ERASan : Efficient Rust Address Sanitizer

Jiun Min*[1], Dongyeon Yu*[1], Seongyun Jeong[1], Dokyung Song[2], Yuseok Jeon[1]

* Equal Contribution

[1]UNIST [2]Yonsei University

Jiun Min
E-mail : min1905@unist.ac.kr

Dongyeon Yu
E-mail : dy3199@unist.ac.kr

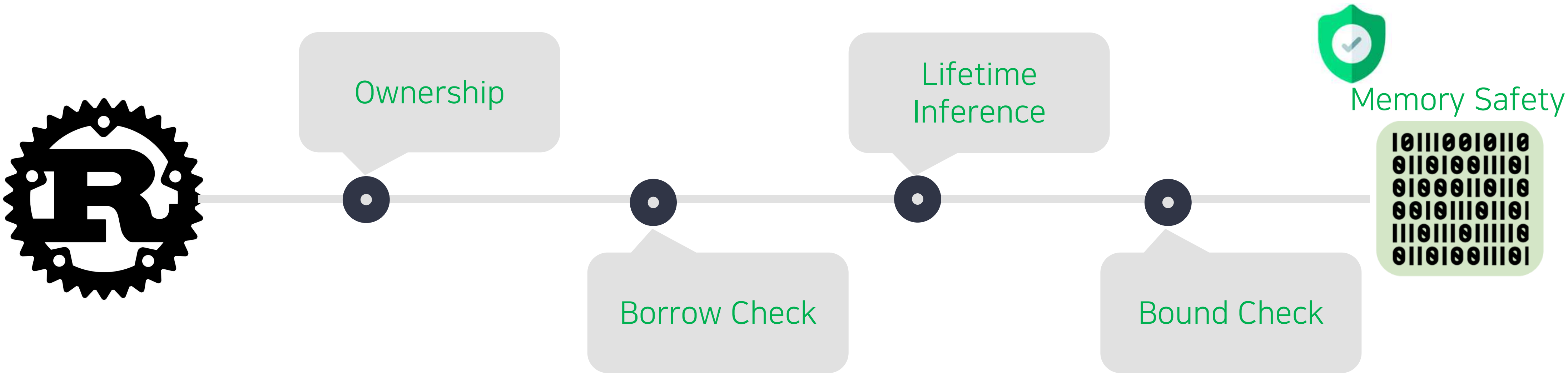
Seongyun Jeong
E-mail : dy3199@unist.ac.kr

Dokyung Song
E-mail : dokyungs@yonsei.ac.kr

Yuseok Jeon
E-mail : ysjeon@unist.ac.kr

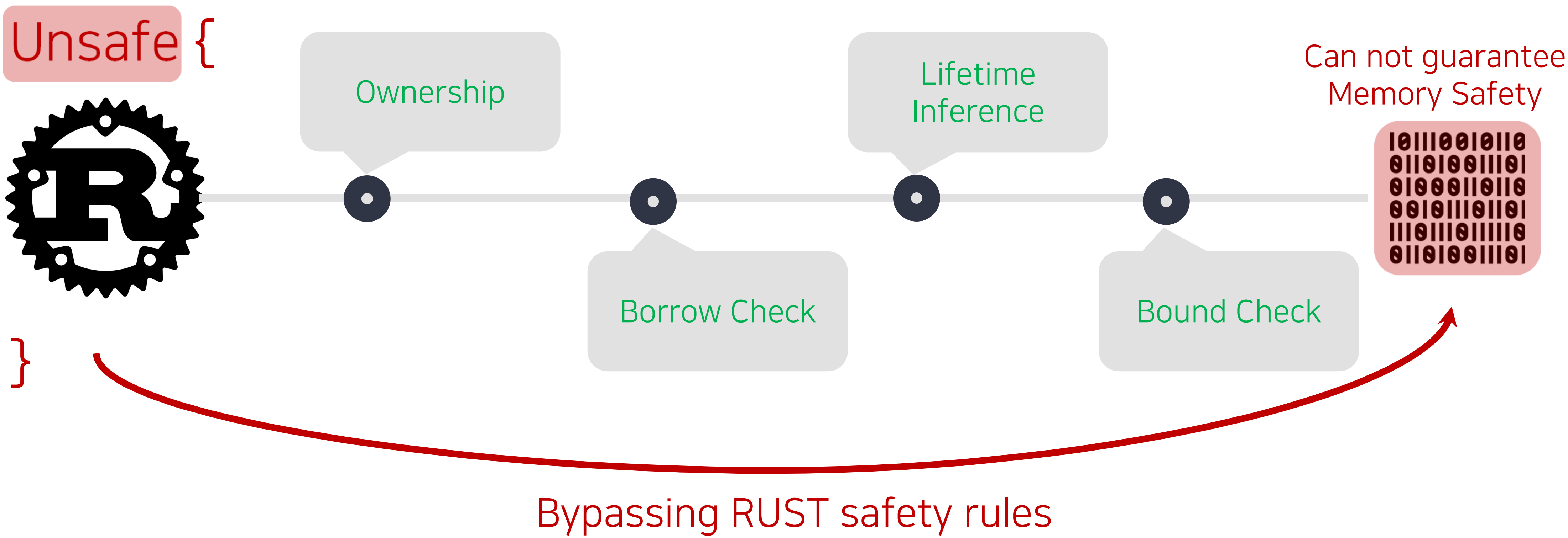
RUST Safety Rules

❖ RUST is designed to guarantee memory safety by leveraging the four main safety rules.



Unsafe RUST

❖ **Unsafe** RUST can not guarantee memory safety to bypass safety rules.

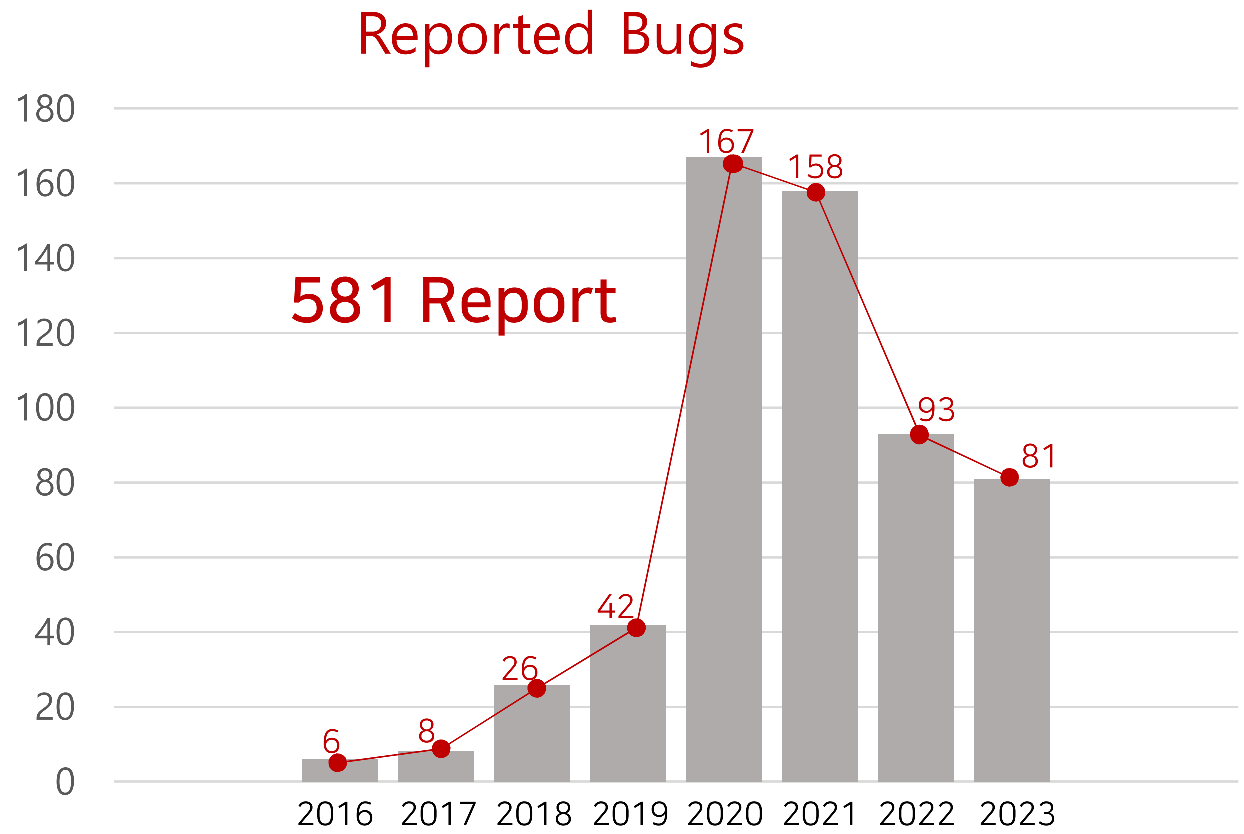


RUST Memory Bugs

❖ Over the seven years, 581 reported bugs have been detected in the RUST program.

Unsafe {

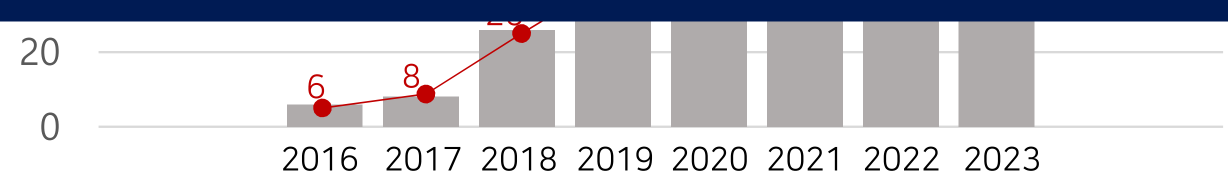
}



RUST Memory Bugs

❖ Over the seven years, 581 reported bugs have been detected in the RUST program.

We should detect bugs caused by using Unsafe RUST.

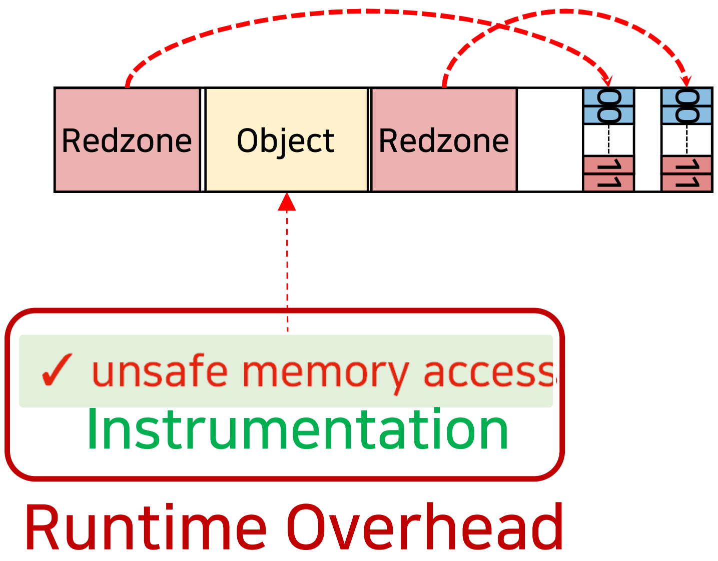


Address Sanitizer

❖ Address Sanitizer can detect memory safety violation such as UAF and Buffer Overflow.

Unsafe {

}



Address Sanitizer

: Detect memory safety violations

- Inserts poisoned Redzone around objects.
- Instrument all memory access to check validity.
- However, it generates significant runtime overhead.
- It incurs about 334% overhead on RUST program.

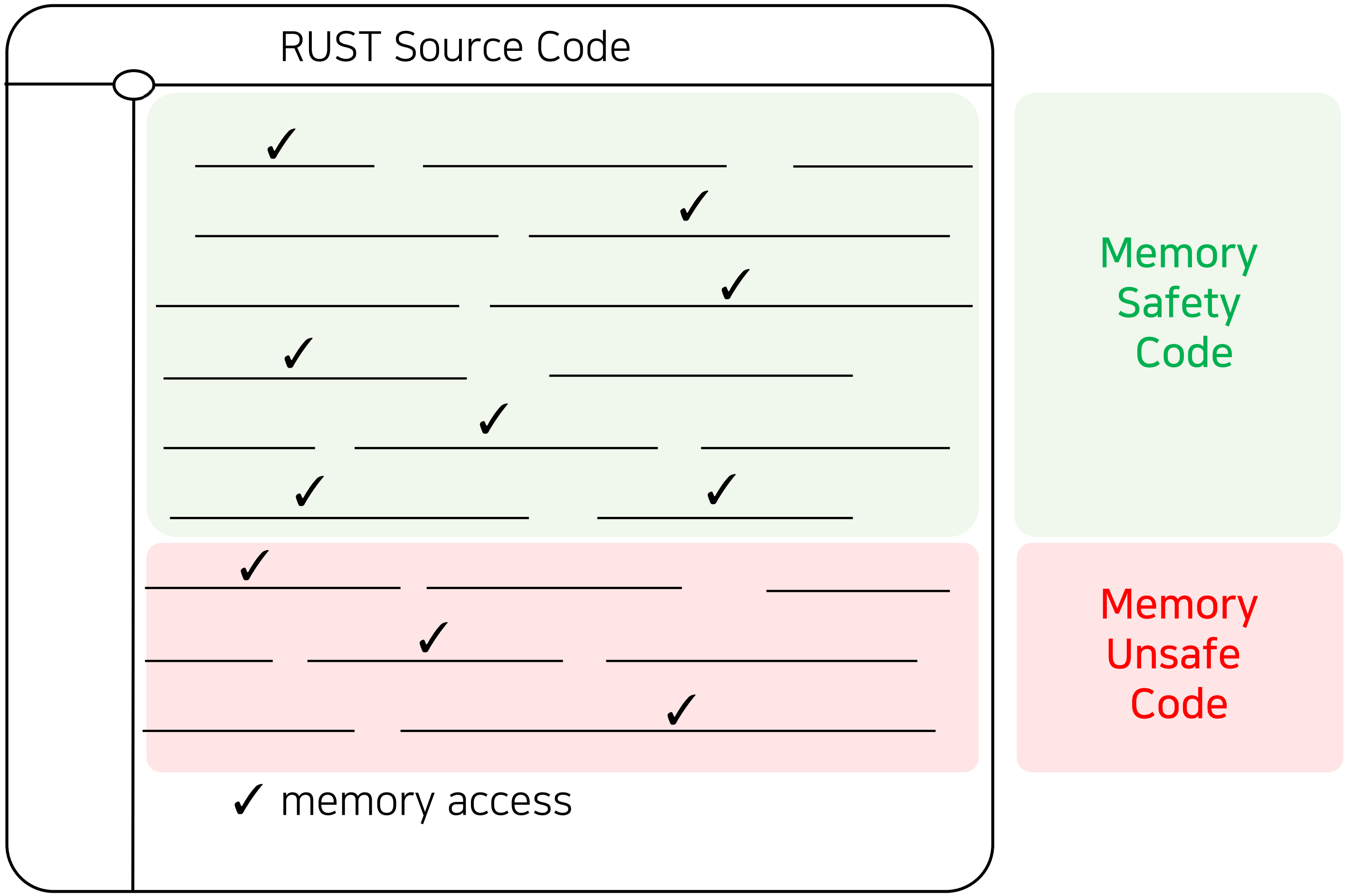
Address Sanitizer

- ❖ Address Sanitizer can detect memory safety violation such as UAF and Buffer Overflow.

Should we apply the Address Sanitizer
to all RUST source codes?

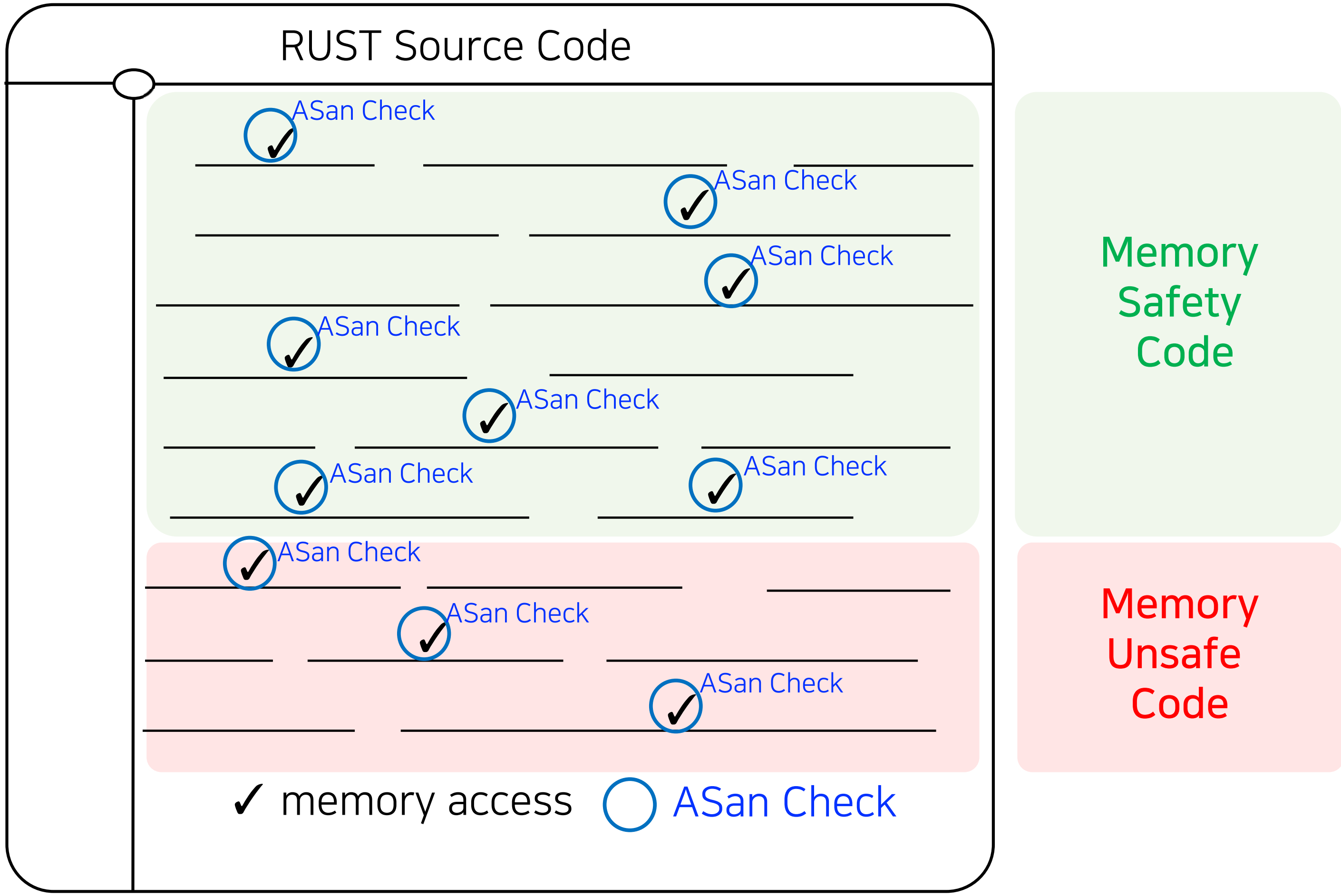
Address Sanitizer for RUST

❖ Address Sanitizer is used to detect temporal and spatial memory violation bugs in RUST.



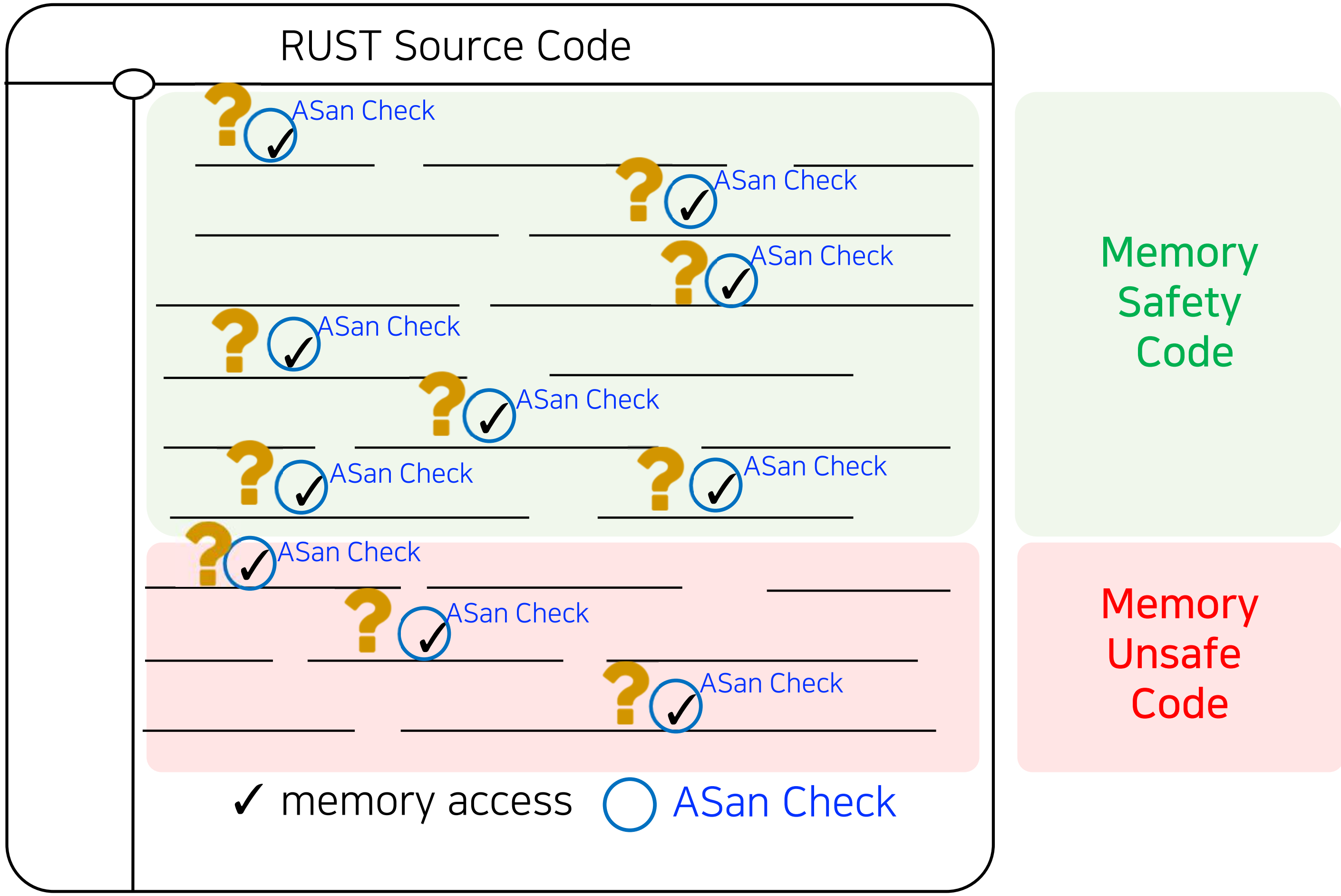
Address Sanitizer for RUST

❖ For detecting memory bugs, Address Sanitizer instruments all memory accesses.



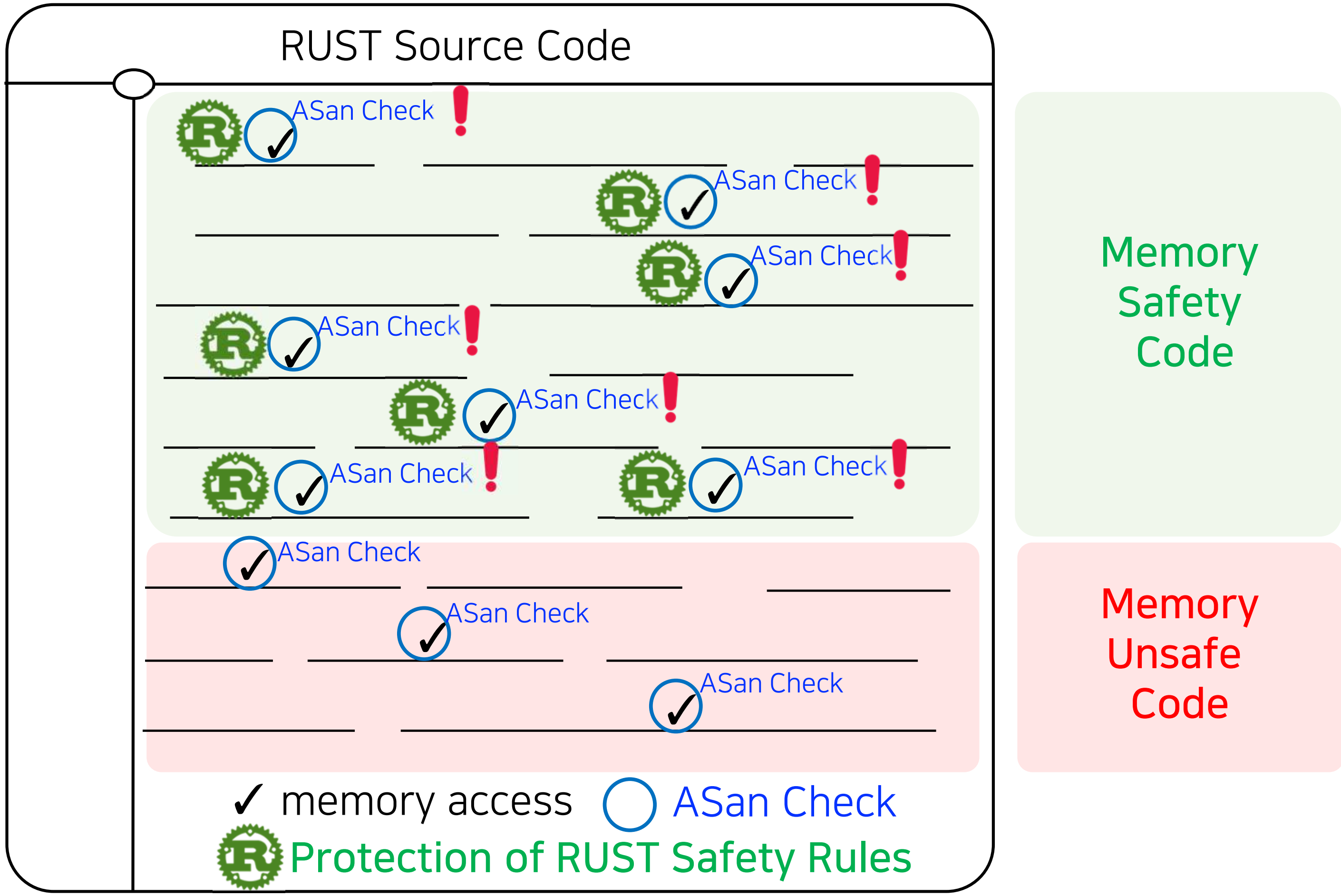
Unnecessary Checks of RUST Address Sanitizer

❖ Address Sanitizer in RUST instruments all memory accesses regarding RUST safety rules.



Unnecessary Checks of RUST Address Sanitizer

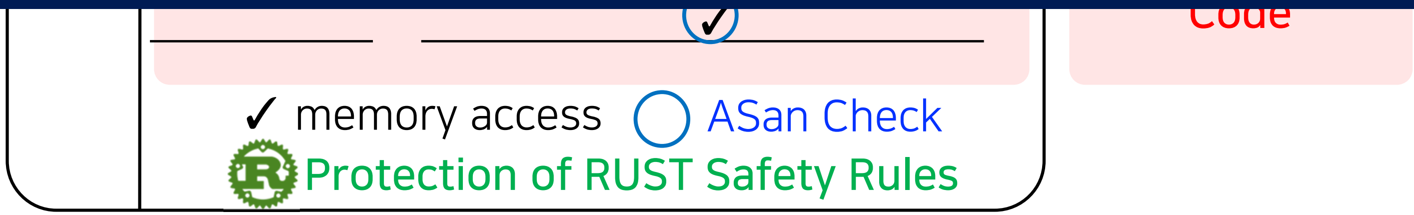
❖ Address Sanitizer performs redundant and unnecessary memory access checks.



Unnecessary Checks of RUST Address Sanitizer

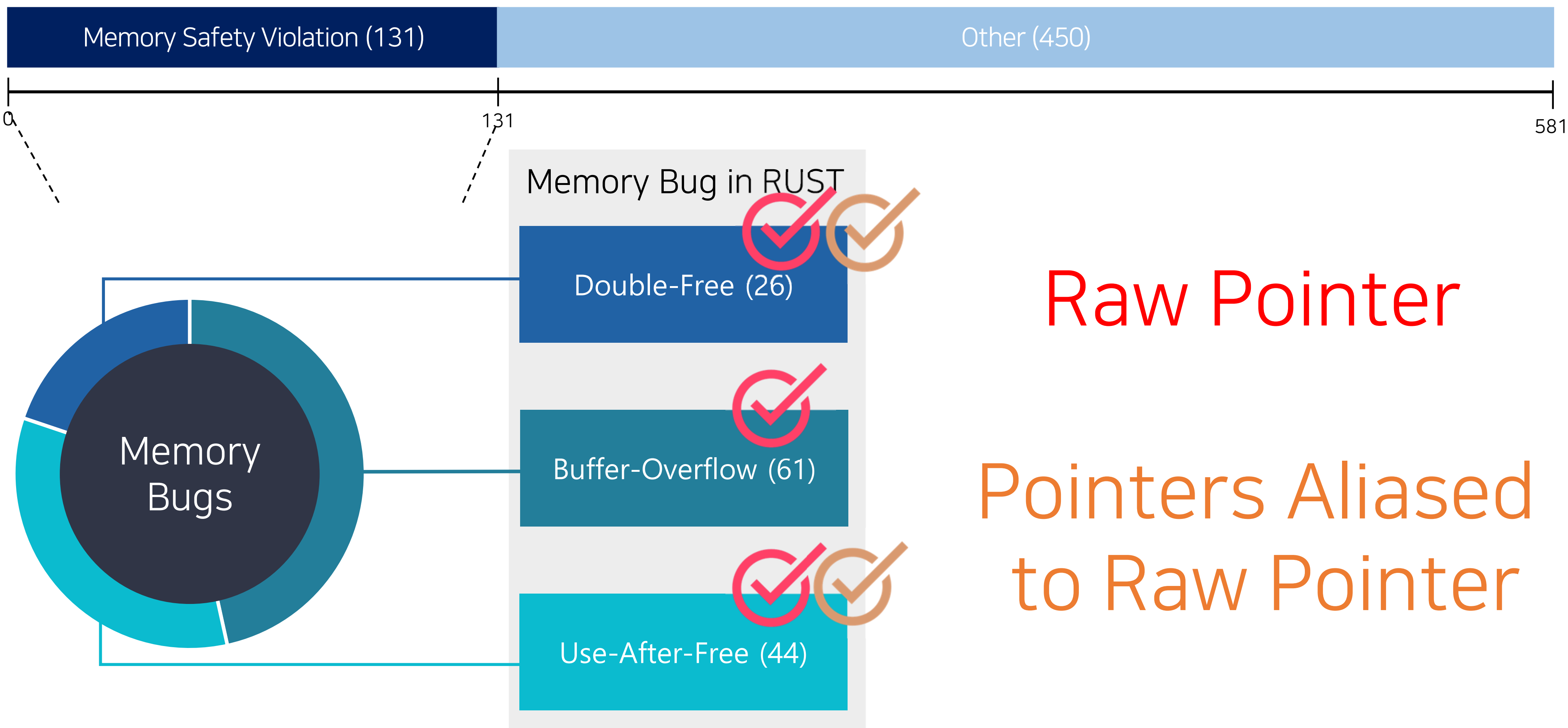
❖ Address Sanitizer performs redundant and unnecessary memory access checks.

To reduce this unnecessary overhead,
we survey when Rust memory bugs occurs.



Real-World RUST Memory Bugs Analysis

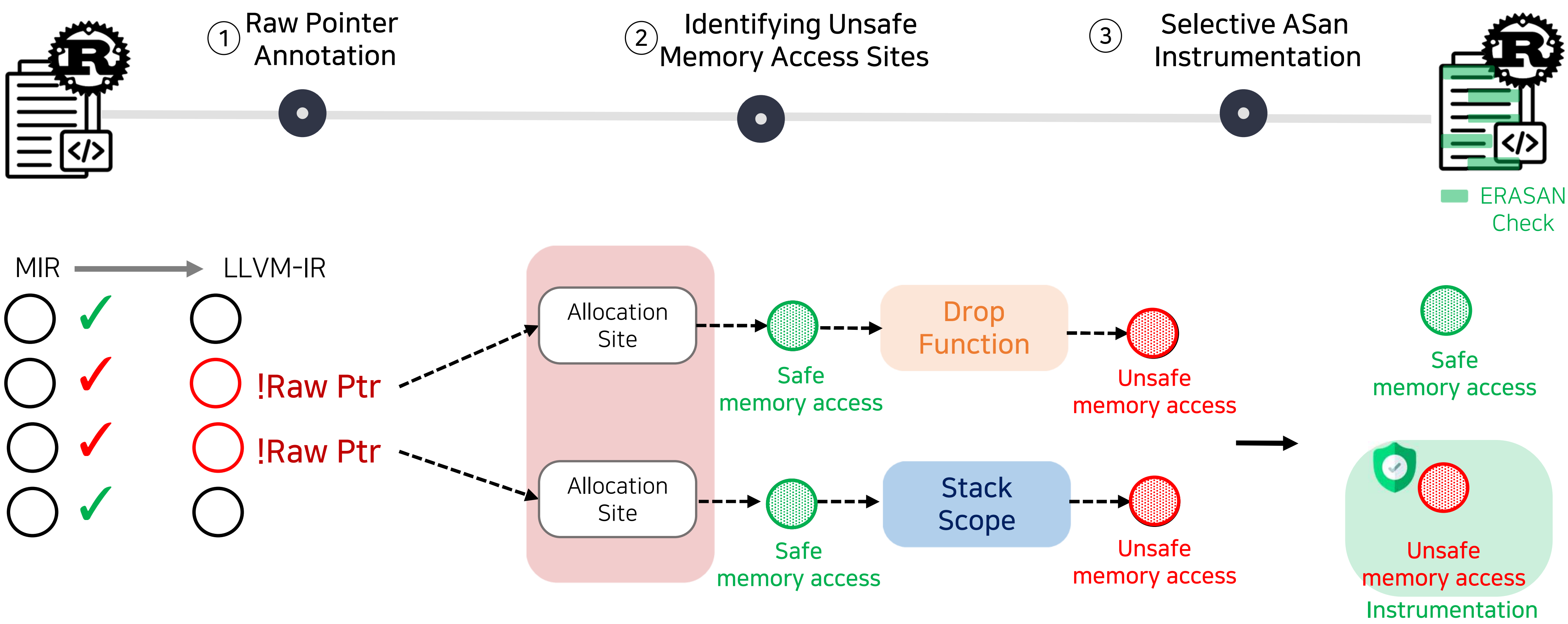
❖ Analyze the 581 Rust bug reports in the RustSec Advisory Database over seven years.



Key Finding

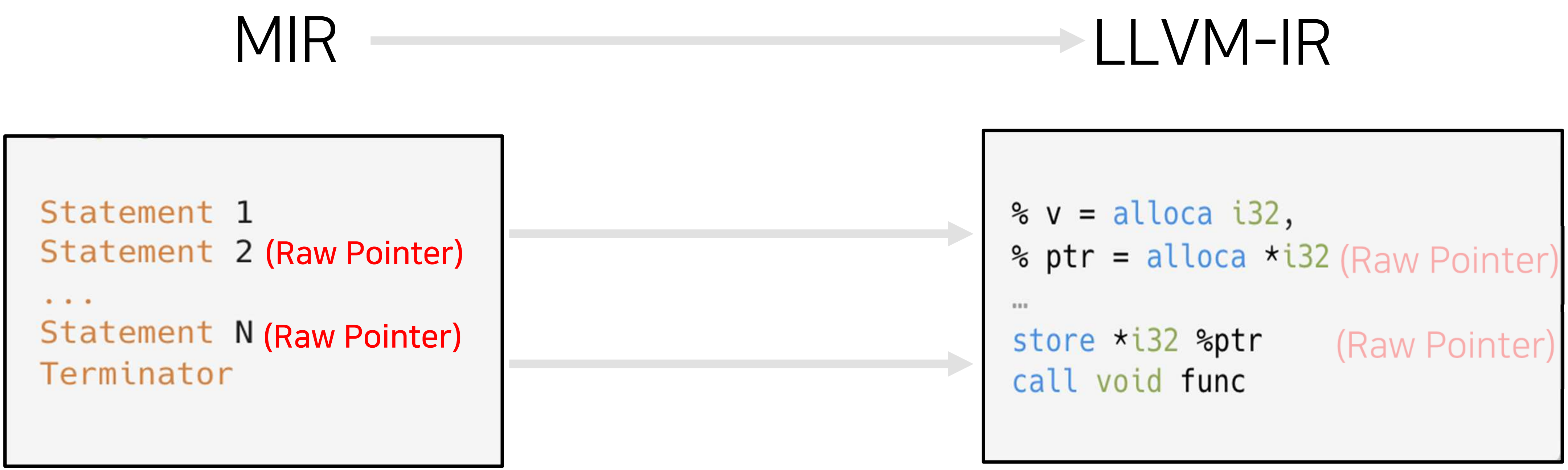
Raw Pointer and Alias pointer with raw pointer	**Key Finding** Temporal and Spatial Memory Safety Violation Bugs can be triggered by Raw Pointer. The safe pointer that pointer aliased to raw pointer can trigger Temporal Memory Safety Violation Bugs.

ERASAN Overview



Raw Pointer Annotation

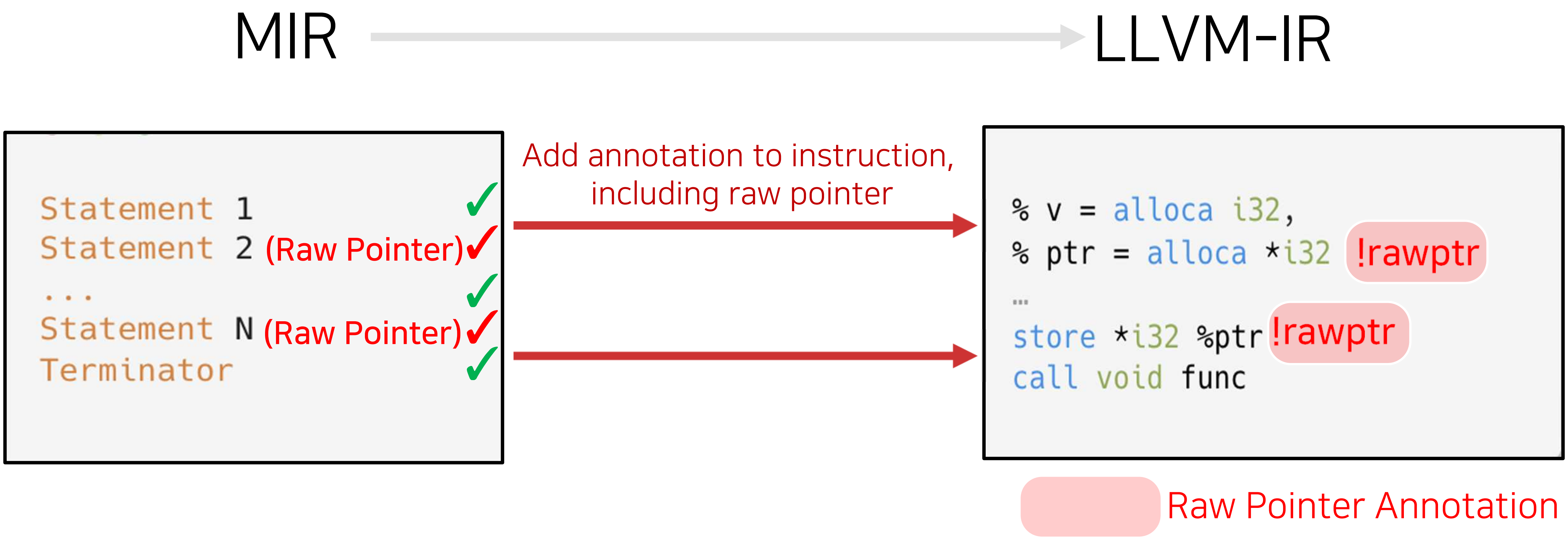
❖ Raw Pointer information is unique type existing during RUST compilation step until MIR.



Raw Pointer Information does not exist in LLVM-IR.

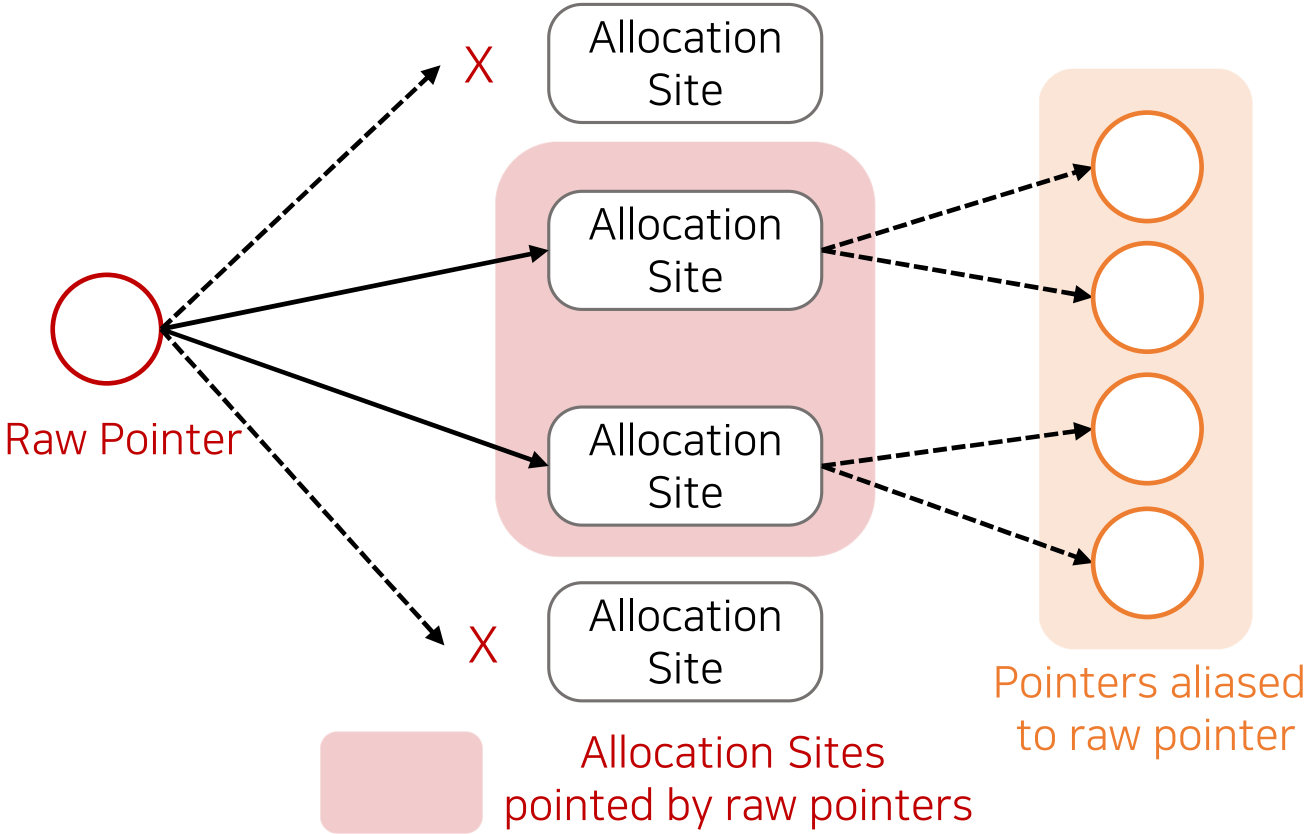
Raw Pointer Annotation

- Raw Pointer information is a unique type existing during the RUST compilation step until MIR.
- ERASAN annotates to llvm instructions related to the raw pointer.



Identifying Memory Allocation Sites

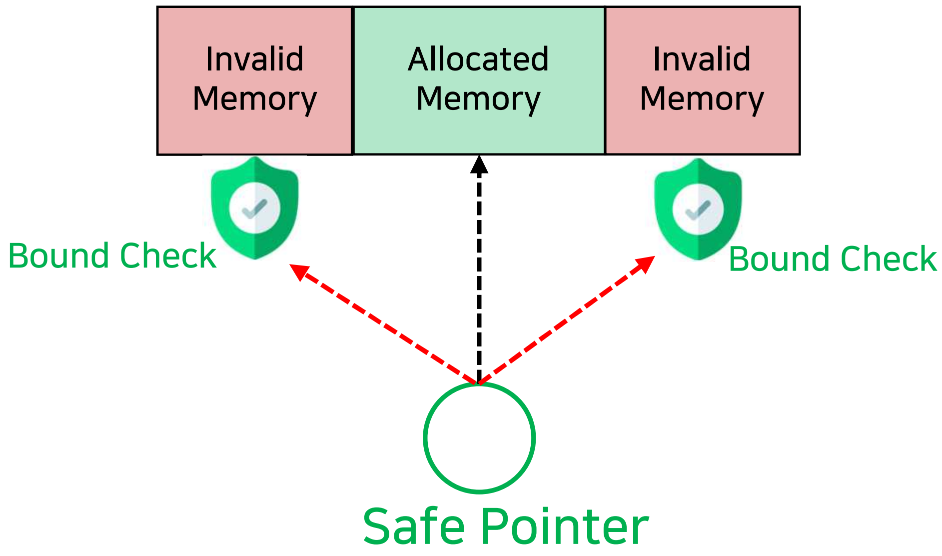
- ERASAN identifies all the memory allocation sites that can be pointed to by raw pointers.



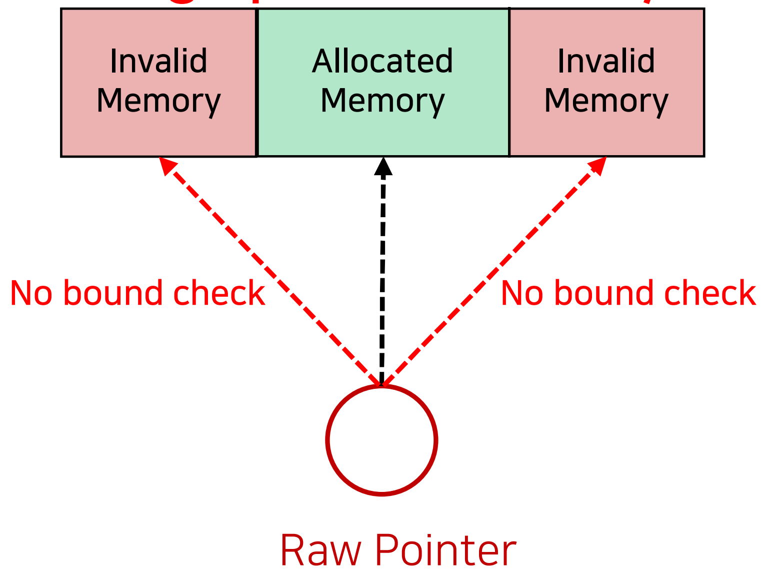
Preventing Spatial Memory Violation Bugs

- Memory access by a safe pointer (e.g., reference) ensured no spatial memory safety violation.
- Memory access by a raw pointer causes a spatial memory bug.

Prevent spatial memory bug

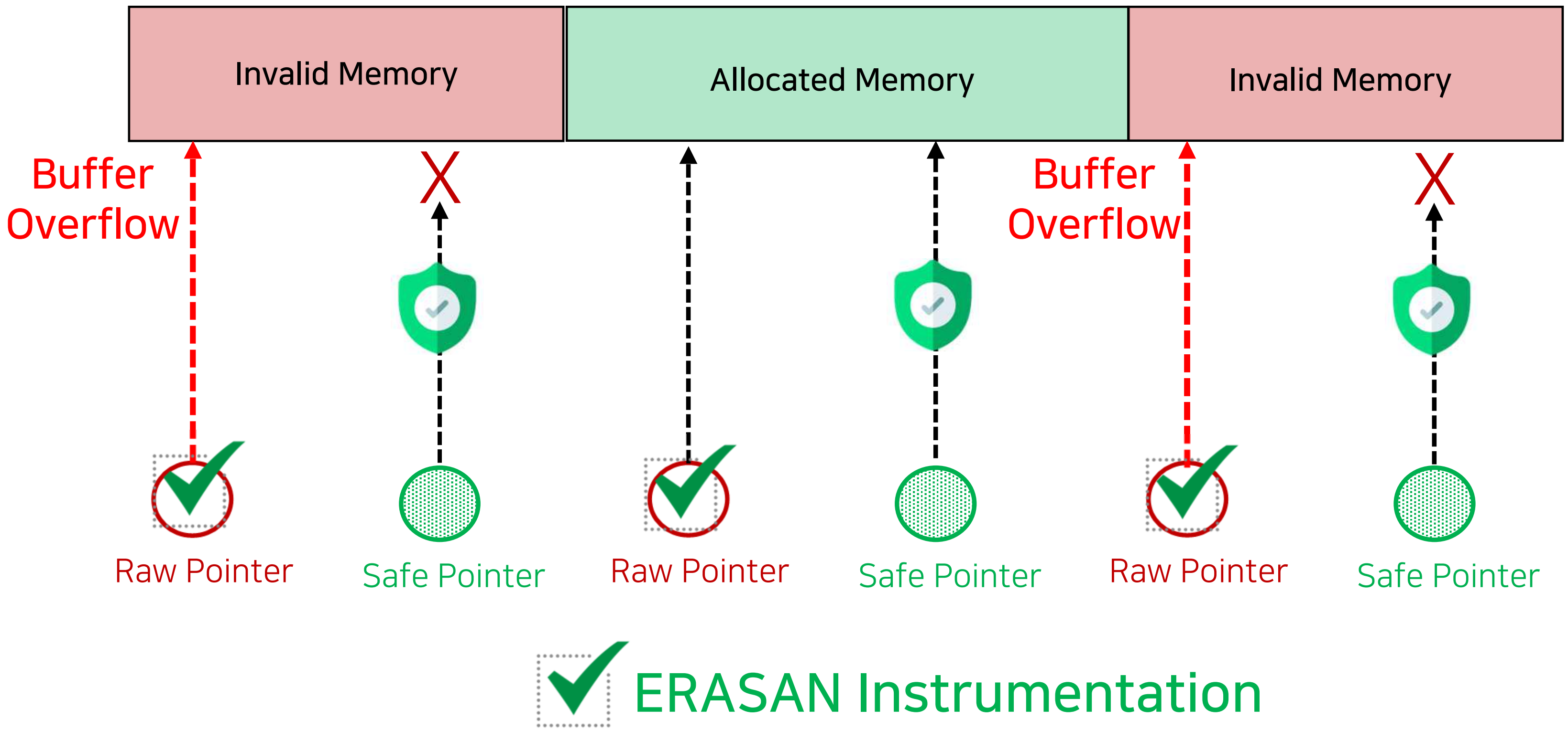


Causing spatial memory bug



Detecting Spatial Memory Violation Bugs

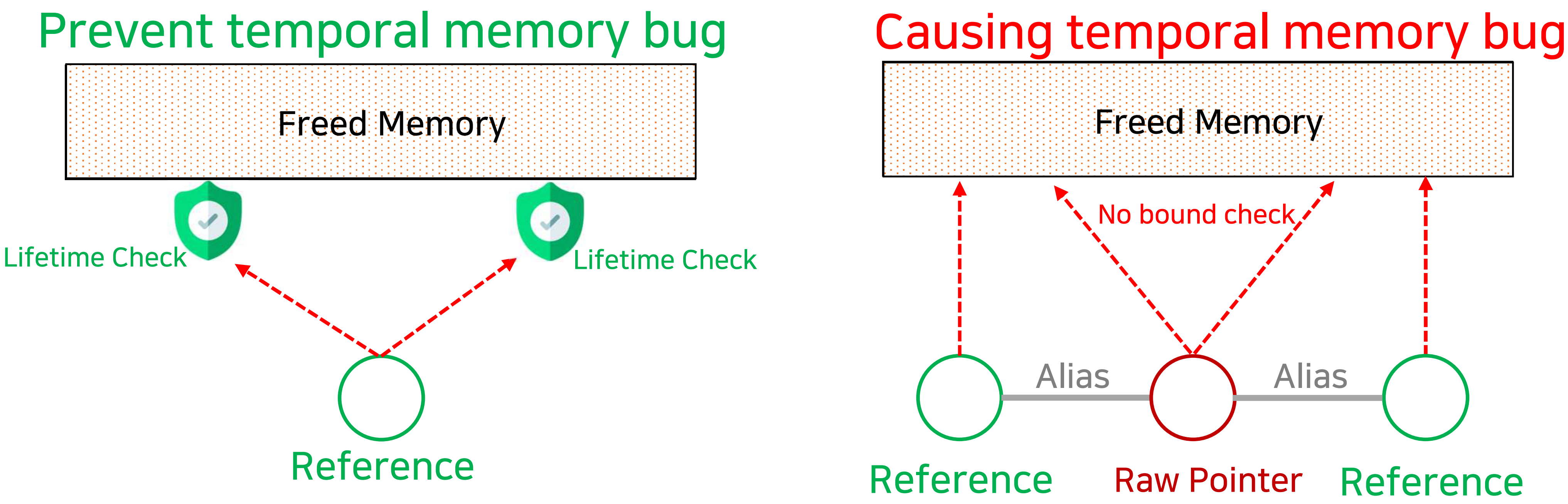
❖ ERASAN instrument to raw pointer access to prevent buffer overflow.



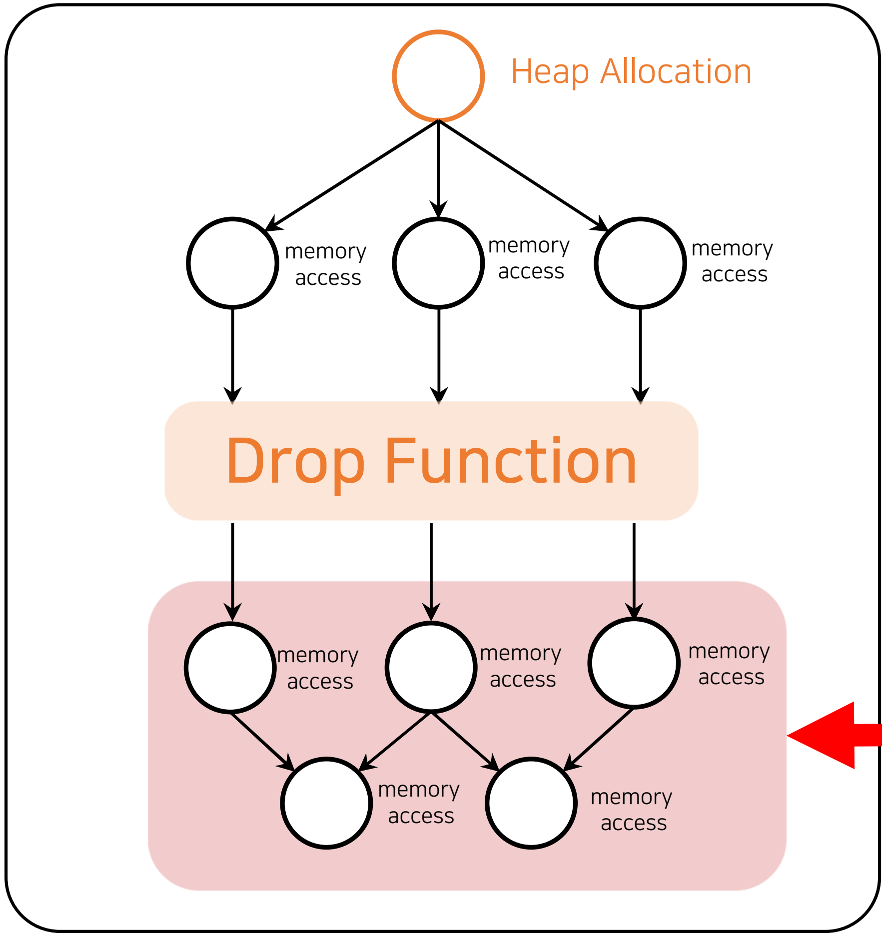
① Raw Pointer Annotation ② Identifying Unsafe Memory Access ③ Selective ASan Instrumentation

Preventing Temporal Memory Violations

- Memory access by a safe pointer (e.g., reference) ensured no temporal memory safety violation.
- The pointer aliased to raw pointer can be exposed to use-after-free.



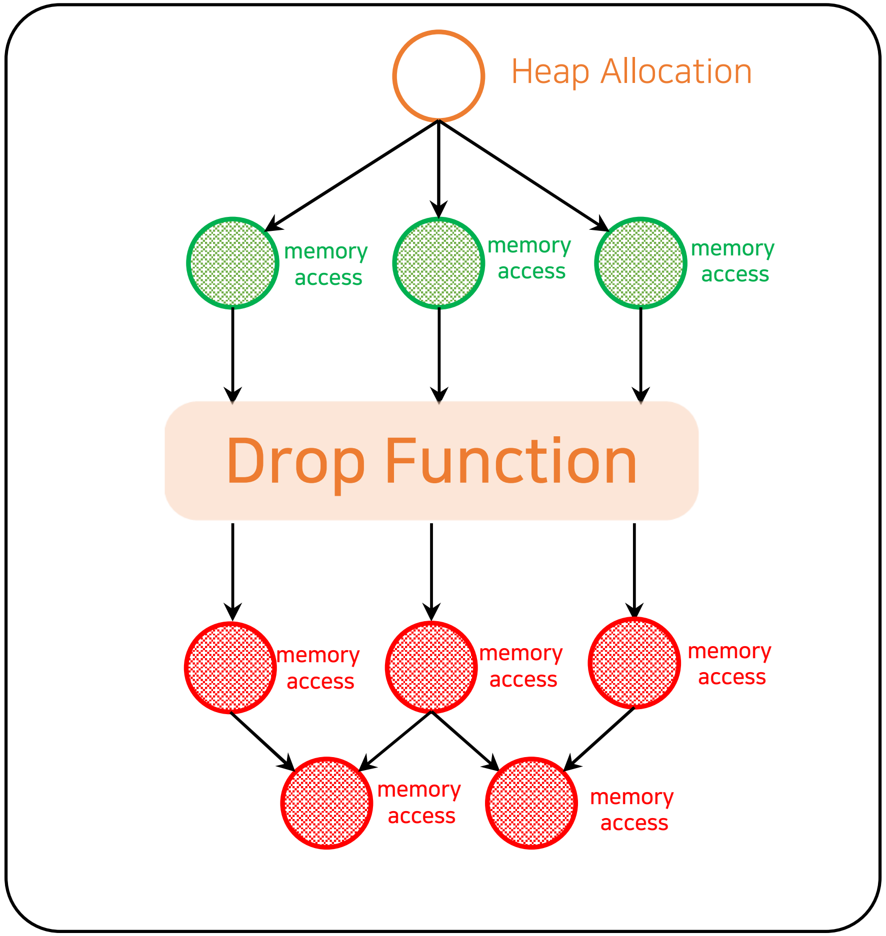
Checking Memory Access Sites After Drop



- Use-after-free occur only after the drop function

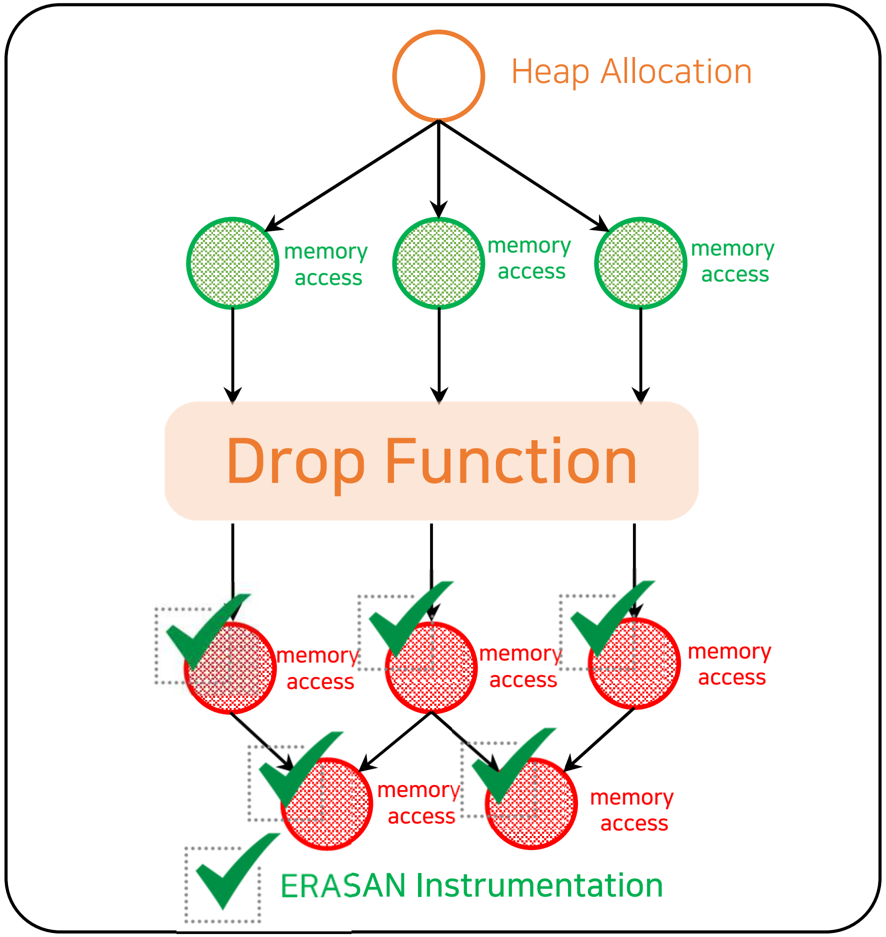
Memory accesses that can cause UAF

Checking Memory Access Sites After Drop



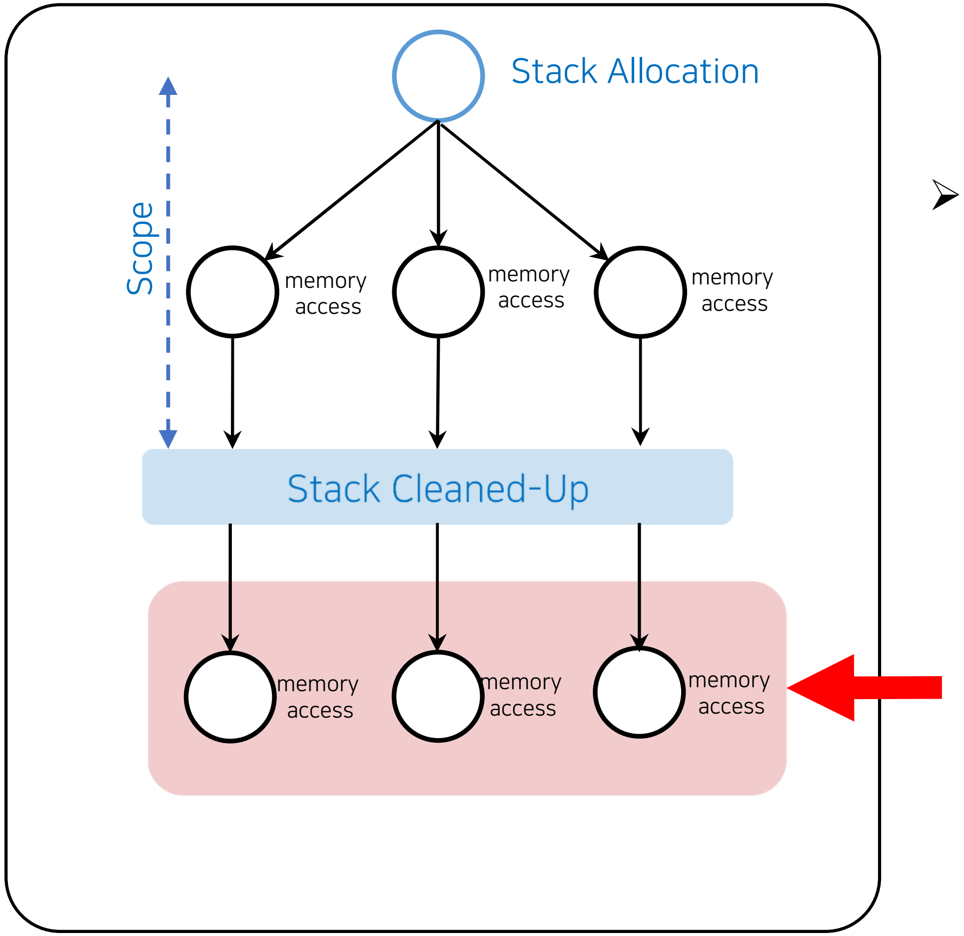
- Use-after-free occur only after the drop function
- Memory accesses after drop function is vulnerable.

Checking Memory Access Sites After Drop



- Use-after-free occur only after the drop function
- Memory accesses after drop function is vulnerable.
- ERASAN checks only memory accesses after drop.

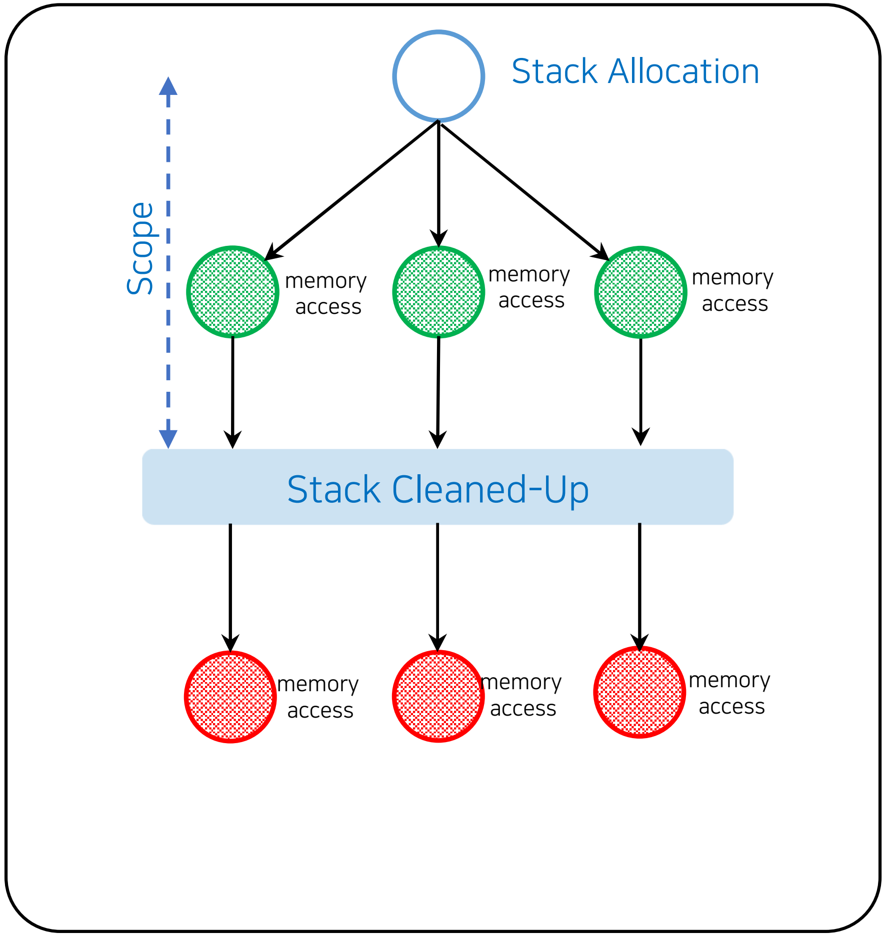
Checking Memory Access Sites After Scope



- The Use-After-Free occur only after stack cleaned-up

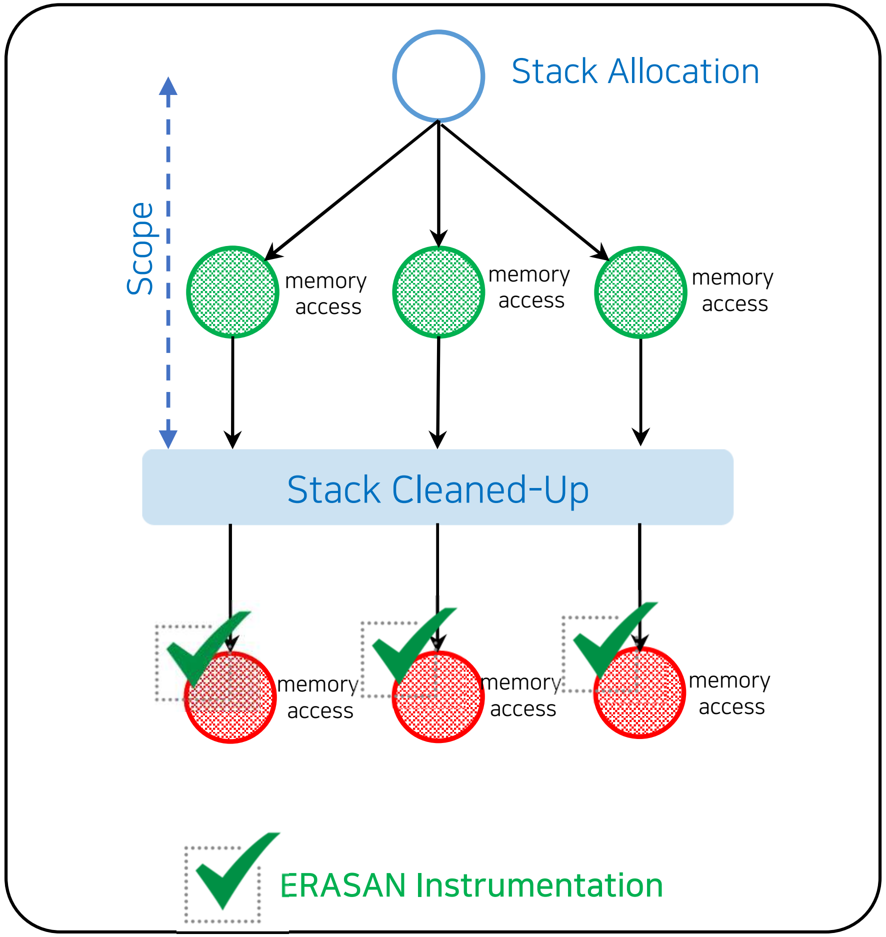
Memory accesses that can cause UAF

Checking Memory Access Sites After Scope



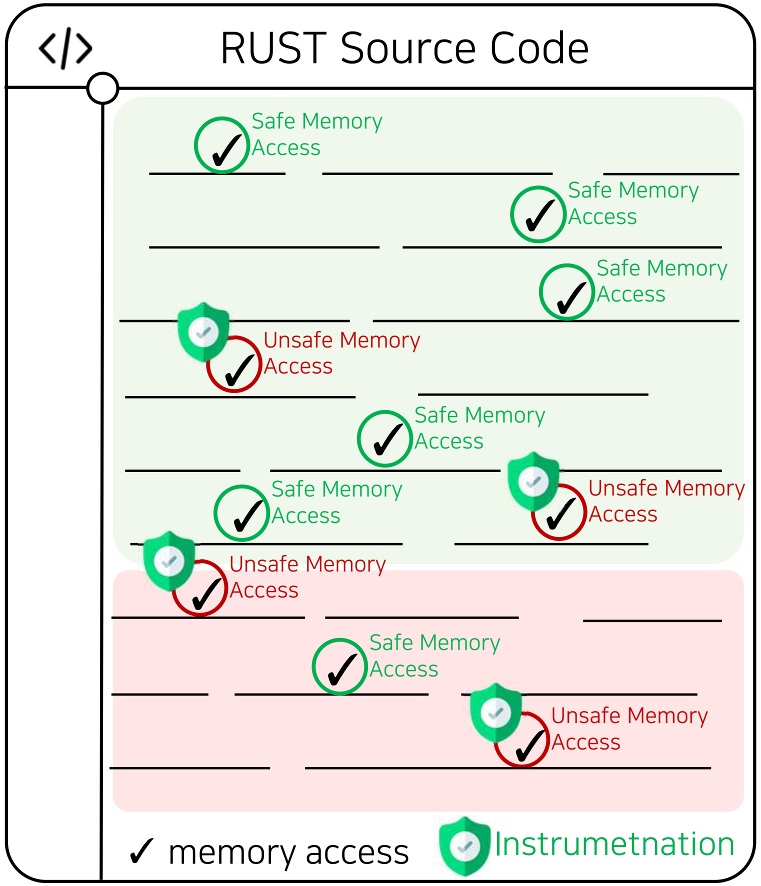
- The Use-After-Free occur only after stack cleaned-up
- Memory access after cleaned-up is vulnerable.

Checking Memory Access Sites After Scope



- The Use-After-Free occur only after stack cleaned-up
- Memory access before the stack is cleaned-up is safe
- ERASAN checks only memory accesses after scope.

Selective ASan Check Instrumentation



ERASAN

: Efficient Rust Address Sanitizer

- ERASAN identifies which memory accesses are unsafe.
- ERASAN instrument only unsafe memory access sites.

Evaluation

ERASAN Evaluation

- Unnecessary Check Reduction
- Runtime Overhead
- Bug Detection Capability
- Comparison with ASAN--
- Compile-time Overhead

Baseline Configuration

Baselines

- ASAN is native address sanitizer, unmodified version.
- ERASAN-unsafe conducts an unsafe block-based static analysis approach.
- ERASAN-rawptr checks all memory accesses through all raw pointers and aliased. pointers, turning off optimization

Our Approach

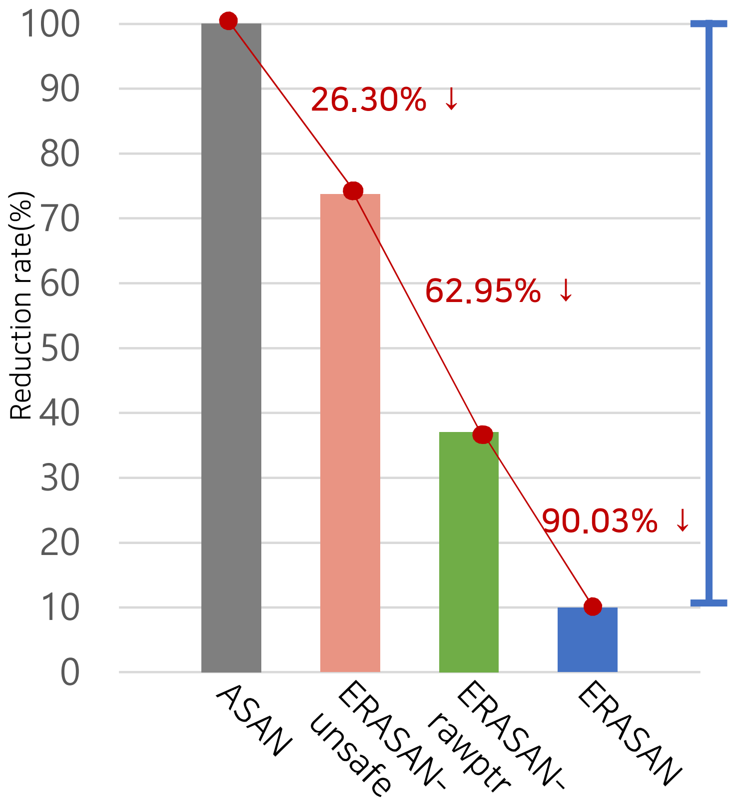
- ERASAN adapted all proposed approaches.

Unnecessary Check Reduction

- Evaluate how ERASAN effectively removes the ASan checks using 23 benchmarks in static time.
- Removes 90.03% of sanitizer checks achieving a higher reduction rate than the other baseline.

	number of checks (#)	Reduction rate (%)
Asan	43,022	-
ERASAN-unsafe	36,116	26.30% ↓
ERASAN-rawptr	24,521	62.95% ↓
ERASAN	10,197	90.03% ↓

Unnecessary check reduction



ERASAN removes

90.03%

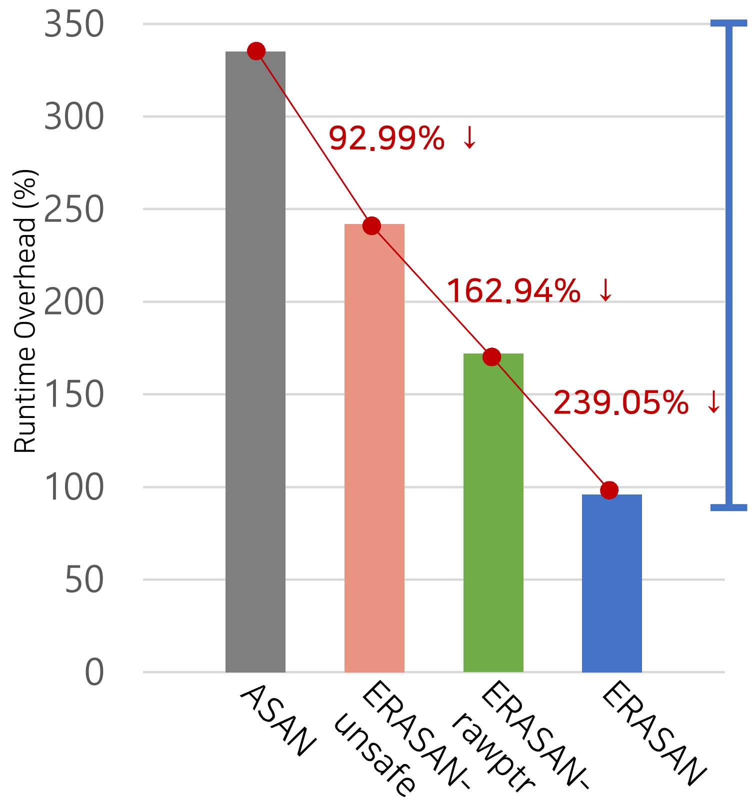
sanitizer checks

Runtime Overhead

- Evaluate how ERASAN runtime overhead reduction due to reduced ASAN's check instrumentation.
- Improve 239.05% performance achieving a higher improvement than the other baseline.

	overhead (%)	Reduction rate (%)
ASan	334.98%	-
ERASAN-unsafe	241.99%	92.99% ↓
ERASAN-rawptr	172.04%	162.94% ↓
ERASAN	95.94%	239.05% ↓

Runtime Overhead

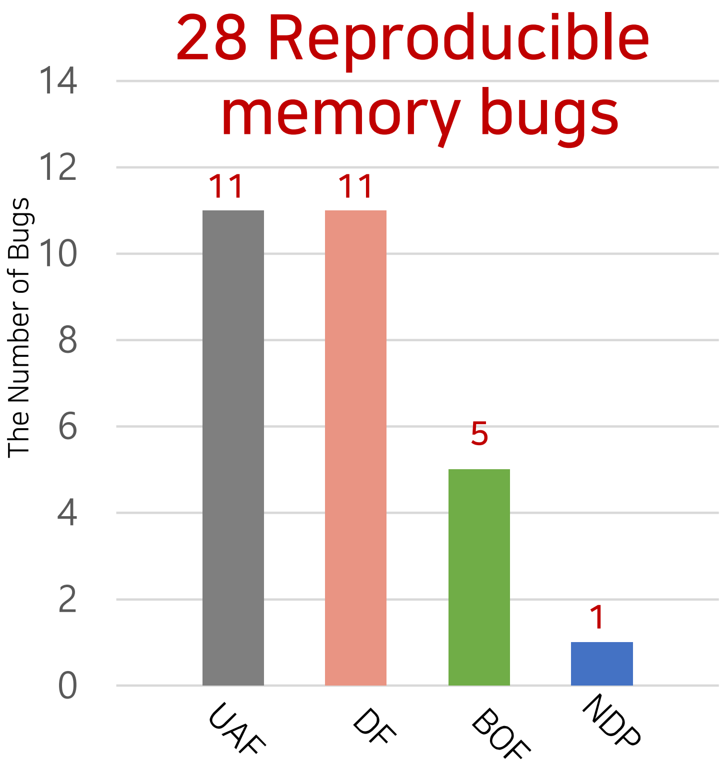


ERASAN improves
239.05%
performance

Bug Detection Capability

- We collect 28 reproducible memory bugs to evaluate ERASAN against real-world memory bugs.

- ERASAN successfully detects all memory bugs in the 28 test cases.



Bug Type	Number	ASAN	ERASAN
Use -After-Free	11	✓	✓
Double-Free	11	✓	✓
Buffer-Overflow	5	✓	✓
Null-Pointer-Dereference	1	✓	✓

ERASAN

Clearly

detects all test cases

Conclusion

- ERASAN efficiently reduces performance overhead, which has same bug detection capability as ASAN.
  - Remove 90.03% of existing Asan Checks.
  - Significantly reduce ASan performance overhead by an average of 239.05%.
  - Successfully detect 28 real-world memory bugs.
  - Eliminate 56.88% more sanitizer checks than the state-of-the-art research (ASAN--).

Thank you

[Paper]

ERASAN : Efficient Rust Address Sanitizer

[Open Soruce]

ERASAN Github repository

Jiun Min
E-mail : min1905@unist.ac.kr

Dongyeon Yu
E-mail : dy3199@unist.ac.kr

Seongyun Jeong
E-mail : dy3199@unist.ac.kr

Dokyung Song
E-mail : dokyungs@yonsei.ac.kr

Yuseok Jeon
E-mail : ysjeon@unist.ac.kr