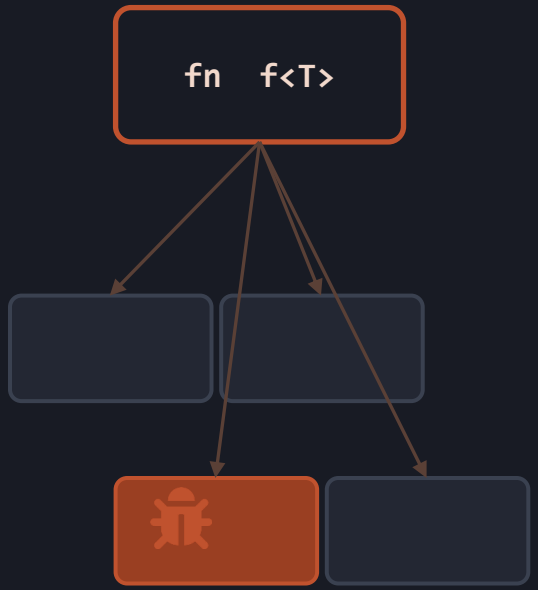
TOSEM · 2025

RuMono

RuMono: Fuzz Driver Synthesis for Rust Generic APIs

Seminar Presentation

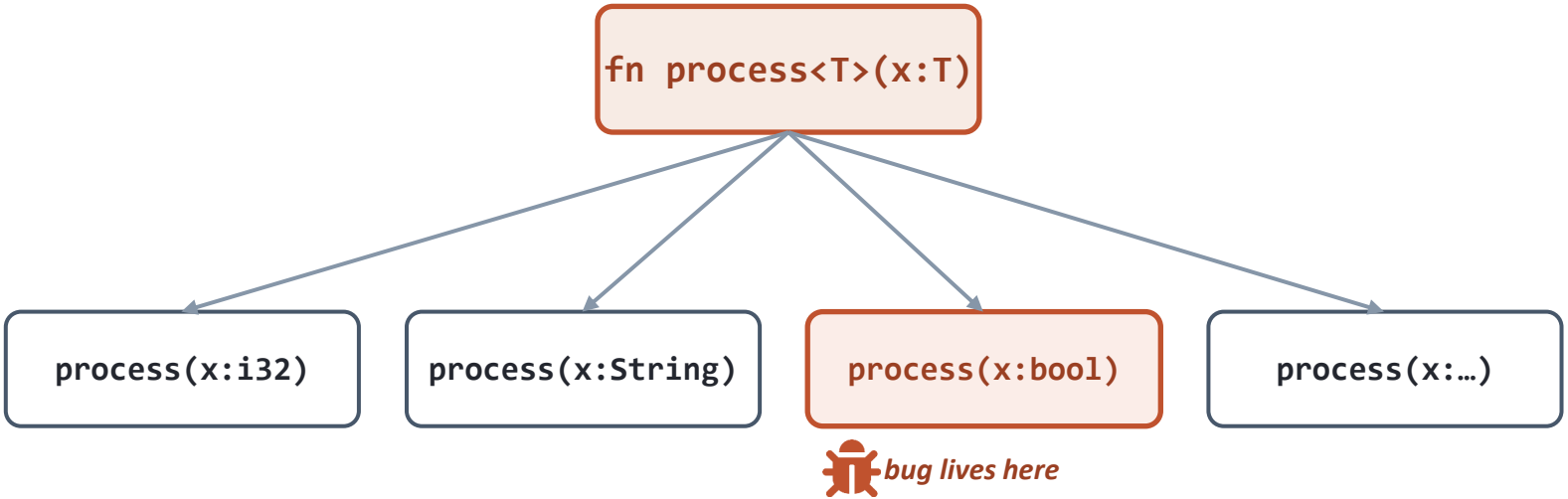


Yehong Zhang, Jun Wu, and Hui Xu— School of Computer Science, Fudan University

How do we test vulnerabilities in libraries?

How do we test for generic functions?

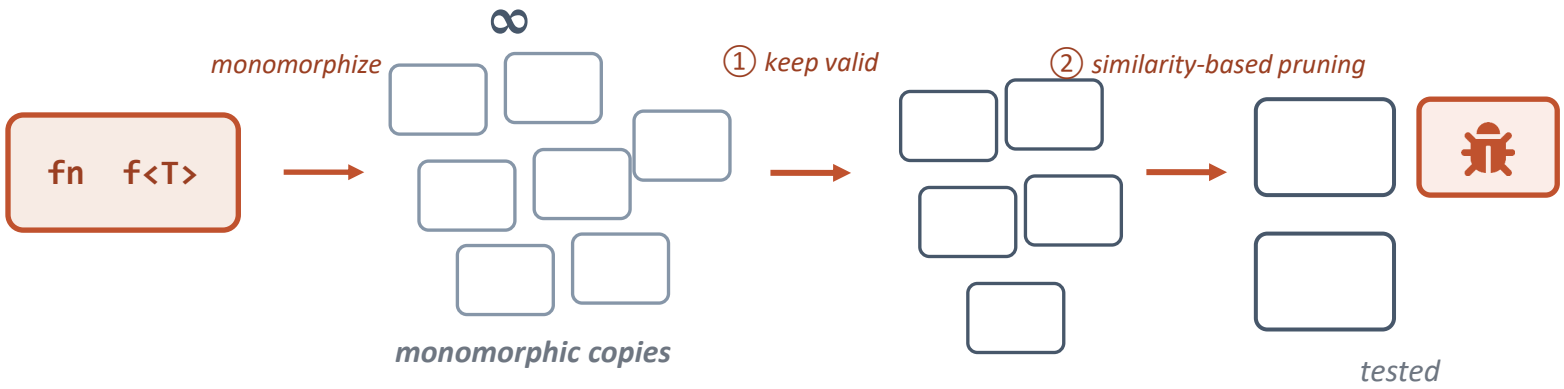
Generic Functions



The compiler makes a separate copy for each type. (Monomorphization)

THE KEY IDEA

Generate the possible copies — then test only the different ones

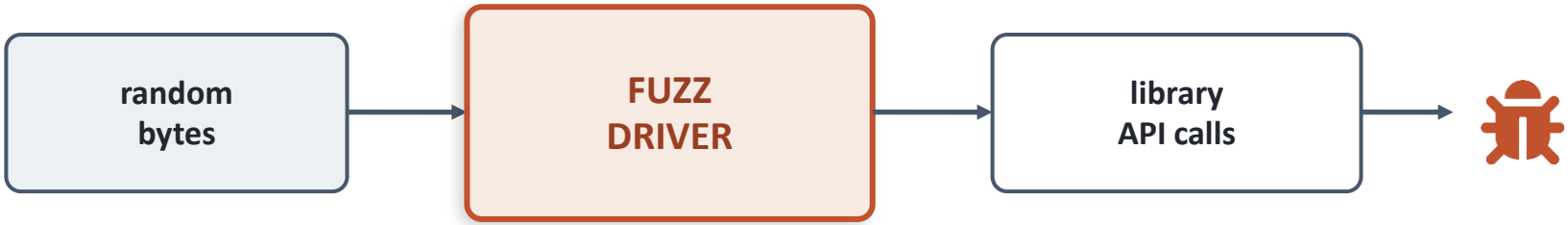


Valid = the type is actually allowed

Different = different behavior (runs different code)

Fuzz drivers

A library can't run by itself — it needs a tiny program (harness) to call it.



Writing these by hand doesn't scale → tools synthesize them. That is the task.

Generics, traits, and trait bounds

Type parameter T

One function that works for many types.

Trait

An interface: a set of abilities a type can promise.

Trait bound T: PartialOrd

A requirement: T must be orderable (comparable).

```rust
use std::cmp::PartialOrd;
fn largest<T:PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item.gt(largest) {
            largest = item;
        }
    }
    largest
}
```

(a) Rust

```cpp
concept Comparable = requires(T a, T b){
    {a < b} -> std::convertible_to<bool>;
};
template <Comparable T>
const T& largest(const std::vector<T>& list){
    size_t largest_index = 0;
    for (size_t i = 1; i < list.size(); ++i)
        if (list[i] > list[largest_index])
            largest_index = i;
    return list[largest_index];
}
```

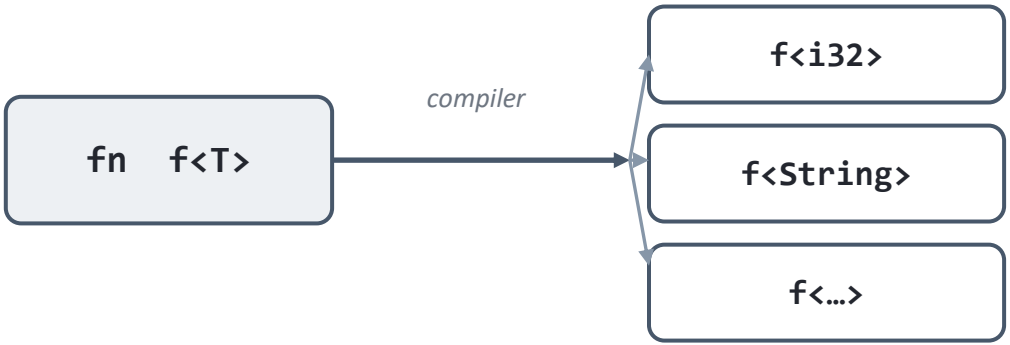
(b) C++

```java
public static <T extends Comparable<T>>
T largest(T[] list) {
    T largest = list[0];
     for (T item: list) {
        if (item.compareTo(largest) > 0) {
            largest = item;
        }
    }
    return largest;
}
```

(c) Java

Monomorphization

Generic functions result in multiple copies of the same function for a concrete type



A bug can hide in just ONE copy

However T can be anything so we can have infinite monomorphized functions to test: Vec<T> → Vec<Vec<T>> → …

How do existing works test libraries?

API dependency graph

Generate a graph by tracking input types and output types to track a sequence of APIs to automate generation of fuzzing harnesses

THE GAP

What happens with generic types?

Which type fits?

A type must satisfy the bounds AND be buildable by some other function.

Too many copies

One generic function can spawn endless variants — you can't fuzz them all.

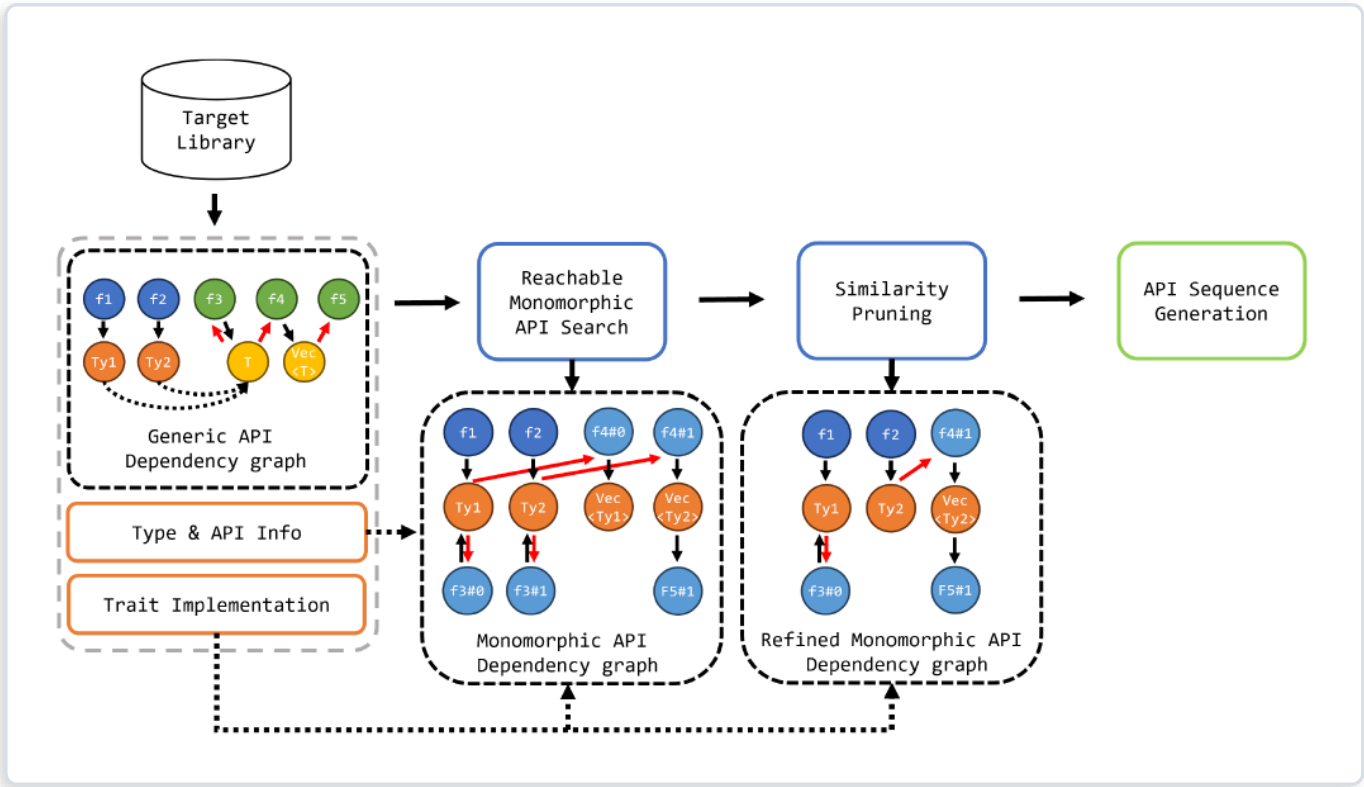
Prior tools either ignored generics, or handled them partially.

PRIOR WORK

How existing works handled generics

RULF	No generics at all.
RPG	Only provide generic support based on a parameter provider instantiated from the tested library
RVFuzz	Only utilize existing concrete types from library
RuMono	**Tracks valid monomorphized functions and prunes similar ones to test**

RuMono's plan: two stages

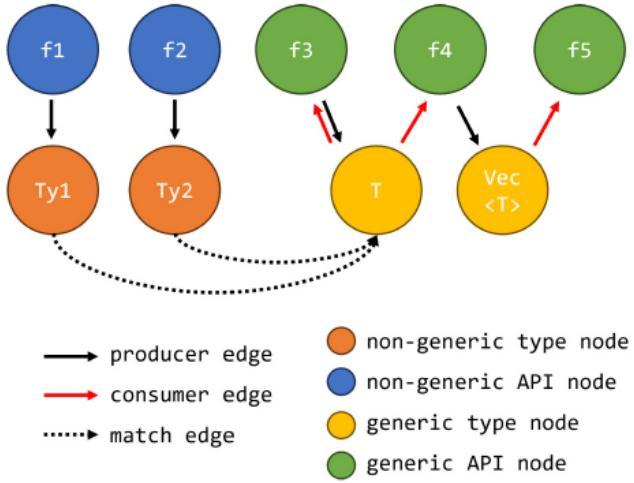


Generic API Dependency Graph

Updated API Dependency Graph to include generics

```
1  struct Ty1;
2  struct Ty2;
3  trait A{}
4  trait B{}
5  impl A for Ty1 {}
6  impl A for Ty2 {}
7  impl B for Ty2 {}
8  fn f1() -> Ty1;
9  fn f2() -> Ty2;
10 fn f3<T:A>(a1:T)->T;
11 fn f4<T> (a1:T)->Vec<T>;
12 fn f5<T:A+B> (a1:Vec<T>);
```

(a) Source code for exp lib.



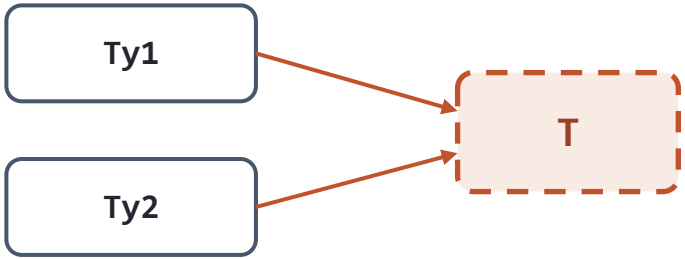
(b) Generic API dependency graph.

Fig. 2 — match edges (dotted) bridge concrete types to the abstract slot.

Matching Concrete Types with Generics

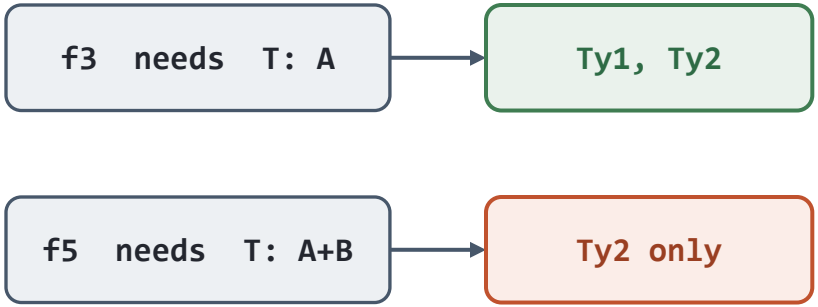
GRAPH · who could fill T?

(permissive — includes everything)



BOUNDS · who's actually allowed?

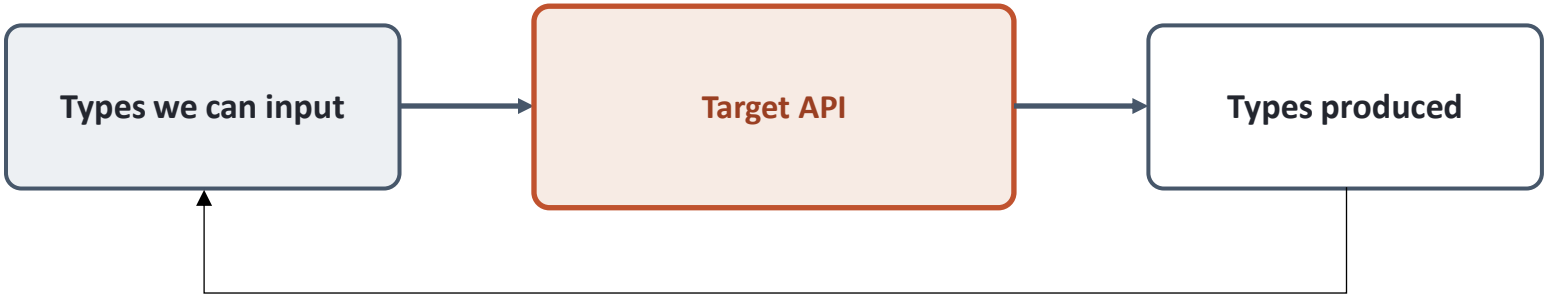
(a separate check, per function)



only Ty2 implements B

Stage 1 — Identify reachable APIs

An API is said to be reachable if each of its arguments are primitive or can be produced by another reachable API



return types feed back in

Stage 2 — Similarity-based Pruning

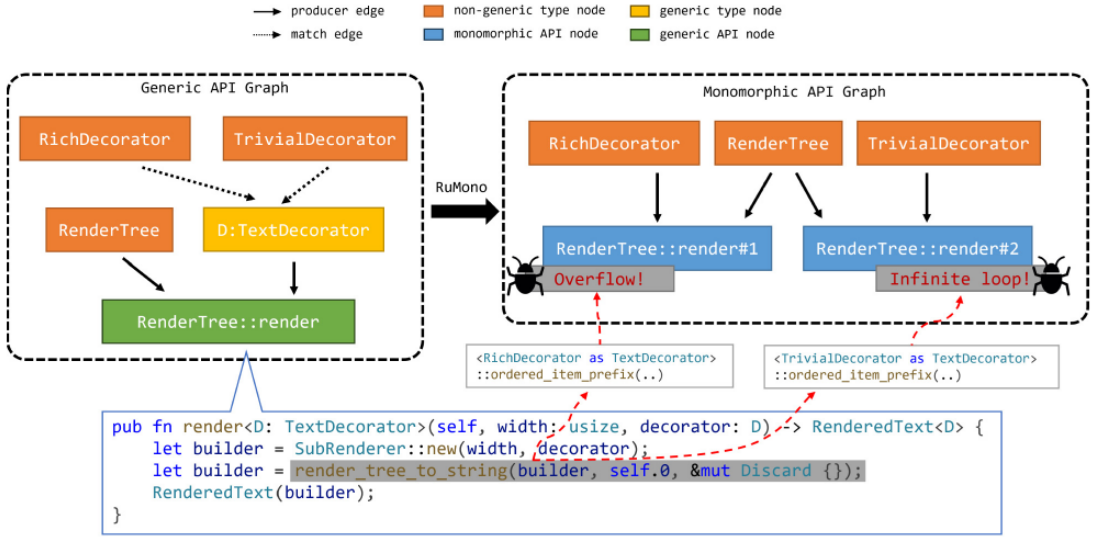
Monomorphic API Explosion: Number of valid monomorphic APIS can be large or even infinite

```rust
struct A;
struct B;
trait Foo{ // Foo provide a default implementation for method foo()
    fn foo()->&'static str { "default impl for Foo" }
}
trait Bar{
    fn bar()->&'static str;
}
// Situation 1: similar behavior by default implementation
// Both A and B do not provide the implementation for foo()
impl Foo for A {}
impl Foo for B {}
// Situation 2: similar behavior by blanket implementation
impl<T:Foo> Bar for T{
    // This implements bar() for both A and B
    fn bar() -> &'static str { "blanket impl for Bar" }
}
fn main(){
    assert_eq!(A::foo(), B::foo());
    assert_eq!(A::bar(), B::bar());
}
```

Fig. 5 — A and B share Foo's default impl and one blanket Bar impl → identical behavior, so test just one.

Case Study

One function, two different bugs



(b) The reason for two bugs and how our approach discovers two bugs.

RuMono tests both copies and finds both bugs — generics-blind tools find at most one.

RESULTS · RQ1 — VALIDITY & COVERAGE

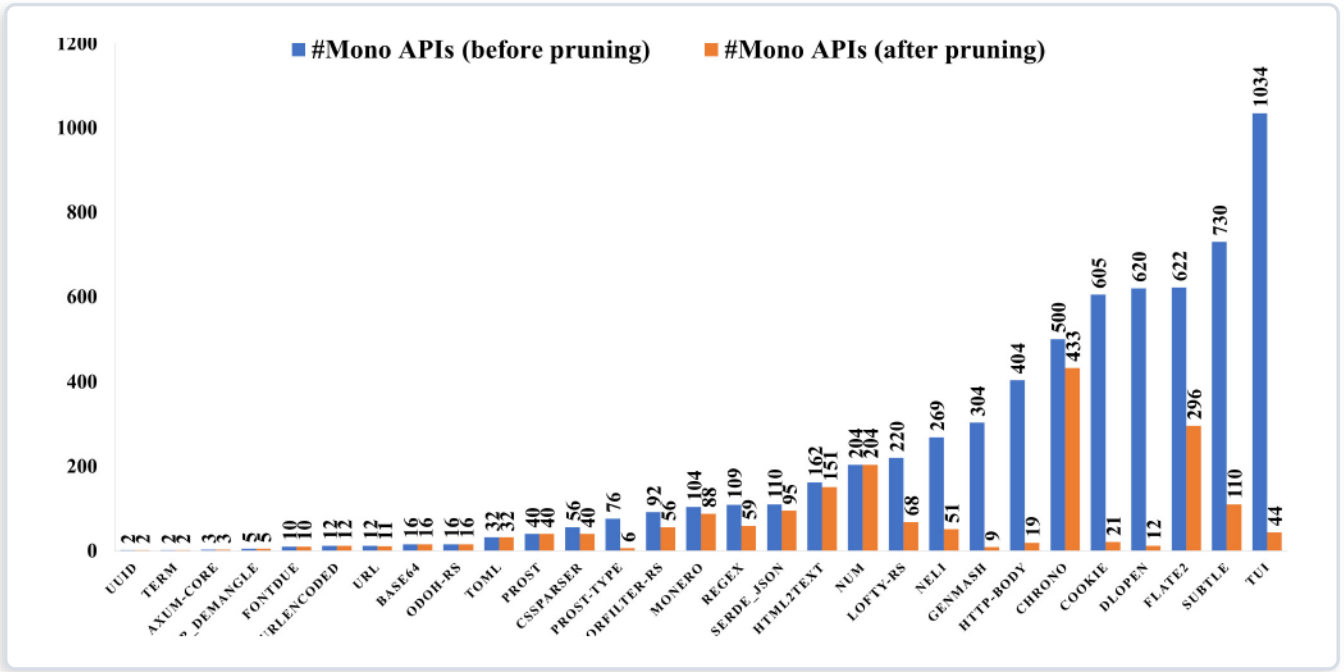
Library	Version	#APIs	#GAPIs	API Coverage	GAPI Coverage	#Mono APIs	#Mono APIs (After Pruning)	Fuzz Targets (Invalid)	Crashes	Bugs
serde_json	1.0.103	586	317	0.49	0.10	110	95	300 (33)	103	0
base64	0.21.4	40	22	0.60	0.64	16	16	17 (4)	0	0
uuid	1.4.1	133	9	0.94	0.22	2	2	7	1	0
form_urlencoded	1.2.0	16	9	0.81	0.67	12	12	10	4	1
flate2	1.0.28	274	228	0.54	0.50	622	296	147 (3)	66	0
http-body	0.4.5	156	143	0.17	0.09	404	19	6 (4)	0	0
subtle	2.5.0	107	19	0.96	0.79	730	110	300	17	0
prost-type	0.12.1	453	17	0.95	0.12	76	6	23	0	0
axum-core	0.3.4	95	73	0.01	0.01	3	3	3 (2)	0	0
monero	0.19.0	469	148	0.66	0.32	104	88	105	554	3
dlopen	0.1.8	55	48	0.29	0.25	620	12	2	0	0
lofty	0.17.1	1,186	144	0.73	0.15	220	68	27 (2)	30	2
html2text	0.6.0	145	78	0.72	0.67	162	151	182 (6)	8,515	5
neli	0.7.0	1,511	448	0.61	0.04	269	51	10	10	4
odoh-rs	1.0.2	39	15	0.74	0.80	16	16	21	6	2
xorfilter-rs	0.5.1	48	35	0.85	0.80	92	56	56	70	3
genmesh	0.6.0	31	30	0.13	0.13	304	9	6	0	0
term	0.7.0	42	19	0.26	0.11	2	2	4	0	0
chrono	0.4.31	639	202	0.88	0.73	500	433	300	347	0
cookie	0.18.0	116	34	0.78	0.44	605	21	22 (13)	0	0
cssparser	0.33.0	148	49	0.72	0.41	56	40	201 (2)	0	0
regex	1.8.3	207	16	0.93	0.69	109	59	57	258	2
tui	0.19.0	366	71	0.82	0.42	1,034	44	45 (22)	47	1
url	2.4.1	113	12	0.90	0.75	12	11	67 (2)	0	0
toml	0.8.2	239	161	0.38	0.14	32	32	300 (8)	22	0
prost	0.12.1	282	183	0.37	0.13	40	40	127	0	0
num	0.4.1	56	56	0.36	0.36	204	204	204	265	1
fontdue	0.8.0	54	12	0.39	0.17	10	10	12 (2)	0	0
cpp_demangle	0.4.3	104	11	0.17	0.27	5	5	5	0	0
Total	-	7,710	2,609	0.66	0.27	6,371	1,911	2,566 (103)	10,319	23

Library	UTF-8	Overflow	OOM	OOB	Other	Generic
form_urlencoded	0	0	1	0	0	1
monero	0	1	1	1	0	3
lofty	0	1	0	0	1	2
html2text	0	1	1	0	3	5
neli	0	3	1	0	0	0
odoh-rs	0	0	2	0	0	2
xorfilter-rs	0	1	0	1	0	2
regex	2	0	0	0	0	2
tui	0	1	0	0	0	0
num	0	1	0	0	0	1
Total	2	9	6	2	4	18

Generic is the number of bugs related to generic API for each library. OOB, Index out of bounds; OOM, out of memory.

23 real bugs across 10 libraries — **18 live in generic code.** Mostly arithmetic overflow, out-of-memory, and index-out-of-bounds.

RESULTS · RQ3 EFFICIENCY



RESULTS · RQ4 vs BASELINES

Library	#APIs	#GAPIs	RULF		RPG		RuMono	
			#Covered APIs	#Covered GAPIs	#Covered APIs	#Covered GAPIs	#Covered APIs (+RULF/+RPG)	#Covered GAPIs (+RULF/+RPG)
serde_json	586	317	237	-	26	2	289 (+52/+263)	31 (+31/+29)
base64	40	22	8	-	7	1	24 (+16/+17)	14 (+14/+13)
uuid	133	9	104	-	65	0	125 (+21/+60)	2 (+2/+2)
form_urlencoded	16	9	7	-	3	0	13 (+6/+10)	6 (+6/+6)
flate2	274	228	34	-	22	0	149 (+115/+127)	115 (+115/+**115**)
http-body	156	143	13	-	11	3	26 (+13/+15)	13 (+13/+10)
subtle	107	19	81	-	0	0	103 (+22/+103)	15 (+15/+15)
prost-types	453	17	158	-	38	0	432 (+274/+**394**)	2 (+2/+2)
axum-core	95	73	0	-	-	-	1 (+1/-)	1 (+1/-)
monero	469	148	143	-	131	0	311 (+168/+180)	48 (+48/+48)
dlopen	55	48	1	-	14	10	16 (+15/+2)	12 (+12/+2)
lofty	1,186	144	269	-	-	-	860 (+**591**/-)	22 (+22/-)
html2text	145	78	33	-	78	36	105 (+72/+27)	52 (+52/+16)
neli	1,511	448	416	-	-	-	917 (+501/-)	19 (+19/-)
odoh-rs	39	15	9	-	4	1	29 (+20/+25)	12 (+12/+11)
xorfilter-rs	48	35	13	-	0	0	41 (+28/+41)	28 (+28/+28)
genmesh	31	30	0	-	0	0	4 (+4/+4)	4 (+4/+4)
term	42	19	7	-	6	0	11 (+4/+5)	2 (+2/+2)
chrono	639	202	364	-	228	36	564 (+200/+336)	148 (+**148**/+112)
cookie	116	34	75	-	38	0	90 (+15/+52)	15 (+15/+15)
cssparser	148	49	82	-	-	-	107 (+25/-)	20 (+20/-)
regex	207	16	155	-	110	0	192 (+37/+82)	11 (+11/+11)
tui	366	71	184	-	80	0	299 (+115/+219)	30 (+30/+30)
url	113	12	82	-	72	8	102 (+20/+30)	9 (+9/+1)
toml	239	161	52	-	-	-	92 (+40/-)	23 (+23/-)
prost	282	183	78	-	43	0	105 (+27/+62)	24 (+24/+24)
num	56	56	0	-	-	-	20 (+20/-)	20 (+20/-)
fontdue	54	12	18	-	-	-	21 (+3/-)	2 (+2/-)
cpp_demangle	104	11	15	-	5	0	18 (+3/+13)	3 (+3/+3)
Total	7,710	2,609	2,638	0	943	97	5,066 (+2,428/+683)	703 (+703/+333)

Key takeaways

1. A generic function is secretly many — bugs hide in specific copies.
2. Separate what CAN fill a slot (graph) from what's ALLOWED (bounds).
3. Build copies from producible types — not just ones lying around.
4. Prune by behavior, not by type.

Same function, two types, two bugs — that's the whole paper.