RustSan: Retrofitting AddressSanitizer for Efficient Sanitization of Rust

Kyuwon Cho, Jongyoon Kim, Kha Dinh Duy, Hajeong Lim, and Hojoon Lee, *Sungkyunkwan University*

https://www.usenix.org/conference/usenixsecurity24/presentation/cho-kyuwon

Safe Programming Language

- C/C++ (unsafe)

- Rust is seeing widespread adoption

(Mostly) Safe Programming Language

- Safety Guarantees are not free
  - Rust enforces complex semantics onto the Programmer!

- unsafe Rust
  - Tradeoff between safety and flexibility
  - Raw Pointer Access
  - Invoke unsafe foreign functions (e.g., C/C++)

Unsafe Rust

- The use of unsafe Rust is nearly the sole source of memory errors in Rust programs

- Solutions?
  - Static Analysis to detect unsafe Rust memory Errors
    - Limited Detection Capability!

  - Runtime Isolation of safe Rust from unsafe Rust in programs
    - Contains the impact rather than detect

Revamping Existing Techniques

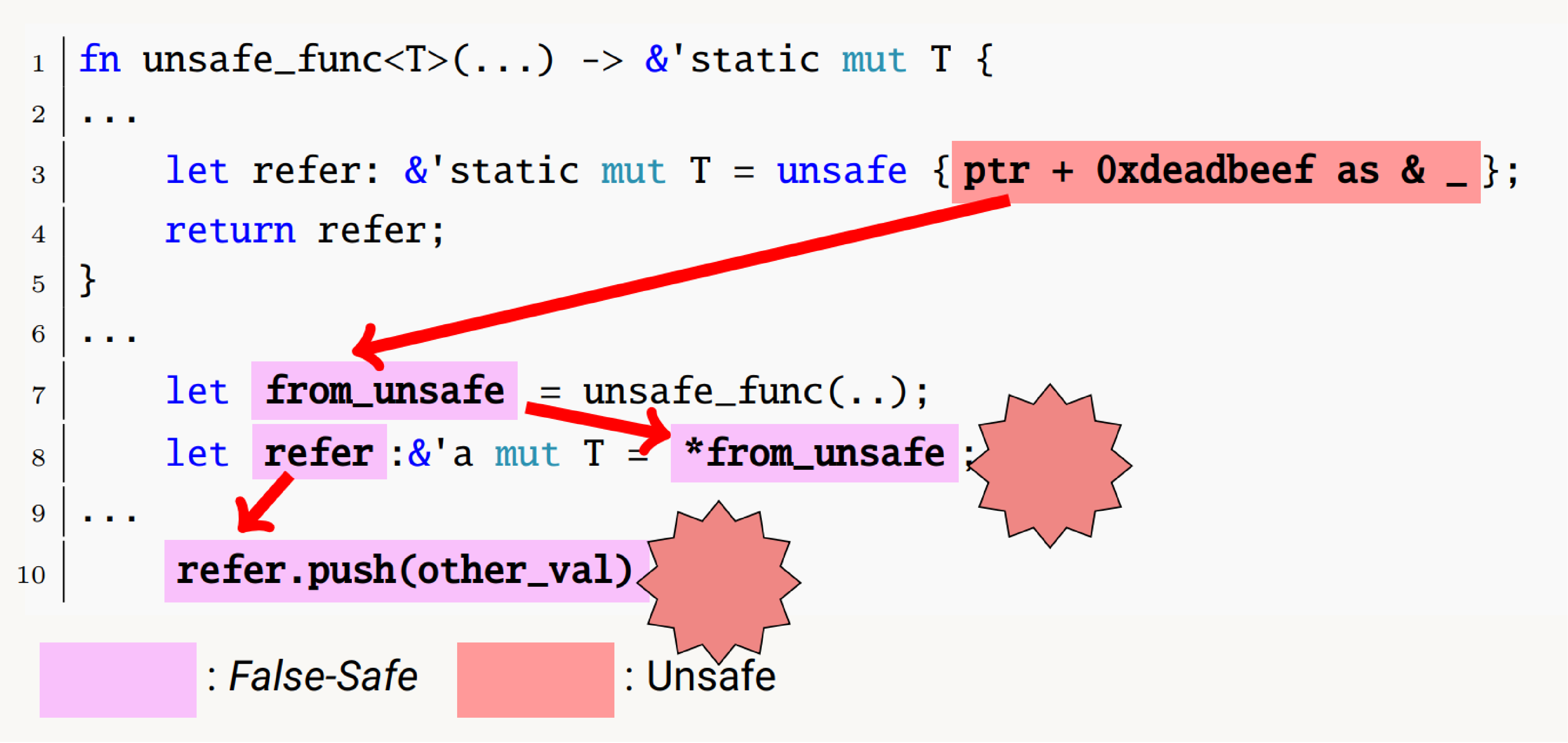
- Fuzzing and Sanitization

- Address Sanitizer (Asan)
  - LLVM-based (easily integrated into Rust)

Address Sanitizer

- Highly compatible and versatile memory error detector
- Intended for C/C++
- Maintains a shadow memory that represents the validity of the process virtual address space
- However
  - Rust is already largely safe (unnecessary checks from Asan)

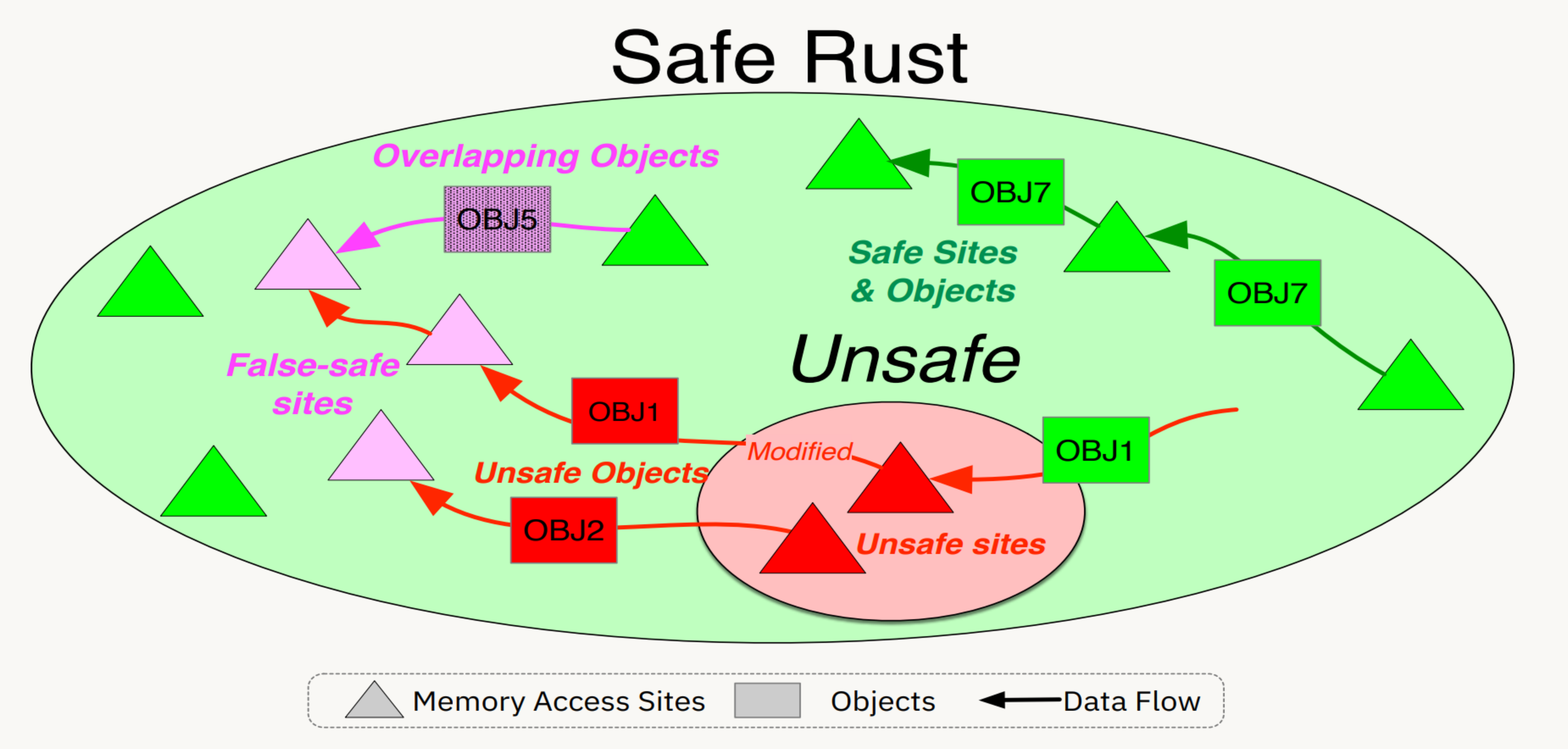
Propagation of Unsafe in Rust



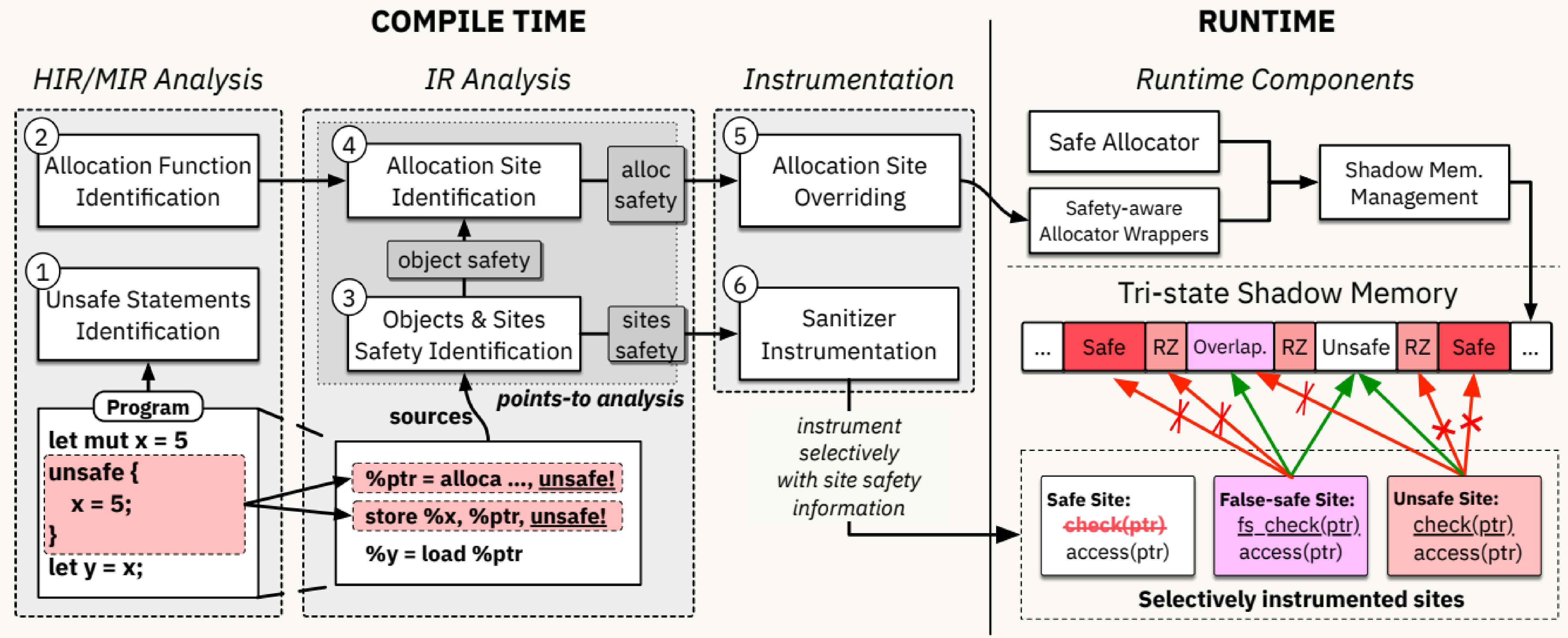
```
fn unsafe_func<T>(...) -> &'static mut T {
...
    let refer: &'static mut T = unsafe {ptr + 0xdeadbeef as & _};
    return refer;
}
...
    let from_unsafe = unsafe_func(..);
    let refer :&'a mut T = *from_unsafe;
...
    refer.push(other_val)
```

🟪 : *False-Safe* 🟥 : Unsafe

Dataflow Analysis



RustSan



COMPILE TIME

RUNTIME

HIR/MIR Analysis

2 Allocation Function Identification

1 Unsafe Statements Identification

Program

```
let mut x = 5
unsafe {
  x = 5;
}
let y = x;
```

IR Analysis

4 Allocation Site Identification

alloc safety

object safety

3 Objects & Sites Safety Identification

sites safety

points-to analysis

sources

```
%ptr = alloca ..., unsafe!
store %x, %ptr, unsafe!
%y = load %ptr
```

Instrumentation

5 Allocation Site Overriding

6 Sanitizer Instrumentation

instrument selectively with site safety information

Runtime Components

Safe Allocator

Safety-aware Allocator Wrappers

Shadow Mem. Management

Tri-state Shadow Memory

...	Safe	RZ	Overlap.	RZ	Unsafe	RZ	Safe	...

Safe Site:
~~check(ptr)~~
access(ptr)

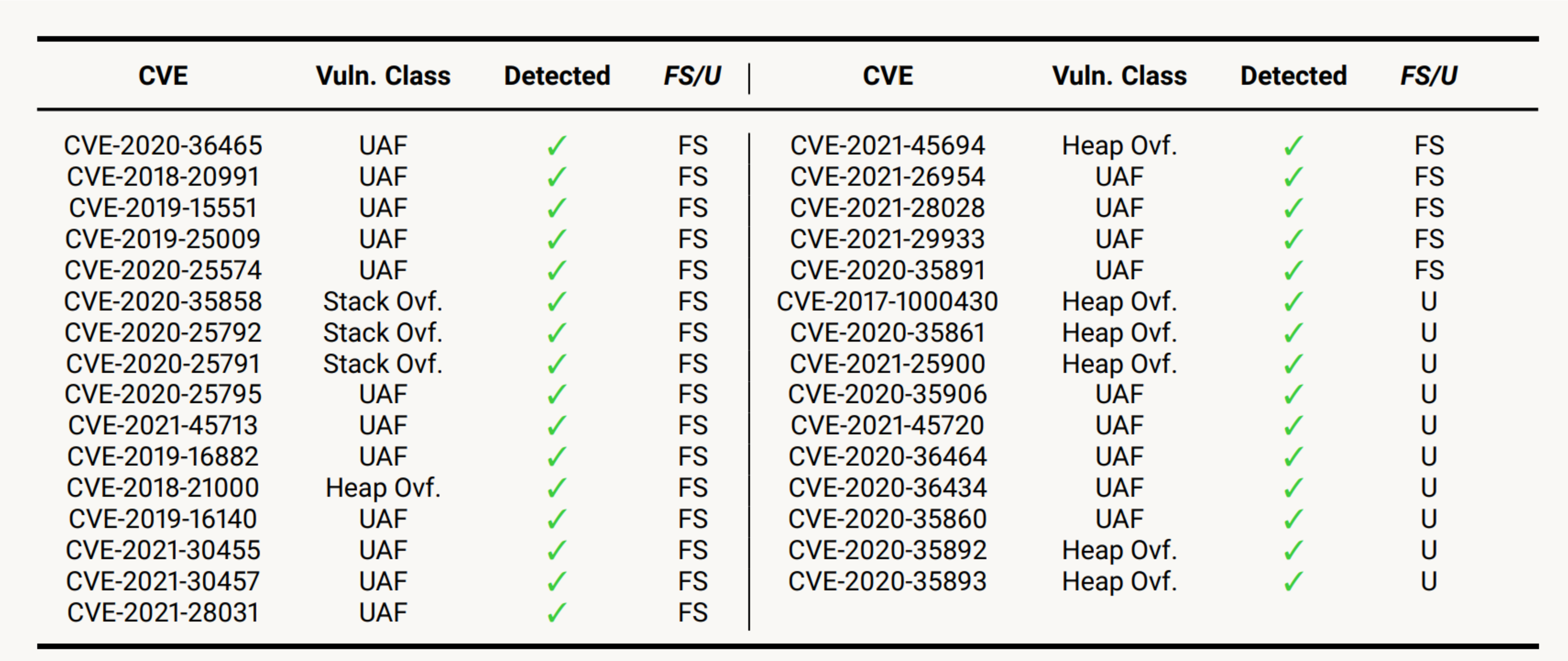
False-safe Site:
fs_check(ptr)
access(ptr)

Unsafe Site:
check(ptr)
access(ptr)

Selectively instrumented sites

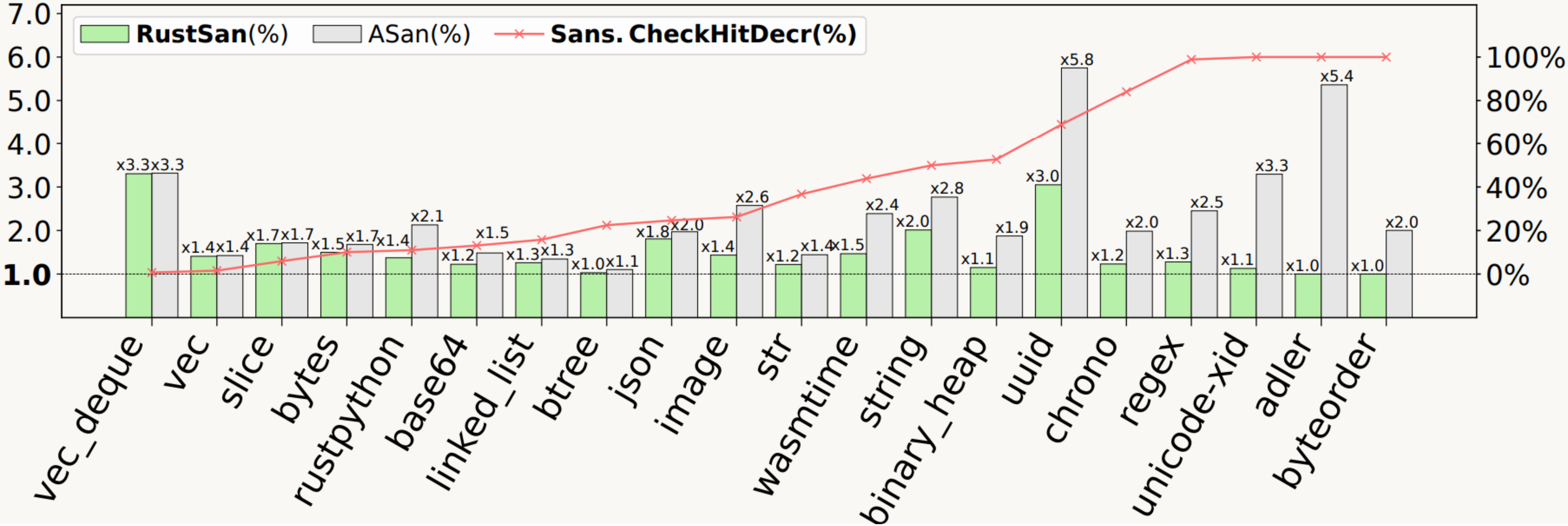
Evaluation

- FS: False Safe, U: Unsafe (Most Vulnerabilities are FS!)



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				

Performance (vs ASan)



Conclusion

- Detection Capability is equal

- Lower Overhead (reduces unnecessary checks)