Red Hat Research

CS@CU

Yuga - Automatically Detecting Lifetime Annotation Bugs in the Rust Language

Vikram Nitin

Sanjay Arora

Anne Mulhern

Baishakhi Ray

The Rust Programming Language

Home / Tech / Services & Software / Operating Systems / Linux

Linus Torvalds: Rust will go into Linux 6.1

At the Kernel Maintainers Summit, the question wasn't, "Would Rust make it into Linux?" Instead, it was, "What to do about its compilers?"

News

Is Rust the Future of Programming?

Ilia Afanasiev
May 13, 2025

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

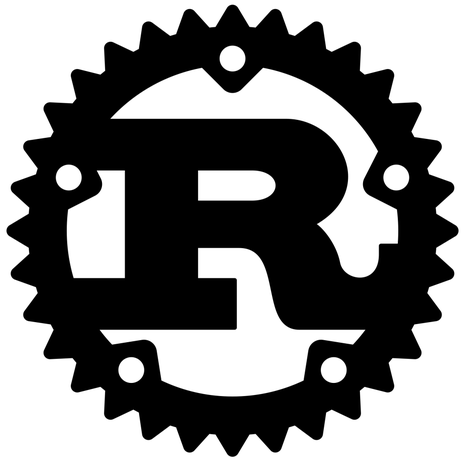
Rust in the Android platform

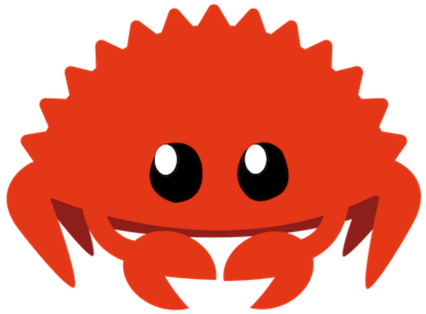
April 6, 2021

1. https://security.googleblog.com/2021/04/rust-in-android-platform.html
2. https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/

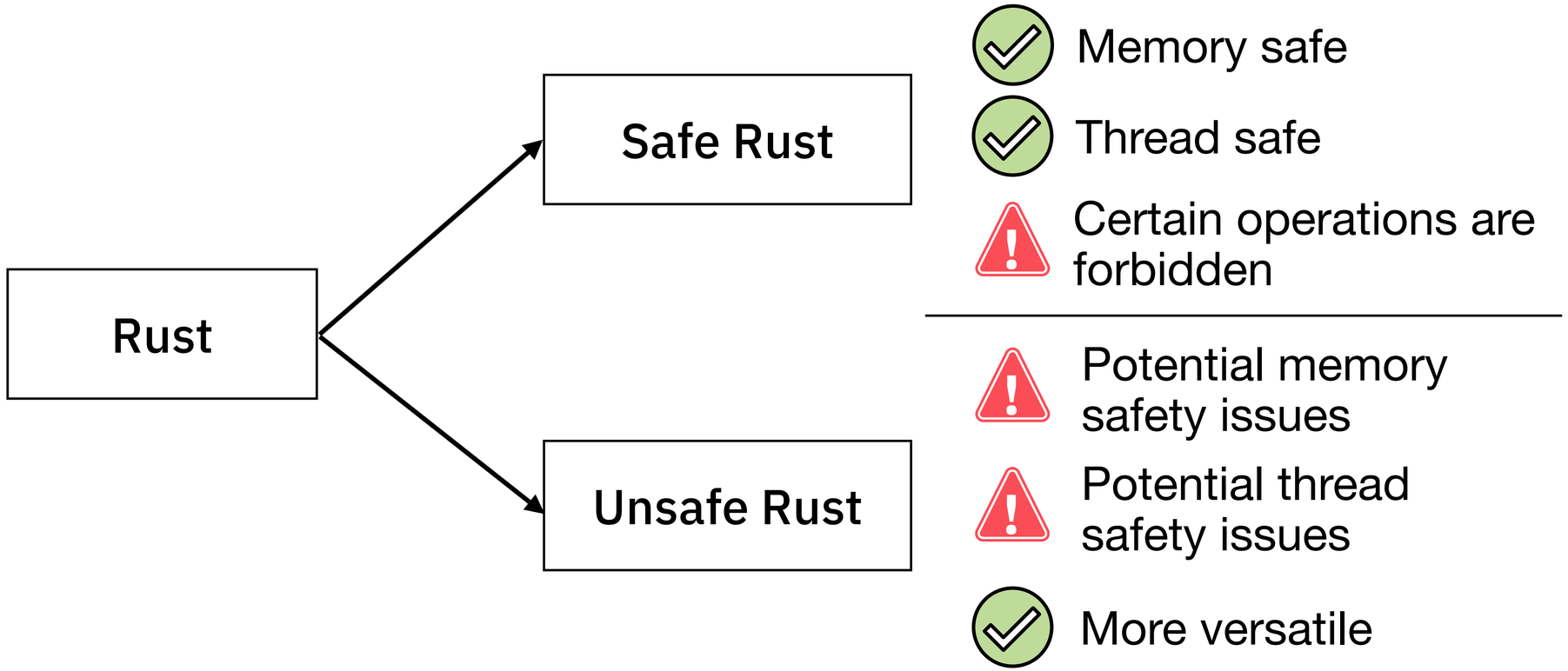
Why Rust?

- Improved **memory safety**
- Improved **thread safety**
- Expressive **type system**
- Fast **runtime performance**

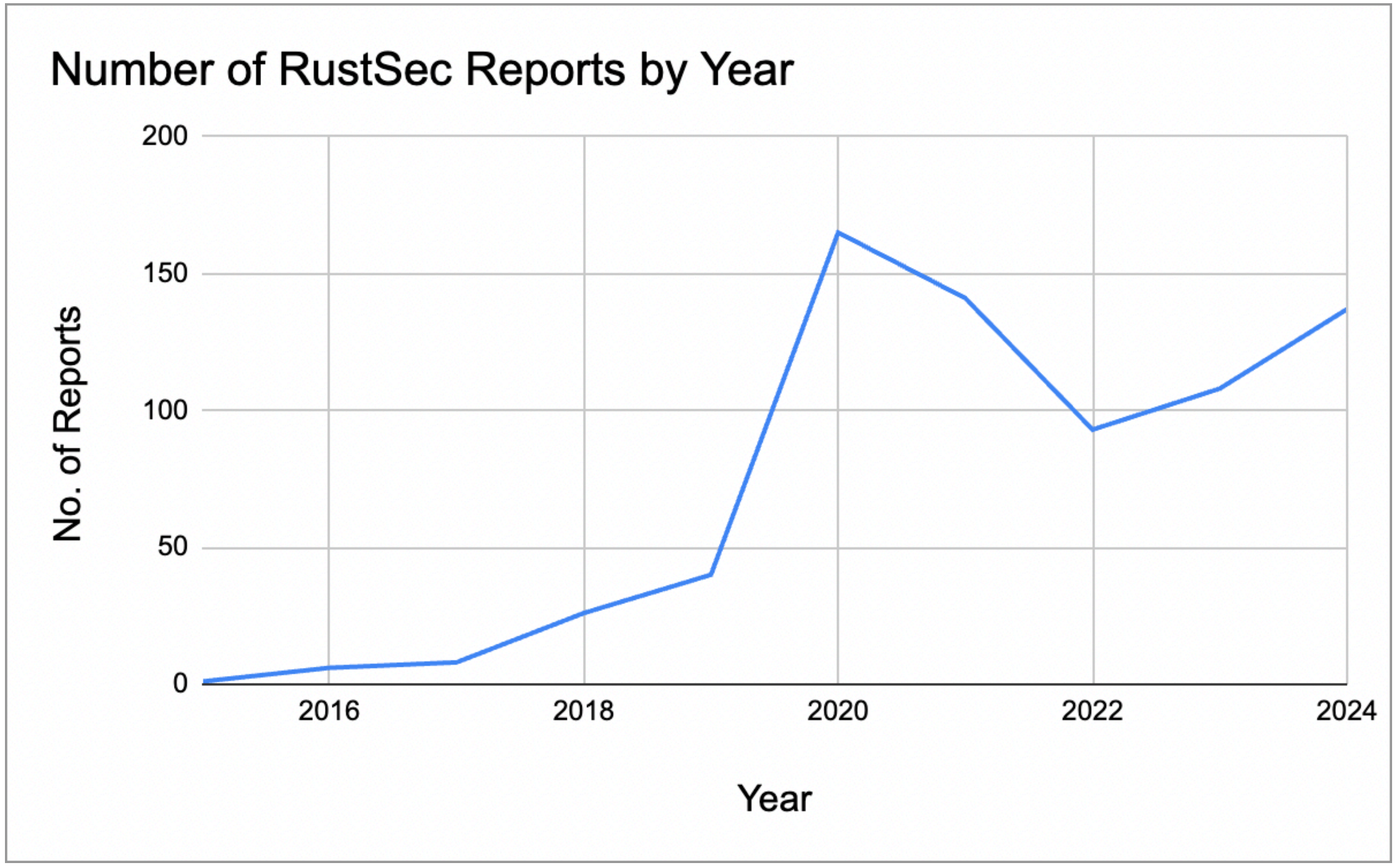




Safe and Unsafe Rust



Security Vulnerabilities in Rust



“Lifetime” Annotation Errors

RUSTSEC-2025-0016

History

Use after free in `Parc` and `Prc` due to missing lifetime constraints

Reported March 13, 2025

Issued March 22, 2025

Several vulnerabilities are attributed to incorrect **“lifetimes”**

RUSTSEC-2022-0078

Use-after-free due to a lifetime error in `Vec::into_iter()`

Reported January 14, 2022

Issued January 14, 2023 (last modified: June 13, 2023)

RUSTSEC-2021-0128

Incorrect Lifetime Bounds on Closures in `rusqlite`

Reported December 7, 2021

Issued December 9, 2021 (last modified: June 13, 2023)

Can we detect these lifetime errors?

This is a hard problem!

- These bugs surface only in execution in specific **edge cases**.
- E.g. - could involve calling multiple methods with carefully constructed arguments.
- Existing **static** and **dynamic** analysis tools completely fail to detect them.

Can we detect these lifetime errors?

Our system, **Yuga**[1], uses the following key ideas :

1 **Characterizing Lifetime Errors :** We model and define patterns by which lifetime errors can occur.

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Can we detect these lifetime errors?

Our system, **Yuga**[1], uses the following key ideas :

1. **Characterizing Lifetime Errors :** We model and define patterns by which lifetime errors can occur.

2. **Lifetime Propagation :** We decompose nested types to propagate lifetime information from annotations to internal fields, references, and pointers.

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Can we detect these lifetime errors?

Our system, **Yuga**[1], uses the following key ideas :

1. **Characterizing Lifetime Errors :** We model and define patterns by which lifetime errors can occur.

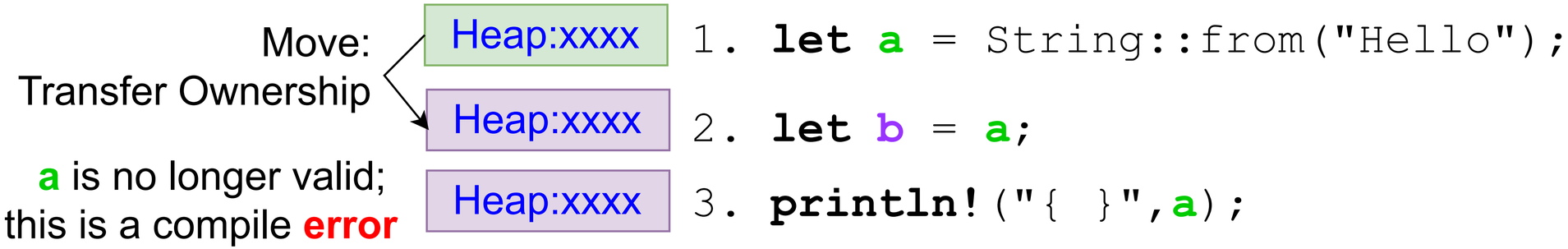
2. **Lifetime Propagation :** We decompose nested types to propagate lifetime information from annotations to internal fields, references, and pointers.

3. **Alias Analysis :** When we detect a potential violation, we implement a field and flow-sensitive alias analysis to confirm the presence of a bug.

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Warm-up : Basic Concepts in Rust

Ownership:



Red Hat Research

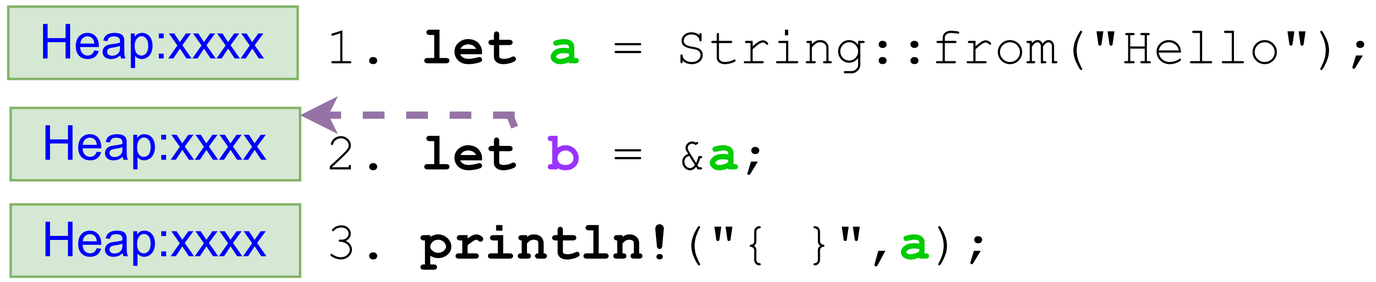
CS @CU

Warm-up : Basic Concepts in Rust

Borrowing:

b is borrowing w/o transferring ownership

a is valid; this **does** compile

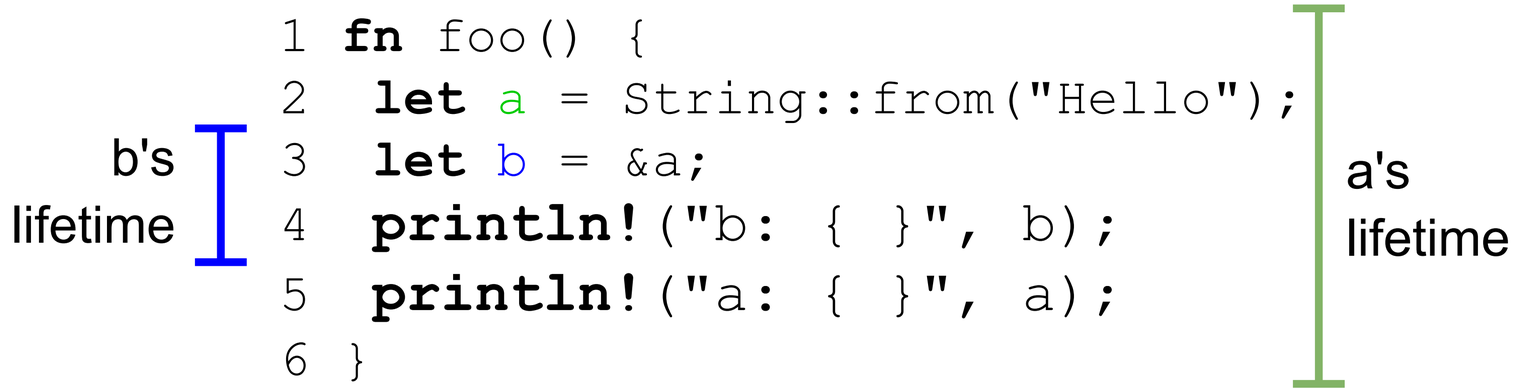


Red Hat Research

CS @CU

Warm-up : Basic Concepts in Rust

- Rust assigns an implicit **“lifetime”** to each borrow.
- Lifetime = **Region of code** in which the borrow is valid.




```
1 fn foo() {
2  let a = String::from("Hello");
3  let b = &a;
4  println!("b: { }", b);
5  println!("a: { }", a);
6 }
```


Red Hat Research

CS @CU

Warm-up : Basic Concepts in Rust

Fundamental rule of borrowing:

> A borrow cannot be valid for longer than the borrowed value.

Warm-up : Basic Concepts in Rust

Fundamental rule of borrowing:

“A borrow cannot be valid for longer than the borrowed value.”

b's lifetime

```
1 fn foo() {
2  let a = String::from("Hello");
3  let b = &a;
4  println!("b: { }", b);
5  println!("a: { }", a);
6 }
```

a's lifetime

Warm-up : Basic Concepts in Rust

Fundamental rule of borrowing:

“A borrow cannot be valid for longer than the borrowed value.”

b's lifetime

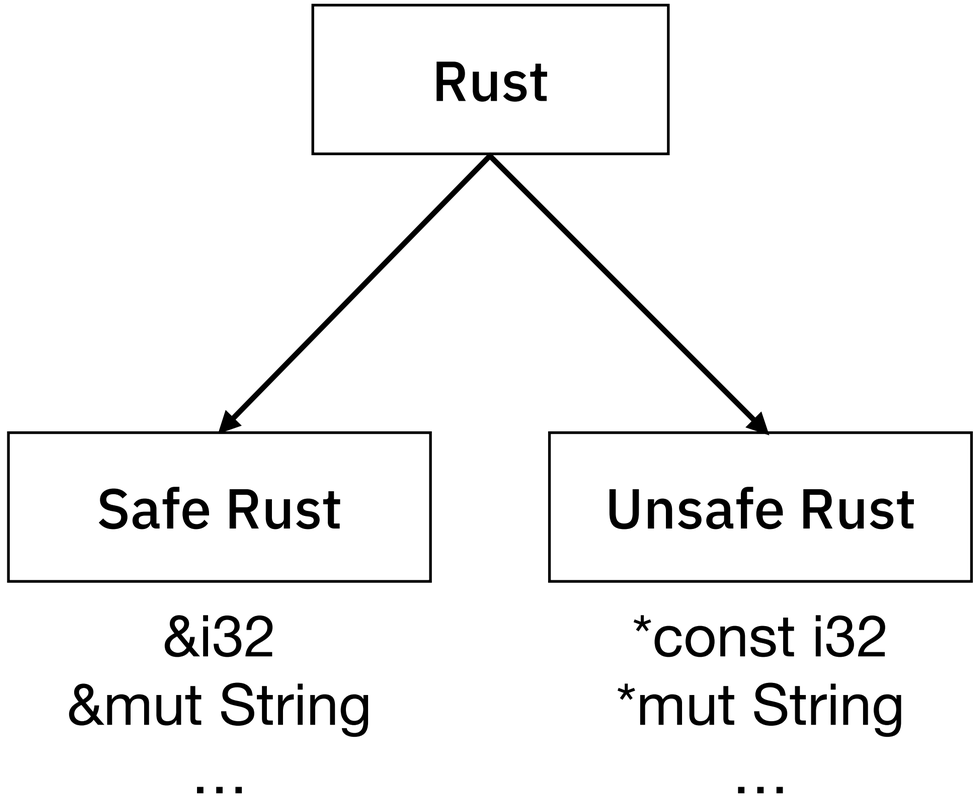
```
1   let b : &i32;
2   {    let a : i32 = 5;
3        b = &a;
4   } println!("{}", b);
```

a's lifetime

Warm-up : Basic Concepts in Rust

Raw Pointers:

- ***const**, ***mut**
- “C-style” pointers
- Rust doesn’t track lifetimes of raw pointer values



Red Hat Research

CS @CU

Lifetime Annotations

- Consider a function that **takes** a borrow and **returns** it:

```
fn foo(x: &String) -> &String {
    x   // Just return the input borrow
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations

- Consider a function that **takes** a borrow and **returns** it:

```
fn foo(x: &String) -> &String {
    x   // Just return the input borrow
}
```

- What is the relationship between the two borrow lifetimes?

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations

- Consider a function that **takes** a borrow and **returns** it:

```
fn foo(x: &String) -> &String {
    x   // Just return the input borrow
}
```

- What is the relationship between the two borrow lifetimes?
  - A. They are the **same**

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations

- Rust allows us to denote this through **annotations** on borrows:

```
fn foo<‘a>(x: &’a String) -> &’a String {
    x   // Just return the input borrow
}
```

- Annotation is part of the type

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

Lifetime annotations can only be used in:

1. Function Signatures

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations

Lifetime annotations can only be used in:

1. Function Signatures
2. Structure/ADT definitions

   struct Foo<‘a> { x: &’a i32 }

Tells the compiler that Foo<‘a> contains a borrow with lifetime ‘a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- **Checkpoint:** What does the type &’b Foo<‘a> mean?

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- **Checkpoint:** What does the type &’b Foo<‘a> mean?
  - **Ans.** A borrow with lifetime ‘b, of a structure object containing a borrow with lifetime ‘a. Example:

    struct Foo<‘a> { x: &’a String }

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- What about a function with two borrow arguments?

```
struct Foo<‘a> { x: &’a String }

fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<_?_> {
    Foo {x}
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- What about a function with two borrow arguments?

```
struct Foo<‘a> { x: &’a String }

fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘a> {
    Foo {x}
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- What about a function with two borrow arguments?

```
struct Foo<‘a> { x: &’a String }

fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b> {
    Foo {y}
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- Body of the function has to be consistent with the type signature:

```
fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b> {
    Foo {x}
}
```

Compile Error

Lifetimes annotations are **descriptive**, not **prescriptive.**

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- Hypothetical: What if Rust allowed this code to compile?

  That is, what if the lifetime annotations were **prescriptive**?

```
fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b>
{
    Foo {x}
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- Hypothetical: What if the lifetime annotations were **prescriptive**?

```
fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b>
{
    Foo {x}
}
```

```
fn main( ) {
    let s1 = “Hello”.to_string( );
    let s2 = “World”.to_string( );
    let ret = foo(&s1, &s2);
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- Hypothetical: What if the lifetime annotations were **prescriptive**?

```
fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b>
{
    Foo {x}
}
```

```
fn main( ) {
    let s1 = “Hello”.to_string( );
    let s2 = “World”.to_string( );
‘a  let ret = foo(&s1, &s2);
    drop(s1); // !!
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations

- Hypothetical: What if the lifetime annotations were **prescriptive**?

```
fn foo<‘a, ‘b>(x: &’a String, y: &’b String) -> Foo<‘b>
{
    Foo {x}
}
```

```
fn main( ) {
    let s1 = “Hello”.to_string( );
    let s2 = “World”.to_string( );
    let ret = foo(&s1, &s2);
    drop(s1); // !!!!!!!
    println!(“{:?}”, ret.x); // Undefined behavior
}
```

‘b ‘a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE’24

Lifetime Annotations in Unsafe Rust

- Common design pattern in unsafe Rust - use **‘proxy’ lifetimes**.

```
struct Foo<‘a> {
   x: *const String, ...
}
fn bar(x: &’a String) -> Foo<‘a> {
   Foo{ x, ... }
}
```

Foo<‘a> contains a raw pointer to a String with lifetime ‘a

Lifetime Annotations in Unsafe Rust

- In **unsafe** Rust, lifetime annotations can be **prescriptive**!

```
struct Foo<'a> {
    x: *const String,
    ..
}
fn bar<'a,'b>(arg1: &'a String, arg2: &'b String)
        -> Foo<'a> {
    Foo{x: arg2 as *const String, ..}
}
```

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations in Unsafe Rust

- In **unsafe** Rust, lifetime annotations can be **prescriptive**!

```
struct Foo<'a> {
    x: *const String,
    ..
}
fn bar<'a,'b>(arg1: &'a String, arg2: &'b String)
        -> Foo<'a> {
    Foo{x: arg2 as *const String, ..}
}
```

```
1  let v1 = "Hello".to_string();
2  let v2 = "World".to_string();
3  let bar_obj = bar(&v1, &v2);
4  drop(v2);
5  // Code that uses bar_obj
```

bar_obj is used till here

v2 is borrowed only till here

Code compiles, but potential memory safety error!

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Lifetime Annotations in Unsafe Rust

- In **safe** Rust, an incorrect lifetime annotation is a **compile error**.
- In **unsafe** Rust, an incorrect lifetime annotation can be a **memory safety error**!

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Our System - Step 1

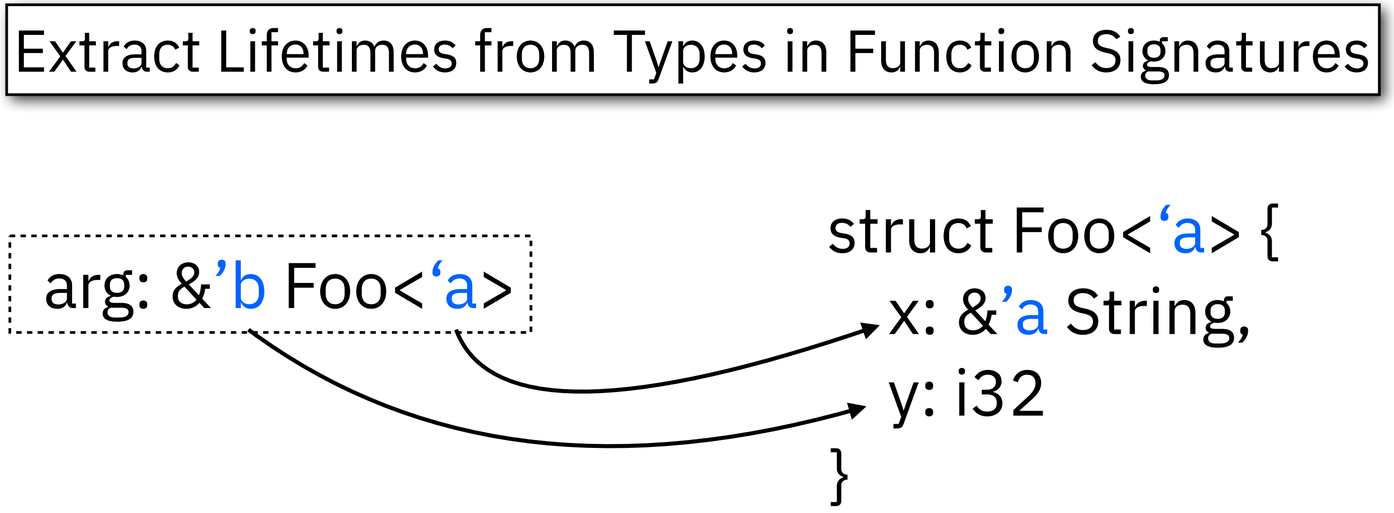
Extract Lifetimes from Types in Function Signatures

arg: &’b Foo<‘a>

Red Hat Research

CS @CU

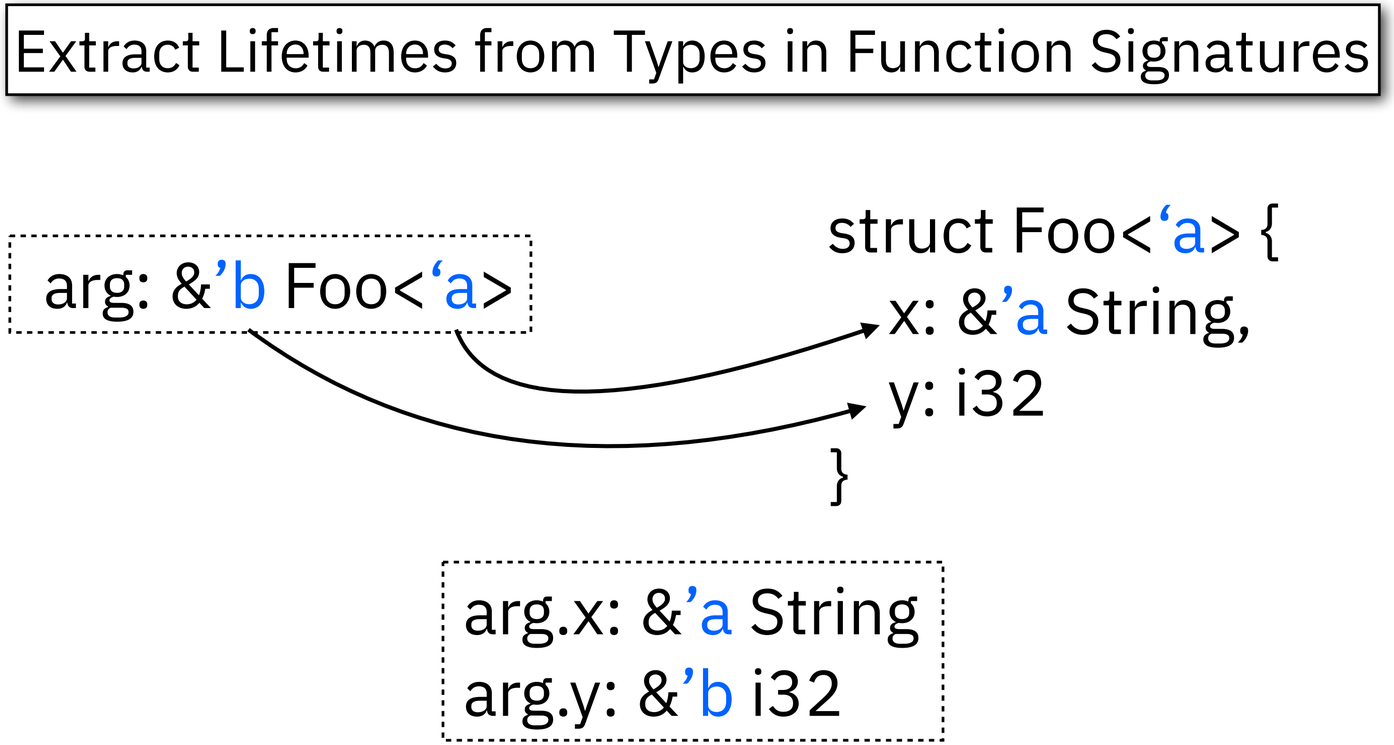
Our System - Step 1



Red Hat Research

CS @CU

Our System - Step 1



Our System - Step 1

```
struct Foo<'a> { x: *mut String,
                 w: &'a mut i32 }
struct Bar { y: String,
             z: *mut i32 }

fn bar<'a,'b>(arg1: &'a mut i32,
              arg2: &'b mut Bar)
        -> Foo<'a> {
    let ret = Foo{ x: &mut (*arg2).y,
                   w: arg1 };
    ret
}
```

Value	Type	Borrowed For
arg1	&'a mut i32	None
*arg1	i32	'a
arg2	&'b Bar	None
*arg2	Bar	'b
(*arg2).y	String	'b
(*arg2).z	*mut i32	'b
*(*arg2).z	i32	'b
ret	Foo<'a>	None
ret.w	&'a mut i32	None
*(ret.w)	i32	'a
ret.x	*mut String	None
*(ret.x)	String	'a

Our System - Step 1

We have developed a system of **typing rules** to describe this (details in paper).

$$\frac{}{O \to O : \epsilon}\ \texttt{contains-self} \qquad \frac{T \to T_1 : \epsilon}{\&LT \to T_1 : L}\ \texttt{borrow} \qquad \frac{T \to T_1 : \epsilon}{\&L\ \texttt{mut}\ T \to T_1 : L}\ \texttt{mut-borrow}$$

$$\frac{O\{T\} \quad T \to T_1 : L}{O \to T_1 : L}\ \texttt{field} \qquad \frac{O\{T\} \quad T \to T_1 : \epsilon}{S \to T_1 : \epsilon}\ \texttt{field-eps} \qquad \frac{T \to T_1 : L_1}{\&LT \to T_1 : L_1}\ \texttt{inner-lifetime}$$

$$\frac{S\{\star\texttt{const}\ T\}}{S \to T : \epsilon}\ \texttt{raw-owned} \qquad \frac{S\{\star\texttt{mut}\ T\}}{S \to T : \epsilon}\ \texttt{raw-mut-owned} \qquad \frac{S\texttt{<}L\texttt{>}\{\star\texttt{const}\ T\}}{S\texttt{<}L\texttt{>} \to T : L}\ \texttt{raw-lifetime}$$

$$\frac{S\texttt{<}L\texttt{>}\{\star\texttt{mut}\ T\}}{S\texttt{<}L\texttt{>} \to T : L}\ \texttt{raw-mut-lifetime} \qquad \frac{T \to T_1 : L_1}{\&LT \to L_1 : L}\ \texttt{B-inner-longer} \qquad \frac{}{T \to L :\ L}\ \texttt{B-reflexive}$$

$$\frac{}{T \to \texttt{'static} :\ L}\ \texttt{B-static} \qquad \frac{T \to T_1 : L \quad T_1 \to L_1 :\ L_2}{T \to L_1 :\ L_2}\ \texttt{B-extract-inner1}$$

$$\frac{T \to T_1 : \epsilon \quad T_1 \to L_1 :\ L_2}{T \to L_1 :\ L_2}\ \texttt{B-extract-inner2}$$

Our System - Step 2

Identify Potential Memory Safety Errors

A memory safety error could occur when a value with **one lifetime** gets assigned to a type with a **different lifetime**.

Value	Type	Borrowed For
arg1	&'a **mut** i32	None
*arg1	i32	'a
arg2	&'b Bar	None
*arg2	Bar	'b
(*arg2).y	String	'b
(*arg2).z	*mut i32	'b
*(*arg2).z	i32	'b
ret	Foo<'a>	None
ret.w	&'a mut i32	None
*(ret.w)	i32	'a
ret.x	*mut String	None
*(ret.x)	String	'a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Our System - Step 2

Identify Potential Memory Safety Errors

Look for values with:

- matching type, and
- different borrow lifetimes

Value	Type	Borrowed For
arg1	&'a **mut** i32	None
*arg1	i32	'a
arg2	&'b Bar	None
*arg2	Bar	'b
(*arg2).y	String	'b
(*arg2).z	*mut i32	'b
*(*arg2).z	i32	'b
ret	Foo<'a>	None
ret.w	&'a mut i32	None
*(ret.w)	i32	'a
ret.x	*mut String	None
*(ret.x)	String	'a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Our System - Step 3

Alias Analysis

For each potential violation pair, analyze the **function body** to see if they **alias**.

Value	Type	Borrowed For
arg1	&'a **mut** i32	None
*arg1	i32	'a
arg2	&'b Bar	None
*arg2	Bar	'b
(*arg2).y	String	'b
(*arg2).z	*mut i32	'b
*(*arg2).z	i32	'b
ret	Foo<'a>	None
ret.w	&'a mut i32	None
*(ret.w)	i32	'a
ret.x	*mut String	None
*(ret.x)	String	'a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Our System - Step 3

Alias Analysis

```
struct Foo<'a> { x: *mut String,
                 w: &'a mut i32 }
struct Bar { y: String,
             z: *mut i32 }

fn bar<'a,'b>(arg1: &'a mut i32,
              arg2: &'b mut Bar)
        -> Foo<'a> {
    let ret = Foo{ x: &mut (*arg2).y,
                   w: arg1 };
    ret
}
```

Value	Type	Borrowed For
arg1	&'a mut i32	None
*arg1	i32	'a
arg2	&'b Bar	None
*arg2	Bar	'b
(*arg2).y	String	'b
(*arg2).z	*mut i32	'b
*(*arg2).z	i32	'b
ret	Foo<'a>	None
ret.w	&'a mut i32	None
*(ret.w)	i32	'a
ret.x	*mut String	None
*(ret.x)	String	'a

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Our System - Step 4

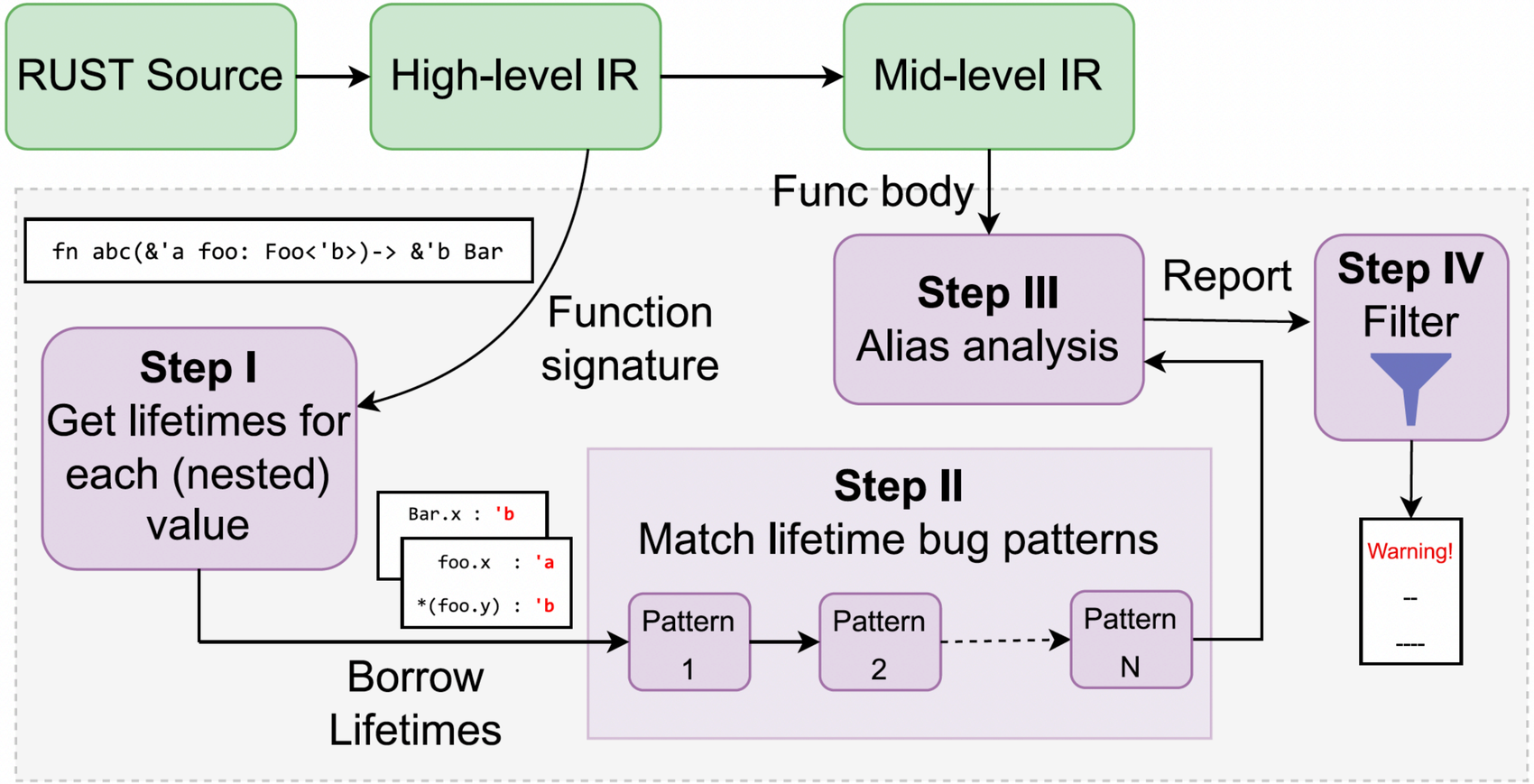
Shallow Filter based on inter-procedural knowledge

Heuristics to filter out common false positives:

- If a structure containing a raw pointer is marked as a violation, only report it if it has a custom **Drop** implementation which could free the pointer's memory.
- Etc…

[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Yuga[1] - Detecting Lifetime Bugs in Rust



[1] Nitin et al. (2023) - *Yuga: Automatically detecting lifetime annotation bugs in the Rust language.* TSE'24

Research Questions

RQ-1 Can Yuga detect known vulnerability patterns?

RQ-2 Can Yuga discover new vulnerabilities in Rust projects?

RQ-3 Can existing tools detect lifetime annotation bugs?

Research Questions

RQ-1 Can Yuga detect known vulnerability patterns?

RQ-2 Can Yuga discover new vulnerabilities in Rust projects?

RQ-3 Can existing tools detect lifetime annotation bugs?

RQ-1 Running Yuga on Known Vulnerabilities

We would like to test whether Yuga can detect lifetime annotations across a diverse set of patterns and project.

Evaluation

- *RustSec Vulnerability Reports*. We pick out **9** where the root cause is incorrect lifetime annotations. These contain a total of **23** functions with bugs.

RQ-1 Running Yuga on Known Vulnerabilities

We would like to test whether Yuga can detect lifetime annotations across a diverse set of patterns and project.

Evaluation

- *RustSec Vulnerability Reports*. We pick out **9** where the root cause is incorrect lifetime annotations. These contain a total of **23** functions with bugs.

- *Synthetic Bug Database:*
  - For each of the 9 projects, we perform program slicing to extract one function containing a minimal version of the bug pattern.
  - Then, we apply transformations to get **27** synthetic functions with bugs.

RQ-1 Running Yuga on Known Vulnerabilities

Results of running Yuga on 9 Rust projects with known vulnerabilities.

- **Precision:** 16/18 ~ **87.5%**
- **Recall:** 16/23 ~ **60.9%**

RustSec ID	#Vulnerable Funcs	#Detected	#FP	#FN
2018-0020	1 + 2*	2	0	1
2020-0023	2	2	0	0
2020-0060	1	0	0	1
2021-0128	7	7	2	2
2021-0130	2 + 3*	5	0	0
2022-0028	2	0	0	2
2022-0040	1	0	0	1
2022-0070	1	0	0	1
2022-0078	1	0	0	1
Total	**23**	**16**	**2**	**9**

RQ-1 Running Yuga on Synthesized Bugs

- Yuga detects **16** out of **27** ~ **59.3%** recall
- No false positives here, because our synthetic dataset contains only buggy functions.

Bug	Detected	Bug	Detected	Bug	Detected
1.1	No	4.1	Yes	7.1	No
1.2	No	4.2	Yes	7.2	No
1.3	No	4.3	No	7.3	No
2.1	Yes	5.1	Yes	8.1	No
2.2	Yes	5.2	Yes	8.2	Yes
2.3	Yes	5.3	Yes	8.3	Yes
3.1	Yes	6.1	Yes	9.1	Yes
3.2	Yes	6.2	Yes	9.2	Yes
3.3	No	6.3	No	9.3	No

Summary

Yuga is able to detect bugs with good precision and recall.

Research Questions

RQ-1 Can Yuga detect known vulnerability patterns?

RQ-2 Can Yuga discover new vulnerabilities in Rust projects?

RQ-3 Can existing tools detect lifetime annotation bugs?

RQ-2 Running Yuga on Rust Packages

Results of running Yuga on 372 Rust packages ("crates") [1]

Category	Sub-category	#
Exploitable bugs		**3**
Code smells	Freed ptr but no deref	14
	User-implemented ref counting	30
	Memory copy, not alias	21
	Different field of struct	16
	Two lifetime annotations	1
Total		**85**

[1] The top 2000 Rust crates by no. of downloads, filtered to include only crates that use unsafe code and lifetime annotations.

RQ-2 Running Yuga on Rust Packages

A new exploitable vulnerability that Yuga uncovered in the **cslice** crate.

```
pub struct CMutSlice<'a, T> {
  base: *mut T, ...
}
pub fn as_mut_slice(&mut self) -> &'a mut [T]
{   unsafe { slice::from_raw_parts_mut(
              self.base, self.len)} }
```

Yuga is able to uncover multiple exploitable vulnerabilities and code smells. This highlights the importance of this problem, and the effectiveness of Yuga.

Research Questions

RQ-1 Can Yuga detect known vulnerability patterns?

RQ-2 Can Yuga discover new vulnerabilities in Rust projects?

RQ-3 Can existing tools detect lifetime annotation bugs?

RQ-3 Existing Tools

We compare with Rudra (static), MirChecker (static) and Miri (dynamic).

Tool	# Bugs	Tool detected	YUGA detected
Rudra	50	0	30
MirChecker	27	0	16
Miri (w/o exploit code)	27	0	16
Miri (w. exploit code)	27	27	16

RQ-3 Existing Tools

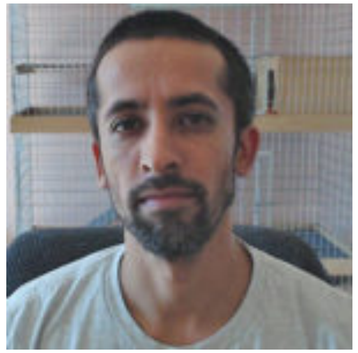
We compare with Rudra (static), MirChecker (static) and Miri (dynamic).

Tool	# Bugs	Tool detected	YUGA detected
Rudra	50	0	30
MirChecker	27	0	16
Miri (w/o exploit code)	27	0	16
Miri (w. exploit code)	27	27	16

Existing tools are unsuited for this problem. Existing static and dynamic tools are completely unable to detect these bugs. Miri, a dynamic analyzer, raises errors only when run with the exploit code.

Acknowledgements

Co-authors:



Sanjay Arora

Anne Mulhern

Baishakhi Ray

Funding and Travel Support:

Red Hat
Research

Thank You!