ERASAN: Efficient Rust Address Sanitizer

Jiun Min*
Department of Computer Science
UNIST

Dongyeon Yu*
Department of Computer Science
UNIST

Seongyun Jeong
Department of Computer Science
UNIST

Dokyung Song
Department of Computer Science
Yonsei University

Yuseok Jeon†
Department of Computer Science
UNIST

2024 IEEE Symposium on Security and Privacy (SP)

Rust security feature

- Ownership & lifetime

```rust
fn ownership_and_borrowing() -> &u32 {
    // creates a `Vec`, a heap allocated buffer
    let vec = vec![1, 2, 3];

    // creates a reference to the first value with borrowing
    let first_val = &vec[0];

    // Ownership: `Vec` is automatically reclaimed
    // when its owner `vec` goes out of the scope.
    //
    // Borrowing: compile error; Rust prevents `first_val`
    // to outlive `vec` by tracking variable lifetimes.
    return first_val;
}
```

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

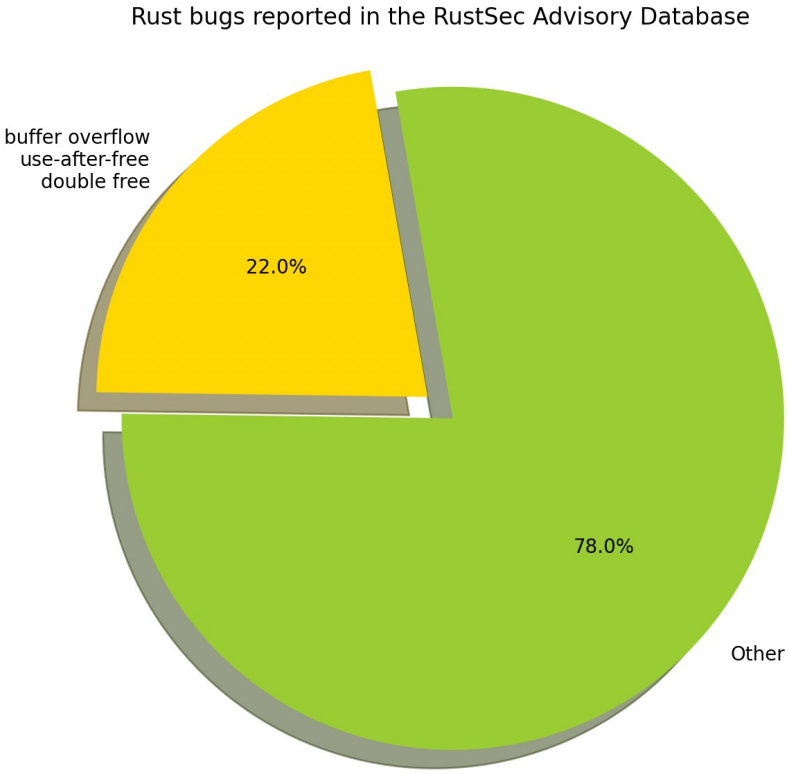
Rust security feature

- Borrowing && mutability

```rust
fn aliasing_xor_mutability() {
    let mut vec = vec![1, 2, 3];

    // exclusive mutable borrowing
    let mut_ref = &mut vec;

    // shared read-only borrowing
    let shared_ref1 = &vec;
    let shared_ref2 = &vec;
    println!("{}", shared_ref1[0]);
    println!("{}", shared_ref2[0]);

    // Aliasing xor Mutability: compile error;
    // Rust invalidates `mut_ref` when `shared_ref1` is
    // used since they cannot coexist at the same time.
    mut_ref.push(4);
}
```

https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html

Some statistic data



- Isolation
- Bug detection
- Dynamic analysis
- Static Analysis

Dynamic Analysis Based

[1] MIRI (MIR Interpreter). https://github.com/rust-lang/ miri.
[2] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In 2012 USENIX annual technical conference (USENIX ATC 12), pages 309–318, 2012.

```
1  fn main(){
2   /* Allocate the value */
3   let v1 = 23;
4
5   /* Safe Reference */
6   let s1 = &v1;
7
8   /* Access Safe Reference */
9   println!("{}", s1);
10
11  /* Raw Pointer */
12  let u1  = &v1 as *const i32;
13
14  /* Access Raw Pointer */
15  unsafe {println!("{}", *u1);}
16 }
```

```
17 define internal void @main() {
18 bb5:
19  ; Memory Access : s1
20  %2 = load i32, i32* %s1
21  ; ASan Instrumentation
22  call void @asan_load64(i32 %r)
23  br label %bb6
24 bb9:
25  ; Memory Access : *u1
26  %3 = load i32*, i32** %u1
27  ; ASan Instrumentation
28  call void @asan_load32(i32 %r)
29  br label %bb6
30  ...
31 }
```

Rust ASAN

```
1  fn main(){
2   /* Allocate the value */
3   let v1 = 23;
4
5   /* Safe Reference */
6   let s1 = &v1;
7
8   /* Access Safe Reference */
9   println!("{}", s1);
10
11  /* Raw Pointer */
12  let u1  = &v1 as *const i32;
13
14  /* Access Raw Pointer */
15  unsafe {println!("{}", *u1);}
16 }
```

```
17 define internal void @main() {
18 bb5:
19   ; Memory Access : s1
20   %2 = load i32, i32* %s1
     ; ASan Instrumentation
22   call void @asan_load64(i32 %r)
23   br label %bb6
24 bb9:
25   ; Memory Access : *u1
26   %3 = load i32*, i32** %u1
     ; ASan Instrumentation
28   call void @asan_load32(i32 %r)
29   br label %bb6
30   ...
31 }
```

Unsafe Rust!

(i) Dereferencing a raw pointer

(ii) Calling an unsafe function/method
(iii) Implementing an unsafe trait
(iv) Accessing fields of unions
(v) Accessing/modifying a mutable static variable

Root Cause

Not directly lead to memory bugs

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Unsafe Rust!

TABLE 1: Applicability of Rust memory bug related memory safety policies (R1:Ownership, R2:Borrow Check, R3:Lifetime Inference, R4: Bound Check in static-time, R5: Bound Check in run-time) to each capability.

Capabilities	Compile Time				Run time
	R1	R2	R3	R4	R5
Dereferencing a Raw Pointer	✗	✗	✗	✗	✗
Calling an Unsafe Function/Method	✓	✓	✓	✓	✓
Implementing an Unsafe Trait	✓	✓	✓	✓	✓
Other Operations	✓	✓	✓	✓	✓

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Unsafe Rust!

```
1  fn main(){
2   /* Allocate the value */
3   let v1 = 23;
4
5   /* Safe Reference */
6   let s1 = &v1;
7
8   /* Access Safe Reference */
9   println!("{}", s1);
10
11  /* Raw Pointer */
12  let u1  = &v1 as *const i32;
13
14  /* Access Raw Pointer */
15  unsafe {println!("{}", *u1);}
16 }
```

Create is safe &
Dereference is unsafe

https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/raw-pointers.html

Unsafe Rust!

```rust
/* RUSTSEC-2020-0097: UAF due to a raw pointer without unsafe
↪  block */
#![forbid(unsafe_code)]

use xcb::base::Error;

fn main() {
    let mut v1: Vec<i8> = vec![1, 2, 3, 0];
    let _ = Error {
        ptr: v1.as_mut_ptr(); // a raw pointer
    };

    // use-after-free in v1
    v1[0] = 123;
}
```

Figure 2: An example of UAF (RUSTSEC-2020-0097) in Rust *xcb* crate

Rust Memory Bug

TABLE 2: Analysis of all 131 memory bugs in RustSec from 2016 to 2023

Memory Bug Type	Detected by Rust		Total
	Yes	No	
Buffer-Overflow (BOF)	12	49	61
Use-After-Free (UAF)	0	44	44
Double-Free (DF)	0	26	26
Total	12	119	131

Rust Memory Bug Pattern

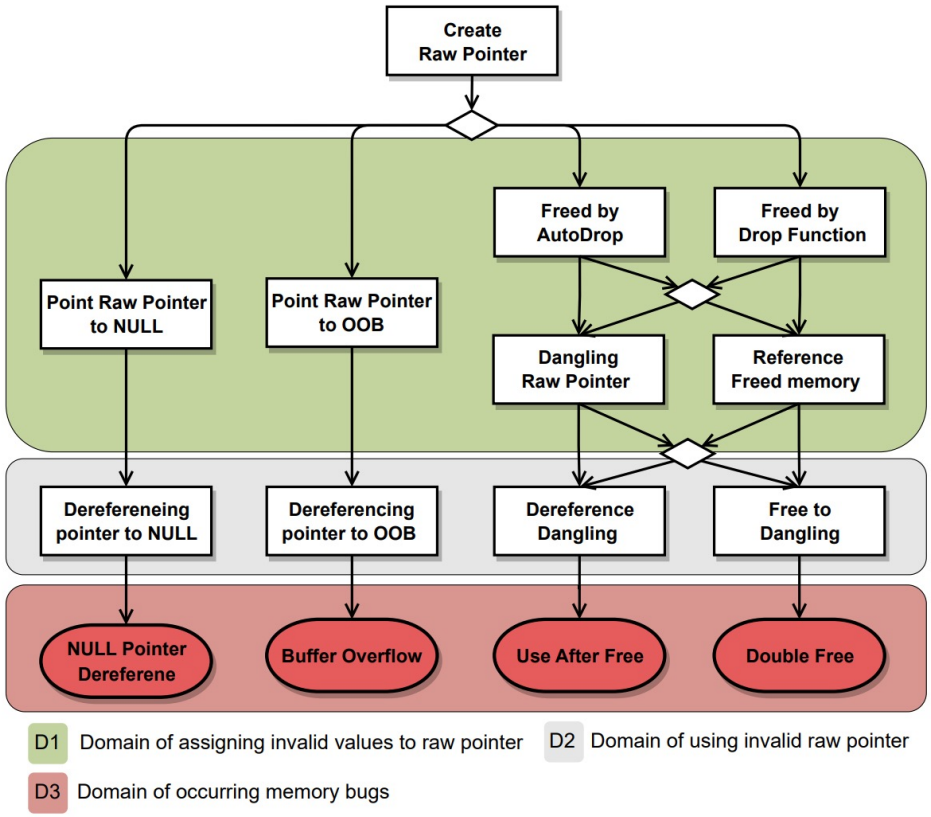


Figure 3: Rust memory bug patterns. Domain means the behaviors related to each domain description at the bottom of the figure.

ERASAN

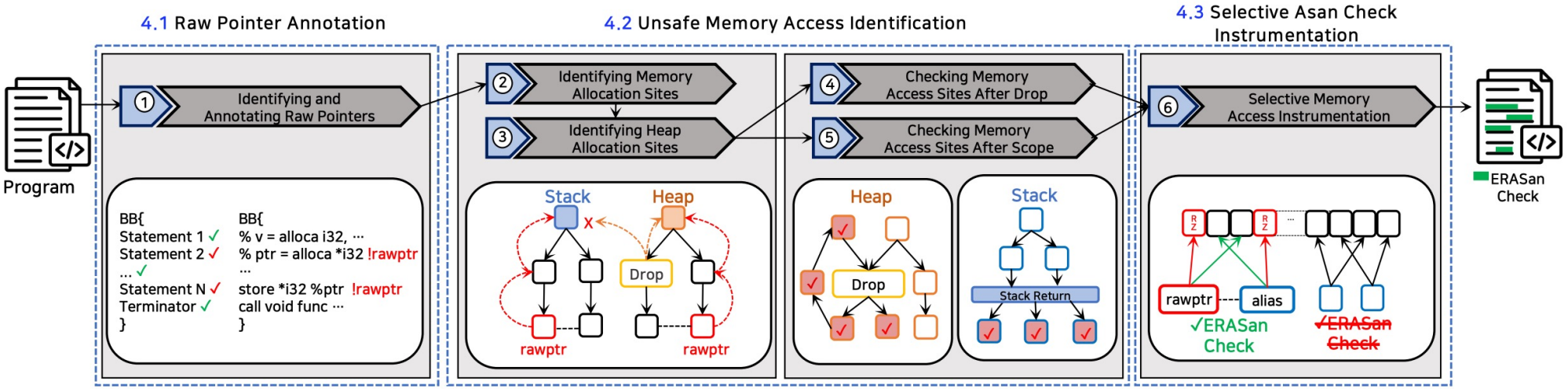
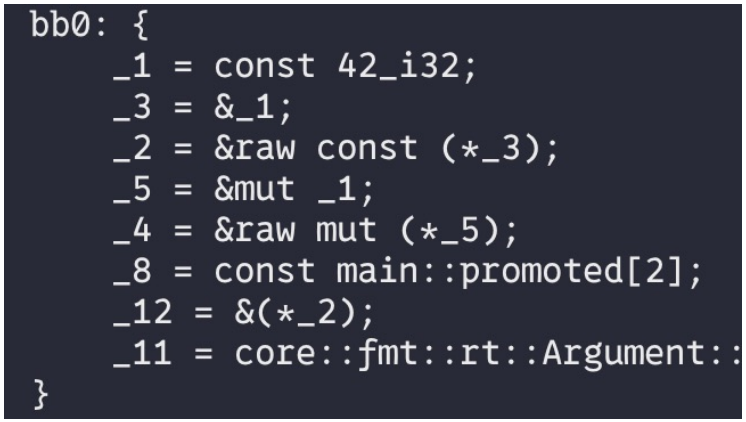


Figure 5: Overview of ERASAN. The small black boxes indicate the memory accesses and the small red boxes present the memory accesses by the raw pointer. The small boxes filled with red represent the memory accesses checked by ERASAN. The arrows indicate the direction of static analysis.

Raw pointer annotation

```rust
fn main() {
    let mut num = 42;

    // 创建不可变和可变原生指针
    let r1 = &num as *const i32;  // 不可变原生指针
    let r2 = &mut num as *mut i32;  // 可变原生指针

    // 安全块外不能解引用原生指针
    unsafe {
        // 解引用不可变原生指针
        println!("r1 points to: {}", *r1);

        // 修改通过可变原生指针指向的值
        *r2 = 58;

        // 再次解引用指针，查看修改后的值
        println!("r2 now points to: {}", *r2);
    }

    // 正常的引用机制仍然可以使用
    println!("num is now: {}", num);
}
```



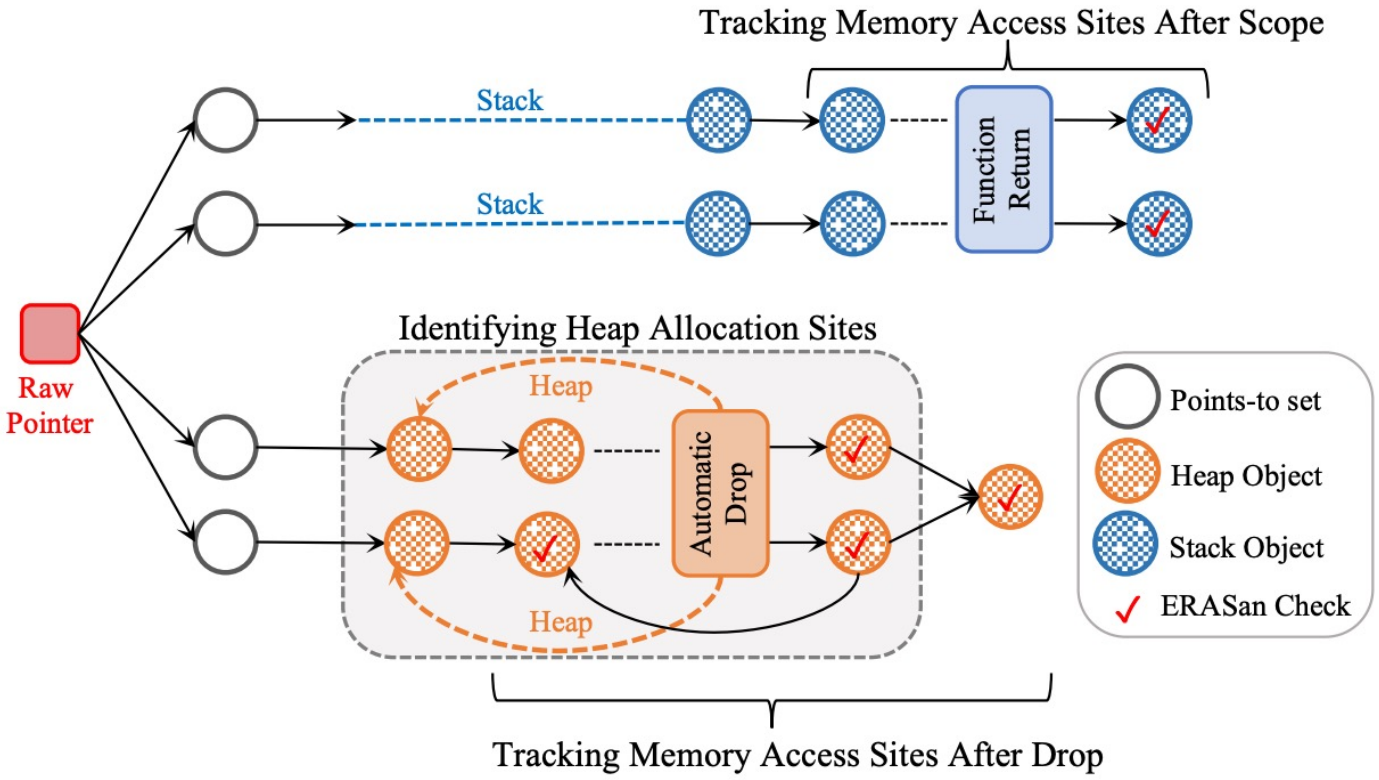
Raw pointer alloc set

Algorithm 1: The core algorithm of ERASAN

Input : I, S_g
I: a given LLVM-IR instruction set;
S_g: a sparse value flow graph, SVFG;
$A_h \Leftarrow$ a set of heap alloc sites accessible by rawptr;
$A_s \Leftarrow$ a set of stack alloc sites accessible by rawptr;
$A_r \Leftarrow$ a set of rawptr memory alloc sites;

1 1. Collects all instructions annotated with rawptr metadata
2 **forall** $inst \in I$ **do**
3 **if** *inst.hasMetaData("!rawptr")* **then**
4 $A_r \Leftarrow$ PointsToAnalysis($inst, S_g$);
5 2. Perform Raw Pointer Allocation Sites Analysis
6 $S_s \Leftarrow \emptyset$; `// a set of stack SVFGNode`
7 $S_h \Leftarrow \emptyset$; `// a set of heap SVFGNode`
8 **forall** $a \in A_r$ **do**
9 **if** $S_s.contains(a)$ **then**
10 A_s.insert(a);
11 return;
12 **else if** $S_h.contains(a)$ **then**
13 A_h.insert(a);
14 return;
15 **else**
16 **if** $is\_drop\_traits(a, S_g)$ **then**
17 A_s.insert(a);
18 S_h.update(a.getDFSPath);
19 **else**
20 A_h.insert(a);
21 S_s.update(a.getAllVisitedPaths);

Identifying Unsafe Memory Access Sites



ERASAN Instrumentation

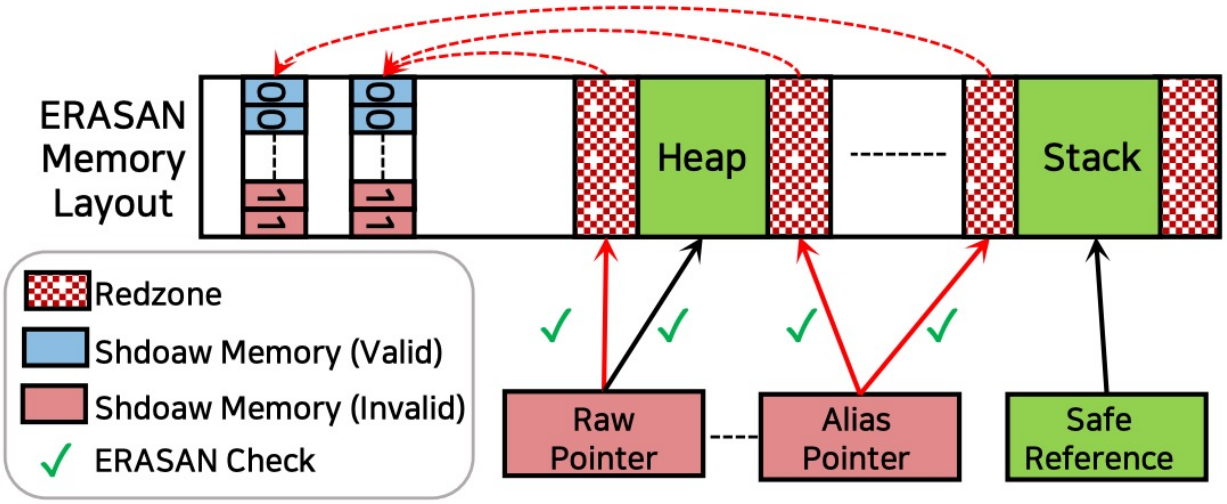


Figure 7: The memory layout of ERASAN with redzone and shadow memory

Check reduction

Benchmark	ASan	ERASAN-unsafe		ERASAN-rawptr		ERASAN-thread		ERASAN	
	Check. (#)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)	Check. (#)	Redu. (%)
syn	260,805	214,566	17.71	155,985	40.17	90,648	65.24	85,235	67.31
rand (generator)	3,729	1,623	56.47	656	82.41	6	99.83	6	99.83
rand (misc)	2,343	1,482	37.74	280	88.05	6	99.83	6	99.74
crossbeam	2,086	766	63.28	496	76.22	179	91.42	179	91.42
itoa	1,167	659	43.53	210	82.01	0	100.0	0	100.0
base64	55,049	46,466	15.59	25,385	53.88	6,878	87.51	5,918	89.25
regex	9,442	7,738	18.04	5,292	43.95	894	90.53	894	90.53
memchr	203,592	190,066	6.644	125,216	38.49	59,459	73.09	26,434	88.03
hashbrown	14,518	11,923	17,87	5,381	62.94	1,396	90.38	1,188	91.82
smallvec	2,244	1,505	32.93	499	77.76	36	98.39	36	98.39
ryu	2,676	2,228	16.74	422	84.23	238	91.11	46	98.28
semver	1,056	759	28.13	250	76.32	30	97.16	30	97.16
strsim	974	897	7.906	452	57.19	44	95.48	34	96.78
bytes (bytes)	4,456	3,511	21.21	419	90.59	133	97.02	133	97.02
bytes (buf)	7,478	2,947	60.59	235	96.86	47	99.37	47	99.37
bytes-mut	5,095	4,138	18.78	722	85.83	85	98.33	85	98.33
indexmap	76,041	66,636	12.37	49,616	34.75	25,168	66.90	25,168	66.90
byteorder	5,242	2,126	59.44	906	82.72	49	99.07	49	99.07
num-integer	22,622	18,866	16.61	13,068	42.23	685	96.97	583	97.42
url	69,188	55,328	20.03	43,034	37.80	26,715	61.39	24,611	64.43
uuid (parse)	109,942	97,123	11.66	67,160	38.91	35,643	67.59	31,898	70.99
uuid (format)	110,558	97,637	11.68	67,141	39.27	35,657	67.75	31,926	71.12
unicode	1,762	1,700	3.518	1,162	34.05	41	97.67	41	97.67
Average	43,022	36,116	26.30% ↓	24,521	62.95% ↓	12,349	88.34% ↓	10,197	90.03% ↓

- ERASAN-unsafe: only check access reachable from unsafe blocks
- ERASAN-rawptr: do not apply the stack and heap optimization
- ERASAN-thread: do not apply the stack and heap opt for threads func (by closure symbol)

Time overhead

Benchmark	ASan	ERASAN-unsafe		ERASAN-rawptr		ERASAN-thread		ERASAN	
	Over. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)	Over. (%)	Redu. (%)
syn	471.14	407.02	64.12	328.40	142.74	302.58	168.56	200.26	270.88
rand (generator)	214.96	78.50	136.46	67.34	147.63	7.65	207.32	7.65	207.31
rand (misc)	58.93	37.22	21.71	10.96	47.97	0.21	58.73	0.21	58.73
crossbeam	45.71	12.35	33.36	2.49	43.22	0.39	45.31	0.39	45.31
itoa	660.27	94.69	565.57	31.76	628.51	21.04	639.22	21.04	639.23
base64	396.47	301.31	95.16	275.45	121.02	230.13	166.33	196.28	200.18
regex	530.32	476.63	53.69	449.37	80.96	306.19	224.13	306.19	224.13
memchr	458.05	225.06	232.99	51.73	406.32	44.42	413.63	31.47	426.58
hashbrown	360.73	332.77	27.96	280.97	79.76	73.01	287.73	65.85	294.87
smallvec	490.26	434.75	55.51	401.39	88.86	354.54	135.72	354.54	135.72
ryu	252.81	197.87	54.94	124.42	128.39	50.15	202.66	46.16	206.65
semver	583.76	555.95	27.81	397.19	186.56	252.88	330.86	252.88	330.87
strsim	275.73	262.22	13.51	97.42	178.32	38.87	330.88	34.39	241.33
bytes (bytes)	150.87	133.02	17.88	73.85	77.05	49.82	101.08	49.82	101.08
bytes (buf)	327.41	63.11	264.28	61.09	266.31	59.39	268.01	59.39	268.01
bytes-mut	134.96	106.59	28.37	98.54	36.42	55.95	79.02	59.95	79.02
indexmap	378.12	331.07	47.04	320.51	57.61	143.77	234.34	143.77	234.34
byteorder	330.15	72.80	257.34	54.59	275.56	15.22	314.93	15.22	314.93
num-integer	173.48	147.36	26.12	84.34	89.13	29.97	143.51	1.05	172.43
url	348.21	337.69	10.52	332.44	15.77	264.74	83.48	221.91	126.29
uuid (parse)	531.49	499.86	31.63	53.33	478.16	46.16	485.33	40.42	491.08
uuid (format)	242.72	170.05	72.69	105.47	137.25	67.31	175.40	55.47	187.25
unicode	288.13	287.89	0.25	253.96	34.18	46.32	241.82	46.32	241.82
Average	334.98%	241.99%	92.99% ↓	172.04%	162.94%↓	106.86%	228.12%↓	95.94%	239.05%↓

Other overhead from ASAN:

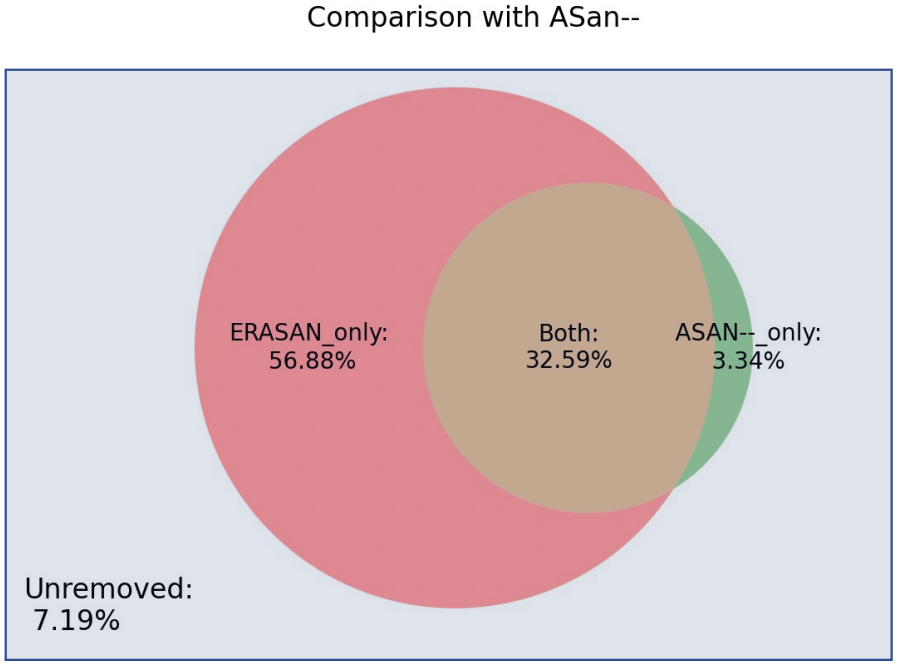
- Poisoning objects
- Shadow memory initialization
- Teardown

Compile overhead

Benchmark	Program	Native	ASan		ERASAN	
	Line (#)	Time (µs)	Time (µs)	Incre. (X)	Time (µs)	Incre. (X)
syn	234,795	86,141	112,300	×1.30	641,181	×7.44
rand (generator)	20,859	1,393	2,058	×1.47	2,509	×1.80
rand (misc)	20,972	1,294	1,791	×1.38	4,041	×3.12
crossbeam	20,980	1,475	1,541	×1.04	4,775	×3.23
itoa	479	577	610	×1.05	976	×2.03
base64	7,120	9,716	27,305	×2.81	135,143	×13.81
regex	75,676	3,915	6,848	×1.74	25,769	×6.58
memchr	63,631	68,265	130,127	×1.91	302,887	×4.44
hashbrown	15,790	2,609	4,831	×1.85	9,203	×3.53
smallvec	3,688	852	1,361	×1.59	1,629	×1.91
ryu	3,930	917	1,478	×1.61	1,859	×2.02
semver	3,188	475	546	×1.14	861	×1.81
strsim	1,141	521	707	×1.36	1,004	×1.93
bytes (bytes)	9,936	1,432	2,180	×1.52	5,170	×3.61
bytes (buf)	10,082	1,173	2,123	×1.81	5,511	×4.69
bytes-mut	10,002	1,394	2,199	×1.58	5,239	×3.75
indexmap	11,166	19,441	40,387	×2.08	94,720	×4.87
byteorder	5,845	1,523	2,468	×1.62	3,309	×2.17
num-integer	2,939	5,130	8,321	×1.62	15,280	×2.98
url	26,864	11,917	26,016	×2.18	63.487	×5.32
uuid (parse)	7,745	18,634	39,103	×2.09	102,997	×5.53
uuid (format)	7,726	20,583	55,602	×2.70	114,980	×5.58
unicode	158,888	945	1,205	×1.28	1,430	×1.51
Average	31,454	9,579	10,527	×1.72 ↑	53,801	×4.07 ↑

RUSTSEC/CVE ID	Crate	Bug Type	ASan	ERASAN
RUSTSEC-2023-0005	tokio	UAF	✔	✔
RUSTSEC-2022-0070	secp	UAF	✔	✔
RUSTSEC-2022-0078	bumpalo	UAF	✔	✔
RUSTSEC-2021-0018	qwutils	DF	✔	✔
RUSTSEC-2021-0028	toodee	DF	✔	✔
RUSTSEC-2021-0031	nano_arena	UAF	✔	✔
RUSTSEC-2021-0033	stack-dst	DF	✔	✔
RUSTSEC-2021-0047	slice-deque	DF	✔	✔
RUSTSEC-2021-0130	lru	UAF	✔	✔
RUSTSEC-2021-0128	rusqlite	UAF	✔	✔
RUSTSEC-2021-0094	rdiff	Heap Ovf.	✔	✔
RUSTSEC-2021-0053	algorithmica	DF	✔	✔
RUSTSEC-2021-0049	through	DF	✔	✔
RUSTSEC-2021-0048	stackvector	Stack Ovf.	✔	✔
RUSTSEC-2021-0042	insert_many	DF	✔	✔
RUSTSEC-2021-0039	endian_trait	DF	✔	✔
RUSTSEC-2021-0003	smallvec	Heap Ovf.	✔	✔
RUSTSEC-2020-0167	pnet_packet	Heap Ovf.	✔	✔
RUSTSEC-2020-0097	xcb	UAF	✔	✔
RUSTSEC-2020-0091	arc-swap	UAF	✔	✔
RUSTSEC-2020-0061	futures	NPD	✔	✔
RUSTSEC-2020-0060	futures	UAF	✔	✔
RUSTSEC-2020-0039	simple-slab	Heap Ovf.	✔	✔
RUSTSEC-2020-0038	ordnung	DF	✔	✔
RUSTSEC-2020-0005	cbox	UAF	✔	✔
RUSTSEC-2019-0023	string-interner	UAF	✔	✔
RUSTSEC-2019-0009	smallvec	DF	✔	✔
RUSTSEC-2019-0034	http	DF	✔	✔

Benchmark	Only ASan--		Only ERASAN		Both		Unremoved	
	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)	Remv. (#)	Redu. (%)
syn	27,038	10.37	118,387	45.39	57,183	21.93	58,197	22.31
rand (generator)	0	0	2,320	62.22	1,403	37.62	6	0.16
rand (misc)	0	0	1,247	53.22	1,090	46.52	6	0.25
crossbeam	40	1.92	1,334	63.95	573	27.47	139	6.66
itoa	0	0	839	71.89	328	28.11	0	0
base64	1,469	2.67	34,855	63.32	14,276	25.93	4,449	8.08
regex	252	2.67	6,663	70.57	1,885	19.96	642	6.79
memchr	18,785	8.49	108,703	49.18	57,598	26.06	35,932	16.26
hashbrown	305	2.10	7,862	54.14	5,471	37.68	883	6.08
smallvec	0	0	1,490	66.39	718	31.99	36	1.61
ryu	9	0.33	1,755	65.58	875	32.69	37	1.38
semver	17	1.61	649	61.46	377	35.70	13	1.23
strsim	4	0.41	611	62.73	329	33.78	30	3.08
bytes (bytes)	45	1.01	2,256	50.63	2,067	46.39	88	1.97
bytes (buf)	15	0.20	3,441	46.01	3,990	53.36	32	0.43
bytes-mut	27	0.52	2,680	52.60	2,330	45.73	58	1.14
indexmap	10,164	13.37	29,426	38.69	21,447	28.20	15,004	19.73
byteorder	11	0.21	3,482	66.43	1,711	32.64	38	0.72
num-integer	197	0.87	14,875	65.75	7,164	31.67	386	1.71
url	8,360	12.08	29,028	41.96	15,550	22.47	16,250	23.49
uuid (parse)	9,711	8.83	50,890	46.28	27,167	24.71	22,287	20.17
uuid (format)	9,712	8.78	51,161	46.27	27,471	24.85	22,214	20.09
unicode	6	0.34	1,120	63.56	601	34.11	35	1.98
Average	3,746	3.34%	20,655	56.88%	10,939	32.59%	7,680	7.19%



Static Analysis Based

[1] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 84–99, 2021.
[2] Cui M, Chen C, Xu H, et al. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(4): 1-21.
[3] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. Rupair: towards automatic buffer overflow detection and rectification for rust. In Annual

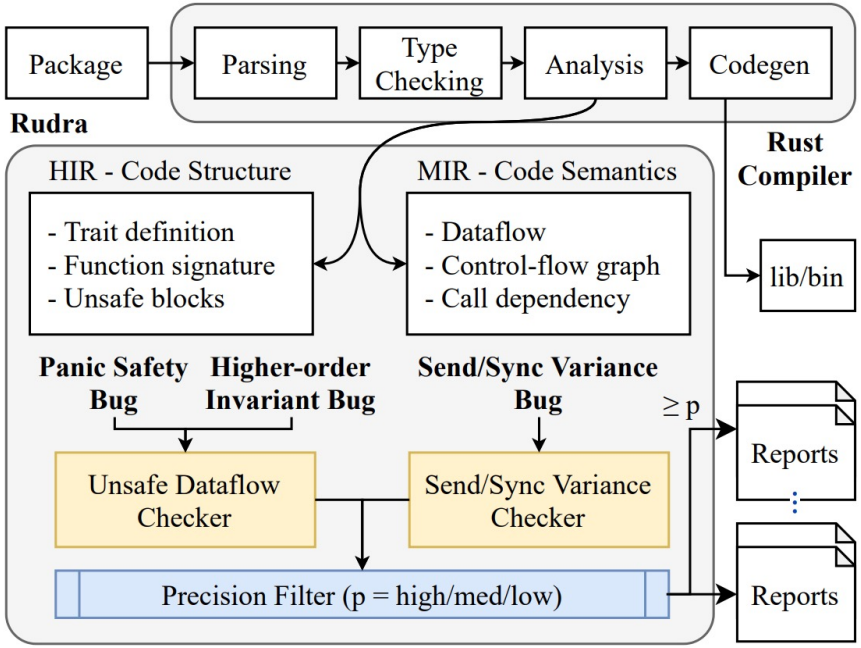


Figure 9. Overview of RUDRA's design.

ecting bugs in rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC
1.
s-language memory management issues in rust. In European Symposium on Research in Computer

Static Analysis Based

[1] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 84–99, 2021.

[2] Cui M, Chen C, Xu H, et al. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(4): 1-21. （FDU）

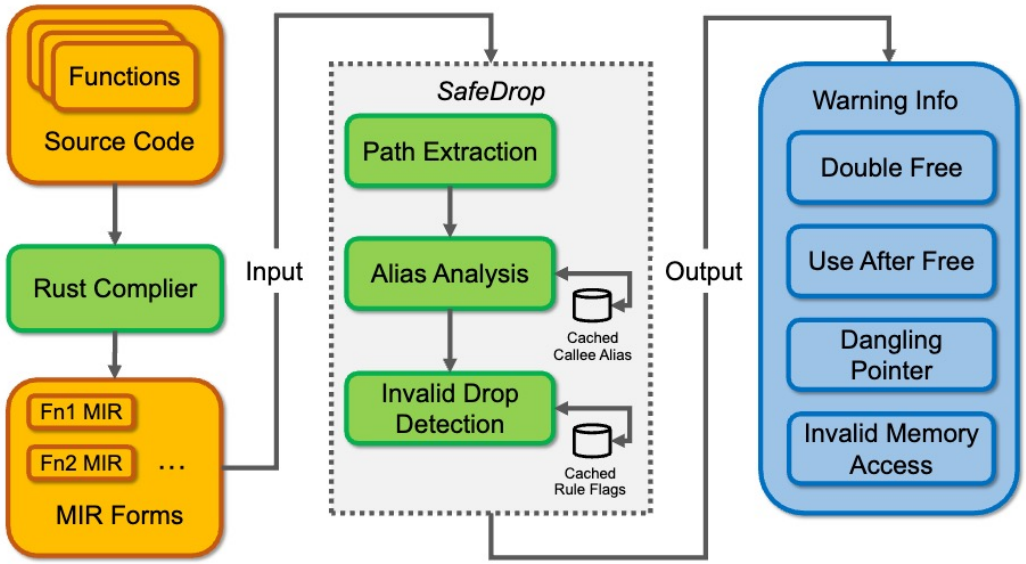


Figure 3: Overall framework of SafeDrop.

wards automatic buffer overflow detection and rectification for rust. In Annual

rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC

nory management issues in rust. In European Symposium on Research in Computer

Static Analysis Based

[1] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: finding memory safety bugs in rust at the ecosystem scale. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pages 84–99, 2021.

[2] Cui M, Chen C, Xu H, et al. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(4): 1-21. （FDU）

[3] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. Rupair: towards automatic buffer overflow detection and rectification for rust. In Annual Computer Security Applications Conference, pages 812–823, 2021. （USTC, ACSAC 2021）

[4] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC Confe

[5] Zh search in Computer Securi

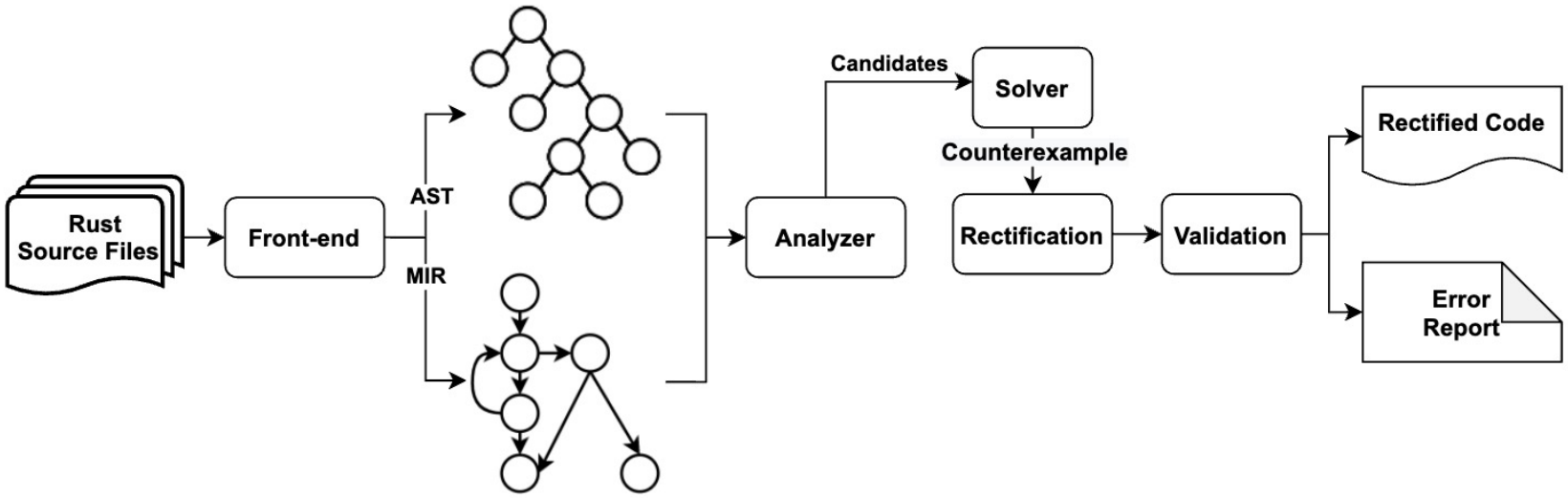


Figure 4: RUPAIR Architecture

Static Analysis Based

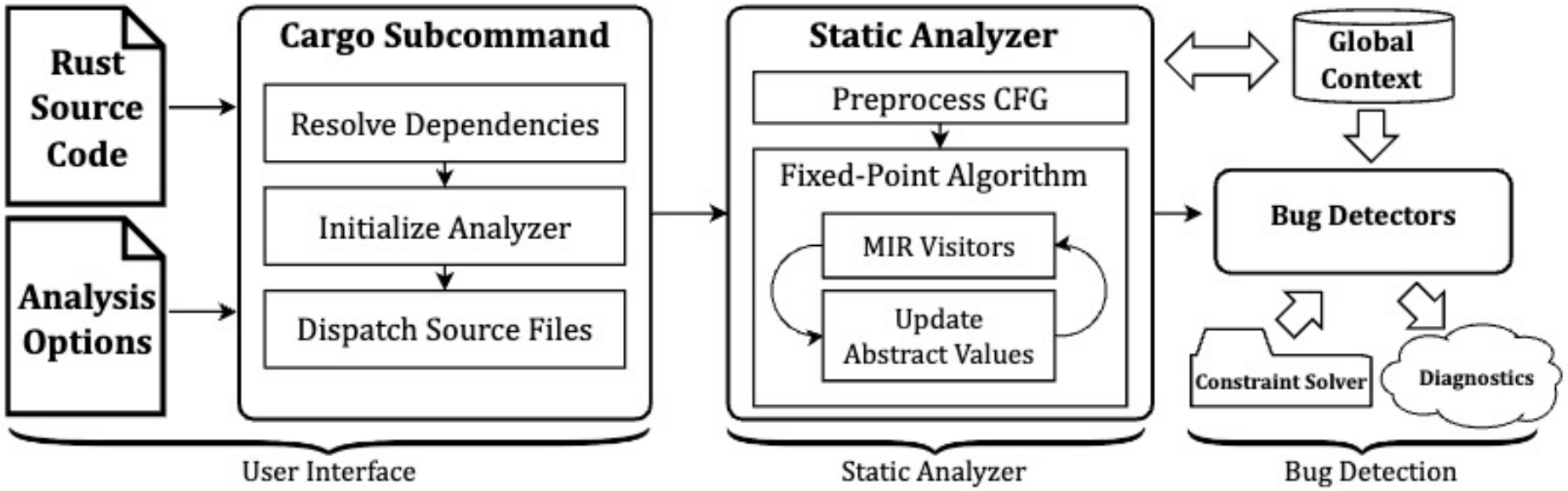


Figure 1: The architecture of MirChecker.

M SIGOPS

and

[3] Baojian Hua, Wanrong Ouyang, Chengman Jiang, Qiliang Fan, and Zhizhong Pan. Rupair: towards automatic buffer overflow detection and rectification for rust. In Annual Computer Security Applications Conference, pages 812–823, 2021. (USTC, ACSAC 2021)

[4] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Mirchecker: detecting bugs in rust programs via static analysis. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2183–2196, 2021. (CUHK)

[5] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in rust. In European Symposium on Research in Computer Security, pages 680–700. Springer, 2022.

Static Analysis Based

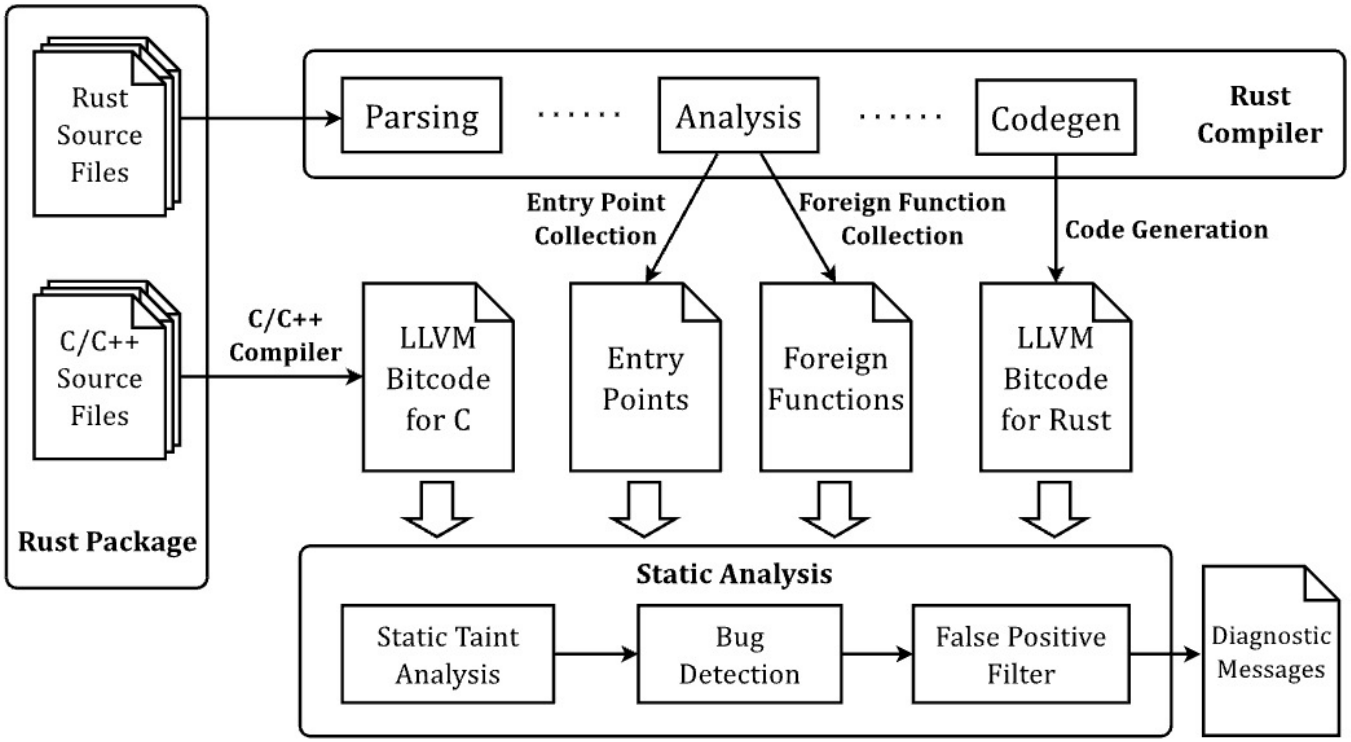


Fig. 2. The architecture of FFICHECKER.

e ecosystem scale. In Proceedings of the ACM SIGOPS

. ACM Transactions on Software Engineering and

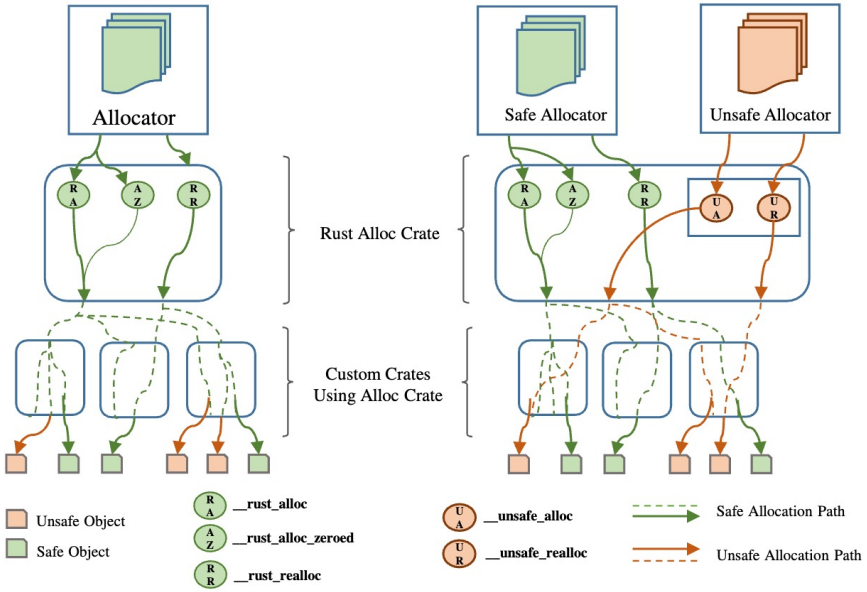
ow detection and rectification for rust. In Annual

is. In Proceedings of the 2021 ACM SIGSAC

[5] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. Detecting cross-language memory management issues in rust. In European Symposium on Research in Computer Security, pages 680–700. Springer, 2022.

Isolation && Bug detection

[1] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. Trust: A compilation framework for inprocess isolation to protect safe rust against untrusted code. In 32nd USENIX Security Symposium (USENIX Security 23). Baltimore, MD: USENIX Association, 2023.
[2] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In Proceedings of the Seventeenth European Conference on Computer Systems, pages 132– 148, 2022.
[3] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in rust. In Proceedings of the 9th Workshop on Programming Languages and Operating Systems, pages 51–57, 2017.



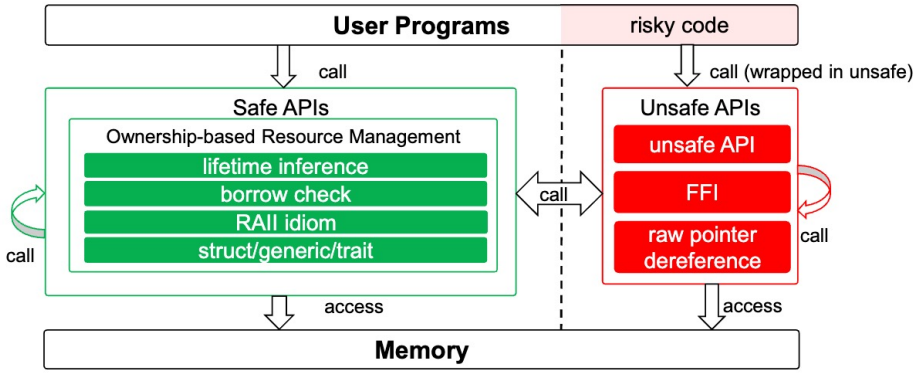


Fig. 1. Idea of Rust for preventing memory-safety bugs.

[1] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. Memory-safety challenge considered solved? an in-depth study with all rust cves. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(1):1–25, 2021.