Understanding the Performance-Effectiveness Trade-Offs of AddressSanitizer in Real-World Programs: An Empirical Study

Jiaxin Xi, Cheng Wen, Zhiwei Lin, Yuandao Cai, Jie Su,
Bin Yu, Zhuohua Li, Shengchao Qin

ASE 2026 Industry Showcase

AddressSanitizer (ASan)

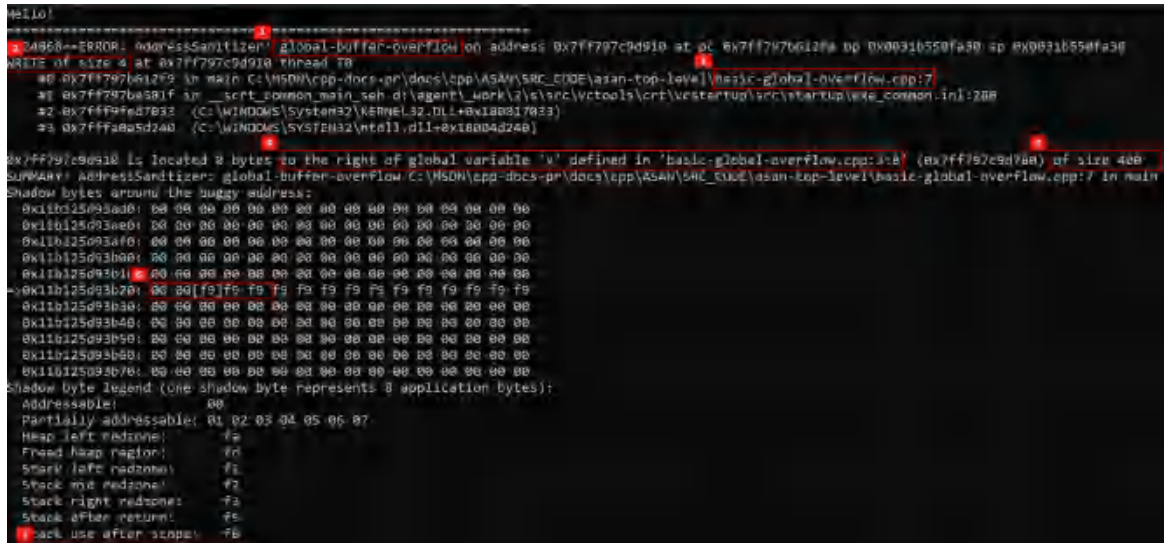
Memory errors can crash programs, corrupt data, and create security vulnerabilities.

A runtime detector for C/C++

Checks memory accesses
as the program runs.

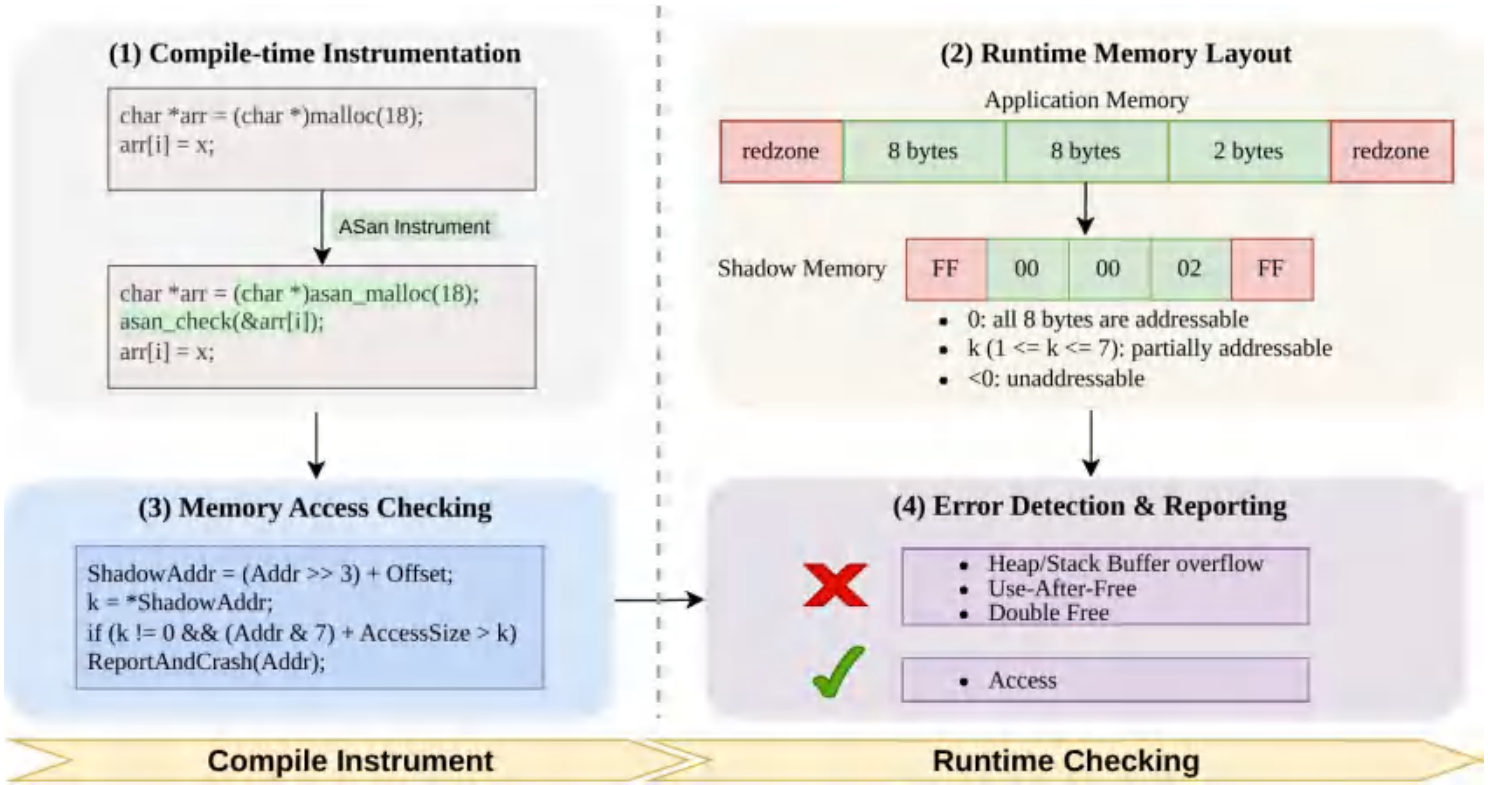
Out-of-bounds access
Use-after-free
Double-free

Example ASan error report



ASan helps developers locate memory errors during debugging, testing, and continuous integration.

How Does ASan Work?



Compiler instrumentation

Checks memory accesses.

Shadow memory

Tracks which addresses are accessible.

Redzones and quarantine

Guard object boundaries and delay reuse of freed heap memory.

Compiler checks and runtime metadata make invalid memory accesses observable.

ASan’s Detection Comes with Resource Costs

Runtime cost	Instrumented checks and runtime operations add execution work.
Memory cost	Shadow memory, redzones, and heap management increase memory use.
Impact on testing	Higher costs can limit continuous integration and production-like testing.

The challenge is to reduce resource costs while retaining useful error detection.

How Have Existing Tools Reduced These Costs?

Sanitizer	Main optimization
ASAP	Check reduction
LFP	Metadata and memory layout
HWASan	Hardware-assisted tagging
MEDs	Metadata and memory layout
FuZZan	Metadata and decoupling
SanRazor	Check reduction
ASan--	Check reduction
GiantSan	Metadata organization
SAND	Decoupled instrumentation
Tech-ASan	Metadata organization
RSan	Range metadata

Why another empirical study?

Optimization goals differ.
Tool availability differs.
Evaluation assumptions differ.

Reported improvements do not provide a consistent basis for practical comparison.

Our focus: a common basis for comparing default ASan, tuned ASan, and optimized sanitizers.

Four Research Questions

RQ1 How do ASan's internal components contribute to runtime overhead, memory consumption, and detection capability?

RQ2 How do ASan's tunable parameters, especially redzone and quarantine sizes, affect the trade-off between overhead reduction and bug-detection capability?

RQ3 How do compiler build modes, including optimization level and link-time optimization, influence ASan's performance and detection behavior?

RQ4 How do representative optimized sanitizers compare with ASan in efficiency, observed effectiveness, and phase-specific deployability?

Evaluation Setup

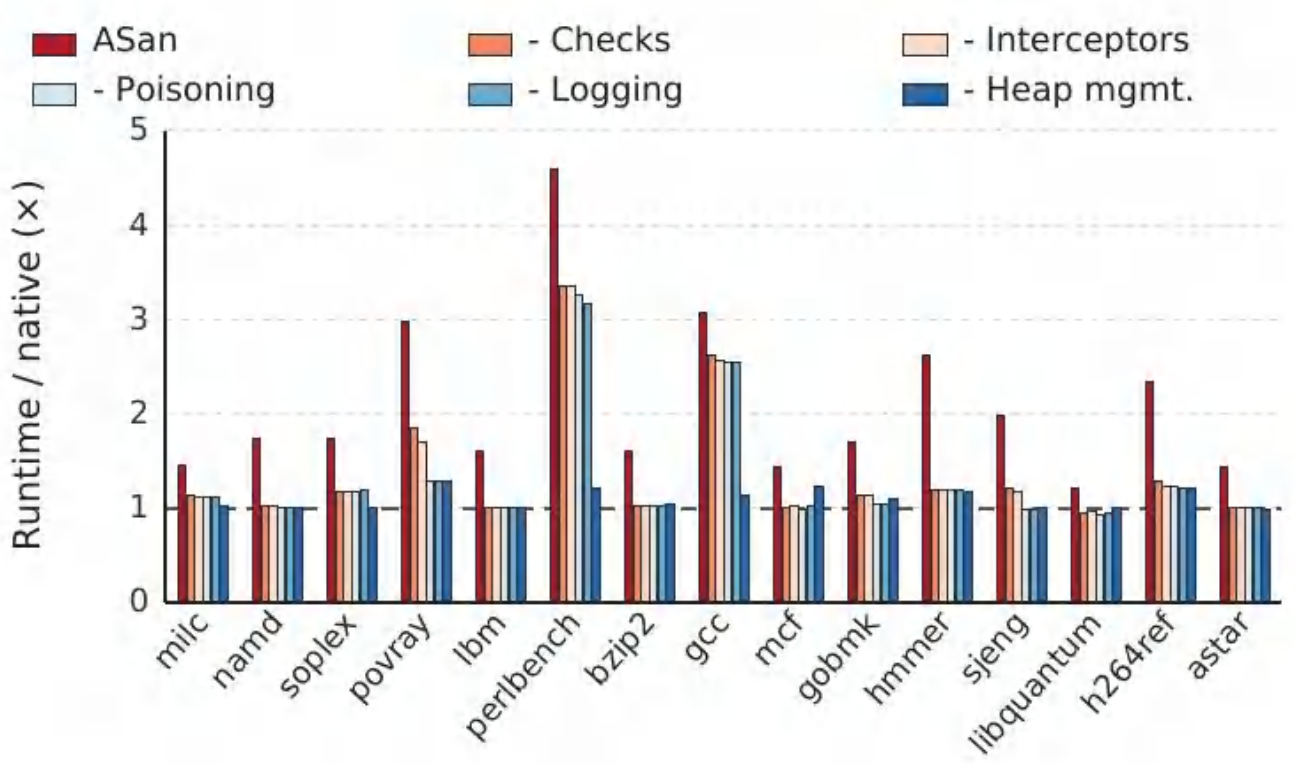
Default ASan, tuned ASan, and seven optimized sanitizers

Workload	Scale	What we measure
SPEC CPU2006	17 programs	Runtime and peak RSS
Juliet	9,301 buggy + 9,301 non-buggy	Reported and missed errors
MSET	232 error patterns	Detection by error category
Real-world CVEs	21 proof-of-concept inputs	Observed vulnerability reports
MySQL 5.7.19	OLTP read/write, 1–50 clients	Build, initialization, throughput, p95

Repeated performance runs on x86-64 Linux with an Intel i9-13900K and 128 GB RAM.

RQ1: Component-Wise Impact

Runtime ratios across 15 shared SPEC CPU2006 programs



Cumulative Ablation Order

1 Sanitizer checks

2 Interceptors

3 Poisoning

4 Logging / symbolization

5 Heap management

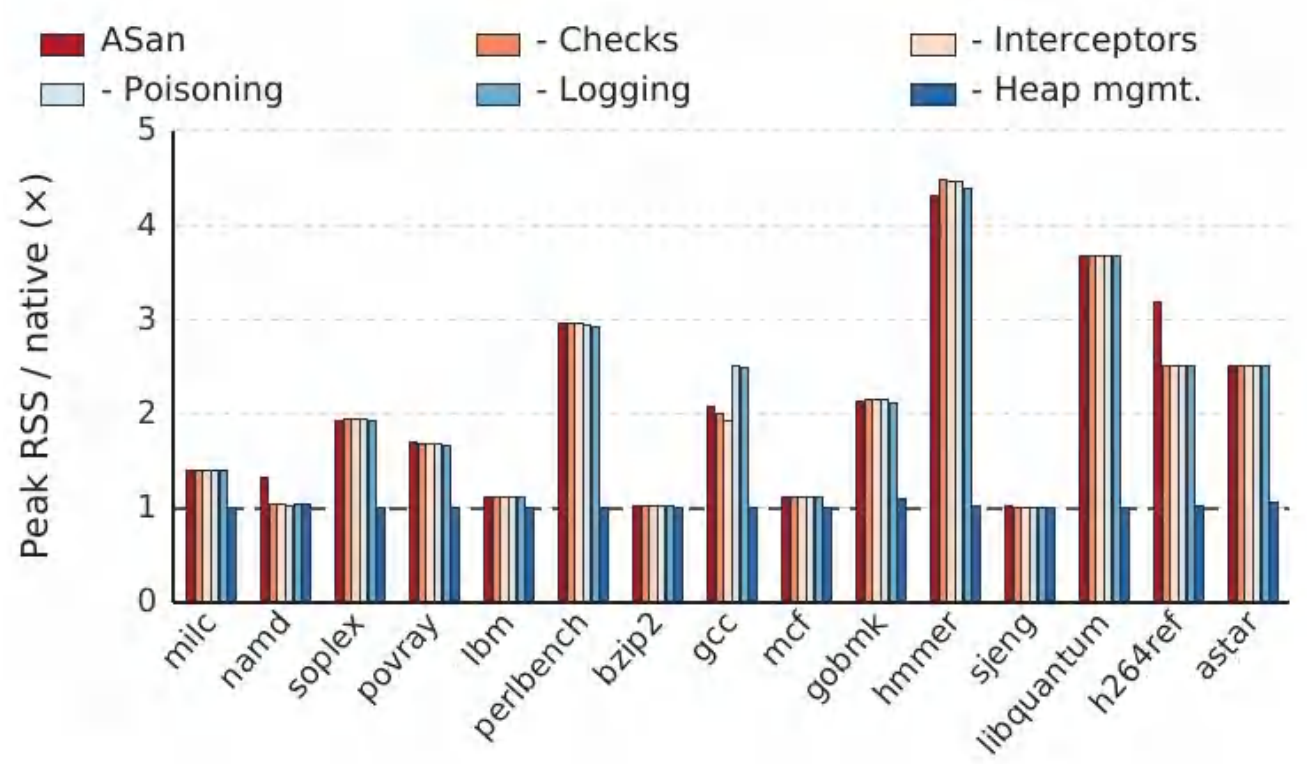
Each step keeps all preceding removals.

Removing checks lowers perlbench runtime by 26.8% (4.59× to 3.36× native).

These are order-dependent changes, not isolated component effects.

RQ1: Component-Wise Impact

Peak-RSS ratios under the same cumulative ablation order



Final heap-management step

2.06× to 1.02×

Mean peak-RSS ratio
15 shared programs

Program	Step 4	Step 5
hmmer	4.32×	1.02×
libquantum	3.67×	1.01×
h264ref	3.19×	1.03×

Final heap-management removal lowers mean peak RSS from 2.06× to 1.02× native.

Checks, interceptors, poisoning, and logging have already been removed.

RQ1: Component-Wise Impact

Juliet detection over 9,301 known-error cases

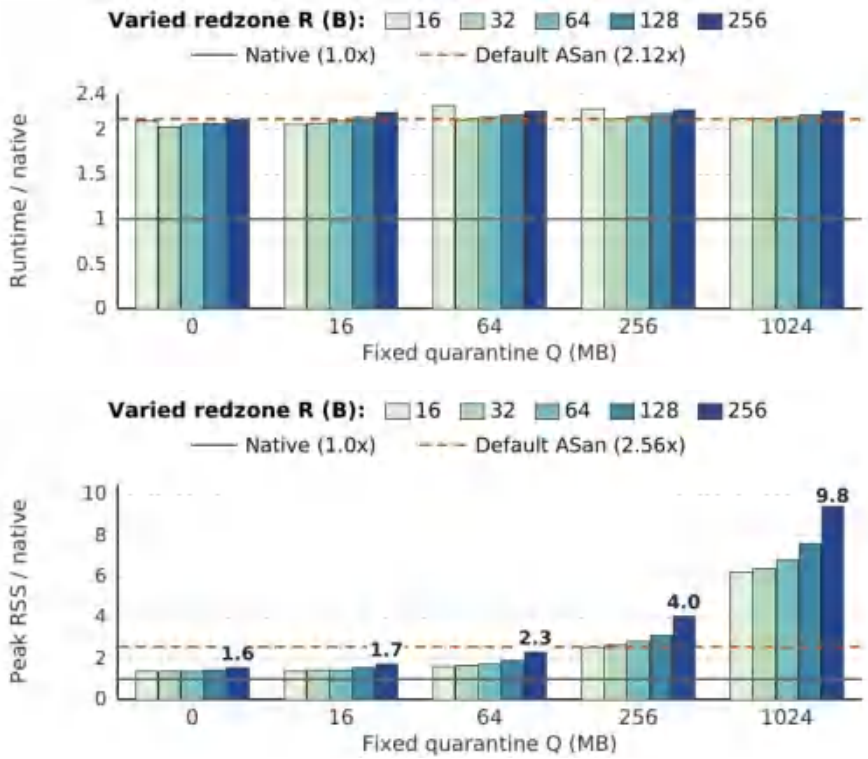
Stage	Additional removal	Found (TP)	Missed (FN)
Default	None	9,245	56
1	Checks	7,306	1,995
2	Interceptors	7,059	2,242
3	Poisoning	3,871	5,430
4	Logging / symbolization	3,871	5,430
5	Heap management	2,296	7,005

Reported errors fall from 9,245 to 2,296 as protection mechanisms are removed.

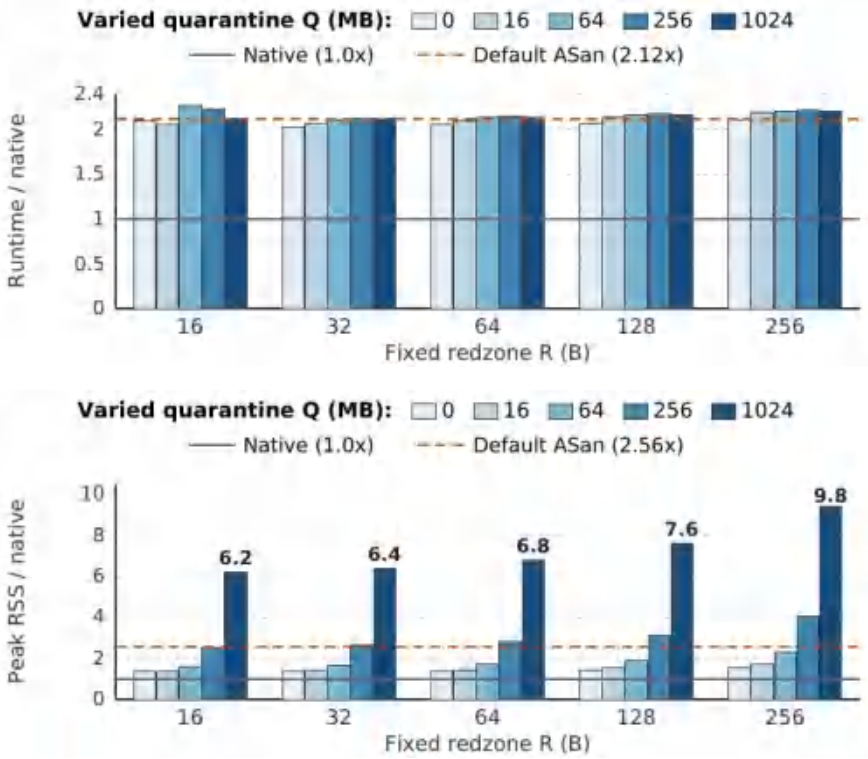
Every stage retains earlier removals. Lower cost comes with reduced observed detection.

RQ2: Parameter Sensitivity

Redzone size R (bytes)



Quarantine size Q (MB)

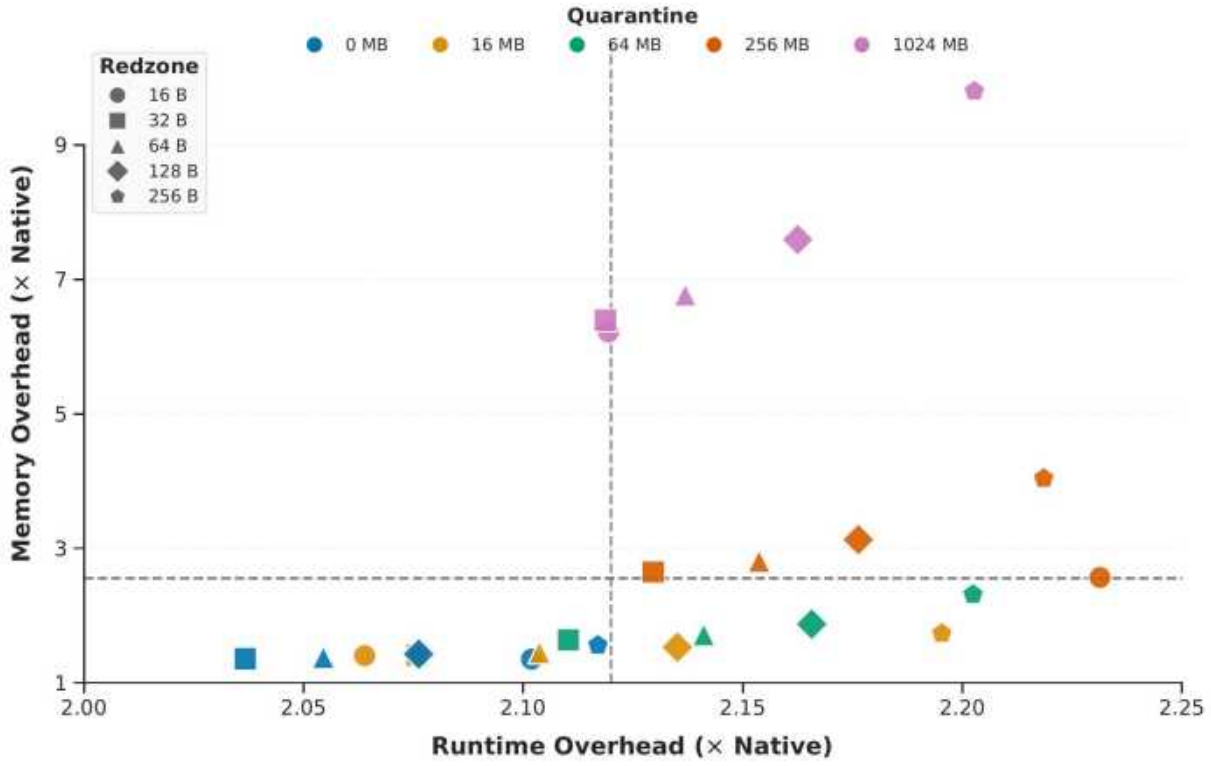


Larger redzones and quarantines mainly raise peak RSS across the tested grid.

Runtime varies over a much narrower range than peak memory. We evaluate 25 R/Q settings.

RQ2: Parameter Sensitivity

Measured runtime and memory costs of 25 R/Q settings



R16 / Q16

16 B redzone
16 MB quarantine

SPEC

Lower peak RSS

JULIET

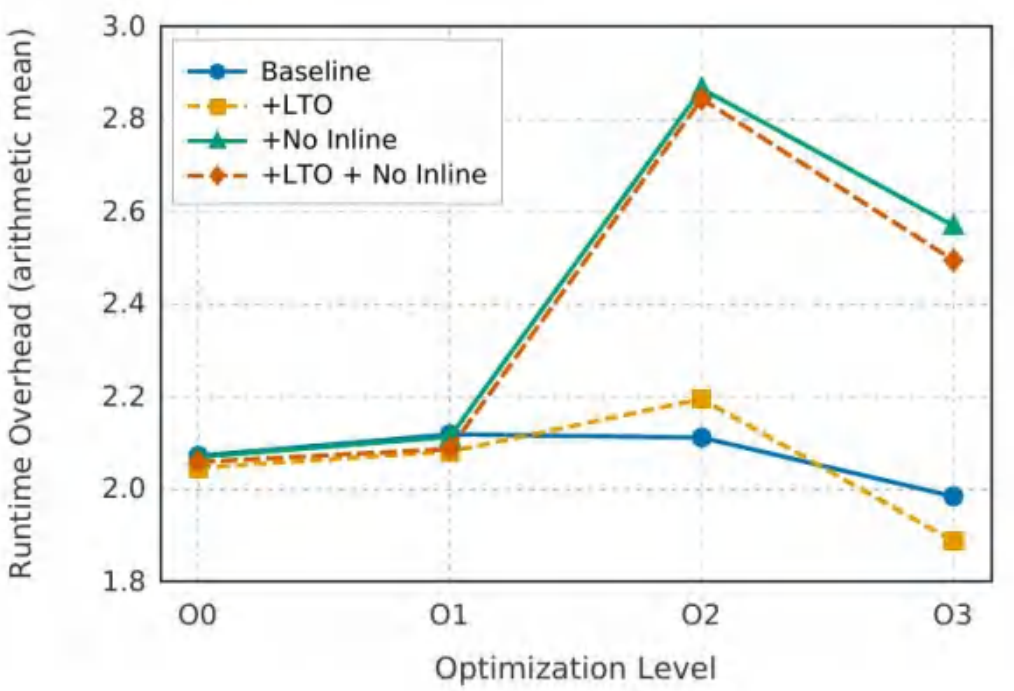
No additional missed errors

R16/Q16 is a lower-memory starting candidate with no additional Juliet misses.

Delayed use-after-free and project-specific detection still require validation.

RQ3: Compiler-Level Optimization

LTO and inlining options



Baseline in the legend means default ASan.
Native references match each optimization level.

Juliet build modes

Build	Found	Missed
O0	9,245	56
O1 + FP	7,999	1,302
O2 + FP + LTO	7,166	2,135
O3 + FP + LTO	7,166	2,135

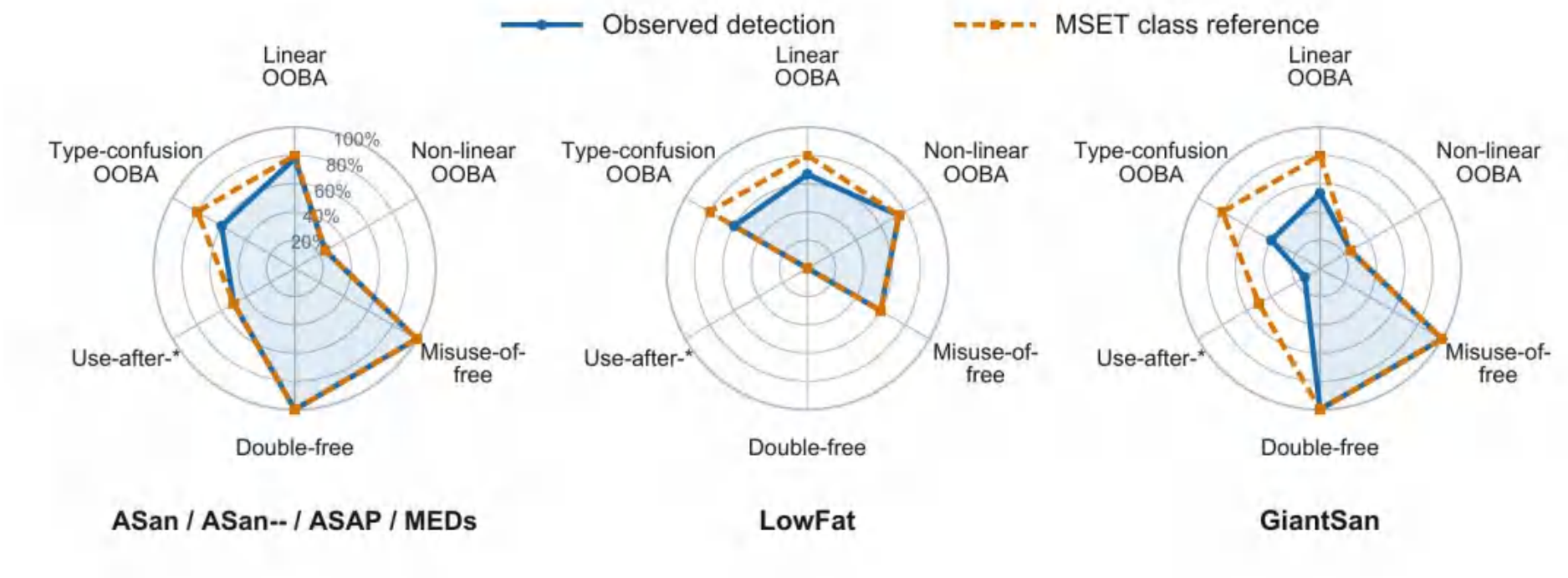
LTO: Link-Time Optimization
Optimizes across source files.
No Inline: restricts function inlining.
FP: frame pointer. Debug settings vary.

Restricting inlining raises O2–O3 runtime. Build choices also change Juliet detection.

The detection configurations change multiple options together and do not isolate LTO.

RQ4: Comparative Utility

Observed detection across 232 MSET error patterns



Solid: observed detection. Dashed: technique-class reference, not guaranteed coverage.

Detection varies by error pattern. No single overall score describes coverage.

RQ4: Comparative Utility

Juliet: 9,301 buggy cases and 9,301 non-buggy cases

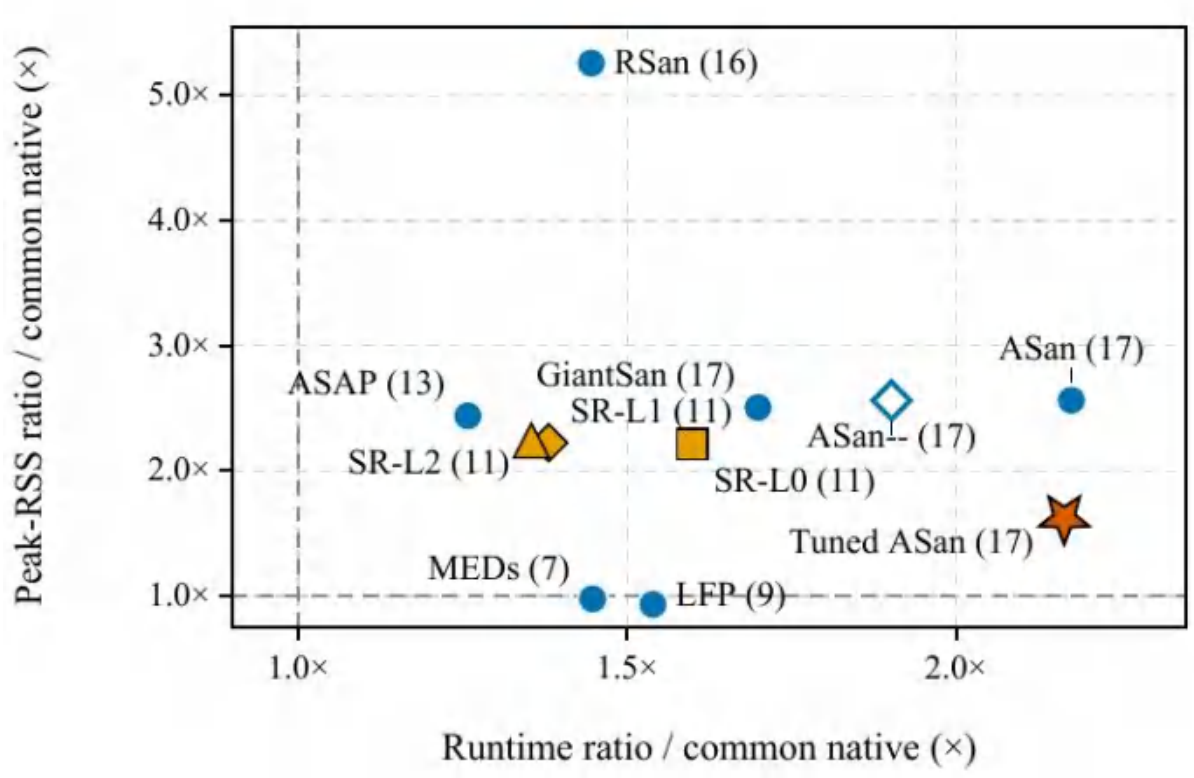
Tool / configuration	Found (TP)	Missed (FN)	False alarms (FP)
Default ASan	9,245	56	0
Tuned ASan (R16/Q16)	9,245	56	0
ASan--	9,245	56	0
LFP	4,710	4,591	0
MEDs	9,240	61	0
ASAP	7,696	1,605	43
SanRazor	9,137	164	4
RSan	9,301	0	0
GiantSan	8,332	969	0

RSan reports all buggy Juliet cases. Default ASan, tuned ASan and ASan-- each report 9,245.

TP and FN refer to buggy cases. FP refers to non-buggy cases. These results are workload-specific.

RQ4: Comparative Utility

SPEC runtime and peak-RSS ratios for completed programs



SPEC completions

Configuration	Count
Default ASan	17 / 17
Tuned ASan	17 / 17
ASan--	17 / 17
GiantSan	17 / 17
RSan	16 / 17
SanRazor L0 / L1 / L2	11 / 17 each
ASAP	13 / 17
LFP	9 / 17
MEDs	7 / 17

Unequal completed subsets and toolchains prevent a formal cross-tool ranking.

Tuned ASan is an essential baseline when judging claimed overhead reductions.

RQ4: Comparative Utility

Memory-error reports on 21 real-world CVE proof-of-concept inputs

Project	PoCs
libzip	1
autotrace	9
imageworsener	4
zziplib	2
potrace	1
mp3gain	3
binutils	1

Sanitizer	Reports
ASan / ASan-- / GiantSan	21 / 21
LFP	19 / 21
SanRazor	15 / 21
MEDs	14 / 21

ASan, ASan-- and GiantSan report all 21 selected PoCs.

Selected PoCs complement synthetic tests but do not establish general vulnerability coverage.

RQ4: Comparative Utility

MySQL 5.7.19: build, initialization, and workload execution

Tool / LLVM	Last stage	Observed outcome
ASan: 4 / 12 / 16	Workload	Built, initialized and benchmarked
ASan--: 4	Workload	Built, initialized and benchmarked
SanRazor-L2: 9	Workload	Built, initialized and benchmarked
GiantSan: 12	Compilation	Compiler exit 139 in yaSSL/TaoCrypt
MEDs: 4	Source generation	gen_lex_hash crashed before mysqld
ASAP: 4	Compilation	llc signal 11 on srv0start.cc
LFP: 4	Initialization	LowFat OOB escape. mysqld aborted
RSan: 16	Initialization	Full-LTO link >4 h. InnoDB initialization crash

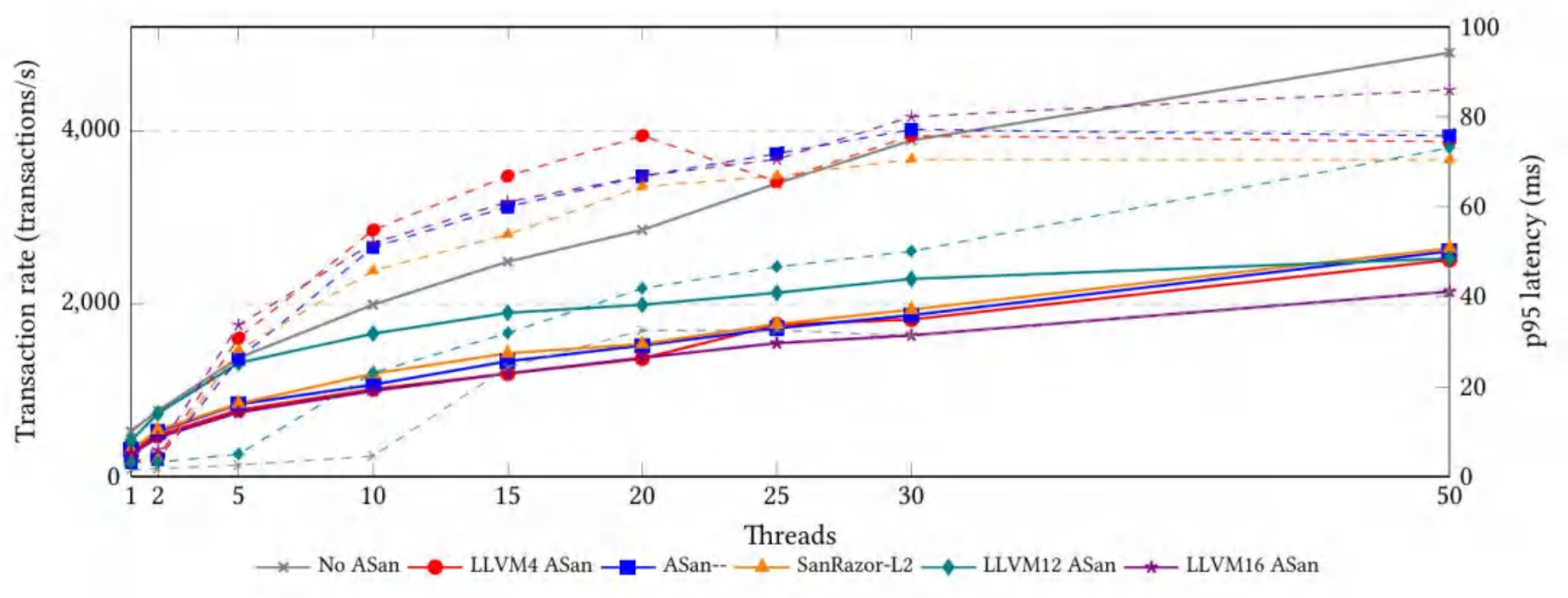
Five instrumented configurations complete the MySQL workload.

Failures before workload execution are deployment outcomes, not detection misses.

RQ4: Comparative Utility

Solid: throughput (left axis). Dashed: p95 latency (right axis).

Repeated runs. The LLVM configurations share one native reference.



At 50 clients, throughput falls by about half, similar in cost scale to ~2× SPEC runtime.

Native: 4,902 tx/s. Instrumented: 2,140–2,641 tx/s. p95: 41.1 vs 70.6–86.0 ms.

Conclusion

A deployment-oriented comparison of default ASan, tuned ASan, and optimized sanitizers.

RQ1 Removing protection lowers some costs but also reduces detection.
The component effects depend on the cumulative removal order.

RQ2 R16/Q16 lowers RSS without additional Juliet misses.
Project-specific and delayed use-after-free tests remain necessary.

RQ3 Compiler choices affect both overhead and reported errors.
The complete build configuration needs validation.

RQ4 Tuned ASan is a required comparison baseline.
Efficiency, observed detection, and deployability belong in the same evaluation.

A lower-cost tool must still detect the errors that matter and run on the target application.



Scope: tested x86-64 Linux workloads, configurations, and tool versions.

Reproduction package: doi.org/10.5281/zenodo.21366704