Isolation in Rust: What is Missing?

Anton Burtsev
University of California, Irvine
Irvine, California, USA

Dan Appel
University of California, Irvine
Irvine, California, USA

David Detweiler
University of California, Irvine
Irvine, California, USA

Tianjiao Huang
University of California, Irvine
Irvine, California, USA

Zhaofeng Li
University of California, Irvine
Irvine, California, USA

Vikram Narayanan
University of California, Irvine
Irvine, California, USA

Gerd Zellweger
VMware Research
Palo Alto, California, USA

Abstract

Rust is the first *practical* programming language that has the potential to provide fine-grained isolation of untrusted computations at the language level. A combination of zero-overhead safety, i.e., safety without a managed runtime and garbage collection, and a unique ownership discipline enable isolation in systems with tight performance budgets, e.g., databases, network processing frameworks, browsers, and even operating system kernels.

Unfortunately, Rust was not designed with isolation in mind. Today, implementing isolation in Rust is possible but requires complex, ad hoc, and arguably error-prone mechanisms to enforce it outside of the language. We examine several recent systems that implement isolation in Rust but struggle with the shortcomings of the language. As a result of our analysis we identify a collection of mechanisms that can enable isolation as a first class citizen in the Rust ecosystem and suggest directions for implementing them.

1 Introduction

Isolation of subsystems was identified as a critical mechanism for improving reliability and security of operating systems at least as early as in the first Multics report in 1977 [9]. Over the years, providing efficient isolation has become an important topic for many other large software systems: browser plugins [55, 68], user-defined database functions [17, 57], network functions [5, 33, 36, 45, 53, 56], device drivers [25, 27, 58], storage systems [15], kernel modules [13, 24, 26, 29–32], and more.

Nevertheless, despite decades of research, fine-grained isolation remained impractical. Today, the main mechanisms that are used to enforce isolation boundaries—hardware isolation primitives, software fault isolation (SFI), and language safety—impose prohibitive overheads in systems that are designed to keep up with the speed of modern I/O interfaces. Commodity hardware isolation primitives (e.g., page-tables) introduce an overhead of several hundred cycles when performing a function call invocation across an isolation boundary [1]. Such overhead is not acceptable for isolation of systems that spend only a few hundred cycles per request, e.g., modern network [18, 21] and disk [37] device drivers. Similarly, SFI [2, 16, 23, 44, 55, 68], which can enforce segment-like boundaries around the isolated code, results in 2x higher CPU utilization, even with legacy 1 Gbps network interfaces. The main cause for the high CPU utilization are added access checks for regions of memory exchanged across the isolation boundaries [44]. Finally, despite many performance advances in garbage collection and just-in-time compilation, the overheads of safe languages remain high for systems designed to operate at line rate: a recent study that implements a DPDK-style device driver in ten different languages shows that even the fastest managed languages, Go and C#, are 36-42% slower than C with realistic packet batch sizes [22].

The development of Rust, a systems programming language that offers memory and type safety without garbage collection [38], allows us to revisit the possibility of practical, low-overhead isolation. Rust enforces safety through a restricted ownership model, allowing only one unique reference to each live object in memory. The compiler can statically track the lifetime of each object and deallocate it without a garbage collector. The runtime overhead of the language is limited to bounds checking, which is in most cases concealed by modern out-of-order CPUs that predict and execute the correct path around the check [22].

Several unique properties of Rust lower the overhead of communication across isolation boundaries and it has been shown how they can be used to provide strong isolation guarantees, e.g., *fault isolation* [48]. Firstly, like any safe language, Rust can enforce isolation with only the overhead of

PLOS '21, October 25, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8707-1/21/10.
https://doi.org/10.1145/3477113.3487272

a function call. In a safe language an invocation between isolated subsystems can continue on the same stack (safety ensures isolation of objects on the stack) and does not require saving and restoring general and extended registers (calling conventions save and restore registers between the caller and callee, and exception handling, i.e., *unwind*, mechanisms allow recovery from a fault in an untrusted subsystem). This significantly lowers the cost of a cross-subsystem invocation compared to hardware and SFI-based solutions in which saving and restoring of general and extended registers and switching the stack requires 111-406 cycles [42]. Second, Rust ownership discipline enables zero-copy communication across isolated subsystems by enforcing *single ownership* of objects passed in cross-subsystem invocations [7, 48, 51]. Hardware or language-based isolation and SFI systems require a copy of all objects passed across isolation boundaries to enforce that the caller can no longer access objects passed to the callee. Finally, the Rust ownership discipline provides a foundation for isolation of object spaces [35], which is key for fault isolation and clean termination of isolated subsystems [35, 48].

Unfortunately, Rust was not designed with support for isolation in mind. Like any safe language, Rust provides mechanisms to control access to the state of the program at module and class boundaries by specifying fields of individual objects as public or private. Isolated parts of the program have access to the state transitively reachable through public global variables and explicitly passed arguments. Control over references and communication channels allows isolating the state of the program on function and module boundaries, enforcing confidentiality and integrity, and, more generally, constructing a broad range of least-privilege systems through a collection of object-capability patterns [46].

Yet, language safety alone is not sufficient for development of systems that isolate untrusted third-party applications, e.g., dynamically loaded browser plugins, network functions [51], user-defined database functions [39], kernel extensions [48], etc. Out of the box, Rust provides no mechanisms for *isolation of faults*, i.e., the ability to cleanly terminate a misbehaving computation in a way that leaves the system in a clean state from which recovery is possible [35, 48]. Clean termination of subsystems is challenging in the face of semantically-rich interfaces that encourage frequent zero-copy exchange of references between subsystems and sharing of state—a crash of a single isolated component leaves the entire system in a corrupted state [66].

Several systems explore the possibility to implement isolation in Rust [7, 12, 34, 39, 48, 51], but all of them struggle with the shortcomings of the language and its ecosystem. Isolation in Rust is possible, but requires custom mechanisms to support isolation of heaps [48], execution unwinding [12, 48], dynamic loading of extensions [39, 48], and enforcing system-specific safety and security invariants on subsystem boundaries [48].

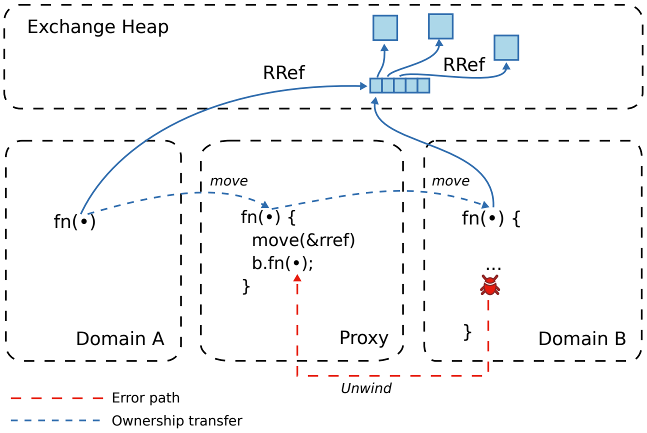

Figure 1. Fault isolation and recovery in RedLeaf.

In this paper, we analyze recent systems that explore software-based isolation in Rust [7, 12, 34, 39, 48, 51] with the goal to identify a minimal set of abstractions and mechanisms that are needed for enabling isolation as a first-class primitive in the Rust language and its ecosystem. Our analysis shows that some of the missing parts require changes to the Rust programming language, better support from the Rust library ecosystem, and others will likely be long-term research efforts. While isolation was not the main design goal for Rust, we argue that a unique combination of design decisions makes Rust an unparalleled platform for providing it. By identifying the missing abstractions, our work takes a step towards enabling isolation as an essential building block for designing secure and reliable systems.

2 Background: Isolation in a Safe System

The idea of using language safety for isolation has its roots in the early language-based operating systems [8, 11, 14, 20, 28, 43, 54, 59]. The first systems implemented an "open" architecture [40], (i.e., a single-user, single-language, single-address space OS) that blurred the boundary between the operating system and the application itself and in general relied on language safety to protect against accidental errors but did not provide isolation of subsystems. SPIN [10] utilized Modula-3 pointers as capabilities to enforce isolation of kernel extensions. Since pointers were exchanged across isolation boundaries, it failed to provide fault isolation—an extension that crashed would leave the system in an inconsistent state. J-Kernel [66] and KaffeOS [6] made the observation that clean termination of untrusted subsystems requires isolation of heaps or object spaces across isolated subsystems. J-Kernel mediated heap access with revocable capabilities, while KaffeOS dynamically monitored every pointer assignment with a "write" barrier.

Singularity was the first system to develop a model that provided support for fault isolation and clean termination of isolated code [35]. To achieve fault isolation, Singularity prohibited sharing of state across isolated subsystems. Instead, isolated subsystems utilized private heaps for regular objects and a special "exchange heap" for communication between domains. Singularity developed novel static analysis and verification techniques to enforce *single ownership* semantics on the exchange heap. At any time, only one isolated subsystem was permitted to have a reference to an object on the shared heap. Unlike previous systems, in Singularity a crashing subsystem was not able to leave the rest of the system in an inconsistent state. Private heaps were not visible to other subsystems, and the single ownership enforced isolation of state on the shared heap.

A primer on isolation in Rust Several recent systems explore the ability to implement isolation in Rust [7, 12, 39, 41, 48]. To illustrate key aspects of implementing isolation in Rust, we consider an isolation model developed in a recent operating system, RedLeaf [48]. In RedLeaf third-party applications are implemented in language-isolated domains, which are dynamically loaded and terminated, similar to processes in a conventional operating system.

To provide clean fault isolation semantics and subsystem termination, RedLeaf adapts the idea that isolated subsystems should not share state on the heap (Figure 1). RedLeaf introduces a special heap construction that separates *private* (per-domain) and *shared* exchange heaps. Objects on the private domain heap are regular Rust data structures with the restriction that they cannot have pointers into other private heaps and a guarantee that they cannot be shared across domains via cross-domain invocations.

To enable cross-domain communication, RedLeaf implements a global *exchange heap* for objects that can be sent across domains. All objects that can be exchanged across domains are allocated on the exchange heap. RedLeaf introduces an abstraction of a *remote reference*, or `RRef<T>`, that is similar to the default Rust `Box<T>` mechanism for allocating objects on the heap. A system-wide registry records all objects allocated on the shared heap. On cross-domain invocations, objects on the exchange heap are either moved between domains or borrowed immutably.

The combination of private and shared heaps eliminates all shared state across domains. When a domain crashes, the private domain heap is reclaimed as raw memory by the kernel. This is safe since no other object in the system outside of the private heap is allowed to have references into the private domain heap. Such organization allows for reclamation of heap without garbage collection which is critical as Rust provides no support for GC. To deallocate objects on the shared heap, RedLeaf walks through the heap registry and deallocates all objects currently owned by the crashing domain. Subsequent invocations of the domain's interfaces return errors, but remain safe and do not trigger panics (RedLeaf mediates all cross-domain interfaces with a layer of trusted, proxy code that blocks future invocations even if the domain is unloaded). Objects allocated by the domain on the shared heap that moved to other domains before the crash remain alive.

```
auto trait MyHeap {}

// Reject pointers and references
impl<T> !MyHeap for *mut T {}
impl<T> !MyHeap for *const T {}
impl<T> !MyHeap for &T {}
impl<T> !MyHeap for &mut T {}

// Allocator, trait bound used to restrict value types
impl<T: MyHeap> MyBox<T> {
    pub fn new(x: T) -> Self { ... }
}
```

Listing 1. Heap isolation invariants with negative traits

Discussion Along with several other systems that recently explored language-based isolation in Rust [7, 12, 34, 39, 51], we find that RedLeaf struggles with the shortcomings of Rust. For example, lacking support to cleanly express the necessary invariants for isolation of heaps in the Rust type system, RedLeaf relies on a complex interface definition language (IDL) that enforces heap isolation invariants outside of Rust. We observe a similar limitation in Splinter, which instead of a zero-copy exchange of objects across isolated subsystems, reverted to a deep copy of objects on cross-subsystem invocations required to maintain isolation invariants [39]. Similarly, due to the limitations of the Rust type system Corundum abandons isolation of function pointers and closures [34]—they can leak between heaps and potentially trigger an unsafe behavior. RedLeaf [48] and Theseus [12] struggle with the lack of support for extendable thread unwinding in `no_std` environments. RedLeaf implements costly continuations to unwind execution across domain boundaries [48]. Theseus implements a custom-built unwind library from scratch. Lacking support from the build environment, RedLeaf [48], Splinter [39], and Theseus [12] rely on an ad hoc trusted build environment to enforce safety of dynamically loaded Rust extensions.

3 Isolation: The Missing Primitives

Rust is uniquely equipped to provide support for practical, low-overhead isolation due to a combination of zero-cost safety and its ownership discipline. In this section, we provide a detailed analysis of Rust's ecosystem with the goal to identify the key mechanisms required to support the development of systems that implement fine-grained isolation of computations.

3.1 Heap Isolation

Fault isolation and clean termination of isolated subsystems is dependent on the ability to enforce isolation of heaps, i.e., the ability to isolate objects that are accessed by individual subsystems on the heap so each subsystem could be cleanly terminated [35]. Specifically, isolation of heaps requires support from the type system to restrict cross-heap pointer references [34, 48]. Unfortunately, the Rust type system provides only partial support to enforce heap isolation invariants.

A natural approach to enforce heap isolation in Rust is to restrict all objects that can be allocated on a specific heap to have pointers only into the same heap. Rust provides a way to encode such a restriction through *trait bounds* on the type of objects that are allocated on the heap, as shown in Listing 1. Here `MyBox<T: MyHeap>::new()` allocates an object of type `T` that is restricted to satisfy the `MyHeap` bound. The `MyHeap` trait is designed to control which pointers are allowed on the heap.

To specify which types satisfy the trait bound and automate derivation of trait bounds for composite types, Rust provides two mechanisms: *negative* and *auto* traits. First, to avoid enumerating all possible types that implement the required trait, Rust allows one to specify the types that *do not* implement the trait with the mechanism of *negative traits* (lines 4–7). For example, we specify that mutable and immutable pointers and references do not implement the `MyHeap` trait (any declared non-pointer types automatically satisfy the trait since we do not negate them). Second, Rust provides a way to automatically check that a composite type, e.g., struct, tuple, array, etc., implements the trait if all subtypes, e.g., fields of a struct, implement the trait with the mechanisms of *auto* traits (line 1).

Unfortunately, the semantics of negative trait implementations is not sufficiently expressive to restrict types that are pointers to functions and closures. Specifically, it is impossible to reject function pointers as they do not have any trait in common and Rust does not provide syntax to express trait bounds on functions with arbitrary number of arguments.

In Rust, all function pointers implement the `Fn` type. However, the precise instance of the function pointer type depends on its argument types. It is possible to use generic traits to implement a marker (or its negation, in this case), for an n-ary function pointer (with arguments and return types), but there is no mechanism to capture function pointers with arbitrary arity. Instead, it's necessary to write a negative `impl` for every arity:

```
impl<R> !MyHeap for fn() -> R {}
impl<A, R> !MyHeap for fn(A) -> R {}
impl<A, B, R> !MyHeap for fn(A, B) -> R {}
```

Similarly it is impossible to restrict pointers to closures (we skip the discussion for brevity).

One possible solution would be to add a *non-generic* trait (e.g., `FptrTrait`), that is implemented by all function pointers. This would allow restricting function pointers as:

```
impl<T: FptrTrait> !MyHeap for T {}
```

In the past, a feature request was made in the Rust project issue tracker to add such a trait to the language motivated by the need to constrain implementations for any function pointer. However, the issue was eventually closed as unresolved due to lack of interest [61]. We argue that it makes sense to revisit the feature request in light of its importance for heap isolation. Alternatively, a more general solution would be:

Language: Support trait bounds on functions and closures with any number of arguments.

3.2 Inter-Process Communication

Support for code generation Isolation of untrusted code relies on the ability to enforce specific policies on cross-domain invocations. For example, cRust [7] and RedLeaf [48] validate that the callee is still alive and the invocation is safe, as well as move the ownership of all objects passed as arguments from caller to callee. In general, mediation of cross-subsystem invocations requires a layer of trusted code to enforce specific checks on each interaction. Manual development of such code is possible only if the interfaces between subsystems are fixed [7, 51]. Even with fixed interfaces the manual development is challenging as language-based systems embrace semantically rich interfaces, e.g., exchange hierarchical data structures, vectors, pointers to interfaces, etc. Automation is needed to generate a layer of code that wraps each interface exposed between subsystems.

Rust provides support for procedural macros [62] that generate code at compile time and interact with the Rust syntax tree. Unfortunately, the syntax tree itself is insufficient for generation of code that requires reasoning about types. For example, lacking support for enforcing heap isolation, RedLeaf checks that arguments and return values of cross-domain functions satisfy specific isolation invariants which requires resolving the full path of each type and its definition. In the past, adding type information to procedural macros was discussed by the Rust community [63]. The agreement was that having type information is beneficial. However, this complicates the internals of the Rust compiler by polluting it with macro knowledge. An alternative solution could be to implement a compiler plugin system that would allow development of Rust compiler extensions [64].

Language: Expose type information in procedural macros.

RTTI for shared types As Rust does not provide support for garbage collection, certain resources of an isolated subsystem, e.g., objects owned by the subsystem on the shared exchange heap, have to be cleaned manually by the execution environment. This requires invocation of a destructor method (`drop`) for each object that needs to be cleaned up. Hence a minimal run-time-type information (RTTI) system is needed to invoke the proper destructor for each object type that can be allocated on the exchange heap. Rust supports RTTI by providing a unique identifier for each type. This identifier is computed at compile time as a hash of a type's AST and the context (i.e., its crate). Unfortunately, this implementation of RTTI does not guarantee collision freedom—an attacker can generate a type with a colliding identifier to trigger an unsafe deallocation of an object.

At the moment it is possible to construct such RTTI information as either an enumerated type that lists all possible types that can be allocated on the shared exchange heap or by implementing a trait that returns a unique type identifier [48]. Both techniques, however, require code-generation. As discussed above, Rust procedural macros provide a convenient code generation interface, however, they lack support for type information which is required for generation of unique identifiers, i.e., macros need to identify aliases to the same type. Therefore, a better solution is to:

> **Language:** Support a collision-free, unique type identifier.

Thread unwinding A panic inside an isolated subsystem requires unwinding of execution of all threads running inside of the subsystem to their entry points. In general, several implementations of unwinding are feasible.

First, it is possible to save the state of all general registers on each cross-subsystem invocation and later unwind the thread of execution to the entry point [48]. Unfortunately, saving and restoring registers is expensive, e.g., about 35-71 cycles for each invocation [42]. This number increases adding another 216-242 cycles if the saved state includes extended registers, i.e., x87 floating point, SSE, AVX registers, etc. [42].

Alternatively, unwinding is possible with native language support for catching exceptions. The benefit of exception handling is that state of all registers can be restored by iterating over the stack frames that contain the saved value for each register. This eliminates the need for explicit register saving and restoring on each invocation and instead incurs the costs of unwinding only if the thread panics.

Rust 1.9.0 introduced support for catching panics, modeled after the Itanium C++ ABI. During the unwind process, a common *personality routine* is invoked repeatedly with information regarding the exception being thrown and the current frame of the stack that should be processed. To restore registers, the unwinder relies on the debugging information embedded into the binary. A commonly used format on Unix systems is DWARF, in which instructions for restoring registers are present for each stack frame that may be unwound.

Several problems complicate the use of Rust's unwind implementation. First, at the moment Rust provides an unwind implementation that relies on the standard library, which cannot be used in baremetal environments. Second, this implementation does not provide any interfaces that would allow system developers to extend it with unwinding of custom frames, e.g., hardware exception frames, low-level assembly and FFI functions, etc. For example, Theseus does not make use of the native unwind facilities in Rust, but rather implements its own unwinder from scratch to allow for more flexibility [12]. We therefore, argue that Rust should:

> **Library:** Support extendable, `no_std` unwind library.

3.3 Safety

Safe loading of unsafe code Similar to how commodity systems create user-processes, a full-featured isolation solution asks for dynamic loading of untrusted third-party Rust code. This allows full flexibility of application development—isolated applications are developed as standalone Rust binaries, and then loaded at runtime. Rust itself provides no support for dynamic extensions. TockOS [41], Netbricks [51] and CRust [7] statically link all the code and produce a single binary.

While dynamic loading of Rust binaries is simple, language-based systems rely on language safety to enforce isolation. Therefore, dynamic code loading requires mechanisms that allow a program to establish that safety guarantees hold for the loaded code. An ideal solution would entail adding support for a proof carrying code (PCC) [49] or a typed assembly language (TAL) [47] which allow verifying safety of a given binary at load time by a small verifier. Despite significant progress demonstrated by Verve [67], which uses TAL to prove system-wide safety, developing TAL for Rust would remain a challenging research problem. In contrast to managed languages, it requires reasoning about object lifetimes. One approach might be to build this functionality into MIR, an intermediate representation for Rust code during compilation that still preserves type and lifetime information.

Lacking support for TAL or PCC, existing Rust systems that provide support for dynamic loading (Splinter [39], Thesesus [12], and Redleaf [48]) rely on a trusted compilation environment. The trusted compilation environment enforces safety by ensuring that types have the same meaning across all loadable components of the system. Specifically, the build environment enforces that all subsystems are compiled with the same versions of types that are visible across subsystems, with the same compiler version and flags, etc. [48]. The build environment measures the state of all components involved in compilation, i.e., all program binaries and their arguments, all code and libraries, etc. Additionally, the build system ensures that isolated subsystems are restricted to only safe Rust, and restrict usage of Rust crates to a white-listed set of libraries and possibly a subset of types from the core Rust libraries. Existing trusted build environments [12, 39, 48] are ad hoc, incomplete, and error prone. We, therefore, need to:

> **Research:** Support typed assembly language for Rust.
> **Ecosystem:** Support trusted build environments.

Stack safety Rust protects against stack overflows with a dedicated guard page at the end of each stack [60]. To prevent stack clashing attacks [3], Rust implements stack probing [60]: a small code sequence is generated by the compiler on every stack allocation to ensure that all pages of the new allocation (including the guard page) are accessed. Unfortunately, this implementation requires support for a memory management unit that can write-protect a region of memory to trigger an exception once the guard page is reached. This requirement prevents language-based systems from executing in environments that do not have MMU support (e.g., trusted execution environments like Intel SGX, system management mode (SMM), and many embedded systems). Ideally, the Rust language should provide support to customize the probing function that can either check for the remaining stack space, or allocate a new stack.

In the past, Rust provided support for *segmented stacks*, a mechanism that allowed growing the stack by allocating stack regions also known as *stacklets* [52]. The compiler generated function prologues to check for the available stack space and allocate more space on-demand. The support for segmented stack was deprecated due to possible performance problems [4, 65]: Segmented stacks could become a "hot spot" if allocated and freed in a loop [65]. We argue that it makes sense to bring back support for software-managed stacks in order to enable language-based isolation in execution environments that do not provide support for write protection.

Runtime: Provide software-only stack guard with extensible probing interface.

Safe serialization As isolation relies on the safety of the language, isolated subsystems are restricted to a safe subset of Rust [7, 12, 39, 41, 48, 51]. While in most cases safe Rust imposes no performance penalty on the isolated code, some corner cases force safe Rust to execute 8-24% slower compared to unsafe code [48]. Specifically, we identify the ability to serialize and deserialize data structures as a significant source of overhead in safe Rust. Serialization is frequent in network and disk workloads when requests and metadata are serialized from raw I/O bytes. For maximum deserialization performance, ideally most bytes in the raw I/O request could be directly cast to a Rust struct. Casting byte buffers to structs is generally unsafe in Rust, since there is no guarantee that either Rust's type alignment requirements nor any type invariants are upheld after the cast (e.g., a string should only contain valid UTF-8 characters). However, in cases where we deal with "plain-old" data (structures containing only basic types) such casts would be safe.

To understand the impact of serialization, we measure the overhead for serializing/deserializing the set response header in the key-value workload used in RedLeaf, a data structure with a size of 89 bytes consisting of 7 distinct fields (Table 1). On RedLeaf, which uses `extend_from_slice()`, serialization takes about 33 cycles and deserialization takes 91 cycles (all results are taken on a 2.20 GHz Skylake CPU). The overhead is significant for a request processing path that takes 200-300 cycles. In Rust it is possible to use procedural macros [62] to check if a struct is a plain-old data object and then generate casts to and from byte arrays. *nue* implements this approach: serialization is a cast with a single copy, which is about 5x faster than the seven invocations of `extend_from_slice()` in RedLeaf. Deserialization is also just a cast (0 cycles). Unfortunately, *nue* does not support borrowing of arrays and strings, so having such elements in a packet would lead to an additional memory allocation during deserialization. Another serialization library, *serde*, allows borrowing of slices and strings without copying them out of the underlying packet buffer, but fails to provide an efficient implementation. An ideal solution would combine the features offered in *nue* and *serde*.

Library	safe	slice borrow	Cycles	
			ser	deser
RedLeaf [48]	y	n	33	91
nue [50]	y	n	6	0
serde/cbor [19]	y	y	803	1678

Table 1. Comparison of different serialization techniques in Rust for plain-old-data structures.

Library: Develop zero-copy serialization of "plain-old" data structures.

4 Conclusions

Isolation is an essential building block for a number of system abstractions that range from fine-grained access control and fault isolation to transactional recovery and persistence. Historically, fine-grained isolation remained prohibitively expensive due to the overhead of crossing isolation boundaries and enforcing isolation of objects on the heap. Arguably, safe languages, and specifically, zero-cost safe languages like Rust enable fine-grained isolation at the speeds that are practical for software with extremely tight performance budgets: operating system kernels, web browsers, database and storage systems, etc. By examining recent systems that leverage Rust for isolation, our work identifies weak points in the Rust programming language and its ecosystem that, in our opinion, hinder development and adoption of isolation mechanisms. We hope that our analysis will help shape the research needed to enable isolation as an "out-of-the-box" primitive in the Rust ecosystem.

Acknowledgments

We would like to thank HotOS'21, APSys'21, and PLOS'21 reviewers for various insights helping us to improve this work. This research is supported in part by the National Science Foundation under Grant Numbers 1837127 and 1840197, and VMWare.

References

[1] seL4 Performance. https://sel4.systems/About/Performance/.

[2] WebAssembly Specification. https://webassembly.github.io/spec/core/.

[3] Qualys Security Advisory. The Stack Clash. https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt, June 2017. Accessed: 2021-02-03.

[4] Brian Anderson. [rust-dev] Abandoning segmented stacks in Rust. https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html. Accessed: 2021-02-03.

[5] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. XOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS'12, pages 49–60, New York, NY, USA, 2012.

[6] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[7] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*, pages 156–161, 2017.

[8] Fred Barnes, Christian Jacobsen, and Brian Vinter. RMoX: A Raw-Metal occam Experiment. In *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 182–196, September 2003.

[9] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE CORP BEDFORD MA, 1976.

[10] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–283, 1995.

[11] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*, pages 11–1–11–51, 2007.

[12] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 1–19, 2020.

[13] Bromium. Bromium micro-virtualization, 2010. http://www.bromium.com/misc/BromiumMicrovirtualization.pdf.

[14] Hank Bromley and Richard Lamson. *LISP Lore: A Guide to Programming the Lisp Machine.* Springer Science & Business Media, 2012.

[15] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N Bairavasundaram, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 25–25, 2009.

[16] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 45–58. ACM, 2009.

[17] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1988.

[18] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.

[19] Serde developers. Serde: a framework for serializing and deserializing Rust data structures efficiently and generically. https://crates.io/crates/serde, 2021. Accessed: 2021-02-03.

[20] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. The Inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[21] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[22] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[23] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 75–88, 2006.

[24] Feske, N. and Helmuth, C. Design of the Bastei OS architecture. 2007.

[25] Vinod Ganapathy, Matthew J Renzelmann, Arini Balakrishnan, Michael M Swift, and Somesh Jha. The design and implementation of microdrivers. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 168–178. ACM, 2008.

[26] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 193–206, 2003.

[27] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 109–114. ACM, 2000.

[28] Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation. *Addison-Wesley Longman Publishing Co., Inc.*, 1983.

[29] Hardy, N. KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[30] Heiser, G. and Elphinstone, K. and Kuz, I. and Klein, G. and Petters, S.M. Towards trustworthy computing systems: taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[31] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[32] Hohmuth, M. and Peter, M. and Härtig, H. and Shapiro, J.S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop*, page 22. ACM, 2004.

[33] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. MSwitch: A Highly-Scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, New York, NY, USA, 2015.

[34] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[35] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[36] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 445–458, Seattle, WA, April 2014.

[37] Intel Corporation. Storage Performance Development Kit (SPDK). `https://spdk.io`.

[38] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, 2019.

[39] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 627–643, Carlsbad, CA, October 2018.

[40] Butler W. Lampson and Robert F. Sproull. An Open Operating System for a Single-User Machine. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79)*, pages 98–105. 1979.

[41] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 234–251, 2017.

[42] Zhaofeng Li, Tianjiao Huang, Vikram Narayanan, and Anton Burtsev. Understanding the overheads of hardware and language-based ipc mechanisms. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*, 2021.

[43] Peter W Madany, Susan Keohan, Douglas Kramer, and Tom Saulpaugh. JavaOS: A Standalone Java Environment. *White Paper, Sun Microsystems, Mountain View, CA*, 1996.

[44] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 115–128, 2011.

[45] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 459–473, Seattle, WA, April 2014.

[46] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, May 2006.

[47] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[48] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 21–39, November 2020.

[49] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'97, pages 106–119, New York, NY, USA, 1997.

[50] nue developers. nue: A collection of tools for working with binary data and POD structs in Rust. `https://crates.io/crates/nue`, 2021. Accessed: 2021-02-03.

[51] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 203–216, Savannah, GA, November 2016.

[52] LLVM Project. Segmented Stacks in LLVM âĂŤ LLVM 12 documentation. `https://llvm.org/docs/SegmentedStacks.html`. Accessed: 2021-02-03.

[53] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 13–24, San Jose, CA, June 2013.

[54] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, 1980.

[55] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*, pages 1–11, 2010.

[56] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 323–336, San Jose, CA, April 2012.

[57] Mark Sullivan and Michael Stonebraker. Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB'91, pages 171–180, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[58] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European Workshop*, pages 102–107, 2002.

[59] Daniel C Swinehart, Polle T Zellweger, Richard J Beach, and Robert B Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):419–490, 1986.

[60] Rust Language Team. API documentation for the Rust 'probestack' mod in crate 'compiler_builtins'. `https://docs.rs/compiler_builtins/0.1.39/compiler_builtins/probestack/index.html`. Accessed: 2021-02-03.

[61] Rust Language Team. A 'functionpointer' trait to represent all 'fn' types. `https://github.com/rust-lang/lang-team/issues/23`. Accessed: 2021-02-03.

[62] Rust Language Team. The Rust Reference: Procedural Macros. `https://doc.rust-lang.org/reference/procedural-macros.html`. Accessed: 2021-02-03.

[63] Rust Language Team. The Rust RFC Book: 1566-proc-macros. `https://rust-lang.github.io/rfcs/1566-proc-macros.html`. Accessed: 2021-02-03.

[64] Rust Language Team. Tracking issue for plugin stabilization ('plugin', 'plugin_registrar' features). `https://github.com/rust-lang/rust/issues/29597`. Accessed: 2021-02-03.

[65] The Go Programming Language team. Go 1.3 Release Notes. `https://golang.org/doc/go1.3`. Accessed: 2021-02-03.

[66] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[67] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 99–110, New York, NY, USA, 2010.

[68] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009.