate memory safety of the wrapping language. Even internally correct libraries may break safety guarantees in subtle ways due to differing cross-language semantics. We argue that memory-safe languages should be able to invoke unsafe, unmodified third-party libraries safely, and with minimal overhead.

These problems are particularly pronounced in the context of secure and safety-critical embedded systems, where certain subsystems, such as cryptography, timing-critical control loops, and wireless communication, require industrial or governmental certification. Rewriting existing implementations in a memory-safe language is often not feasible: Apart from development and re-certification overheads, software implemented in industry-standard programming languages benefits from a mature ecosystem of toolchains, verification infrastructure, established best practices, and alignment to industrial certification processes. For example, the flight control software of the Airbus A380 is verified to be sufficiently bounded on its worst-case execution time (WCET), an important functional safety property, using analysis techniques currently only available for C [1, 2]. Similarly, industry standards are written for C-based implementations.

Despite these external constraints enforcing the use of non-memory safe languages like C, evidence suggests that memory-safe languages make it easier to write correct code. According to Microsoft, around 70% of security vulnerabilities addressed in Microsoft products are caused by memory safety issues [14]. Even within the space of cryptographic libraries, an empirical study attributes 37.2% of vulnerabilities to memory safety issues [5]. A report by Google indicates that with the usage of Rust as a memory-safe language in Android 13 the frequency of these issues has dropped significantly—from 76% to 35% [23]. The absence of memory-safety issues is required to maintain a system’s functional safety and security, both especially important properties for many embedded systems.

In fact, Rust is uniquely suited for the requirements of these constrained environments: Its compilation to native
machine code, static memory management, and use of zero-cost abstractions promises to support systems with strict resource and timing constraints. Moreover, its use of a static and strong type system alongside an ownership-based memory management model establishes both spatial (e.g., preventing out-of-bounds accesses) and temporal (e.g., accessing reclaimed memory) memory safety. Projects such as the Tock embedded operating system demonstrate the feasibility of Rust in severely resource-constrained devices [10].

A reasonable step towards memory-safety in these systems may be to take advantage of Rust’s safety guarantees wherever possible, and integrate existing legacy or certified implementations of specific subsystems. Unfortunately, it is hard to correctly interact with foreign, unsafe code from a memory-safe language like Rust; any single memory-safety issue in the foreign code can break Rust’s memory safety arbitrarily. Moreover, even if foreign code operates correctly, the interactions between such code and Rust can still wreak havoc: Rust has an extensive set of safety requirements that must be maintained, which forbid many otherwise legal behaviors in foreign languages. Notably, these requirements are not just limited to spatial and temporal safety. Rust further places extensive restrictions on valid values. For example, a bool type must have a value of either 0 or 1 [21]. Simply having a bool-reference to any other value in scope (without dereferencing it) is undefined behavior. Other such violations include a null-reference, or an enum-discriminant not included in its type-definition.

To facilitate safe cross-language interactions, this paper presents Encapsulated Functions: a framework to execute untrusted foreign code in the context of a memory-safe system. Making Encapsulated Functions special is its ability to work in severely resource-constrained embedded systems, such as microcontrollers, and it being able to safely execute unmodified binary code. These properties make it uniquely suited to foster adoption of safe programming languages in safety and security critical embedded devices, while allowing users to reuse existing industrially-certified C libraries.

Throughout this paper we illustrate the two interoperating mechanisms composing Encapsulated Functions: an efficient hardware-based memory isolation mechanism, using hardware features commonly found in modern microcontroller systems, and a set of type-level abstractions to interact with untrusted foreign memory. We further demonstrate the applicability of our system by integrating it with the Tock embedded OS and implementing an HMAC-based one-time password generator (HOTP) on the OpenTitan root-of-trust (RoT) platform, using its C-based CryptoLib cryptography library.

2 Background
Throughout this section we provide an overview of existing hardware- and software-based isolation mechanisms.

2.1 Hardware-Mediated Process Isolation
Operating system processes are well-suited to isolate untrusted code. As a process operates in its own, isolated address space, memory-safety violations are contained within this process. Furthermore, a process does not have unlimited control over the execution of other processes. Instead, it can signal events and share data with other processes through OS-mediated channels, such as UNIX domain sockets.

Unfortunately, such IPC-based isolation mechanisms induce significant overheads when used for individual function calls. As the isolated function executes in a different address space, its parameters must be serialized and transferred into the remote process. Upon return from the function, the remote process must further serialize its result and transfer it back to the original process. Every such transition induces a full context switch which saves the currently executing process’ state, performs scheduling decisions in the kernel, and finally restores the to be scheduled process’ state. The required extraneous memory allocations and context switch overheads exceed resources available in many constrained embedded systems. Calls into other dynamically-scheduled units of execution also introduce unpredictable delays.

2.2 Software-Based Fault Isolation
In contrast to hardware-based isolation techniques, which execute untrusted code and fault upon executing certain dangerous interactions (such as memory writes), Software-Based Fault Isolation (SFI) ensures that such interactions are never issued in the first place [24]. This can be achieved through an ahead-of-execution analysis of the untrusted binary to ensure that all potentially dangerous instructions are limited to a given protection domain, modification the untrusted binary to introduce runtime checks ahead of any potentially dangerous instructions, or a combination of both.

While SFI does not require hardware isolation support and avoids significant context-switch overheads to switch between trusted and untrusted code, it is not able to isolate arbitrary untrusted and unmodified binaries. Still, its tightly defined execution model provides inspirations for this work; specifically we mirror Native Client’s clear separation of and switches between trusted and untrusted contexts through the trampoline and springboard mechanisms [25].

3 Design
In this section we present Encapsulated Functions, a framework to execute untrusted foreign code in the context of a single-threaded memory-safe system. Encapsulated Functions is composed of two interoperating mechanisms—lightweight context switches, an implementation of function calls engaging hardware-based memory protection, and a set of type-level abstractions for safely interacting with untrusted foreign memory, which we present separately.
writeable memory regions do not overlap with any Rust allocations. Separating memory accessible to Rust and foreign code trivially maintains Rust’s memory-safety, but is not particularly useful on its own; Section 3.2 presents considerations for interacting with this memory.

To enforce these memory protection rules, Encapsulated Functions must switch to a lower privilege level when executing untrusted functions. For this, we use the same hardware mechanisms as employed for context switches to processes. However, our isolation granularity of function invocations, compared to scheduled and concurrent execution units such as OS processes, allow us to optimize this switching process significantly; we refer to this mechanism as function calls through lightweight context switches. Importantly, this mechanism maintains the same synchronous execution model as regular function calls, avoiding overheads of saving state as required when invoking an asynchronous task.

To invoke a function, privileged Rust code (mirroring the SFI trampoline mechanism) prepares the function’s execution environment by placing function call arguments in their corresponding registers and onto a newly allocated separate stack, located within a foreign-code accessible, writeable memory region. This process implements and follows the untrusted binary’s ABI. To strengthen guarantees concerning confidentiality, the privileged code may wish to further clear other unused registers. To allow foreign code to pass control back to the privileged Rust code, we use an analog to the springboard approach from SFI and set the return address register to a well-known inaccessible or privileged and immutable instruction. This ensures that an attempted function return by the untrusted code will trigger a context switch back into the privileged execution context. Finally, the function is executed by switching to a lower privilege mode, enforcing hardware memory protection, and setting the program counter (PC) accordingly. Upon return, the privileged code must assume all registers to be clobbered. Such a function call largely avoids the overhead of a full context switch, which would include scheduling decisions, and saving and restoring a full process execution environment.

The foreign code may choose to never return to the springboard instruction, violating liveness guarantees. The system may optionally use a timer-interrupt to switch back to privileged code in the case that foreign code does not return.

### 3.2 Unsafe Cross-Language Interactions

The lightweight context switch mechanism of the previous section is sufficient to fully isolate untrusted code in the context of a memory-safe system. However, such an overly constrained system has limited utility; Rust code will need to interact with foreign code by passing data to functions and interpreting returned results. These interactions, however, introduce a yet another way to violate memory safety: While all objects in foreign memory are sufficiently constrained concerning spatial- and temporal-safety (memory accesses...
by foreign code are limited to the time this function is executing and do not overlap any Rust allocations, they do not necessarily conform to the safety constraints as illustrated in Section 1; specifically they may not be valid values.

Furthermore, depending on how memory is divided between Rust and foreign code, this mechanism may also endanger Rust’s guarantees around temporal memory safety. For example, if a system were to reuse the Encapsulated Functions’ stack memory region while the foreign code is not running, references into this memory may then potentially alias Rust allocations and would thus be able to cause undefined behavior.

One possible approach to overcome this limitation are pseudo-pointers (for instance, as used by Galeed [16]). Encapsulated Functions would pass such pointers to foreign code, which can provide them back to trusted Rust accessor functions that provide access to a limited set of Rust allocations. Unfortunately, this model has multiple issues in the context of Encapsulated Functions. For instance, each such memory access would incur a lightweight context switch—a significant overhead compared to other memory protection mechanisms like Intel MPK. Furthermore, usage of accessor methods requires modification of foreign code, which would be in conflict with our requirements. Instead, we develop a set of type-abstractions which allow both Rust and foreign code to directly access shared memory, while maintaining Rust’s safety requirements.

3.2.1 Rust Primitives to Access Foreign Memory. To eliminate the aforementioned safety issues when interacting with foreign code, we propose a set of type-level abstractions for Rust that facilitate safe interactions with foreign memory. Our abstractions place an explicit focus on soundness and runtime-efficiency. In this section we introduce select Rust language constructs useful for designing these abstractions.

Raw Pointers act as the foundation of our type abstractions. Rust’s raw pointers (such as *mut u8) are not subject to the same constraints that Rust values or references are. Retaining a mis-aligned, dangling or inaccessible pointer to some invalid value is safe, so long as this pointer is never dereferenced or cast into a Rust reference. For these reasons, dereferencing Rust’s raw pointers is an unsafe operation.

Valid Values. To support safely referencing memory which may not contain a valid instance of some type T, Rust provides the MaybeUninit<T> type [20]. This type can be used to represent a memory allocation that has the same size, alignment and ABI as a type T, but does not support safely dereferencing it. Still, as this type is guaranteed to be well-aligned and contained in accessible memory, writing to it is a safe operation. These semantics are suitable to wrap foreign memory, which cannot be guaranteed to always conform to Rust’s requirements on valid values [21].

Mutable Aliasing. Additionally, Rust’s type system distinguishes between shared (immutable) references, for which it assumes that referenced memory cannot be mutated, or unique (mutable) references, which reference memory that can only be mutated through this reference. These restrictions prevent mutable aliasing, where a given memory location is mutated and referenced by at least one other Rust reference simultaneously. However, foreign code does not have to adhere to these constraints: In C, two pointers can generally coexist while referencing (partially) overlapping allocations. Rust provides an escape hatch to circumvent these aliasing restrictions, namely the UnsafeCell<T> type. A shared (immutable) reference to this type is allowed to point to data that is being modified [19].

As a consequence, a raw pointer to some type T can be converted into a shared reference of some UnsafeCell<T> or MaybeUninit<T>, assuming the pointer is well-aligned, fully contained in mutably accessible memory, and not aliasing other Rust allocations not contained in an UnsafeCell or of a different type. A reference to this composite type can be used to safely write to its backing memory, even if it were to alias another UnsafeCell or MaybeUninit reference over a different type U. Still, such a reference cannot be safely dereferenced, as it is not guaranteed to contain a valid instance of type T.

Finally, for Rust to safely access such references into foreign memory, their memory contents must be validated to conform to Rust’s requirements on valid values. For instance, a bool type is required to have a numeric value in {0, 1} [21]. The memory must stay consistent with these requirements for the entire duration that a dereferencable (non-MaybeUninit) reference is in scope. This cannot be guaranteed in the case of concurrent execution of Rust and foreign code. Thus, we place a single-threaded restriction on the Encapsulated Functions execution model: A foreign binary is always executed within a single thread, and only a single Rust thread may execute foreign functions or access foreign memory at any given time.

Nonetheless, the aforementioned validity of values can be violated from safe Rust even without handing over control to the foreign code. Because C pointers may arbitrarily alias memory allocations, a Rust-issued write through one reference may modify the content of another reference. For instance, a C binary may provide Rust with two byte-sized pointers to the same memory, one represented as a bool and another as a u8. While the numerical value 2 is a valid member of the u8 type, it is not valid for the bool type. Thus, validated references can only be assumed to remain valid until either control is handed over to the untrusted foreign code, or foreign memory is modified from within Rust.

---

1C does place restrictions around aliasing across incompatible types.
2In Rust, this can be achieved by marking the Encapsulated Functions runtime type as Send (allowed to cross thread-boundaries), but not Sync (not safe to share between threads).
3.2.2 Safe Type-Abstractions for Foreign Memory.

We can use the presented Rust primitives to establish a set of type-level abstractions which safely interact with foreign memory. They integrate with an allocator that can allocate objects within foreign memory and track whether a given allocation is contained in said memory. The type-abstracts further interact with the mechanism to hand over control to foreign code, in order to invalidate any validated references across such invocations. Our type-abstracts follow the typestate programming paradigm, which encodes changes in the validation state of a foreign memory reference as a transition between types, which in turn expose a set of safe methods applicable in that state [3]. Notably, we further utilize Rust’s lifetimes to impose restrictions on the duration of validity of certain type representations. Throughout this section we present types applicable to mutably accessible memory (i.e. referencing RAM); our implementation provides an analog set of types for immutable memory (e.g., memory-mapped flash). We illustrate the interactions between our type-abstracts in Figure 2.

As illustrated in Section 3.2.1, Rust’s raw pointers form the foundation of our type abstractions. The `EFPtr<T>` type wraps a Rust raw pointer and can be passed across FFI boundaries. It exposes convenience methods useful for working with Encapsulated Functions and prevents type confusion, but can be safely constructed from arbitrary raw pointers.

If an `EFPtr<T>` points to some well-aligned type `T` wholly contained in mutably accessible foreign memory, it can be converted into an `EFMutRef<T>`. This type allows writing through the reference, but does not support dereferencing its memory. It thus represents a useful intermediate type: Even across invocations of untrusted code or modifications of foreign memory, an `EFMutRef` is still guaranteed to be well-aligned and fully contained in foreign memory, avoiding re-validation of these properties. The `EFMutRef<T>` type internally wraps an `&' alloc UnsafeCell<MaybeUninit<T>>` shared reference, and is bound to an allocation scope `&' alloc AllocScope`. To ensure that references into foreign memory never outlive the foreign memory reservation itself, we introduce allocation scope marker types. These types serve as a proxy to bind references into foreign memory to Rust’s rules around references and lifetimes. For instance, the constructor of an `EFMutRef` borrows a shared reference to an instance of the `AllocScope` marker type, for the entire lifetime of the resulting `EFMutRef< alloc, T>`. To implement this check, we require that types provide an unsafe `validate` method which, given a well-aligned and accessible raw pointer, must establish whether the referenced value is a valid member of the given type. Notably, for types where every possible memory state represents a valid instance of said type, this method may be implemented as a no-op[^5]. The `EFMutVal<T>` type is further bound to an access scope `&' access, T>`, which must ensure that only a single instance of this type can exist at any given time. Instantiating an `EFMutVal< alloc, T>` type is further bound to an access scope `&' access, T>`, conceptually similar to allocation scopes. An `AccessScope` is issued by the facility to execute untrusted code, which must ensure that only a single instance of this type can exist at any given time. Instantiating an `EFMutVal< alloc, T>` type is further bound to an access scope `&' access, T>`.

[^5]: `MaybeUninit` nonetheless requires such allocations to be initialized; the compiler may otherwise determine them to be undefined (undefined / poison) [18, 20]. While in practice the compiler is unable to track this taint through invocations of foreign code, to comply with this requirement Encapsulated Functions requires explicit initialization of all foreign memory.

Figure 2. Typestate-transitions between the `EFPtr`, `EFMutRef`, and `EFMutVal` reference types for foreign memory. Pointers can be converted into more capable reference types subject to runtime checks. Reference types are bound to compile-time enforced allocation- and access-scoops, which expire on changes to the foreign-code accessible memory regions or memory writes and foreign code execution, respectively.
To evaluate the applicability and performance of Encapsulated Functions, we eliminate dangling references. Validated preemptively scheduled, unprivileged and language-agnostic and uses hardware memory protection mechanisms to isolate processes. While Tock supports both ARM Cortex-M and 32 bit RISC-V platforms, our current implementation targets only RISC-V systems and depends on a Physical Memory Protection (PMP) unit.

For our evaluation, we choose to target the OpenTitan open-source silicon root-of-trust (RoT) system synthesized for the ChipWhisperer CW310 FPGA board. OpenTitan is a mature RISC-V based RoT platform integrating a set of hardened cryptography primitives. It encompasses CryptoLib, a C-library to interact with these primitives and provide high-level cryptography interfaces. We demonstrate Encapsulated Functions by integrating the HMAC subsystem of CryptoLib into the Tock kernel. Encapsulated Functions has also been verified to work on other RISC-V targets supported by Tock, without any target-specific modifications.

Our implementation requires a change to Tock’s RISC-V trap handler implementation, in order to remove hard-coded assumptions about interactions with user-mode code. No other modifications to Tock’s core kernel infrastructure or architecture support are required. The Encapsulated Functions runtime and type abstractions are implemented as a Rust library to interact with these primitives and provide high-level cryptography interfaces. We demonstrate Encapsulated Functions by integrating the HMAC subsystem of CryptoLib into the Tock kernel. Encapsulated Functions has also been verified to work on other RISC-V targets supported by Tock, without any target-specific modifications.

When switching to a process or invoking a foreign function, the system must configure the PMP to enforce an appropriate set of memory access rules. Once configured, repeated executions of the same process or invocations of the same foreign binary do not require re-configuration of the PMP. The overhead required by PMP configuration is largely independent of the number of memory regions configured and, using an optimized configuration routine, takes approx. 240 instructions in the Tock operating system. Once the PMP is configured, invoking a foreign function through a lightweight context switch induces an overhead of approx. 120 instructions. This includes setting of the foreign code’s stack, re-configuring the CPU for user-mode, and switching execution to the function code (64 instructions), as well as 55 instructions to handle a system trap, interpret the context switch reason, extract the function’s return values and restoring other machine-state on the return path. In comparison, an end-to-end context switch to a Tock process requires approx. 530 instructions on a RISC-V RV32IMC system. This includes scheduling the process, restoring the userspace register file, re-configuring the CPU for user-mode, as well as saving the user-space context and machine-state configuration on the return path; it excludes any kernel work, PMP configuration or debug information tracking. Table 1 summarizes these measurements.

To perform an HMAC calculation, both the key and data parameters need to be located in CryptoLib-accessible memory, which we achieve by copying them into appropriately sized allocations on the isolated Encapsulated Functions stack. These operations are conducted through our type abstractions illustrated in Section 3.2. Notably, due to the
### Table 1. End-to-end overheads induced by lightweight context switches and context switches to Tock processes, measured on a RISC-V RV32IMC system. In both cases, PMP re-configuration constitutes a substantial overhead. If the PMP is already configured for a given process or foreign binary, it does not have to be re-configured.

<table>
<thead>
<tr>
<th>PMP Pre-configured</th>
<th>Lightweight Context Switch</th>
<th>Tock Process Context Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>120 instr.</td>
<td>530 instr.</td>
</tr>
<tr>
<td>✗</td>
<td>360 instr.</td>
<td>770 instr.</td>
</tr>
</tbody>
</table>

23% vs. 47%

fine-grained typestate information contained in these reference types, in many cases the compiler is able to optimize these abstractions into direct memory accesses. This holds as long as memory accesses are not made through indirect pointers located in foreign memory, and as long as read operations are limited to types where every memory state constitutes a valid instance of that type.

5 Related Work

There is a significant body of research exploring safety around cross-language interactions and isolation techniques. For instance, [13] establishes Cross-Language Attacks, an entirely new set of attack vectors caused by cross-language interactions, such as between Rust and C. The Rust community is addressing these issues by drafting documentation and guidelines around unsafe code and FFI usage [21, 22].

Contributions such as Galeed, XRust, TRust, PKRU-Safe, and SDRaD-FFI utilize heap isolation techniques to protect against a subset of these safety issues [4, 6, 7, 12, 16]. Whereas Galeed, PKRU-Safe, and SDRaD-FFI use a hardware-based memory protection mechanism to isolate Rust from foreign code, TRust additionally employs SFI techniques to confine unsafe Rust code to an untrusted memory domain. By modifying foreign code to use pseudo pointers, Galeed supports interactions with Rust objects. Similar to Encapsulated Functions, XRust and TRust and Intra-Unikernel Isolation [17] confine unsafe or foreign code to accessing limited memory respectively. Sandcrust uses IPC to automatically isolate unsafe Rust code within a separate process [9]. Both Sandcrust and SDRaD-FFI utilize serialization to convey (updated) variables between Rust and untrusted code.

Works like RLBox and RLBox-Rust combine a set of type-system abstractions with a WebAssembly sandbox to isolate untrusted code from C++ and Rust [15, 26]. Furthermore, progress on static analysis and SFI promises to alleviate overheads associated with hardware-based isolation techniques. For example, [8] uses structural information of code within representations such as WebAssembly to reduce the number of runtime checks required with SFI FFIChecker employs static analysis techniques on the generated LLVM IR to identify potential cross-language memory management issues [11].

Notably, most existing work around cross-language interactions with Rust focuses on maintaining its spatial and temporal memory safety properties; few contributions address more subtle safety issues around valid values.

6 Conclusion

In this paper we present Encapsulated Functions, a framework for integrating untrusted and unmodified code into a memory-safe system. Through our prototype implementation we demonstrate the feasibility of using hardware-based memory protection mechanisms present on modern microcontrollers, along with a set of safe type-abstractions, to facilitate safe interactions with foreign language code while incurring minimal overheads. We further optimize the switch to a hardware-isolated execution environment through our lightweight context switches mechanism.

Compared to prior work in this field, Encapsulated Functions is a particularly lightweight cross-language isolation mechanism. It does not require a heap allocator, avoids spurious memory allocations as required for IPC-based approaches, and performs runtime validation of type and memory safety lazily, solely for memory that is accessed by the memory-safe language. Both the memory-safe language and foreign code can access shared memory directly, without any runtime indirection to bypass hardware protection mechanisms.

We implement Encapsulated Functions using the RISC-V Physical Memory Protection (PMP) subsystem. Our type abstractions for maintaining cross-language memory safety, however, are independent of the memory protection mechanism used. We believe that these type abstractions can be equivalently applied to similar mechanisms for other architectures (e.g., the ARM Cortex-M MPU), more complex Memory Management Units of general-purpose CPUs, and userspace-configurable hardware memory protection implementations such as Intel MPK. We hope to explore these additional application domains in future work.

This paper validates the design of Encapsulated Functions and performs a limited set of performance evaluations by integrating the OpenTitan CryptoLib HMAC subsystem into the Tock embedded OS kernel. By demonstrating support for a wider range of software libraries and porting Encapsulated Functions to other operating systems we hope to further evaluate the expressiveness of our solution. This will also enable an extensive performance evaluation and allow us to revisit certain design restrictions, such as Encapsulated Functions’ single-threaded execution model.

Finally, we hope that this work will foster adoption of memory-safe languages even in severely constrained embedded systems, subject to industrial standards and certification processes.
References


