RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust

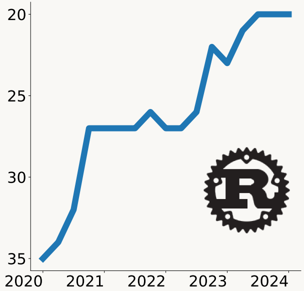
Kyuwon Cho 🎙, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee

Systems Security Lab
Dept. of Computer Science and Engineering
Sungkyunkwan University

August 15, 2024

Rust: The Safe Programming Language

- Rust is safer alternative to C/C++ in system programming with its language-level safety guarantees
- Rust is seeing widespread adoption



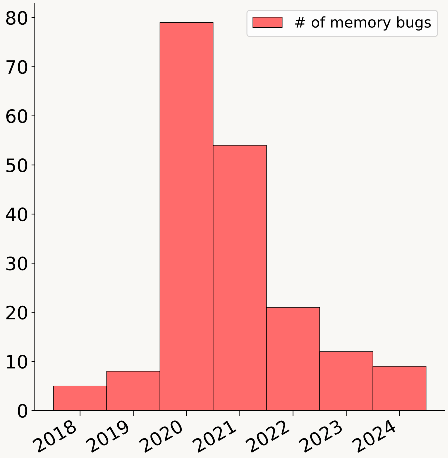
Github, Top 50 Programming Languages Globally

Rust: The (*Mostly*) Safe Programming Language

- Rust's safety guarantees are not free
  - Programmers are forced to participate in Rust concepts and semantics such as ownership.
- `unsafe` Rust allows programmers to temporarily trade safety for flexibility
  - Raw pointer access
  - Override ownership rules
  - Invoke unsafe foreign functions (e.g., C/C++)
  - Etc..

Rust: The (*Mostly*) Safe Programming Language

- Rust is certainly not infallible to memory bugs
- Study shows 99%(184/185) all reported memory bugs stem from `unsafe` use[1]



[1] Hui Xu et al. *"Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs."*, ACM Trans. SW. Eng. Methodol. (Sept. 2021).

Rust: The (*Mostly*) Safe Programming Language

- Rust has built-in option for compiling with **ASan** since 2017 [1]
  - E.g., `rustc -Zsanitizer=address ..`
- Many Rust developers have been using ASan for testing to have found numerous memory errors across many crates

[1] https://github.com/rust-lang/rust/pull/38699

Our Observation

Sanitizers for unsafe languages assume memory error anywhere in program, while most of Rust program code retains safety even with `unsafe`.

- But.. bugs reside in `unsafe` in fact can have cascading effect on safe code.
- Accurately identifying *true-safe* code and *false-safe* code is not trivial

Safety of Rust

```
fn unsafe_func<T>(...) -> &'static mut T {
...
    let refer: &'static mut T = unsafe { ptr + 0xdeadbeef as & _ };
    return refer;
}
...
    let from_unsafe = unsafe_func(..);
    let refer:&'a mut T = *from_unsafe ;
...
    refer.push(other_val);
```

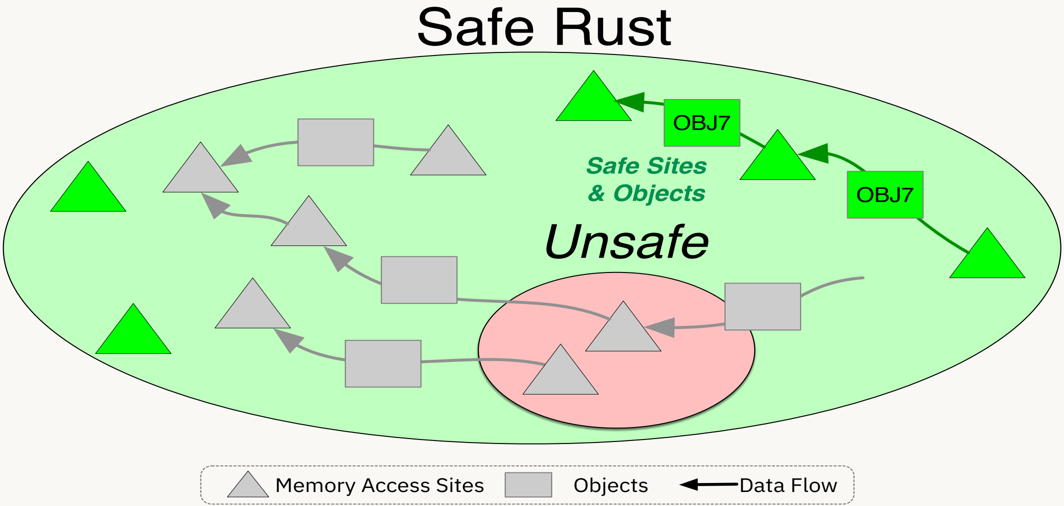
: *False-Safe* : Unsafe

Safety of Rust

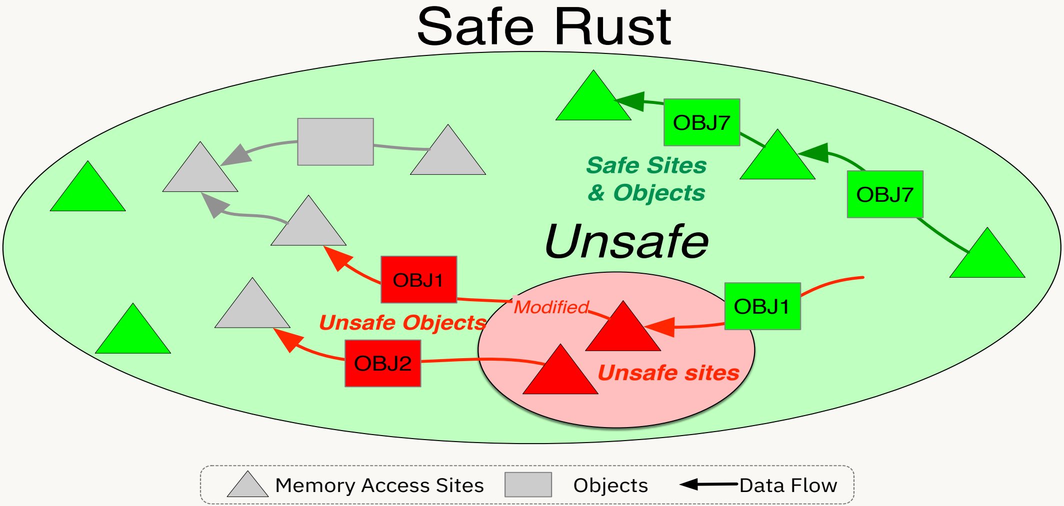
```
fn unsafe_func<T>(...) -> &'static mut T {
...
    let refer: &'static mut T = unsafe { ptr + 0xdeadbeef as & _ };
    return refer;
}
...
    let from_unsafe = unsafe_func(..);
    let refer :&'a mut T = *from_unsafe ;
...
    refer.push(other_val)
```

: *False-Safe*   : Unsafe

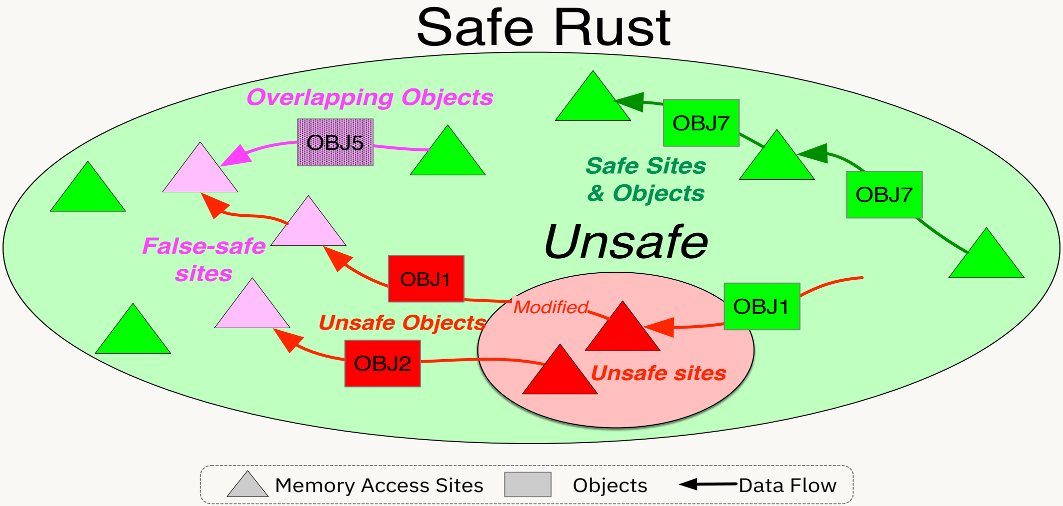
RUSTSAN Terminology



RustSan Terminology



RustSan Terminology

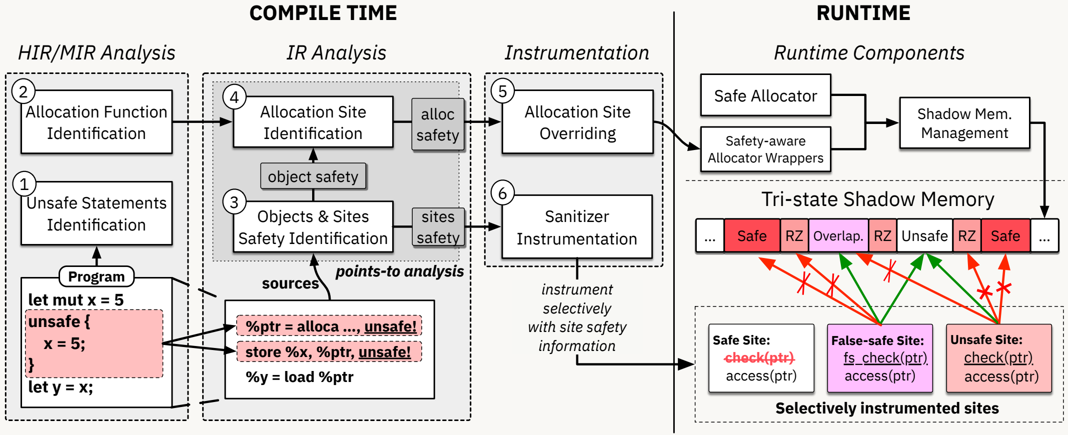


RustSan Research Statement

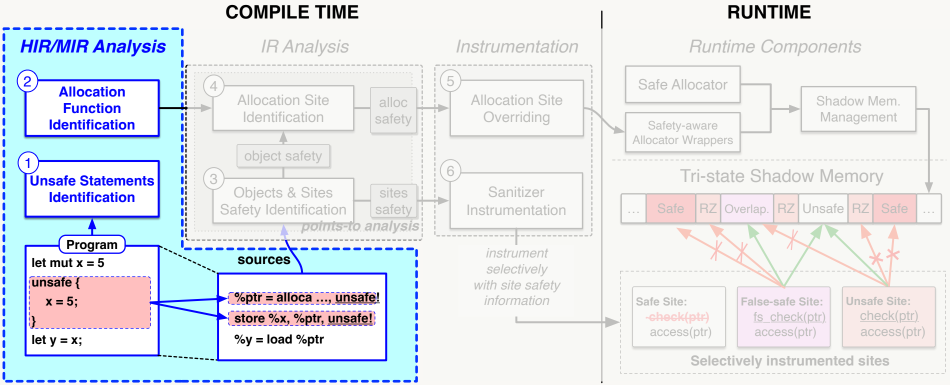
RustSan retrofits ASan to pinpoint unsafe and potentially unsafe memory access sites while lifting costly shadow memory checks on safe sites.

- Bridging Rust semantics and LLVM-based sanitizer with **Cross-IR analysis**
- Non-binary memory access validation model with **Tri-state shadow memory**

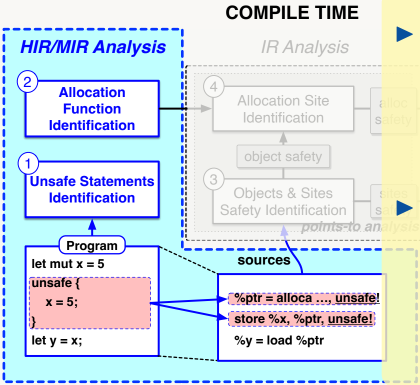
RustSan: Overall Design



RustSan: Rust HIR/MIR-level

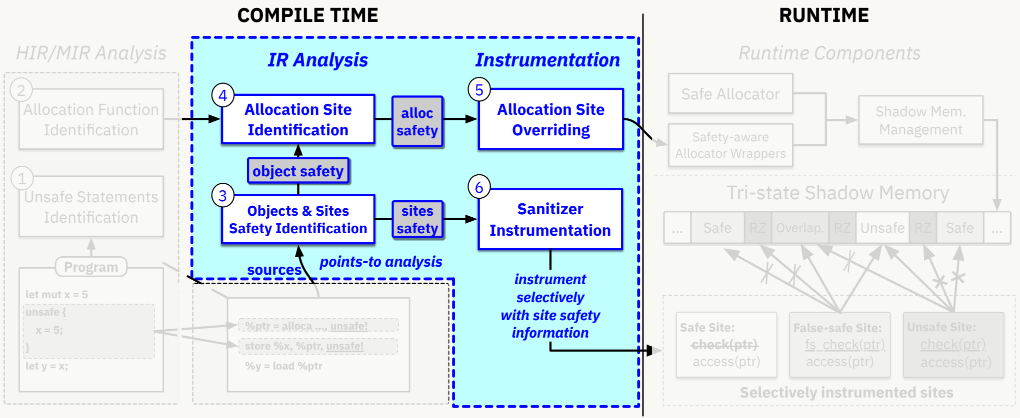


RustSan: Rust HIR/MIR-level

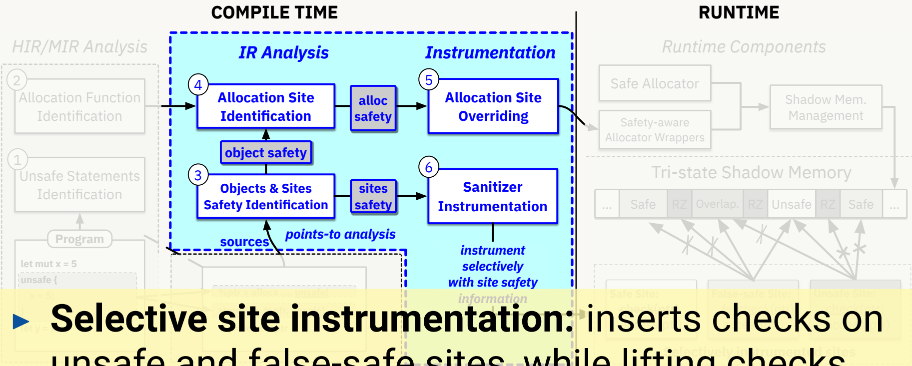


- **`unsafe` analysis** Rust HIR/MIR semantics such as `unsafe` are lost during translation to LLVM IR
- **Bridging two IR forms** Our analysis identifies variable modifications within `unsafe`, and propagates to LLVM IR stage

RustSan: LLVM IR-level



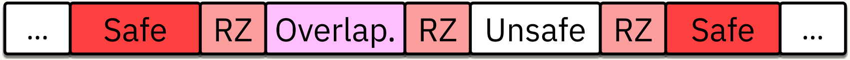
RustSan: LLVM IR-level



- **Selective site instrumentation:** inserts checks on unsafe and false-safe sites, while lifting checks checks on *safe* sites
- **Object safety coloring**: Intercept and instrument object allocations and color objects according to object safety

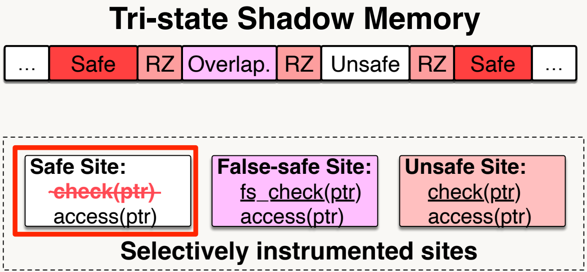
Safety-aware Object Allocation

Tri-state Shadow Memory



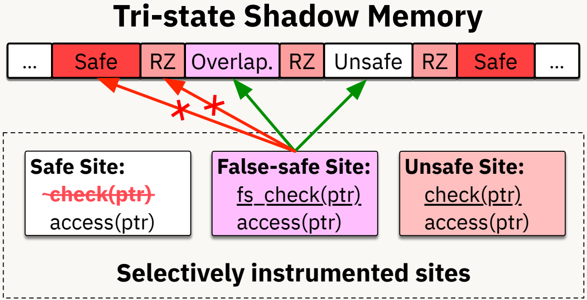
- Heap objects are allocated with different colors according to object safety identified during analysis

Tri-state Shadow Memory Enforcement



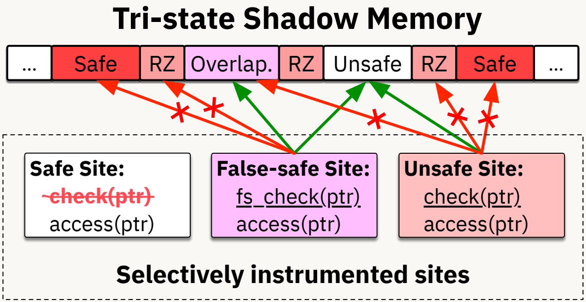
- Checks are *eliminated* on safe sites

Tri-state Shadow Memory Enforcement



- Checks are *eliminated* on safe sites
- False-safe can access unsafe/overlapping object

Tri-state Shadow Memory Enforcement



- Checks are *eliminated* on safe sites
- False-safe can access unsafe/overlapping object
- Unsafe can only access unsafe object

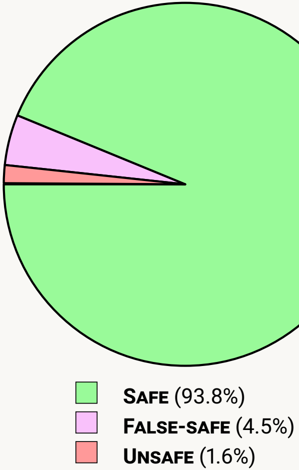
Evaluation

- Detection Capability Evaluation
  - Collect 31 CVEs that ASan can detect and reproduce with RustSan
- Performance Evaluation
  - 20 real-world applications from `Crate.io`

Evaluation: Site Safety Statistics

- Average site safety distribution In 33 applications:

RUSTSAN eliminates 93.8% of ASan checks!

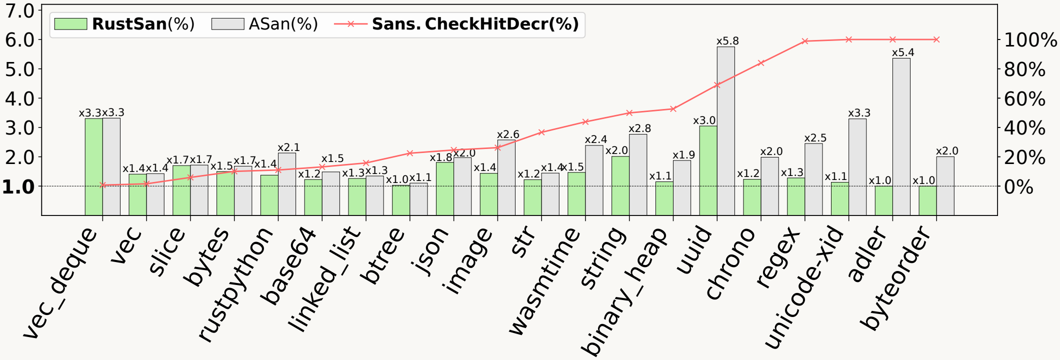


Evaluation: Detection Capability

CVE	Vuln. Class	Detected	*FS/U*	CVE	Vuln. Class	Detected	*FS/U*
CVE-2020-36465	UAF	✓	FS	CVE-2021-45694	Heap Ovf.	✓	FS
CVE-2018-20991	UAF	✓	FS	CVE-2021-26954	UAF	✓	FS
CVE-2019-15551	UAF	✓	FS	CVE-2021-28028	UAF	✓	FS
CVE-2019-25009	UAF	✓	FS	CVE-2021-29933	UAF	✓	FS
CVE-2020-25574	UAF	✓	FS	CVE-2020-35891	UAF	✓	FS
CVE-2020-35858	Stack Ovf.	✓	FS	CVE-2017-1000430	Heap Ovf.	✓	U
CVE-2020-25792	Stack Ovf.	✓	FS	CVE-2020-35861	Heap Ovf.	✓	U
CVE-2020-25791	Stack Ovf.	✓	FS	CVE-2021-25900	Heap Ovf.	✓	U
CVE-2020-25795	UAF	✓	FS	CVE-2020-35906	UAF	✓	U
CVE-2021-45713	UAF	✓	FS	CVE-2021-45720	UAF	✓	U
CVE-2019-16882	UAF	✓	FS	CVE-2020-36464	UAF	✓	U
CVE-2018-21000	Heap Ovf.	✓	FS	CVE-2020-36434	UAF	✓	U
CVE-2019-16140	UAF	✓	FS	CVE-2020-35860	UAF	✓	U
CVE-2021-30455	UAF	✓	FS	CVE-2020-35892	Heap Ovf.	✓	U
CVE-2021-30457	UAF	✓	FS	CVE-2020-35893	Heap Ovf.	✓	U
CVE-2021-28031	UAF	✓	FS				

- RustSan reproduced all detected cases with ASan in memory errors in the `Advisory-DB`
- 67% of bugs(21/31) were detected in a false-site site

Evaluation: Performance



- 62% performance advantage over ASan on average
- 43% less shadow memory check encounter during runtime

More details

- **Implementation details**
  - HIR/MIR analysis improvements over previous works
  - SVF framework extensions for Rust
  - Shadow memory encoding
- **Experiment data omitted in this talk**
  - Ratio of sites and objects of varying safety classification for 33 crates.
  - Real-world performance gains in fuzz testing scenario
- **Thorough analysis of threats to validity**

👉 For more details, please check out our paper!

Thank you

Q&A time!!