RedLeaf: Isolation and Communication in a Safe Operating System

Vikram Narayanan¹, Tianjiao Huang¹, David Detweiler¹, Dan Appel¹, Zhaofeng Li¹, Gerd Zellweger², Anton Burtsev¹

OSDI ’20

¹University of California, Irvine

²VMware Research
History of Isolation

- Isolation of kernel subsystems
  - Final report of Multics (1976)
  - Scomp (1983)
- Systems remained monolithic
  - Isolation was expensive
Isolation mechanisms

- **Hardware Isolation**
  - Segmentation (46 cycles)
  - Page table isolation (797 cycles)
  - VMFUNC (396 cycles)
  - Memory protection keys (20-26 cycles)

- **Language based isolation**
  - Compare drivers written (DPDK-style) in a safe high-level language (C, Rust, Go, C#, etc.)
  - Managed runtime and Garbage collection (20-50% overhead on a device-driver workload)

---

1. L4 Microkernel: Jochen Liedtke
2. https://sel4.systems/About/Performance/
3. Lightweight Kernel Isolation with Virtualization and VM Functions, VEE 2020
4. Hodor: Intra-process isolation for high-throughput data plane libraries
5. The Case for Writing Network Drivers in High-Level Programming Languages, ANCS 2019
Traditional Safe languages vs Rust

Java, C# etc.
Traditional Safe languages vs Rust

Java, C# etc.

A → Vector
Traditional Safe languages vs Rust

Java, C# etc.

A Vector B

• Linear types
• Enforces type and memory safety
• Statically checked at compile time
• Safety without runtime garbage collection overhead
Traditional Safe languages vs Rust

Java, C# etc.

Vector
Traditional Safe languages vs Rust

Java, C# etc.

Vector
Traditional Safe languages vs Rust

Java, C# etc.

Vector Garbage collection?
Traditional Safe languages vs Rust

Java, C# etc.

Vector Garbage collection?
Traditional Safe languages vs Rust

- Rust
- Linear types
- Enforces type and memory safety
- Statically checked at compile time
- Safety without runtime garbage collection overhead

Java, C# etc.

Vector

Garbage collection?
Traditional Safe languages vs Rust

- Rust
  - Linear types
  - Enforces type and memory safety
  - Statically checked at compile time
  - Safety without runtime garbage collection overhead

Java, C# etc.

Vector collection?
Traditional Safe languages vs Rust

Java, C# etc.

Rust

Vector

Garbage collection?
Traditional Safe languages vs Rust

Java, C# etc.

- Linear types

Vector Garbage collection?
Traditional Safe languages vs Rust

Java, C# etc.

Rust

- Linear types
- Enforces type and memory safety
Traditional Safe languages vs Rust

Java, C# etc.

Vector Garbage collection?

Rust

- Linear types
- Enforces type and memory safety
- Statically checked at compile time
Traditional Safe languages vs Rust

- Rust
  - Linear types
  - Enforces type and memory safety
  - Statically checked at compile time
  - Safety without runtime garbage collection overhead

Java, C# etc.
- Mostly use Rust as a drop-in replacement for C
- Numerous possibilities
  - Fault Isolation
  - Transparent device-driver recovery
  - Safe Kernel extensions
  - Fine-grained capability-based access control etc.
Fault isolation in Language-based systems

- e.g., SPIN (using Modula-3 pointers)
Fault isolation in Language-based systems

- e.g., SPIN (using Modula-3 pointers)
Fault isolation in Language-based systems

- e.g., SPIN (using Modula-3 pointers)
Fault isolation in Language-based systems

- e.g., SPIN (using Modula-3 pointers)
Fault isolation in Language-based systems

- e.g., SPIN (using Modula-3 pointers)
Language-based isolation: Deep copy

```
fn foo(obj1: Object1, obj2: Object2) {
   call_other(obj1, obj2);
}

fn bar(....) {
   do_work(&obj1, &obj2);
}
```

- e.g., J-Kernel, KaffeOS
Language-based isolation: Deep copy

- e.g., J-Kernel, KaffeOS
Language-based isolation: Deep copy

- e.g., J-Kernel, KaffeOS
Language-based isolation: Deep copy

- e.g., J-Kernel, KaffeOS
Language-based isolation: Capabilities

- e.g., J-Kernel
Language-based isolation: Capabilities

- e.g., J-Kernel
Language-based isolation: Capabilities

- e.g., J-Kernel
Language-based isolation: Capabilities

• e.g., J-Kernel
Language-based isolation: Capabilities

- e.g., J-Kernel
Language-based isolation: Capabilities

- e.g., J-Kernel
Language-based isolation: Singularity

- Statically enforced ownership discipline
Language-based isolation: Singularity

- Statically enforced ownership discipline
- Single ownership model

```rust
fn foo(obj1: Object1, obj2: Object2) {
    call_other(obj1, &obj2);
}
fn bar(...) {
    do_work(&obj1, &obj2);
}
```

Static analysis
Verification tools

• Statically enforced ownership discipline
• Single ownership model
Language-based isolation: Singularity

- Statically enforced ownership discipline
- Single ownership model
Language-based isolation: Singularity

- Statically enforced ownership discipline
- Single ownership model
Language-based isolation: Singularity

- Statically enforced ownership discipline
- Single ownership model
- Static analysis and verification tools (Sing#)
Language-based isolation: Singularity

- Statically enforced ownership discipline
- Single ownership model
- Static analysis and verification tools (Sing#)
- Reusing the moved object return an error
- zero-copy
RedLeaf
Architecture

[Diagram showing the architecture of the system with components like NVMe Driver, Ixgbe Driver, Trusted crate, Microkernel, FS, Proxy, and RedLeaf User.]
Architecture

Shared Heap

rv6 Core

Proxy

rv6 User

RedLeaf User

rv6

Proxy

FS

Net

Proxy

NVMe Driver

Trusted crate

Igxbe Driver

Trusted crate

Microkernel

Compiler-enforced protection domains

RRef
- After a domain crash:
  - Unwind all threads running inside
  - Subsequent invocations return error
  - All resources are deallocated
  - Other threads continue execution
Fault Isolation

- After a domain crash:
  - Unwind all threads running inside

- Subsequent invocations return error
- All resources are deallocated
- Other threads continue execution
Fault Isolation

- After a domain crash:
  - Unwind all threads running inside
  - Subsequent invocations return error
• After a domain crash:
  • Unwind all threads running inside
  • Subsequent invocations return error
  • All resources are deallocated
• After a domain crash:
  • Unwind all threads running inside
  • Subsequent invocations return error
  • All resources are deallocated
  • Other threads continue execution
Heap Isolation

- Domains never hold pointers into other domains
• Domains never hold pointers into other domains
• Domains never hold pointers into other domains
• Special shared heap for passing objects between domains
Objects in shared heap can only be exchangeable types
• Objects in shared heap can only be exchangeable types
• Cannot point to normal pointers in shared heap or private heap
Ownership tracking

- `RRef<T>`'s can be passed between domains
Ownership tracking

- `RRef<T>`'s can be passed between domains
- Metadata keeps track of owner domain and ref count
Ownership tracking

- `RRef<T>`'s can be passed between domains
- Metadata keeps track of owner domain and ref count
- Mediated through trusted proxies
• Maintains a global registry of allocated objects
Heap reclamation

- Maintains a global registry of allocated objects
- On panic
  - Deallocate all objects owned by the crashing domain
  - Defer borrowed `RRef`'s until ref count is zero
Cross-domain call proxying

- Checks if domain is alive
Cross-domain call proxying

- Checks if domain is alive
- Creates continuation
Cross-domain call proxying

- Checks if domain is alive
- Creates continuation
- Moves ownership of all RRef<T>
• Validate domain interfaces
• Generate proxies to enforce ownership discipline
• e.g., Block Device domain Interface

```rust
pub trait BDev {
    fn read(&self, block: u32, data: RRef<[u8; BSIZE]>) -> RpcResult<RRef<[u8; BSIZE]>>;
    fn write(&self, block: u32, data: &RRef<[u8; BSIZE]>) -> RpcResult<()>;
}
```
• Heap isolation
• Exchangeable types
• Ownership tracking
• Interface validation
• Cross-domain call proxying
Device driver Recovery
Device driver Recovery

- Support transparent device driver recovery
- Wraps the interface to expose an identical interface
- Interposes on all communication
Device driver Recovery

- Support transparent device driver recovery
- Wraps the interface to expose an identical interface
- Interposes on all communication
Device driver Recovery

- Support transparent device driver recovery
- Wraps the interface to expose an identical interface
- Interposes on all communication
Support transparent device driver recovery
Wraps the interface to expose an identical interface
Interposes on all communication
Evaluation
System setup

- 2 x Intel E5-2660 v3 10-core CPUs at 2.60 GHz (Haswell EP)
- Disabled: Hyper-threading, Turbo boost, CPU Idle states
- Linux and DPDK benchmarks run on version 4.8.4
- RedLeaf benchmarks run on baremetal
## Communication costs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>seL4</td>
<td>834</td>
</tr>
<tr>
<td>VMFUNC</td>
<td>169</td>
</tr>
<tr>
<td>VMFUNC-based call/reply invocation</td>
<td>396</td>
</tr>
<tr>
<td>RedLeaf cross-domain invocation</td>
<td>124</td>
</tr>
<tr>
<td>RedLeaf cross-domain invocation (passing an RRef)</td>
<td>141</td>
</tr>
<tr>
<td>RedLeaf cross-domain invocation via shadow</td>
<td>279</td>
</tr>
<tr>
<td>RedLeaf cross-domain via shadow (passing an RRef)</td>
<td>297</td>
</tr>
</tbody>
</table>
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C-style Rust: No higher order functions
- Idiomatic Rust - Option<(usize, usize)>
- Vary the size (2^{12} to 2^{26} at 75% full)
- Idiomatic Rust - 25% overhead, C-style - Closer to C
- Cycles for a set on 2^{20} (C - 51, idrust - 81, cst-rust - 48)
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions `usize, usize`
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions usize, usize
- Idiomatic Rust - Option<(usize, usize)>
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions `usize, usize`
- Idiomatic Rust - `Option<(&usize, &usize)>`
- Vary the size (2^12 to 2^26 at 75% full)
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions usize, usize
- Idiomatic Rust - Option<(usize, usize)>
- Vary the size (2^{12} to 2^{26} at 75% full)
- Idiomatic Rust - 25% overhead, C-style - Closer to C
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions `usize, usize`
- Idiomatic Rust - `Option<(usize, usize)>`
- Vary the size ($2^{12}$ to $2^{26}$ at 75% full)
- Idiomatic Rust - 25% overhead, C-style - Closer to C
Language overheads: C vs Rust

- Hashtable - (FNV hash, open addressing, <8B, 8B>)
- C, Idiomatic Rust, C-style Rust,
- C-style Rust: No higher order functions `usize`, `usize`
- Idiomatic Rust - `Option<(usize, usize)>`
- Vary the size ($2^{12}$ to $2^{26}$ at 75% full)
- Idiomatic Rust - 25% overhead, C-style - Closer to C
- Cycles for a `set` on $2^{20}$ (C - 51, idrust - 81, cst-rust - 48)
Configurations

- redleaf-driver
Configurations

- redleaf-driver
- redleaf-domain
Configurations

- redleaf-driver
- redleaf-domain
- rv6-domain
Case Study: Device Drivers

Configurations

- redleaf-driver
- redleaf-domain
- rv6-domain
- redleaf-shadow
Configurations

- redleaf-driver
- redleaf-domain
- rv6-domain
- redleaf-shadow
- rv6-shadow
- Linux application (2921 cycles)
• Linux application (2921 cycles)
• DPDK (388 cycles) and Redleaf driver (400 cycles)
Ixgbe performance benchmark

- Linux application (2921 cycles)
- DPDK (388 cycles) and Redleaf driver (400 cycles)
- Linux application (2921 cycles)
- DPDK (388 cycles) and Redleaf driver (400 cycles)
- Redleaf-domain and Redleaf-shadow
- Linux application (2921 cycles)
- DPDK (388 cycles) and Redleaf driver (400 cycles)
- Redleaf-domain and Redleaf-shadow
- Rv6-domain and shadow
Application Benchmarks

- Maglev load balancer
Application Benchmarks

- Maglev load balancer
- Network-attached key-value store
Application Benchmarks

- Maglev load balancer
- Network-attached key-value store
- a minimal webserver
Application benchmarks: Maglev

- Load balancer developed by Google
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
- Compare C vs Rust Maglev versions
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
- Compare C vs Rust Maglev versions

- Linux Application - Limited due to synchronous socket interface
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
- Compare C vs Rust Maglev versions

- Linux Application - Limited due to synchronous socket interface
- DPDK Application (batch of 32) - 9.7 Mpps per-core
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
- Compare C vs Rust Maglev versions

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>DPDK</th>
<th>redleaf-driver</th>
<th>redleaf-domain</th>
<th>redleaf-shadow</th>
<th>rv6-domain</th>
<th>rv6-shadow</th>
<th>line rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pkts/s (Millions) maglev-32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9.7 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.2 M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Linux Application - Limited due to synchronous socket interface
- DPDK Application (batch of 32) - 9.7 Mpps per-core
- Redleaf-driver - 7.2 Mpps
Application benchmarks: Maglev

- Load balancer developed by Google
- Lookup or insert into the flow tracking table
- Compare C vs Rust Maglev versions

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>DPDK</th>
<th>redleaf-driver</th>
<th>redleaf-domain</th>
<th>redleaf-shadow</th>
<th>rv6-domain</th>
<th>rv6-shadow</th>
<th>line rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pkts/s (Millions)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>maglev-32</td>
<td>1 M</td>
<td>9.7 M</td>
<td>7.2 M</td>
<td>5.3 M</td>
<td>5.1 M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Linux Application - Limited due to synchronous socket interface
- DPDK Application (batch of 32) - 9.7 Mpps per-core
- Redleaf-driver - 7.2 Mpps
- Rv6-domain - 5.3 Mpps, Rv6-shadow - 5.1 Mpps
• Network attached key-value store
Application: Key Value Store

- Network attached key-value store
- FNV Hash with open addressing (linear probing)
Application: Key Value Store

- Network attached key-value store
- FNV Hash with open addressing (linear probing)
- Rust (C-Style) vs a DPDK application
• Network attached key-value store
• FNV Hash with open addressing (linear probing)
• Rust (C-Style) vs a DPDK application
• Various Key value sizes (<8B, 8B>, <16B, 64B>, <64B, 64B>)
Application: Key Value Store

- Network attached key-value store
- FNV Hash with open addressing (linear probing)
- Rust (C-Style) vs a DPDK application
- Various Key value sizes (<8B, 8B>, <16B, 64B>, <64B, 64B>)
- Achieves 61-86% performance (`extend_from_slice()` x 3)
• Network attached key-value store
• FNV Hash with open addressing (linear probing)
• Rust (C-Style) vs a DPDK application
• Various Key value sizes (<8B, 8B>, <16B, 64B>, <64B, 64B>)
• Achieves 61-86% performance (extend_from_slice() x 3)
• With Unsafe Rust 85-94% performance of the C DPDK
Device driver recovery

- Rv6 program <-> in-memory block device
Device driver recovery

- Rv6 program <-> in-memory block device
- Trigger crash every second
Device driver recovery

- Rv6 program <-> in-memory block device
- Trigger crash every second
- Small drop in performance
  - Read: 2062 MB/s (restart), 2164 MB/s (without restart)
Device driver recovery

- Rv6 program <-> in-memory block device
- Trigger crash every second
- Small drop in performance
  - Read: 2062 MB/s (restart), 2164 MB/s (without restart)
  - Writes: 356 MB/s (restart), 423 MB/s (without restart)
Conclusion

- Heap isolation, exchangeable types, ownership tracking, interface validation, cross-domain call proxying
- Provides a collection of mechanisms for enabling isolation
- A step forward in enabling future system architectures
  - Secure kernel extensions, fine-grained access control, transparent recovery etc.

Source code

https://mars-research.github.io/redleaf/
Thank you!