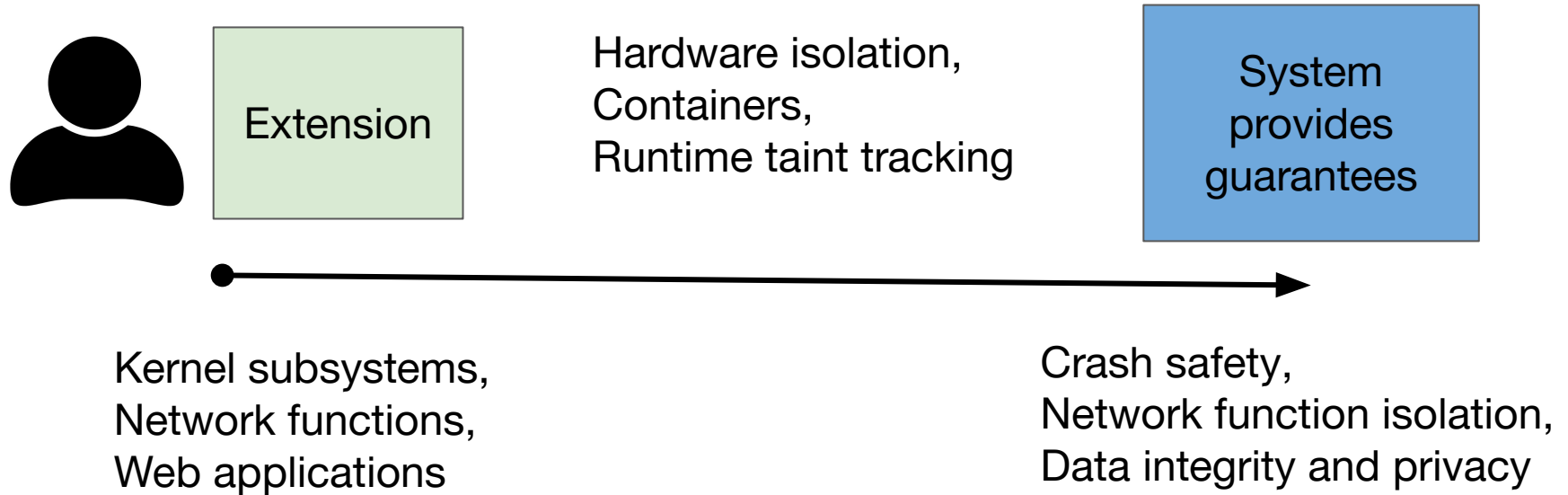


We Need to Talk About the Actual Guarantees of Rust-based Systems

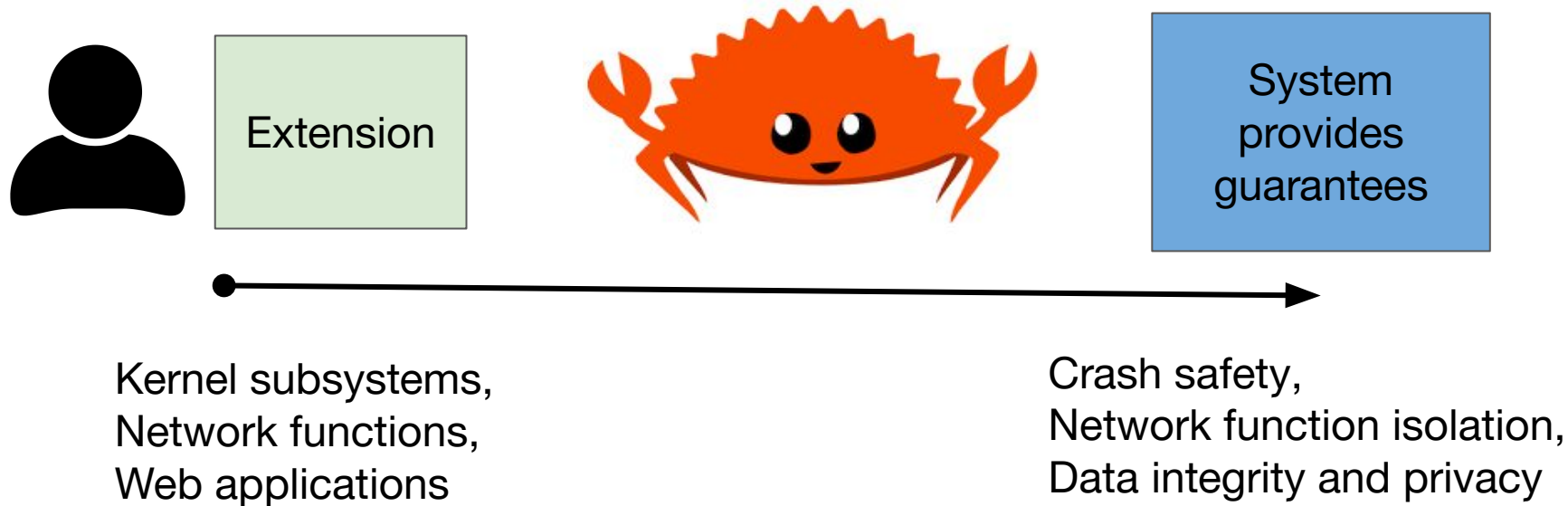
Corinn Tiffany Artem Agvanian Ziyun Song Isabella Szabo Kinan Dak Albab[†]
Will Crichton Philip Levis[‡] Akshay Narayan Deepti Raghavan Malte Schwarzkopf
Brown University [†]Boston University [‡]Stanford University and Google

PLOS '26, Prague, Czech Republic

Systems provide security & isolation properties over *extensions*



Systems bootstrap security & isolation guarantees from **Rust**



Rust-based systems!

Putting the “Micro” Back in Microservices
NetBricks: Taking the V out of NFV

RedLeaf: Isolation and Communication in a Safe Operating System
DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency

Cocoon: Static Information Flow Control in Rust

Modular Information Flow Analysis to check file-system crash consistency
SquirrelFS: using

Sesame: Practical End-to-End Privacy Compliance with Policy Containers and Privacy Regions

Caveats for **unsafe** code...

“safe subset of Rust
programs”

no “malicious” unsafe code

“restricted to safe Rust”

“absence of unsafe Rust”

“safe Rust”

Reality: unsafe code is **common**.

More than half of the 500 most-used crates
call into unsafe code [ICSE ‘20].

Caveats for `unsafe` code...

“safe subset of Rust

no “malici

grams”

“restricted to safe R

unsafe Rust”

Impractical!

safe Rust

Reality: unsafe code is **common**.

More than half of the 500 most-used crates
call into unsafe code [ICSE ‘20].

What if we use the systems anyway?

System	Violation
Sesame [SOSP '24]	privacy policy violations
RedLeaf [OSDI '20]	no isolation between subsystems
NetBricks [OSDI '16]	packet modified after NF sent
Cocoon [OOPSLA '24]	data leakage
Flowistry [PLDI '22]	misses existing flows (unsound)

Extensions that invoke **reasonable, correct library code** can still **break the guarantees of a system**.

What about safe Rust is useful for systems?

Example properties:

Encapsulation	Private members of structs cannot be accessed outside their module.
Immutability of References	All data accessible through a reference will not mutate during the lifetime of the reference.

Libraries *intentionally* break properties that systems need

In safe code, the compiler enforces the absence of undefined behavior (UB), and also properties like **encapsulation** and **immutability of references**.

We call these **incidental properties**; violating them requires unsafe code, but does not cause UB.

Libraries often *intentionally* break incidental properties in order to provide useful functionality.

What unsafe code breaks system guarantees?

With respect to an incidental property, unsafe code may be one of the following:

Unsound	Has undefined behavior and no guarantees
Sound	No undefined behavior, but may or may not preserve the incidental property
Property-preserving	Sound and preserves the incidental property

What unsafe code breaks system guarantees?

With respect to an incidental property, unsafe code may be one of the following:

Unsound	
Sound	Orthogonal: Miri, Kani, Verus, etc.
Property-preserving	The focus of this work!

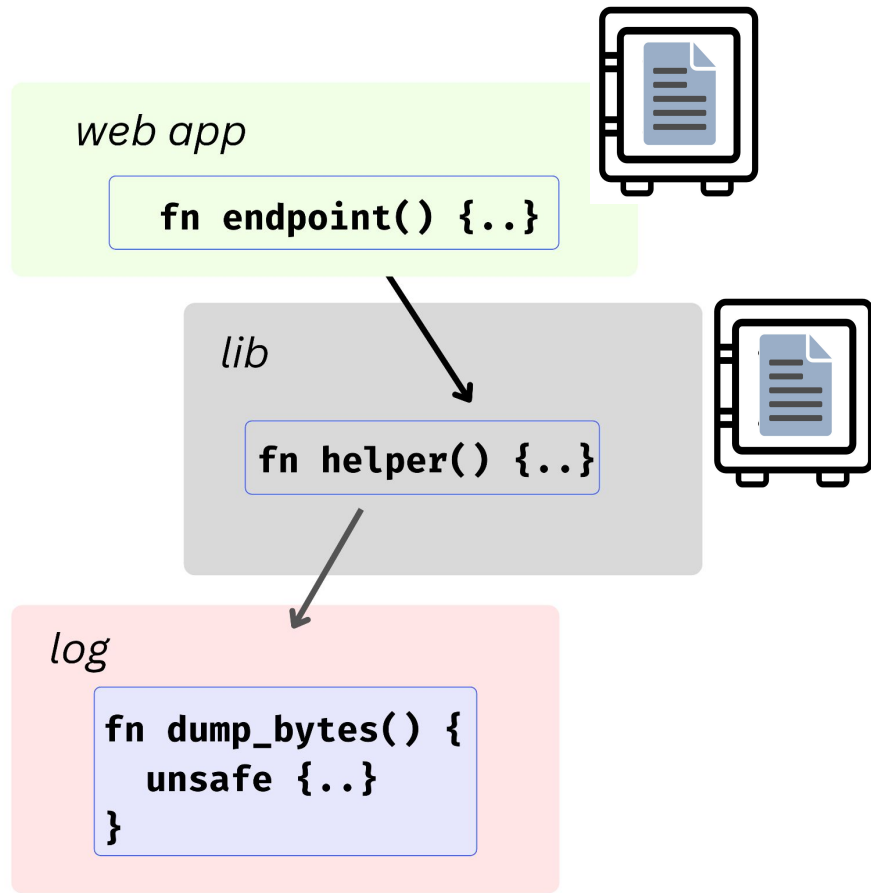
Case studies:
Sesame [SOSP '24] and **RedLeaf** [OSDI '20]

Leaking Private Data with Sesame

`PCon<T, P>` stores data `T` as a private member.

Relies on *encapsulation* to lock data in.

```
struct PCon<T, P> {  
  data: T,  
  policy: P  
}
```

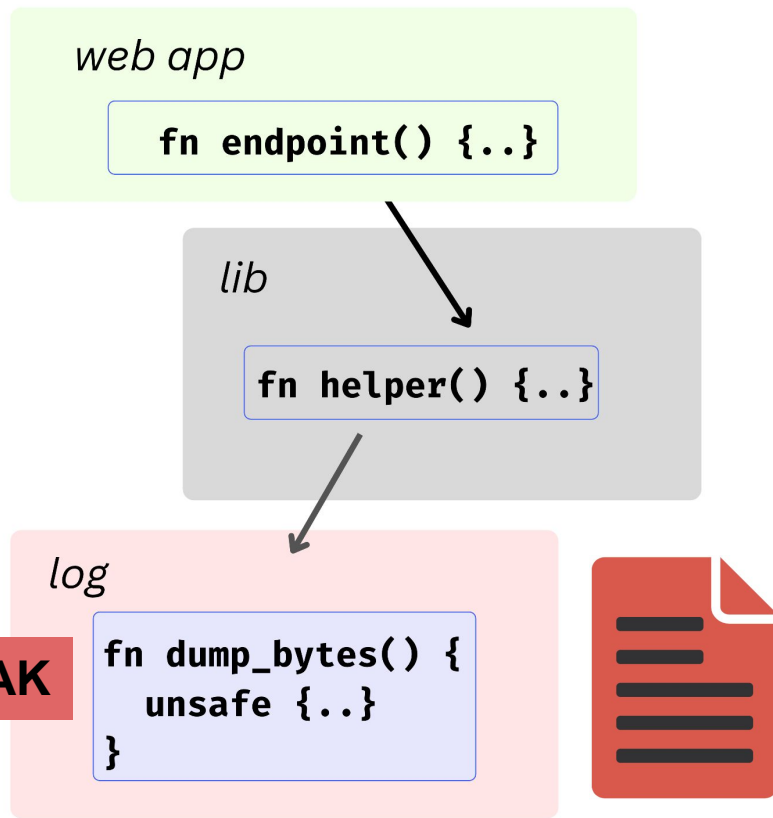


Leaking Private Data with Sesame

`PCon<T, P>` stores data `T` as a private member.

Relies on *encapsulation* to lock data in.

```
struct PCon<T, P> {  
  data: T,  
  policy: P  
}
```



Leaking Private Data with Sesame

`PCon<T, P>` stores data `T` as a private member.

Relies on *encapsulation* to lock data in.

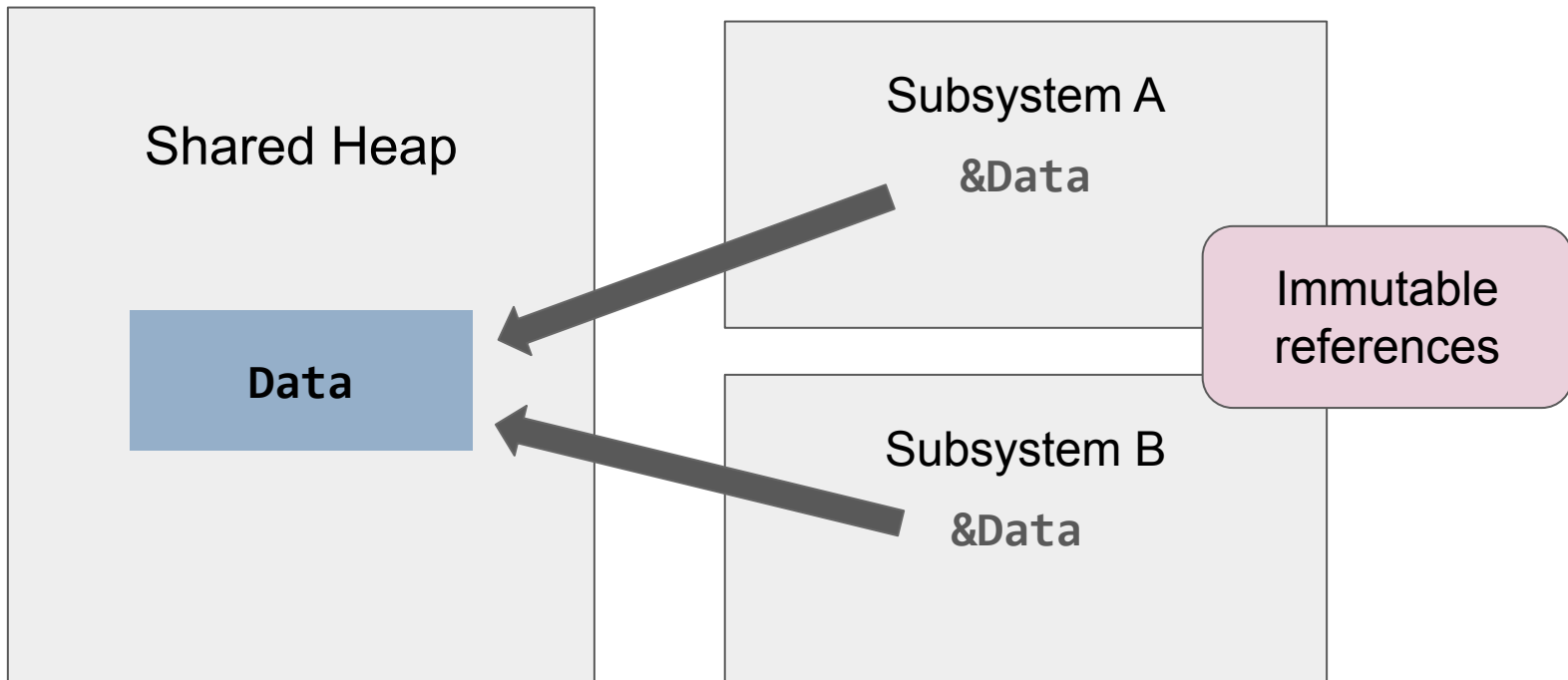
```
struct PCon<T, P> {  
    data: T,  
    policy: P  
}
```



Unsafe code
violates
encapsulation →
catastrophic
data leak

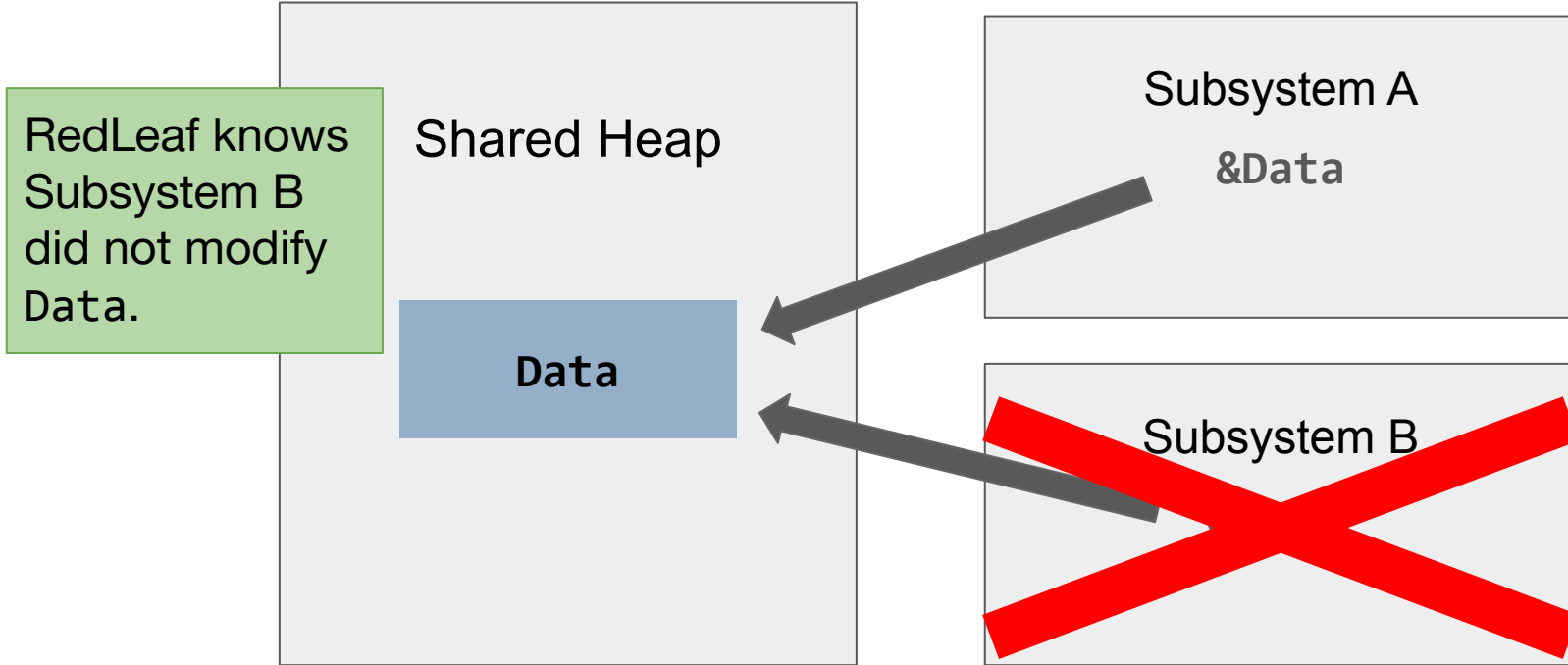
Kernel Subsystem Isolation with RedLeaf

Assumption: subsystem with an immutable reference to data cannot modify it.



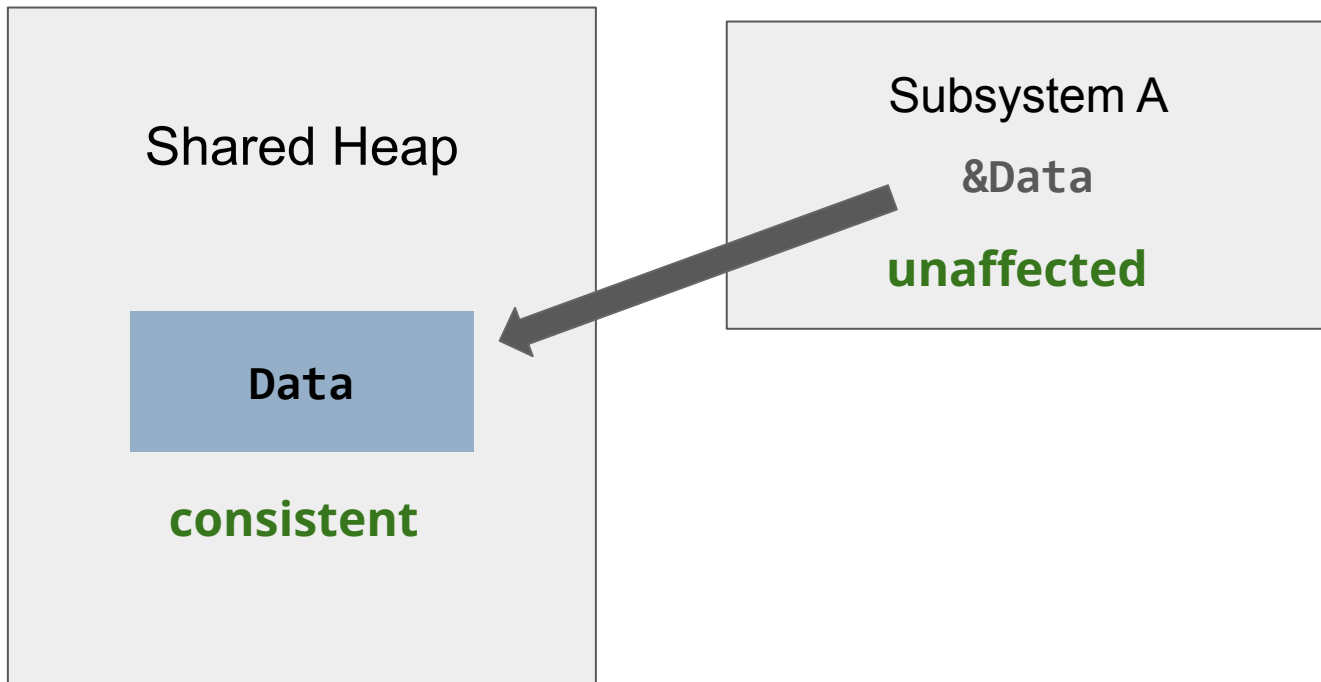
Kernel Subsystem Isolation with RedLeaf

Assumption: subsystem with an immutable reference to data cannot modify it.



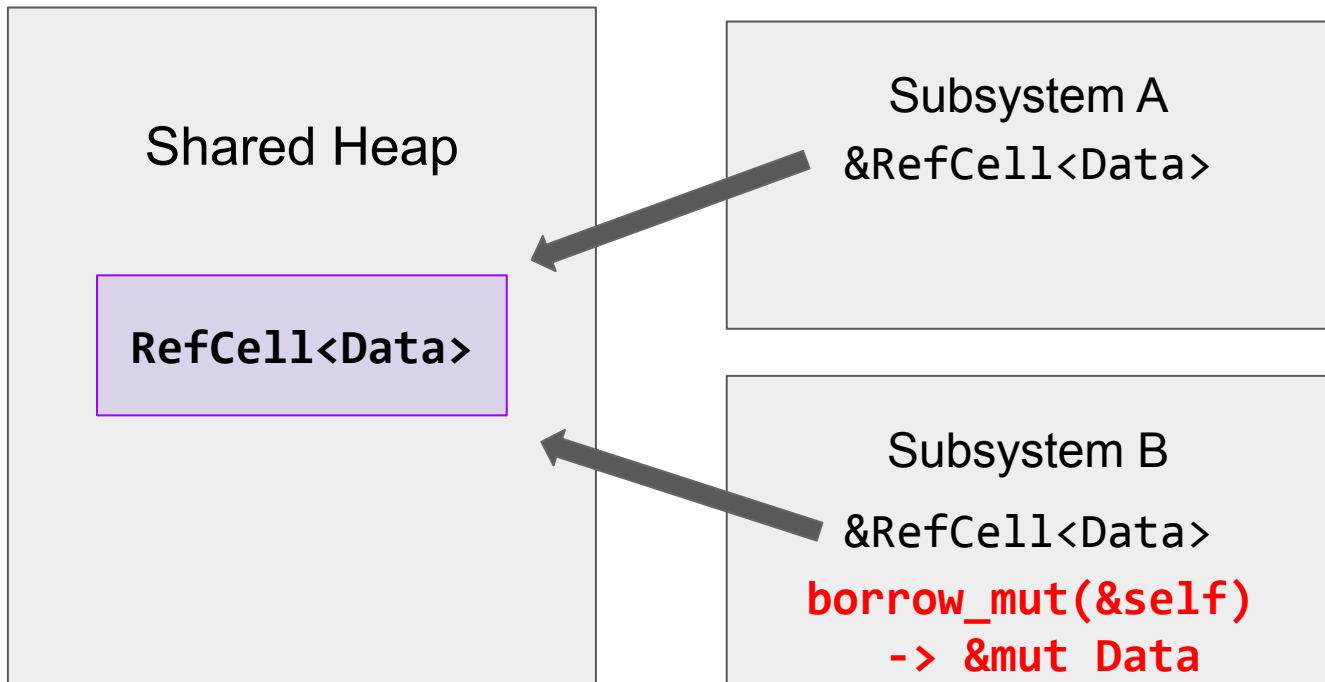
Kernel Subsystem Isolation with RedLeaf

Assumption: subsystem with an immutable reference to data cannot modify it.



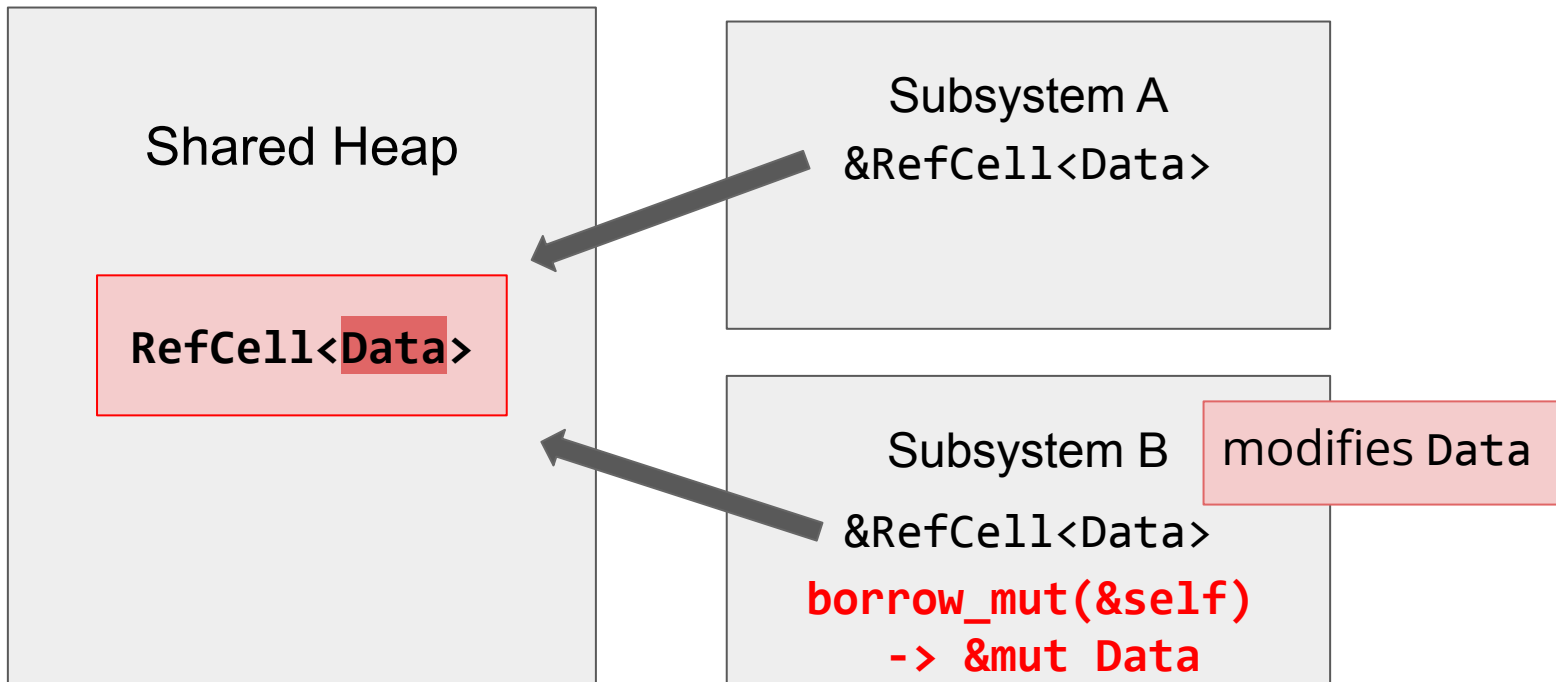
Breaking Subsystem Isolation in RedLeaf

Unsafe code in std allows mutating data behind immutable references.



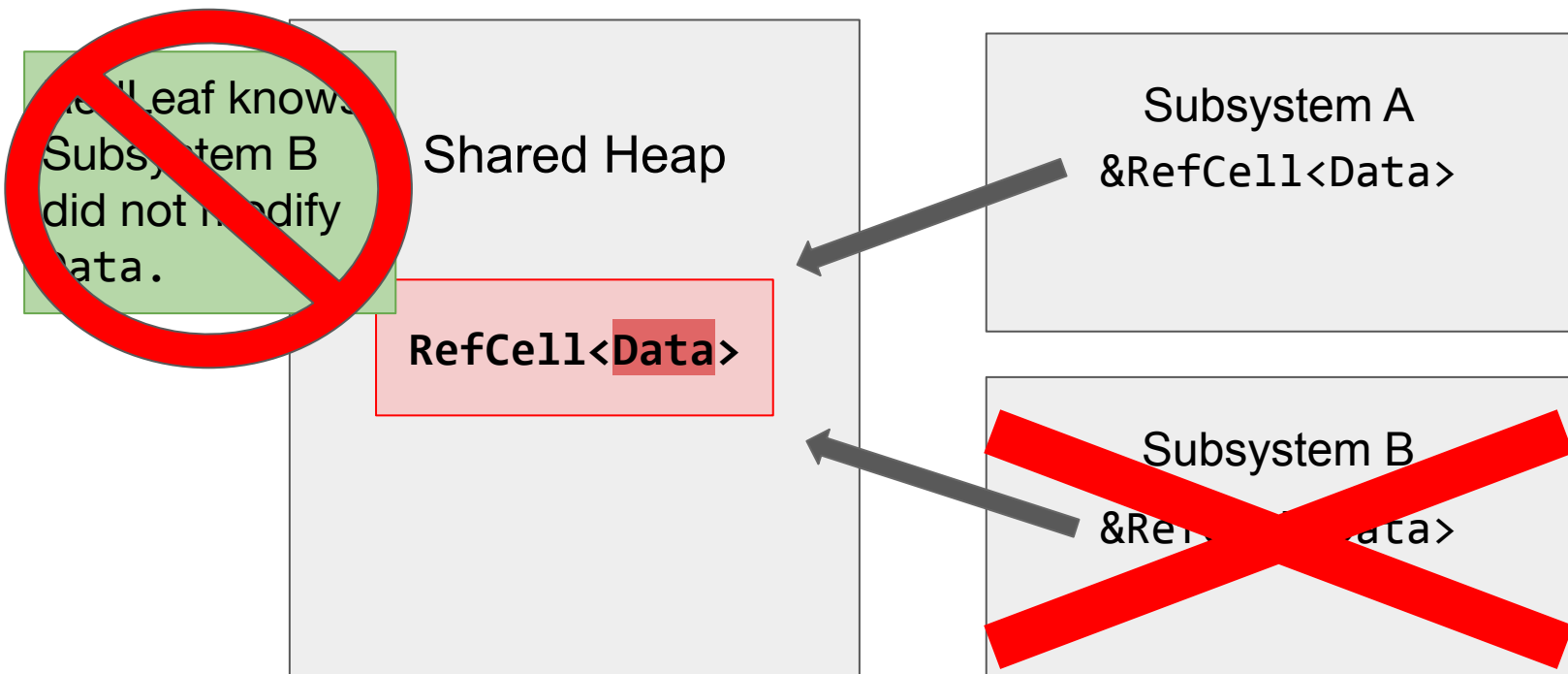
Breaking Subsystem Isolation in RedLeaf

Unsafe code in std allows mutating data behind immutable references.



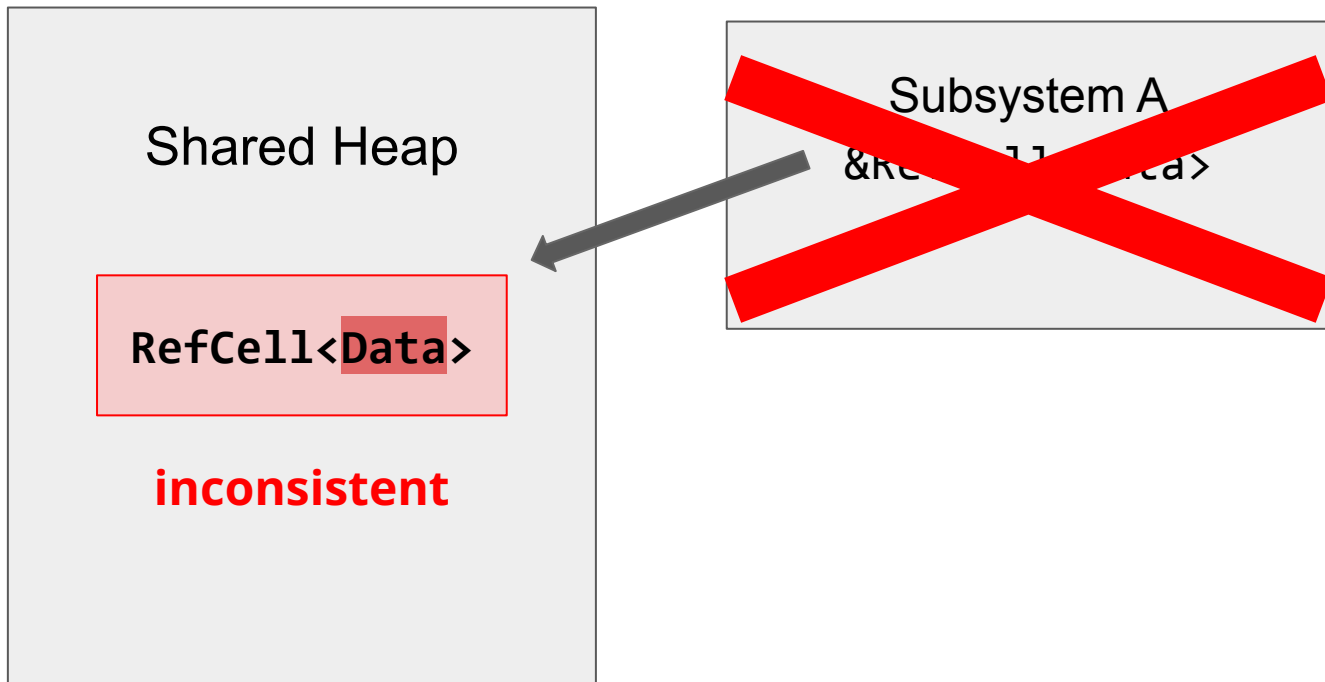
Breaking Subsystem Isolation in RedLeaf

Unsafe code in std allows mutating data behind immutable references.



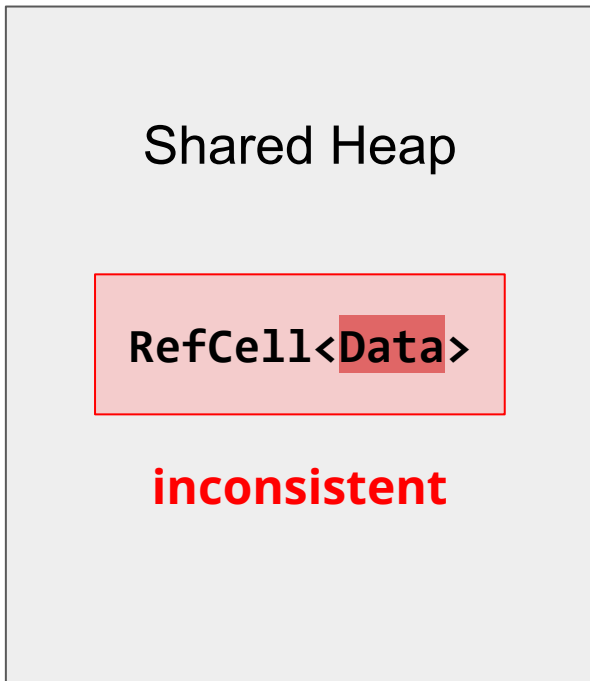
Breaking Subsystem Isolation in RedLeaf

Unsafe code in std allows mutating data behind immutable references.



Breaking Subsystem Isolation in RedLeaf

Unsafe code in std allows mutating data behind immutable references.



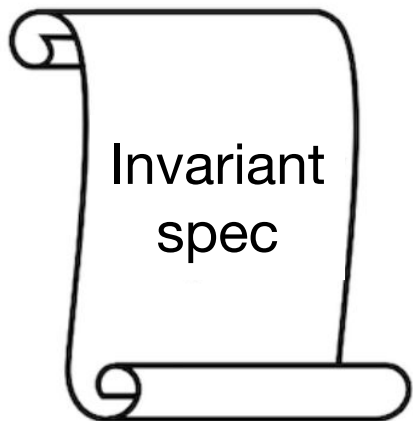
Unsafe code breaks
immutability of
references →
kernel crashes

Can we provide guarantees even in the presence of unsafe code?

Need ways to pinpoint
guarantee-breaking unsafe code

Make system invariants explicit

Which **incidental properties**, and in **what context**?



Sesame: “Encapsulation of PCon structs is not violated”

RedLeaf: “References to data on the shared heap are not mutated”

Open challenge: how to formalize and express these invariants in a way that analysis tools can use?

new research direction:

Tooling to guide
invariant-targeted auditing of
unsafe Rust code

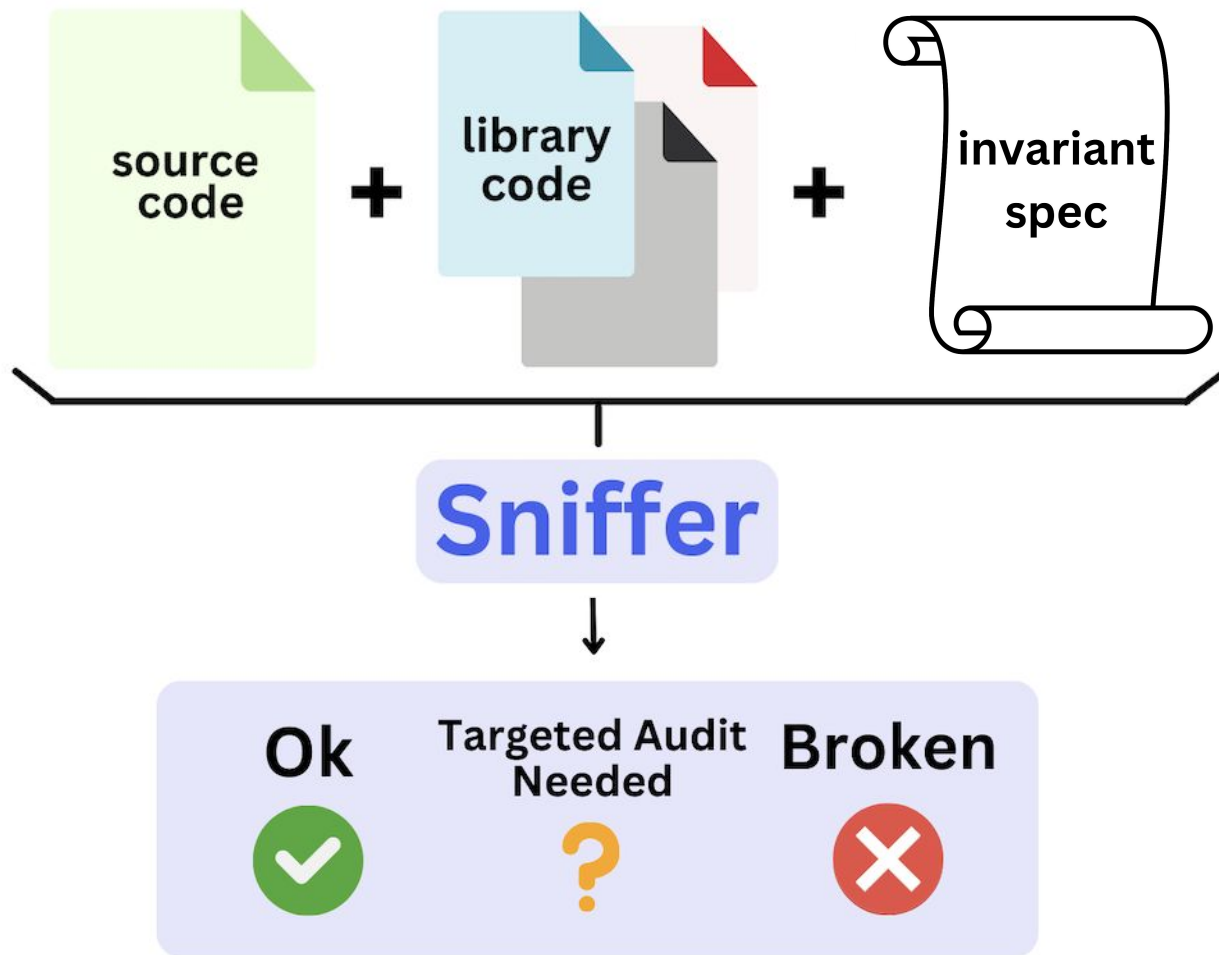
Benefits of *invariant-targeted auditing* approach

- Low specification effort
 - System designer defines invariants once; all extensions reuse the spec
- Aligned with the threat model of many Rust-based systems
 - Trust the extension developer to run an analysis tool and audit results
- Tools can use specific, scoped system invariants to filter out irrelevant code

proof-of-concept

Sniffer





Filters out irrelevant unsafe code; conservative analysis

- Use precise call graph information to skip unreachable code
- Heuristics to identify possible violations of system invariants
- Finds all possible violations, but may flag innocuous regions

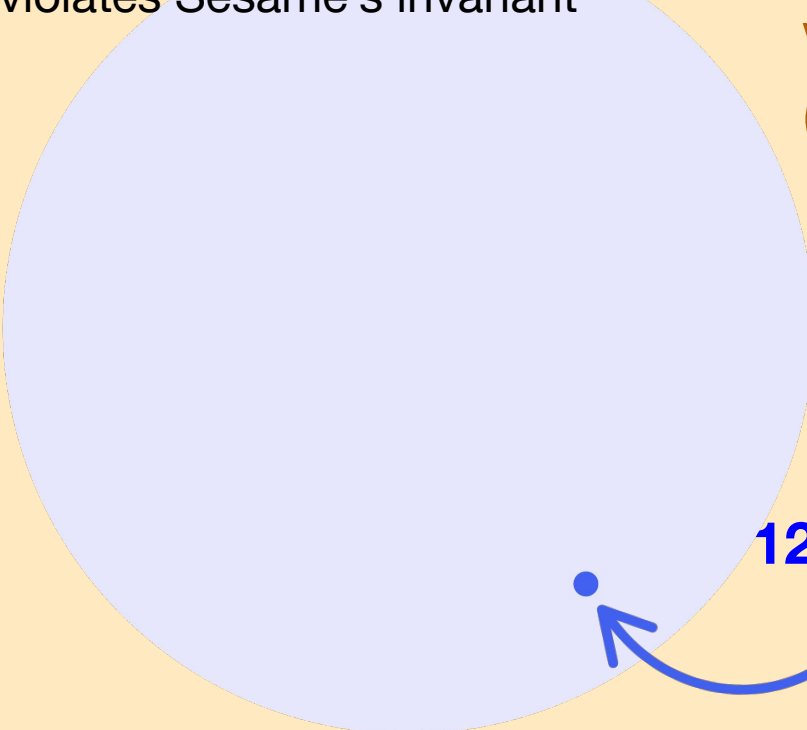
Invariant-targeted analysis reduces audit burden

Auditing a web app endpoint for unsafe code that violates Sesame's invariant

33,735 unsafe blocks
w/o reachability
(cargo-geiger)

9,982 unsafe blocks
with reachability

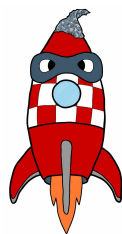
12 flagged by Sniffer
1 real violation
+ 11 false positives



Goal: enable Rust-based systems to provide guarantees over *practical* extensions that invoke unsafe code.

Call to action: let's define precise invariants for our systems.
Which incidental properties, and under what conditions?

Research direction: New tooling can help extension developers surface unsafe code that violates system invariants.



Corinn Tiffany <corinn_tiffany@alumni.brown.edu>