ACM SIGSOFT Distinguished Paper Award (ASE'22)

Efficient Greybox Fuzzing to Detect Memory Errors

Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury

National University of Singapore

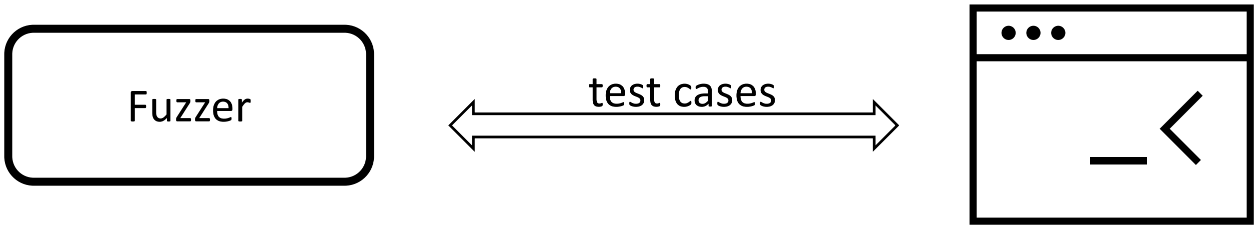
Memory Errors

- A memory error is any access not intended by the programmer:
  - Buffer overflow
  - Use-after-free
- Memory errors are a common source of security vulnerability.
  - Chrome: 70% of all security bugs are memory safety issues*.
- Fuzzing and memory sanitizer are popular techniques to detect memory errors.

*https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

Fuzzing

- Fuzzing (e.g. AFL) is an automatic test case generation method.
- A (biased) random search to generate test cases that lead to a crash (memory error).



However, **not all memory errors lead to crashes**!

Memory Sanitizer

- Sanitizers (e.g. Address Sanitizer) use instrumentation to detect memory errors, even if no crash would otherwise occur:

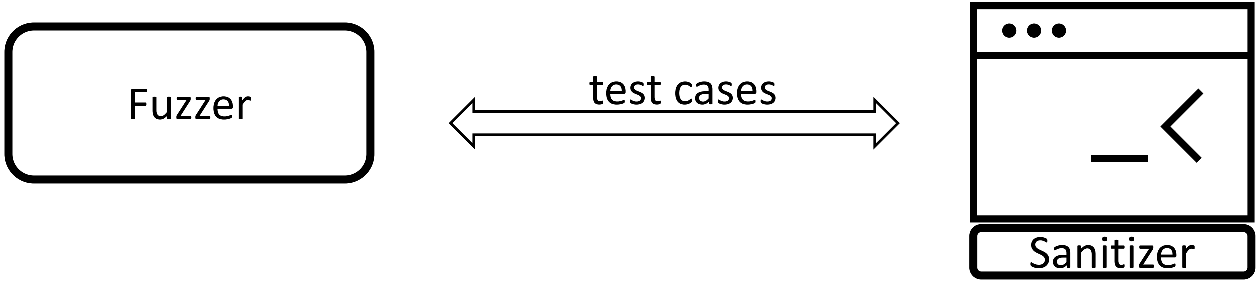
```
if (isOutOfBounds(p+i))
    abort()
p[i] = v
```

Sanitizer Instrumentation

- Sanitizers make memory errors visible, but require test cases.

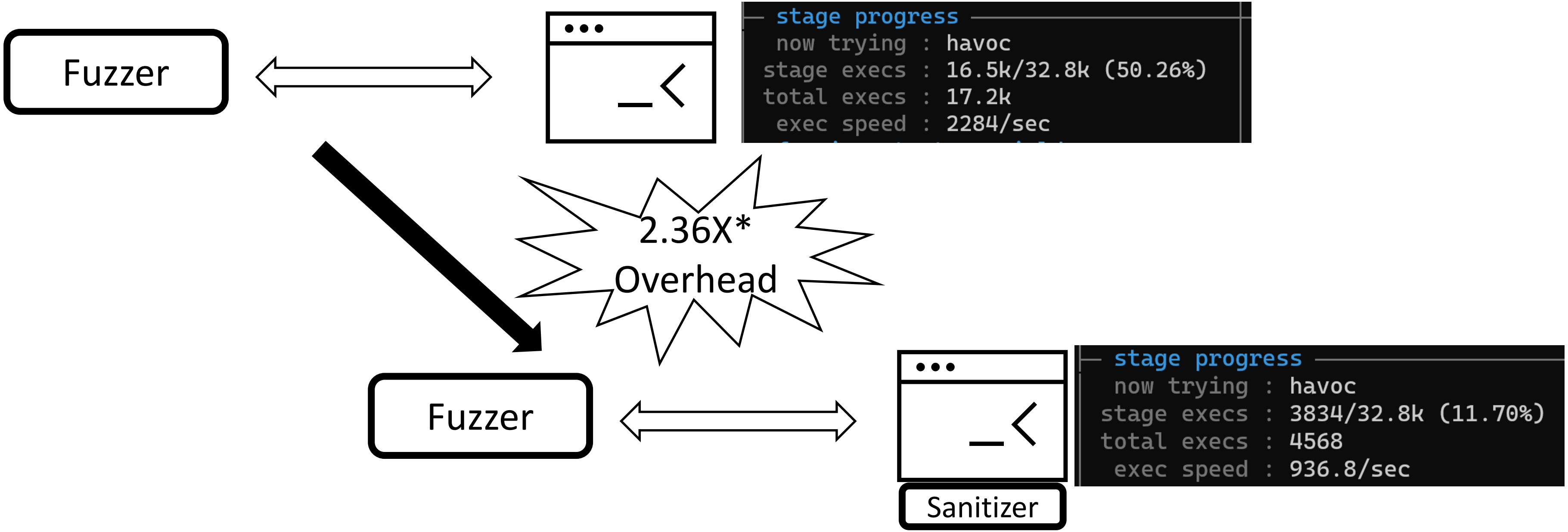
Fuzzing + Sanitizer

- It is natural to combine fuzzing with sanitizers:
- The fuzzer generates test cases, and the sanitizer identifies memory errors.



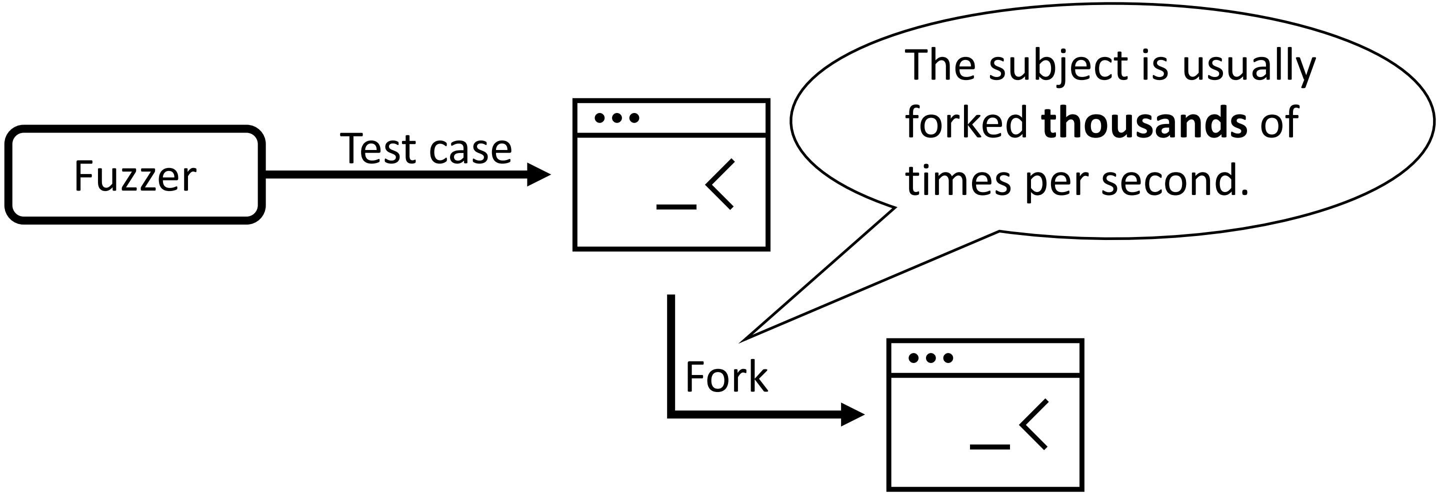
Problem

- Significant performance overhead.



* AFL + Address Sanitizer (ASan) vs AFL in testing libpng.

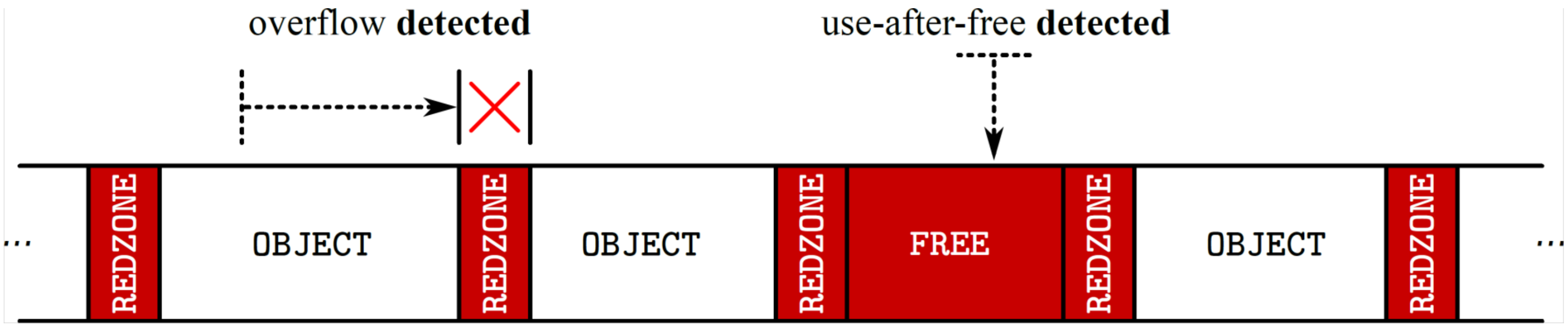
Why the performance overhead is huge?



- fork() is slow, especially when the program uses a lot of memory.
- More memory ⇒ more copying ⇒ slower

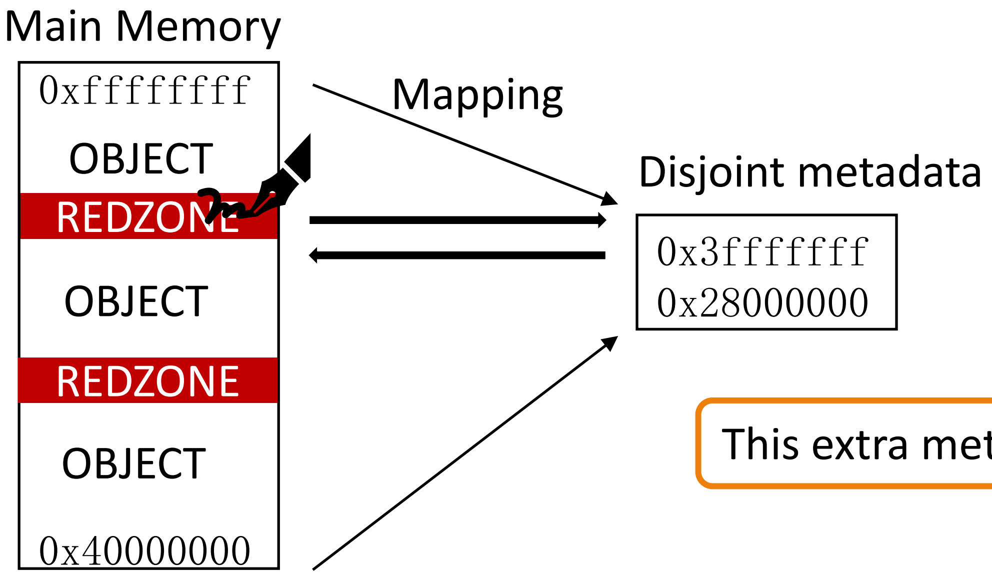
Sanitizers Use a Lot Of Memory

- Sanitizers (like ASan) work by memory poisoning:
- Redzone memory is poisoned ⇒ program cannot access
  - Detects buffer overflows
  - Detects use-after-free



Sanitizers Use a Lot Of Memory

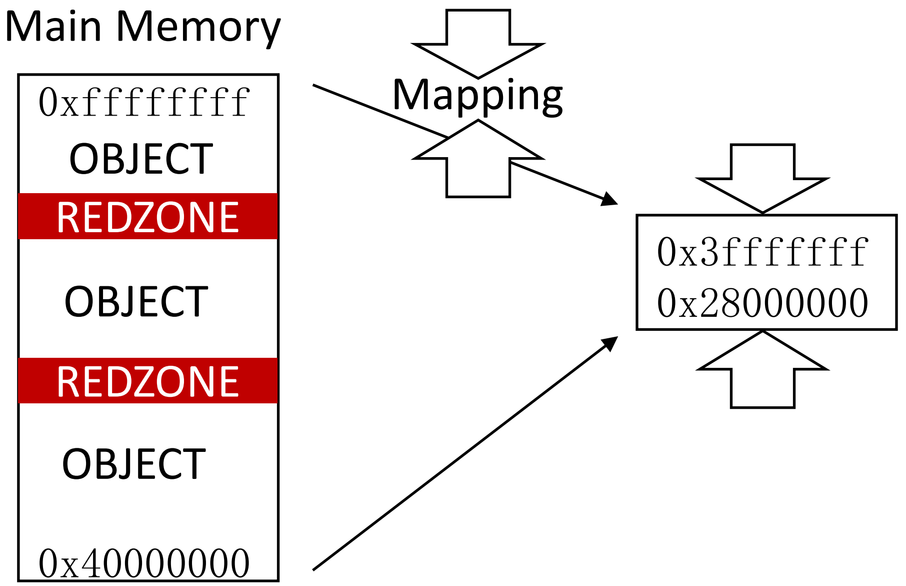
- ASan tracks poisoned memory using a disjoint metadata.



This extra metadata slows down fork() a lot!

Previous Works

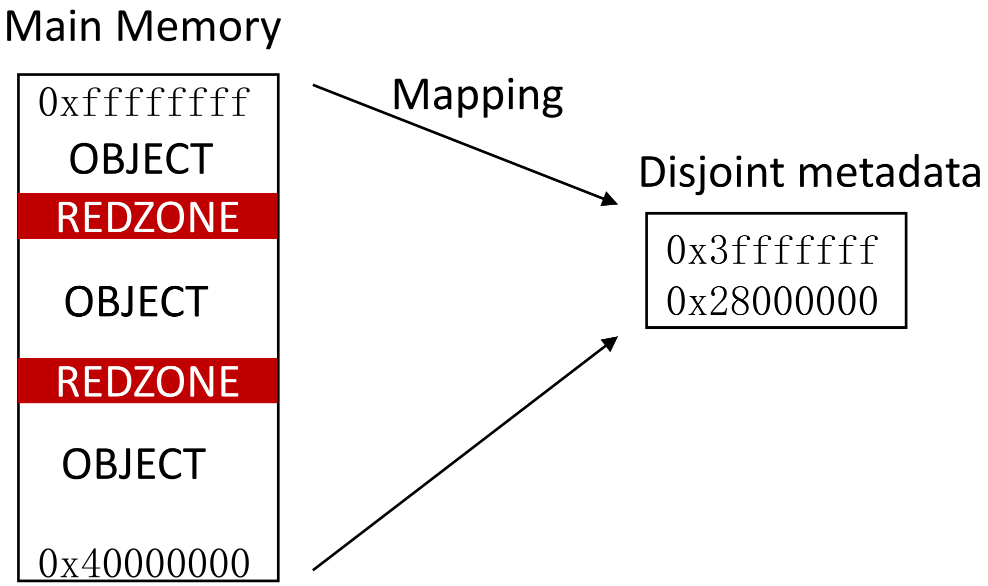
- SANRAZOR/ASan--: Remove redundant checking
- FuZZan: Compact the metadata



The performance overhead is improved, but still significant.

Our Idea

- Since disjoint metadata slows down fork() a lot, can we eliminate it?
- Yes! We represent poisoned memory by Randomized Embedded Tokens.

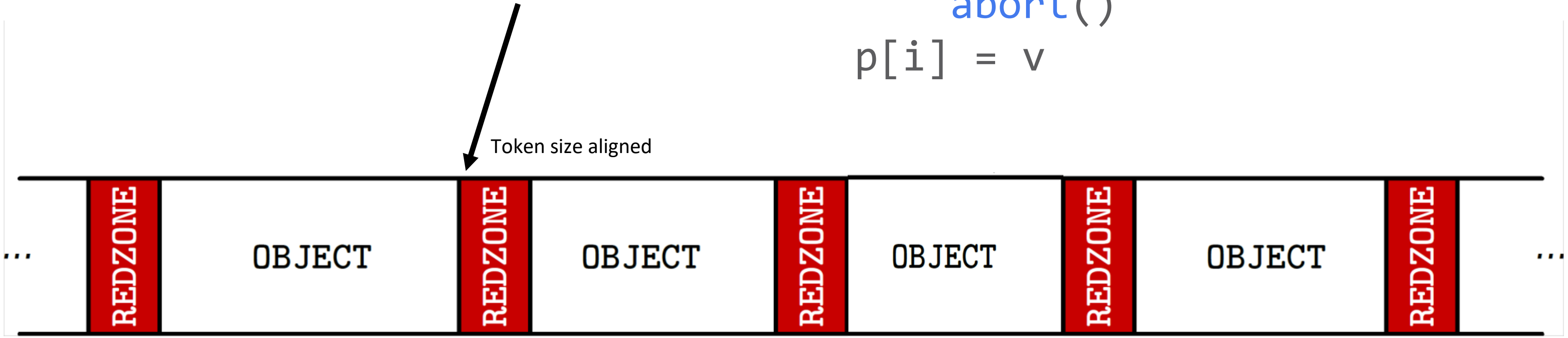


Our Design

- The presence of the token can determine if the memory is poisoned or not.

```
TOKEN = random()
```

```
if (p[i] == TOKEN)
    abort()
p[i] = v
```



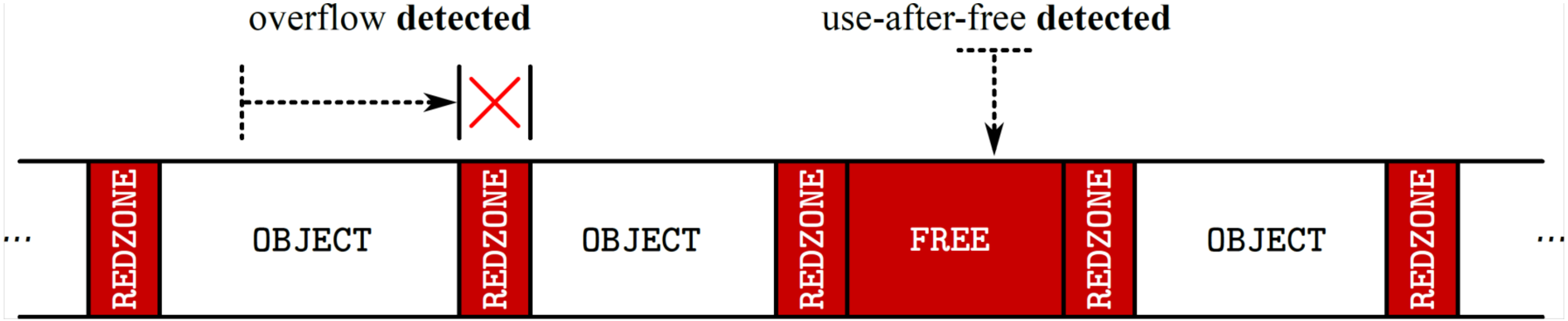
The disjoint metadata is eliminated.

Challenge 1: False Positive

- The content in objects could be the same as the random token.
- Our implementation uses a 64-bit token size.

More false positives  Lower bug detection granularity

Token Size

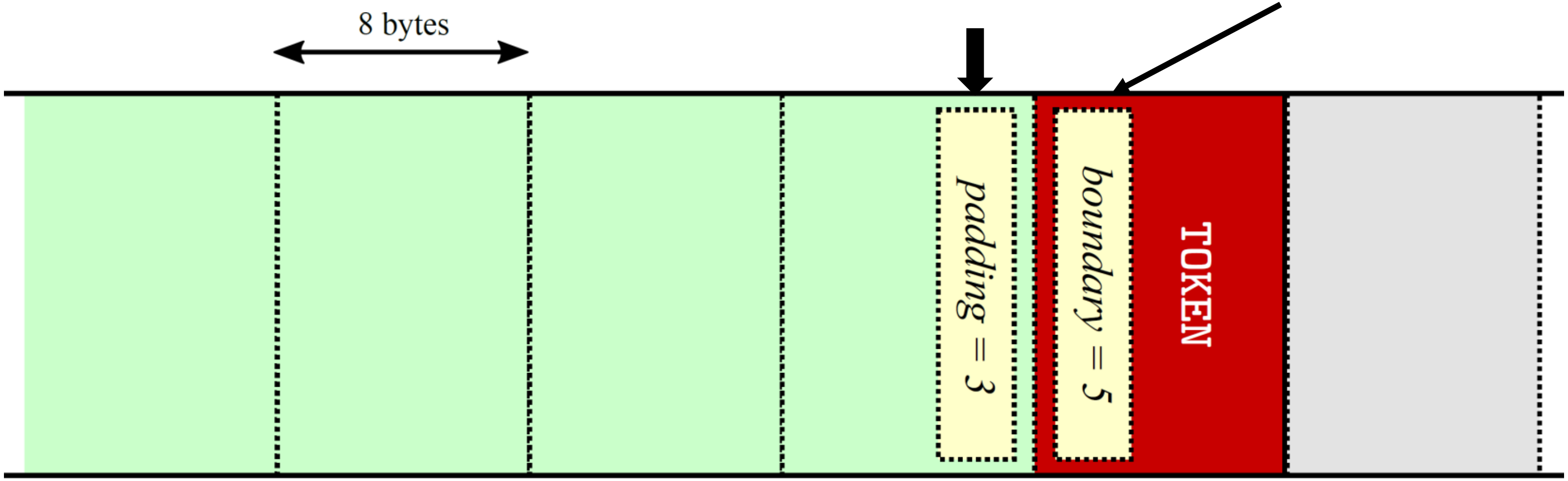


In theory, the first false positive occurs after ~584.9 years of CPU time.
In practice, we rerun program with a new random token to exclude false positives.

Challenge 2: Byte-accurate Boundary Checking

- We use the last three bits in the token to store the boundary of last object.

```
struct Token {uint64_t random :61; uint64_t boundary :3;};
```



Evaluation--Detection Capability

ReZZan : Byte-accurate
$ReZZan_{lite}$: Token-accurate

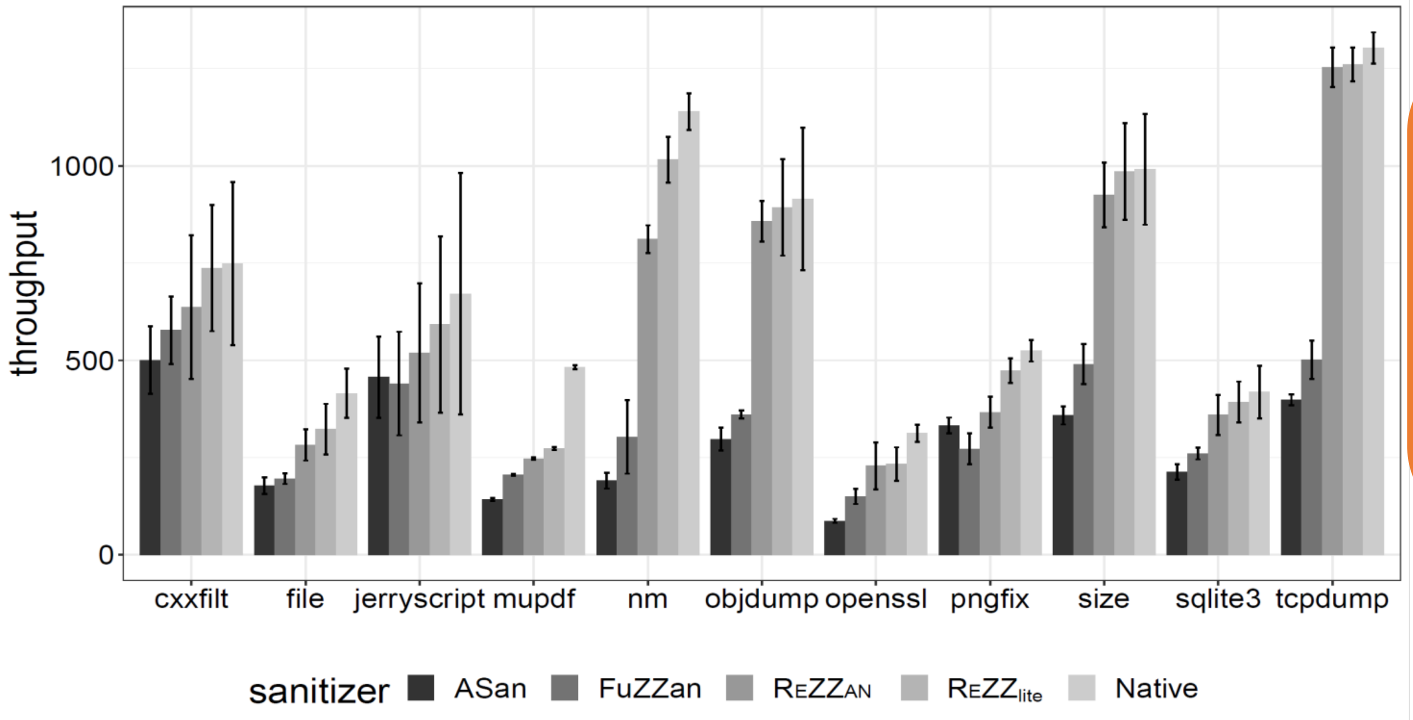
- The number of detected bugs (Juliet Benchmark).

CWE ID	Total	ASan	ReZZan	$ReZZan_{lite}$
Stack Buffer Overflow (121)	2,860	2,856	2,860	2,380
Heap Buffer Overflow (122)	3,246	3,189	3,246	2,724
Buffer Underwrite (124)	928	928	890	890
Buffer Overread (126)	630	610	630	630
Buffer Underread (127)	928	928	880	880
Use After Free (416)	392	392	392	392
Pass rate:		**99.10%**	**99.04%**	**87.89%**

ReZZan has the same level of bug detection capability as ASan.

Evaluation--Performance Overhead

- The average throughput (execs/sec)



Average performance loss:

***ReZZan$_{lite}$*: -12.45%**

***ReZZan:* -21.34%**

***FuZZan:* -50.11%**

***ASan:* -57.67%**

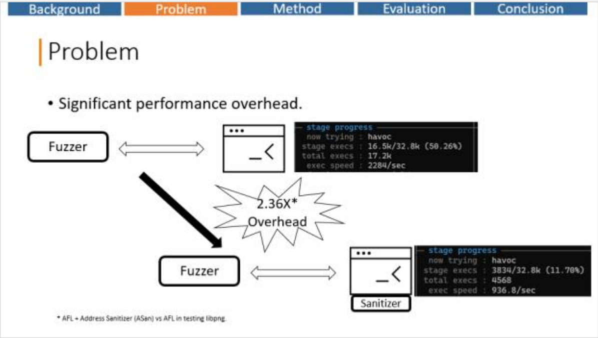
Evaluation--Bug Finding Effectiveness

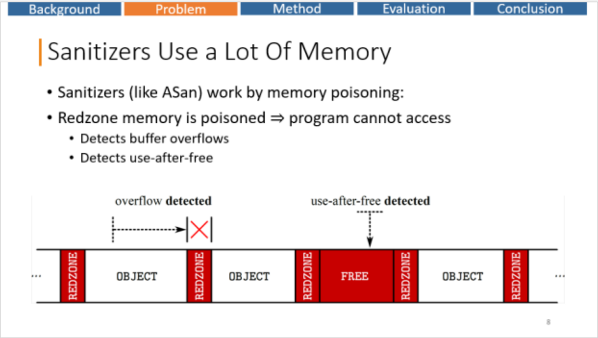
The time (second) to find the corresponding bug (Google fuzzer-test-suite).

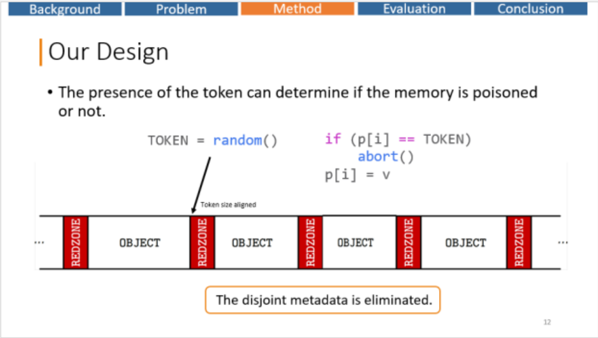
Subject	ASan	FuZZan	ReZZan	ReZZan_{lite}	Factor
c-ares	80.00	47.65	22.65	171.95	3.53
json	485.70	410.70	320.05	148.85	1.52
libxml2	29,328.75	21,462.88	6,301.00	6,318.63	4.65
openssl (A)	1,736.40	223.50	210.15	219.25	8.26
openssl (B)	26,589.50	21,431.00	12,750.00	-	2.09
pcre2	7,994.80	6,438.60	3,900.30	3,090.95	2.05
				Average:	**3.68X**

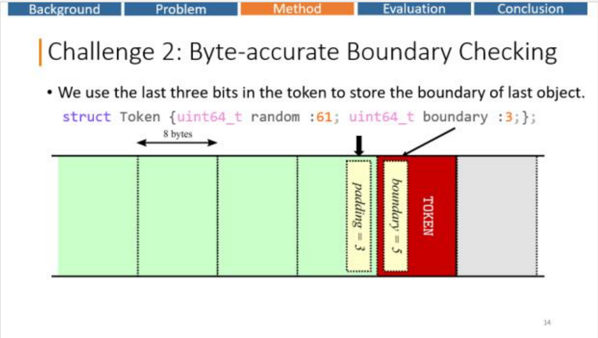
ReZZan exposes bugs 3.68 times faster than Asan.

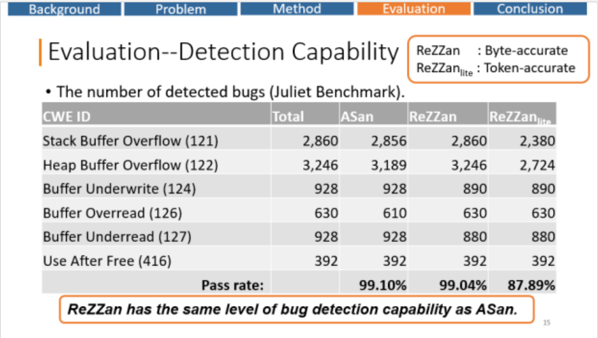
Conclusion





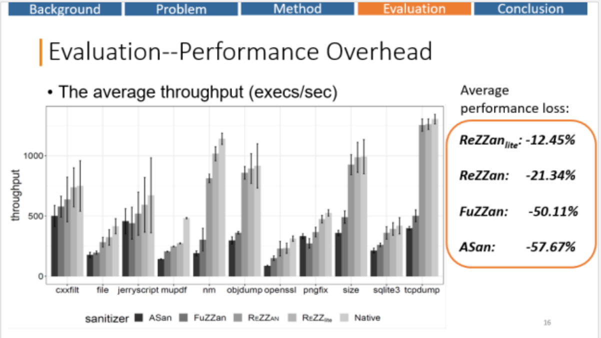






CWE ID	Total	ASan	ReZZan	$ReZZan_{lite}$
Stack Buffer Overflow (121)	2,860	2,856	2,860	2,380
Heap Buffer Overflow (122)	3,246	3,189	3,246	2,724
Buffer Underwrite (124)	928	928	890	890
Buffer Overread (126)	630	610	630	630
Buffer Underread (127)	928	928	880	880
Use After Free (416)	392	392	392	392
Pass rate:		99.10%	99.04%	87.89%

ReZZan has the same level of bug detection capability as ASan.



https://github.com/bajinsheng/ReZZan